

# Introduction to High Performance Scientific Computing

---

# SOFTWARE • ENVIRONMENTS • TOOLS

---

The SIAM series on Software, Environments, and Tools focuses on the practical implementation of computational methods and the high performance aspects of scientific computation by emphasizing in-demand software, computing environments, and tools for computing. Software technology development issues such as current status, applications and algorithms, mathematical software, software tools, languages and compilers, computing environments, and visualization are presented.

---

## Editor-in-Chief

Jack J. Dongarra

University of Tennessee and Oak Ridge National Laboratory

## Series Editors

**Timothy A. Davis**

Texas A & M University

**Laura Grigori**

INRIA

**Padma Raghavan**

Pennsylvania State University

**James W. Demmel**

University of California,  
Berkeley

**Michael A. Heroux**

Sandia National Laboratories

**Yves Robert**

ENS Lyon

## Software, Environments, and Tools

D. L. Chopp, *Introduction to High Performance Scientific Computing*

Thomas Huckle and Tobias Neckel, *Bits and Bugs: A Scientific and Historical Review of Software Failures in Computational Science*

Thomas F. Coleman and Wei Xu, *Automatic Differentiation in MATLAB Using ADMAT with Applications*

Walter Gautschi, *Orthogonal Polynomials in MATLAB: Exercises and Solutions*

Daniel J. Bates, Jonathan D. Hauenstein, Andrew J. Sommese, and Charles W. Wampler, *Numerically Solving Polynomial Systems with Bertini*

Uwe Naumann, *The Art of Differentiating Computer Programs: An Introduction to Algorithmic Differentiation*

C. T. Kelley, *Implicit Filtering*

Jeremy Kepner and John Gilbert, editors, *Graph Algorithms in the Language of Linear Algebra*

Jeremy Kepner, *Parallel MATLAB for Multicore and Multinode Computers*

Michael A. Heroux, Padma Raghavan, and Horst D. Simon, editors, *Parallel Processing for Scientific Computing*

Gérard Meurant, *The Lanczos and Conjugate Gradient Algorithms: From Theory to Finite Precision Computations*

Bo Einarsson, editor, *Accuracy and Reliability in Scientific Computing*

Michael W. Berry and Murray Browne, *Understanding Search Engines: Mathematical Modeling and Text Retrieval, Second Edition*

Craig C. Douglas, Gundolf Haase, and Ulrich Langer, *A Tutorial on Elliptic PDE Solvers and Their Parallelization*

Louis Komzsik, *The Lanczos Method: Evolution and Application*

Bard Ermentrout, *Simulating, Analyzing, and Animating Dynamical Systems: A Guide to XPPAUT for Researchers and Students*

V. A. Barker, L. S. Blackford, J. Dongarra, J. Du Croz, S. Hammarling, M. Marinova, J. Wasniewski, and P. Yalamov, *LAPACK95 Users' Guide*

Stefan Goedecker and Adolfy Hoisie, *Performance Optimization of Numerically Intensive Codes*

Zhaojun Bai, James Demmel, Jack Dongarra, Axel Ruhe, and Henk van der Vorst, *Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide*

Lloyd N. Trefethen, *Spectral Methods in MATLAB*

E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' Guide, Third Edition*

Michael W. Berry and Murray Browne, *Understanding Search Engines: Mathematical Modeling and Text Retrieval*

Jack J. Dongarra, Iain S. Duff, Danny C. Sorensen, and Henk A. van der Vorst, *Numerical Linear Algebra for High-Performance Computers*

R. B. Lehoucq, D. C. Sorensen, and C. Yang, *ARPACK Users' Guide: Solution of Large-Scale Eigenvalue Problems with Implicitly Restarted Arnoldi Methods*

Randolph E. Bank, *PLTMG: A Software Package for Solving Elliptic Partial Differential Equations, Users' Guide 8.0*

L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley, *ScalAPACK Users' Guide*

Greg Astfalk, editor, *Applications on Advanced Architecture Computers*

Roger W. Hockney, *The Science of Computer Benchmarking*

Françoise Chaitin-Chatelin and Valérie Frayssé, *Lectures on Finite Precision Computations*

# Introduction to High Performance Scientific Computing

**D. L. Chopp**  
Northwestern University  
Evanston, Illinois



Society for Industrial and Applied Mathematics  
Philadelphia

Copyright © 2019 by the Society for Industrial and Applied Mathematics

10 9 8 7 6 5 4 3 2 1

All rights reserved. Printed in the United States of America. No part of this book may be reproduced, stored, or transmitted in any manner without the written permission of the publisher. For information, write to the Society for Industrial and Applied Mathematics, 3600 Market Street, 6th Floor, Philadelphia, PA 19104-2688 USA.

Trademarked names may be used in this book without the inclusion of a trademark symbol. These names are used in an editorial context only; no infringement of trademark is intended.

AMD Opteron is a trademark of Advanced Micro Devices, Inc.

Intel is a registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

LAPACK is a software package provided by Univ. of Tennessee; Univ. of California, Berkeley; Univ. of Colorado Denver; and NAG Ltd.

MATLAB is a registered trademark of The MathWorks, Inc. For MATLAB product information, please contact The MathWorks, Inc., 3 Apple Hill Drive, Natick, MA 01760-2098 USA, 508-647-7000, Fax: 508-647-7001, [info@mathworks.com](mailto:info@mathworks.com), [www.mathworks.com](http://www.mathworks.com).

METAL and OpenCL are registered trademarks of Apple, Inc.

MPI is a registered trademark of Autodesk, Inc.

OpenACC and NVidia are registered trademarks of NVIDIA Corporation in the U.S. and other countries.

OpenMP and the OpenMP logo are registered trademarks of the OpenMP Architecture Review Board in the United States and other countries. All rights reserved.

*Publications Director* Kivmars H. Bowling

*Acquisitions Editor* Elizabeth Greenspan

*Developmental Editor* Gina Rinelli Harris

*Managing Editor* Kelly Thomas

*Production Editor* Louis R. Primus

*Copy Editor* Susan Fleshman

*Production Manager* Donna Witzleben

*Production Coordinator* Cally A. Shrader

*Compositor* Cheryl Hufnagle

*Graphic Designer* Doug Smock

#### **Library of Congress Cataloging-in-Publication Data**

Names: Chopp, D. L. (David L.), author.

Title: Introduction to high performance scientific computing / D.L. Chopp,  
Northwestern University, Evanston, Illinois.

Description: Philadelphia : Society for Industrial and Applied Mathematics,  
[2019] | Series: Software, environments, and tools ; 30 | Includes  
bibliographical references and index.

Identifiers: LCCN 2018056823 (print) | LCCN 2018058010 (ebook) | ISBN  
9781611975642 | ISBN 9781611975635 (print)

Subjects: LCSH: Computer programming. | Engineering—Data processing. |  
Science—Data processing.

Classification: LCC QA76.6 (ebook) | LCC QA76.6 .4554 2019 (print) | DDC  
005.1-dc23

LC record available at <https://lccn.loc.gov/2018056823>

# Contents

<b>Preface</b>	<b>xi</b>
<b>1 Tools of the Trade</b>	<b>1</b>
1.1 Integrated Development Environments . . . . .	1
1.2 Text Editors . . . . .	1
1.3 Compilers . . . . .	2
1.4 Makefiles . . . . .	4
1.5 Debugging . . . . .	6
1.6 Profilers . . . . .	7
1.7 Version Control . . . . .	9
Exercise . . . . .	10
<b>I Elementary C Programming</b>	<b>11</b>
<b>2 Structure of a C Program</b>	<b>15</b>
2.1 Comments . . . . .	17
2.2 The Preamble . . . . .	17
2.3 The Main Program . . . . .	18
Exercise . . . . .	19
<b>3 Data Types and Structures</b>	<b>21</b>
3.1 Basic Data Types . . . . .	21
3.2 Mathematical Operations and Assigning Values . . . . .	24
3.3 Boolean Algebra . . . . .	27
3.4 Mathematical Functions . . . . .	29
3.5 Arrays and Pointers . . . . .	30
3.6 Structures . . . . .	36
Exercises . . . . .	38
<b>4 Input and Output</b>	<b>41</b>
4.1 Terminal I/O . . . . .	41
4.2 File I/O . . . . .	45
Exercises . . . . .	49
<b>5 Flow Control</b>	<b>51</b>
5.1 <code>for</code> Loops . . . . .	51
5.2 <code>while</code> and <code>do-while</code> Loops . . . . .	54
5.3 <code>if-then-else</code> . . . . .	54

5.4	switch-case Statements . . . . .	57
	Exercises . . . . .	57
<b>6</b>	<b>Functions</b>	<b>59</b>
6.1	Declarations and Definitions . . . . .	59
6.2	Function Arguments . . . . .	62
6.3	Measuring Performance . . . . .	65
	Exercises . . . . .	70
<b>7</b>	<b>Using Libraries</b>	<b>73</b>
7.1	BLAS and LAPACK . . . . .	74
7.2	FFTW . . . . .	82
7.3	Random Number Generation . . . . .	87
	Exercises . . . . .	89
<b>8</b>	<b>Projects for Serial Programming</b>	<b>93</b>
8.1	Random Processes . . . . .	93
8.2	Finite Difference Methods . . . . .	93
8.3	Elliptic Equations and Successive Overrelaxation . . . . .	94
8.4	Pseudospectral Methods . . . . .	95
<b>II</b>	<b>Parallel Computing Using OpenMP</b>	<b>97</b>
<b>9</b>	<b>Intro to OpenMP</b>	<b>101</b>
9.1	Creating Multiple Threads: #pragma omp parallel . . . . .	101
9.2	The num_threads Clause . . . . .	103
9.3	The shared Clause . . . . .	104
9.4	The private Clause . . . . .	105
9.5	The reduction Clause . . . . .	106
	Exercise . . . . .	107
<b>10</b>	<b>Subdividing for Loops</b>	<b>109</b>
10.1	Subdividing Loops: #pragma omp for . . . . .	109
10.2	The private Clause . . . . .	111
10.3	The reduction Clause . . . . .	113
10.4	The ordered Clause . . . . .	113
10.5	The schedule Clause . . . . .	114
10.6	The nowait Clause . . . . .	119
10.7	Efficiency Measures . . . . .	120
	Exercises . . . . .	124
<b>11</b>	<b>Serial Tasks Inside Parallel Regions</b>	<b>125</b>
11.1	Serial Subregions: #pragma omp single . . . . .	125
11.2	The copyprivate Clause . . . . .	128
	Exercise . . . . .	130
<b>12</b>	<b>Distinct Tasks in Parallel</b>	<b>131</b>
12.1	Multiple Parallel Distinct Tasks: #pragma omp sections . . . . .	131
12.2	The private Clause . . . . .	133
12.3	The reduction Clause . . . . .	133

12.4	The nowait Clause . . . . .	134
Exercise . . . . .		134
<b>13</b>	<b>Critical and Atomic Code</b>	<b>135</b>
13.1	Atomic Statements . . . . .	135
13.2	Critical Statements . . . . .	137
Exercise . . . . .		138
<b>14</b>	<b>OpenMP Libraries</b>	<b>139</b>
14.1	FFTW . . . . .	139
14.2	Random Numbers . . . . .	141
Exercises . . . . .		145
<b>15</b>	<b>Projects for OpenMP Programming</b>	<b>147</b>
15.1	Random Processes . . . . .	147
15.2	Finite Difference Methods . . . . .	148
15.3	Elliptic Equations and Successive Overrelaxation . . . . .	149
15.4	Pseudospectral Methods . . . . .	150
<b>III</b>	<b>Distributed Programming and MPI</b>	<b>153</b>
<b>16</b>	<b>Preliminaries</b>	<b>157</b>
16.1	Parallel “Hello World!” . . . . .	157
16.2	Compiling and Running MPI Code . . . . .	158
16.3	Submitting Jobs . . . . .	159
Exercise . . . . .		162
<b>17</b>	<b>Passing Messages</b>	<b>163</b>
17.1	Blocking Send and Receive . . . . .	163
17.2	Nonblocking Send and Receive . . . . .	166
17.3	Combined Send and Receive . . . . .	174
17.4	Gather/Scatter Communications . . . . .	176
17.5	Broadcast and Reduction . . . . .	178
17.6	Error Handling . . . . .	180
Exercises . . . . .		181
<b>18</b>	<b>Groups and Communicators</b>	<b>183</b>
18.1	Subgroups and Communicators . . . . .	183
18.2	Communicators Using Split . . . . .	186
18.3	Grid Communicators . . . . .	189
Exercises . . . . .		192
<b>19</b>	<b>Measuring Efficiency and Checkpointing</b>	<b>193</b>
19.1	Efficiency Measures . . . . .	193
19.2	Checkpointing . . . . .	195
<b>20</b>	<b>MPI Libraries</b>	<b>197</b>
20.1	ScaLAPACK . . . . .	197
20.2	FFTW . . . . .	210
Exercises . . . . .		215

<b>21 Projects for Distributed Programming</b>	<b>217</b>
21.1 Random Processes . . . . .	217
21.2 Finite Difference Methods . . . . .	218
21.3 Elliptic Equations and SOR . . . . .	224
21.4 Pseudospectral Methods . . . . .	226
<b>IV GPU Programming and CUDA</b>	<b>227</b>
<b>22 Intro to CUDA</b>	<b>231</b>
22.1 First CUDA Program . . . . .	231
22.2 Selecting the Correct GPU . . . . .	233
22.3 CUDA Development Tools . . . . .	234
Exercises . . . . .	238
<b>23 Parallel CUDA Using Blocks</b>	<b>239</b>
23.1 Running Kernels in Parallel . . . . .	239
23.2 Organization of Blocks . . . . .	242
23.3 Threads . . . . .	245
23.4 Error Handling . . . . .	245
Exercises . . . . .	247
<b>24 GPU Memory</b>	<b>249</b>
24.1 Shared Memory . . . . .	249
24.2 Constant Memory . . . . .	254
24.3 Texture Memory . . . . .	256
24.4 Warp Shuffles . . . . .	261
Exercises . . . . .	267
<b>25 Streams</b>	<b>269</b>
25.1 Stream Creation and Destruction . . . . .	269
25.2 Asynchronous Memory Copies . . . . .	270
25.3 Single Stream Example . . . . .	271
25.4 Multiple Streams . . . . .	274
25.5 General Strategies . . . . .	281
25.6 Measuring Performance . . . . .	282
Exercises . . . . .	285
<b>26 CUDA Libraries</b>	<b>287</b>
26.1 MAGMA . . . . .	287
26.2 cuRAND . . . . .	292
26.3 cuFFT . . . . .	297
26.4 cuSPARSE . . . . .	302
Exercises . . . . .	311
<b>27 Projects for CUDA Programming</b>	<b>313</b>
27.1 Random Processes . . . . .	313
27.2 Finite Difference Methods . . . . .	314
27.3 Elliptic Equations and SOR . . . . .	317
27.4 Pseudospectral Methods . . . . .	318

<b>V GPU Programming and OpenCL</b>	<b>321</b>
<b>28 Intro to OpenCL</b>	<b>325</b>
28.1 First OpenCL Program . . . . .	325
28.2 Setting Up the Context . . . . .	329
Exercise . . . . .	335
<b>29 Parallel OpenCL Using Work-Groups</b>	<b>337</b>
29.1 Parallel OpenCL Example . . . . .	337
29.2 Compiling Kernel Functions . . . . .	343
29.3 Error Handling . . . . .	348
29.4 Organization of Work-Groups . . . . .	351
29.5 Work-Items . . . . .	353
Exercises . . . . .	355
<b>30 GPU Memory</b>	<b>357</b>
30.1 Local Memory . . . . .	357
30.2 Constant Memory . . . . .	363
Exercises . . . . .	367
<b>31 Command Queues</b>	<b>369</b>
31.1 Events . . . . .	369
31.2 Measuring Performance . . . . .	373
31.3 Concurrency . . . . .	375
31.4 Using Multiple Devices . . . . .	380
Exercises . . . . .	381
<b>32 OpenCL Libraries</b>	<b>383</b>
32.1 Random123 . . . . .	383
32.2 clMAGMA . . . . .	388
32.3 clFFT . . . . .	393
Exercises . . . . .	399
<b>33 Projects for OpenCL Programming</b>	<b>401</b>
33.1 Random Processes . . . . .	401
33.2 Finite Difference Methods . . . . .	402
33.3 Elliptic Equations and SOR . . . . .	405
33.4 Pseudospectral Methods . . . . .	406
<b>VI Applications</b>	<b>407</b>
<b>34 Stochastic Differential Equations</b>	<b>411</b>
34.1 Mathematical Description . . . . .	411
34.2 Numerical Methods . . . . .	413
34.3 Problems to Solve . . . . .	414
<b>35 Finite Difference Methods</b>	<b>419</b>
35.1 Approximating Spatial Derivatives . . . . .	419
35.2 Finite Difference Grid . . . . .	420
35.3 Approximating Temporal Derivatives . . . . .	422

35.4	Problems to Solve . . . . .	425
<b>36</b>	<b>Iterative Solution of Elliptic Equations</b>	<b>433</b>
36.1	Problems to Solve . . . . .	435
<b>37</b>	<b>Pseudospectral Methods</b>	<b>441</b>
37.1	Fourier Transform . . . . .	441
37.2	Spectral Differentiation . . . . .	443
37.3	Pseudospectral Method . . . . .	444
37.4	Higher Dimensions . . . . .	444
37.5	Problems to Solve . . . . .	445
<b>Bibliography</b>		<b>449</b>
<b>Index</b>		<b>451</b>

# Preface

This book is intended to provide a general overview of concepts and algorithmic techniques useful for doing modern scientific computing. It is based on a 10-week course put together by the author in the Engineering Sciences and Applied Mathematics Department at Northwestern University. It is assumed that the reader understands some elementary programming concepts, such as writing scripts in MATLAB. All the example codes in this text can be downloaded from [www.siam.org/books/se30](http://www.siam.org/books/se30). The website also contains C++ analogues of each of the examples for those using C++ instead of C.

Each part of this book is intended to be largely independent of the others so that the parts can be assembled into an introductory course based on the available computer hardware. For example, a 10-week course can include Part I on C programming, Part III on MPI and distributed architectures, and Part IV on CUDA. Your hardware may differ so it may make sense to substitute Part II on OpenMP in place of MPI if you are using a single computer with multiple processors as opposed to a rack of computers. Likewise, CUDA is designed for NVIDIA graphics cards only, so if using computers with other graphics cards, it may make sense to substitute Part V on OpenCL in place of CUDA. The author covers Parts I, III, and IV when teaching a 10-week course, leaving Part VI as separate reading. For a 15-week course, a suggested path would be Parts I–III plus either Part IV or Part V and adding a little more time devoted to the tools discussed in Chapter 1 and/or applications in Part VI.

Chapter 1 introduces the tools that are needed to implement the example codes and projects through this text including development environments, editors, compilers, makefiles, debuggers, and profilers. It is well worth the effort to explore the various options discussed to find what works best for you, and it will undoubtedly save you time later to be well acquainted with these tools.

Part I of this book is a crash course on C programming to get up to speed for what will come later. The C language is chosen because it is commonly available, widely used, and yet simple enough in structure that it is easy to learn. At the same time, students familiar with C++ will find the content familiar and can use any of the techniques presented in the text without modification. This book is not intended to be a comprehensive description of any of the topics, but rather an entry point to get started and to see how the techniques can be applied to typical numerical methods applications. There are numerous books on C programming if you need a more complete reference, e.g., [1]. The website <http://cppreference.com> is also a great resource for recalling the core functions and their arguments.

The C++ language shares many common features with C while adding object-oriented features of more modern programming languages. While C is the language used in this text, some readers may already be familiar and/or more com-

fortable with the C++ language. For those readers, special comments are added and marked with a gray background for C++ specific concepts. The bulk of these comments are in Part I, where the differences between C and C++ are noted.

The remainder of the book has very little C++ specific information because the interfaces for Parts II–V are entirely based on C and can equally be accessed from C++ in the exact same fashion, so no additional comments will be required.

Part II of this book introduces parallel programming for shared memory machines using OpenMP. It is likely the easiest of the methods of parallelism and should be used at any opportunity. Nearly all modern computers have multiple CPUs, also known as cores. Harnessing that power is not difficult and can result in significant improvement in performance with little effort. It requires very little modification to the algorithms normally written for a single-core serial implementation, usually only the addition of a few compiler directives, to take advantage of opportunities for parallelism. Since this requires no special hardware to implement, it's a natural first step toward understanding the concepts and implementation of parallel algorithms.

Part III addresses programming on distributed architectures. The purpose of distributed computing is to not only use the multiple cores on a single computer but to also utilize cores from other computers (often called nodes) connected by high-speed connections. For example, Northwestern University's Quest system currently has 679 nodes with a total of 16,028 cores. This is tiny in comparison to the current fastest supercomputer, as of this writing, the Sunway TaihuLight, which has 40,960 nodes, each with 256 cores, for a total of 10,649,600 cores. To put all those cores to use on a task, programs must be written that break the task into smaller pieces that each core can do simultaneously. These many subtasks have to be coordinated to get a final answer, which is accomplished by enabling message passing between nodes. MPI programming is one method of message passing between nodes.

Part IV addresses growing interest in high performance computing using graphics cards, also known as GPUs. According to Newzoo [2], global video gaming sales are approaching \$138 billion per year. Competing console manufacturers are in a massive arms race to deliver ever more immersive and realistic experiences in video games. One critical component in this technology is the video cards themselves. Video games (and now maybe science) have driven the performance requirements on video cards to extreme levels, to the point where high-end video cards must be able to draw, update, and redraw at blazing speeds. The high speeds are accomplished by using simpler command sets, i.e., video card processors are able to do fewer and simpler operations compared to their general purpose CPU brethren, but they make up for it by doing those simpler commands on multiple data sources simultaneously. To put it in perspective, a typical computer node may have 4, 8, or more CPUs, but the top-of-the-line NVIDIA TITAN V GPU has 5,120 cores. So where the CPUs individually have more capability, GPU cards make up for it by using large numbers. For people doing scientific computing, it is often the case that an array of data is updated many times repeatedly using the same basic formula. By doing those tasks simultaneously, dramatic improvements in speed can be achieved. CUDA is a language extension to many popular computing languages such as C that makes it straightforward to take advantage of GPU computing power. Part IV is devoted to learning CUDA.

Part V introduces OpenCL, which is an alternative to CUDA in Part IV for those without NVIDIA graphics cards, or perhaps those with NVIDIA cards but who want to develop programs that are more broadly portable. Many of the concepts and techniques in OpenCL are very similar to what is in the CUDA language, but the actual computer

implementation is very different. OpenCL lacks the polish and extensive library support of CUDA, but it is still a very viable means of programming GPUs and can compete in terms of efficiency [3, 4]. Without an NVIDIA card, it's pretty much your only option.<sup>1,2</sup>

In my experience, there's no substitute for learning programming but to build something with it. Part VI discusses some representative scientific computing applications to use as test projects to facilitate learning the concepts. Different types of computing tasks are presented along with enough information about the mathematics and the numerical methods to implement a solution using the various strategies covered in Parts I–V. Because the different types of parallelism may suggest different ways of solving the same problem, a more detailed discussion about algorithms and strategies is provided in the last chapter of Parts I–V.

**Acknowledgments.** I would like to thank the reviewers for their many helpful comments; their insights led to many valuable improvements. Thanks also to Sergey Sheomyakin, Gary Lyons, and the Northwestern University IT Quest support team for helping maintain the systems needed to test the codes and projects in the text. Finally, thanks to the many students who have persevered through my teaching and offered many useful ideas about what should be included to reach the goal of being proficient in scientific computing.

I dedicate this book to my father, Melvyn Chopp, who inspired me to pursue the field of mathematics and taught me how to program an Apple II at an impressionable age, beginning my lifelong journey into scientific computing. This book would not exist without him. Thanks, Dad!

I also would like to thank my mother, Ruth Chopp, sons Henry and Kendel, and dear friend Deb Sokol for their love, support, and endless patience while I was immersed in writing.

---

<sup>1</sup>OpenACC is a rapidly developing platform for heterogeneous hardware that is similar in usage to OpenMP. It has made the trade in favor of greater ease of use in exchange for less direct control of GPU operation.

<sup>2</sup>Apple recently introduced their GPU language METAL [5] into the landscape. At present it is primarily aimed at graphics applications and does not support double precision floating point values, but hopefully this will be added in the future as the language matures.



# **Chapter 1**

# **Tools of the Trade**

Scientific computing requires a minimum number of tools to get the job done. At a bare minimum you will need a means of editing text files and compiling your code. For added convenience a means of organizing complex collections of code files is strongly recommended. These are the tools of the trade in computing, and so some very basic information is provided here to get you started. This book aims for the greatest common denominator and assumes a fairly typical base installation of Linux. Your installation may differ, so if the commands below are not correct for your system, work with your system administrator to find the appropriate tools and libraries in order to build your programs.

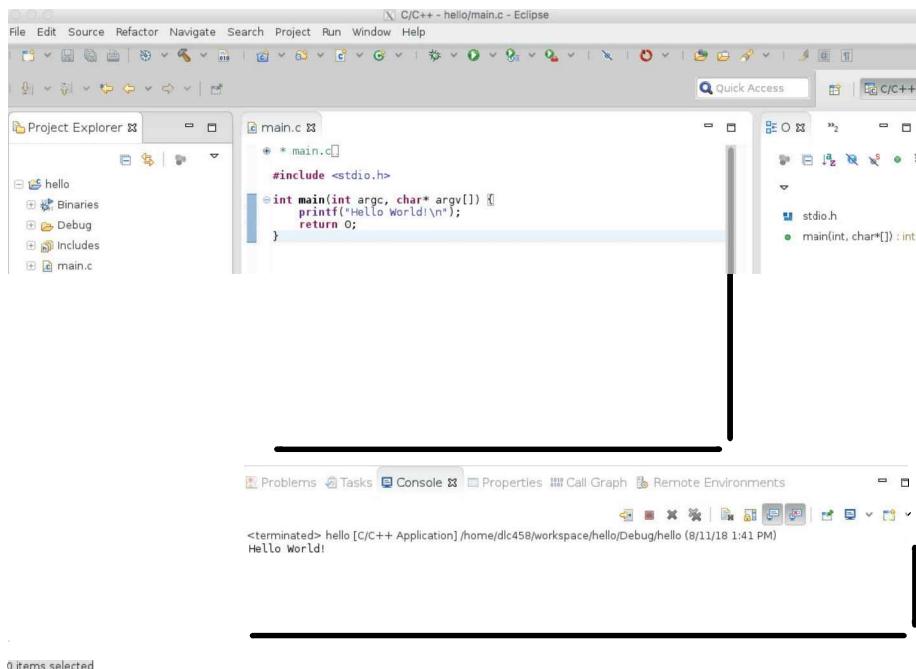
## **1.1 • Integrated Development Environments**

Development environments such as Eclipse [8], the Windows Development Environment [6], and XCode [7] all have integrated edit/compile/debug tools that are nice to use. These are just a few of the most popular development environments available. The Eclipse tool is commonly used on the Linux platform and has the ability to adapt its tools for many languages including C, C++, OpenMP, and MPI, among others. For CUDA, there is the IDE Nsight [9] Eclipse Edition that is Eclipse for CUDA.

A sample view of the Eclipse development environment is shown in Figure 1.1. Within this view, files in a project can be organized and managed, files can be edited, and the project can be built and debugged. The platform also has tutorials to help get started, so the interested reader is encouraged to explore the tutorials there to learn more.

## **1.2 • Text Editors**

Without a development environment, you will need a text editor in order to write and edit your code. Two common editors that are readily available are `vi` (the more modern version is called `vim`) and `emacs` (sometimes called `xemacs`). The former is a bit more obscure in terms of operation but is fast when it comes to doing quick edits and small changes. For larger tasks, `emacs` is a much better choice; it offers things like assistance with indentation, keyword coloring, and a more intuitive interface. To assist you, here



**Figure 1.1.** Example of the Eclipse development environment. The tools for managing files, building code, debugging, and running code are all at the top. The left panel keeps the list of your projects as well as the files within them. The middle panel is a text editor for editing your code, and the right panel keeps track of the various functions defined in your project across multiple files. At the bottom, information such as errors from the compiler or output from a sample execution are shown.

are some useful links to online resources:

**vim:** <https://www.linux.com/learn/tutorials/228600-vim-101-a-beginners-guide-to-vim>

**emacs:** <http://www.gnu.org/software/emacs/tour/>

## 1.3 • Compilers

On Linux platforms, a commonly used open-source compiler is **gcc**. There are two steps to converting program code into an executable: compiling and linking. Compiling takes the written program into pieces of binary code, and linking combines the pieces with libraries of additional code to produce a standalone executable file. The same command is issued for both compiling and linking. Which step being invoked is determined by the inputs and parameters supplied. The two steps can be combined into one as well. When compiling programs, the compiler assumes the suffix of a file determines its type: code files end with “.c” and header files end with “.h”. The output of the compiler is object files that end with the suffix “.o”. Only code files get compiled; header files are included by code files using the **#include** compiler directive.

Suppose, for example, a program consists of two code files, **main.c** and **func.c**, and one header file, **func.h**. The program can be compiled and linked into an executable in one combined command by entering the following at the prompt:

```
$ gcc main.c func.c
```

where the \$ symbol is the Linux command prompt used in this text. Your command prompt may be different. If there are any errors in the program, messages for those errors will be issued. If there are no errors, then the result is a new executable program called `a.out`. To name the program something a little more intuitive, e.g., `myprog`, the output binary can be renamed by using the `-o` option:

```
$ gcc -o myprog main.c func.c
```

Typical projects contain many code files to be compiled, and it is inefficient to compile and link all of them every time the program is built. For large codes, compiling can take a little while, and recompiling all the files because a small change was made in one file wastes time. In that case, only the updated file and any that depend on it should be compiled and then joined to the other previously compiled files. This is done by using the `-c` compiler option, which translates a single code file into an object file, with corresponding suffix “`.o`”. To compile this way is a multistep process where the first two lines invoke the compiler and the last line invokes the linker:

```
$ gcc -c main.c  
$ gcc -c func.c  
$ gcc -o myprog main.o func.o
```

If the file `func.c` gets modified, then only the second two commands must be run to bring the code up to date. The first two commands output the object files `main.o` and `func.o`, respectively. The object files are self-contained except for possibly some loose ends that are resolved during the linking process.

Linking is the last step above that connects all the loose ends together. It is at this stage that functions from different files and functions referenced from external libraries are assembled into a completed executable file. To avoid problems with the linker, be sure that function names are not duplicated and that exactly one function named “`main`” exists. The `main` function is always where the program will start.

For convenience, some external libraries are included by default. For example, the commands for doing input/output at the terminal are in a library that is automatically linked in. At the end of each part of this book, there are other libraries that will be needed and they will also be linked at this stage using the `-l` option. For example, to add the standard math library, the linking line would be changed to

```
$ gcc -o myprog main.o func.o -lm
```

The suffix for C++ code files can be either “`.cp`” or “`.cpp`”, and the corresponding compiler is `g++` instead of `gcc`.

A feature of C++ is that functions can be overloaded, meaning that multiple functions may have the same function name but take different arguments. In that case, the function name plus its arguments must be unique to prevent a conflict during linking. The one exception is the function `main`, which must always be unique and may only take an integer and an array of strings as inputs.

**Table 1.1.** Optimization options for the *gcc* compiler.

Flag	Meaning
-O0	No optimization
-O1	Basic optimization without additional compile time (default)
-O2	Optimize further without increasing size of the executable
-O3	Optimize even further for speed
-Ofast	Optimize for speed cutting some corners (caution!)

**Table 1.2.** Illustration of improvements possible with various optimization flag values. Results based on compiling and running Example 6.6. The data illustrates how the compiler makes a trade-off between the size of the executable and the speed of the execution.

Flag	Mean Run Time	Executable Size
-O0	$6.428022 \times 10^{-2}$ secs	13272 kbytes
-O1	$4.091858 \times 10^{-2}$ secs	13280 kbytes
-O2	$4.106516 \times 10^{-2}$ secs	13280 kbytes
-O3	$2.015679 \times 10^{-2}$ secs	17376 kbytes
-Ofast	$1.297233 \times 10^{-2}$ secs	18856 kbytes

### 1.3.1 • Optimization Flags

Compilers have many options that can be used to fine-tune the final product, most of which are well beyond the scope of this text. However, one important option for scientific computing is the “-O” option (note this is the letter “O,” not the number zero). This option determines the level of optimization that the compiler can use to improve the performance of the executable. The options may vary for other compilers, but the options for the *gcc* compiler are listed in Table 1.1.

The various levels of optimization should be used with care. Higher levels of optimization should have the results validated against the base level of optimization to ensure that the results are not compromised in any way. The results of using the `gettimeofday` method in Example 6.6 with the different optimization levels are shown in Table 1.2. A six-fold increase in speed justifies the importance of the optimization setting, but the actual amount of improvement for a given code will vary.

## 1.4 • Makefiles

It is not unreasonable to have tens or even hundreds of code files that will be used to build an application and, in addition, their corresponding header files. Suppose a change is made in a few files, but not all, that requires recompiling the program. The steps above could be repeated for the changed files, but that requires remembering which files were edited, typing in the correct commands to build those object files, and then linking them all together. This is a process that is prone to mistakes, so to make it more manageable, the `make` command was created. A makefile consists of a bunch of targets, file dependencies, and commands that describe how to make those targets.

To use `make`, create a file called “`makefile`”. It can also be named “`Makefile`”, but any other name would require explicitly typing in the name of the makefile as part of the command, so it’s best to stick with `makefile`. Following the compiling example above, the makefile would then have contents such as this:

```
1 myprog: main.o func.o
2 <tab> gcc -o myprog main.o func.o
3
4 main.o: main.c func.h
5 <tab> gcc -c main.c
6
7 func.o: func.c func.h
8 <tab> gcc -c func.c
```

Note that where <tab> is written, a tab key has been pressed, not those literal characters. It must be a tab, not just space.

To understand what this file does, let's walk through what happens when the `make` command is issued. To begin, the target to be made must be identified. The targets are the expressions that begin at the left and end with a colon, such as `myprog`. The default target is the one at the top of the `makefile`. To build the default target `myprog`, simply use the command `make`. To make a different target such as `main.o`, the command would be `make main.o`. This will only build `main.o` and not update `myprog`. There are other targets that people often create such as `clean` that could be added to the above `makefile`:

```
1 clean:
2 <tab> -rm myprog *.o
```

This target would delete the executable and all the object files in the directory. This can be useful if something gets corrupted, or there's some dependency problem that's not being resolved, or just to quickly remove stray data files that are generated during execution of a program. The simple command `make clean` would clean up all the stray object files and the executable. Note that the extra “-” before the `rm` command lets `make` know that it's okay if the command fails, for example, if neither `myprog` nor any `.o` files exist in the directory.

After the colon on the line with the target are the dependencies. When `make myprog` is entered, the first thing `make` does is look for the target and check for the dependencies (if any). If no dependencies are listed, then the commands for that target are executed. If there are dependencies, for example, in this case `main.o` and `func.o` are listed, each dependency is checked to see if it is also a target. If a dependency is a target, then a `make` command is generated for that dependency. After those subsidiary `make` commands are completed, the timestamp on the target is compared to the dependencies. If any of the dependencies are newer, then the commands listed below that target are executed. In this way `make` works recursively through the `makefile` to ensure that every part of the code that is used to build the target is up to date.

For example, suppose that the file `func.c` is edited, which results in the timestamp on `func.c` being newer than `func.o`. Then when issuing the command `make myprog`, the dependency `func.o` is found, and so the command `make func.o` is automatically executed. The target `func.o` depends on the files `func.c` and `func.h`. Since neither of these is a target in the `makefile`, they are assumed to be files. Checking the timestamps reveals that `func.c` is newer than `func.o` because of the recent edit; therefore the command `gcc -c func.c` is executed to update `func.o`. Updating the file `func.o` also updates the timestamp. Returning to the original target `myprog`, the timestamp of `myprog` is compared to `func.o` and `main.o`. Since `func.o` was recently updated, its timestamp is newer than `myprog`, and hence the command `gcc -o myprog main.o func.o` is executed to bring `myprog` up to date.

So why is this better? A carefully constructed `makefile` means that a single `make` command brings the program up to date in the most efficient manner. There is no need

to track which files were modified and require recompiling because `make` will check the timestamps and update according to the dependency tree so that compiles are done in the right order to produce the most up-to-date version of the executable code.

The description of makefiles here is quite simple. To learn more, check out this website:

```
http://www.gnu.org/software/make/manual/make.html
```

There are other tools that perform the same task, e.g., the tool `ninja` [10], and there are tools that generate makefiles, such as `cmake` [11].

## 1.5 • Debugging

No program longer than a few lines works perfectly on the first attempt. As programs become longer and more complex, issues arise that may defy explanation by just staring at the code. Often, the symptom of a problem appears much later than the cause of the problem so that tracking down the cause and correcting it can be very difficult without some help such as from a debugger. A debugger steps through the code while it's running and can display the values of variables so that the logic and flow of the program can be checked against what was planned. If a program reports a segmentation fault or other unexpected termination, a debugger can show where in the code it happened and hopefully a clue as to why. Debugging code is definitely an art form that every programmer must develop to be effective.

In order to use a debugger, code must be compiled with an extra `-g` flag:

```
$ gcc -g -c main.c  
$ gcc -g -o myprog main.o
```

This extra flag asks the compiler to put debugging information into the object file, in this case `main.o`, so that the debugger can display the lines of code and variable names associated with memory while it executes the program.

The base-level debugger on Linux systems is the GNU debugger called `gdb`. To run a program with the debugger, type

```
$ gdb myprog
```

This will bring up a new command prompt, (`gdb`), within the debugger. To run the program, at the (`gdb`) command prompt type

```
(gdb) run myargs
```

where `myargs` are the arguments to run the program on the command line. To stop at a particular line in the code, before running the program type

```
(gdb) break 123
```

to stop the debugger at line 123 of the current file. If the line is in a different file, for example, `myfunc.c`, then the `break` command would be

```
(gdb) break myfunc.c:123
```

When the program stops at a break point, it can be stepped through one line at a time by using the **next** command or the **step** command to follow the progress through the program more carefully. To resume running the program use the **continue** command. Many commands in **gdb** can be abbreviated; these commands are so frequent it's only necessary to type **n**, **s**, or **c** for **next**, **step**, or **continue**.

The **gdb** debugger has many more capabilities that can be found in the help files by typing **help** at the command prompt. It is well worth making the effort to learn how to use it. Documentation can be found at this website:

<https://www.gnu.org/software/gdb/>

There is a nice graphical front end for **gdb** called **ddd** available on Linux systems using X11 windows. It is invoked the same way as **gdb** but provides a nice view of the code and variables that is easier to use than **gdb** alone. Documentation for **ddd** can be found at this website:

<https://www.gnu.org/software/ddd/>

Most development environments have a debugger linked to them so debugging can be done within the environment. Check the documentation for the development environment to learn how to use it.

## 1.6 • Profilers

Once code is written and tested, the next step is to optimize performance. Profiling tools are used to report the amount of time or number of instructions used in each part of the code so that the parts of the code that are the most time-consuming are easily identified. The classic tool for this task is **gprof**, but a more modern and useful tool that is worth learning is **valgrind**. The **valgrind** tool has a couple of very handy uses, particularly for those new to programming.

To use **valgrind**, first compile and build the program with the debugging flag, **-g**, turned on as described in Section 1.5. The default **valgrind** tool is **memcheck**, which is invoked by calling **valgrind** before running the program. For example, to test the program in Example 6.6, which has the name “**timecheck**”, call **valgrind** followed by the normal program command including arguments:

```
$ valgrind timecheck 1000000 10
```

Suppose Line 163 in Example 6.6 is removed; then the results of running **valgrind** are shown in Example 1.1. Note that under the leak summary, it is noted that 8 megabytes are definitely lost. This is an indication that there is a memory leak, which is a result of commenting out Line 163. Ideally, there should be no lost memory. Memory leaks will be discussed further in Section 3.5. This tool will also check whether accessing an array goes outside its bounds or variables are used before they're initialized.

### Example 1.1.

```
$ valgrind ./timecheck 1000000 10
==29016== Memcheck, a memory error detector
==29016== Copyright (C) 2002–2015, and GNU GPL'd, by Julian Seward et al.
==29016== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
```

```

==29016== Command: ./timecheck 1000000 10
==29016==
mean elapsed time = 5.935650e-02, variance = 3.746721e-06
mean elapsed time = 1.000000e-01, variance = 1.000000e-01
mean elapsed time = 5.865679e-02, variance = 4.142121e-07
mean elapsed time = 5.874093e-02, variance = 3.920762e-07
==29016==
==29016== HEAP SUMMARY:
==29016== in use at exit: 8,000,000 bytes in 1 blocks
==29016== total heap usage: 3 allocs , 2 frees , 16,001,024 bytes allocated
==29016==
==29016== LEAK SUMMARY:
==29016== definitely lost: 8,000,000 bytes in 1 blocks
==29016== indirectly lost: 0 bytes in 0 blocks
==29016== possibly lost: 0 bytes in 0 blocks
==29016== still reachable: 0 bytes in 0 blocks
==29016== suppressed: 0 bytes in 0 blocks
==29016== Rerun with --leak-check=full to see details of leaked memory
==29016==
==29016== For counts of detected and suppressed errors, rerun with: -v
==29016== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

```

In addition to the memory usage checker, **valgrind** can be used for checking the computational cost of subroutines. This tool is called **callgrind** and is invoked by specifying the tool

```
$ valgrind --tool=callgrind ./timercheck 10000 10
```

After the code is finished running, a file called **callgrind.out.#** will be created where the # will be digits. To interpret the results, run

```
$ callgrind_annotate callgrind.out.#
```

Sample results are shown in Example 1.2 for the code in Example 6.6. The number in front of each line of code is the number of instructions executed in that part of the code. For a more detailed listing, use the command

```
$ callgrind_annotate --auto=yes callgrind.out.#
```

Note that the ends of some of the lines in Example 1.2 have been truncated in order to fit on the page.

### Example 1.2.

```

$ callgrind_annotate callgrind.out.23604
Profile data file 'callgrind.out.23604' (creator: callgrind-3.11.0)

II cache:
D1 cache:
LL cache:
Timerange: Basic block 0 - 640177
Trigger: Program termination
Profiled target: ./timercheck 10000 10 (PID 23604, part 1)
Events recorded: Ir
Events shown: Ir
Event sort order: Ir
Thresholds: 99
Include dirs:
User annotated:

```

Auto—annotation: on
Ir
13,170,766 PROGRAM TOTALS
Ir file : function
11,599,720 timercheck.c: diff [/ stash/chopp/ serial /timercheck] 1,081,666 /build/glibc-C15G7W/glibc-2.23/math../sysdeps/ieee754/dbl-64/s_sin.c:__sin_avx [/ lib/x86_64- 160,021 timercheck.c: init [/ stash/chopp/ serial /timercheck] 90,000 /build/glibc-C15G7W/glibc-2.23/math../sysdeps/i386/fpu/fenv_private.h:__sin_avx 56,475 /build/glibc-C15G7W/glibc-2.23/elf/dl-addr.c:__dl_addr [/lib/x86_64-linux-gnu/libc-2.23.so] 31,617 /build/glibc-C15G7W/glibc-2.23/elf/dl-lookup.c:__do_lookup_x [/lib/x86_64-linux-gnu/ld-2.23.so] 20,689 /build/glibc-C15G7W/glibc-2.23/elf/dl-lookup.c:__dl_lookup_symbol_x [/lib/x86_64-linux-gnu/ld]

The number on the left indicates how many computer instructions were executed for each function, which can roughly translate into the time cost. In this example, clearly the time cost of executing the finite difference in the `diff` function was substantial, but that's to be expected. What may be surprising is that the `sin` function, listed second, turned out to be expensive. Many built-in functions, such as trig functions, are expensive, so reducing their use where possible would be to your advantage.

## 1.7 • Version Control

Professional programmers rely heavily on version control software to manage projects. Version control makes software development more robust because it allows programmers to test new algorithms or subroutines without risking losing previous versions in case a modification does not work out as planned. A popular version control system in common use is `git`. Many development environments also include some form of version control software, either `git` or something similar, as part of the environment. If using a development environment, check the documentation for the use of version control. In the subsequent discussion, it is assumed that a single programmer is using `git` from the Linux command line.

The basic idea behind version control is that a programming project that contains one or more files is all kept within a repository. To begin working on a project, the current version of the project is checked out from the repository, leaving the programmer with a copy of the current version of all the files. It is now possible to make changes to any of the files and hopefully improve them. When a file is updated and the new version is to be inserted back into the repository, the file is committed to the repository.

To make this more concrete, suppose a new project is created in a directory named `mycode`, with two files: `main.c` and `makefile`. To create a new `git` repository, type

```
$ git init /path/to/mycode
```

This creates a new directory inside `mycode` called “`.git`” containing all the repository information necessary to maintain the project.<sup>3</sup> That directory should not be touched directly, only through the use of the `git` commands.

The next step is to add the files to the repository. Assuming the current directory is `mycode`, the two files are added to the repository with the command

<sup>3</sup>In Linux, a file or directory that has a name beginning with “.” does not appear in a default listing of files using `ls`. To verify that the “`.git`” directory is created, used the command `ls -a` to list all files.

```
$ git add main.c makefile
```

The current version of the files is not yet in the repository; it just makes the repository aware of the files for future tracking.

To see what the current status is, type

```
$ git status
```

which results in the following output:

```
On branch master
Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

    new file:   main.c
    new file:   makefile
```

This means that `main.c` and `makefile` have been made part of the repository, but the current versions have not yet been committed.

To make the current version of these files part of the repository, they must be committed using the command

```
$ git commit main.c makefile
```

You will be prompted to enter a message for this update in the repository. The message should contain useful information for future reference that describes the changes made. At this point, using the command `git status` will report that there is nothing to commit, which means every file is consistent with the current version in the repository.

Suppose `main.c` gets updated; then `git status` will see that it differs from the repository and report that it needs to be committed. However, suppose the changes made turn out to cause problems and it would be better to return to the previous version; then it can be retrieved with the command

```
$ git checkout -- main.c
```

which will replace the current version in the directory that had unwanted changes with the version last committed to the repository. Even if the file is deleted, it can be recovered from the repository with the `git checkout` command.

The commands discussed here only scratch the surface of what version control software can do. For further information, there are several books on the subject, e.g., see [12].

---

## Exercise

- 1.1. Type the program listed in Example 2.1 and save it in a file called `hello.c`. Create a makefile called `makefile` that will build the executable named `hello`. Use the `make` command to build the program. Finally, run the program. Extra credit: make a new target in your makefile called `run` so that when you type `make run` it automatically builds your program and then executes it.

## **Part I**

# **Elementary C Programming**



The programming language that will be used in this book is the C programming language. There are of course many languages that could be used to explore high performance computing. Other popular languages include Fortran, C++, and Python, to name a few. Some languages include higher-level functionality such as object-oriented languages that encapsulate functions into objects, making it easier to break complicated problems down into smaller, more manageable pieces. The reader is certainly encouraged to explore these other options, but C will be the language of choice here because it is widely available, is less complicated to explain, and yet is a gateway to understanding a more sophisticated language like C++. Furthermore, the fundamental concepts for high performance computing are essentially the same across all the languages, it's just the syntax that changes.

In Chapter 2, the ubiquitous “Hello World!” program is used to introduce the fundamental components of every C program including how to add comments to the program, how useful functions like input/output (I/O) functions are accessed by adding header files, and how input arguments are passed from the operating system to the main program.

In Chapter 3, the data types used in C programming are discussed. This includes the base data types, such as integers, floating point values, and strings, and the basic mathematical operations that can be applied to them. The chapter also covers how to create collections of data into arrays and multitype structures. Much of scientific programming revolves around computing operations on arrays, so this is a critical section to understand.

It does no good to do a long and complex calculation and then not save the results for later use. Therefore, Chapter 4 discusses the various ways of reading input and printing or storing output both to/from the terminal and files on disk.

Operations on arrays require a robust understanding of loops for flow control, so Chapter 5 introduces the basic program flow control structures for creating loops and for branching according to conditions.

Writing a large and complex algorithm into a single file is not just impractical but very bad practice. Experienced programmers always have an eye for building code that will be reusable for other purposes, either for the task at hand or for codes that may come in the future. This is possible only through the use of defining functions, which are the fundamental building blocks of programs. Thus, Chapter 6 introduces functions and how to interact with them. The chapter concludes with some specialized functions for measuring time. These tools are useful for determining the time cost of running a program and where to spend the most effort to improve efficiency.

There’s no reason to develop your entire code from scratch: stand taller by standing on the shoulders of those who have come before you. In other words, there’s no reason to write your own linear system solver or Fourier transform function when others have already done it, tested it, and optimized it. Chapter 7 presents some of the standard libraries of code available for solving linear algebra problems, taking Fourier transforms,

and generating random numbers. Companions to these libraries in the other parts of this book devoted to parallel programming will be introduced so that many of the concepts covered in this chapter will be easier to understand later.

Finally, Chapter 8 gives specific instructions for solving the sample projects described in Part VI. The projects are an opportunity to test your abilities in C programming by solving some representative scientific computing applications. They will also serve as a basis for comparison with implementations using various parallel computing techniques so that you appreciate the improvement in performance that parallel computing provides.

## Chapter 2

# Structure of a C Program

Those already familiar with a programming language will no doubt find many parallels with C. Languages such as C++ and Java have very strong similarities, even including much of the syntax, but all languages have the same underlying structure. No matter what language is used, it is absolutely essential to comment the code. To be an effective programmer, code should be made reusable, and it is reusable only if it is clear what each function needs for input and what it produces for output. It is not reusable if its purpose and usage are easily forgotten. When working in a team, it is even more essential that comments are clear and consistent so that others can reliably use your work. Careful and frequent comments can help the reader understand the logic of the program so that if changes are necessary later, it can be done easily and confidently. This chapter introduces the core components of a program: where it starts, how input values can be included on the command line, and how to finish. It also covers the different styles of comments and what is good practice for commenting your code.

C programs consist of a collection of functions in one or more files that describe the operations required to complete a task. For C programs, one particular function is important, and that is the `main` function. When a C program begins, the `main` program is the place where the program will start. It doesn't matter which file contains the `main` function, but there can be only one.

Like almost every beginning program text dating back to the original C programming language [1], we will start with the classic “Hello World!” program to understand the basic structure of a C program. Example 2.1 (with many comments) accomplishes the simple task of printing “Hello World!” and illustrates the main pieces required for C programs.

### Example 2.1.

```
1 /*  
2  Hello World  
3  
4  Demonstration showing a simple program that is self-contained.  
5  
6  Program written by David Chopp  
7  
8  Editing History:  
9  
10 2/14/14: Initial draft
```

```

11  7/29/17: Changed formatting of comments
12 */
13
14 // Programs often require functions from built-in libraries
15 // or user-written files
16 // Interface information for those functions is stored in header files
17 // and loaded before the code is compiled.
18 // This program uses terminal I/O, so it uses the stdio library.
19 // This is very common...
20
21 #include <stdio.h>
22
23 /*
24 int main(int argc, char* argv[])
25
26 The main program is the starting point for a C program.
27 There can only be one "main" function. All others must be
28 uniquely named something else.
29
30 Inputs:
31     int argc: When the program is run, argc contains the number of
32             arguments given to the program, e.g. if you type
33             "hello there" to the command line, then argc will have
34             the value 2.
35
36     char* argv[]: When the program is run, argv contains an array of
37                 strings, where each string is what was typed, e.g. if
38                 you type "hello there" to the command line, then
39                 argv[0] will have the value "hello" and argv[1] will
40                 have the value "there"
41
42 Outputs:
43     int: The main program should return 0 if successful and return
44           a nonzero value if it encountered an error.
45 */
46
47 int main(int argc, char* argv[])
48 {
49     // It is important to get in the habit of thoroughly commenting
50     // your code.
51
52     printf("Hello World!\n");
53
54     // All done, no errors, so return 0
55
56     return 0;
57 }
```

If using C++, Line 21 would be replaced with the line

21 #include <iostream>

and Line 52 would be replaced with the corresponding line

52 std::cout << "Hello World!" << std::endl;

## 2.1 • Comments

Lines 1–12 in Example 2.1 are comments that indicate the title, authorship, purpose, possible input arguments, output results, and editing history for the program. Because it is so important for minimizing effort when doing computing, using thorough commenting and documentation throughout programs is essential. Effort put into coding now will pay off handsomely later. So make comments thorough and detailed.

There are two ways in which comments can be put into programs, and Lines 1–12 illustrate one of them. Text sandwiched between “`/*`” and “`*/`” are comments. This style of comment is most useful when making multiline comments. Lines 14–19 illustrate another form of comments. Any text following “`//` *on the same line*” is also a comment. This is best for short one- or two-line comments. Either method of adding comments is fine, as long as **code is commented!**

Example 2.1 illustrates three key places where comments should be routinely added to code:

1. At the top of each file include information about the contents of the file like that included on Lines 1–12. This may include descriptive information, authorship, and editing history.
2. At the beginning of each function, comments should be provided that describe the function, what inputs it expects, and what outputs it produces. Lines 23–45 illustrate the comments related to the `main` function that begins on Line 47.
3. Each substantive step of a program should receive a comment so that others can follow the logic of the program. This first example is too simple to provide a good illustration, but Line 54 is the idea; many more examples can be found throughout this text.

In the remainder of this book, the first and second types of comment may be omitted in the code examples for the sake of brevity, and it should not be interpreted that they are not necessary in practice.

## 2.2 • The Preamble

For many programs, libraries of code are needed to handle certain operations. For example, rather than having to write the instructions for how to translate text from the program to draw pixels on the screen for every program, it's better to use code others have written for that purpose. Collections of code like this are contained in libraries to handle commonly needed tasks so that attention can be focused on more unique higher-level tasks. To include these libraries and prebuilt functions, there are two steps required: (1) the input and output parameters of the functions must be declared in order to compile, and (2) the precompiled code stored in the library must be added to the final executable during linking. The latter will be discussed later; for now it should suffice that many basic library functions are automatically linked to the final code, so no additional actions are required.

Every function must be declared. Even for a simple statement for printing text to the screen, there are a host of functions required to make it happen. Rather than having to type all that in, the function declarations are gathered into a single header file. A header file is a collection of function declarations, definitions of constants, macros, and other statements required for the associated library. All that information can be included into a program by using an include compiler directive. A compiler directive is a line that

begins with a “#” character and it instructs the compiler to perform an action before continuing. On Line 21, the compiler is directed to include the header file “`stdio.h`”. This header file lists the function declarations relevant to text I/O commands for the terminal window, among other things. Thus, the program will not compile without including this file because otherwise the compiler will not know about the function `printf` on Line 52. Note also that `stdio.h` is inside angle brackets, i.e., “`<stdio.h>`”. Angle brackets are used for system header files, i.e., files that you did not write.

Not all header files are system files. Suppose a programmer writes a function that multiplies two matrices together. To reuse that function again in the future for other purposes, the function may be kept in a file `matrixops.c` that contains many functions for doing matrix operations. To make use of it in a new program, a header file called `matrixops.h` would be created that contains the declarations for the functions in the `matrixops.c` file. The new program would then be able to use the functions in `matrixops.c` without knowing the details of how they are implemented by including the header file. In this case, when `matrixops.h` is included, the filename should be inside double quotes in the compiler directive instead of in angle brackets:

```
#include "matrixops.h"
```

The different ways to specify the file to be included provide clues to the compiler for where to find the file requested.

Global variables, compiler macros, and type definitions, among other things, are all put in the preamble, either entered directly or included through a header file. We will expand on this as we go.

## 2.3 • The Main Program

As noted above, every program must contain exactly one function called `main`. Line 47 of Example 2.1 will be the same for every C program, so it will be instructive to take apart Line 47 of the example to understand the structure of a function definition.

The input arguments for a function are given as a list following the name of the function inside parentheses. Thus, the `main` program here contains two input arguments, the first an integer named `argc`, and the second an array of strings named `argv`. These particular names for the input are not required, but it's generally the way `main` programs are set up so there's no advantage to changing them. The typing of variables and construction of arrays will be discussed later; for now just accept that's what the arguments are.

When the program is run, the variable `argc` will contain the number of arguments in the command line. For example, suppose Example 2.1 is compiled into the executable program called `hello`, and the following command is issued:

```
$ hello there my friend
```

Then `argc` will equal four because there are four space-separated expressions in the command. The `argv` array will always have length given by `argc` and will contain the string expressions {“hello”, “there”, “my”, “friend”}. The first string is always the name of the program, in this case “`hello`”.

The return type of the `main` function appears in front of the function declaration; in this case it is specified to return an `int` on Line 47. The `main` program communicates success or failure to the operating system through this value. It returns the value of

zero if the program ran successfully and a nonzero value, usually one, if there was an error or it failed in some way. For this simple program, the only things that could go wrong would be within the I/O system itself, which is beyond our control. Therefore, after successfully printing “Hello World!” the value of zero is returned to the operating system on Line 56.

The `main` function is defined by the sequence of statements between the characters “{” on Line 48 and “}” on Line 57. Anything outside those brackets is not part of the `main` function execution. The `main` function could contain the entire program, but typically it will involve calls to other functions, for example, `printf`, written to complete the program. In this simple example, Line 52 is the only line that will actually *do* something where it prints “Hello World!” in the terminal (without the double quotes). The function `printf` is used to send formatted text to the terminal window. The `printf` function will be discussed in detail in Chapter 4.

Finally, note that each statement of the program is terminated by a semicolon. This is a requirement that separates statements, which are the steps in a program. Any line not terminated by a semicolon is assumed to be continued on the following line to complete the statement. A line of the program may be broken over several lines in the text file, and the line is completed by terminating with a single semicolon.

---

## Exercise

- 2.1. Type Example 2.1 into an editor, compile the program, and run it.



# Chapter 3

# Data Types and Structures

A critical component of scientific computing is data storage, retrieval, and organization. The better the data is organized and designed, the more efficient a program will be. After all, the reason for using high performance computing is to solve *hard* equations that require significant effort to compute. In this chapter the various data types and their properties and limitations are discussed. We will also look at elementary mathematical operations such as addition and multiplication, logical operations like AND and OR, and standard mathematical functions such as `sin` and `exp`. Finally, we'll look at how data can be organized into arrays and structures.

## 3.1 • Basic Data Types

C programs require all data to have a type, which will determine how it can be used. Conversions between data types exist and are generally intuitive, for example, assigning an integer value of 2 to a double precision floating point variable will result in the value 2.0 as expected, and assigning the double precision value 2.73 to an integer will result in the value 2 (truncation, not rounding). Nonetheless, it is best to use precisely the data types needed for a computation and not just define everything to be a floating point value for convenience.

Table 3.1 lists the relevant basic data types. In addition to each of these basic data types, each of the types can be modified with qualifiers: `short`, `long`, `long long`, `signed`, and `unsigned`. The modifier `short` means that the type should use half the normal storage if possible. The modifier `long` means that the type should use double the normal storage if possible. The modifier `unsigned` is applied only to decimal values and means that the value will only be nonnegative. Assuming the value is nonnegative frees up the bit used for the sign to allow the variable to have double the range. For example, compare the range of values for an `int` to an `unsigned int` in Table 3.2. The modifier `signed` can be used to be explicit, but it is generally assumed and so is used infrequently. In other words, `signed int` is equivalent to simply using `int`.

In addition to the types listed here, another set of useful data types is `float complex`, `double complex`, and `long double complex` for storing complex-valued data. A discussion of these data types and their associated complex-valued functions will be postponed until Section 7.2.1.

**Table 3.1.** Table of basic and modified data types in C programming. Note that the type `bool` may require the use of the header file `stdbool.h`. The constants `true` and `false`, which evaluate to 1 and 0, respectively, are also defined in `stdbool.h`. The sizes listed here may differ on some systems; thus the function `sizeof` should be used to determine the values for your system.

Type Name	Usual Size	Description
<code>char</code>	1 byte	Single character
<code>bool</code>	1 byte	Boolean value
<code>int</code>	4 bytes	Integer value
<code>float</code>	4 bytes	Single precision floating point
<code>double</code>	8 bytes	Double precision floating point
<code>short int</code>	2 bytes	Short integer
<code>long int</code>	8 bytes	Long integer
<code>long long int</code>	8 bytes	Extralong integer (rare)
<code>long double</code>	16 bytes	Quadruple precision floating point
<code>unsigned char</code>	1 byte	Single character
<code>unsigned int</code>	4 bytes	Integer value
<code>unsigned short int</code>	2 bytes	Short integer
<code>unsigned long int</code>	8 bytes	Long integer
<code>unsigned long long int</code>	8 bytes	Extralong integer (rare)

The type name `bool` is part of the C++ language and does not require an additional header file for its use.

Tables 3.2 and 3.3 also introduce the topic of macros. Macros are character strings that the compiler will replace during compile time with a defined value. A macro is defined using the `#define` compiler directive. The macros listed in Tables 3.2 and 3.3 are usually defined in the system header file `limits.h` in this way:

```
#define INT_MAX 2147483647
```

The macro `M_PI` is sometimes defined to be the number  $\pi$  in the header file `math.h`, but not always. If it is not, you may add a macro to define it when you need it, e.g.,

```
#define M_PI 3.1415926535897932384626433832795
```

To be really safe, a guard against a duplicate definition should be used in case it is defined:

```
#ifndef M_PI
#define M_PI 3.1415926535897932384626433832795
#endif
```

This tells the compiler to check whether `M_PI` is already defined. If not, then define it so it can be used later. The compiler watches for character strings that are not inside double quotes and that match the list of known macros. When there is a match, the compiler replaces the matching text with the macro definition, then compiles the resulting code.

**Table 3.2.** Table of basic and modified data types and their value ranges. Note that for floating point types, the ranges are from the smallest positive representable number to the largest. The actual range of values would be plus or minus the maximum. These values are system dependent, so when needed, the macros listed should be used instead of the explicit values shown in the table.

Type Name	Macros for Limits	Typical Values
char	[CHAR_MIN, CHAR_MAX]	[-128, 127]
bool	[CHAR_MIN, CHAR_MAX]	[-128, 127]
int	[INT_MIN, INT_MAX]	[-2147483648, 2147483647]
float	[FLT_MIN, FLT_MAX]	$1.175494 \times 10^{-38}$ , $3.402823 \times 10^{38}$
double	[DBL_MIN, DBL_MAX]	$2.225074 \times 10^{-308}$ , $1.797693 \times 10^{308}$
short int	[SHRT_MIN, SHRT_MAX]	[-32768, 32767]
long int	[LONG_MIN, LONG_MAX]	[-9223372036854775808, 9223372036854775807]
long long int	[LLONG_MIN, LLONG_MAX]	[-9223372036854775808, 9223372036854775807]
long double	[LDBL_MIN, LDBL_MAX]	$3.362103 \times 10^{-4932}$ , $1.189731 \times 10^{4932}$
unsigned char	[0, UCHAR_MAX]	[0, 255]
unsigned int	[0, UINT_MAX]	[0, 4294967295]
unsigned short int	[0, USHRT_MAX]	[0, 65535]
unsigned long int	[0, ULONG_MAX]	[0, 18446744073709551615]

**Table 3.3.** Table of machine epsilon values for different floating point types and their associated macros. Note that these values are system dependent, so when needed, the macros listed should be used instead of the explicit values shown in the table.

Type Name	Macro for Machine Epsilon	Value
float	FLT_EPSILON	$1.192093 \times 10^{-7}$
double	DBL_EPSILON	$2.220446 \times 10^{-16}$
long double	LDBL_EPSILON	$1.084202 \times 10^{-19}$

In C++, programmers are encouraged to use constants declared with a type instead of using macros. This way, overloaded functions will know what precision the value is and it can be used properly. Thus, instead of defining the macro M\_PI, the value of  $\pi$  would be defined as

```
#ifdef M_PI
const double pi = M_PI;
#else
const double pi = 3.1415926535897932384626433832795;
#endif
```

and then used elsewhere in the program simply as pi.

Table 3.3 shows the machine epsilon values for the different floating point types. Machine epsilon is defined as the smallest number that when added to one will give a distinct value. Checking the exponent of machine epsilon shows the number of significant digits that a type can accurately represent. Thus, single precision `float` offers about 7 digits of accuracy, while `long double` offers 19 digits of accuracy. Choosing the right precision is important when speed is a factor. Using `float` will be faster but results in less precision, while using `long double` will be slower but offers greater precision. This is a trade-off that must be weighed in every numerical method code. As an aside, one common strategy when writing code is to use a `typedef` in a header file like this,

```
typedef float real;
#define EPSILON FLT_EPSILON
#define REAL_MIN FLT_MIN
#define REAL_MAX FLT_MAX
```

and then defining all floating point variables as type `real`. The `typedef` command makes it possible to define a new type based on existing types. In this case, a new type named `real` is created and is the same as the type `float`. By doing this, the underlying precision in the code can be adjusted by changing it in one location.

Variables to be used in a given function can be defined anywhere before they are used, but it is good practice to define them at the beginning of the function definition. Using naming conventions for all your functions, variables, etc., is a wise decision. As codes get increasingly more complex, particularly when incorporating parallel and multithreaded codes, understanding the code can get increasingly challenging. Using variable names that are meaningful and using a consistent naming scheme will ease the pain of modifications and improvements later on.

Variables may optionally be given an initial value when they are first declared. For example, in this main program, two variables named `index` and `count` are declared. The variable `index` will be initialized later, while `count` is initially set to be zero.

### Example 3.1.

```
1 int main(int argc, char* argv[])
2 {
3     int index;           // index is declared here, but not initialized.
4                     // It must be initialized before it is used.
5     long int count = 0; // count is initialized to zero here.
6
7     // Rest of the program...
8 }
```

Note that some compilers will initialize all variables to zero by default, but that is not a language requirement, so it is strongly recommended that all variables be initialized with a value before they are used. Some compilers, and the tool `valgrind`, will issue warnings about use of uninitialized variables when it occurs.

## 3.2 • Mathematical Operations and Assigning Values

An assignment statement is how variables are given values. The variable on the left receives the value computed on the right. For example, in the statement below, the

variable **x** will be one more than twice the value in variable **y**:

```
x = 2*y + 1;
```

The basic arithmetic operators are `+`, `-`, `*`, `/`, `(`, and `)`, with order of operations the same as for mathematical expressions. For integers, there is also the modulus operator “`%`”. C does not have an exponential operator, so `x^2` is not the square of `x` but is in fact a bitwise exclusive-or operator.<sup>4</sup> Exponentials are calculated using the `pow` function, discussed later.

Integer arithmetic is not always the same as floating point arithmetic. Addition, subtraction, and multiplication work the same, but division does not. For example:

```
int x = 5;
int y = 2;
int z = x / y; // z will be assigned the value 2
int w = y / x; // w will be assigned the value 0
int r = x % y; // r will be assigned the value 1
```

For integer arithmetic, division truncates the remainder. The `%` operator is modulo arithmetic, so that the following integer arithmetic for positive values of `x` and `y` will be true:

```
x == (x/y)*y + (x%y);
```

When dealing with different numerical data types, conversions are done automatically at the time of assignment. As noted earlier, floating point values are truncated, not rounded, when assigned to integers as a result of conversion. Sometimes it is desirable to force a conversion within an arithmetic expression, and that is done through a type cast. A type cast is specified by putting the type to be cast to in parentheses before the variable or expression. For example:

```
1 int m = 2;
2 int n = 4;
3 double w = 2./4;           // w is assigned 0.5
4 double x = m/n;           // x is assigned 0
5 double y = (double)m/n;    // y is assigned 0.5
6 double z = (double)(m/n);  // z is assigned 0
```

A binary operation between a floating point value and an integer results in promoting the integer to floating point, and the result is a floating point. Thus, the assignment to `w` results in a floating point value. As discussed above, since both `m` and `n` are integers, `m/n` uses integer arithmetic resulting in the value zero for `x`. When the variable `m` is type cast to `double`, there is a binary operation between a floating point and an integer, so the integer is promoted to floating point and `y` is assigned the value `0.5`. The type cast of the expression, which is still between two integers, means that the type cast only succeeds in converting an integer zero to a floating point zero, which is what `z` receives. Type casts are used for nonnumeric types, such as pointers, but that should be done with suitable caution.

---

<sup>4</sup>The bitwise operators are “`&`”, “`|`”, and “`^`” corresponding to “and”, “or”, and “exclusive-or”. Suppose `x` and `y` are integers with values `12 = 1100` (binary) and `10 = 1010` (binary), respectively. Then `x&y = 8 = 1000` (binary), `x|y = 14 = 1110` (binary), and `x^y = 6 = 0110` (binary).

Type casts in C also work in C++, but there are additional type cast forms as well. The more common way to cast an `int` into a `double` is to treat the type like a function; for example, Line 5 can be written instead in either of these ways:

```
5  double y = double(m)/n;
6  double y = static_cast<double>(m)/n;
```

These are fairly straightforward when converting between numerical types. Converting between nonnumerical types is accomplished with a `reinterpret_cast`. This is useful when dealing with pointers to data. For example:

```
char pi[9] = "3.141592";
double* p = reinterpret_cast<double*>(pi); // *p = 9.35009e-67
char* q = reinterpret_cast<char*>(p); // q = "3.141592"
```

As shown by the value pointed at by `p`, the `reinterpret_cast` should be used with suitable caution.

In addition to the “`=`” assignment statement, C provides some shortcuts for other common assignments, namely `+=`, `-=`, `*=`, `/=`, `&&=`, and `||=`. These assignments are a useful shorthand; the equivalent statements are shown below:

```
x += rhs;    ⇔ x = x + (rhs);
x -= rhs;    ⇔ x = x - (rhs);
x *= rhs;    ⇔ x = x * (rhs);
x /= rhs;    ⇔ x = x / (rhs);
x &&= rhs;   ⇔ x = x && (rhs);
x ||= rhs;   ⇔ x = x || (rhs);
```

The boolean operators `&&` and `||` will be discussed in Section 3.3.

There is also a conditional assignment that is written as

```
result = test ? true value : false value;
```

where the right-hand side consists of a boolean value followed by a question mark. If the boolean value is true, then the conditional expression evaluates to the expression between the question mark and the colon. Otherwise, the conditional expression evaluates to the expression following the colon. For example, to compute the maximum value between two variables, it could be written as

```
maxValue = y > z ? y : z;
```

This example would assign to `maxValue` the larger of the two values `y` and `z`.

Another form of shortcut is used for integer arithmetic, the increment and decrement operators. The expression `++i` where `i` is a decimal type, `int` or `long`, means increment the value of `i` by one. The expression `i++` is almost exactly the same. The difference between these has to do with order of operation when used in an assignment statement. The expression `++i` increments the value first and then uses the result in the arithmetic, whereas the expression `i++` increments the value *after* it is used in the arithmetic. The expressions `--i` and `i--` have a similar purpose but for decrementing the value. The following program and output illustrate this point. The increment and decrement operators are extremely useful in loop constructs, as we shall soon see.

**Example 3.2.**

```

1 #include <stdio.h>
2
3 int main(int argc, char* argv[])
4 {
5     // We'll use two integer variables: val and i
6     // val will show the result of the equation, and
7     //      i will be the variable being incremented or decremented
8     int val = 0;
9     int i = 1;
10
11    // Before/after each operation, you can see the resulting values of
12    //      the increment and decrement operators
13    printf("Initial values:\nval = %d, i = %d\n\n", val, i);
14    val = ++i;
15    printf("After val = ++i:\nval = %d, i = %d\n\n", val, i);
16    val = i++;
17    printf("After val = i++:\nval = %d, i = %d\n\n", val, i);
18    val = --i;
19    printf("After val = --i:\nval = %d, i = %d\n\n", val, i);
20    val = i--;
21    printf("After val = i--:\nval = %d, i = %d\n\n", val, i);
22
23
24    return 0;
25 }
```

The output of this program is shown below:

Initial values:  
val = 0, i = 1

After val = ++i:  
val = 2, i = 2

After val = i++:  
val = 2, i = 3

After val = --i:  
val = 2, i = 2

After val = i--:  
val = 2, i = 1

Note how when the operator is applied first, e.g., `++i`, `val` receives the new value, but when the operator is applied second, e.g., `i++`, `val` receives the value *before* the increment.

### 3.3 • Boolean Algebra

The last set of operations is the boolean algebra operations. In C, an expression is false if it evaluates to 0 and is true otherwise. Occasionally, you will come across code that will have something like

```

1 int n;
2 ...
3 if (n) {
4     \\ do something
5 } else {
6     \\ do something else
7 }

```

The first part will be executed if `n` is not zero; otherwise, the other part will be executed. While this is perfectly acceptable, it's not as readable as using the explicit test, so it would be better if Line 3 read “`if (n != 0)`” so that it's explicit that `n` is being checked for whether it is not zero.

The basic test operations are

<code>x == y</code>	true if the values of <code>x</code> and <code>y</code> are equal
<code>x != y</code>	true if the values of <code>x</code> and <code>y</code> are <i>not</i> equal
<code>x &lt; y</code>	true if the value of <code>x</code> is less than <code>y</code>
<code>x &lt;= y</code>	true if the value of <code>x</code> is less than or equal to <code>y</code>
<code>x &gt; y</code>	true if the value of <code>x</code> is greater than <code>y</code>
<code>x &gt;= y</code>	true if the value of <code>x</code> is greater than or equal to <code>y</code>

These operations can be combined by using “and”, “or”, and “not” operations, given by the operators `&&`, `||`, and `!` terms, respectively. Here are some examples and their meaning:

<code>(x &gt; 0) &amp;&amp; (x &lt;= 10)</code>	true if $0 < x \leq 10$
<code>(x &lt;= 0)    (x &gt; 10)</code>	true if $x \leq 0$ or $x > 10$
<code>!(x &lt; 0)</code>	false if $x < 0$

**Important:** When doing scientific computing, which often demands the use of floating point values, it is important to understand the limitations of the “`==`” operator in the context of floating point values. Consider the following very simple program where two floating point values are compared:

### Example 3.3.

```

1 int main(int argc, char* argv[])
2 {
3     double a = 0.15 + 0.15;
4     double b = 0.10 + 0.20;
5
6     if (a == b) {
7         printf("They are equal.\n");
8     } else {
9         printf("They are unequal.\n");
10    }
11
12    return 0;
13 }

```

Amazingly, many systems will report that `a` and `b` are *unequal*! The problem is that the arithmetic is not exact and the value of 0.3 cannot be represented exactly in the computer. How the rounding is done on the last bit of the representation depends upon the arithmetic operation and the values being computed. Therefore it is important to re-

**Table 3.4.** Commonly used mathematical functions. All functions are declared in the header file `math.h` and linked with the math library with the flag `-lm`. Functions listed here are for double precision variables; single precision versions have a trailing `f` and long double versions have a trailing `l`. For example, use `expf(x)` or `expl(x)` for single precision or long double precision, respectively. (a) Be sure to use `fabs` for floating point values, not `abs`, which is for integers. (b) The `pow` function should not be used for simple integral exponents; the function `pow(x, 2)` is much slower than using `x*x`. (c) Returns values in the range  $[-\frac{\pi}{2}, \frac{\pi}{2}]$ . (d) Returns values in the range  $[-\pi, \pi]$  using the signs of `x, y` to determine the quadrant (see Figure 3.1). (e) A variable has the value `NaN` when a calculation is invalid, e.g., `x = 0/0`; (f) A variable has the value `±Inf` when the value is actually or effectively infinite, e.g., `x = exp(1000.)`.

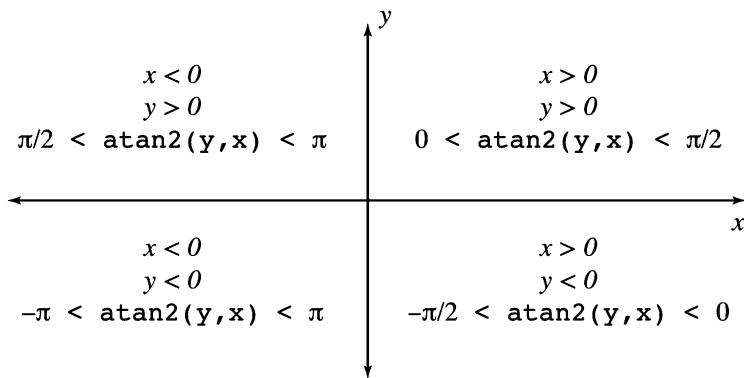
Function	Description	Function	Description
<code>abs(n)<sup>a</sup></code>	$ n $ for int <code>n</code>	<code>fabs(x)<sup>a</sup></code>	$ x $ for double <code>x</code>
<code>fmax(x, y)</code>	$\max\{x, y\}$	<code>fmin(x, y)</code>	$\min\{x, y\}$
<code>exp(x)</code>	$e^x$	<code>log(x)</code>	$\ln x$
<code>exp2(x)</code>	$2^x$	<code>log2(x)</code>	$\log_2 x$
<code>pow(x, y)<sup>b</sup></code>	$x^y$	<code>log10(x)</code>	$\log_{10} x$
<code>sqrt(x)</code>	$\sqrt{x}$	<code>hypot(x, y)</code>	$\sqrt{x^2 + y^2}$
<code>sin(x)</code>	$\sin x$	<code>asin(x)</code>	$\sin^{-1} x$
<code>cos(x)</code>	$\cos x$	<code>acos(x)</code>	$\cos^{-1} x$
<code>tan(x)</code>	$\tan x$	<code>atan(x)<sup>c</sup></code>	$\tan^{-1} x$
		<code>atan2(y, x)<sup>d</sup></code>	$\tan^{-1} \frac{y}{x}$
<code>sinh(x)</code>	$\sinh x = \frac{e^x - e^{-x}}{2}$	<code>asinh(x)</code>	$\sinh^{-1} x$
<code>cosh(x)</code>	$\cosh x = \frac{e^x + e^{-x}}{2}$	<code>acosh(x)</code>	$\cosh^{-1} x$
<code>tanh(x)</code>	$\tanh x = \frac{e^x - e^{-x}}{e^x + e^{-x}}$	<code>atanh(x)</code>	$\tanh^{-1} x$
<code>ceil(x)</code>	$\lceil x \rceil$ (integer ceiling)	<code>floor(x)</code>	$\lfloor x \rfloor$ (integer floor)
<code>trunc(x)</code>	$\lfloor x \rfloor$ if $x \geq 0$ ; $\lceil x \rceil$ if $x < 0$	<code>round(x)</code>	$\lceil x \rceil$ if $x - \lfloor x \rfloor \geq \frac{1}{2}$ ; $\lfloor x \rfloor$ if $x - \lfloor x \rfloor < \frac{1}{2}$
<code>isnan(x)<sup>e</sup></code>	<code>true</code> if <code>x</code> is <code>NaN</code>	<code>isinf(x)<sup>f</sup></code>	<code>true</code> if <code>x</code> is infinite
<code>isfinite(x)</code>	<code>true</code> if <code>x</code> is not <code>NaN</code> nor infinite		

member to *never* trust equality or inequality operations applied to floating point values. This topic will be revisited in Chapter 5.

## 3.4 • Mathematical Functions

In addition to basic arithmetic operations, there are many mathematical functions available to use. These functions are all declared in the header file `math.h`. The functions are defined in the C math library, which can be linked with code by adding the `-lm` option to the linking command. Table 3.4 shows a list of the most commonly used functions.

Note that there is an important difference between the `abs` and the `fabs` functions, the former is for integer values only, while the latter is for floating point types. Thus,



**Figure 3.1.** Illustration of the quadrants as determined by the `atan2(y,x)` function.

the result of the expression `abs(-3.5)` will be 3, not 3.5, because the value will be truncated to an integer before the absolute value is evaluated.

Most of these functions have different versions depending on the precision of the floating point variable. For example, to use the exponential function `exp(x)` it is assumed `x` is a `double`. There are companion functions `expf(y)` and `expl(z)` for when `y` is a `float` and `z` is a `long double`. To avoid losing precision or wasting time unnecessarily, be sure to use the proper version for the variables being used.

The `atan2(y,x)` function is worth a careful look because it uses the sign information in the two arguments to determine the output. Figure 3.1 illustrates how the quadrants and function values are determined.

In C++, the math functions are declared in the header `cmath`. The functions are also overloaded so that the same function names will work for types `float`, `double`, and `long double`. Similarly, the function `abs` is overloaded to handle both integral and floating point values. Because of this, the function `fabs` is redundant, but it is still available for compatibility.

## 3.5 • Arrays and Pointers

The primitive data types can be combined into arrays of data. Obviously, when solving differential equations, this is a frequent requirement in order to approximate functions in one or more spatial and/or temporal dimensions.

Memory for arrays must be allocated before it can be used, and that memory must be contiguous in order for it to work properly. Always bear in mind that memory is allocated linearly, so two- and higher-dimensional arrays will require extra care. Arrays can be either statically allocated or dynamically allocated. Statically allocated memory is very easy to program, but it also can limit the size of the array that can be allocated due to the way that static memory is created by the system. It is not recommended for handling large grids. Dynamically allocated memory requires the use of data types called pointers, so they require a little more work but allow for much larger arrays. In general, statically allocated arrays are best for smaller, fixed-size arrays, while dynamically allocated arrays are best for larger arrays or arrays whose size cannot be determined *a priori*.

A statically defined array is declared by giving the dimensions in square brackets. Below are two examples where a one-dimensional array of length 5 called `data` is

declared, and the two-dimensional array `grid` is declared and initialized with values one through nine.

### Example 3.4.

```

1 int main(int argc, char* argv[])
2 {
3     // data is an array of floats: data[0], ..., data[4]
4     float data[5];
5
6     // grid is a 3x3 array arranged in column-major order
7     double grid[3][3] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
8
9     // Rest of the program...
10 }
```

Arrays in C are zero-based, which means that arrays begin with index zero. For example, in the above code, the float array `data[5]` will contain five elements, indexed by `data[0]` to `data[4]`. The value `data[5]` will not generate an error, but its contents will be unpredictable. This is a common source of errors causing many off-by-one bugs. These sorts of errors are easily detected with the `valgrind` tool discussed in Section 1.6. Those new to C should take care with loops and other constructs that rely on indices until accustomed to this style of indexing.

The order in which memory is arranged can be critical to the ultimate performance achieved. For that reason, it's important to understand how the memory is organized in multi-dimensional arrays. Figure 3.2 illustrates how the memory for the `grid` variable would be organized in contiguous memory. Simply put, incrementing the last index will be the next element in the array. This is called row-major ordering, which is different from column-major ordering used by Fortran where incrementing the first index will be the next element. The equivalent statement in Fortran would give the transpose of the matrix on the left in Figure 3.2. As will be seen in Chapter 7, this can cause conflicts when using libraries written in Fortran. Many commonly used numerical libraries are written in Fortran for historical reasons, so understanding this difference and adjusting accordingly is important.

Statically defined arrays are more intuitive at the outset, but to handle big arrays in scientific computing, they are not the right choice. Therefore, it is important to under-

<code>grid[0][0] = 1</code>	<code>grid[0][1] = 2</code>	<code>grid[0][2] = 3</code>	<code>grid[1][0] = 4</code>	<code>grid[1][1] = 5</code>	<code>grid[1][2] = 6</code>	<code>grid[2][0] = 7</code>	<code>grid[2][1] = 8</code>	<code>grid[2][2] = 9</code>
<code>grid[1][0] = 4</code>	<code>grid[1][1] = 5</code>	<code>grid[1][2] = 6</code>						
<code>grid[2][0] = 7</code>	<code>grid[2][1] = 8</code>	<code>grid[2][2] = 9</code>						

**Figure 3.2.** Layout of memory in an array. On the left is how continuous memory corresponds to entries in a matrix. On the right is how the data is arranged in physical memory. This orientation is called row-major ordering, where the fastest changing index is always the last index, and the slowest changing index is the first index.

stand how to create dynamically allocated arrays as well. These arrays are allocated during runtime and so can have a different size every time they are created. A simple example would be to allow the user to choose the dimensions of their computational domain.

### 3.5.1 • Dynamically Allocated Arrays

Before discussing how to construct dynamic arrays, it is necessary to take a small detour and discuss pointers. They are simple constructs, but they often give beginning programmers fits because of their perceived abstraction. Every point in memory space has an address. When talking about 32-bit or 64-bit systems (most modern computers are now 64-bit systems), it refers to the number of bits used to identify a point in memory space, i.e., its address. A pointer is simply a variable whose value contains a memory address.

It's not sufficient to just point to a location in memory without understanding at what type of data it is pointing. A pointer can be declared in two different ways, but the asterisk is the indicator of a pointer declaration. In the example below, compare the location of the asterisk in declaring the two pointers on Lines 10 and 14. On Line 10, `floatPointer` is declared to be a pointer variable that points to the type `float`. On Line 14, `anotherPointer` is declared to be a variable that has type pointer to `float`. Both locations of the “`*`” are acceptable.

#### Example 3.5.

```

1 int main(int argc, char* argv[])
2 {
3     // data is an array of floats: data[0], ..., data[4]
4     float data[5];
5
6     // grid is a 3x3 array arranged in column primary order
7     double grid[3][3] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
8
9     // pointer to floating point data, uninitialized
10    float *floatPointer;
11
12    // variable of type pointer to float data initialized to point to the
13    // start of the data array.
14    float* anotherPointer = data;
15
16    // pointer to double precision data, it points to the start of the
17    // grid array
18    double *dblPointer = &(grid[0][0]);
19    float* x;
20    float y;
21
22    anotherPointer[2] = 3.;           // this will set data[2] to 3.
23    *anotherPointer = 7;             // this will set data[0] to 7.
24    data[3] = *anotherPointer + 1;   // this will set data[3] to 8.
25
26    // this assigns the memory address for y to the pointer x
27    // so that x points to the value of y.
28    x = &y;
29 }
```

To retrieve the value of the data pointed at by a pointer, the asterisk is again used. The value of the `float` pointed at by the variable `x` is `*x`. For example, examine the

use of `*anotherPointer` on Lines 23, 24. Pointers can point to a single location in memory, but they can also be used to access arrays of data. For example, on Line 14, `anotherPointer` is set to point to the array `data`. It can be treated like an array as on Line 22, which will set `data[2]` to have the value 3.

The memory address of a variable is obtained using the “`&`” operator so that the value of a pointer can be set, for example, `x = &y` from Line 28. The “`&y`” means to take the memory address of the variable `y`, not its value. Thus, `dblPointer` points to the first entry in the memory storing `grid`, and the data in `grid[3][3]` can then be accessed sequentially by using indexing. Thus, the expression `dblPointer[5]` would be a double precision number with the value 6 (remember, C uses 0-based indexing!).

It is unfortunate that dereferencing a pointer and the multiplication operator use the same character, “`*`”, but with judicious use of spaces the meaning can still be made clear enough:

```
int x = 0;
int y[5] = {1, 2, 3, 4, 5};
x = 2 * *y + 1;
```

Here, because `y` is an array, it is also a pointer. The expression `*y` dereferences the pointer, which is equivalent to using `y[0]`. Therefore, in this program, `x` will contain the value of 3 when it is done.

So far, pointers have only been used to access data created statically. To create an array dynamically, an appropriate amount of memory must be created, and then a pointer is used to point to the beginning of that memory. Suppose the `data` array from the above example is to be allocated dynamically; then the equivalent of Line 4 would be

```
float* data = (float*)malloc(5 * sizeof(float));
```

The function `malloc` is used to allocate memory in terms of the number of bytes, in this case five floats. C provides a utility function `sizeof` that returns how many bytes a variable type will occupy. So in this case, `malloc` is asked to allocate enough memory to contain five `floats`. In front of the `malloc` call is the type cast (`float*`) which converts the generic memory pointer of type `void*` returned by `malloc` into an array of type `float`. In this case, type casting is not strictly required, but it is a good habit to use a type cast when mixing different data types in an equation.

After any memory is allocated dynamically, it must also be deallocated or it will result in a memory leak, which is where memory is requested and then not returned to the system. Leaked memory is unusable until the program terminates. For small short programs, memory leaks are unlikely to be a problem, but as programs start squeezing available memory for every ounce of space, careful memory management is essential to avoid crashing. As a rule, **every allocation of memory created with malloc should be matched with a corresponding call to the function free to release it**. Thus, from the example above where the array `data` was created dynamically, it should be matched with a corresponding call to release the memory when it is finished:

```
free(data);
```

In comparing static versus dynamic allocation of memory, it may seem like the two are equivalent, but there is one critical difference that is of importance in the context of scientific computing where often large arrays are desired for various purposes. There is a compiler-dependent limit on the amount of memory that can be allocated statically

and it is not too difficult to exceed that limit for some applications. Errors of this type can be very difficult to identify and remedy without specialized tools or knowledge. To be safe, it is recommended that static allocation be limited to smaller and fixed-size arrays, while dynamic allocation be used for larger arrays such as data on grids and/or when the dimensions of the array may depend upon input values.

### 3.5.2 • Strings

A special kind of array that is useful for communicating text is an array of characters, also known as a string. Strings can also be allocated either statically or dynamically, but there is one extra twist for string variables. Suppose a string is created this way:

```
char name[5] = "Dave"; // equivalent to: = {'D', 'a', 'v', 'e', '\0'};
```

This results in the memory pointed at by `name` containing the values “D”, “a”, “v”, “e”, “\0”. The last character is equivalent to zero. The zero character is essential because it is used by string handling functions like `printf` to know when to stop reading characters. So when saving strings of text, make sure to add at least one more byte for the zero terminator. For example, consider the following code snippet:

```
char name[5] = "Dave";
name[2] = 'n'; // name now contains "Dane"
name[3] = '\0'; // name now contains "Dan"
```

All characters after the “\0” are ignored because they are after the zero terminator.

### 3.5.3 • Higher-Dimensional Arrays

Allocation of single-dimensional arrays is quite straightforward. Higher-dimensional arrays are a bit more problematic. Example 3.6 illustrates how to construct a two-dimensional and also a three-dimensional array and two different ways in which the data can be accessed.

#### Example 3.6.

```
1 #include <stdio.h>
2 #include <stdlib.h> // need this include for the atoi function
3
4 /*
5 int main(int argc, char* argv[])
6
7 The main program is the starting point for a C program.
8 This program displays the size requirements in bytes for each
9 of the basic data types.
10
11 Inputs: argc should be 4
12     argv[1] will contain the number of rows
13     argv[2] will contain the number of columns
14     argv[3] will contain the number of grid layers
15
16 Outputs: Prints the contents of the array
17 */
18
19 int main(int argc, char* argv[])
20 {
21     // We've assumed that args are set up correctly
22     int numberOfRows = atoi(argv[1]);
23 }
```

```

24 // atoi is a function that converts strings into integers
25 int numberOfRows = atoi(argv[2]);
26 int numberofLayers = atoi(argv[3]);
27
28 // malloc "memory allocation" creates the memory for the array
29 int (*array)[numberOfCols] = malloc(sizeof(*array) * numberOfRows);
30
31 // This sets pointer to be the first entry in the doubly indexed array
32 int* pointer = &(array[0][0]);
33
34 printf("Example of double array storage:\n");
35 for (int i=0; i<numberOfRows; ++i) {
36     for (int j=0; j<numberOfCols; ++j) {
37         array[i][j] = 100*i + j;
38         printf("array[%d][%d] = %d\n", i, j, array[i][j]);
39     }
40 }
41
42 for (int i=0; i<numberOfRows*numberOfCols; ++i) {
43     printf("pointer[%d] = %d\n", i, pointer[i]);
44 }
45
46 // Dynamically allocated memory must be freed so it can be used again.
47 free(array);
48
49 printf("\nExample of triple array storage:\n");
50 int (*array3D)[numberOfCols][numberofLayers]
51     = malloc(sizeof(*array3D) * numberOfRows);
52
53 // This sets pointer to be the first entry in the triply indexed array
54 int* pointer3D = &(array3D[0][0][0]);
55
56 for (int i=0; i<numberOfRows; ++i) {
57     for (int j=0; j<numberOfCols; ++j) {
58         for (int k=0; k<numberofLayers; ++k) {
59             array3D[i][j][k] = 10000*i + 100*j + k;
60             printf("array3D[%d][%d][%d] = %d\n", i, j, k, array3D[i][j][k]);
61         }
62     }
63 }
64
65 for (int i=0; i<numberOfRows*numberOfCols*numberofLayers; ++i) {
66     printf("pointer3D[%d] = %d\n", i, pointer3D[i]);
67 }
68
69 // Dynamically allocated memory must be freed so it can be used again.
70 free(array3D);
71
72 return 0;
73 }
```

This is the first example where the input arguments are put to use. If the resulting executable file is called `arrays`, then this program is run by typing the following at the command prompt:

```
$ arrays 3 4 5 <return>
```

As noted earlier, when the `main` function is invoked, the arguments are passed through from the command line. The variable `argv[]` declares that it is an array of strings. The

first entry in the array will be the program name, hence `argv[0]` has value “arrays”. The next three entries contain the three arguments as defined by being separated by spaces. Characters and numbers are not the same, but strings can be converted to integers by using the `atoi` function, which is declared in the header file `stdlib.h`. Therefore, Lines 22–26 take the three numerical arguments and convert them to integers that will be the dimensions of the arrays later on.

On Line 29 a two-dimensional array is allocated dynamically. Explaining the way the compiler interprets this line would take too much time, so we just focus on the result. For a two-dimensional,  $M \times N$  array, the proper construction is

```
float (* data)[N] = (float(*)[N]) malloc(M * sizeof(* data));
```

where `data` is the variable name for the array. The expression `(float(*)[N])` is the proper way to type cast the result from `malloc`, but it is not necessary to do the explicit type cast. This construction is not the most intuitive way to allocate a two-dimensional array, but it is how it is designed in the language. The three-dimensional analogue is located on Lines 50, 51. The values in the array can be accessed using the square bracket notation as shown on Lines 38 and 60.

Line 32 shows how to access the data sequentially using a pointer. The pointer is given the address of the first entry in the array, where `array[0][0]` is the value of the first entry in the array, and the ampersand character returns the location of that value in memory. The three-dimensional analogue is on Line 54.

The dynamically allocated memory is released, or freed, when the memory is no longer required, which is on Lines 47 and 70. Note that there are the same number of calls to `free` as there are to `malloc`. This ensures that a memory leak is not generated somewhere in the program.

This program populates the arrays and shows how the data is arranged in memory by sequentially traversing the allocated memory. The program contains some `for` loops and more complicated `printf` statements than previously seen. Those topics will be covered more thoroughly later.

Memory allocation in C++ is a little different. Instead of using the `malloc` function, the operator `new` is used. Allocation of one- and two-dimensional arrays of length  $M$  and  $M \times N$  is done by the lines

```
double* data1 = new double[M];  
double (*data2)[N] = new double[M][N];
```

## 3.6 • Structures

Data types can be combined into larger data structures. Combining multiple data types into a single variable helps to keep data organized in a form that is manageable. In C++ programs, this is called encapsulation, and it is a concept that is worth utilizing even when programming in C. As an example, suppose a grid is to be allocated dynamically; then it may be helpful to keep additional information about the grid with it, such as the dimensions of the grid, the grid size increments, and the values of the grid. This can be accomplished with a struct. A struct is a composite variable type, so treat it as a type like `int` or `float`. For example, a two-dimensional grid organized into a struct could be set this way:

**Example 3.7.**

```
struct myGridType {
    int dim[2];
    double del[2];
    double x0, y0;
    double *dataPtr;
} grid;
```

In this example, `grid` is a variable of type `struct` with several fixed-size data types. Note that it is not possible to do dynamic memory allocation for multidimensional arrays inside of a struct, so a simple pointer is used for keeping track of the memory. Here, the dimensions of the data are kept in the `dim` variable, the grid spacing in `del`, and the location of the lower left corner of the domain in `x0` and `y0`. It need not be organized this way—it's just one example of how the data can be grouped together.

To access the entries in a struct, use the “`.`” notation. For example, to set the dimensions of the array use

```
grid.dim[0] = 20;
grid.dim[1] = 40;
```

The variable `grid` is of type `struct myGridType`, and to access an entry within the struct, add a period followed by the variable name inside the struct. Though it probably won't come up often in the context of scientific computing, structs can be nested, and if so, then the dot would be used to traverse down through the hierarchy of structs.

Pointers to structs use a specialized arrow notation “`->`” for accessing the entries in the struct. Using the same example struct as above,

```
struct myGridType *gridptr;
grid->dim[0] = 20;
grid->dim[1] = 40;
```

In this example, the struct was given the name `myGridType`. Naming a struct is optional but is sometimes helpful to keep different structs distinct. If defining many variables with the same struct, then it could also become a type using the `typedef` command:

**Example 3.8.**

```
1 typedef struct {
2     int dim[2];
3     double del[2];
4     double x0, y0;
5     double *dataPtr;
6 } myGridType;
7
8 myGridType grid1;
9 myGridType grid2;
```

This code has the same result as that listed above for `grid`, but now multiple grids can easily be constructed using the same definition. The component entries for each of the two variables `grid1` and `grid2` are separate and distinct.

C++ programmers will recognize `struct` as a very simplified version of a public `class` construct in object-oriented programming. Anything more than the very simplest structures is better handled using classes instead.

## Exercises

- 3.1. Suppose  $\{x_i\}$  is a set of random numbers that are all in the range  $\frac{1}{2} \leq x_i \leq \frac{3}{2}$ . Let

$$T_n = \sum_{i=0}^n x_i.$$

If  $T_n$  is declared as a `float`, estimate how large  $n$  must be in order for  $T_{n+k} = T_n$  for all  $k \geq 0$ . What if  $T_n$  is declared as a `double`? `long double`?

- 3.2. Suppose the variable `A` is defined as

```
double A[n][n];
```

How large can `n` be if the amount of memory must be confined to 1 megabyte? What if `A` is defined as below?

```
float A[n][n][n];
```

- 3.3. Suppose the variable `double z;` is declared on Line 21 in Example 3.5. For each case below, suppose the given code is also inserted on Line 25 in Example 3.5. Give the value in the variable `z` at the end of the program.

1. 25 z = grid[1][2];

2. 25 z = dblPointer[4];

3. 25 z = \*dblPointer - 1;

4. 25 z = grid[2][-1];

5. 25 z = grid[(int)(grid[0][1])][(int)(grid[0][0])];

6. 25 // this is tricky  
26 z = (double)(\*(anotherPointer + 2));

- 3.4. Write a program that takes five input arguments on the command line in order of types `int`, `double`, `long int`, `char`, and `char*`. Use the function `atoi` to convert `argv[1]` to convert the string into an `int`. Similarly, the function `atof` converts a string into a `double`, and `atol` converts a string into a `long int`. Use the function `strcpy(dest, argv[5])` to copy the string `argv[5]` to the variable `dest` which is of type `char dest[256]`.

- 3.5. Create a struct for storing complex values in double precision. Modify Example 3.6 so that it allocates complex values and initializes the two-dimensional array `array[i][j]` with the complex values

$$\cos(2\pi i/\text{numberOfRows}) + I \sin(2\pi j/\text{numberOfCols}).$$

Initialize the three-dimensional array with

$$k \cos(2\pi i/\text{numberOfRows}) + Ik \sin(2\pi j/\text{numberOfCols}).$$

Note that the real and complex parts will have to be assigned separately.

- 3.6. Write a program that allocates an  $M \times N$  grid where  $M, N$  are given as arguments in the command line. The program should also create a struct as in Example 3.8 and populate its entries where `dim` should contain the dimensions of the array, and `del`, `x0`, `y0` are also given on the command line. Use the function `atof(string)` to convert input strings into double precision floating point values.
- 3.7. Write a program that takes two integers as input on the command line (use the function `atoi` to convert `argv[1]` and `argv[2]` into integers) and prints out the largest value and the sum.
- 3.8. Write an assignment statement that will compute the expressions below:

1.  $\frac{x}{(1-x)^2}$
2.  $\frac{1}{1+x} + \frac{1}{1-x}$
3. 
$$\begin{cases} (1+x)^2 & x \geq 0 \\ (1-x)^2 & x < 0 \end{cases}$$
4.  $x_i + x_{i+1}$

- 3.9. Determine the range of values for the floating point variable `x` or the integer `n` for which the following expressions will evaluate to true:

1. `n > -10`
2. `(n < 3.5 ? n : 10-n) >= 3`
3. `!(x > 0.5) || !(x < 6)`
4. `(x > 4.5)*(x < 6.5) == 0`



## Chapter 4

# Input and Output

There's no point in doing lots of sophisticated calculations if the results aren't printed or saved. There are essentially two ways in which data can be communicated to/from the program. For user interactivity or printing diagnostic information, terminal I/O is the easiest. However, for larger sets of data, it is critical to understand how to read and write from/to files stored on a disk. In this chapter, we'll explore how user input can be provided through the terminal and results can be saved through data storage.

### 4.1 • Terminal I/O

The primary terminal I/O functions are `scanf` for input and `printf` for output. We have already seen some printing statements, so it's time to discuss how the `printf` function works in more detail. The `printf` function is an example of a function that can use a variable number of arguments:

```
printf( formatString , var1 , var2 , ... );
```

The first argument of the `printf` function is the format string. The format string is a specification of the text that will be printed. In its simplest form, the format string may only contain plain text, as was shown in Example 2.1. The extra “`\n`” that appears in that example illustrates the use of an escape sequence for a special character. In this case, “`\n`” represents a new line. Table 4.1 shows the different escape codes that can be used.

In addition to the escape codes, there are also key codes for reading and writing variables. Key codes are designated by an initial “%” character. The choices for key codes are shown in Table 4.2. For example, suppose there are two double precision floating point variables `x`, `y` to be printed out as a matrix on their own line of text; then the format string might look like

```
printf( "[%lf , %lf ]\n" , x , y );
```

Note that the number of key codes for data must match the number of variables listed as arguments. In this case, there are two key codes “`%lf`”, and there are two variables listed, `x`, `y`. Note also that not every line must end with “`\n`”, though it is very common. If the format string does not contain an “`\n`”, then the next `printf` will continue printing *on the same line*. This is handy when printing a sequence of numbers on the

**Table 4.1.** Table of escape sequences for the `scanf` and `printf` format string. **Bold** rows are the most commonly used.

Escape Code	Meaning
\a	audible bell
\b	backspace
\f	form feed
<b>\n</b>	<b>new line</b>
\r	carriage return
<b>\t</b>	<b>tab</b>
\v	vertical tab
\\"	<b>double quote</b>
\'	<b>single quote</b>
\?	<b>question mark</b>
\\\	<b>backslash</b>
<b>% %</b>	<b>percent sign</b>

**Table 4.2.** Table of basic key codes for reading and writing variables. A complete list of variable types can be found in the header file `stdint.h`.

Key	Variable Type	Description
d	int	decimal number
o	int	octal number
x	int	hexadecimal number
u	unsigned int	unsigned decimal number
ld	long int	long decimal number
lo	long int	long octal number
lx	long int	long hexadecimal number (useful for printing memory addresses)
lu	unsigned long int	unsigned long decimal number
lld	long long int	long long decimal number
f	float	floating point
g	float	floating point
e	float	floating point in scientific notation
lf	double	double precision floating point
lg	double	double precision floating point
le	double	double precision floating point in scientific notation
Lf	long double	quadruple precision floating point
Lg	long double	quadruple precision floating point
Le	long double	quadruple precision floating point in scientific notation
c	char	single character
s	char*	<code>printf</code> : character string, prints until the first “\0” <code>scanf</code> : character string, reads until the first white space or new line

**Table 4.3.** Examples of `printf` format modifiers. Blank spaces are indicated by underscore characters.

Format	Result
<code>printf(":%d:", 123);</code>	<code>:123:</code>
<code>printf(":%6d:", 123);</code>	<code>:____123:</code>
<code>printf(":%06d:", 123);</code>	<code>:000123:</code>
<code>printf(":%.3f:", 12.45);</code>	<code>:12.450:</code>
<code>printf(":%5.1f:", 12.45);</code>	<code>:_12.5:</code>

same line and it's more convenient to use a loop (to be discussed later). The typing can be sent to the next line by an extra `printf("\n")`; when done. As a useful exercise, go through the sample code in this book and look for `printf` statements; see if you understand what the output will be.

All but the last of the escape codes in Table 4.1 are valid in C++ output as well. The `%%` escape code is unnecessary because the format string construct is not used in C++.

The key codes can be further modified with number format modifiers. These are very helpful when trying to format text into columns or for other special considerations. An integer inserted between the “`%`” and the key code means the width of the field to be used. For floating point numbers, the integer followed by a period and another number indicates the field width and the decimal precision. Last, a leading “`0`” indicates that the field should be padded with zeros. Table 4.3 has examples of these modifiers.

In C++, terminal I/O is handled by `std::iostream` class and its subclasses. To print to the terminal, use the object `std::cout`, and use the “`<<`” operator to print text to the screen. For example, the line

```
printf("[%lf, %lf]\n", x, y);
```

would be replaced with

```
std::cout << "[" << x << ", " << y << "]" << std::endl;
```

There are numerous formatting functions that can be inserted into the output stream that can be found in the `iomanip` header file. It is beyond the scope of this text to detail all of them and their equivalents in C, but to produce almost the same output as in Table 4.3, the following code will work:

```
1 std::cout << ":" << 123 << ":" << std::endl;
2 std::cout << ":" << std::setw(6) << 123 << ":" << std::endl;
3 std::cout << ":" << std::setfill('0') << std::setw(6) << 123
4     << ":" << std::setfill(' ') << std::endl;
5 std::cout << ":" << std::setprecision(3)
6     << std::setiosflags(std::ios_base::fixed) << 12.45
7     << ":" << std::setiosflags(std::ios_base::basefield)
8     << std::endl;
9 std::cout << ":" << std::setw(5) << std::setprecision(1)
10    << 12.45 << ":" << std::endl;
```

Here the `std::setw` stream manipulator sets the width of the output field for the next object printed. The default fill character is a space, but it can be changed to another character using the `std::setfill` manipulator. Note that the new fill character remains in effect until the next time `std::setfill` is used, so this time the fill character is changed back to being a space right after. To set the number of digits to be displayed, use the `std::setprecision` manipulator. By itself it specifies the number of significant digits to be displayed, but when combined with setting the fixed point flag via `std::setiosflags(std::ios::fixed)`, the precision sets the number of digits after the decimal point to be displayed. Like `std::setfill`, the flag `std::ios::fixed` remains in effect until it is changed back to the default `std::ios::basefield`. The last line is the one that produces a slightly different output, where the value is truncated rather than rounded, causing the value 12.4 to be displayed instead. One other flag worth mentioning is the `std::ios::scientific` flag that will change the output to scientific notation.

The `scanf` function is a companion function to `printf` but is for reading user input through the terminal window. The format string describes what text to expect from the user and what variable types are in the argument list. The format string in this case only corresponds to user input, not to providing a prompt for the user. For example, to prompt a user to enter an integer, **do not** write `scanf("What is your number? %d", &num)`, because it will expect the user to literally type “What is your number?” before it scans for a number. Instead, use a combination of `printf` and `scanf`:

```
1 printf("What is your number? ");
2 scanf("%d", &num);
```

Another distinction between `scanf` and `printf` is that the extra variables for `scanf` after the format string *must be memory addresses, not values*. This is why in the example above there is an “`&`” in front of `num`. The memory address provided is where `scanf` will store the value entered by the user. To see the distinction, consider the snippet below, where the user input is echoed back to the user:

```
1 int num;
2 // prompt the user for input
3 printf("Pick any number from 0 to 100: ");
4 scanf("%d", &num); // get the user response
5 // echo the response back to the user
6 printf("Was your number %d? Amazing!\n", num);
```

Whereas `printf` can do variable type conversion before printing, though it’s not recommended and sometimes produces warnings from the compiler, `scanf` requires the variable type to match the key code in the format string. For example, if the format string above had been `%ld`, then it would have expected to store a long integer, but `num` is a regular integer and hence the store would overwrite extra bytes. Compilers will generally catch this, but not always, so be sure to match variable type with the proper key code.

In C++, terminal input is read through the object `std::cin` using the operator “`>>`”. The operator is overloaded so that the input data will be matched to the variable type put in the stream. Thus, if `x` is a `double`, and `n` is an `int`, then

```
std::cin >> x >> n;
```

will read a `double` and an `int` in order, the compiler making the adjustment ac-

cording to the declared type of the variable. The equivalent to the above user input with prompt would be

```

1 int num;
2 // prompt the user for input
3 std::cout << "Pick any number from 0 to 100: ";
4 std::cin >> num; // get the user response
5 // echo the response back to the user
6 std::cout << "Was your number " << num << "?" << std::endl;
```

## 4.2 • File I/O

For high performance computing, it is safe to say that the amount of input variables will be more than one or two, and the amount of output will be far more than a user will want to read on the screen. Therefore, it is important to learn how to read and write files as well. There are two types of file access that are useful in this context: formatted text and unformatted raw data.

To begin, let's start with a simple example program that we can take apart.

### Example 4.1.

```

1 #include <stdio.h>
2 #include <math.h>
3
4 /*
5 int main(int argc, char* argv[])
6
7 The program opens a new file with the given name, writes some data to
8 it, then reads those numbers back in and prints them to the screen.
9
10 Inputs: argc should be 2,
11 argv[1] will contain the filename to be created
12
13 Outputs: creates the file and writes some data to it.
14 */
15
16 int main(int argc, char* argv[])
17 {
18     // First thing is to create the file we will use for storage
19     // the "w" indicates we are opening the file for writing
20     FILE *fileid = fopen(argv[1], "w");
21
22     // Once the file is open, we can write to it with fprintf functions
23     // that act the same as the printf commands we learned earlier
24     // Note the use of M_PI. This is a useful macro that defines the
25     // number pi and is found in the header file math.h
26     fprintf(fileid, "%d %.3f %s", 17, M_PI, "Hello World!\n");
27
28     // When we're done writing, we must close the file so that it is
29     // terminated properly
30     fclose(fileid);
31
32     // To read the data back in, we must define the variables
33     int num;
34     double pi;
35     char greeting[256];
36
37     // The file is closed, so let's open it again, but this time
38     // use "r" to indicate that we will read the file
```

```

39 // Note that we don't have to redeclare fileid because
40 // we did it already
41 fileid = fopen(argv[1], "r");
42
43 // Now we use fscanf the same way we learned scanf
44 // Note that greeting does not have an ampersand because greeting is
45 // an array, so greeting has type char* already
46 fscanf(fileid, "%d %lf %s", &num, &pi, greeting);
47
48 // All done reading, so close the file
49 fclose(fileid);
50
51 // Now print the results. Did we get what we expected?
52 printf("%d %lf %s\n", num, pi, greeting);
53 return 0;
54 }

```

When writing or reading a file, there is a rhythm to it, namely, open, read or write, close. That rhythm is on display in this code. For the initial write, the file is opened for writing on Line 20, the data is written on Line 26, and the file is closed on Line 30. Then when reading the file in, it is opened for reading on Line 41, the data is read on Line 46, and then the file is closed on Line 49. Keeping that rhythm in mind will help to keep all the file I/O contained and correct.

When a file is opened using `fopen`, there are multiple choices for the mode of opening, the most common being "`w`" for writing, "`a`" for appending, and "`r`" for reading. The distinction between the first two modes is sometimes confusing for novices. When a file is opened for writing, any existing file with the same name is removed, and a new empty file is created in its place ready for writing. By comparison, when a file is opened for appending, the existing file is left intact and the next data written to the file will be added at the end. If the file doesn't exist, then opening a file for appending is the same as opening for writing. The distinction between these is important because both have their place in scientific computing, and misuse can lead to either lost data or unintentionally excessive data storage.

The `fprintf` and `fscanf` functions are very similar to the terminal I/O counterparts except there's an additional argument in the front that corresponds to the pointer to the file buffer, a variable of type `FILE*`. In fact, `printf` and `scanf` are just specializations where the file buffer is the predefined terminal input and output buffers commonly called `stdin` and `stdout`.

The output of the program is shown below:

```
17 3.142000 Hello
```

Is this what you expected? The first number seems fine but  $\pi$  cut off at three decimal places and the text was also cut off. To figure out what happened, look at the file that was saved:

```
17 3.142 Hello World!
```

When the value of  $\pi$  was saved the format statement specified only three decimal places, so only the first four digits of  $\pi$  were saved to the file. When the data was read back in from the file, that was all the accuracy available in the file. This highlights an important point to remember when storing numerical data, particularly floating point values.

**Saving numbers in text format is horribly inefficient and there is risk of losing accuracy.** This will be remedied shortly.

For the string, it appears that the whole string was saved, so it was written properly, but it didn't get read back in as expected. In this case, when `scanf` or `fscanf` reads a string, it will stop at the first white space, i.e., space character, tab, carriage return, or new line. This too can be remedied, but it requires a different way of reading the data.

The type of file I/O shown above is really best suited for reading files of parameters written by a user. For example, suppose a user-supplied parameter file gives the input values for a program to run, such as

```
Parameters for run on January 18, 2019
x dimension: 50
y dimension: 100
```

A program could read this data in, look for the key parameters defined, extract the two values needed, and then use the results to control the program. The advantages of this approach are that (a) a record is kept of the input arguments provided to the program in readable form to match with the output, and (b) the input can be done offline. The latter advantage is rather important because when a program is run on a large cluster, it will have to be scheduled and won't be able to ask for interactive user input. Finally, the more flexible a program is for reading the user input file, the easier and more reliably it will execute when its turn on the cluster finally arrives. There's nothing more disappointing than queuing up a large run only to return a day later and find it was unable to read or incorrectly read the input data file.

Returning to the faulty data storage problem, a different strategy is needed to avoid loss of accuracy. Consider the following corrected program:

### Example 4.2.

```
1 #include <stdio.h>
2 #include <math.h>
3
4 /*
5 int main(int argc, char* argv[])
6
7 The program opens a new file with the given name, writes data to it,
8 then reads those numbers back in and prints them to the screen.
9
10 Inputs: argc should be 2
11      argv[1] will contain the filename to be created
12
13 Outputs: creates the file and writes some data to it.
14 */
15
16 int main(int argc, char* argv[])
17 {
18     // Start by creating the data to be stored
19     int num = 17;
20     double pi = M_PI;
21     char greeting[256] = "Hello World!";
22
23     // First thing is to create the file we will use for storage
24     // the "w" indicates we are opening the file for writing
25     FILE *fileid = fopen(argv[1], "w");
26
27     // Once the file is open, we can write to it.
```

```

28 // fwrite is for writing the actual contents of memory.
29 // Each write has to have a single address of memory,
30 // so we'll do each variable separately.
31 // First let's write a single integer.
32 fwrite( &num, sizeof(int), 1, fileid);
33
34 // Next we write the double precision value of pi
35 fwrite( &pi, sizeof(double), 1, fileid);
36
37 // Finally, let's write the whole string.
38 // Note that sizeof(greeting) will return 256
39 fwrite( greeting, sizeof(char), sizeof(greeting), fileid);
40
41 // When we're done writing, we must close the file so that it is
42 // terminated properly
43 fclose(fileid);
44
45 // The file is closed, so let's open it again, but this time use "r"
46 // to indicate that we will read the file
47 fileid = fopen(argv[1], "r");
48
49 // Now we use fread in the same order as we wrote the data
50 // to get it back.
51 // Let's read the data into new variables so it's clear it's
52 // not left over from above.
53 int read_num;
54 double read_pi;
55 char read_greeting[256];
56
57 // Read the number
58 fread( &read_num, sizeof(int), 1, fileid);
59
60 // Read pi
61 fread( &read_pi, sizeof(double), 1, fileid);
62
63 // Read the string
64 fread( read_greeting, sizeof(char), sizeof(read_greeting), fileid);
65
66 // All done reading, so close the file
67 fclose(fileid);
68
69 // Now print the results. Did we get what we expected?
70 printf("%d %lf %s\n", read_num, read_pi, read_greeting);
71 return 0;
72 }

```

The primary difference between the two examples is the actual read/write statements. Take a look at Line 32 where the value of `num` is stored. The first argument for the function is a pointer to the memory that should be written to the file, hence it is given `&num`. The next two arguments will be multiplied together to get the total amount of memory to be written. The `sizeof` function returns how big the data type being stored is, and the following argument says how many of those type will be stored. In this case, it's a total of four bytes because that's the amount of storage required for a single `int`. If storing an array of length  $N$  of integers, the “1” on Line 32 would be replaced with “ $N$ ”. Line 35 is analogous, but this time for a `double`. Line 39 stores the string in `greeting`, where `sizeof(greeting)` returns 256 because that was the number of characters declared on Line 21. It's safer to store the entire string rather than trying to trim it according to its length.

The data is read back in the same order as written using the complementary function `fread`, which has the same arguments as `fwrite`. Those calls are on Lines 58, 61, and 64. This time, when the results are printed, the results are better:

17 3.141593 Hello World!

Direct reading and writing to files in C++ are done through the file stream classes `std::ifstream` and `std::ofstream`. The former is for reading from a file, i.e., an input file stream, and the latter is for writing to a file, i.e., an output file stream. The equivalent of Lines 25–67 in Example 4.2 is shown below:

```
25 std :: ofstream  out(argv [1]);
26
27 out . write( reinterpret_cast <char*>(&num) , sizeof(int));
28 out . write( reinterpret_cast <char*>(&pi) , sizeof(double));
29 out . write( greeting , 256* sizeof(char));
30
31 out . close ();
32
33 std :: ifstream  in(argv [1]);
34
35 int  read_num;
36 double  read_pi;
37 char  read_greeting [256];
38
39 in . read( reinterpret_cast <char*>(&read_num) , sizeof(int));
40 in . read( reinterpret_cast <char*>(&read_pi) , sizeof(double));
41 in . read( read_greeting , 256* sizeof(char));
42
43 in . close ();
```

Most of this should be self-explanatory, with the exception of the type casting operator `reinterpret_cast<char*>`. This tells the compiler to interpret the next pointer as a `char*` instead of the actual type of `int*`. Since the function `std::ofstream::write` expects a character string as the first argument, this enables the conversion of the memory into a straight character stream. The second argument is the number of characters to write, which is easily calculated with the `sizeof` function. Reading the data back in from the file is essentially the same.

---

## Exercises

- 4.1. Write a program to read a  $4 \times 4$  array of double precision numbers from the terminal using `scanf` and then print the array so that the columns align on the decimal point using floating point notation. Repeat the exercise for scientific notation. Be sure to use a range of values with several different orders of magnitude to make sure it is correct.
- 4.2. Write two programs. The first reads a  $4 \times 4$  array of floating point numbers and the name of a file as a string from the terminal and then saves the data to a file with that filename using `fwrite`. The second program should take as input a filename and then read the data from the file with that filename using `fread` and print the results on the terminal.

- 4.3. Write a program that creates this struct:

```
struct inputarg {  
    char name[20];  
    char value[50];  
}
```

The program should read a text file, the name specified in `argv[1]`, that takes five input arguments separated by tabs like the one below:

```
numberOfRows      10  
numberOfCols      5  
outputFileName   foo.dat  
studentName      John Smith  
todaysDate       4/1/2018
```

The first column should be read into the `name` part of the struct and the second column should be read into the `value` part.

- 4.4. Write a program that reads one `long int` from the file called “`/dev/urandom`” and print the answer to the screen. Run the program more than once to verify that the results are different every time.
- 4.5. It is often useful to initialize an array with a fixed initial value such as zero. One quick way to initialize an array with zero is to use the function `memset`, which is defined in the header file `string.h`. To set an integer array `x` of length `N` to all zeros, use

```
int* x = (int*)malloc(N*sizeof(int));  
memset(x, 0, N*sizeof(int));
```

Try writing a program that takes `N` as input and initializes  $x[0], \dots, x[N/2-1]$  with the value zero, and  $x[N/2], \dots, x[N-1]$  with the value one, and then print the values of  $x[0], \dots, x[N-1]$ . Do you get what you expect? If not, why? Hint: Compare with the value of the expression  $1 + 2^8 + 2^{16} + 2^{24}$  and remember that characters have 8 bits.

# Chapter 5

# Flow Control

My thesis advisor once said, “if you can write a `for` loop, then you can do scientific computing.” It is true that a lion’s share of scientific computing involves working with large arrays of data, and that in turn requires using `for` loops, but I would argue that `if` statements are also quite important. But with those two, you’re ready for scientific computing. In this chapter we’ll explore the details of loop structures such as `for` and `while` loops and also explore branching structures such as the `if-then-else` and `switch` statements.

## 5.1 • for Loops

The basic loop structure is quite simple. For illustration purposes, suppose an array is to be created with the values  $1^2, 2^2, 3^2, \dots, 100^2$ . This could be accomplished with

```
int array [100];
for (int n=0; n<100; ++n)
    array [n] = (n+1)*(n+1);
```

Remember, **C arrays begin with 0!** Let’s dissect Line 2 to understand how the `for` loop works. A `for` loop is constructed with the keyword “`for`” followed by an expression in parentheses with three optional arguments separated by required semicolons. The first argument is the initializer and is executed once before the loop begins. In this example, an `int` variable `n` is declared and initialized to 0. In this construction, `n` will be a valid variable *only inside the loop*. If `n` were declared outside the loop, then it could be initialized without the extra declaration. This concept is illustrated by the following program and output:

### Example 5.1.

```
1 #include <stdio.h>
2
3 /*
4 int main(int argc, char* argv[])
5
6 Program illustrates the concept of the scope of the incrementing
7     variable in the loop
8
9 Inputs: none
```

```

10
11 Outputs: shows the values of the incrementing variables for
12 different loop constructions
13 */
14
15 int main(int argc, char* argv[])
16 {
17 // In this loop, i is an undefined variable outside the loop itself
18 for (int i=0; i<5; ++i)
19 printf("i = %d\n", i);
20 printf("\n");
21
22 // Here, j is declared outside the loop. j is again declared
23 // as an int inside the loop
24 // The two j variables are different, and the one declared
25 // outside the loop remains unchanged.
26 int j = 0;
27 for (int j=0; j<5; ++j)
28 printf("j = %d\n", j);
29 printf("j = %d\n", j);
30 printf("\n");
31
32 // Here, the outside loop j is used because it wasn't redeclared
33 // inside the for loop
34 // This time, when the loop terminates the value of j is left with
35 // the value 5 that caused the loop to terminate.
36 for (j=0; j<5; ++j)
37 printf("j = %d\n", j);
38 printf("j = %d\n", j);
39 return 0;
40 }

```

The output of this program is shown below:

```

i = 0
i = 1
i = 2
i = 3
i = 4

j = 0
j = 1
j = 2
j = 3
j = 4
j = 0

j = 0
j = 1
j = 2
j = 3
j = 4
j = 5

```

In the first loop, the variable **i** is defined only inside the loop. If it were used outside the loop, a compiler error would be generated. In the second loop, the variable **j** is used

to control the loop but is declared again in the initializer of the **for** loop. In this case, the two *j* variables are different, and the outside *j* is unchanged by the loop. In the third loop, the outside *j* variable is used and initialized. Because the outside *j* is used, its value is modified as a result of the loop. The construct used for the second loop is very bad form and should not be used. The first loop is ideal, and the last is acceptable. Again, it just depends on the purpose of the loop within the algorithm.

The second argument of the **for** loop is the termination condition. The loop will continue so long as the condition in the second argument evaluates to true. Thus, in the example above, the loop terminates and does not execute its contents when the value of the increment variable reaches 5. The termination condition does not necessarily have to depend on the loop variable—it just has to evaluate to true or false.

The third argument is the increment argument, and the statements there are executed after each pass through the loop and before the termination condition is evaluated.

Finally, it should be noted that a **for** loop will usually contain many more than a single line. When multiple lines of code are desired inside the loop, the curly bracket notation is used.

This example shows how multiple variables can be initialized, separated by a comma in the initializer part, and multiple variables can be updated in the increment part:

```

1 // In this loop, there are two variables being updated
2 for (int i=0, j=10; j>=i; ++i, --j) {
3     printf("i = %d, j = %d\n", i, j);
4 }
5 printf("\n");
6
7 // In this loop, i is incremented by 2, while j is incremented by 1
8 for (int i=0, j=10; j>=i; i+=2, ++j) {
9     printf("i = %d, j = %d\n", i, j);
10 }
11 printf("\n");

```

Examine the program and output in the above example to ensure a clear understanding of how the loop structure works.

As a final example, suppose a loop is needed that will take fixed steps over a time interval, e.g.,  $0 \leq t < 1$  in increments of  $\Delta t = \frac{1}{30}$ . The code below illustrates some choices to print the values  $\frac{0}{30}, \frac{1}{30}, \dots, \frac{29}{30}$ :

```

1 double dt = 1./30.;
2 // using a float to increment and stopping with !=
3 for (double t=0.; t != 1.; t += dt) {
4     printf("t = %f\n", t);
5 }
6
7 // using a float to increment and stopping with <
8 for (double t=0.; t < 1.; t += dt) {
9     printf("t = %f\n", t);
10 }
11
12 // using integers (best solution)
13 for (int i=0; i < 30; ++i) {
14     t = i*dt;
15     printf("t = %f\n", t);
16 }

```

The loop on Line 3 is a poor choice because, as discussed in Section 3.3, the boolean operators “`==`” and “`!=`” are unreliable when used with floating point variable types.

In fact, it is very likely that the loop will continue indefinitely because the expression `t != 1.` will always be true. The loop on Line 8 is better but still not ideal. It is still possible that due to round-off errors the value for `t` in the last iteration will be 0.999999999999, which is still less than 1 even though that is not what was intended. The loop on Line 13 is the best solution because integer arithmetic is exact and using `t = i*dt` will not result in accumulated round-off errors by repeated addition of `dt`. In this case, the loop is guaranteed to terminate when intended and the value of `t` will be as accurate as possible.

## 5.2 • while and do-while Loops

There are two other loop structures that can be used. The `while` loop is equivalent to a `for` loop where the initializer and increment parts are empty. In other words, the following two loops are equivalent:

```
for ( ; test; ) {
    ...
}

while (test) {
    ...
}
```

`while` loops arise, for example, when doing iterative methods where the loop should continue while a residual is larger than a specified error tolerance. Here is how Newton's method could be programmed for solving  $f(x) = 0$ , where  $f'(x)$  is given by the function `df(x)`:

```
1 double x = 0.;
2 double residual = 100.;
3 while (fabs(residual) > 1.0e-8) {
4     residual = -f(x)/df(x);
5     x += residual;
6 }
```

The `do-while` loop is similar to the `while` loop except the contents of the loop are guaranteed to be executed at least once. The Newton's method with that structure would look like this:

```
1 double x = 0.;
2 double residual;
3 do {
4     residual = -f(x)/df(x);
5     x += residual;
6 } while (fabs(residual) > 1.0e-8);
```

## 5.3 • if–then–else

Another essential flow control construct is the branching construct. There are two different branching constructs, the `if–then–else` construct and the `switch–case` construct. The former handles situations when there is a choice between two alternate paths, and the latter handles situations when there are more than two alternate paths.

The most common branch is the `if–then–else` construct. Suppose `test` is a boolean variable; then the `if–then–else` construct looks like this:

```

if (test) {
    // statements to execute if test is true
} else {
    // statements to execute if test is false
}

```

The **else** clause is optional if nothing is to be done when *test* is false. As an example, a common thing to do in a main program is to check whether the arguments provided are valid. Suppose a main program expects one argument; then it should expect *argc* to be two (remember the name of the program itself counts as one of the arguments). A robust program would put a check for the number of arguments given at the beginning of the program, and then print an error message or reminder of the proper way to call the function if it's incorrect as shown here:

```

1 int main(int argc, char* argv[])
2 {
3     // Check for the correct number of arguments,
4     // this program expects a call of the form:
5     // myprog inputfile
6     if (argc != 2) {
7
8         // argc has got the wrong value, so either we received
9         // too few or too many arguments,
10        // so print a reminder of how to do it
11        printf("Usage: myprog inputfile\n");
12        return 1; // reply that the program ended incorrectly
13
14    } else {
15
16        // argc is 2, so we got one argument
17        FILE* fileid = fopen(argv[1], "r");
18        ...
19    }
20
21    return 0;
22 }

```

Recall that true/false evaluation is based on the value of the expression. The value of the expression can be stored in a variable at the same time. This permits the condensing of multiple lines. For example, consider this code:

```

1 FILE* fileid = NULL;
2 float x[100];
3 if (fileid = fopen("myfile.dat", "r")) {
4     // fileid != 0, so it must be valid,
5     // i.e. the file exists and could be opened
6     fread(x, sizeof(float), 100, fileid);
7 } else {
8     // fileid == 0 means the fopen command failed
9     for (int i=0; i<100; ++i)
10        x[i] = 0.0;
11    }

```

Note that the value of *fileid* is assigned inside the test condition for the **if** statement. The **if** statement will evaluate to true if *fileid* is not zero, i.e., it has a valid pointer to a FILE structure. If *fopen* fails to open the file, maybe because the file doesn't exist, then it returns NULL, which evaluates to zero, and therefore if *fileid* is treated as a boolean, its value is zero, or false. So this **if** statement says that if the file is successfully opened, go ahead and read from it; otherwise handle the case that the file

doesn't exist. This is perfectly acceptable. However, this freedom also comes with a warning: **always be sure to distinguish between an assignment that uses one equal sign, with testing for equality that uses two equal signs.** It is very easy to mistakenly put = in place of == or vice versa, and the results can be quite different.

A second, more advanced comment is about the meaning of curly brackets. The curly brackets essentially bundle all the statements within the curly brackets into a single "statement." However, having only one statement surrounded by curly brackets is superfluous. For example, the following if and for lines are equivalent:

```

1  if (a < b) {
2      c = a;
3  } else {
4      c = b;
5  }
6
7  if (a < b)
8      c = a;
9  else
10     c = b;
11
12 for (int i=0; i<10; ++i) {
13     x[i] = i;
14 }
15
16 for (int i=0; i<10; ++i)
17     x[i] = i;

```

One place where this can get ambiguous is for if–then–else clauses. For example, consider the following code:

```

1  if (a < b)
2      if (c < b)
3          d = c;
4  else
5      d = b;
6
7
8  if (a < b)
9      if (c < b)
10     d = c;
11 else
12     d = b;
13
14 if (a < b) {
15     if (c < b)
16         d = c;
17 } else
18     d = b;

```

For the version on Lines 1–5, the indentation suggests that the else clause corresponds to the second if statement. By comparison, the else clause for Lines 8–12 suggests it corresponds to the first if statement. However, C does not care about the organization of white space in a program, so the two clauses will produce the same answer no matter which way the lines are indented. Therefore, in situations like this, use curly brackets to indicate the intent even if it seems like it may be redundant. For example, consider the version on Lines 14–18 where the else clause belongs to the first if despite the deceptive formatting of the text.

## 5.4 ■ switch-case Statements

The `if-then-else` construct is useful when a program can branch in one of two directions, but sometimes the program must be able to branch in one of many, i.e., more than two, directions. These cases can be handled by using multiple `if` statements, but sometimes a case can be handled more easily using the `switch` statement. The `switch` statement has a test value that is not a boolean but an integral or character value. This structure may be helpful in distributed computing situations because at some point the program may have to branch depending on which processor is running (more on that later). For example, consider the following code:

```

1 int rank;
2 MPI_Comm_rank (MPI_COMM_WORLD, &rank );
3 switch ( rank ) {
4     case 0:
5         // do work on process with rank == 0
6         break;
7     case 1:
8         // do work on process with rank == 1
9         break;
10    case 2:
11        // do work on process with rank == 2
12        break;
13    default:
14        // do work for any other process not already covered
15 }
```

We'll learn more about the rank of an MPI program in Part III, so for now, just note that `rank` is an integer. In the `switch` statement, the expression in parentheses after the `switch` is evaluated, then compared against each of the values after each `case`. If there's a match, then execution of the program jumps to that `case` statement. The `default` label is optional and will be executed if none of the other cases match. Note the presence of `break` statements on Lines 6, 9, and 12. These are necessary unless it is desired for the flow to continue to the next `case`. For example, suppose the `break` on Line 9 is removed and `rank` equals 1; then when the code following `case 1` is completed, the program will continue with the code following `case 2` until it reaches the `break` statement at the end of `case 2`. There are instances where this may be helpful, but it is mostly presented here as a warning against inadvertently forgetting the `break` statements.

## Exercises

- 5.1. Write a program that prints to the screen the number of arguments provided on the command line, which are all assumed to be integers. The last argument should be one of the operations from the list of “+”, “\*”, and “-”. The program should print out the sum, product, or negative of the sum of the arguments, respectively. The last argument should be optional, and if not in the list it is assumed to be “+”.
- 5.2. Write a program that constructs an  $N \times N$  matrix and then creates a second  $N \times N$  matrix that is the transpose of the original.
- 5.3. Write a program to construct a two-dimensional grid with data of the form  $\text{value}[i][j] = \cos(x_i) \sin(y_j)$ , where the  $x_i = \frac{2\pi i}{M}$ ,  $y_j = \frac{2\pi j}{N}$  for  $0 \leq i < M$ ,  $0 \leq j < N$ . The dimensions of the grid,  $M$  and  $N$ , should be read from the

command line using `argv[1]`, `argv[2]`, respectively. The results should be saved to a binary data file with the  $M$  and  $N$  stored first, followed by the array data.

- 5.4. Write a program that can read the data from Exercise 5.3 into an array of the correct dimensions, computes an approximation of the partial derivative of the function with respect to  $x$ , and stores it in a new array of the same dimensions. Given grid values  $v_{i,j}$ , the partial derivative with respect to  $x$  is given by

$$vx_{i,j} = \frac{v_{i+1,j} - v_{i-1,j}}{2\Delta x},$$

where  $\Delta x = 2\pi/M$ . Use periodic boundary conditions, hence when  $i = 0$ ,  $v_{-1,j}$  is replaced with  $v_{M-1,j}$ . Similarly, when  $i = M - 1$ ,  $v_{M,j}$  is replaced with  $v_{0,j}$ . It should output the results to a file in the same manner as in Exercise 5.3.

- 5.5. Write a program that will transpose an array of dimensions  $M \times N$  into an array of dimensions  $N \times M$ .
- 5.6. Write a program using a `while` loop to implement Newton's method to find the root of a quadratic polynomial. Let  $f(x) = ax^3 + bx^2 + cx + d$ , where the coefficients  $a$ ,  $b$ ,  $c$ , and  $d$  are read in from the command line as floating point values. Newton's method has the following update equation:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} = x_n - \frac{ax_n^3 + bx_n^2 + cx_n + d}{3ax_n^2 + 2bx_n + c}.$$

The initial guess  $x_0$  should also be read from the command line. The iteration should terminate when  $|f(x_n)| < 10^{-5}$  and print the results to the terminal.

- 5.7. The code in Exercise 5.6 is not robust because if the values of the coefficients do not admit a solution, the code will run indefinitely. Replace the `while` loop with a `for` loop that will force the method to terminate if more than 100 iterations are taken without converging. If this happens, the output should be a warning message rather than the last value of  $x_n$ .
- 5.8. Write a program to use the Runge–Kutta method (see Section 35.3) to solve the system of differential equations

$$\frac{d}{dt} \begin{bmatrix} x(t) \\ y(t) \\ z(t) \end{bmatrix} = \begin{bmatrix} 4 & -28 & -1 \\ 18 & -41 & 18 \\ -18 & 46 & -13 \end{bmatrix} \begin{bmatrix} x(t) \\ y(t) \\ z(t) \end{bmatrix}, \quad \begin{bmatrix} x(0) \\ y(0) \\ z(0) \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}. \quad (5.1)$$

# Chapter 6

# Functions

In practice no program will be completely contained within the `main` function. For one thing, the complexity of the code makes this an impractical solution, for reasons of speed, ease of understanding, etc. Another important consideration, one that should be present in your mind continuously as you develop your own algorithms and code, is the idea of reusability. Developing complex algorithms and code is difficult and time-consuming. Recoding everything every time a new problem is solved will destroy productivity. Keeping programs as modular as possible, and as generic as possible, will make code reusable and will speed development later. One way to ensure this is to compartmentalize algorithms into functions. Functions are a way to put repeated tasks in one place, to simplify algorithms into simpler building blocks, to make testing and validating code easier, and, if done properly, to create units that can be used in new applications later. In this chapter, we'll dig into the details of the basic building blocks of functions. At the end of the chapter, we'll also discuss functions used to measure the performance of a program, i.e., the time it takes for a program to complete its calculations.

## 6.1 • Declarations and Definitions

The concept of functions is not foreign; they've already appeared in many places such as the `printf` and `scanf` functions, among others. Just like mathematical functions, they can be defined to take any number of input arguments, but they can return only a single value, although there are ways around that limitation. Before a function can be used, it must be declared.

A function declaration is a short recipe for the input and output arguments for the function. For example, the function declaration for  $\sin(x)$  would look like

```
double sin(double x);
```

This declares that `sin` is a function that accepts one double precision argument and returns a double precision result. This line, or one that looks very much like it, appears in a header file called `math.h`. Trying to use the `sin` function without including the `math.h` header file will result in the compiler reporting an error because it doesn't natively know what that function is. The declaration in the header file provides enough information for the compiler to create a socket into which the `sin` function can be

plugged later during the linking process. In this instance, the actual code that computes `sin(x)` is in a library file called `libm.a`. A library file is a collection of object files concatenated together that typically have a common theme, in this case math, that can be used for many other applications as well.

In addition to the function declaration, the function must also be defined. A function definition is the actual code that contains the instructions for that function. The function definition may be in a file that you never see but was used to build an object file or (more likely) a library file. This is the case for the `sin` function. When writing a function, the function definition may be in its own file or in the same file in which it is being used.

The key issue here is that the compiler needs to know about a function before it can be called. So part of what must be considered is whether a planned function is for use in many other code files or whether it is a specialized function that is relevant only to the current code. These two scenarios get to the point of the scope of a function. Consider the creation of a new function `squareOf`, which takes as an argument an integer value and returns the square of the value. The simplest way to create and use this function is to define it before it is called, as is done in this file:

### Example 6.1.

```

1 #include <stdio.h>
2 #include <stdlib.h> // need this include for the atoi function
3
4 /* unsigned int squareOf(int n)
5
6     returns the square of the input integer
7
8     Input: number to be squared
9
10    Output: square of input value
11 */
12 // Note that squareOf is defined before it is called in main
13 unsigned int squareOf(int n)
14 {
15     return n*n;
16 }
17
18 /*
19 int main(int argc, char* argv[])
20
21 Function prints the square of the integer argument
22
23 Inputs: argc should be 2
24     argv[1] will contain the integer input, can be positive or negative
25
26 Outputs: Prints the square of the input
27 */
28
29 int main(int argc, char* argv[])
30 {
31     int n = atoi(argv[1]); // Get the input number
32
33     printf ("%d^2 = %u\n", n, squareOf(n));
34
35     return 0;
36 }
```

In this example, the `squareOf` function is implicitly declared because it is completely defined in this file. In this case, no separate declaration is required, unless the `squareOf`

function is needed in a different code file as well. This case is the example of a local file definition; its scope is limited to this code file and is not visible to code written in other code files.

For some people, it is stylistically unappealing to put the subroutines ahead of the main program, and they prefer the `main` function to be listed first. A slight modification of this version, leaving out the comments and include lines, is listed below:

### Example 6.2.

```

1 // This is the declaration of the function , the definition is at the end
2 unsigned int squareOf(int n);
3
4 int main(int argc , char* argv []);
5 {
6     int n = atoi(argv[1]); // Get the input number
7
8     printf ("%d^2 = %u\n" , n , squareOf(n));
9
10    return 0;
11 }
12
13 // The definition here matches the declaration above
14 unsigned int squareOf(int n) {
15     return n*n;
16 }
```

In this example, the actual definition of `squareOf` is put at the end. But the function is called in `main`, so a declaration of the function is needed so that the compiler can check that it is used properly.

This is a nice introduction to how functions can be put in separate files. The same strategy would apply. Below is an example of the use of multiple code files and a header file. Remember, the `#include` line does exactly that—insert the named file directly into the file being compiled.

### Example 6.3.

#### File `squareof.h`:

```

1 // Declaration of the functions in the code file squareof.c
2 unsigned int squareOf(int n);
```

#### File `squareof.c`:

```

1 #include "squareof.h"
2
3 // The definition matches the declaration in the header file
4 unsigned int squareOf(int n)
5 {
6     return n*n;
7 }
```

#### File `main.c`:

```

1 #include "squareof.h"
2
3 int main(int argc , char* argv [])
4 {
5     int n = atoi(argv[1]); // Get the input number
```

```

7   printf ("%d^2 = %u\n", n, squareOf(n));
8
9   return 0;
10 }
```

In this example, the two files would be compiled separately, then linked together at the end. The use of `squareOf` in the `main` function still requires a declaration in advance, and that is accomplished by including the header file. As far as the compiler is concerned, compiling `main.c` in this example is exactly the same as Lines 2–11 in Example 6.2.

Notice that the `#include` line on Line 1 of `main.c` uses double quotes rather than angle brackets when specifying the file to be included. This distinction has to do with where the compiler will look for the header file. If angle brackets are used, it will search through the system header files, for example, in the directory `/usr/include` and similar directories. These are files you did not write yourself. When double quotes surround the filename, the compiler will look for local files, looking first in the directory where the file is being compiled. Additional local directories can be added to the search using the option `-I/path/to/header/files` when compiling.

## 6.2 • Function Arguments

Communication with functions is through their arguments and return value. Functions may receive any fixed number of arguments (variable length argument lists are possible but are too advanced for this text). Each of those arguments can be passed by value or passed by reference. The variable `n` in Example 6.2 is an example of a variable passed by value. When a variable is passed by value, a copy of the variable is created and used within the function. That means that if the value of `n` were changed in the `squareOf` function, then that change would only be within the function. The value of `n` in the `main` function would remain the same. Passing by value has the advantages that the original variable is kept safe and that it permits the use of constants as an argument. The latter point means that it is permissible, for example, to call `squareOf(7)` as well.

A key drawback, particularly in the world of high performance computing, is that a call by value creates a copy. This is not a cheap operation, particularly if the variable contains a lot of data, because new memory has to be allocated, and the contents of the variable copied into the duplicate. This can be avoided by passing arguments by reference. Passing arguments by reference is where a pointer to the argument is passed rather than the value. Rewriting Example 6.2 to use an argument passed by reference looks like this:

### Example 6.4.

```

1 // This is the declaration of the function ,
2 // the definition is at the end
3 unsigned int squareOf(int* n);
4
5 int main(int argc, char* argv[])
6 {
7     int n = atoi(argv[1]); // Get the input number
8
9     printf ("%d^2 = %u\n", n, squareOf(&n));
10
11    return 0;
12 }
```

```
13 // The definition here matches the declaration above
14 unsigned int squareOf(int* n)
15 {
16     return *n * *n;
17 }
18 }
```

The key changes are that the argument is passed as a pointer by changing the argument on Lines 3, 15. The call of the function on Line 9 uses the “&” to get the address of the variable, and then the value is recovered inside the function by dereferencing the pointer on Line 17. When passed by reference, no new memory allocation or data copying is required—it’s the original data. This means that the value of the argument can be changed within the function, and hence passing arguments by reference is also a way that a function can return multiple values when that’s needed.

When trying to decide which type of method to pass arguments, as a rule of thumb if your variable is a single numerical value, then passing by value is acceptable, but for anything more than that in terms of storage requirements, passing by reference is preferred.

The return value of a function behaves in a very similar way, but in the opposite direction. Because function variables are only temporary, returning variables by reference requires careful attention. The hazard is that if a reference to a local variable is returned, that memory may be gone before it can be used. For example, consider the following functions:

### Example 6.5.

```
1 int* badfunction(int n)
2 {
3     int m = n*n;
4     return &m;
5 }
6
7 int* alsobadfunction(int n)
8 {
9     n = n*n;
10    return &n;
11 }
12
13 int* goodfunction(int n)
14 {
15     int* m = (int*)malloc(sizeof(int));
16     *m = n*n;
17     return m;
18 }
19
20 int* alsogoodfunction(int* n)
21 {
22     *n = *n * *n;
23     return n;
24 }
```

The first example is no good because a reference to a local variable is returned. When the calling function receives this value, the memory where that value is stored will be gone. Because the memory may not be immediately overwritten, the value may still be valid, but there’s no guarantee and it shouldn’t be trusted. The example beginning

on Line 7 is similar but more subtle. The argument `n` is being passed by value so a temporary copy of the argument is created when the function is called. The function is returning a reference to the temporary copy, which again may not exist when returning to the calling function.

The example on Line 13 is acceptable, though it carries a caveat of its own. The call of `malloc` creates memory that will persist after the function returns, so that means the return value will be valid. The caveat concerns the possibility of a memory leak. Recall that every call to `malloc` should be balanced by a call to `free`. In this case, the program must ensure that the memory allocated in this function is subsequently freed before the end of the program. This can be impossible if not handled properly. For example, consider the following code:

```
int* ref = goodfunction(1);
ref = goodfunction(2);
```

When the second call is made, the original address stored in `ref` will be lost, and hence there will be no way to free the memory allocated in the first call.

The last example is also acceptable and is actually quite common among many of the built-in string functions. In this case, the variable in the calling function will be squared and the return value will be a pointer to the same location in memory as was passed in as an argument.

Note that it is not required to assign the return value of a function to a variable. For example, using the last function of Example 6.5, it could be called like this:

```
int n = 5;
alsogoodfunction(&n);
```

The value of the variable `n` upon return will still be 25 and the return value will be discarded.

There may be times when it doesn't make sense to have a return value. In that instance, the function declares the return value to be `void`, and in that case the `return` statement inside the function is omitted. For example, the function above could be rewritten as below and the variable `n` will still be changed to 25:

```
void alsogoodfunction(int* n)
{
    *n = *n * *n;
}
```

One of the key distinguishing features that sets C++ apart from C is the ability to overload functions. C++ allows for multiple versions for the same function name so long as the argument list is different. For example, the function `squareOf` could be declared in two ways:

```
int squareOf(int n)
{
    return n*n;
}

double squareOf(double x)
{
    return x*x;
}
```

The two functions are distinguished because they take different variable types as arguments. Templates in C++ make creating multiple functions like this even easier:

```
template <class T>
T squareOf(T x)
{
    return x*x;
}
```

Any type or class for which the operation  $x*x$  makes sense can then use the `squareOf` function and the compiler will substitute the type of the input variable `x` for `T`.

Passing variables by reference is also easier in C++ by the addition of the reference operator “`&`”. The following two functions both put the result back into the input argument:

```
1 void squareOf(int* x)
2 {
3     *x *= *x;
4 }
5
6 void squareOf(int& x)
7 {
8     x *= x;
9 }
10
11 int main(int argc, char* argv[])
12 {
13     int n = 4;
14     squareOf(&n); // n == 16 after this line
15     squareOf(n); // n == 256 after this line
16     return 0;
17 }
```

As noted above, the two functions can coexist because they take different arguments, the first being a pointer to an integer and the second taking a reference to an integer. However, under the hood both functions do exactly the same thing. In the first version, the pointer is explicitly shown and to dereference the pointer the `*` operator is used. In the second version, `x` is implicitly a pointer and there is essentially an implied `*` operator on the variable `x` inside the function. The consequence of this difference is perhaps more aesthetic in that the first call to `squareOf` requires the explicit `&n` to indicate a pointer to `n` is being passed, while the second call does not require the extra `&` yet accomplishes the same task.

## 6.3 • Measuring Performance

The purpose of this text is to teach you how to do scientific computing faster using modern computer architectures. Thus there must be a means of evaluating that performance that will be based on measuring the time required to solve various problems. In particular, it will be informative to compare the performance of the various parallel methods compared to the serial methods in order to develop intuition for which type of hardware configuration algorithms are best suited for a given task, and to have an idea

**Table 6.1.** Results from running the code in Example 6.6 with parameter values  $N = 10,000,000$  and  $M = 100$ . Linux `time` is using the `time` command when running the program and recording the user time. The other rows are the time checking functions that appear in Example 6.6.

Method	Mean Time (secs)	Variance (secs)
Linux <code>time</code>	$9.7132 \times 10^{-3}$	$5.9064 \times 10^{-3}$
<code>clock</code>	$1.3594 \times 10^{-2}$	$9.6061 \times 10^{-6}$
<code>time</code>	$2.0000 \times 10^{-2}$	$1.9797 \times 10^{-2}$
<code>clock_gettime</code>	$1.3319 \times 10^{-2}$	$5.4800 \times 10^{-6}$
<code>gettimeofday</code>	$1.3060 \times 10^{-2}$	$5.1496 \times 10^{-10}$

of the expected improvement to be gained by utilizing that hardware. This requires a means of computing the elapsed time taken by a computer program.

One set of tools for this purpose has already been discussed: the profiling tools described in Section 1.6. These are powerful tools for analyzing code in detail, but those tools also impose time costs in order to record their data. To get a practical sense of how long an optimized code will run requires simpler timing tools.

There are some methods that do not require additional programming. For example, on Linux systems, the `time` command can be used to measure the elapsed time to run a program. To do this, suppose the program `myprog` is to be run that takes one integer argument such as the dimensions of the problem. Then at the command line, the program is run like this with the following results:

```
$ time myprog 1000
real 0m0.041s
user 0m0.016s
sys 0m0.024s
```

The `time` command measures the time elapsed from beginning to end to run the program in seconds. The `real` time is the so-called wall clock time, i.e., the time you have to surf the Web or drink your coffee while you wait for your program to finish. That time is further subdivided into `user` time and `sys` (system) time. Part of the wall clock time was taken up doing other tasks the computer must perform such as running the time program and sharing CPU resources with other background tasks. To evaluate the program alone, those other extraneous tasks should not be included because they will vary depending on the load on the computer at the time the program was executed. On the other hand, the `user` time is the fraction of time that was exclusively devoted to executing your program. That is the measurement to use to check computational cost.

One drawback to the `time` command is that it is not very refined; it can only measure the performance of an entire code, not break things down into smaller pieces. There are better methods to measure timing, but they require adding functions to the code. On Linux, there are multiple means of checking the time from within a program, including the functions `clock`, `time`, `clock_gettime`, and `gettimeofday`. Table 6.1 compares these methods in Example 6.6 and to the Linux `time` command applied to the equivalent code. The lowest variance, which also indicates it's the most reliable, is reported by the method `gettimeofday`.

**Example 6.6.**

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4 #include <sys/time.h>
5 #include <unistd.h>
6 #include <math.h>
7
8 /*
9 void diff(double* u, int N, double dx, double* du)
10
11 Computes a finite difference approximation for du/dx
12
13 Inputs:
14   u: pointer for input array
15   N: length of the array
16   dx: the space step size
17   du: pointer to output array
18
19 Outputs:
20   du: contains the finite difference approximation
21 */
22 void diff(double* u, int N, double dx, double* du) {
23
24   int i;
25   du[0] = (u[1]-u[N-1])/dx/2.;
26   for (i=1; i<N-1; ++i) {
27     du[i] = (u[i+1]-u[i-1])/dx/2.;
28   }
29   du[N-1] = (u[0]-u[N-2])/dx/2.;
30
31 }
32
33 /*
34 void init(double* u, int N, double dx)
35
36 Initializes the data array with the sin function
37
38 Inputs:
39   u: pointer to array will values will be stored
40   N: length of the array
41   dx: the space step size
42
43 Outputs:
44   u: contains data for a sin function
45 */
46 void init(double* u, int N, double dx)
47 {
48   int i;
49   for (i=0; i<N; ++i)
50     u[i] = sin(i*dx);
51 }
52
53 /*
54 int main(int argc, char* argv[])
55
56 Function tests the timing accuracy of 4 different methods
57
58 Inputs: argc should be 3
59   argv[1] is the integer length of the data array
60   argv[2] is the number of timing samples to take
61
```

```

62 | Outputs: Prints the mean and variance of the measured execution time
63 |
64 | int main(int argc, char* argv[])
65 | {
66 |     int N = atoi(argv[1]);
67 |     int M = atoi(argv[2]);
68 |
69 |     double* u = (double*)malloc(N*sizeof(double));
70 |     double* du = (double*)malloc(N*sizeof(double));
71 |     double dx = 2.*M_PI/N;
72 |
73 |     init(u, N, dx);
74 |
75 |     // Test the function clock()
76 |     int i;
77 |     double times[M];
78 |     double total = 0.;
79 |     for (i=0; i<M; ++i) {
80 |         // start the timer
81 |         clock_t begin1 = clock();
82 |
83 |         // run some code where timing is desired
84 |         diff(u, N, dx, du);
85 |
86 |         // stop the timer and print the results
87 |         clock_t end1 = clock();
88 |         times[i] = ((double)(end1-begin1))/CLOCKS_PER_SEC;
89 |         total += times[i];
90 |
91 |     double mean = total/M;
92 |     double var = 0.;
93 |     for (i=0; i<M; ++i)
94 |         var += (times[i]-mean)*(times[i]-mean);
95 |     printf("mean elapsed time = %e, variance = %e\n", total/M, var/(M-1));
96 |
97 |     // Test the function time()
98 |     total = 0.;
99 |     for (i=0; i<M; ++i) {
100 |         // start the timer
101 |         time_t begin2 = time(NULL);
102 |
103 |         // run some code where timing is desired
104 |         diff(u, N, dx, du);
105 |
106 |         // stop the timer and print the results
107 |         time_t end2 = time(NULL);
108 |         times[i] = difftime(end2, begin2);
109 |         total += times[i];
110 |
111 |     mean = total/M;
112 |     var = 0.;
113 |     for (i=0; i<M; ++i)
114 |         var += (times[i]-mean)*(times[i]-mean);
115 |     printf("mean elapsed time = %e, variance = %e\n", total/M, var/(M-1));
116 |
117 |     // Test the function clock_gettime()
118 |     total = 0.;
119 |     for (i=0; i<M; ++i) {
120 |         // start the timer
121 |         struct timespec start3;
122 |         clock_gettime(CLOCK_MONOTONIC, &start3);
123 |
124 |         // run some code where timing is desired

```

```

125     diff(u, N, dx, du);
126
127     // stop the timer and print the results
128     struct timespec end3;
129     clock_gettime(CLOCK_MONOTONIC, &end3);
130     times[i] = (end3.tv_sec + end3.tv_nsec/1.0e9)
131         - (start3.tv_sec + start3.tv_nsec/1.0e9);
132     total += times[i];
133 }
134 mean = total/M;
135 var = 0.;
136 for (i=0; i<M; ++i)
137     var += (times[i]-mean)*(times[i]-mean);
138 printf("mean elapsed time = %e, variance = %e\n", total/M, var/(M-1));
139
140 // Test the function gettimeofday()
141 total = 0.;
142 for (i=0; i<M; ++i) {
143     // start the timer
144     struct timeval start4;
145     gettimeofday(&start4, NULL);
146
147     // run some code where timing is desired
148     diff(u, N, dx, du);
149
150     // stop the timer and print the results
151     struct timeval end4;
152     gettimeofday(&end4, NULL);
153     times[i] = (end4.tv_sec + end4.tv_usec/1.0e6)
154         - (start4.tv_sec + start4.tv_usec/1.0e6);
155     total += times[i];
156 }
157 mean = total/M;
158 var = 0.;
159 for (i=0; i<M; ++i)
160     var += (times[i]-mean)*(times[i]-mean);
161 printf("mean elapsed time = %e, variance = %e\n", total/M, var/(M-1));
162
163 free(u);
164 free(du);
165
166 return 0;
167 }
```

Example 6.6 shows how to use each of these methods to measure the elapsed user time from within a program. In this example, they are used to measure how much time is being spent in the function `my_function`. Think of this as starting and stopping a stopwatch.

The first method is the `clock` function, which is declared in the header `time.h`. On Line 87, the stopwatch is started running by measuring the time that this point in the program is reached. Note that the output of the `clock` function is of type `clock_t`. The stopwatch is stopped on Line 88. Actually, the `clock` continues to run and it can be sampled many times. The elapsed time is the difference between the two measured time points. In this case, the elapsed time is the difference between the `end` and `begin` values measured in fractions of a second. To convert to seconds use the conversion factor `CLOCKS_PER_SEC` declared in the header file as on Line 88.

The second method is the `time` function, which is also declared in the header `time.h`. The `time` function returns a value of type `time_t`. The difference between

the start and end times is measured using the `difftime` on Line 108, returning a double precision value for the time elapsed in seconds.

The third method is the `clock_gettime` function. For this function, the samples of type `struct timespec` are passed by reference as the second argument, with the first argument being the flag `CLOCK_MONOTONIC` as done on Lines 122, 129. The time measurement includes both the current number of seconds in struct member `tv_sec` plus the number of nanoseconds in `tv_nsec`. These are added together and compared at two different time points on Line 130 to get the elapsed time.

The last method, the function `gettimeofday`, works in a very similar fashion to the previous case. The function is defined in a different header file `sys/time.h`. The type of the time measurement is slightly different, using the type `struct timeval`, and this struct counts microseconds in the struct member `tv_usec` instead of nanoseconds as shown on Line 153.

It is important to understand that, as shown in Table 6.1, the measurement of time is not absolute in the sense that there can be some small variation in the measured elapsed execution time. This means that to do a proper study, one should do multiple runs to get a statistical measurement of the execution time. In practice, depending on the size of the problem, this may or may not be practical. In the example used for Table 6.1, almost all the methods agreed on the first two digits, so measuring time over relatively long time scales is not very sensitive to the measurement method used. With that in mind it is tempting to use the easiest method, but when working to improve the performance of code, it is important to focus effort where it is needed most: the costliest steps in your algorithm, which may have many iterations for an operation that takes very little time. In that case, the accuracy of these measurements becomes more important when trying to shave critical micro- or nanoseconds from an algorithm. In that case, the latter two methods are the better choices.

Review the discussions in Sections 1.3 and 1.6 to recall how to improve performance through optimization at compile time and how to track down the portions of your code that are most responsible for the time elapsed.

---

## Exercises

- 6.1. Write a function that computes the complex exponential function from a variable that is a complex struct. The function declaration should be

```
struct complex compexp(struct complex z);
```

Write a second function that computes the product of two complex variables. Test the functions by computing  $e^{i\pi/2}e^{i3\pi/2}$ .

- 6.2. Write a function with the following declaration:

```
void saveData(char* filename, int rows, int cols, double* data);
```

that writes a two-dimensional array of double precision data into a file. When you write the data, you should write the two dimensions of the matrix and then the actual data. Write a second function with the declaration

```
bool readData(char* filename, int* rows, int* cols, double** data);
```

which reads a file with the given filename. It should store the dimensions of the data in `rows` and `cols`, allocate the memory to store the data, and then read the data into the memory pointed to by `data`. It should return the value of `true` if the data was successfully read and `false` if not. Test the two functions by writing and reading the data and comparing to the original data.

- 6.3. Write a function with the following declaration:

```
void derivative(int M, int N, double (*data)[N],  
                double (*deriv)[N], char direction);
```

The function should compute the finite difference approximation of the derivative using periodic boundary conditions in the direction specified as either '`x`' or '`y`'. The results should be stored in the array `deriv`.



# Chapter 7

# Using Libraries

There is absolutely no reason to reinvent the wheel. Fortunately, there are computational scientists that are really good at creating useful wheels. A classic example is the widely used numerical linear algebra package called LAPACK. It is a collection of useful functions for handling numerical linear algebra problems. The package is large and has many more options than can be covered in this text, so just a few examples are included of calling representative functions that will be particularly useful for the projects in this book. Use the documentation for the libraries to take full advantage of their capabilities. In this chapter, we will see how to incorporate library content into programs both during the compile stage and in the link stage of building an executable. We will then explore the BLAS and LAPACK libraries for linear algebra operations, the FFTW library for doing fast Fourier transforms (FFTs), and also some basic concepts related to random number generation. These topics will all be of particular relevance to the computing projects proposed at the end of Part I.

As discussed in Section 1.3, the compiler leaves dangling references to functions that are not in the code file being compiled at that time. At link time, the linker seeks to resolve all the dangling function references in order to create a complete program. A library is a collection of compiled code that is lumped together into a single file called a library file. These files can typically be found in the `/usr/lib` directory and will have a name that begins with “lib” and end with either “`.a`” or “`.so`”. When included during the linking process, the label used for the library is the text that follows the leading “lib” and excludes the suffix. For example, suppose a trigonometric function is used in a program, which is included in the C math library. The math library has the name `libm.a`. When linking, use the link library option “`-l`” followed by the stripped down library name, which in this case is just “`m`”. Thus, the math library is linked this way:

```
$ gcc -o myprog mycode.o -lm
```

The suffixes “`.a`” and “`.so`” are different enough that they are worth discussing. Libraries with suffix “`.a`” are called static libraries. When a static library is linked, the code for the subroutines required to satisfy any dangling function references will be copied from the library and saved in the final binary program. If a program calls many different library functions, then it can lead to a rather large and bloated binary file. On

the other hand, the binary file is self-contained and is more insulated from operating system upgrades and other confounding events. Libraries with the suffix “.so” are called dynamic libraries. When a dynamic library is linked, the linker is given reference information about how to find the function, but the code itself is not copied. Instead, when the code is actually run and a function is needed from the library, a run-time loader will run the function directly from the library as needed. The advantage of this strategy is that the binary is smaller and can take advantage of helpful upgrades automatically. The risk is that if an upgrade changes the interface or builds the code with a different and incompatible version of the compiler, then the program may require recompiling. Both strategies have their merits; however, library users don’t typically have a choice but to use whichever form of library that is available.

## 7.1 • BLAS and LAPACK

The BLAS library is a collection of highly efficient basic linear algebra subroutines that are typically (but not always) optimized for the hardware. It includes simple operations like vector and matrix arithmetic but does not include any matrix inversion or linear equation solvers. The LAPACK library [13] is built on top of the BLAS library to provide the higher order linear equation solvers of various flavors. A sample code that uses the BLAS library is shown below:

### Example 7.1.

```

1 #include <stdio.h>
2 #include <cblas.h> // This is the C interface for the BLAS library
3
4 /*
5 int main(int argc, char* argv[])
6
7 The main program creates two static vectors and then computes their
8 dot product using the BLAS routine ddot (double precision dot product)
9
10 Inputs: none
11
12 Outputs: The dot product of the two vectors
13 */
14
15 int main(int argc, char* argv[])
16 {
17     // x and y are the two vectors that we will use
18     double x[5] = {1., 2., 3., 4., 5.};
19     double y[5] = {1., -1., 1., -1., 1.};
20
21     // This is the call to the BLAS function.
22     // See the BLAS documentation for other functions and
23     // the definition of the arguments.
24     // In this case, the first argument is the length of the
25     // vectors, and the 3rd and 5th arguments are the increments
26     // to use when traversing the vectors.
27     double dotprod = cblas_ddot(5, x, 1, y, 1);
28
29     // print the result
30     printf("<x,y> = %lf\n", dotprod);
31
32     return 0.;
33 }
```

This example shows how to call the dot product function. The BLAS library function declarations are included on Line 2. Line 27 is where the function is called. For this particular function, it takes five arguments. The first argument is the length of the vector. In this case, two statically allocated arrays of length five are created on Lines 18, 19. The second and fourth arguments are the two vectors, and the third and fifth arguments are the strides. Many of these functions require the stride because it allows easy traversal of matrices where the stride may be one for going along columns but larger than one for traversing rows.

Building the program requires linking the BLAS library; thus the commands to build the executable ddot are

```
$ gcc -c main.c
$ gcc -o ddot main.o -lblas
```

A complete list of the BLAS subroutines can be found at

<http://www.netlib.org/blas/>

LAPACK is a linear algebra package that builds upon the basic blocks in the BLAS library and is used to solve linear systems and to compute eigenvalues, among other things. It also has specialized linear solvers for inverting matrices with particular structures, for example, tridiagonal or banded. This package is useful for solving small to medium systems but *LAPACK is not a parallel implementation*. Thus, it is fine to use within a single process but not across multiple processes or for inverting very large systems.

One thing to bear in mind when using LAPACK is that it was originally built in Fortran and ultimately those Fortran subroutines will be called. In Fortran, matrices are stored in column-major order, which is the opposite of arrays in C that are stored in row-major order, i.e., the rows, or the first index, increment slowest, and the columns or second index increment fastest. Consequently, Fortran interprets a matrix as the transpose of how C interprets a matrix. Also, Fortran function arguments are all passed by reference, so passing scalar values to these functions can be awkward.

Fortunately, the LAPACKE library provides a C-friendly interface that navigates around these difficulties. A matrix may be stored in either row-major or column-major order and is specified in the function call by using the constant `LAPACK_ROW_MAJOR` or `LAPACK_COL_MAJOR`. The scalar arguments, where appropriate, are passed by value instead of by reference, and the error-checking variable `info` is now returned by the function rather than passed through the argument list.

To illustrate these points consider Example 7.2, where the function `LAPACKE_dgesv` solves a single linear system of equations. Note that for convenience of the discussion here, the matrix is stored in column-major order.

### Example 7.2.

```
1 #include <stdio.h>
2 #include <lapacke.h> // The C interface for LAPACK
3
4 /*
5 int main(int argc, char* argv[])
6
7 The main program creates a statically allocated linear system,
8 and then solves Ax = b.
9 The original problem with the solution is printed out at the end.
```

```

10
11 Inputs: none
12
13 Outputs: Prints the original problem with the solution
14 */
15
16 int main(int argc, char* argv[])
17 {
18     // The dimension of the matrix A
19     // lapack_int is a special type defined for LAPACK,
20     // it is usually the same as int
21     lapack_int n = 5;
22
23     // The number of columns of b.  nrhs can be >= 1 if multiple solutions
24     // for the same A are desired
25     lapack_int nrhs = 1;
26
27     // The matrix A stored in column-major order
28     double a[5][5] = {{-2.,2.,0.,0.,0.},
29                         {1.,-2.,1.,0.,0.},
30                         {0.,1.,-2.,1.,0.},
31                         {0.,0.,1.,-2.,1.},
32                         {0.,0.,0.,1.,-2.}};
33
34     // The solver will not preserve the matrix, so we will keep a copy of
35     // the original A so that we can print it out at the end.
36     double aorig[5][5];
37     for (int i=0; i<5; ++i)
38         for (int j=0; j<5; ++j)
39             aorig[i][j] = a[i][j];
40
41     // The function asks for a pointer to the matrix array, so we have to
42     // create a pointer of the right type.
43     double* A = &(a[0][0]);
44
45     // This is the right hand side, also stored in column-major format.
46     double b[5] = {1.,2.,3.,4.,5.};
47
48     // Upon return, the answer will be stored in b, so we have to keep
49     // a copy of b so we can print the original problem at the end.
50     double borig[5];
51     for (int i=0; i<5; ++i)
52         borig[i] = b[i];
53
54     // Variable lda is the leading dimension of the matrix A,
55     // generally the same as n above.
56     lapack_int lda = 5;
57
58     // Variable ipiv is an array for storing the row order.
59     // If there are row interchanges during Gaussian elimination, that
60     // info is stored in ipiv.
61     // The data will be initialized within LAPACK, so we only have to
62     // create it, and it has to have storage for as many integers
63     // as the rows in A.
64     lapack_int ipiv[5];
65
66     // This is the leading dimension of the right-hand side, generally
67     // will be the same as lda.
68     lapack_int ldb = 5;
69
70     // If there is an error, a code indicating the type of error will be
71     // stored in info upon return.
72     lapack_int info = 0;

```

```

73
74 // This is the actual LAPACK call.
75 // Note that all variables are passed by reference.
76 // Upon return A and ipiv will have the encoded LU decomposition,
77 // and b will contain the solution to the original linear system.
78 info = LAPACKE_dgesv(LAPACK_COL_MAJOR, n, nrhs, A, lda, ipiv, b, ldb);
79
80 // Print out the original system Ax = b replacing x with the solution.
81 printf("Check this:\n");
82 for (int i=0; i<5; ++i) {
83     printf("[");
84     for (int j=0; j<5; ++j) {
85         // Note that the aorig is printed out as if a transpose,
86         // because that's the way Fortran and LAPACK interpret the array
87         // (different from C).
88         printf("%9.3f", aorig[j][i]);
89     }
90     printf("] [%9.3f]", b[i]);
91     if (i==2)
92         printf(" = ");
93     else
94         printf("    ");
95     printf("[%5f]\n", borig[i]);
96 }
97 return 0.;
98 }
```

The output of this program is shown below:

**Check this:**

```

[-2.000 1.000 0.000 0.000 0.000] [-17.500] [1.000000]
[ 2.000 -2.000 1.000 0.000 0.000] [-34.000] [2.000000]
[ 0.000 1.000 -2.000 1.000 0.000] [-31.000] = [3.000000]
[ 0.000 0.000 1.000 -2.000 1.000] [-25.000] [4.000000]
[ 0.000 0.000 0.000 1.000 -2.000] [-15.000] [5.000000]
```

To begin, the C interface for LAPACK is included on Line 2. Not all systems have this header file installed by default; it can be downloaded from Netlib [14] and other websites. If the LAPACKE library is not available, then it is still possible to use the LAPACK library in the original Fortran form, but it requires some modifications. First, the header file on Line 2 is removed and replaced with an explicit reference to the function `dgesv_`:

```
2 extern void dgesv_();
```

This signals that the function exists and that it will be linked in later. It is not required to insert all the arguments, but if they are the line would be

```
2 extern void dgesv_(int*, int*, double*, int*, int*, double*, int*, int*);
```

Next, the type `lapack_int`, which is defined in `lapacke.h`, is replaced with `int` throughout. Finally, Line 78 is replaced with the original Fortran version of the subroutine `dgesv_` with all the variables passed by reference and assuming the matrix is stored in column-major order

```
78 dgesv_(&n, &nrhs, A, &lda, ipiv, b, &ldb, &info);
```

The underscore character trailing the name of the function is required even though the Fortran code does not have that character in the original code. It is a device for the linker to understand it is connecting to Fortran style code instead of C. It is not uncommon to have libraries of numerical codes that are written in Fortran, so it is important to understand how to access those libraries from another language.

In LAPACK, and similarly with BLAS, the function names actually have some information about their purpose built into the name. The first letter conveys the number type being used, where “s” means single precision, “d” means double precision, “c” means single precision complex, and “z” means double precision complex. In LAPACK, the next two letters indicate the type of matrix to be operated on. There are several, but a couple useful examples are “ge” for general, i.e., a full matrix with no special structure, and “gt” for general tridiagonal. The rest of the name represents the actual operation. In this example, “sv” means solve. See the LAPACK manual for more information about the naming convention (<http://www.netlib.org/lapack>).

To build this program with the LAPACKE library, the link line will look like

```
$ gcc -o linsolve main.o -llapacke -llapack -lblas
```

The BLAS library is also listed even though no BLAS functions were called directly because the LAPACK library is built on top of the BLAS library, meaning it uses functions found in the BLAS library. If the original LAPACK and not the LAPACKE library is being used, then `-llapacke` should be removed from this line.

Note that the link to `-llapacke` is before the links to `-llapack` and `-lblas`. This turns out to be important because of how the linker works. The linker starts with the final product, in this case `linsolve`, and searches for any loose function calls that are unresolved. In this case, it finds the function `LAPACKE_dgesv`. It then goes to the first library and checks to see if any of these functions can be resolved by this library. It finds `LAPACKE_dgesv` in the LAPACKE library, but this function in turn calls some functions in the LAPACK library, most notably `dgesv_`. That function in turn uses a host of functions in the BLAS library. Fortunately, those functions are found in the BLAS library, and everything is resolved. If the two libraries `lapack` and `blas` were listed in reverse order, then there would be a problem. If that happened, then the linker would attempt to resolve `dgesv_` in the BLAS library and would come up empty. It would then go to the LAPACK library, where it resolves that, but it now has a collection of BLAS functions to resolve. Those are now left dangling, and so the linker will fail. Thus, it is important to understand what all the dependencies are among the code and the libraries so that all the functions that use the libraries get properly resolved.

The LAPACK library has many linear algebra functions for matrices with special structures such as tridiagonal. Because tridiagonal solvers are particularly useful in many numerical methods including one of the projects in Part VI, Example 7.3 demonstrates the tridiagonal solver in LAPACK to solve the same problem as in Example 7.2. This time, the solution method is broken down into two steps, the LU factorization followed by the back substitution, because it is more efficient to factor the matrix **A** once, then reuse the factorization repeatedly whenever a solution for a different right-hand side is needed. This is illustrated in the following example:

### Example 7.3.

```
1 #include <stdio.h>
2 #include <lapacke.h> // The C interface for LAPACK
3 /*
4 */
```

```

5   int main(int argc, char* argv[])
6
7   The main program creates a statically allocated linear system,
8   and then solves  $Ax = b$ .
9   The original problem with the solution is printed out at the end.
10
11  Inputs: none
12
13  Outputs: Prints the original problem with the solution
14 */
15
16 int main(int argc, char* argv[])
17 {
18     // The dimension of the matrix A
19     // lapack_int is a special type defined for LAPACK,
20     // it is usually the same as int
21     lapack_int n = 5;
22
23     // The number of columns of b. nrhs can be >= 1 if multiple solutions
24     // for the same A are desired.
25     lapack_int nrhs = 1;
26
27     // The tridiagonal matrix A stored as 3 diagonals
28     double ald[5] = {2.0, 1.0, 1.0, 1.0, 0.0};
29     double ad[5] = {-2.0, -2.0, -2.0, -2.0, -2.0};
30     double aud[5] = {1.0, 1.0, 1.0, 1.0, 0.0};
31     double auud[5] = {0.0, 0.0, 0.0, 0.0, 0.0};
32
33     // The factorization will not preserve the diagonals, so we will keep
34     // a copy of the original diagonals so that we can print it out
35     // at the end.
36     double aorig[3][5];
37     for (int i=0; i<5; ++i) {
38         aorig[0][i] = ald[i];
39         aorig[1][i] = ad[i];
40         aorig[2][i] = aud[i];
41     }
42
43     // This is the right hand side, also stored in column-major format.
44     double b[5] = {1., 2., 3., 4., 5.};
45     double c[5] = {5., 4., 3., 2., 1.};
46
47     // Upon return, the answer will be stored in b, so we have to keep
48     // a copy of b so we can print the original problem at the end.
49     double borig[5], corig[5];
50     for (int i=0; i<5; ++i) {
51         borig[i] = b[i];
52         corig[i] = c[i];
53     }
54
55     // Variable ipiv is an array for storing the row order.
56     // If there are row interchanges during Gaussian elimination ,
57     // that info is stored in ipiv.
58     // The data will be initialized within LAPACK, so we only have to
59     // create it, and it has to have storage for as many integers as
60     // the rows in A.
61     lapack_int ipiv[5];
62
63     // If there is an error, a code indicating the type of error will be
64     // stored in info upon return.
65     lapack_int info = 0;
66
67     // This is where the matrix A is factored.

```

```

68 // Upon return ald, ad, aud, auud and ipiv will have the encoded LU
69 // decomposition for use with the companion function dgtrrs.
70 info = LAPACKE_dgtrrf(n, ald, ad, aud, auud, ipiv);
71
72 // To solve Ax=b, we set the transpose character to 'N'
73 char trb = 'N';
74
75 // To solve A'x=c, we set the transpose character to 'T'
76 char trc = 'T';
77
78 // This is where the backsubstitution is done.
79 // This function does not alter the factorization so it can be reused.
80 info = LAPACKE_dgtrrs(LAPACK_COL_MAJOR, trb, n, nrhs, ald, ad, aud,
81 auud, ipiv, b, n);
82 info = LAPACKE_dgtrrs(LAPACK_COL_MAJOR, trc, n, nrhs, ald, ad, aud,
83 auud, ipiv, c, n);
84
85 // Print out the original system Ax = b replacing x with the solution
86 printf("Check Ax=b:\n");
87 for (int i=0; i<5; ++i) {
88     printf("[");
89     for (int j=0; j<5; ++j) {
90         switch (j-i) {
91             case -1: printf("%9.3f", aorig[0][i-1]); break;
92             case  0: printf("%9.3f", aorig[1][i]);   break;
93             case  1: printf("%9.3f", aorig[2][i]);   break;
94             default: printf("%9.3f", 0.); break;
95         }
96     }
97     printf("] [%9.3f]", b[i]);
98     if (i==2)
99         printf(" = ");
100    else
101        printf("   ");
102    printf("[%5f]\n", borig[i]);
103}
104
105// Print out the original system A'x = c replacing x with the solution
106printf("\nCheck A'x=c:\n");
107for (int i=0; i<5; ++i) {
108    printf("[");
109    for (int j=0; j<5; ++j) {
110        switch (j-i) {
111            case -1: printf("%9.3f", aorig[2][i-1]); break;
112            case  0: printf("%9.3f", aorig[1][i]);   break;
113            case  1: printf("%9.3f", aorig[0][i]);   break;
114            default: printf("%9.3f", 0.); break;
115        }
116    }
117    printf("] [%9.3f]", c[i]);
118    if (i==2)
119        printf(" = ");
120    else
121        printf("   ");
122    printf("[%5f]\n", corig[i]);
123}
124
125return 0.;
}

```

The output of this program is

**Check Ax=b:**

```
[ -2.000  1.000  0.000  0.000  0.000] [ -17.500] [1.000000]
[  2.000 -2.000  1.000  0.000  0.000] [ -34.000] [2.000000]
[  0.000  1.000 -2.000  1.000  0.000] [ -31.000] = [3.000000]
[  0.000  0.000  1.000 -2.000  1.000] [ -25.000] [4.000000]
[  0.000  0.000  0.000  1.000 -2.000] [ -15.000] [5.000000]
```

**Check A'x=c:**

```
[ -2.000  2.000  0.000  0.000  0.000] [ -42.500] [5.000000]
[  1.000 -2.000  1.000  0.000  0.000] [ -40.000] [4.000000]
[  0.000  1.000 -2.000  1.000  0.000] [ -33.500] = [3.000000]
[  0.000  0.000  1.000 -2.000  1.000] [ -24.000] [2.000000]
[  0.000  0.000  0.000  1.000 -2.000] [ -12.500] [1.000000]
```

In Example 7.3, the matrix is represented by three arrays representing the lower (`ald`), main (`ad`), and upper (`aud`) diagonals of the matrix  $A$ , which are initialized on Lines 28–30. For purposes of the factorization, space for an extra upper diagonal must also be provided to the library (see Line 31). The matrix is factored on Line 70. Like for the function `LAPACKE_dgesv` in the previous example, the function `LAPACKE_dgttrf` also modifies the diagonals, so if the original matrix is needed, be sure to make a copy. However, the back substitution function `LAPACKE_dgttrs` does not modify the output of the `LAPACKE_dgttrf` function. That means the matrix is factored only once, and then the factorization can be reused multiple times as illustrated on Lines 80–82.

The function `LAPACKE_dgttrs` has another argument that didn't appear in the earlier example. The first argument after the matrix ordering allows the problem to be modified. If the argument is the character '`N`', then the function will solve the equation  $\mathbf{Ax} = \mathbf{b}$ . However, if the first argument is '`T`', then the function solves  $\mathbf{A}^T \mathbf{x} = \mathbf{b}$  (the diagonals are still entered in the same order). These two cases are illustrated on Lines 80–82, respectively. In general, if many equations of the form  $\mathbf{Ax} = \mathbf{b}$  have to be solved for a given matrix  $\mathbf{A}$ , but for many different  $\mathbf{b}$  vectors, then the most efficient way to handle that is to separate the forward elimination and back substitution steps and then only do the back substitution for the different  $\mathbf{b}$  vectors. Multiple right-hand sides can be solved simultaneously by setting the variable `nrhs` to the number of right-hand-side vectors, and the vector  $\mathbf{b}$  would be a matrix stored as indicated by the matrix ordering argument and where each column is a different right-hand-side vector. This is by far the most efficient way to solve multiple problems simultaneously.

In case the `LAPACKE` library is unavailable, the corresponding calls in Example 7.3 would be

```
69 dgttrf_(&n, ald, ad, aud, auud, ipiv, &info);
```

and

```
79 dgttrs_(&trb, &n, &nrhs, ald, ad, aud, auud, ipiv, b, &n, &info);
80 dgttrs_(&trc, &n, &nrhs, ald, ad, aud, auud, ipiv, c, &n, &info);
```

where again the right-hand-side matrices  $\mathbf{b}$  and  $\mathbf{c}$  are stored in column-major order.

For general matrices, the functions `LAPACKE_dgttrf`, `LAPACKE_dgttrs` correspond to `LAPACKE_dgetrf`, `LAPACKE_dgetrs` and the arguments follow a similar format:

```
info = LAPACKE_dgetrf(LAPACK_COL_MAJOR, n, n, A, n, pivot);
info = LAPACKE_dgetrs(LAPACK_COL_MAJOR, tr, n, nrhs, A, n, pivot,
                      b, n);
```

Alternatively, the Fortran versions are

```
dgetrf_(&n, &n, A, &n, pivot, &info);
dgetrs_(&tr, &n, &nrhs, A, &n, pivot, b, &n, &info);
```

Here **n** is the dimensions of the matrix **A**, **nrhs** is the number of columns in the right-hand-side matrix **b**, **pivot** is an integer array of length **n**, and **tr** is the character that is “**N**” for solving  $\mathbf{Ax} = \mathbf{b}$  and “**T**” for solving  $\mathbf{A}^T \mathbf{x} = \mathbf{b}$ . If using the Fortran interface and the matrix **A** is stored in row-major format instead of the expected Fortran-style column-major format, then simply set **tr** = ‘**T**’. Of course, if the matrix **b** has more than one column, then it still must be stored in column-major order either way.

## 7.2 • FFTW

The FFT has many uses for studying signals or for developing pseudospectral methods for solving partial differential equations (PDEs). Very briefly, the FFT does a Fourier transform of real data into discrete spectral coefficients that describe the strength of each fundamental wave mode within the data. A more substantial discussion about what the FFT calculates is given in Section 37.1. For now, the focus is on how to call the library to compute Fourier transforms using the FFT provided in the library FFTW. Documentation for the library can be found at

[www.fftw.org](http://www.fftw.org)

### 7.2.1 • Complex Numbers in C

Before discussing the FFT, a discussion on how to handle complex numbers in C is needed. Complex numbers are not native to the language, so to use them the system header file **complex.h** is included. The header defines three complex types: **float complex**, **double complex**, and **long double complex**. Constant C values can also be specified by using the constant **I** like this:

```
double complex z = 1. + 2.*I
```

Many complex functions are also defined in the header file, such as

function	description
cabs	absolute value
carg	complex argument
cimag	imaginary part
creal	real part
conj	complex conjugate
cexp	complex exponential
csqrt	complex square root
cpow	complex power
csin	complex sine
ccos	complex cosine

In C++, complex values are declared in the header file `complex`. The `complex<T>` class is a template class where the type T can be either floating or integral type. The equivalent data types are `std::complex<float>`, `std::complex<double>` and `std::complex<long double>`. The complex-valued functions described above are also available in C++. When a complex variable is printed as plain text, the values are written as  $(r, c)$ , where r is the real part and c is the imaginary part of the variable.

If the `complex.h` file is included *before* including the `fftw3.h` header, then the `complex` types defined in `complex.h` can be used in FFTW. If `complex.h` is not included, then the FFTW package provides its own complex type, `fftw_complex`, which is equivalent to `double[2]`, where the first element of the array is the real part and the second part of the array is the imaginary part. In the examples below, the `fftw_complex` type will be used. This means that arrays of complex numbers are organized as

<code>real(<math>z_0</math>)</code>	<code>imag(<math>z_0</math>)</code>	<code>real(<math>z_1</math>)</code>	<code>imag(<math>z_1</math>)</code>	$\dots$	<code>real(<math>z_{n-1}</math>)</code>	<code>imag(<math>z_{n-1}</math>)</code>
-------------------------------------	-------------------------------------	-------------------------------------	-------------------------------------	---------	---	---

Furthermore, for various efficiency reasons, the memory allocation is recommended to be done through the FFTW package using the function `fftw_malloc`. It works essentially the same as the regular `malloc` but pays more attention to aligning memory for vector arithmetic accelerators. To allocate a one-dimensional array of length N, the following two statements are equivalent:

```
data = (fftw_complex *) fftw_malloc (sizeof(fftw_complex) * N);
data = fftw_alloc_complex(N);
```

### 7.2.2 • One-Dimensional Serial FFT

The basic strategy for calculating an FFT using the FFTW library is to (1) allocate memory for the array of data, (2) create an execution plan, and (3) apply the FFT to the data by executing the plan. It is important to keep in mind that the execution plan can be used many times so that if the plan is executed inside a loop, the plan should be created *before* the loop. The following example illustrates the calculation of a forward Fourier transform followed by a reverse transform:

#### Example 7.4.

```
1 #include <stdio.h>
2 #include <math.h>
3 #include <fftw3.h>
4
5 /*
6 int main( int argc , char* argv[] )
7
8 Compute the Fast Fourier Transform for data from sin(x)
9
10 Inputs: none
11
12 Outputs: Prints the Fourier coefficients and the data after
13 both forward and backward transformation.
14 */
15
```

```

16 int main(int argc, char* argv[])
17 {
18 #ifndef M_PI
19     const double M_PI = 4.0 * atan(1.0);
20 #endif
21     int N = 16;
22     fftw_complex *in, *out, *out2;
23     fftw_plan p, pinv;
24     in = (fftw_complex *) fftw_malloc(sizeof(fftw_complex)*N);
25     out = (fftw_complex *) fftw_malloc(sizeof(fftw_complex)*N);
26     out2 = (fftw_complex *) fftw_malloc(sizeof(fftw_complex)*N);
27     p = fftw_plan_dft_1d(N, in, out, FFTW_FORWARD, FFTW_ESTIMATE);
28     pinv = fftw_plan_dft_1d(N, out, out2, FFTW_BACKWARD, FFTW_ESTIMATE);
29     for (int i=0; i<N; ++i) {
30         double dx = 2.*M_PI/N;
31         in[i][0] = sin(i*dx);
32         in[i][1] = 0.;
33     }
34     fftw_execute(p);
35     fftw_execute(pinv);
36     for (int i=0; i<N; ++i)
37         printf("a[%d]: %f + %fi\n", (i<=N/2 ? i : i-N), out[i][0]/N,
38                                         out[i][1]/N);
39     printf("----\n");
40     for (int i=0; i<N; ++i)
41         printf("f[%d]: %f + %fi == %f + %fi\n", i, out2[i][0]/N,
42                                         out2[i][1]/N, in[i][0], in[i][1]);
43
44     fftw_destroy_plan(p);
45     fftw_destroy_plan(pinv);
46     fftw_free(in);
47     fftw_free(out);
48     fftw_free(out2);
49 }
```

The first observation is that the header file for FFTW functions is called `fftw3.h`, which appears on Line 3. On Lines 24–26, the memory for the input and output arrays are allocated. On Lines 27, 28, the execution plans are created using the function

```
fftw_plan_dft_1d(N, in, out, FFTW_FORWARD, FFTW_ESTIMATE);
```

The arguments for this function are the number of points in the array, the input array, the output array (may be the same as the input array), the direction of the transform (may also use `FFTW_BACKWARD`), and the optimization scheme (fastest is `FFTW_ESTIMATE`, can also be `FFTW_MEASURE`). Note that the input and output arrays are permitted to be the same. If they are the same, then the algorithm is slightly different, but it is transparent to the end user. Lines 29–33 are where the input data is set up. The FFT is actually executed on Line 34 in the forward direction, and on Line 35 in the reverse direction, by calling `fftw_execute`. Lines 44–48 show how the plan and the memory are freed upon completion.

The output of this program is shown below:

```
a[0]: -0.000000 + 0.000000i
a[1]: -0.000000 + -0.500000i
a[2]: -0.000000 + -0.000000i
a[3]: 0.000000 + -0.000000i
a[4]: 0.000000 + -0.000000i
a[5]: 0.000000 + -0.000000i
a[6]: 0.000000 + -0.000000i
```

```

a[7]: 0.000000 + 0.000000i
a[8]: 0.000000 + 0.000000i
a[-7]: 0.000000 + 0.000000i
a[-6]: 0.000000 + 0.000000i
a[-5]: 0.000000 + 0.000000i
a[-4]: 0.000000 + 0.000000i
a[-3]: 0.000000 + 0.000000i
a[-2]: -0.000000 + 0.000000i
a[-1]: -0.000000 + 0.500000i
-
f[0]: 0.000000 + 0.000000i == 0.000000 + 0.000000i
f[1]: 0.382683 + 0.000000i == 0.382683 + 0.000000i
f[2]: 0.707107 + 0.000000i == 0.707107 + 0.000000i
f[3]: 0.923880 + 0.000000i == 0.923880 + 0.000000i
f[4]: 1.000000 + 0.000000i == 1.000000 + 0.000000i
f[5]: 0.923880 + 0.000000i == 0.923880 + 0.000000i
f[6]: 0.707107 + 0.000000i == 0.707107 + 0.000000i
f[7]: 0.382683 + 0.000000i == 0.382683 + 0.000000i
f[8]: 0.000000 + 0.000000i == 0.000000 + 0.000000i
f[9]: -0.382683 + 0.000000i == -0.382683 + 0.000000i
f[10]: -0.707107 + 0.000000i == -0.707107 + 0.000000i
f[11]: -0.923880 + 0.000000i == -0.923880 + 0.000000i
f[12]: -1.000000 + 0.000000i == -1.000000 + 0.000000i
f[13]: -0.923880 + 0.000000i == -0.923880 + 0.000000i
f[14]: -0.707107 + 0.000000i == -0.707107 + 0.000000i
f[15]: -0.382683 + 0.000000i == -0.382683 + 0.000000i

```

It is important to note that the order of the coefficients produced by the transform are not necessarily in the order that may be expected. The indices shown in the sample output above are not array indices but rather the indices used in the construction of the FFT as described in Section 37.1. This example is looking at the first wave mode; to change to the  $k$ th mode, replace  $i*dx$  on Line 31 with  $k*i*dx$ . Practice with this simple code to understand how different values of  $k$  change the resulting output. Once that is understood, try combinations of the form  $a*\cos(k*i*dx)+b*\sin(k*i*dx)$  to see what happens.

Another consideration for this package is that the backward transform is not exactly the reverse transform of the forward transform. In the forward transform, the coefficients returned are multiplied by the number of collocation points, but in the reverse transform, that multiplier is not divided back out. For the above example, this is compensated for by dividing by  $N$  as shown on Lines 37, 42. Thus, if the Fourier coefficients themselves are needed, the values must be divided by  $N$ , the number of coefficients after the forward transform. The multiplier does not apply in the reverse transform; hence the data is divided by  $N$ , not  $N^2N$ , on Line 41 to recover the original function.

### 7.2.3 • Two-Dimensional Transforms

Transforming data in two and higher dimensions is accomplished in very much the same way as described for one dimension above. It is important to remember that data is stored in column-major order. This will be particularly important when computing spectral derivatives in the pseudospectral method applications in Chapter 37. A two-dimensional plan  $p$  for an  $M \times N$  grid to compute the forward transform is constructed like this:

where `datain` and `dataout` are allocated as before:

<code>fftw_complex *datain = fftw_malloc (M*N*sizeof (fftw_complex));</code>
--

The plan is executed using the same function `fftw_execute(p)` as for one dimension described earlier. The order of the coefficients will have the same structure in each dimension as for the one-dimensional results so that the array is organized as below:

$a_{0,0}$	$a_{0,1}$	$\cdots$	$a_{0,\frac{N}{2}}$	$a_{0,1-\frac{N}{2}}$	$a_{0,2-\frac{N}{2}}$	$\cdots$	$a_{0,-1}$
$a_{1,0}$	$a_{1,1}$	$\cdots$	$a_{1,\frac{N}{2}}$	$a_{1,1-\frac{N}{2}}$	$a_{1,2-\frac{N}{2}}$	$\cdots$	$a_{1,-1}$
$\vdots$	$\vdots$		$\vdots$	$\vdots$	$\vdots$		$\vdots$
$a_{\frac{M}{2},0}$	$a_{\frac{M}{2},1}$	$\cdots$	$a_{\frac{M}{2},\frac{N}{2}}$	$a_{\frac{M}{2},1-\frac{N}{2}}$	$a_{\frac{M}{2},2-\frac{N}{2}}$	$\cdots$	$a_{\frac{M}{2},-1}$
$a_{1-\frac{M}{2},0}$	$a_{1-\frac{M}{2},1}$	$\cdots$	$a_{1-\frac{M}{2},\frac{N}{2}}$	$a_{1-\frac{M}{2},1-\frac{N}{2}}$	$a_{1-\frac{M}{2},2-\frac{N}{2}}$	$\cdots$	$a_{1-\frac{M}{2},-1}$
$a_{2-\frac{M}{2},0}$	$a_{2-\frac{M}{2},1}$	$\cdots$	$a_{2-\frac{M}{2},\frac{N}{2}}$	$a_{2-\frac{M}{2},1-\frac{N}{2}}$	$a_{2-\frac{M}{2},2-\frac{N}{2}}$	$\cdots$	$a_{2-\frac{M}{2},-1}$
$\vdots$	$\vdots$		$\vdots$	$\vdots$	$\vdots$		$\vdots$
$a_{-1,0}$	$a_{-1,1}$	$\cdots$	$a_{-1,\frac{N}{2}}$	$a_{-1,1-\frac{N}{2}}$	$a_{-1,2-\frac{N}{2}}$	$\cdots$	$a_{-1,-1}$

## 7.2.4 • Transforming Real-Valued Data

To transform real-valued data, some savings can be gained by using the plan functions that transform between real and complex-valued data. For one dimension, the functions are

<code>fftw_plan fftw_plan_dft_r2c_1d (N, r_in, c_out, FFTW_ESTIMATE);</code>
<code>fftw_plan fftw_plan_dft_c2r_1d (N, c_in, r_out, FFTW_ESTIMATE);</code>

where `r_in`, `r_out` are real-valued arrays and `c_in`, `c_out` are `fftw_complex` valued arrays. Note that it is assumed that the first plan is for a forward transform, and the second plan is for a backward transform, so those parameters do not need to be specified. For the one-dimensional transform, only the complex coefficients  $a_0, \dots, a_{N/2}$  are computed because the remaining coefficients are the complex conjugates of these. In higher dimensions, only the last dimension is cut in half, so that the coefficients are arrayed as shown below:

$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$\cdots$	$a_{0,\frac{N}{2}},$
$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$\cdots$	$a_{1,\frac{N}{2}},$
$\vdots$	$\vdots$	$\vdots$		$\vdots$
$a_{\frac{M}{2},0}$	$a_{\frac{M}{2},1}$	$a_{\frac{M}{2},2}$	$\cdots$	$a_{\frac{M}{2},\frac{N}{2}},$
$a_{1-\frac{M}{2},0}$	$a_{1-\frac{M}{2},1}$	$a_{1-\frac{M}{2},2}$	$\cdots$	$a_{1-\frac{M}{2},\frac{N}{2}},$
$a_{2-\frac{M}{2},0}$	$a_{2-\frac{M}{2},1}$	$a_{2-\frac{M}{2},2}$	$\cdots$	$a_{2-\frac{M}{2},\frac{N}{2}},$
$\vdots$	$\vdots$	$\vdots$		$\vdots$
$a_{-1,0}$	$a_{-1,1}$	$a_{-1,2}$	$\cdots$	$a_{-1,\frac{N}{2}},$

### 7.2.5 • Building Code with FFTW

Linking the FFTW library is straightforward. It requires linking both the FFTW library and the math library because of the trigonometric function calls involved. Hence, to compile and link the above example, use

```
$ gcc -o main1d main1d.c -lfftw3 -lm
```

## 7.3 • Random Number Generation

Most installations of C provide a few random number generators such as `rand`, `random`, and `drand48`. These are all basic linear congruential generators and as such are not high-quality generators for sensitive numerical work. A popular and fast generator available on the Internet is the Mersenne Twister:

<http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html>

For our purposes here, the stock pseudorandom number generator `drand48` will be used.

For most random number generators, there are two tasks to generate random values; the first is to seed the generator, i.e., initialize the random generator, and the second is to call the generator to get random values. It is important to seed a random number generator for two reasons. First, if the generator is not intentionally seeded, it may default to a particular seed, hence the random values generated will be the same every time the program is run. For production runs, this is definitely not desired. Second, there is utility in reporting the seed used for future reference in case the program runs into an unexpected result or bug. By setting the seed to the same value as when a problem is found, the same sequence of random values will be generated, and hence the problem in the code can be made reproducible enabling the debugging process.

Seeding `drand48` is accomplished by calling the function `srand48` with a seed of type `long int`. The more difficult part is how to choose the value of the seed. One can rely on the user to provide a number, but it is better to do something automated. One solution is to seed it using the current time. For example, use the line

```
srand48((long int)time(NULL));
```

which calls the `time` function to return the current time, encoded as a long integer. Presumably, `time` has elapsed since the previous call, hence the seed will be different every time. Note that a bit of a caveat here is that in a parallel setting, this method may not work because multiple threads may invoke the `time` function at the same time, hence generating the same sequence of random values.

On Linux systems, an alternative method involves reading from a random source of bytes made available by the system in `/dev/urandom`. The good news is that this will produce a truly random value that can be used to seed the generator. Example 7.5 illustrates how to use `/dev/urandom` to seed the random number generator.

#### Example 7.5.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 /*
4 int main(int argc, char* argv[])
5 {
```

```

6
7 Create an array of random values in the range [0,1]
8 It uses a random seed unless a seed is given as an
9 argument.
10
11 Inputs: argc should be 1 or 2
12 argv[1], if given, will be the value of the RNG seed
13
14 Outputs: Prints the contents of the array
15 */
16
17 int main(int argc, char* argv[])
18 {
19     long int seed;
20
21     if (argc > 2) {
22
23         // initial seed is given on the command line, use that one
24         seed = atol(argv[2]);
25
26     } else {
27
28         // initial seed not given, so generate one from /dev/urandom
29
30         // Open the source of random bits
31         FILE* urand = fopen("/dev/urandom", "r");
32
33         // Read enough random bits from the source to fill a long int
34         fread(&seed, sizeof(long int), 1, urand);
35
36         // Close when done, just like a file.
37         fclose(urand);
38     }
39
40     // Print the seed value in case it is needed later
41     printf("seed = %ld\n", seed);
42
43     // Set the seed value
44     srand48(seed);
45
46     // Generate the N random values
47     int N = atoi(argv[1]);
48     double* data = (double*) malloc(sizeof(double)*N);
49     int i;
50     for (i=0; i<N; ++i) {
51         data[i] = drand48();
52         printf("%f\n", data[i]);
53     }
54
55     return 0;
56 }
```

On Line 21, the number of arguments provided by the user is checked. If `argc` is three, then the user has provided a seed as an input argument; otherwise a seed will be read from `/dev/urandom`. The file `/dev/urandom` should be treated like a data file that contains random bits, which is essentially infinite in length, but can be slow to read if a large amount of data is requested. It's not efficient to use as a standalone random generator, but it is a reliable way to set an initial random seed. The `/dev/urandom` file is opened on Line 31, a single random `long int` is read on Line 34, and finally the file is closed on Line 37.

Whether the seed is user provided or read from `/dev/random`, the seed is printed to the screen. This way, if something goes wrong with a particular run and the sequence of random values must be recreated for debugging, the seed that started it will be known. To illustrate this, the program is first run without specifying the seed value, and then on the subsequent run, the seed value is added, resulting in the same sequence:

```
$ myprog 5
seed = -2344942493513672252
0.148597
0.122495
0.582979
0.663548
0.289388
$ myprog 5 -2344942493513672252
seed = -2344942493513672252
0.148597
0.122495
0.582979
0.663548
0.289388
```

The random number generator is actually seeded on Line 44, and thereafter the random number generator `drand48` can be called repeatedly, as is done on Line 51, to generate a sequence of random double precision values in the range  $[0, 1)$  using a linear congruential generator.

## Exercises

- 7.1. Modify Example 7.2 so that it factors the matrix separately from the back substitution. In that case, you will replace the solver `LAPACKE_dgesv` with the pair of functions given below:

```
int LAPACKE_dgetrf(int layout, int m, int n, double* A, int lda,
                     int ipiv);
int LAPACKE_dgetrs(int layout, char T, int n, int nrhs,
                    double* A, int lda, int* ipiv, double* b, int ldb);
```

Without LAPACKE, the functions are

```
dgetrf_(int* m, int* n, double* A, int* lda, int* ipiv,
         int* info);
dgetrs_(char* T, int* n, int* nrhs, double* A, int* lda,
        int* ipiv, double* b, int* ldb, int* info);
```

For `LAPACKE_dgetrf`, the values of `m` and `n` should be the same because they are the dimensions of the matrix `A`. The variables `lda`, `ipiv`, and `info` are the same as for `LAPACKE_dgesv`. When `LAPACKE_dgetrf` is called, the matrix `A` and pivot vector `ipiv` are modified so that it is in an LU decomposition form suitable for back substitution when calling `LAPACKE_dgetrs`.

For LAPACKE\_dgetrs, the first argument is a pointer to a character that has one of three values, 'N', 'T', or 'C', indicating whether the equation to be solved is  $\mathbf{Ax} = \mathbf{b}$ ,  $\mathbf{A}^T\mathbf{x} = \mathbf{b}$ , or  $\mathbf{A}^H\mathbf{x} = \mathbf{b}$ . For this exercise, use 'N'. The arrays  $\mathbf{A}$  and  $\mathbf{ipiv}$  will be the output of calling LAPACKE\_dgetrf, while the remaining arguments are the same as for LAPACKE\_dgesv.

Once this works, modify the program again to make  $\mathbf{b}$  a matrix the same size as  $\mathbf{A}$  (this will require setting  $\text{nrhs}$  to be the same as  $n$ ). Verify that all the solutions returned in the matrix  $\mathbf{b}$  are correct.

- 7.2. Write a function that uses the functions LAPACKE\_dgttrf, LAPACKE\_dgttrs to solve the  $N \times N$  tridiagonal system  $\mathbf{Ax} = \mathbf{B}$  where

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & & & \\ -\delta & 1 + 2\delta & -\delta & & \\ & \ddots & \ddots & \ddots & \\ & & -\delta & 1 + 2\delta & -\delta \\ & & & 0 & 1 \end{bmatrix}$$

The value  $\delta$  and the  $N \times N$  matrix  $\mathbf{B}$  are inputs to the function returning the answer in  $\mathbf{B}$ .

- 7.3. Write a function that uses the functions LAPACKE\_dgetrf, LAPACKE\_dgetrs to solve the  $N \times N$  system  $\mathbf{Ax} = \mathbf{B}$  where

$$\mathbf{A} = \begin{bmatrix} 1 + 2\delta & -\delta & 0 & \cdots & \cdots & 0 & -\delta \\ -\delta & 1 + 2\delta & -\delta & 0 & & & 0 \\ 0 & -\delta & 1 + 2\delta & -\delta & 0 & & \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ & & & 0 & -\delta & 1 + 2\delta & -\delta & 0 \\ 0 & & & & 0 & -\delta & 1 + 2\delta & -\delta \\ \delta & 0 & \cdots & \cdots & 0 & -\delta & 1 + 2\delta \end{bmatrix}$$

The value  $\delta$  and the  $N \times N$  matrix  $\mathbf{B}$  are inputs to the function returning the answer in  $\mathbf{B}$ .

- 7.4. Write a program to read in a double complex value and then print it.
- 7.5. Modify Example 7.4 so that one wave mode is specified by an integer input. Exchange `sin` for `cos` and see what happens with the coefficients. Experiment with different wave modes to see how the output changes.
- 7.6. Modify Example 7.4 to do a two-dimensional FFT transform. In this case, the forward plan is constructed by calling the function

```
fftw_plan fftw_plan_dft_2d(int m, int n, fftw_complex *in,
                           fftw_complex *out, int sign, unsigned in_flags);
```

Aside from the data arrays `in` and `out` being pointers to a two-dimensional array of dimensions  $m \times n$ , the remainder of the arguments are the same as before. Take as inputs the integer wave modes  $k_1$  and  $k_2$  so that the complex variable  $\text{in}[i][j][0] = \cos((k_1*i+k_2*j)*dx)$  and  $\text{in}[i][j][1] = \sin((k_1*i+k_2*j)*dx)$ . Print out the values of the spectral coefficients after the forward transform. Perform the reverse transform and verify that the initial data is recovered after making a suitable adjustment. Try different values of  $k_1$  and  $k_2$  and note how the spectral coefficients depend on the values.

- 7.7. Write a program that generates an array of length  $N$  values that have a Gaussian distribution with mean  $A$  and variance  $B$ . See Section 34.2.2 for an explanation of how to generate the distribution.



# Chapter 8

# Projects for Serial Programming

The background for the projects below is in Part VI of this book. Each of these projects can be built based upon the information provided in this text. The other parts of this book will also draw upon the same projects so the parallel implementation can be compared with the serial implementation. This is also an opportunity to practice building the code without the extra distraction of parallelism. Much of these codes will be reusable later, so be sure to use sufficient comments in the code with detailed comments about what each part of the code is doing. Use modular functions, where appropriate to maximize reusability.

## 8.1 • Random Processes

### 8.1.1 • Monte Carlo Integration

Program the assignment in Section 34.3.1. Be sure to use the random number generator functions `srand48`, `drand48` and not `srand`, `rand` as discussed in Section 7.3. Measure the time required to complete the calculation by using the `gettimeofday` function at the beginning and end of your program and print the elapsed time in seconds.

### 8.1.2 • Duffing–Van der Pol Oscillator

Program the assignment in Section 34.3.2. Measure the time required to complete the calculation as a function of  $NM$  by using the `gettimeofday` function at the beginning and end of your program. Plot an estimate for  $p(t)$  for  $0 \leq t \leq T = 10$ .

## 8.2 • Finite Difference Methods

### 8.2.1 • Brusselator Reaction

Program the assignment in Section 35.4.1. Your program must use the LAPACK functions `LAPACKE_dgttrf`, `LAPACKE_dgttrs` (or the Fortran equivalents `dgttrf_`, `dgttrs_`) to solve the tridiagonal systems in the algorithm. The factorization step using `LAPACKE_dgttrf` should be done only once at the beginning outside the main loop because the matrix to be inverted is constant, while `LAPACKE_dgttrs` should be used inside the main loop. Measure the time required to complete the calculation by using

the `gettimeofday` function at the beginning and end of your program and print the elapsed time in seconds.

### 8.2.2 • Linearized Euler Equations

Program the assignment in Section 35.4.2 using an  $N \times N$  grid. Your program must use the LAPACK functions `LAPACKE_dgetrf`, `LAPACKE_dgetrs` to solve the tridiagonal systems in the algorithm. The factorization step using `LAPACKE_dgetrf` should be done only once at the beginning outside the main loop because the matrix to be inverted is constant, while `LAPACKE_dgetrs` should be used inside the main loop. Measure the time required to complete the calculation by using the `gettimeofday` function at the beginning and end of your program and print the elapsed time in seconds.

## 8.3 • Elliptic Equations and Successive Overrelaxation

### 8.3.1 • Diffusion with Sinks and Sources

Program the assignment in Section 36.1.1 using a  $(2N - 1) \times N$  grid. For a serial implementation, it is not necessary to use red/black ordering, though you may wish to do so to prepare for implementations later in this book. Experiment to find the optimal  $\omega$  that requires the fewest iterations to reach a tolerance of  $10^{-9}$ . Compute the time required to complete the calculation using the `gettimeofday` function, then divide by the number of iterations to give the average time used for a single pass through the grid. Repeat these steps for a grid of dimensions  $(4N - 1) \times 2N$ . Verify that the time cost for a single pass through the grid is proportional to the number of grid points.

### 8.3.2 • Stokes Flow 2D

Program the two-dimensional assignment in Section 36.1.2 using a marker-and-cell (MAC) grid that is approximately  $N \times N$ , i.e., the grid for  $u$  should be  $N \times N - 1$ , the grid for  $v$  should be  $N - 1 \times N$ , and the grid for  $p$  should be  $N - 1 \times N - 1$ . For a serial implementation, it is not necessary to use red/black ordering, though you may wish to do so to prepare for implementations later in this book. Experiment to find the optimal  $\omega$  that requires the fewest iterations to reach a tolerance of  $10^{-9}$ . Compute the time required to complete the calculation using the `gettimeofday` function, then divide by the number of iterations to give the average time used for a single pass through the grid. Verify that you get a parabolic profile for the velocity field in the  $x$ -direction.

### 8.3.3 • Stokes Flow 3D

Program the three-dimensional assignment in Section 36.1.2 using a MAC grid that is approximately  $N \times N \times N$ , i.e., the grid for  $u$  should be  $N \times N - 1 \times N - 1$ , the grid for  $v$  should be  $N - 1 \times N \times N - 1$ , the grid for  $w$  should be  $N - 1 \times N - 1 \times N$ , and the grid for  $p$  should be  $N - 1 \times N - 1 \times N - 1$ . For a serial implementation, it is not necessary to use red/black ordering, though you may wish to do so to prepare for implementations later in this book. Experiment to find the optimal  $\omega$  that requires the fewest iterations to reach a tolerance of  $10^{-9}$ . Compute the time required to complete the calculation using the `gettimeofday` function, then divide by the number of iterations to give the average time used for a single pass through the grid. Verify that you get a roughly parabolic profile for the velocity field in the  $x$ -direction.

## 8.4 • Pseudospectral Methods

### 8.4.1 • Complex Ginsburg–Landau Equation

Program the assignment in Section 37.5.1 using an  $N \times N$  grid. Compute the time required to execute the solution to the indicated terminal time. Plot the results for the phase and identify where the spiral waves have emerged through pattern formation. Measure the time required to complete the calculation by using the `gettimeofday` function at the beginning and end of your program and print the elapsed time in seconds.

### 8.4.2 • Allen–Cahn Equation

Program the assignment in Section 37.5.2 in two dimensions using an  $N \times N$  grid. Compute the time required to execute the solution to the indicated terminal time. Plot the results and verify that phase separation is occurring. Measure the time required to complete the calculation by using the `gettimeofday` function at the beginning and end of your program and print the elapsed time in seconds.



## **Part II**

# **Parallel Computing Using OpenMP**



The types of programs discussed in Part I of this book are called serial programming, which are programs that are designed to run on a single processor, each task performed sequentially in time. The purpose of this text is to explore parallel programming, or programs that are designed to divide their computation among more than one processor. The goal is to divide up the computational effort in such a way that all processors are kept as busy as possible with the hopes that using  $N$  processors will result in a calculation that takes  $1/N$  amount of time compared to the serial version.

OpenMP is just one way in which parallel computing can be implemented to speed up serial programs. It is designed to work on shared memory architectures, which are computers where multiple processors (or cores) use the same memory for storage. Standard desktop and laptop computers are shared memory architectures as they generally have two or more cores within the single processor chip. Each core can do independent tasks, but they still access the same memory. The programmer must be keenly aware of this fact when using OpenMP so as to avoid times when multiple cores try to access the same memory location either for reading or writing. If not handled properly, there can be latency problems as one core must wait for another core to finish before it can proceed, and there can be race conditions, where the outcome of the algorithm depends on the serendipitous timing of which core reads/writes to the same memory location first. The OpenMP framework makes handling these problems fairly straightforward, but it is something that must be considered each time a section of code is parallelized.

The advantage of OpenMP, and one reason for its popularity, is that it does not require substantial extra programming to manage the multiple cores doing various tasks. Not only that, but for many applications, the underlying program will be the same as a corresponding serial program with only a handful of additional compiler directives. That means that the code should, in principle, run without any changes even if OpenMP is not available.

The reader should note that not all clauses available for each compiler directive will be discussed in this book. For some clauses, for example, the **shared** and **private** clauses, their meaning is essentially the same no matter where they appear. For a complete list of the clauses and the directives to which they can be applied, the reader is referred to more comprehensive texts, such as the presentation in [15].

In Chapter 9, multithread computing using OpenMP is introduced. The means of controlling how many threads are used for a task and how variables are allocated between the threads, whether they are shared, common to all threads, or private, with each thread using an independent copy are all discussed. Finally, a **reduction** clause is introduced that makes it possible to combine multiple private variables into a single value.

One of the most useful ways that multithreading improves performance is by breaking loops into smaller pieces with each thread handling its own part of the loop. Subdivision of loops and the corresponding modifiers that control how the loops are divided are addressed in Chapter 10.

Some tasks are inherently serial due to certain dependencies. For example, it's not possible to parallelize solving a single ordinary differential equation because each time step depends on the preceding one. How to handle those instances, where an inherently serial task appears in the middle of a multithreaded region, is discussed in Chapter 11.

When an algorithm has multiple serial tasks that are not interdependent, then different threads can be designated to do unique tasks. Handling multiple *distinct* tasks by using individual threads for each task is covered in Chapter 12.

When multiple threads need to access the same shared variable, care must be taken to ensure that multiple threads don't collide and accidentally write over what another thread has already written. This kind of problem can be handled by designating an assignment statement as `atomic` or `critical`. This forces the threads to cooperate so that if a thread is using a particular variable, that variable is temporarily blocked from other threads that must then wait. Setting up this kind of arrangement is discussed in Chapter 13.

Much of the library discussion presented in Chapter 7 carries over unchanged to the OpenMP framework. The only catch is that ideally *multithread-safe* versions of the libraries should be used. Otherwise, the interfaces for these libraries are unchanged. Chapter 14 discusses the parts of the libraries that are specialized for OpenMP. Of particular note are the special considerations for handling random numbers in a multi-threaded environment.

Finally, Chapter 15 explores how multithreading can be applied to improve computational performance to solve the projects in Part VI.

# Chapter 9

# Intro to OpenMP

For OpenMP, the various parallel “programs” that are created are called threads. A program can create many threads that run in parallel on different cores so that a given task can be broken into smaller pieces, each piece handled by a different thread. For example, when adding two vectors, the vectors could be cut in half, and then one thread may add the first half of the vectors, while the second thread adds the last half. If those two tasks can be done simultaneously, then the time required to perform the task of adding two vectors will effectively be cut in half. One of the key concepts to keep in mind when using OpenMP is to understand the role of variables that are used inside regions where multithreading is being used: do the threads share the same memory space for a variable, or does each get its own distinct copy? In this chapter, we’ll explore how to create a multithread region and discuss the nuances of shared versus private memory for variables. We will also discuss what happens to variables at both the point of entry and the exit of the multithreaded region.

## 9.1 • Creating Multiple Threads: `#pragma omp parallel`

To make each thread do a specific task, the program must identify which thread it is. Example 9.1 shows a bare-bones OpenMP program that has each thread report its thread ID.

### Example 9.1.

```
1 #include <stdio.h>
2 #include <omp.h> // This is the header file for OpenMP directives
3
4 /*
5  int main( int argc , char* argv[] )
6
7  The main program is the starting point for an OpenMP program .
8  This one simply reports its own thread number .
9
10 Inputs : none
11
12 Outputs: Prints "Hello World #" where # is the thread number .
13
14 */
15
```

```

16 int main(int argc, char* argv[])
17 {
18
19 // OpenMP does parallelism through the use of threads.
20 // #pragma are compiler directives that are handled by OpenMP.
21 // The directive applies to the next simple or compound statement.
22 // By default, the number of threads created will correspond to
23 // the number of cores in the computer.
24 #pragma omp parallel
25
26 // This is the simple statement that will be done in parallel threads.
27 printf("Hello World! I am thread %d out of %d\n",
28        omp_get_thread_num(), omp_get_num_threads());
29 printf("All done with the parallel code.\n");
30
31 return 0;
32 }
```

A collection of new functions designed for creating and managing threads will be introduced. The header file containing the declarations of these functions is included on Line 2.

Parallelism using OpenMP is handled by using compiler directives that begin with `#pragma omp` as on Line 24. In this case, the line

```
24 #pragma omp parallel
```

means that the next *statement* of the program will be done in parallel by the default number of threads. Sample output of this example is shown here:

```

Hello World! I am thread 4 out of 8
Hello World! I am thread 6 out of 8
Hello World! I am thread 5 out of 8
Hello World! I am thread 1 out of 8
Hello World! I am thread 0 out of 8
Hello World! I am thread 7 out of 8
Hello World! I am thread 3 out of 8
Hello World! I am thread 2 out of 8
All done with the parallel code.
```

Note how the output is not sequential in terms of the printing of each thread. This illustrates how the threads are executed in parallel but the threads may not execute in sequential order without specific instructions to do so.

Each of the threads has a thread ID so that the behavior of specific threads can be controlled. The thread ID is obtained by calling the function `omp_get_thread_num`, and the function `omp_get_num_threads` returns the total number of threads currently in use. These functions are on Line 28 of the example and are what allow the thread numbers to appear in the output.

As noted above, the compiler directive on Line 24 says that the following *statement* will be executed in parallel. In this example, the statement is the single `printf` call on Line 27. The second call to `printf` is not done in parallel because it is not part of the single statement that is parallelized. The word *statement* is emphasized because in C, a statement is not necessarily limited to a single line of code but can be many lines of code contained within “{}” braces. For example, if Example 9.1 were modified as below, then both `printf` calls would be done in parallel:

```

1 #pragma omp parallel
2
3 {
4     printf("Hello World! I am thread %d out of %d\n",
5            omp_get_thread_num(), omp_get_num_threads());
6     printf("All done with the parallel code.\n");
7 }
```

In this case, the output would contain eight copies of the line “All done with the parallel code.”

Finally, before getting into a more refined discussion about the OpenMP system, note that building the executable is also very easy. First, when compiling code, the language option -fopenmp is added:

```
$ gcc -c hello.c -fopenmp
```

This enables the extra compiler directives such as on Line 24 of Example 9.1. Second, the OpenMP library is linked in using -lgomp when making the executable:

```
$ gcc -o hello hello.o -lgomp
```

The parallelism in Example 9.1, while easy to implement, is not really that helpful in the context of scientific computing because it just duplicated the same computation rather than divide the computation to break a big problem into smaller ones. In order to put OpenMP to work, more refined control is necessary.

The #pragma lines in OpenMP can be modified by using additional clauses to better control how the parallelism is implemented. Clauses can be applied to modify the initial #pragma omp parallel line and also can be used to modify the other #pragma omp directives to be discussed later. In the remainder of this chapter, the clauses used for the #pragma omp parallel directive are examined.

## 9.2 • The num\_threads Clause

The number of threads used in a particular parallel region can be controlled with the num\_threads clause. For the output of Example 9.1, the default number of threads was eight. To limit the number of threads to four, the num\_threads clause is added to Line 24:

```
24 #pragma omp parallel num_threads(4)
```

The resulting output is

```
Hello World! I am thread 1 out of 4
Hello World! I am thread 0 out of 4
Hello World! I am thread 3 out of 4
Hello World! I am thread 2 out of 4
All done with the parallel code.
```

The maximum available number of threads can be obtained using the function omp\_get\_max\_threads. Alternatively, in the Linux environment, the command nproc --all will give the same answer. The maximum number of threads is implementation dependent but typically is equal to a multiple of the number of available cores on the

computer. The examples in this chapter are being run on a system with four cores, and there are a maximum of eight threads available for OpenMP.

## 9.3 • The shared Clause

When doing parallel computing on a shared memory architecture, it's important to make sure that the various threads understand which variables are safe to access at the same time and which variables need to be made unique for their own thread. For example, when entering a parallel region, the length of an array is something that may not change and every thread needs to know, so it makes sense to have the length be a shared variable. To illustrate the use of shared and private variables, consider the following simple program that counts the number of times certain integers appear in an input list:

### Example 9.2.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <omp.h>
4
5 /*
6  int main(int argc, char* argv[])
7
8   computes the number of instances of a given index
9
10  Inputs: input array of integers
11
12  Outputs: Prints the count of integers that match the thread ID
13
14 */
15
16 int main(int argc, char* argv[])
17 {
18 // number of arguments is the length of the list except for argv[0]
19 int N = argc - 1;
20 int a[N];
21 int i;
22 // convert the arguments into an array of ints
23 for (i=0; i<N; ++i)
24     a[i] = atoi(argv[i+1]);
25 int count = 0;
26
27 #pragma omp parallel num_threads(4) shared(N)
28 {
29 // each thread counts how many inputs match their thread id
30     for (i=0; i<N; ++i)
31         count += a[i] == omp_get_thread_num() ? 1 : 0;
32     printf("The number %d appears in the list %d time(s).\n",
33                                     omp_get_thread_num(), count);
34 }
35
36 return 0;
37 }
```

Running this example with the input

```
$ ./shared 0 1 2 3 1 5 8 2 3 1
```

produces the output

```
The number 1 appears in the list 3 time(s).
The number 3 appears in the list 2 time(s).
The number 0 appears in the list 1 time(s).
The number 2 appears in the list 2 time(s).
```

Note the variable `N` is declared a shared variable by using the `shared(N)` clause on Line 27. It is safe to have `N` be shared because while each thread uses the value of `N`, it doesn't modify it. But there's another variable, `count`, for which it was not specified whether it is shared. It is strongly recommended that all variables defined outside a parallel region, and used inside the parallel region, are specified to be either shared or private using clauses. To see what can happen if one is not careful, suppose the variable `count` is designated to be shared by changing Line 27 to be

```
27 #pragma omp parallel num_threads(4) shared(N, count)
```

This results in the following output:

```
The number 1 appears in the list 5 time(s).
The number 3 appears in the list 2 time(s).
The number 0 appears in the list 6 time(s).
The number 2 appears in the list 8 time(s).
```

The results are completely wrong because by declaring the variable `count` to be shared, when `count` is incremented on Line 31, it is incremented for *all* the threads.

There are lessons to be drawn from this example. First, while it worked out well to let OpenMP assume that `count` is not shared, it is much safer to specify the status for every variable used in the parallel region to avoid unexpected or incorrect results. Variables that are not modified are good for shared variables, while variables that are modified within each thread should not be shared.

## 9.4 • The private Clause

In the context of OpenMP, the opposite of shared is private. If Line 27 is modified to be

```
27 #pragma omp parallel num_threads(4) shared(N) private(count)
```

the results are again wrong:

```
The number 1 appears in the list 1 time(s).
The number 3 appears in the list 1 time(s).
The number 0 appears in the list 1 time(s).
The number 2 appears in the list 1 time(s).
```

There is a simple fix, but first consider what happens when a thread gets a private variable. When a variable is declared private, a separate memory address is set aside for that variable within each thread. In this case, in the parallel region, there are four separate variables called `count`, one for each thread. That memory is created new and uninitialized, so that means that the initial value of `count` is unpredictable. One remedy for this problem is to initialize the value of `count` inside the parallel region so that each thread initializes its own instance of `count`:

```

27 #pragma omp parallel num_threads(4) shared(N) private(count)
28 {
29     count = 0;
30     for (i=0; i<N; ++i)
31         count += a[i] == omp_get_thread_num() ? 1 : 0;
32     printf("The number %d appears in the list %d time(s).\n",
33                                     omp_get_thread_num(), count);
34 }
```

For this simple example, this is probably the best solution, but in more complex situations, use the **firstprivate** clause instead:

```

27 #pragma omp parallel num_threads(4) shared(N) firstprivate(count)
28 {
29     for (i=0; i<N; ++i)
30         count += a[i] == omp_get_thread_num() ? 1 : 0;
31     printf("The number %d appears in the list %d time(s).\n",
32                                     omp_get_thread_num(), count);
33 }
```

By using the **firstprivate** clause, the variable **count** is declared private *and* it also instructs each thread to initialize the variable with the value of **count** when the parallel region was entered. Thus, by initializing **count** to be zero on Line 25 of Example 9.2 and using the **firstprivate** clause, each thread properly initializes **count** to be zero.

As a cautionary tale of why all variables should be declared either shared or private, note that the two other variables in Example 9.2: **i**, **a** are also not specified. Try running this example with different uses of the **shared**, **private**, and **firstprivate** clauses for the variables **i** and **a** to see which combinations produce the correct results.

Since private variables have a unique instance for each thread, what happens to the various values of **count** at the end of the parallel region? To answer that, print the value of **count** after the parallel region, and it should come as no surprise that the value reported is zero. That is because when the parallel region is completed, the various private instances of **count**, which are distinct from the one declared on Line 25, are discarded, and hence the variable **count** in the scope of the main program, which was unchanged by the work done in the parallel region, remains set at zero. The next clause is one way in which the multiple private variables can be recombined.

## 9.5 • The reduction Clause

As in the previous section, the fate of private variables at the end of the parallel region is that they are discarded. However, the **reduction** clause is one way in which those various private variables can be salvaged from the parallel region. If the variable **count** is made private using the **reduction** clause, then all the private instances of **count** can be combined by adding them up and storing them in the **count** variable in the main scope outside the parallel region. The modified code is

```

27 #pragma omp parallel num_threads(4) shared(N) reduction(:count)
28 {
29     for (i=0; i<N; ++i)
30         count += a[i] == omp_get_thread_num() ? 1 : 0;
31     printf("The number %d appears in the list %d time(s).\n",
32                                     omp_get_thread_num(), count);
33 }
34 printf("Total of all found numbers is %d\n", count);
```

**Table 9.1.** List of operators available for the reduction clause. The “-” operator in the reduction clause is equivalent to using the + operator and multiplying by  $-1$  at the end. The logical “**&&**” operator returns true if the variable is true in all the threads, while the logical “**||**” operator returns true if the variable is true in at least one of the threads.

Operator	Description
+	Sum
*	Product
-	Negative sum
<b>&amp;&amp;</b>	Logical and
<b>  </b>	Logical or
<b>max</b>	Maximum
<b>min</b>	Minimum

Running the program with the input

```
$ ./shared 0 1 2 3 1 5 8 2 3 1
```

produces the output

```
The number 1 appears in the list 3 time(s).
The number 3 appears in the list 2 time(s).
The number 0 appears in the list 1 time(s).
The number 2 appears in the list 2 time(s).
Total of all found numbers is 8.
```

The clause `reduction(+:count)` tells the compiler to take all the final values of the private variable `count`, sum them up, and then store the result in the `count` variable from the outer scope. If multiple variables are to be combined using the reduction clause and using the same operator, then they can be added in a list, e.g., `reduction(+:count, anothercount)`. Alternatively, multiple clauses can be used, such as

```
#pragma omp parallel reduction (+:count) reduction (*:product)
```

The most common operators used in the reduction clause are listed in Table 9.1.

## Exercise

- 9.1. Write a program to compute the sum of  $2^N$  random values generated using `drand48`, where  $N$  is specified as an input. Seed the random number generator with a shared seed and then each thread adds its thread ID to the seed. Use the `reduction()` clause to accumulate the results. Verify that each thread generates a unique sequence of numbers and that the final result is approximately  $2^{N-1}$ . Explicitly declare each variable as either shared or private.



## Chapter 10

# Subdividing for Loops

A large part of scientific computing is manipulating large arrays of data, and this is one way in which OpenMP excels because it is an embarrassingly simple task to break a `for` loop into smaller pieces and do the task in parallel. Much of the discussion in Chapter 9 about shared versus private variables carries over to this chapter as well, but there are some additional nuances when dealing with variables within the loop. In this chapter, we introduce the compiler directive `#pragma omp for` that is used to subdivide `for` loops. Private variables and how they interact with the code region outside the loop will be revisited when there are additional nuances to consider. Finally, we will look at more refined control of the subdivision in terms of how the loop is subdivided and whether program control can continue safely without waiting for the other threads. These issues are important for doing load balancing, where ideally each thread completes an equal amount of work so that no thread remains idle for long.

### 10.1 • Subdividing Loops: `#pragma omp for`

To illustrate the simplicity of parallelizing a loop, consider Example 10.1, where the task is to initialize an array with some data.

#### Example 10.1.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <omp.h> // This is the header file for OpenMP directives
4 /*
5  * 
6  * int main(int argc, char* argv[])
7  *
8  * The main program illustrates how to divide a loop into smaller
9  * pieces, each handled in parallel by different threads.
10 */
11 Inputs: argc should be 2
12      argv[1] is the length of the array
13
14 Outputs: Initializes an array in parallel
15 */
16 */
17
18 int main(int argc, char* argv[])
```

```

19 {
20 // Get the length of the array
21 int N = atoi(argv[1]);
22
23 // Allocate the memory for the array
24 double* u = (double*) malloc(N*sizeof(double));
25
26 int i;
27 // Initialize an array of length N
28 #pragma omp parallel shared(u,N) private(i)
29 {
30 #pragma omp for
31     for (i=0; i<N; ++i) {
32         printf("u[%d] is set by thread %d\n", i, omp_get_thread_num());
33         u[i] = (double)i;
34     }
35 }
36
37 return 0;
38 }
```

Sample output for Example 10.1 is shown below for the case of  $N = 16$ :

```

u[10] is set by thread 5
u[4] is set by thread 2
u[5] is set by thread 2
u[14] is set by thread 7
u[15] is set by thread 7
u[0] is set by thread 0
u[1] is set by thread 0
u[8] is set by thread 4
u[9] is set by thread 4
u[11] is set by thread 5
u[12] is set by thread 6
u[13] is set by thread 6
u[2] is set by thread 1
u[3] is set by thread 1
u[6] is set by thread 3
u[7] is set by thread 3
```

The `for` loop is broken down into smaller chunks by using the `#pragma omp for` directive *within* the section to be parallelized, i.e., inside the section of code contained in the “{}” brackets after the `#pragma omp parallel` directive. In this example, the default number of eight threads was created, and each thread initialized 2 of the 16 entries in the array, all in parallel. Notice how this was accomplished without having to do anything more than add a couple of compiler directives, and in fact, if the compiler directives are ignored, then the program is still written in exactly the way it would be done on a serial computer. This is a key appeal of OpenMP—that very little additional code must be written to achieve a parallel implementation and that the code is largely exactly the same had the parallelism not been added.

Note how in this example, the array `u` and the integer `N` are designated as shared variables. The variable `N` is safe to share because the value will not be changed in the parallel code, and it is more efficient to just have one location in memory be used to

store the value. The array *u* is going to be changed, but the key is that each iteration in the loop is completely independent of the other iterations and will only change a unique subset of the array values, so it is safe to do the assignment statements in parallel using shared memory for *u*.

By contrast, the variable *i* is set to be private. It would seem to not matter, but in fact it does because each thread will be updating *i* for its own part of the loop. For the sample output above, thread 0 used *i* for value 0 initially, and then incremented it to 1 in order to set the values for *u[0]*, *u[1]*. Before that, thread 7 set *i* to 14 and then 15. There would then be a risk that if thread 0 and thread 7 were working simultaneously using the shared variable *i*, then thread 0 may set *i* to 1, and before it can do the assignment statement, thread 7 sets the value of *i* to 15 resulting in thread 0 setting *u[1]* to 15. To avoid this conflict, each thread will have its own private version of *i* so that the different iterations can safely assign values to *i* that won't conflict.

The `#pragma omp for` directive does not work for all loops. For example, a loop for which the number of iterations cannot be determined a priori cannot be parallelized as in this simple search loop:

```

1 #pragma omp for
2   for (i=0; u[i] != 3; ++i) {
3     printf("Thread %d searched for the number 3 in u[%d]\n",
4           omp_get_thread_num(), i);
5 }
```

In general, the `for` loop has to be of the form

```

1   for (var=start_value; var<end_value; ++var) { ... }
```

where the termination condition must be one of `<`, `>`, `<=`, or `>=`. The increment must use one of these methods: `++var`, `-var`, `var+=incr`, `var-=incr`, `var=var+incr`, or `var=var-incr`. While this may seem like a tight restriction, the truth is that for scientific programming it is most often the case that the number of iterations is known ahead of time and can use loops of this sort. The compiler will fail if the conditions for parallelizing the loop are not met.

Before moving on to the clauses, the construct of setting up a parallel region in order to subdivide a `for` loop is so common that there is shorthand for it. The two compiler directives can be combined into one so that Lines 28–35 become

```

28 #pragma omp parallel for shared(u,N) private(i)
29   for (i=0; i<N; ++i) {
30     printf("u[%d] is set by thread %d\n", i, omp_get_thread_num());
31     u[i] = (double)i;
32 }
```

## 10.2 • The private Clause

Similar to the `#pragma omp parallel` directive, clauses can also be applied to other directives such as `#pragma omp for`. One example is the use of the `private` clause on Line 28 in Example 10.1. That part could have been rewritten this way to get the same result:

```

27 #pragma omp parallel shared(u,N)
28 {
29   #pragma omp for private(i)
```

```

30   for (i=0; i<N; ++i) {
31     printf("u[%d] is set by thread %d\n", i, omp_get_thread_num());
32     u[i] = (double)i;
33   }
34 }
```

where the `private(i)` clause has been moved to the `for` loop directive. The difference is that in the latter case, the variable `i` will be private only inside the `for` loop.

The use of the `private` and `firstprivate` clauses is the same when applied to the `#pragma omp for` directive as for the `#pragma omp parallel` directive, so the reader is referred to Section 9.4. However, the `#pragma omp for` directive permits an additional `lastprivate` clause for handling private variables. In this case, the value the variable would have at the conclusion of the loop *if done serially* is assigned to the variable listed in the `lastprivate` clause at the conclusion of the `for` loop. For example, consider the following loop where the variable `k` is taken to be private:

```

27 int k = -1;
28 #pragma omp parallel shared(u,N) num_threads(2)
29 {
30 #pragma omp for private(i,k)
31   for (i = 0; i < N; ++i) {
32     printf("On entry: k = %d in thread %d\n", k,
33                                     omp_get_thread_num());
34     k = i;
35     printf("On exit: k = %d in thread %d\n", k, omp_get_thread_num());
36   }
37 }
38 printf("Final k = %d\n", k);
```

The output of this loop for the case `N = 2` is

```

On entry: k = 0 in thread 1
On exit: k = 1 in thread 1
On entry: k = 32765 in thread 0
On exit: k = 0 in thread 0
Final k = -1
```

Recall from the discussion about private variables in Section 9.4 that the value of `k` reverted to the value it had before the loop because the private `k` variables in each thread are separate from the one created outside the parallel region. However, if Line 30 is changed to

```
30 #pragma omp for private(i) lastprivate(k)
```

then the last line of the output is changed to

```
Final k = 1
```

because in the loop, the last time the loop would be executed if it were done in serial would be for the case `i = 1` (because `N = 2`). Note that the final value is according to the last value of `k` as if the loop had not been run in parallel and does not necessarily correspond to the order in which the threads are executed.

## 10.3 • The reduction Clause

The reduction clause for the `#pragma omp for` directive is very similar to the use in Section 9.5 but is reviewed again here to avoid some confusion in the usage.

### Example 10.2.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <omp.h>
4
5 /*
6   int main(int argc, char* argv[])
7
8   computes the number of instances of a given index
9
10  Inputs: argc should be >= 2
11    argv[*] input array of integers
12
13  Outputs: Prints the sum of the integers
14
15 */
16
17 int main(int argc, char* argv[])
18 {
19   // Convert the arguments into the input array
20   int N = argc - 1;
21   int a[N];
22   int i;
23   for (i=0; i<N; ++i)
24     a[i] = atoi(argv[i+1]);
25   int sum = 0;
26
27 #pragma omp parallel num_threads(4) shared(N)
28 {
29 #pragma omp for private(i) reduction(+:sum)
30   for (i=0; i<N; ++i)
31     sum += a[i];
32   }
33   printf("The sum is %d\n", sum);
34
35   return 0;
36 }
```

In Example 10.2, each of the threads will add a subset of the entries in the array `a`. When the `for` loop is completed, each of the private `sum` variables contains a partial sum of the array. Through the `reduction` clause, they will be added together to compute the sum for the full array. In this way, it operates in much the same way as discussed earlier, but it's somewhat hidden that the loop itself has been broken into smaller pieces through the `#pragma omp for` directive.

## 10.4 • The ordered Clause

Recall that the output from Example 10.1 was printed in a seemingly random order. The threads can be coerced into working in sequential order. To do this, there are two changes required as illustrated below:

```

27 #pragma omp parallel shared(u,N) private(i)
28 {
```

```

29 #pragma omp for ordered
30   for (i=0; i<N; ++i) {
31 #pragma omp ordered
32   {
33     printf("u[%d] is set by thread %d\n", i, omp_get_thread_num());
34   }
35   u[ i ] = (double)i;
36 }
37 }
```

The `ordered` clause is added to Line 29 to indicate that there will be some part of the loop that will be required to run in sequential order. Within that loop, the parts that will be run in order need to be identified using the `#pragma omp ordered` directive. In this example, the `printf` statements are to be done in order so that the output is the same as if it were run in serial; thus Line 31 has been inserted. Like other `#pragma omp` directives, it applies to the next statement. Thus, the “`{}`” braces are optional in this case because it was only meant to apply to the `printf` function. On the other hand, if the assignment statement was meant to be sequential as well, then it could be inserted inside the “`{}`” braces to force them to be sequential. Of course, that would then risk defeating the purpose of parallelism. The more constraints placed on the computation, the less efficient the resulting code may be. Making the changes as shown for Lines 27–37 results in the output expected:

```

u[0] is set by thread 0
u[1] is set by thread 0
u[2] is set by thread 1
u[3] is set by thread 1
u[4] is set by thread 2
u[5] is set by thread 2
u[6] is set by thread 3
u[7] is set by thread 3
u[8] is set by thread 4
u[9] is set by thread 4
u[10] is set by thread 5
u[11] is set by thread 5
u[12] is set by thread 6
u[13] is set by thread 6
u[14] is set by thread 7
u[15] is set by thread 7
```

## 10.5 • The schedule Clause

The strategy for how OpenMP will assign threads to the various iterations of a loop can be modified to some extent using the `schedule` clause. For the simple loops seen so far, the default strategy called `static` is ideal. There is the least amount of overhead for setting up the threads, and the iterations are handed out in small chunks to try to keep all threads occupied. More complicated loops, particularly loops where the time to complete an iteration may be more variable, may benefit from altering the thread scheduling strategy. To illustrate the effects of various strategies, consider Example 10.3, where an artificial delay is set in odd iterations of the loop. The output of this program can be

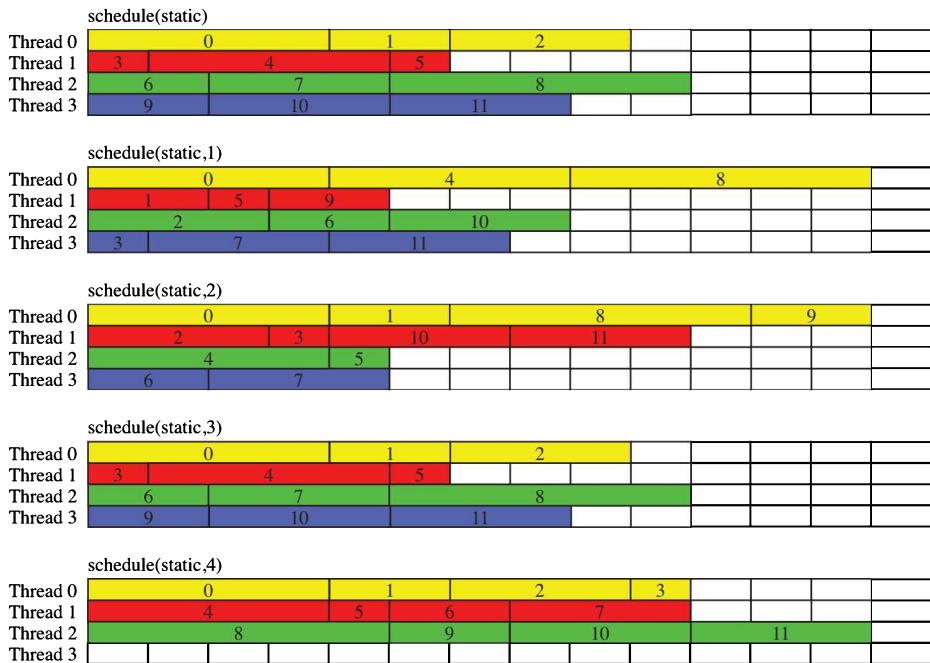
used to represent graphically how the threads are allocated and the total time used to complete the loop.

### Example 10.3.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <omp.h>
5
6 /*
7   int main(int argc, char* argv[])
8
9   The main program illustrates how iterations are assigned to
10  threads according to the scheduling strategy
11
12  Inputs: argc should be 3
13    argv[1]: Number of iterations in the loop
14    argv[2]: Block size used in the schedule clause
15
16  Outputs: Time points when each iteration begins/ends
17 */
18
19 int main(int argc, char* argv[])
20 {
21   // Get the number of iterations and the block size from the arguments.
22   int N = atoi(argv[1]);
23   int bsize = atoi(argv[2]);
24
25   // start the timer
26   double t0 = omp_get_wtime();
27   double t1, t2;
28   int duration[N];
29
30   int i;
31   for (i=0; i<N; ++i) duration[i] = 1+rand()%5;
32 #pragma omp parallel shared(N) private(i,t1,t2) num_threads(4)
33 {
34 #pragma omp for schedule(static ,bsize)
35   for (i=0; i<N; ++i) {
36     // report time of random sleep duration
37     t1 = omp_get_wtime();
38     sleep(duration[i]);
39     t2 = omp_get_wtime();
40     printf("Iteration %d: Thread %d, started %e, duration %e\n",
41           i, omp_get_thread_num(), (double)(t1-t0), (double)(t2-t1));
42   }
43 }
44
45 // stop the timer
46 t2 = omp_get_wtime();
47 printf("CPU time used: %.2f sec\n", t2-t0);
48 return 0;
49 }
```

One new function introduced here is the function `omp_get_wtime`, which returns the current wall clock time from the system. The methods discussed in Section 6.3 will not work as expected when creating a multithreaded code because many of those tools measure the performance of whatever thread is measuring the time. When multiple threads are used, the work done by the other threads may not be measured appropriately.



**Figure 10.1.** Chart of threads executing iterations in the loop in Example 10.3 for  $N = 12$  iterations and four threads. Time required for each iteration was chosen at random, the same for all cases. Static scheduling with various block sizes are illustrated. Optimal case is for block size 3 in this example.

For OpenMP, there is a better solution, namely the function

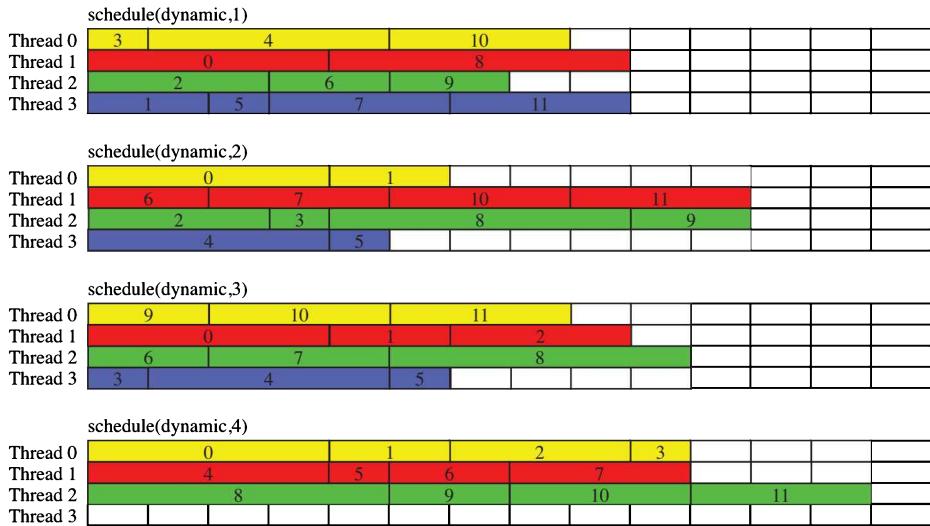
```
double omp_get_wtime ( void )
```

The value returned by a single call to this function is not very meaningful; it gives the time since some fixed time in the past. Thus, to measure the performance of some code, this function is called twice, once at the beginning of the code and once at the end. The difference between the two calls will measure the number of seconds, in wall clock time, to complete the code between the two calls.

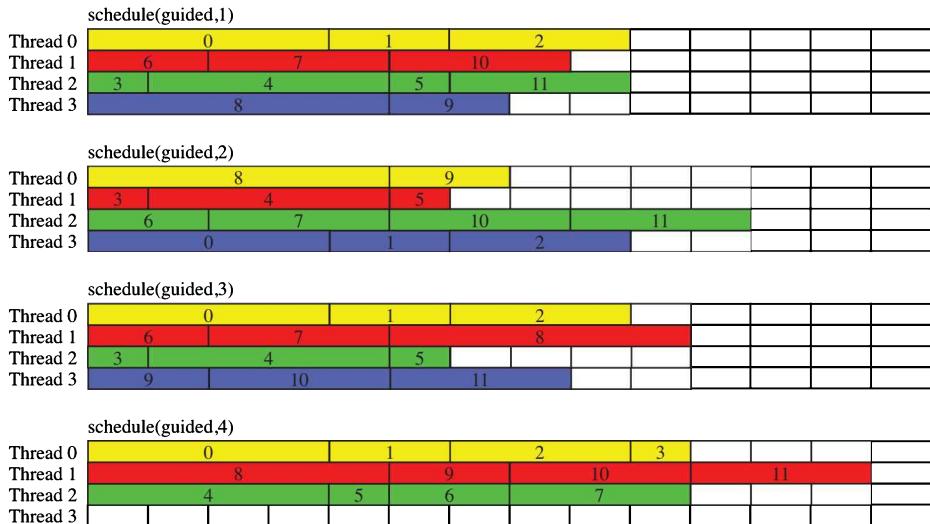
Another new function introduced in this example is the `sleep` function, which pauses the program for the number of seconds given in the input. Thus, beginning with Line 37, the current time is sampled, the loop is paused for a randomly chosen predetermined length of time, and then the time is sampled again on Line 39. This example illustrates what happens when different iterations have different durations.

Figures 10.1–10.3 illustrate how the sequence of iterations are blocked out among the available four threads. On Line 34, the scheduling of threads is set to be `static` with block size `bsize`. When using the `static` schedule, the iterations are assigned in order to the available threads in groups of size `bsize`. The block size parameter is optional and can be left to the default, which in this example would be the number of iterations divided by the number of threads. Thus, the example of `schedule(static, 3)` is equivalent to the default value of the block size. It is independent of the time required to complete each iteration.

Figure 10.1 shows what happens when `static` scheduling is used for block sizes of 1, 2, 3, and 4 and varying times for each iteration. For block size 1, the iterations



**Figure 10.2.** Chart of threads executing iterations in the loop in Example 10.3 for  $N = 12$  iterations and four threads. Time required for each iteration was chosen at random, the same for all cases. Dynamic scheduling with various block sizes are illustrated. Optimal case is for block sizes 1 in this example.



**Figure 10.3.** Chart of threads executing iterations in the loop in Example 10.3 for  $N = 12$  iterations and four threads. Time required for each iteration was chosen at random, the same for all cases. Guided scheduling with various block sizes are illustrated. Optimal case is for block size 1 in this example.

are dealt to the threads in sequential order one at a time. Thus, thread 0 starts with iteration 0 and will continue with every fourth iteration thereafter (because there are four threads allocated on Line 32). Similarly, thread 1 will start with iteration 1 and then continue with every fourth iteration, and so forth for the other two threads. Since it's not known a priori how long each iteration will take to run, using a static scheduling

scheme is problematic if one particular thread happens to disproportionately get longer tasks. Thus, for the case of block size 1, thread 0 got three longer iterations, while thread 1 got three shorter ones. Since the loop can't finish until *all* threads finish, the loop is limited to the case when thread 0 finishes.

This example illustrates how static scheduling is ideal when the time required to complete each iteration is roughly the same but may be less than optimal when iterations vary in execution time.

To handle the case of variable length iterations, an alternative scheduling strategy is **dynamic**, which is put in place of **static** on Line 34 of the example code. In dynamic scheduling, iterations are dealt, in groups of block size, to threads as they become available. This means that the ordering is independent of the thread number. The results for dynamic scheduling, for the very same variable length iterations, are shown in Figure 10.2. The example with block size 1 is a good illustration of the advantage of dynamic scheduling in this case. Initially, the first four iterations are given to the first four threads in the order that they are ready. In this example, thread 0 was assigned iteration 3, which happens to be the shortest iteration. Since it finishes first, it is given the next available iteration, 4. Thread 3 then finishes its first iteration, so it gets iteration 5, and so on. By using this dynamic scheduling, the loop finished 4 seconds earlier than the corresponding statically scheduled loop shown in Figure 10.1.

As for the static scheduling, the iterations are doled out in groups of block size. For dynamic scheduling, the default is block size 1, which coincidentally produced the most efficient results. So if block size 1 is clearly the most efficient on paper, why does it make sense to use any other size? There is additional overhead cost to making block sizes smaller. For this simple example, where the execution time of the individual iterations far exceeds the time required to manage the multiple threads, it is clear that using block size 1 is optimal. But when the number of iterations gets large, and the execution time of individual iterations is small, which is much more common in scientific computing applications, the overhead cost of small block sizes can add up and cause performance degradation. It can be worth the effort to experiment with different block sizes to see what sizes produce the most efficient code.

The same argument about overhead applies to the choice between static and dynamic scheduling. Dynamic scheduling incurs a cost related to monitoring threads as they complete their tasks, then assigning iterations to the available threads. In static scheduling, the threads know from the beginning which iterations they will implement, and hence no additional thread monitoring overhead is required. Consequently, it is most efficient to use static scheduling when the execution times of the iterations are well balanced, while it can be helpful to use dynamic scheduling when the iterations are not well balanced.

The final scheduling strategy is **guided**, which is just the dynamic strategy with one change for block sizes bigger than one. For guided scheduling, when the number of remaining iterations gets small, the block sizes are scaled accordingly so that load balancing is better at the tail end of the loop. This difference can be seen when comparing the case of block size 2 in Figures 10.2 and 10.3. For dynamic scheduling with block size 2, iterations are doled out in groups of size 2 regardless of how many iterations remain. Thus, for the case of `schedule(dynamic, 2)`, iterations 10 and 11 are both assigned to thread 1 even though threads 0 and 3 are available to execute iteration 11 while thread 1 finishes iteration 10. For guided scheduling, when the number of remaining iterations dwindles, the block size also shrinks, and hence threads 1 and 3 have an odd number of iterations even though the block size is 2.

Again, this additional level of adaptive scheduling will incur an additional overhead cost, so it would be used only in instances where there are a significant number of idle threads toward the end of a loop when using larger block sizes.

For most scientific computing applications, static scheduling with large block sizes is often the most efficient. It is also advantageous to plan for loops with a number of iterations that are multiples of the number of threads times the block size so that the threads are scheduled optimally, and the number of idle threads at the end is kept to a minimum.

## 10.6 • The nowait Clause

In the discussion about scheduling threads, it should be clear that there is a potential for idle threads and that idle threads lead to lower computational efficiency. Another way in which idle threads can be harnessed through scheduling is by using the `nowait` clause. To see how it works, a second parallel `for` loop is added to Example 10.3:

### Example 10.4.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <omp.h>
5
6 /*
7  int main(int argc, char* argv[])
8
9  The main program illustrates how iterations are assigned to
10 threads according to the scheduling strategy
11
12 Inputs: argc should be 3
13      argv[1]: Number of iterations in the loop
14      argv[2]: Block size used in the schedule clause
15
16 Outputs: Time points when each iteration begins/ends
17 */
18
19 int main(int argc, char* argv[])
20 {
21     // Convert the args to the integer parameters
22     int N = atoi(argv[1]);
23     int bsize = atoi(argv[2]);
24
25     // Start the timer
26     double t0 = omp_get_wtime();
27     double t1, t2;
28     int duration[N];
29
30     int i;
31     for (i=0; i<2*N; ++i) duration[i] = 1+rand()%5;
32 #pragma omp parallel shared(N) private(i,t1,t2) num_threads(4)
33 {
34 #pragma omp for schedule(static,bsize) nowait
35     for (i=0; i<N; ++i) {
36         // report time of random sleep duration
37         t1 = omp_get_wtime();
38         sleep(duration[i]);
39         t2 = omp_get_wtime();
40         printf("Loop 1: Iteration %d: Thread %d, start %e, duration %e\n",
41               i, omp_get_thread_num(), (double)(t1-t0), (double)(t2-t1));
42 }
```

```

42 }
43
44 #pragma omp for schedule(static ,bsize)
45 for (i=0; i<N; ++i) {
46     // report time of random sleep duration
47     t1 = omp_get_wtime();
48     sleep(duration[N+i]);
49     t2 = omp_get_wtime();
50     printf("Loop 2: Iteration %d: Thread %e, start %e, duration %e\n",
51            i, omp_get_thread_num(), (double)(t1-t0), (double)(t2-t1));
52 }
53
54
55 // stop the timer
56 t2 = omp_get_wtime();
57 printf("CPU time used: %.2f sec\n", t2-t0);
58 return 0;
59 }
```

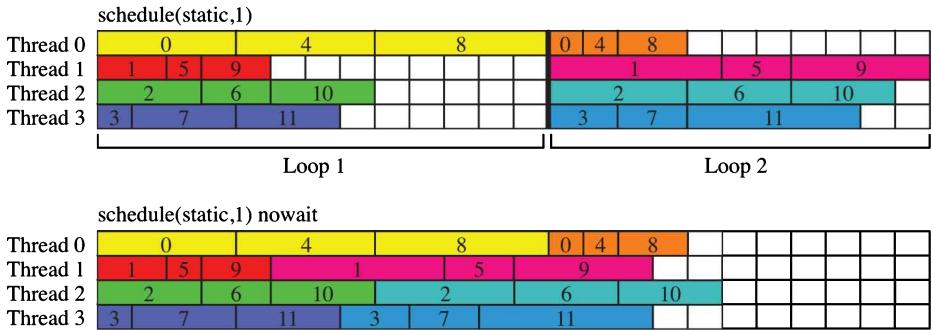
When the various OpenMP directives are given in the code, there is an implied barrier at the end of the affected code where all threads pause until the section is complete. Thus, in the previous Example 10.3, there are barriers at Lines 42, 43 where the execution pauses to wait for all threads to complete the `for` loop and the parallel region, respectively. In Example 10.4, the implied barrier at Line 42 is removed by adding the `nowait` clause on Line 34. That means that when a given thread completes its iterations in the first loop, it is free to continue by doing its assigned iterations in the second loop without waiting for other threads to do the same. Using the same graphical representation of the thread activity, Figure 10.4 illustrates how using the `nowait` clause can reduce the amount of time wasted in idle threads.

The `nowait` clause must be used judiciously, however, because should the second loop have any dependencies based on the calculation in the first loop, erroneous results can arise. For example, suppose the calculation in iteration 1 of the second loop depends upon iteration 8 of the first loop. Without the `nowait` clause, iteration 1 will not begin until iteration 8 is completed thanks to the implied barrier at the end of the first loop. With the `nowait` clause, thread 1 will start iteration 1 of the second loop *before* thread 0 begins iteration 8 of the first loop. Thus, thread 1 will use incomplete or nonexistent results from iteration 8 of the first loop, leading to unexpected results.

## 10.7 • Efficiency Measures

The purpose of using multiple threads is to take advantage of the multicore capabilities of modern processors so that a large problem can be broken down into smaller problems that can be computed in parallel. Ideally, if using twice as many threads, the job should be done in half the time. Unfortunately, that's not entirely possible because adding threads also incurs overhead costs that are not present for a single-threaded program.

There are two fundamental measures for efficiency of parallel algorithms commonly used: strong scaling and weak scaling. Strong scaling measures how much time is required to complete a fixed-size problem compared to the number of threads used. A perfectly efficient algorithm is strongly scalable if a problem that takes time  $T$  for one thread requires time  $T/N$  for  $N$  threads. Weak scaling is a measure of how the time to completion changes when the size of the problem grows with the number of threads. In this case, suppose a problem has size  $M$ , where, for example,  $M$  could be the number



**Figure 10.4.** Chart of threads executing iterations in the loop in Example 10.4 for  $N = 12$  iterations and four threads. Time required for each iteration was chosen at random, the same for all cases. Without the `nowait` clause, all threads wait at the barrier, indicated by the thick vertical line, until all threads finish the first loop. With the `nowait` clause, threads that finish early in the first loop can continue into the second loop.

of grid points when solving a PDE, or the number of entries in a matrix to be inverted. A perfectly efficient algorithm is weakly scalable if when the problem size increases by a factor, and the number of threads also increases by that factor, then the time to compute the solution should remain constant. For example, if the total number of grid points in a solver for a PDE is doubled, and the number of threads used to do the computation is doubled, does the time  $T$  remain the same?

To test for strong scaling for a given code that solves a problem, run it once with one thread, and then run it again with two threads, and see if the elapsed time is cut in half. To test this more systematically, collect data on the time elapsed for a fixed problem size solved for various numbers of threads. Let  $T_N$  be the time to completion using  $N$  threads. The strong scaling efficiency is then measured by

$$\text{strong efficiency} = \frac{T_1}{NT_N}.$$

Testing for weak scaling requires the problem to be able to change in size. For example, if solving a PDE problem, the measure of problem size could be the number of grid points used in the problem. Let  $T_1$  be the time to solve a problem of size  $M$  on a single thread, and let  $T_N$  be the time to solve a problem of size  $NM$  using  $N$  threads; then the weak scaling efficiency is measured by

$$\text{weak efficiency} = \frac{T_1}{T_N}.$$

For example, a PDE is solved on a three-dimensional  $M \times M \times M$  grid, hence  $M^3$  grid points, and the time required to solve the problem for a single thread is  $T_1$ . Now double the number of grid points in each dimension so that the grid has  $8M^3$  grid points and solve that problem on eight threads and let  $T_8$  be the time to solution. The weak efficiency would then be  $T_1/T_8$ .

To illustrate measuring the scaling properties of OpenMP, Example 10.5 is used as an example of solving a one-dimensional heat equation on the interval  $[0, 1]$ .

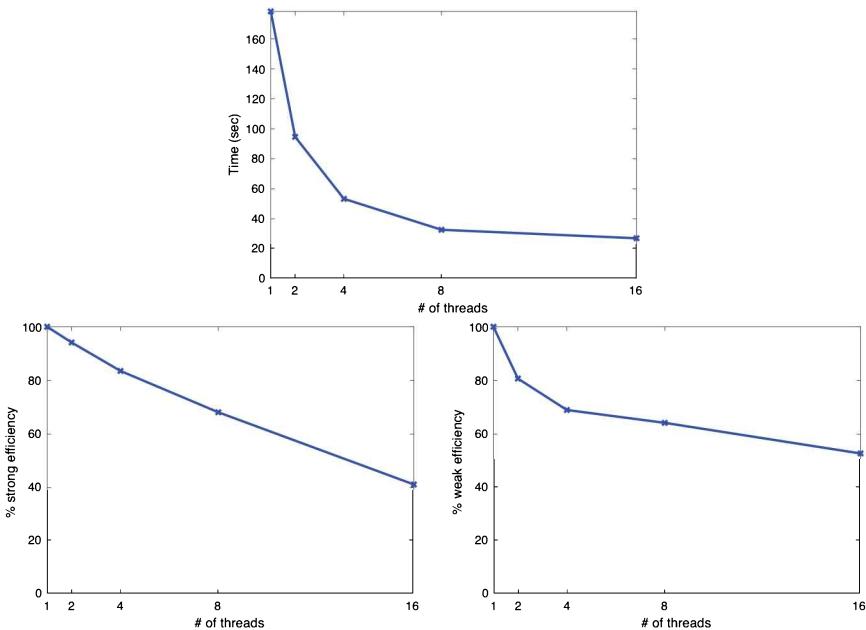
### **Example 10.5.**

```

62 // store the final array in binary format to file
63 FILE* fileid = fopen("finalu.dat", "w");
64 fwrite(u[(i+1)%2], sizeof(double), N, fileid);
65 fclose(fileid);
66
67 // clean up
68 free(u[0]);
69 free(u[1]);
70
71 // get the elapsed time
72 double timeelapsed = omp_get_wtime() - starttime;
73 printf("%d threads: Execution time = %le seconds.\n", P, timeelapsed);
74
75 return 0;
76
77 }

```

Figure 10.5 shows the strong and weak scaling results using Example 10.5. The computer used is capable of 8 parallel threads, so there is a marked degradation in the performance when 16 threads are requested because the extra threads cannot be done fully in parallel. Scaling is impacted because there are overhead costs for creating threads and there are parts of the code that cannot be done in parallel such as saving data to a file.



**Figure 10.5.** Top: Raw time measurements for Example 10.5 for  $N = 1600$  and 1, 2, 4, 8, and 16 threads. Bottom left: Strong efficiency for  $N = 1600$ . Bottom right: Weak efficiency for  $N = 100, 200, 400, 800$ , and 1600 grid points and 1, 2, 4, 8, and 16 threads. Time step size used for weak scaling was constant for all cases so that the same number of iterations were calculated.

## Exercises

- 10.1. Write a program to generate a two-dimensional  $N \times N$  grid of randomly generated values in the range  $[-2, 2]$ . Calculate the maximum value of each column in the grid and the maximum value for the whole grid. The program should also check whether any two values in the grid have the same value.
- 10.2. Write a program to construct a two-dimensional grid with data of the form  $\text{value}[i][j] = \cos(x_i) \sin(y_j)$ , where the  $x_i = \frac{2\pi i}{M}$ ,  $y_j = \frac{2\pi j}{N}$  for  $0 \leq i < M$ ,  $0 \leq j < N$ . The dimensions of the grid,  $M$  and  $N$ , should be read from the command line using `argv[1]`, `argv[2]`, respectively. The results should be saved to a binary data file with the  $M$  and  $N$  stored first, followed by the array data. This will require using a double loop; experiment with making either the inner loop or the outer loop parallel and compare the execution times to see which is faster. Experiment with different scheduling choices to get the optimal efficiency.

## Chapter 11

# Serial Tasks Inside Parallel Regions

Sometimes, a single thread is needed within a parallel region. There is computational overhead when creating parallel regions, so when serial code is mixed in with parallel regions, it may be more efficient to handle it with a single thread within the parallel region. This is accomplished by using the `#pragma omp single` directive. In this chapter, the use of single thread regions inside of a multithread region is explored. The interaction between the single thread and the multithread regions also requires attention to the storage status of variables, whether they are shared or private, so the nuances of shared and private variables will also be discussed when they differ from earlier discussions.

### 11.1 • Serial Subregions: `#pragma omp single`

Example 11.1 shows two uses of a single thread.

#### Example 11.1.

```
1 #include <stdio.h>
2 #include <omp.h>
3
4 /*
5   int main(int argc, char* argv[])
6
7   computes the number of positive numbers in each row and
8   the total sum
9
10  Inputs: none
11
12  Outputs: Prints the counts for each row and the total sum
13 */
14
15 int main(int argc, char* argv[])
16 {
17     int M;
18     int N;
19     int a[4][16] = {10, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
20                   6, 9, 10, 7, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
21                   2, 3, 4, 5, 6, 8, 1, 3, 3, 3, 9, 3, 6, 8, 6, 9,
22                   2, 9, 4, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
23     int sum = 0;
```

```

24 int count[4] = {0, 0, 0, 0};
25
26 #pragma omp parallel num_threads(4) shared(N,M,a,count) reduction(+:sum)
27 {
28     // initialize the shared variables, only one thread needed
29 #pragma omp single
30 {
31     printf("Thread %d is initializing.\n", omp_get_thread_num());
32     M = 4;
33     N = 16;
34 }
35
36 // This code will be run by each thread in parallel
37 int m = omp_get_thread_num();
38 int i;
39 for (i=0; i<N && a[m][i]!=0; ++i) {
40     ++count[m];
41     sum += a[m][i];
42 }
43
44 // use one thread to print the results
45 #pragma omp single
46 {
47     printf("Thread %d is reporting.\n", omp_get_thread_num());
48     for (i=0; i<M; ++i)
49         printf("Thread %d reports %d numbers.\n", i, count[i]);
50 }
51
52 printf("The sum is %d\n", sum);
53 return 0;
54 }
```

On Lines 29 and 45, the code inside the braces will be run by a single thread, while the code in Lines 37–42 is done in parallel, one for each of the four threads requested. An example of the output is shown below.

```

Thread 2 is initializing.
Thread 0 is reporting.
Thread 0 reports 2 numbers.
Thread 1 reports 4 numbers.
Thread 2 reports 16 numbers.
Thread 3 reports 0 numbers.
The sum is 139
```

The thread selected to run the single section is not specified; it takes the first available thread. In this instance, thread 2 handled the first single thread region, and thread 0 handled the last. The code in the first single thread region is a common use of the `#pragma omp single` directive, where shared variables are initialized by a single thread rather than redundantly having each thread do it.

The code in the last single thread region is there for illustrative purposes; we will see better ways to implement this later. However, before doing that, there is another important lesson to learn from the output. The exceptionally observant reader will notice that thread 3 reported finding zero numbers, even though when checking the array, there were three numbers before the first zero. Not only that, but the sum of all the numbers

appears to be correct, including the three seemingly uncounted numbers. How did this happen?

Consider what is happening in the various threads. When the single region on Line 29 is executed, thread 2 was the first to be ready, so it was chosen to initialize the shared variables. Within each OpenMP region of code, there is an implied barrier which makes all threads wait for the current region to be completed before moving on to the next task. Thus, the other threads waited for thread 2 to initialize the shared variables M and N before moving on. Next, all four threads tackle Lines 37–42, and there is no barrier to stop them from continuing on. Thread 0 only has two numbers to add up, so it finishes first and immediately begins the single thread region on Line 45. The problem is, thread 3 hadn't finished before thread 2 started reporting the results, and hence thread 2 mistakenly reports that thread 3 found zero numbers. However, the implied barrier at the end of the single region means that all threads must complete their tasks at the end of the single region, and again at the end of the parallel region. Since by then thread 3 is finished, when the sum is computed via the `reduction(+:sum)` clause, all the values are included.

To fix this example, what is needed is a way to put in a pause to make sure all the threads are done before the reporting begins, and this is done via the `#pragma omp barrier` directive. In this case, the barrier is placed before Line 45 to make sure that all the threads have completed their work before moving on. The corrected example is shown in Example 11.2, with the barrier placed on Line 45.

### Example 11.2.

```

1 #include <stdio.h>
2 #include <omp.h>
3
4 /*
5  int main(int argc, char* argv[])
6
7  computes the number of positive numbers in each row and
8  the total sum
9
10 Inputs: none
11
12 Outputs: Prints the counts for each row and the total sum
13 */
14
15 int main(int argc, char* argv[])
16 {
17     int M;
18     int N;
19     int a[4][16] = {10, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
20                   6, 9, 10, 7, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
21                   2, 3, 4, 5, 6, 8, 1, 3, 3, 3, 9, 3, 6, 8, 6, 9,
22                   2, 9, 4, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
23     int sum = 0;
24     int count[4] = {0, 0, 0, 0};
25
26 #pragma omp parallel num_threads(4) shared(N,M,a,count) reduction(+:sum)
27 {
28     // initialize the shared variables, only one thread needed
29 #pragma omp single
30 {
31     printf("Thread %d is initializing.\n", omp_get_thread_num());
32     M = 4;
33     N = 16;

```

```

34    }
35
36    // This code will be run by each thread in parallel
37    int m = omp_get_thread_num();
38    int i;
39    for (i=0; i<N && a[m][i]!=0; ++i) {
40        ++count[m];
41        sum += a[m][i];
42    }
43
44    // This barrier is inserted to prevent starting the reporting too early
45    #pragma omp barrier
46
47    // use one thread to print the results
48    #pragma omp single
49    {
50        printf("Thread %d is reporting.\n", omp_get_thread_num());
51        for (i=0; i<M; ++i)
52            printf("Thread %d reports %d numbers.\n", i, count[i]);
53    }
54
55
56    printf("The sum is %d\n", sum);
57    return 0;
58}

```

This example also shows how there can be subtle errors in code that may not appear every time but may randomly malfunction if proper attention and care are not paid to these details. For this very small example, it had to be run a dozen times to get the incorrect results to appear once, but the key lesson is that they did appear that one time. These kinds of intermittent errors can lead to significant frustration when they occur and are not in the explicit logic of the code, which appears sound. This is why it's important to have a clear understanding of what the various threads are doing and not just blindly trust what otherwise appears to be a very simple and robust means of parallelizing code.

## 11.2 • The copyprivate Clause

The `#pragma omp single` directive supports the `private()`, `firstprivate()`, and `nowait` clauses, which behave the same as discussed for the `#pragma omp parallel` directive, so no additional comments are necessary. There is one additional clause not encountered yet, which is the `copyprivate` clause. This clause will take the private variable value that is computed inside the single region and copy the result to all the threads in the parallel region. For example, suppose in Example 11.2 the variables `M`, `N` were private and the private variable `sum` was initialized inside the parallel region. Then, the `copyprivate` clause would be used to make sure the values of `M`, `N`, and `sum` are initialized the same for all the threads as done in Example 11.3.

### Example 11.3.

```

1 #include <stdio.h>
2 #include <omp.h>
3
4 /*
5     int main(int argc, char* argv[])

```

```

7   computes the number of positive numbers in each row and
8   the total sum
9
10  Inputs: none
11
12  Outputs: Prints the counts for each row and the total sum
13 */
14
15 int main(int argc, char* argv[])
16 {
17     int M;
18     int N;
19     int a[4][16] = {10, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
20                 6, 9, 10, 7, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
21                 2, 3, 4, 5, 6, 8, 1, 3, 3, 3, 9, 3, 6, 8, 6, 9,
22                 2, 9, 4, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
23     int sum;
24     int count[4] = {0, 0, 0, 0};
25
26 #pragma omp parallel num_threads(4) shared(a,count) private(M,N) reduction(+:sum)
27 {
28     // initialize the shared variables, only one thread needed
29 #pragma omp single copyprivate(M,N,sum)
30 {
31     printf("Thread %d is initializing.\n", omp_get_thread_num());
32     M = 4;
33     N = 16;
34     sum = 0;
35 }
36
37 // This code will be run by each thread in parallel
38 int m = omp_get_thread_num();
39 int i;
40 for (i=0; i<N && a[m][i]!=0; ++i) {
41     ++count[m];
42     sum += a[m][i];
43 }
44
45 // This barrier is inserted to prevent starting the reporting too early
46 #pragma omp barrier
47
48 // use one thread to print the results
49 #pragma omp single
50 {
51     printf("Thread %d is reporting.\n", omp_get_thread_num());
52     for (i=0; i<M; ++i)
53         printf("Thread %d reports %d numbers.\n", i, count[i]);
54     }
55 }
56
57 printf("The sum is %d\n", sum);
58 return 0;
59 }
```

Line 26 now designates the variables `M`, `N` to be private. The `copyprivate` clause is used on Line 29 so that the values of the variables initialized in that single region are copied to the other threads at the completion of the single region. Without that clause, only the thread executing the single region will have properly initialized values in those variables and the others will be unspecified.

## Exercise

- 11.1. Write a program that creates an array  $x_j$  of  $N$  values generated using `drand48`. Within a single parallel region use a subdivided loop that computes

$$x_j \leftarrow \frac{1}{4}(x_{j+1} - 2x_j + x_{j-1})$$

for each  $j$  and then saves the result to a file using the `omp single` directive. For the formula above, assume  $x_{-1} \equiv 0$  and  $x_N \equiv 1$ . Iterate  $M$  times. Verify that  $x_j$  converges to  $\frac{j+1}{N+1}$ .

## Chapter 12

# Distinct Tasks in Parallel

Up to this point, either all threads performed the exact same task or one thread performed a single task while other threads waited. Sometimes it can be useful to have multiple threads handling very different tasks in parallel. For example, one thread may be reading data from a disk, one or more threads process the data, and yet another thread may write results to disk, all simultaneously. In this chapter, the directive `#pragma omp sections` is used to allow distinct sections of code to be done in parallel using one thread per section.

### 12.1 • Multiple Parallel Distinct Tasks: `#pragma omp sections`

Suppose for a given set of data, the data is to be written to the display and the max, min, and mean of the data computed. These can all be done in parallel using sections as illustrated in Example 12.1.

#### Example 12.1.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <omp.h> // This is the header file for OpenMP directives
4
5 /*
6  int main(int argc, char* argv[])
7
8  The main program illustrates how to divide threads into multiple
9  distinct tasks
10
11 Inputs: argc should be 2
12      argv[1]: Length of the data set
13
14 Outputs: Computes the mean, max, and min of the data
15
16 */
17
18 int main(int argc, char* argv[])
19 {
20     // Get the length of the array from the input arguments
21     int N = atoi(argv[1]);
22     int* u = (int*) malloc(N*sizeof(int));
```

```
24  double mean;
25  int maxu;
26  int minu;
27  int i;
28
29  // Initialize the array with random integers
30 #pragma omp parallel private(i) shared(u)
31 {
32 #pragma omp for
33   for (i=0; i<N; ++i)
34     u[i] = rand()%100;
35 }
36
37 #pragma omp parallel private(i) shared(u,minu,maxu,mean) num_threads(4)
38 {
39 #pragma omp sections
40 {
41
42   // This section will print the list of numbers
43 #pragma omp section
44 {
45   printf("Thread %d will print the contents of the array\n",
46         omp_get_thread_num());
47   for (i=0; i<N-1; ++i)
48     printf("%d, ", u[i]);
49   printf("%d\n", u[N-1]);
50 }
51
52   // This section computes the mean
53 #pragma omp section
54 {
55   printf("Thread %d will compute the mean\n",
56         omp_get_thread_num());
57   mean = 0;
58   for (i=0; i<N; ++i)
59     mean += u[i];
60   mean /= (double)N;
61 }
62
63   // This section computes the max
64 #pragma omp section
65 {
66   printf("Thread %d will compute the max\n",
67         omp_get_thread_num());
68   maxu = u[0];
69   for (i=1; i<N; ++i)
70     maxu = u[i] > maxu ? u[i] : maxu;
71 }
72
73   // This section computes the min
74 #pragma omp section
75 {
76   printf("Thread %d will compute the min\n",
77         omp_get_thread_num());
78   minu = u[0];
79   for (i=1; i<N; ++i)
80     minu = u[i] < minu ? u[i] : minu;
81 }
82 }
83 }
84
85 // print the results
86 printf("mean(u) = %4.1f\n", mean);
```

```

87   printf("max(u) = %d\n", maxu);
88   printf("min(u) = %d\n", minu);
89
90   return 0;
91 }
```

Sample output for Example 12.1 is shown below:

```

Thread 2 will compute the mean
Thread 3 will print the contents of the array
35, 49, 21, 86, 77, 15, 93, 86, 92, 83
Thread 0 will compute the max
Thread 1 will compute the min
mean(u) = 63.7
max(u) = 93
min(u) = 15
```

In this example, one thread is designated for each of four sections, one to print the data, and one each to compute the mean, min, and max of the data. To begin, on Line 32, the random data is generated using the directive `#pragma omp for` that was introduced in Chapter 10, so all four threads subdivide the array and fill it out. Next, the `#pragma omp sections` directive is given on Line 39, which indicates that the computation will now be broken down into multiple sections, each designated by the directive `#pragma omp section` on Lines 43, 53, 64, and 74. One thread will be assigned to each section, and any additional threads in the parallel region will be left idle.

## 12.2 • The private Clause

Variables inside the `#pragma omp sections` region can be designated as private even if the parallel region within which it is contained designated the variables as shared. In that instance, the variables will be private within each section. So long as the data in these variables is not needed outside their respective sections, the private variables behave in the same manner as before, where the values inside the section are independent of the variables outside the sections. The `firstprivate` clause also works as before, where the value of the variable entering the sections is copied to the local private variable.

One difference, however, is for the `lastprivate` clause. When this clause is used, the value of the private variable will be copied outside the sections region, *but only for the last section in the code*. For Example 12.1, the clause `lastprivate(minu)` may be used safely because the value of `minu` is computed in the last section in the code beginning on Line 74. By comparison, had the `lastprivate(maxu)` clause been used, the results would be unpredictable because the value of `maxu` was not computed in the last section. To avoid this problem, and because the different sections can operate independently, the variables `mean`, `maxu`, and `minu` were all declared as shared on Line 30.

## 12.3 • The reduction Clause

The reduction clause works in the same way as discussed in Section 9.5, so the reader is referred there for an explanation of how it works. Specific to sections, it is worth noting that this would be an alternative way to recover private variables from multiple

sections. For example, use the clause `reduction(+:maxu)` and set `maxu` to be zero in each section except for the one where `maxu` is computed; then the reduction will add zeros to the true value and `maxu` is recovered. Again, for this simple example, it makes more sense to keep `maxu` as shared.

## 12.4 • The nowait Clause

The `nowait` clause behaves in the same way as discussed in Section 10.6. Ordinarily, there is an implied barrier at the end of the `#pragma omp sections` region, where each thread waits until all the sections are complete. If the `nowait` clause is used, then when a thread completes its assigned section, or was an extra thread that was not assigned a section, the thread may proceed to the next task in the parallel region without waiting for the other sections to complete. Again, caution must be used to make sure that there are no hidden dependencies between the next task and the prior incomplete sections to avoid unpredictable results.

---

### Exercise

- 12.1. Write a program that will read an array from a sequence of files named “`input.X.dat`” where `X` is replaced with integers 0, 1, .... Let  $x_j^n$  be the  $j$ th entry in the  $n$ th file. The program should compute the sum array  $y_j^n$ , where

$$y_j^n = \sum_{k=0}^n x_j^n,$$

and the resulting array should be written to the file names “`output.X.dat`”. The sum should be done using an `omp for` directive, and then use the `omp sections` directive to have one thread write the output file at the same time that a second thread reads the new input file.

## Chapter 13

# Critical and Atomic Code

There are times when using shared variables can lead to race conditions where multiple threads updating the same data are desired but lead to incorrect results. To do this safely, threads must coordinate so that two threads are not updating the same shared variable simultaneously. Without coordination, this kind of data collision can cause unexpected results. In this chapter, two different options are presented for dealing with shared variables that may be frequently updated by different threads. For a single variable with a very simple update step, directing a variable to be atomic is the easiest and most efficient. For small regions or slightly more complicated updates that don't fit within the restricted conditions for atomic statements, there is the alternative critical statement type.

### 13.1 • Atomic Statements

Consider Example 13.1, where a count is made of how many random numbers are less than the middle value in an array.

#### Example 13.1.

```
1 #include <stdio.h>
2 #include <stdlib.h> // RAND_MAX is defined in here
3 #include <omp.h>
4
5 /*
6   int main(int argc, char* argv[])
7
8   computes the number of values less than RAND_MAX/2
9   in a random array
10
11  Inputs: argc should be 2
12      argv[1]: length of the random array
13
14  Outputs: Prints the number of values less than RAND_MAX/2
15 */
16
17 int main(int argc, char* argv[])
18 {
19     // Get the number of random values to sample from the arg list
20     int N = atoi(argv[1]);
```

```

21 int a[N];
22
23 // Use mid as the cutoff value
24 int mid = RAND_MAX/2;
25 int i;
26 int count = 0;
27
28 // Generate the random values
29 #pragma omp parallel for private(i) shared(a,N)
30   for(i=0; i<N; ++i)
31     a[i] = rand();
32
33 #pragma omp parallel private(i) shared(a,N,mid,count)
34 {
35   // initialize the shared variable count using only one thread
36 #pragma omp single
37   {
38     count = 0;
39   }
40
41 #pragma omp for
42   for (i=0; i<N; ++i) {
43     if (a[i] < mid) {
44       // make sure count is updated one thread at a time
45 #pragma omp atomic
46       ++count;
47     }
48   }
49 }
50
51 printf ("%d/%d numbers are below %d\n", count, N, mid);
52 return 0;
53 }
```

In this example, the variable `count` is declared to be shared. Typically, it's best to avoid having multiple threads saving to the same shared variable to avoid conflicts, also called a data race condition. The strategy used before for this type of situation was to declare `count` as private and then combine the different `count` variables at the end via the `reduction(+:count)` clause. An alternate strategy illustrated in Example 13.1 is to use the `#pragma omp atomic` directive to prevent data collisions.

The `#pragma omp atomic` directive is a very simple directive that applies *only to the following single simple statement*. For some versions of OpenMP, it only permits increment and decrement commands, e.g., `++j`, `-k`, or `m += 2`. More advanced versions of OpenMP will also permit commands of the form

```
count = count + increment;
```

The directive on Line 45 is inserted to make sure that `count` is being updated by only one thread at a time. To see why it is necessary, consider the steps a thread takes to update `count`:

1. Read `count` from shared memory.
2. Add 1 to `count`.
3. Store `count` back into shared memory.

Suppose threads 0 and 1 arrive at step 1 simultaneously when `count` has value 0, then they each temporarily have a copy of `count` with value 0, which they then each in

parallel increment by 1, and then dutifully update the shared value of `count` with value 1. The problem is, if both of them were meant to increment, then the value of `count` should have been 2. By introducing Line 45, the threads are forced to take turns when they implement that incremental update. Thus, thread 0 would complete steps 1–3 above before thread 1 is permitted to begin. This way, thread 0 updates `count` to 1, and thread 1 updates it to 2 as intended. If the example is run with and without the `#pragma omp atomic` directive, the impact can be seen immediately. For the exact same data, the following results are obtained:

With the `#pragma omp atomic` directive:

`50038/100000 numbers are below 1073741823`

Without the `#pragma omp atomic` directive:

`26791/100000 numbers are below 1073741823`

Clearly, there were many collisions that occurred for this simple example that would not have been caught had the atomic directive been left out.

## 13.2 • Critical Statements

The `#pragma omp atomic` directive is very restrictive in terms of its usage in that it applies only to a single statement, and that single statement has to be of a limited set of options. A more general directive is the `#pragma omp critical` directive. It operates in much the same way but is able to handle a more general condition. For example, Example 13.1 can be modified to also sum all the numbers that are below the threshold resulting in Example 13.2.

### Example 13.2.

```

1 #include <stdio.h>
2 #include <stdlib.h> // RAND_MAX is defined in here
3 #include <omp.h>
4
5 /*
6   int main(int argc, char* argv[])
7
8   computes the number of values less than RAND_MAX/2
9   in a random array
10
11  Inputs: argc should be 2
12      argv[1]: length of the random array
13
14  Outputs: Prints the number of values less than RAND_MAX/2
15      and the sum of those small values
16 */
17
18 int main(int argc, char* argv[])
19 {
20     // Get the number of random values to sample from the arg list
21     int N = atoi(argv[1]);
22     int a[N];
23
24     // Use mid as the cutoff value
25     int mid = RAND_MAX/2;
26     int i;
27     int count = 0;
28     double sum = 0.;
```

```

29 // Generate the random values
30 #pragma omp parallel for private(i) shared(a,N)
31   for(i=0; i<N; ++i)
32     a[i] = rand();
33
34
35 #pragma omp parallel private(i) shared(a,N,mid,count)
36 {
37   // initialize the shared variable count using only one thread
38 #pragma omp single
39   {
40     count = 0;
41     sum = 0.;
42   }
43
44 #pragma omp for
45   for (i=0; i<N; ++i) {
46     if (a[i] < mid) {
47       // make sure count and sum are updated one thread at a time
48 #pragma omp critical
49   {
50     ++count;
51     sum += a[i];
52   }
53   }
54 }
55
56 printf("%d/%d numbers are below %d\n", count, N, mid);
57 printf("The sum is %e.\n", sum);
58 return 0;
59
60 }
```

On Line 48, the `#pragma omp atomic` directive is replaced with the `#pragma omp critical` directive. The critical directive allows for multiple lines surrounded by braces and allows for more complex statements.

The `#pragma omp atomic` directive should be used for very simple incrementing operations because of its improved efficiency over the more general `#pragma omp critical` directive, but the critical directive is available when the atomic directive is not sufficient.

## Exercise

- 13.1. Use the `omp atomic` directive to compute the maximum value of an array. This would be an alternative to using the `reduction` clause.

## Chapter 14

# OpenMP Libraries

There are some libraries that utilize OpenMP as part of their construction, and for those, their use is no different from the libraries discussed in Chapter 7, and the implementation of the OpenMP code is done in the library hidden from view. On the other hand, libraries that are designed for serial programs may or may not work in the OpenMP framework. A library is safe for use in conjunction with OpenMP if the library is reported to be thread-safe. There are thread-safe versions of the BLAS and LAPACK, but all the details of that are under the hood and the use of these libraries from the context of OpenMP is no different from how they would be used in serial code as discussed in Section 7.1. Consequently, that discussion won't be repeated here. For other libraries there can be some differences, and so those differences are addressed in this chapter.

### 14.1 • FFTW

One thing to bear in mind about using the OpenMP version of FFTW is that the OpenMP implementation is all inside the library. In principle, the OpenMP directives are not needed in order to take advantage of the parallel implementation of FFTW. Example 14.1 shows how a multithreaded OpenMP implementation of Example 7.4 is done.

#### Example 14.1.

```
1 #include <stdio.h>
2 #include <math.h>
3 #include <omp.h>
4 #include <fftw3.h>
5
6 /*
7  int main( int argc , char* argv[] )
8
9  The main program illustrates how to use the multithreaded
10 version of the FFTW library
11
12 Inputs: none
13
14 Outputs: Prints the original data and the result of a forward
15 and backward transform for comparison
16
17 */
```

```

18
19 int main(int argc, char* argv[])
20 {
21 #ifndef M_PI
22     const double M_PI = 4.0*atan(1.0);
23 #endif
24     int N = 16;
25     fftw_complex *in, *out, *out2;
26     fftw_plan p, pinv;
27
28 // initialize the threads to be used in the FFT execution
29 fftw_init_threads();
30
31 // allocate memory for the data
32 in = (fftw_complex *) fftw_malloc(sizeof(fftw_complex)*N);
33 out = (fftw_complex *) fftw_malloc(sizeof(fftw_complex)*N);
34 out2 = (fftw_complex *) fftw_malloc(sizeof(fftw_complex)*N);
35
36 // create an FFT plan that uses all the available threads
37 fftw_plan_with_nthreads(omp_get_max_threads());
38
39 // construct the plans for the forward and backward transforms
40 p = fftw_plan_dft_1d(N, in, out, FFTW_FORWARD, FFTW_ESTIMATE);
41 pinv = fftw_plan_dft_1d(N, out, out2, FFTW_BACKWARD, FFTW_ESTIMATE);
42
43 // construct the initial data
44 for (int i=0; i<N; ++i) {
45     double dx = 2.*M_PI/N;
46     in[i][0] = sin(i*dx);
47     in[i][1] = 0.;
48 }
49
50 // perform the forward transform, results are in variable out
51 fftw_execute(p);
52
53 // perform the backward transform, results are in out2
54 fftw_execute(pinv);
55
56 // print the original data and the results for comparison
57 for (int i=0; i<N; ++i)
58     printf("a[%d]: %f + %fi\n", (i<=N/2 ? i : i-N), out[i][0]/N,
59                                     out[i][1]/N);
60     printf(" ---\n");
61 for (int i=0; i<N; ++i)
62     printf("f[%d]: %f + %fi ==%f + %fi\n", i, out2[i][0]/N,
63                                     out2[i][1]/N, in[i][0], in[i][1]);
64
65 // clean up the threads from OpenMP
66 fftw_cleanup_threads();
67
68 // free the remaining memory
69 fftw_destroy_plan(p);
70 fftw_destroy_plan(pinv);
71 fftw_free(in);
72 fftw_free(out);
73 fftw_free(out2);
74 }
```

The program is almost identical to Example 7.4 except for three additional lines. First, the threaded version of FFTW requires the threads to be initialized before any FFTW functions are called, which is done on Line 29. For multithreading to work it's

essential to put that at the beginning of the program. Next, on Line 37, the number of threads available to use in the multithreaded plan is specified. This function can be called multiple times if the number of threads to be used is different for different plans. In this instance, all available threads will be used for constructing the plan by using `omp_get_max_threads` to give the number of threads. The plans are now set to use multiple threads and the FFT computations done on Lines 51, 54 are done in parallel using OpenMP. Finally, the extra memory used for the multithreaded version must be cleaned up, hence there is an additional thread clean-up on Line 66.

Like the serial version of FFTW, the input and output data is assumed to be stored in column-major order, which is important to remember when computing the spectral derivatives described in Chapter 37.

Finally, when linking this program, the library `libfftw3_omp` must be used in addition to the ones used to build Example 7.4. Thus, to compile and link Example 14.1, the following command is used:

```
$ gcc -o fftomp fftomp.c -fopenmp -lgomp -lfftw3_omp -lfftw3 -lm
```

## 14.2 • Random Numbers

Generation of random numbers in parallel is more complicated than it may seem at first glance. The first problem is that the standard random number generators, for example, `random` or `drand48` used in Part I, are not necessarily thread-safe. For example, consider Example 14.2, where `random` is used to populate multiple tables of random long integers, one table for each thread.

### Example 14.2.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4 #include <omp.h>
5
6 /*
7  int main(int argc, char* argv[])
8
9   The main program generates a table of random integers
10
11  Inputs: argc should be 3
12  argv[1]: the length of the table
13  argv[2]: the RNG seed
14
15  Outputs: Computes the mean, max, and min of the data
16
17 */
18
19 int main(int argc, char* argv[])
20 {
21     int N = atoi(argv[1]); // input the length of each table
22     long int seed = atol(argv[2]); // input seed value
23     int i, k;
24     int nums[omp_get_max_threads()][N]; // table of random variables
25
26 #pragma omp parallel shared(N, seed, nums) private(i, k)
27 {
28     // fill the table with random values
29     srand(seed);

```

```

30     k = omp_get_thread_num();
31     for (i=0; i<N; ++i)
32         nums[k][i] = random() %100;
33 }
34
35 // print the results
36 for (i=0; i<N; ++i) {
37     for (k=0; k<omp_get_max_threads(); ++k)
38         printf("%d\t", nums[k][i]);
39     printf("\n");
40 }
41
42 return 0;
43 }
```

The intent of this example is to have each thread produce a list of  $N$  random integers and store them in the shared array. The first thing the astute reader should catch is that on Line 29, the initial seed for the random values to be assigned to the array is the same for every thread, and hence it should be expected that the same list of numbers will result for each thread. Instead, the following results are obtained when run with the arguments  $N= 3$  and  $seed= 1$ :

```

10  83  83  15  83  83  83  83
 0   86  86  93  86  86  86  86
 -1  77  77  35  77  77  77  77
```

As can be seen, most of the entries did what is expected, which is to generate the repeated sequence 83, 86, 77, yet threads 0 and 3 came up with completely different answers. Not only that, but the function `random` is supposed to produce only nonnegative integer values, but the last entry in the first column is negative. What is happening is that the functions `random` and `srandom` are not thread-safe. Thus, the results are not completely predictable.

Fortunately, thread-safe versions of the random functions are available, but they do require a bit more effort. The main issue is that the standard random number generator uses a global variable to maintain its state, and it's not possible to make the state variable private within the thread. Therefore, the thread-safe versions are entirely different, and a private state variable must be maintained within each thread. The corrected version of Example 14.2 is in Example 14.3.

### Example 14.3.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4 #include <omp.h>
5
6 /*
7  int main(int argc, char* argv[])
8
9  The main program generates a table of random integers
10
11 Inputs: argc should be 3
12 argv[1]: Length of the table
13 argv[2]: Initial RNG seed
14 }
```

```

15  Outputs: Computes the mean, max, and min of the data
16
17 */
18
19 int main(int argc, char* argv[])
20 {
21     int N = atoi(argv[1]); // input the length of each table
22     long int seed = atol(argv[2]); // input seed value
23     int i, k;
24     int nums[omp_get_max_threads()][N]; // table of random variables
25     struct random_data* state; // pointer to random number state
26
27 #pragma omp parallel shared(N, seed, nums) private(i, k, state)
28 {
29     k = omp_get_thread_num();
30
31     // allocate space for the state variable
32     state = malloc(sizeof(struct random_data));
33
34     // populate the state variable for each thread
35     char statebuf[32];
36     initstate_r(seed, statebuf, 32, state);
37
38     // set the initial seed with thread-safe version
39     srand(statebuf);
40     int temp;
41     for (i=0; i<N; ++i) {
42         // get a random value using the thread-safe version
43         random_r(state, &temp);
44         nums[k][i] = temp%100;
45     }
46
47     // release the memory allocated for the state
48     free(state);
49 }
50
51 // print the results
52 for (i=0; i<N; ++i) {
53     for (k=0; k<omp_get_max_threads(); ++k)
54         printf("%d\t", nums[k][i]);
55     printf("\n");
56 }
57
58 return 0;
59 }
```

There are several changes that had to be made so that the random number generator is thread-safe. To begin, the functions on Lines 36, 39, 43 now have a suffix of “\_r”. These functions are designed to be thread-safe versions of the standard functions. However, the changes are more substantial than just adding the suffix.

Each thread will need to track its own state variable rather than relying on an underlying global state variable. A pointer to the state variable is declared on Line 25. Each thread allocates memory for a state variable within the parallel region on Line 32. It is not sufficient to just have a local state variable; the state variable also relies on a separate variable length buffer, which is declared on Line 35. This also must be private to each thread.<sup>5</sup> The size of the buffer should be a power of two and can be anywhere

---

<sup>5</sup>Any variable that is declared inside a parallel region is private by definition and is not listed in a `private` clause.

from 8 to 256. The larger the buffer, the more complex the random number generation. Before using the state variable, it must be initialized using the `initstate_r` function. It takes an initial seed, a pointer to the temporary buffer and the size of that buffer, and finally a pointer to the state that is to be initialized. Once the state is set up, the initial seed for the random number generator can be set using `srandom_r` as done on Line 39. Compared to the non-thread-safe version, the state variable is also passed to `srandom_r` as an argument so that the underlying function can operate specifically on the data and memory allocated to the current thread. Similarly, on Line 43, the `random` function is also replaced with the new function `random_r(state, &randval)`, where the first argument is a pointer to the state variable, and the second argument is a pointer to where the next random value should be stored. Finally, because the state variable was dynamically allocated within each thread, the memory for the state variable must be released again before the thread is destroyed as is done on Line 48. By making these changes, the expected results are obtained.

```
63 63 63 63 63 63 63 63
40 40 40 40 40 40 40 40
15 15 15 15 15 15 15 15
```

However, this is still not exactly what is desired because the goal is to have each column of numbers be independent. To do that, each thread needs to start with a different seed. It's a simple change on Lines 36, 39 to replace `seed` with `seed+k`, where `k` is the thread number calculated on Line 29, resulting in the following output:

```
63 10 21 20 33 80 91 57
40 62 55 77 1 75 16 50
15 72 14 71 67 24 18 72
```

There are corresponding versions of the random number generator `drand48`, the thread-safe version also using the “`_r`” suffix. The generator for `drand48_r` doesn't require the extra state storage; instead it is maintained by the `struct drand48_data` structure and initialized when it is seeded using the function `srand48_r`. A corresponding example for generating double precision, uniformly distributed random values is given in Example 14.4.

#### Example 14.4.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4 #include <omp.h>
5
6 /*
7  int main(int argc, char* argv[])
8
9  The main program generates a table of random integers
10
11 Inputs: argc should be 3
12      argv[1]: Length of the table
13      argv[2]: Initial RNG seed
14
15 Outputs: Computes the mean, max, and min of the data
16
17 */
```

```

18
19 int main(int argc, char* argv[])
20 {
21     int N = atoi(argv[1]); // input the length of each table
22     long int seed = atol(argv[2]); // input seed value
23     int i, k;
24     double nums[omp_get_max_threads()][N]; // table of random variables
25     struct drand48_data* state; // pointer to random number state
26
27 #pragma omp parallel shared(N, seed, nums) private(i, k, state)
28 {
29     k = omp_get_thread_num();
30
31     // allocate space for the state variable
32     state = malloc(sizeof(struct drand48_data));
33
34     // set the initial seed and state with thread-safe version
35     srand48_r(seed+k, state);
36     for (i=0; i<N; ++i)
37         // get a random value using the thread-safe version
38         drand48_r(state, &(nums[k][i]));
39
40     // release the memory allocated for the state
41     free(state);
42 }
43
44 for (i=0; i<N; ++i) {
45     for (k=0; k<omp_get_max_threads(); ++k)
46         printf("%f\t", nums[k][i]);
47     printf("\n");
48 }
49
50 return 0;
51 }
```

As shown on Line 25, the random number generator state has a different type, which again must be private for each thread. The space for the state is allocated within each thread on Line 32, and then the state is initialized with a specified seed value on Line 35. Note that there is no need for a separate storage buffer for the state vector as was needed in Example 14.3. Finally, the random values are stored directly into the data array on Line 38.

## Exercises

- 14.1. Modify Example 14.1 to do a two-dimensional FFT transform. In this case, the forward plan is constructed by calling the function

```
fftw_plan fftw_plan_dft_2d(int m, int n, fftw_complex *in,
                           fftw_complex *out, int sign, unsigned in_flags);
```

Aside from the data arrays `in` and `out` being pointers to a two-dimensional array of dimensions  $m \times n$ , the remaining arguments are the same as before. Take as inputs to your program the integer wave modes  $k_1$  and  $k_2$  so that the complex variable  $\text{in}[i][j][0] = \cos((k_1*i+k_2*j)*dx)$  and  $\text{in}[i][j][1] = \sin((k_1*i+k_2*j)*dx)$ . Print out the values of the spectral coefficients after

the forward transform. Perform the reverse transform and verify that the initial data is recovered. Try different values of  $k_1$  and  $k_2$  and note how the spectral coefficients depend on the values.

- 14.2. Write a program to generate an  $N \times N$  array of random values using a uniform distribution in the range  $[0, 1]$ . Verify that no two values are the same and that if the same seed is used twice, then the full array is exactly the same.

## Chapter 15

# Projects for OpenMP Programming

The background for the projects below is given in Part VI of this book. You should be able to build each of these projects based on the information provided in this text and utilize some or all of the OpenMP compiler directives to improve the speed of your program. For some of the projects, there may be some additional comments to give hints on how to best take advantage of the OpenMP system, while for others, it should be obvious, such as using `#pragma omp for` to parallelize `for` loops.

### 15.1 • Random Processes

#### 15.1.1 • Monte Carlo Integration

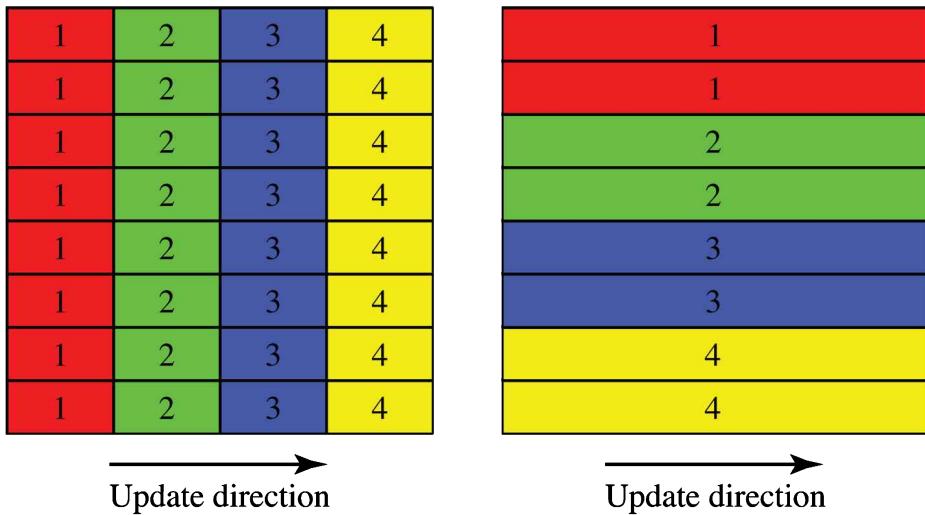
Program the assignment in Section 34.3.1 using  $N$  samples, where  $N$  is an input parameter. Measure the time required to complete the calculation by using the function `omp_get_wtime` at the beginning and end of your program and print the elapsed time in seconds. Use the timing information to estimate the cost to compute a solution with error  $10^{-5}$  with 99.5% confidence. Compare the time required to complete the calculation compared with the serial version of this problem. Verify that the cost also increases linearly, but with a smaller coefficient depending on the number of threads used.

Be sure to use a thread-safe random number generator. Each thread should generate its own subtotal, then use the `reduction` clause to combine the subtotals to get the final result.

#### 15.1.2 • Duffing–Van der Pol Oscillator

Program the assignment in Section 34.3.2 using  $N$  samples and using  $M$  time steps, i.e., take  $\Delta t = T/M$  in the Euler–Maruyama method. The values of  $N$ ,  $M$ , and  $\sigma$  should be given on the command line. Use  $\alpha = 1$ , which should be defined in your program. Measure the time required to complete the calculation by using the `omp_get_wtime` function at the beginning and end of your program and print the elapsed time in seconds. Plot an estimate for  $p(t)$  for  $0 \leq t \leq T = 10$ .

This is an example of an embarrassingly parallelizable problem. In this case, each sample path should be generated by a single thread. Each thread will compute multiple



**Figure 15.1.** Organization of threads for doing updates in the  $x$ -direction. On the left, each row is updated by subdividing the inner loop. On the right, each row is updated by a single thread, and the rows are subdivided among the threads. Either method will work for an explicit update, but only the right-hand case can be used for an implicit update.

sample paths, and the resulting binning of the data can be handled by using a `#pragma omp critical` or `atomic` directive to ensure that different threads do not conflict when updating the terminal location.

## 15.2 • Finite Difference Methods

In the alternating directions implicit (ADI) method, the most efficiency is gained by putting the full time step update inside a parallel region. In the method, there are both explicit and implicit temporal updates. Explicit updates can be handled by using the `#pragma omp for` directive either on the outer loop (the loop over all rows or columns of the data) or the inner loop (loop along the row or column to do the explicit update). The two options are illustrated in Figure 15.1. In this case, it is more efficient to put the `#pragma omp for` directive on the outer loop.

For the implicit step, the strategy depends upon whether the LAPACK library being used is multithreaded. If the library is multithreaded, then you should only parallelize the explicit steps and then use the optimized LAPACK library from a single thread to solve the full system. On the other hand, if the LAPACK library is not multithreaded, then the call to the LAPACK solver should be inside a parallel region so that each thread calls the LAPACK solver such as `dgttrs_` or `dgetrs_` to solve a subset of the columns on the right-hand side. Thus, if there are  $k$  threads in the parallel region, then each thread will solve the linear system for  $N/k$  rows or columns depending on the direction of the update where the grid is  $N \times N$ . Parallelizing the implicit steps will yield the most gain in efficiency in the algorithm.

### 15.2.1 • Brusselator Reaction

Program the assignment in Section 35.4.1 using an  $N \times N$  grid. Measure the time

required to complete the calculation by using the `omp_get_wtime` function at the beginning and end of your program and print the elapsed time in seconds.

### 15.2.2 • Linearized Euler Equations

Program the assignment in Section 35.4.2 using an  $N \times N$  grid. Measure the time required to complete the calculation by using the `omp_get_wtime` function at the beginning and end of your program and print the elapsed time in seconds.

## 15.3 • Elliptic Equations and Successive Overrelaxation

As discussed in Chapter 36, a parallel implementation of the successive overrelaxation (SOR) method should use the red/black ordering strategy. However, within that strategy, the loops can again be organized by subdividing either the inner or the outer loop. The two choices are illustrated in Figure 15.2. If the inner loop is subdivided, then each thread takes a piece of each row. If the outer loop is subdivided, then each row is updated by a single thread. Once all these red grid points are updated, the black grid points are updated in the same manner. You will want to use the `reduction` clause with the `max` operator so that the maximum residuals computed by each of the threads can be combined at the end of each subdivided loop.

### 15.3.1 • Diffusion with Sinks and Sources

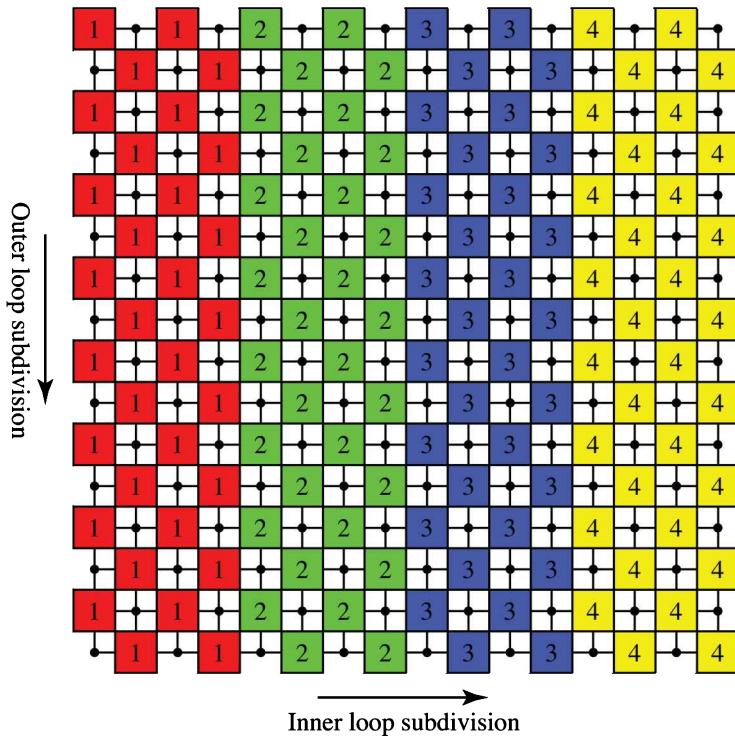
Program the assignment in Section 36.1.1 using a  $(2N - 1) \times N$  grid. Experiment to find the optimal  $\omega$  that requires the fewest iterations to reach a tolerance of  $10^{-9}$ . Compute the time required to complete the calculation using the `omp_get_wtime` function, then divide by the number of iterations to give the average time used for a single pass through the grid. Repeat these steps for a grid of dimensions  $(4N - 1) \times 2N$ . Does the optimal value of  $\omega$  remain the same? Verify that the time cost for a single pass through the grid is proportional to the number of grid points.

### 15.3.2 • Stokes Flow 2D

Program the two-dimensional assignment in Section 36.1.2 using a MAC grid that is approximately  $N \times N$ , i.e., the grid for  $u$  should be  $N \times N - 1$ , the grid for  $v$  should be  $N - 1 \times N$ , and the grid for  $p$  should be  $N - 1 \times N - 1$ . Experiment to find the optimal  $\omega$  that requires the fewest iterations to reach a tolerance of  $10^{-9}$ . Compute the time required to complete the calculation using the `omp_get_wtime` function, then divide by the number of iterations to give the average time used for a single pass through the grid. Verify that you get a parabolic profile for the velocity field in the  $x$ -direction.

### 15.3.3 • Stokes Flow 3D

Program the three-dimensional assignment in Section 36.1.2 using a MAC grid that is approximately  $N \times N \times N$ , i.e., the grid for  $u$  should be  $N \times N - 1 \times N - 1$ , the grid for  $v$  should be  $N - 1 \times N \times N - 1$ , the grid for  $w$  should be  $N - 1 \times N - 1 \times N$ , and the grid for  $p$  should be  $N - 1 \times N - 1 \times N - 1$ . Experiment to find the optimal  $\omega$  that requires the fewest iterations to reach a tolerance of  $10^{-9}$ . Compute the time required to complete the calculation using the `omp_get_wtime` function, then divide by the number



**Figure 15.2.** Organization of threads for doing updates for red/black ordering in SOR. This figure represents updating the red points. The colored boxes represent the red points in the red/black ordering, and are updated first. The other points are black points and would be updated afterward. If the inner loop is subdivided, then the data is updated row by row as illustrated. If the outer loop is updated, then the columns represent the organization, where each column is updated by a single thread.

of iterations to give the average time used for a single pass through the grid. Verify that you get a roughly parabolic profile for the velocity field in the  $x$ -direction.

## 15.4 • Pseudospectral Methods

The FFTW package for computing the Fourier transform is already multithreaded, so it is best to perform the Fourier transforms using the multithread capability of the FFTW library. However, the computation of the spatial derivatives requires looping through the Fourier modes, and the Runge–Kutta temporal updates require looping over the grid, so those can be done by subdividing the `for` loops. Since the derivative calculation has no dependency between Fourier modes, this can be safely parallelized without having to worry about conflicts.

### 15.4.1 • Complex Ginsburg–Landau Equation

Program the assignment in Section 37.5.1 using an  $N \times N$  grid. Plot the results for the phase and identify where the spiral waves have emerged through pattern formation. Measure the time required to complete the calculation by using the `omp_get_wtime` function at the beginning and end of your program and print the elapsed time in seconds.

### 15.4.2 ▪ Allen–Cahn Equation

Program the assignment in Section 37.5.2 in two dimensions using an  $N \times N$  grid. Plot the results and verify that phase separation is occurring. Measure the time required to complete the calculation by using the `omp_get_wtime` function at the beginning and end of your program and print the elapsed time in seconds.



## **Part III**

# **Distributed Programming and MPI**



So far the focus has been on methods that use a single computer with perhaps multiple processors. In those settings, there is an advantage in terms of simplicity of implementation and also low cost of communications between threads. However, these strategies are limited to the number of processors that a single node can house. To get to the very large systems, such as the Sunway TaihuLight with over 10 million cores, programs must be able to communicate across multiple nodes (i.e., multiple independent computers coupled together by high-speed connections). This is called a distributed system, and the communications are handled by a specialized library called MPI (for message passing interface).

When designing parallel algorithms for distributed systems, there are two key computational costs that must be considered: latency costs and communication costs. Latency costs arise when a core is unable to proceed and hence stalls because it is waiting for necessary information to arrive. Communication costs are the cost of sending data from one node to another. Ideally, send as little data as possible between different nodes, while keeping all processors active. When successful, the cost of the computation will be inversely proportional to the number of cores used to tackle the problem.

MPI codes can run on configurations similar to that for OpenMP, but to really get a feel for using MPI it's best to use a cluster of computers. Most computer clusters use a queuing system for submitting jobs. The purpose of a queuing system is to keep the cluster busy by scheduling programs to run on a requested subset of the nodes in the cluster exclusively by that program. That exclusivity is important because timing of processes is critical for optimal use of computer resources. If some small program is running on one node shared with a large computation running on many nodes, then the delay caused by the small program can disrupt the timing of the large computation so that all the cores in the computation end up waiting around from time to time, leading to significantly less efficiency. Therefore, some basics of shell scripting and job submissions for a queuing system will be discussed with the understanding that there may be significant differences between systems. Check with your systems administrator to get the job scheduling commands for your system.

A brief description of a queuing system called Moab is provided in Chapter 16, where the form of a job script and the command used to submit the job are discussed. The concepts are common among many queuing systems so while the details for your system may vary, the basic ideas will translate. The chapter also presents an MPI version of "Hello World!" to learn how MPI creates multiple processes within a single program. The chapter concludes with instructions on how to compile, link, and execute programs that use MPI.

At the heart of MPI is the means of getting multiple processes on multiple compute nodes to communicate with each other to coordinate distributed computational tasks. In Chapter 17 the various forms of communication are discussed, including one-to-one, many-to-one, one-to-many, and many-to-many communication patterns. Some of these

communication types come in blocking and nonblocking versions that permit processes to continue working while waiting for communications to be completed.

The communications in Chapter 17 are all done in the context of all available processes. Sometimes group communications are needed in only a subset of the processes. Chapter 18 explains how subgroups can be formed quickly and the communication patterns controlled among subgroups.

Chapter 19 covers how to measure efficiency in the context of a multicore system and includes some important practical considerations to prevent wasting the valuable computing resource. Not only are the efficiency measures important to validate the use of multiple cores over a single core, but also the measures can be helpful in identifying bottlenecks in the code that can bear greater scrutiny. There is also a discussion about checkpointing, where data is stored periodically in such a way that should a process fail for some reason, the program could be restarted from the last checkpoint, saving much computational cost.

While the standard C libraries discussed in Chapter 7 can be used within a single MPI process in exactly the same way, there are many libraries that are adapted to a parallel architecture in order to further gain computational speed. Taking advantage of these libraries can significantly reduce the time to completion of a code and the corresponding probability for introducing errors. Chapter 20 introduces the parallel versions of the popular libraries.

Finally, in Chapter 21 there is considerable discussion about how to adapt a distributed architecture to solve the projects in Part VI. Domain decomposition is a fairly standard way to break up a computational task into smaller pieces, but that also requires coordinated communications to ensure that the code doesn't end up blocking or using the wrong data.

For a more extensive discussion of MPI in the context of scientific computing the interested reader is referred to the text by Karniadakis and Kirby [16].

# Chapter 16

# Preliminaries

Using MPI on a cluster requires more than just learning how to write a program that handles communications. Clusters require coordination by a scheduler to ensure that all the cores requested are started at the same time and that communication channels between the cores are established. In order to accomplish this, a queuing system is used that prioritizes computational jobs among many users, sets aside requested resources, and then follows the instructions provided in a job script submitted to its queue. In this chapter, we will begin with a parallel version of “Hello World!” and discuss how to compile it, how to write a job script to submit it to the queue, and how to submit the job to the queuing system.

## 16.1 • Parallel “Hello World!”

The first thing to grasp when writing code using MPI is that the code will be used on multiple processes simultaneously. That means that if the original “Hello World!” program from Part I were run in a parallel environment, say, using four cores, then four copies of “Hello World!” would be written out. The four different executions would not know about each other and will happily dump their output to the terminal in no particular order. The challenge is to get different processes to do *different* things in parallel. That is where MPI comes into play. The goal is to break a task down into smaller pieces, and then each piece should complete its own semi-independent task.

To make each process do a specific task, the program must identify which process it is. Example 16.1 shows an MPI version of “Hello World!” that has each process report its status.

### Example 16.1.

```
1 #include <stdio.h>
2 #include <mpi.h> // This is the header file for MPI functions
3
4 /*
5 int main( int argc , char* argv[] )
6
7 The main program is the starting point for an MPI program.
8 This one simply reports its own process number.
9
10 Inputs : none
```

```

11 // Outputs: Prints "Hello World #" where # is the rank of the process.
12 /*
13 */
14 */
15
16 int main(int argc, char* argv[])
17 {
18     // First thing is to initialize the MPI interface. Some arguments can
19     // be passed through to the MPI interface, so the argument list is
20     // sent by sending the argument list
21     MPI_Init(&argc, &argv);
22
23     // Determine the rank of the current process
24     int rank;
25     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
26
27     // Determine the number of processes
28     int size;
29     MPI_Comm_size(MPI_COMM_WORLD, &size);
30
31     printf("Hello World! I am process %d out of %d\n", rank, size);
32
33     // Must shut down the MPI system when you're done.
34     // If you don't, there can be lots of junk left behind.
35     MPI_Finalize();
36
37     return 0;
38 }
```

MPI involves a collection of new functions designed for interprocess communications. Those new MPI functions are declared in the header file `mpi.h` on Line 2. The first new MPI function encountered is on Line 21, where the input arguments are passed to the `MPI_Init` function so that it can set up communications. The arguments MPI does not recognize will still be left in the argument list so that they can be processed in the program. Generally, the first line of an MPI program should call `MPI_Init`, and the last line before returning should call `MPI_Finalize`. The function `MPI_Finalize` closes the communications opened by `MPI_Init` and can be seen on Line 35. It's very important to do the clean-up; otherwise it can potentially cause problems on the system.

The rank of a process is the identifying ID for that process and is obtained by calling `MPI_Comm_rank`. Processes can be grouped into smaller units for various reasons; creating groups will be discussed later in Chapter 18. The default group of processes, called `MPI_COMM_WORLD`, consists of all processes requested for the job. The rank of the present process is returned in the variable `rank`, which is passed by reference. The total number of available processes can be obtained by calling the `MPI_Comm_size` function. Like C, the indexing is zero based, so if there are four processes, then the rank values will be 0–3.

## 16.2 • Compiling and Running MPI Code

Compiling Example 16.1 using `gcc` would require a host of include directives, library linkages, etc., that are specific to the machine on which it is done. To simplify this process, a wrapper program, called `mpicc`, is created that handles all that. Thus, compiling the program is simply accomplished by the command

```
$ mpicc -o hello main.c
```

Think of `mpicc` as the MPI version of `gcc`. Under the hood it is still `gcc` or whichever C compiler that was used to build the MPI library.

Similarly, there is a wrapper program for running MPI programs called `mpirun`. When an MPI program is run, the number of processes to be used is defined at runtime by using the argument `-np #`, where `#` is the number of processes. This argument is not something the program would handle, necessarily, but is something that MPI needs to know so it can set up all the communication patterns. Because of this, the arguments passed to the program must first be passed to the MPI software so it can process the options that it understands and/or expects. This is accomplished on Line 21, where `argc` and `argv` are passed by reference to `MPI_Init` first. The program will only be left with the arguments given after the executable name, in this case `hello`. To run this program on four processes at the terminal, type

```
$ mpirun -np 4 hello
```

A sample output of running this program is shown here:

```
Hello World! I am process 3 out of 4
Hello World! I am process 0 out of 4
Hello World! I am process 1 out of 4
Hello World! I am process 2 out of 4
```

Note that each of the processes zero through three reported their status and that there are a total of four processes being used in this run. Running this program a second time would produce the same output, except the order the processes report their output may be different. This shows how the different processes are independent and are not synchronized automatically.

## 16.3 • Submitting Jobs

Computer clusters are a shared resource so everyone must take turns to execute their programs. Because the type of jobs to be run on such systems are computationally intensive and may push the very limits of the resources available, and may degrade very rapidly if having to share resources, clusters are set up so that programs are run sequentially rather than simultaneously. Simple things, such as editing files, compiling code, or other tasks, are handled on a special computer called the head node, which runs very much the same as a regular computer, handling multiple tasks at the same time. The compute nodes are separate and where the parallel algorithms are executed; they are not shared.

To run a program on the compute nodes requires a file called a job, and to run a job it must be submitted to a scheduler, also called a queuing system. The scheduler takes many factors into account about the jobs in the queue, such as how many cores, how much memory, how much time will be required to complete the job, and what the user's priority status is (i.e., how much did you pay to run this job?). The latter data is maintained by the system, but the other information is all required in order to submit a job to the scheduler.

There are a variety of job scheduling systems available, and the commands for running those systems are not standard. For purposes of the discussion here, we will use

the scheduler called Moab. More detailed information about Moab can be found on their website:

<http://www.adaptivecomputing.com>

A job is a shell script that enumerates the resources required and the sequence of commands necessary to run a program. A shell script is yet another language to learn, but very little of it is needed for the shell script. A very simple shell script is shown below.

### Example 16.2.

```

1 #!/bin/bash
2 #MSUB -N <name of job>
3 #MSUB -A <account number>
4 #MSUB -m abe
5 #MSUB -l nodes=<N>:ppn=<p>
6 #MSUB -l walltime=<hh>:<mm>:<ss>
7 #MSUB -q <queue name>
8 cd $PBS_O_WORKDIR
9 module load mpi
10 mpirun -np <N*p> <program name> <program arguments>
```

Let's go through this script line by line. Note that entries in the file surrounded by angle brackets are values to be inserted by the user. For example, to request 8 nodes with 16 processes per node, Line 5 would be

```
5 #MSUB -l nodes=8:ppn=16
```

**Line 1:** This is simply an alert to the system that this is a bash shell script that can be run as an executable. It is required to be written as shown in the first line of your file.

**Line 2:** The name of the job is set here; when checking on the status of a job, or when a job is finished and sends an email, this name is what will be used as a reference.

**Line 3:** The user account number is put here.

**Line 4:** This line determines when information about the job will be emailed. For this to work, create the file `~/.forward` which contains an email address where email notices will be sent. Alternatively, the email address where messages are to be sent can be specified with the additional line

```
#MSUB -M <email address>
```

The letters `abe` refer to abort, begin, and end, corresponding to the events for which messages will be generated. Any subset of those letters can be used and in any order.

**Line 5:** This states how many computer resources the job will require. The number of nodes requested is `N`, and `p` is the number of processes per node. Check the hardware to find out the maximum number of cores per node because if `p` is too big, then the job either will be rejected or will have significant performance degradation due to multiple processes sharing time on a node. Communication

**Table 16.1.** Available queues on Northwestern University's Quest system. The listed amounts are maximums for that queue. The fewer resources requested, the sooner the job will be executed. The names and parameters for the queues vary for each system; check with your system administrator to learn what the queues are for your system.

Queue	Time Limit	Max Cores
Short	4 hours	1024
Normal	48 hours	1536
Long	7 days	512

costs within a single node are *much* cheaper than between different nodes, so ideally keep  $N$  as small as possible and  $p$  as large as possible within the confines of the system configuration. In the end, when the MPI code is run with the option “-np #”, the number specified must be less than or equal to the number of nodes times the number of processes per node requested on this line.

**Line 6:** This line specifies the length of time expected for the code to run to completion. If communications are not correct, it can easily cause a problem called blocking, where the program freezes because the communications are interdependent and waiting on each other and hence unable to proceed. By giving the job a time limit, if the program blocks the scheduler will automatically terminate the job when the time limit is hit. It's a useful backstop to keep the program from idling indefinitely. The amount of time allocated is important because it will also determine which queue the job will end up in when submitted. If less time is specified than required, the job will terminate prematurely. If too much time is specified, the job could end up in a longer queue, which may mean the program may have a lower priority to get executed. Providing a fair estimate of what resources are needed is the best way to achieve a positive outcome in as little time as possible.

**Line 7:** Depending on the system configuration, this line may be optional or unnecessary. Different systems have different definitions for their queues, but the basic idea is that jobs are scheduled according to their priority. Short, fast small jobs are typically scheduled quicker than long slow and large jobs because they can fit in easier. Imagine a tray of blocks, where each of the blocks are different sizes. The scheduler has to piece together these blocks in the tray, where there is continuous turnover of blocks. The scheduler sorts the blocks by size to make it easier. Bigger blocks may have to wait longer for enough space to free up for it to fit on the tray. Small blocks can find spaces between the big blocks to get their work done. A sample of a set of three queues is shown in Table 16.1 with highest priority given to short jobs and lower priority for long jobs. The configuration for your system will certainly vary from this.

**Line 8:** This line will move the current working directory to the directory where this script is located. This way data files saved from the program will be stored in that directory.

**Line 9:** On some systems, the collections of libraries being used may have to be specifically loaded onto the system beforehand. Because this script will not be run interactively, the environment for running the job has to be set up. Since this program is using MPI, the MPI tools are needed in order to use `mpirun`. Therefore,

the MPI module of libraries are loaded prior to running the program. If using LAPACK, or other libraries, they would be loaded here as well.

**Line 10:** This is where the program is actually executed, and the number of processes should be no more than the number of nodes times the number of processes per node specified on Line 5.

Although it's not required, shell scripts are typically saved in a file with the suffix “.sh”, indicating that it is a shell script.

Once the shell script is created, the next task is to submit the job to the scheduler. Suppose the shell script has the name “`myjob.sh`”; then the job is submitted to the scheduler by issuing the command

```
$ msbatch myjob.sh
```

After the job is submitted, the status of the job can be monitored and managed in the queue using additional commands. The `showq` command will show the current list of jobs waiting in the queue. A specific job status can be checked using the `checkjob` command, and the job can be placed on hold, released from hold, or deleted using the `mjobctl` command. The names of these tools may vary, so check with your system administrator for the corresponding tools.

---

## Exercise

- 16.1. Compile and run the code in Example 16.1. Verify that you are able to access multiple cores. On most clusters, codes are run using job control software that queues jobs before they are run. Familiarize yourself with the steps required to submit jobs to the queue. These tools make it possible for multiple users to have exclusive control of the requested cores so that timing of calculations can be more reliable, leading to less latency.

# Chapter 17

# Passing Messages

There are certainly many applications where the same program must be run many times using varying input parameters in order to achieve its goal. Using a cluster in this instance is sometimes called “poor man’s parallelism” or “embarrassingly parallelizable” because essentially no effort beyond learning how to get multiple versions of the code to run is required in order to achieve perfectly scalable parallelism. These are not the type of applications intended here. Instead, the focus is on tasks that require coordination between processes and on how to get multiple processes to talk to each other by passing messages. This chapter covers the various communications patterns including point-to-point communications between two individual processes and group communications including many-to-one, one-to-many, and many-to-many.

## 17.1 • Blocking Send and Receive

Considering Example 16.1 again, the order in which the processes report can be made sequential by forcing them to talk to each other so that they don’t go too early. The idea is to have the Rank 0 process make its statement, and then send a message to the Rank 1 process that it’s finished. The Rank 1 process will wait for the message that Rank 0 is done, print its message, and then relay that it’s finished to Rank 2, and so on until the last rank is finished. Algorithmically, the strategy for each of the processes is shown in Table 17.1, where the steps for each rank are aligned to the corresponding steps for the others. Note how all the processes share common tasks, but it’s the communications that must be directed specifically.

**Table 17.1.** Steps required for the ordered sequence of printing “Hello” by using communications.

Rank 0	Rank 1	...	Rank $j$	...	Rank $N - 2$	Rank $N - 1$
	Wait for Rank 0	...	Wait for Rank $j - 1$	...	Wait for Rank $N - 3$	Wait for Rank $N - 2$
Print “Hello”	Print “Hello”	...	Print “Hello”	...	Print “Hello”	Print “Hello”
Send message to Rank 1	Send message to Rank 2	...	Send message to Rank $j + 1$	...	Send message to Rank $N - 1$	

The basic message functions come in pairs consisting of a Send and a Receive. The declarations for these functions are shown below, put side by side so that the corresponding information that must be matched are aligned:

<pre>MPI_Send(     void*      message,     int        count,     MPI_Datatype  datatype,     int        dest_rank,     int        tag,     MPI_Comm    comm     )</pre>	<pre>MPI_Recv(     void*      message,     int        count,     MPI_Datatype  datatype,     int        source_rank,     int        tag,     MPI_Comm    comm,     MPI_Status*  status   )</pre>
---	--

The arguments for these functions must match up except for the destination and source designations. The `message` argument is a pointer to the memory that will be sent/received. The length of the memory to be sent/received is the product of the value of `count` times the size of the `datatype`. The `dest_rank/source_rank` values are the rank of the processes that will receive/send the message. The `comm` argument is the communication group within which the rank has meaning, e.g., `MPI_COMM_WORLD`, meaning all processes. The `tag` argument is used for identifying a message. There can be applications where different messages may be coming from the same source and the order may not be predetermined. In that case, the `tag` can help to identify which message type has been received. Finally, the `status` argument is used for diagnostic purposes and can be ignored by using the constant value `MPI_STATUS_IGNORE`.

The steps outlined in Table 17.1 are coded in Example 17.1, where some fictitious data that is the square of the rank is passed along the chain.

### Example 17.1.

```

1 #include <stdio.h>
2 #include <mpi.h> // This is the header file for MPI functions
3 /*
4 int main(int argc, char* argv[])
5
6 The main program is the starting point for an MPI program.
7 This one simply prints each process rank in order. Sorting is done
8 by using message passing to coordinate.
9
10 Inputs: none
11
12 Outputs: Prints "Hello World #" where # is the rank of the process
13 in order.
14
15 */
16
17
18 int main(int argc, char* argv[])
19 {
20     // First thing is to initialize the MPI interface. Some arguments can
21     // be passed through to the MPI interface, so the argument list is
22     // sent by sending the argument list
23     MPI_Init(&argc, &argv);
24
25     // Determine the rank of the current process
26     int rank;
27     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
28 }
```

```

29 // Determine the number of processes
30 int size;
31 MPI_Comm_size(MPI_COMM_WORLD, &size);
32
33 int data;
34 // If not the first process, wait for permission to proceed
35 if (rank > 0) {
36     // Wait for a message from rank-1 process
37     MPI_Recv(&data, 1, MPI_INT, rank-1, 0, MPI_COMM_WORLD,
38             MPI_STATUS_IGNORE);
39     printf("Rank %d has received message with data %d from rank %d\n",
40            rank, data, rank-1);
41 }
42
43 // All processes print hello
44 printf("Hello from rank %d out of %d\n", rank, size);
45
46 // All processes send the go ahead message except the last process
47 if (rank < size-1) {
48     data = rank*rank;
49     // Send the go ahead message to rank+1. Using rank^2 as fake data
50     MPI_Send(&data, 1, MPI_INT, rank+1, 0, MPI_COMM_WORLD);
51 }
52
53 // Must shut down the MPI system when you're done.
54 // If you don't, there can be lots of junk left behind.
55 MPI_Finalize();
56
57 return 0;
58 }
```

Like the previous example, the MPI system is initialized and the rank and size of the system is retrieved. The data to be shared between processes will be stored in `data`. The value in `data` is not needed to accomplish the task, but it is there to show how data would be passed. In this case, the square of the rank is passed through that variable.

Looking at the first row of Table 17.1, the first task for all processes except for Rank 0 is to wait for a message before proceeding. This is done on Line 37, where a request is made for one integer (`MPI_INT`) from the process of one lower rank. The ID tag for the message is arbitrarily set to zero. The status information is not needed here, so it is ignored by using the predefined `MPI_STATUS_IGNORE`. Some text is printed on Line 39, to report when a message is received and with what information in the `data` variable.

The second row of Table 17.1 has every process print the “Hello” message as shown on Line 44.

Finally, the third row of Table 17.1 has every process except the last one send a message to the next process of one higher rank, which is done on Line 50. Note how the function arguments match up with the receive function on Line 37.

At first, it may seem counterintuitive that the receive should be written before the send, but studying Table 17.1 makes it clear why that sequence is appropriate. The proof is in the output shown below:

```

Hello from rank 0 out of 4
Rank 1 has received message with data 0 from rank 0
Hello from rank 1 out of 4
Rank 2 has received message with data 1 from rank 1
```

```
Hello from rank 2 out of 4
Rank 3 has received message with data 4 from rank 2
Hello from rank 3 out of 4
```

The reason this method works is because the style of the communications is called blocking. When blocking communications are used, a given process with a send or receive instruction must stop and wait for the communication to be completed before it is permitted to continue. When different processes are required to synchronize, then blocking communications are one way to force them to wait for each other. There are potential pitfalls with this style of communication. For example, suppose the initial receive in Example 17.1 mistakenly failed to exclude Rank 0, i.e., left out Line 35; then the program would get stuck, unable to proceed, because Rank 0 would wait for a message that will never arrive. However, even when the code is written correctly, blocking communications can also be a hindrance depending on the tasks to be done because time spent waiting for a message is time wasted when it could be put to better use.

## 17.2 • Nonblocking Send and Receive

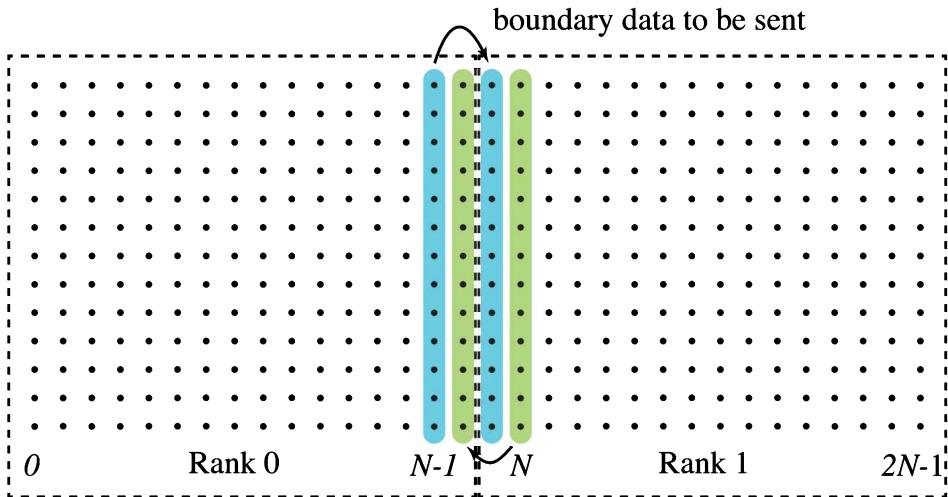
It can be advantageous to continue working on some other task while waiting for the communication transaction to be completed. For example, when the volume of data to be sent via a message is large enough that waiting for it to be copied into its destination location may require a short delay, it would be wise to get the communication transaction started early while completing other tasks so that when the data is needed, it is ready to go. This way, processors are kept busy without letting communications cause delays and hence inefficiency. This is the principle behind nonblocking communications.

Nonblocking communications can offer greater efficiency, depending on the algorithm, but it comes at the price of requiring greater care and a bit of a choreographed dance for the transaction to take place. To begin, look at the steps of this dance where process Rank 0 wants to send a message to process Rank 1:

Rank 0	Rank 1
Identify data to be sent	Identify where data will be stored
Issue nonblocking send command	Issue nonblocking receive command
Do work that <b>does not alter the data to be sent</b>	Do work that doesn't require the data to be received
Wait for the send to be completed	Wait for the receive to be completed
Continue with work, it's safe to alter the data	Continue with work, it's safe to use the received data

To see how this might work in practice, suppose the heat equation is to be solved using a simple explicit finite difference time-stepping method.<sup>6</sup> Rank 0 will work with the grid values  $u_{i,j}^n$  for  $i = 0, \dots, N - 1$  at time step  $n$ , and Rank 1 will work with grid values  $i = N, \dots, 2N - 1$  as illustrated in Figure 17.1. Suppose the initial values of the data are prescribed by an initial data function  $u_{i,j} = f(x_i, y_j)$ . In Figure 17.1, the data in the blue and green shaded points will be duplicated in both domains. The Rank

<sup>6</sup>See Chapter 35 for more information on finite difference methods.



**Figure 17.1.** Simple layout for domain decomposition of a finite difference grid. Rank 0 will send its boundary data to Rank 1, and Rank 1 will do the same in the opposite direction following the arrows. The color-coded data will be the same after the communication transactions. Each rank will do one send and one receive.

Rank 0 process will update the blue domain using the evolution equation, and to do so, it will depend on receiving data from the Rank 1 process to fill out the green shaded data. The more complete algorithm would then look like this:

Rank 0	Rank 1
Allocate array $u_{i,j}^0$ for $i = 0, \dots, N$	Allocate array $u_{i,j}^0$ for $i = N-1, \dots, 2N-1$
Initialize $u_{i,j}^0 = f(x_i, y_j)$ for $i = 0, \dots, N-1$	Initialize $u_{i,j}^0 = f(x_i, y_j)$ for $i = N, \dots, 2N-1$
Begin time step loop	Begin time step loop
Begin nonblocking send of blue data, $u_{N-1,j}^n$ , to Rank 1	Begin nonblocking send of green data, $u_{N,j}^n$ , to Rank 0
Begin nonblocking receive of green data, $u_{N,j}^n$ , from Rank 1	Begin nonblocking receive of blue data, $u_{N-1,j}^n$ , from Rank 0
Compute $u_{i,j}^{n+1}$ for $i = 0, \dots, N-2$	Compute $u_{i,j}^{n+1}$ for $i = N+1, \dots, 2N-1$
Wait for both nonblocking send and receive to be completed	Wait for both nonblocking send and receive to be completed
Compute $u_{N-1,j}^{n+1}$	Compute $u_{N,j}^{n+1}$
Go back to top of loop	Go back to top of loop

There are a few observations to make about this algorithm. First, when initializing the data, there's no reason why the initial data can't be set in the overlap region at the beginning. For example, the data for Rank 0 could have been initialized for  $i = 0, \dots, N$  and then skipped the first send/receive pair. The cost savings would be minimal since

this would be valid only on the first time step, and at the same time, it would have made writing the algorithm a little more complicated, so that change was not used. In practice, either way is acceptable since, as noted, the cost savings would be minimal.

The second observation is that in the algorithm where each rank is to wait for communications, it is very unlikely that much waiting will actually take place because by the time the updates in the interior of the two subdomains are completed, the boundary data will have been transmitted and received.

The third observation is that when sending/receiving data, the memory space must be contiguous. That means that when planning the ordering of the arrays, the memory locations for  $u_{N,j}^n$  and  $u_{N,j+1}^n$  must be adjacent. In this simple example, it's easy to do, but for a three-dimensional domain subdivided into  $4^3$  cubic subdomains, it gets much more difficult. In that case, it will be better to subdivide the domain into slabs segmenting the first dimension. If cubic subdomains are necessary, then a special buffer would be needed to hold the data before it is sent to the neighboring process. There is another good reason for doing this, which concerns the next observation.

The fourth observation is that in the waiting step, **both the send and the receive must be completed before proceeding**. Obviously, Rank 0 doesn't want to update the  $i = N - 1$  data until the boundary data at  $i = N$  has been received. At the same time, even if the data at  $i = N$  is ready to go, the data at  $i = N - 1$  in Rank 0 should not be updated until the send is also complete, because that data won't be read until the send is completed. Thus, if the data  $u_{N-1,j}^n$  gets updated to  $u_{N-1,j}^{n+1}$  before the send is complete, Rank 1 will receive the  $u_{N-1,j}^{n+1}$  rather than the time step  $n$  data, which would lead to errors. One way to navigate around this restriction is to create a separate data transfer buffer that is used as temporary storage for nonblocking communications. In practical terms, a separate array, call it  $v_j$ , just for storing  $u_{N-1,j}^n$  could be created. Before doing the nonblocking send, the grid data would be copied into this buffer, i.e.,  $v_j = u_{N-1,j}^n$ , and then the nonblocking send would send the  $v_j$  data rather than  $u_{N-1,j}^n$ . By doing this, the  $u_{N-1,j}^{n+1}$  data can be computed as soon as the  $u_{N-1,j}^n$  is received without waiting for the send to be completed because this would not alter the values in  $v_j$ . This is an important detail to keep in mind about nonblocking communications, but in applications like this, where a sizable amount of work can be done while waiting for the communications to be completed, the communications will almost certainly be finished by the time the boundary update is to be done.

This algorithm is implemented in Example 17.2 for a one-dimensional heat flow problem.

### Example 17.2.

```

1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <stdbool.h>
4 #include <mpi.h> // This is the header file for MPI functions
5
6 /*
7  int main( int argc, char* argv[] )
8
9 Heat flow solver. All data is initially zero, boundary conditions are
10 provided as input
11
12 Inputs: argc should be 2
13      argv[1]: number of grid points
14 User prompted for left and right boundary conditions
15
16 Outputs: Prints out the final values.

```

```
17
18 */
19
20 int main(int argc, char* argv[])
21 {
22     // First thing is to initialize the MPI interface.
23     // Some arguments can be passed through to the MPI interface,
24     // so the argument list is sent by sending the argument list
25     MPI_Init(&argc, &argv);
26
27     int N = atoi(argv[1]); // Get the number of grid points
28
29     // Determine the rank of the current process
30     int rank;
31     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
32
33     // Determine the number of processes
34     int size;
35     MPI_Comm_size(MPI_COMM_WORLD, &size);
36
37     // allocate memory. Each core should have roughly N/size + 2 grid
38     // points, but if N is not evenly divisible by size, give one extra
39     // grid point to the low ranks to make up the difference. Also, the
40     // rank=0 and rank=size-1 processes do not have to store boundary
41     // data for a neighbor.
42     int localN = N/size + (N%size > rank ? 1 : 0) + (rank > 0 ? 1 : 0)
43                                     + (rank < size-1 ? 1 : 0);
44     double* u[2];
45     // need two copies to do updates
46     u[0] = (double*)malloc(localN * sizeof(double));
47     u[1] = (double*)malloc(localN * sizeof(double));
48
49     // Initialize the data
50     for (int i=0; i<localN; ++i)
51         u[0][i] = 0.;
52
53     if (rank == 0) {
54         // Read the boundary conditions from the input list.
55         // Get the left boundary condition.
56         u[0][0] = atof(argv[2]);
57         u[1][0] = u[0][0];
58     }
59
60     if (rank == size-1) {
61         // Get the right boundary condition
62         u[0][localN-1] = atof(argv[3]);
63         u[1][localN-1] = u[0][localN-1];
64     }
65
66     // CFL condition: dt < 0.5 dx^2
67     double dx = 1. / (N-1);
68     double dx2 = dx*dx;
69     double dt = 0.25 * dx2;
70
71     // Storage for tracking communication info
72     MPI_Request sendLeftRequest;
73     MPI_Request sendRightRequest;
74     MPI_Request recvLeftRequest;
75     MPI_Request recvRightRequest;
76
77     // main loop: terminal time is T=1
78     int i, newi, oldi;
79     for (i=0; i*dt < 1.0; ++i) {
```

```

80    newi = (i+1)%2;
81    oldi = i%2;
82    // Exchange end data to the left ,
83    //      the tag will correspond to the step #.
84    if (rank > 0) {
85        MPI_Isend(&(u[oldi][1]), 1, MPI_DOUBLE, rank-1, i, MPI_COMM_WORLD,
86                               &sendLeftRequest);
87        MPI_Irecv(&(u[oldi][0]), 1, MPI_DOUBLE, rank-1, i, MPI_COMM_WORLD,
88                               &recvLeftRequest);
89    }
90    // Exchange end data to the right ,
91    //      the tag will correspond to the step #
92    if (rank < size-1) {
93        MPI_Isend(&(u[oldi][localN-2]), 1, MPI_DOUBLE, rank+1, i,
94                               MPI_COMM_WORLD, &sendRightRequest);
95        MPI_Irecv(&(u[oldi][localN-1]), 1, MPI_DOUBLE, rank+1, i,
96                               MPI_COMM_WORLD, &recvRightRequest);
97    }
98
99    // Now do update in the interior where it doesn't matter if we have
100   //      the current data from neighboring processes
101   for (int j=2; j<localN-2; ++j)
102     u[newi][j] = u[oldi][j]+dt*(u[oldi][j+1] - 2*u[oldi][j]
103                               + u[oldi][j-1])/dx2;
104
105   // Can update points next to boundary
106   if (rank == 0)
107     u[newi][1] = u[oldi][1]+dt*(u[oldi][2] - 2*u[oldi][1]
108                               + u[oldi][0])/dx2;
109   if (rank == size-1)
110     u[newi][localN-2] = u[oldi][localN-2]+dt*(u[oldi][localN-1]
111                               - 2*u[oldi][localN-2] + u[oldi][localN-3])/dx2;
112
113   // Now check to see boundary data is ready.
114   // Indicate whether data is ready {sent left , received left ,
115   //      sent right , received right}
116   int ready[4] = {0, 0, 0, 0};
117   // Indicate that the update of the endpoint has been done after the
118   //      data transfer.
119   bool done[2] = {false, false};
120
121   // There is no data transfer at the ends , so mark those as ready.
122   if (rank == 0) {
123     ready[0] = 1;
124     ready[1] = 1;
125   }
126   if (rank == size-1) {
127     ready[2] = 1;
128     ready[3] = 1;
129   }
130
131   // Keep checking until data is ready and endpoints updated.
132   while (!done[0] || !done[1]) {
133
134     // Check whether interchange of left data is complete
135     if (rank > 0 && !done[0]) {
136       if (!ready[0])
137         MPI_Test(&sendLeftRequest, &(ready[0]), MPI_STATUS_IGNORE);
138       if (!ready[1])
139         MPI_Test(&recvLeftRequest, &(ready[1]), MPI_STATUS_IGNORE);
140     }
141

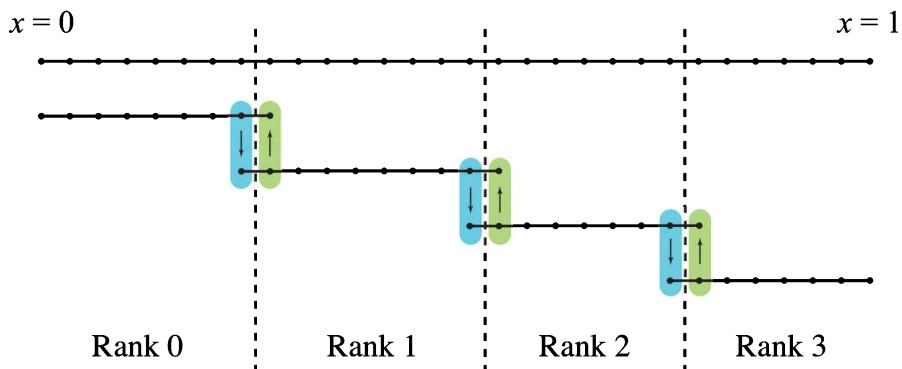
```

```

142      // If the data is exchanged and endpoint hasn't been updated yet,
143      // then update it.
144      if (ready[0] && ready[1] && !done[0]) {
145          // data on the left has been sent and received, it's safe to
146          // update now
147          u[newi][1] = u[oldi][1] + dt*(u[oldi][2] - 2*u[oldi][1]
148                                         + u[oldi][0])/dx2;
149          done[0] = true;
150      }
151
152      // Check whether interchange of right data is complete
153      if (rank < size-1 && !done[1]) {
154          if (!ready[2])
155              MPI_Test(&sendRightRequest, &(ready[2]), MPI_STATUS_IGNORE);
156          if (!ready[3])
157              MPI_Test(&recvRightRequest, &(ready[3]), MPI_STATUS_IGNORE);
158      }
159
160      // If the data is exchanged and endpoint hasn't been updated yet,
161      // then update it
162      if (ready[2] && ready[3] && !done[1]) {
163          // data on the right has been sent and received, it's safe to
164          // update now
165          u[newi][localN-2] = u[oldi][localN-2] + dt*(u[oldi][localN-1]
166                                         - 2*u[oldi][localN-2] + u[oldi][localN-3])/dx2;
167          done[1] = true;
168      }
169  }
170
171
172 // We have to gather all the data from the various
173 // processes so it can be printed.
174 // We'll send all the data to rank==0.
175 // We'll see a better way to do this later.
176 if (rank == 0) {
177     // allocate space for the full array
178     double* finalu = (double*)malloc(N * sizeof(double));
179
180     // copy the local data to the full array
181     for (int j=0; j<localN-1; ++j)
182         finalu[j] = u[newi][j];
183
184     // Track where the next array data will be inserted in the
185     // full array
186     int nextj = localN-1;
187     int datalen;
188
189     // request the data from each of the other processes,
190     // appending as we go.
191     for (int r=1; r<size; ++r) {
192         // amount of data in rank=r process excluding internal boundary
193         // data
194         datalen = N/size + (N%size > r ? 1 : 0);
195         MPI_Recv(&(finalu[nextj]), datalen, MPI_DOUBLE, r, 0,
196                  MPI_COMM_WORLD, MPI_STATUS_IGNORE);
197         nextj += datalen; // update the new insertion point
198     }
199
200     // Store the final array in binary format to file finalu.dat
201     FILE* fileid = fopen("finalu.dat", "w");
202     fwrite(finalu, sizeof(double), N, fileid);
203     fclose(fileid);
204 }
```

```

205 // Release the final u allocation of memory
206 free(finalu);
207
208 } else {
209 // All ranks other than 0 must send their interior data to rank 0
210 // The last rank doesn't have a boundary point,
211 // so it's one bigger than the others.
212 if (rank < size - 1)
213 MPI_Send(&(u[oldi][1]), localN - 2, MPI_DOUBLE, 0, 0,
214 MPI_COMM_WORLD);
215 else
216 MPI_Send(&(u[oldi][1]), localN - 1, MPI_DOUBLE, 0, 0,
217 MPI_COMM_WORLD);
218 }
219
220 free(u[0]);
221 free(u[1]);
222
223 // Must shut down the MPI system when you're done.
224 // If you don't, there can be lots of junk left behind.
225 MPI_Finalize();
226
227 return 0;
228 }
```



**Figure 17.2.** Diagram of local data allocation for the one-dimensional heat equation. The colored points indicate points that must be synchronized across processes, and the arrow indicates the flow of the communication. The processes update the heat equation only for the nodes between the dotted lines.

Example 17.2 begins by getting the number of grid points to be used and by initializing the MPI system. Remember, it's important to call `MPI_Init` first before anything else is done.

On Lines 42–47, the local data space is allocated. The data is allocated following the diagram in Figure 17.2. For simplicity the forward Euler time step method is used, so two copies of the data are needed. The data is initialized in Lines 50–51. In this case, the initial value is zero everywhere in the interior, and the boundary conditions are read from the input parameters. The space and time step sizes are determined on Lines 67–69 following the CFL condition [17].<sup>7</sup>

<sup>7</sup>The Courant–Friedrichs–Lewy condition specifies the maximum stable time step for an explicit method for solving a PDE. For the heat equation as shown here, the restriction is  $\Delta t \leq \Delta x^2/2$ .

The two matching functions for nonblocking communications are `MPI_Isend` and `MPI_Irecv`, and their call statements are similar to the blocking send and receive from before:

<code>MPI_Isend(</code> <code>void* message,</code> <code>int count,</code> <code>MPI_Datatype datatype,</code> <code>int dest_rank,</code> <code>int tag,</code> <code>MPI_Comm comm,</code> <code>MPI_Request* request )</code>	<code>MPI_Irecv(</code> <code>void* message,</code> <code>int count,</code> <code>MPI_Datatype datatype,</code> <code>int source_rank,</code> <code>int tag,</code> <code>MPI_Comm comm,</code> <code>MPI_Request* request )</code>
--	--

The communications are not complete with these calls—they are just requests to start the process of communications. As discussed above, the data that is involved in a send or receive pair should not be used before the transaction is completed, so that means it must be checked for completion. That is what the `MPI_Request` object is for, to track all outstanding communication requests.

Like the send and receive, there are blocking and nonblocking checks on the status of communications. The blocking check is

```
MPI_Wait(MPI_Request* request, MPI_Status* status);
```

When `MPI_Wait` is used, the request to be checked is given. The program will then stop and wait at this point until the requested transaction is completed.

Example 17.2 has two sets of requests for each end of the grid, and the two ends are independent of each other. To avoid waiting for one end while the other end may be ready, it is better to use a nonblocking check and continue to loop through open communications and responding accordingly as they become available. A nonblocking check is done by using `MPI_Test`:

```
MPI_Test(MPI_Request* request, int* ready, MPI_Status* status);
```

The value of `ready` will be nonzero if the communication is complete and zero if it is incomplete. With this information in hand, the rest of Example 17.2 is straightforward.

As discussed in the algorithm outline, the first thing to do in the main loop is initiate the communications of boundary data, which is done on Lines 84–97. Depending on the local rank, a request to send and a request to receive boundary data for one or both ends of the local domain are initiated. Once that’s done, the interior nodes in the local domain can be updated because they don’t depend on the received boundary data, and they won’t alter the sent boundary data. This is important to remember because the data involved in communications must not be altered before being sent and must not be used before being received.

The indexing in the arrays on Line 102 and following may look a little odd. Copying data from the “new” array back into the “old” array at the end of every loop is a waste of time. It can be avoided by just going back and forth. On Line 102 of Example 17.2, the first time through the loop, `i = 0`, so `oldi = 0` and `newi = 1`, and hence `u[1]` is getting the update from `u[0]`. The next time through the loop, `i = 1`, and following modular arithmetic `oldi = 1` and `newi = 0`, so `u[0]` is on the left side and `u[1]` is on the right side.

Once the interior update is complete, the next step is to deal with completing the communications started at the top of the loop. The initial assumption is that none of

the transactions are ready and that the endpoint updates have not been done (except at  $x = 0, 1$ ). The program then begins looping through the communications, checking on them as it goes. On Lines 135–140, the left end is checked only so long as it isn't ready. If the communications are ready and the endpoint value hasn't already been updated, then it is updated on Lines 147–149. Lines 153–168 are the same construction but for the right end. Once both ends are updated, the communication loop terminates and the program moves on to the next time step.

Finally, on Lines 176–218, the data is gathered in order to save to a file. Remember, no single process has a complete set of the data, so the data has to be sent to the lead process, which is designated as Rank 0. The Rank 0 process creates a single array to store the data on Line 178, and then copies its local data to it on Lines 181–182. For simplicity here, blocking communications are used to retrieve the local data from each of the other processes so that it can be stored in the full array. Rank 0 will be the recipient using Lines 191–198, while the remaining processes must send their data, which is on Lines 212–218. Once all the data is received in the Rank 0 process, it is written to a single file on Lines 201–203. There is a better way to gather the data together, and that will be discussed in Section 17.4.

### 17.3 • Combined Send and Receive

The communications in the algorithm described in Section 17.2 and implemented in Example 17.2 essentially involve a very simple concept: two processes wish to swap information with each other. Using nonblocking communications as described above is very efficient but also involves a fair amount of code to implement. These kinds of data swaps are common enough in scientific computing that it is worth trying to condense these data swaps into a more compact form, and this is accomplished by the function `MPI_Sendrecv`:

```
int MPI_Sendrecv( void* sendData,
                  int sendCount,
                  MPI_Datatype sendDataType,
                  int dest_rank,
                  int send_tag,
                  void* receiveData,
                  int receiveCount,
                  MPI_Datatype receiveDataType,
                  int source_rank,
                  int receive_tag,
                  MPI_Comm comm,
                  MPI_Status* status )
```

This function combines a send and a receive pair in the same function. For example, suppose the Rank 0 process and the Rank 1 process want to exchange an integer in the variable `x` and store the other process's value in the variable `y`. It can be accomplished in one call:

```
other = (rank+1)%2;
MPI_Sendrecv(&x, 1, MPI_INT, other, 0,
            &y, 1, MPI_INT, other, 0, MPI_COMM_WORLD, &status);
```

It's equivalent to the sequence

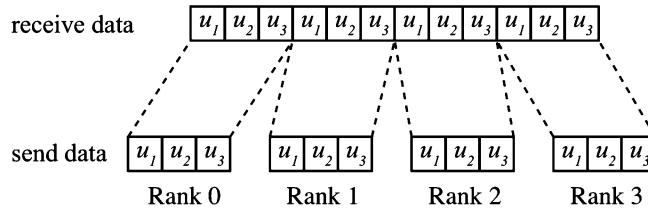
```
other = (rank+1)%2;
MPI_Isend(&x, 1, MPI_INT, other, 0, MPI_COMM_WORLD, &request1);
MPI_Irecv(&y, 1, MPI_INT, other, 0, MPI_COMM_WORLD, &request2);
MPI_Wait(&request1, &status);
MPI_Wait(&request2, &status);
```

Example 17.3 is the result of substituting the following lines in place of Lines 71–170 in Example 17.2.

### Example 17.3.

```
// Storage for tracking communication info
71 MPI_Status status;
72
73
74 // main loop: terminal time is T=1
75 int i, newi, oldi;
76 for (i=0; i*dt < 1.0; ++i) {
77     newi = (i+1)%2;
78     oldi = i%2;
79     // Exchange end data
80     if (rank > 0 && rank < size-1) {
81         // Move data to the left
82         MPI_Sendrecv(&(u[oldi][1]), 1, MPI_DOUBLE, rank-1, i,
83                         &(u[oldi][localN-1]), 1, MPI_DOUBLE, rank+1, i,
84                         MPI_COMM_WORLD, &status);
85         // Move data to the right
86         MPI_Sendrecv(&(u[oldi][localN-2]), 1, MPI_DOUBLE, rank+1, i,
87                         &(u[oldi][0]), 1, MPI_DOUBLE, rank-1, i,
88                         MPI_COMM_WORLD, &status);
89     }
90     // Handle end cases
91     if (rank == 0 && size > 0) {
92         // if data moves left, then rank 0 receives from the right,
93         // but does not send left
94         MPI_Recv(&(u[oldi][localN-1]), 1, MPI_DOUBLE, 1, i,
95                         MPI_COMM_WORLD, &status);
96         // if data moves right, then rank 0 sends to the right,
97         // but does not receive from left
98         MPI_Send(&(u[oldi][localN-2]), 1, MPI_DOUBLE, 1, i,
99                         MPI_COMM_WORLD);
100    }
101    if (rank == size-1 && size > 0) {
102        // if data moves left, then rank size-1 sends to the left
103        MPI_Send(&(u[oldi][1]), 1, MPI_DOUBLE, rank-1, i, MPI_COMM_WORLD);
104        // if data moves right, then rank size-1 receives from the left
105        MPI_Recv(&(u[oldi][0]), 1, MPI_DOUBLE, rank-1, i, MPI_COMM_WORLD,
106                         &status);
107    }
108    // Now do the update
109    for (int j=1; j<localN-1; ++j)
110        u[newi][j] = u[oldi][j]+dt*(u[oldi][j+1]-2*u[oldi][j]
111                                + u[oldi][j-1])/dx2;
112 }
```

When using MPI\_Sendrecv, it is important to make sure that the send and the receive match up correctly or the function will block the execution. In the simple exchange of integers between two processes, each process needs exactly one send and one receive message to/from the other process. In that case, it's simple to match the send



**Figure 17.3.** Diagram of the MPI\_Gather operation. The different processes send their data to the destination process, where it is assembled into a single concatenated array.

and the receive. However, in Example 17.3, the situation is a bit more complicated because Rank 0 sending data to the left is not matched with receiving from the left. If data is sent to the left, then it must be received from the right. Thus, on Line 82, when data moves to the left, the send part is sent to rank-1, but the receive part is from rank+1. Note also that only the interior ranks communicate in both directions, hence Line 80 excludes the end ranks. The pattern is completed by handling the end cases, where Rank 0 receives data when it is moving left and sends data when it is moving right. The opposite is true for the Rank size-1 process, where it sends data when it is moving left and receives data when it is moving right. Failure to match the send and receives will cause the program to block and hence fail.

## 17.4 • Gather/Scatter Communications

Example 17.2 was perhaps a little too general because it allowed for different processes to have different sized subdomains. For purposes of load balancing, it is better to require each subprocess to be responsible for exactly the same number of nodes. With that constraint, the process of gathering all the data into a single array, as was done on Lines 176–218, can be simplified down to a single command: MPI\_Gather.

```

195 double* finalu = NULL;
196 if (rank == 0) {
197     // Allocate space for the full array
198     finalu = (double*)malloc(N * sizeof(double));
199 }
200 MPI_Gather(&(u[i % 2][1]), localN - 2, MPI_DOUBLE, finalu, localN - 2,
201             MPI_DOUBLE, 0, MPI_COMM_WORLD);

```

This operation is an example of a many-to-one type of communication and is illustrated in Figure 17.3 for the case where localN is five and there are four processes. This single call collects all the data from each of the processes and assembles them into the root process as a single array.

The arguments for MPI\_Gather are given here:

```

int MPI_Gather( void*           sendData,
                int            sendCount,
                MPI_Datatype   sendDataType,
                void*          receiveData,
                int            receiveCount,
                MPI_Datatype   receiveDataType,
                int            dest_rank,
                MPI_Comm       comm            )

```

Under most circumstances, the send count and the receive count will be the same because you want the data to have no gaps, and also the send and receive data types will be the same as well. The sent data refers to the local data space, while the received data points to the full array. The destination rank is the process that will receive the full array, which in this case is Rank 0, and hence only Rank 0 need allocate memory for the full array.

This is a useful method for gathering all the data into a single process, but what if the gathered data should be available to all processes? The resulting array could be sent to each of the processes from the destination, but a better method would be to use `MPI_Allgather`. The argument list is the same as for `MPI_Gather` except the destination rank is omitted:

```
int MPI_Allgather( void* sendData,
                   int sendCount,
                   MPI_Datatype sendDataType,
                   void* receiveData,
                   int receiveCount,
                   MPI_Datatype receiveDataType,
                   MPI_Comm comm )
```

At the end of this operation, every process that contributed to the concatenated array will receive a copy of the full array. Thus, every process would have to allocate space to store it as well.

Recall that in Example 17.2 the actual buffer sizes were not necessarily all the same, while `MPI_Gather` requires them all to be the same. The function `MPI_Gatherv` is used for the more general case when not all buffers are the same size:

```
int MPI_Gatherv( void* sendData,
                  int sendCount,
                  MPI_Datatype sendDataType,
                  void* receiveData,
                  int* receiveCount,
                  int* displacement,
                  MPI_Datatype receiveDataType,
                  int dest_rank,
                  MPI_Comm comm )
```

The difference is in the receive data, where the receive count is a list of integers indicating how much data to expect from each rank, and the displacement array lists the index in the full array that each rank should place its data. So typically,

```
displacement[0] == 0
displacement[1] == receiveCount[0]
displacement[n+1] == displacement[n] + receiveCount[n]
```

Only the destination rank must allocate memory for the receive count and displacement arrays; the sending processes can use `NULL`.

In keeping with the theme of symmetry in MPI operations, there must be an opposite to gathering, and that is scattering. For example, suppose the initial data for the heat equation problem is read from a file. In that case, the data would be read by the Rank 0

process and then it would send the pieces of data to each of the other processes, exactly the opposite of the gather. The argument list looks very similar to the `MPI_Gather` function:

```
int MPI_Scatter( void* sendData,
                  int sendCount,
                  MPI_Datatype sendDataType,
                  void* receiveData,
                  int receiveCount,
                  MPI_Datatype receiveDataType,
                  int send_rank,
                  MPI_Comm comm )
```

One difference from `MPI_Gather` is that the destination rank is replaced by the source rank, the process that has the original full data set to be distributed. As for `MPI_Gather`, the send and receive counts should be the same, as well as the send and receive data types. For the case where the data is not distributed evenly, use the `MPI_Scatterv` function:

```
int MPI_Scatterv( void* sendData,
                  int* sendCount,
                  int* send_offset,
                  MPI_Datatype sendDataType,
                  void* receiveData,
                  int* receiveCount,
                  MPI_Datatype receiveDataType,
                  int src_rank,
                  MPI_Comm comm )
```

The amount of data to send to each of the other processes is given by `sendCount`, and the `send_offset` array gives the starting index within `sendData` that is sent to each process. Again, all processes must be sure to allocate sufficient space for the data before this function is called.

## 17.5 • Broadcast and Reduction

As discussed in Part I, it is best to control simulations by using a parameter file to contain the input information for each run. When this is done, it is not recommended that each process read the file but rather have the Rank 0 process read the file and then distribute the information to the rest of the processes. Of course, point-to-point communications of send/receive pairs could be used to distribute the parameter values, but that would require many communications calls. Instead, a better method is to broadcast, where one process sends a message to all at the same time.

The declaration of the broadcast is given by

```
int MPI_Bcast( void* data,
                int count,
                MPI_Datatype dataType,
                int src_rank,
                MPI_Comm comm )
```

**Table 17.2.** Sample list of operations available for the MPI\_Reduce function.

Operator Name	Usage
MPI_MAX	Maximum value
MPI_MIN	Minimum value
MPI_SUM	Sum of the values
MPI_PROD	Product of the values
MPI_LAND	Logical and value
MPI_LOR	Logical or value
MPI_LXOR	Logical exclusive or value

After this call, all processes will contain a copy of the data. That means that each process must allocate space to contain the data as well. For example, the following code sends the time step size `dt` to all the processes:

```
double dt;
if (rank == 0)
    dt = 0.25/dx/dx;
MPI_Bcast(&dt, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
// dt is now set for all processes
```

Keeping the theme of symmetry, if a broadcast is a one-to-many communication, then the opposite of a broadcast might be a many-to-one communication, called a reduction. A reduction operation is where many processes contribute data to an aggregate in a single process. For example, suppose the maximum value of a variable across all processes, or the sum of all the values across the processes is needed. This is where a reduction is useful. The function `MPI_Reduce` aggregates the values of a variable across the processes, and a collection of operators are available as a separate argument. The function declaration is

```
int MPI_Reduce( void*           data,
                 void*           result,
                 int            count,
                 MPI_Datatype   dataType,
                 MPI_Op          operator,
                 int            dest_rank,
                 MPI_Comm        comm      )
```

In its simplest usage, assume that `data` is a pointer to a single value which every process has; then `count` will be one, and the length of storage for `result` will be one. Even though both the `data` and the `result` are defined using the generic `void*`, make sure that both are defined to be of the same type. In other words, values of type `double` cannot be reduced directly into a variable of type `float`. The rest of the arguments are similar to the gather operation except for a new argument `operator` of type `MPI_Op`. This argument describes the reduction operation to be performed. A list of the most common operators is given in Table 17.2; many more operators, and even user-defined operators, can be used. The result of the operation will be available to the destination process only. If `data` and `result` are arrays, then the operation is applied element-wise, e.g., if the operator `MPI_SUM` is used, then `result` will be an array where the first entry contains the sum of the first entries of `data`, the second entry contains the sum of the second entries of `data`, and so forth. To make the result available to all processes, use the function `MPI_Allreduce`, which has all the same arguments excluding the destination rank.

## 17.6 • Error Handling

With the exception of the functions `MPI_Wtime` and `MPI_Wtick`, all the MPI functions discussed in this text return an integer value that reports whether the function had an error. The default way that MPI handles errors is to abort the job when an error occurs, which is why the return values are typically ignored. However, a more robust program would include an error handler that will report the location and type of error, and then exit more gracefully. When a program is solid and speed is of greater importance, then it would also be handy to be able to turn off the error handling without editing the code.

One way to accomplish this is to use an error handler that can be turned on or off. For MPI, there are two parts to developing an error handler. The first part is to inform MPI that the default error handling process should be replaced with a user-defined one that will check the error values returned from MPI functions. This is accomplished with the function

```
MPI_Comm_set_errhandler (MPI_COMM_WORLD, MPI_ERRORS_RETURN) ;
```

Next, an error handler can be added to an MPI program by creating a header file called, for example, “`mp ierrors.h`” that is included in files that contain MPI functions and contains the following code:

### Example 17.4.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #ifdef DO_ERROR_CHECKING
5
6 static void CHECK_ERROR( int err, const char* file, int line ) {
7     if (err != MPI_SUCCESS) {
8         int len;
9         char errstring [MPI_MAX_ERROR_STRING];
10        MPI_Error_string(err, errstring, &len);
11        printf("%s in %s at line %d\n", errstring, file, line);
12        fflush(stdout);
13        MPI_Finalize();
14        exit(1);
15    }
16 }
17 #define CheckError( err ) (CHECK_ERROR(err, __FILE__, __LINE__))
18
19 #else
20
21 #define CheckError( err ) err
22
23#endif
```

The compiler directive `#ifdef` on Line 4 checks whether `DO_ERROR_CHECKING` is defined, and if so, then Lines 6–17 are compiled. Otherwise, Line 21 is compiled. Either way, the macro `CheckError` is defined.

To see how this works, suppose a file named “`mpiprog.c`” includes “`mp ierrors.h`” described above and is compiled with the line

```
$ mpicc -c mpiprog.c -DDO_ERROR_CHECKING
```

Suppose also the file `mpiprog.c` contains the following line:

```
17 CheckError( MPI_Send( data, N, MPI_DOUBLE, rank-1, 0, MPI_COMM_WORLD ) );
```

Since the macro `DO_ERROR_CHECKING` is defined, then according to Line 17 in Example 17.4, the compiler preprocessor would replace this with

```
17 CHECK_ERROR( MPI_Send( data, N, MPI_DOUBLE, rank-1, 0, MPI_COMM_WORLD),
18                                     __FILE__, __LINE__ );
```

which is then further processed by the preprocessor to be

```
17 CHECK_ERROR( MPI_Send( data, N, MPI_DOUBLE, rank-1, 0, MPI_COMM_WORLD),
18                                     "myprog.c", 17 );
```

With this now in place, when `MPI_Send` returns an error status value, it is passed to the function `CHECK_ERROR` with the filename and line number. If the error status is not `MPI_SUCCESS`, then an error message is retrieved using `MPI_Error_string` on Line 10 of `mpierrors.h` so that the error message along with the filename and line number in the file are reported before MPI is shut down.

On the other hand, if `DO_ERROR_CHECKING` is not defined on the compile line, then Line 21 of `mpierrors.h` is used, which replaces

```
17 CheckError( MPI_Send( data, N, MPI_DOUBLE, rank-1, 0, MPI_COMM_WORLD ) );
```

with

```
17 MPI_Send( data , N, MPI_DOUBLE, rank -1, 0 , MPI_COMM_WORLD );
```

In other words, the returned error value is ignored.

To make this all consistent, the line with `MPI_Comm_set_errhandler` also should be used only if `DO_ERROR_CHECKING` is defined, so it should be entered this way:

```
#ifdef DO_ERROR_CHECKING
    MPI_Comm_set_errhandler (MPI_COMM_WORLD, MPI_ERRORS_RETURN );
#endif
```

## Exercises

- 17.1. Modify Example 17.1 so that the Rank 3 process sends a message back to Rank 0, and the process is repeated N times. Each rank can count the number of times it's been executed and then terminate when it reaches N. The output should look like this:

```
Hello from rank 0 out of 4: iteration 1
Rank 1 has received message with data 0 from rank 0
Hello from rank 1 out of 4: iteration 1
Rank 2 has received message with data 1 from rank 1
Hello from rank 2 out of 4: iteration 1
Rank 3 has received message with data 4 from rank 2
Hello from rank 3 out of 4: iteration 1
Rank 0 has received message with data 9 from rank 3
Hello from rank 0 out of 4: iteration 2
Rank 1 has received message with data 0 from rank 0
```

```
Hello from rank 1 out of 4: iteration 2
Rank 2 has received message with data 1 from rank 1
Hello from rank 2 out of 4: iteration 2
Rank 3 has received message with data 4 from rank 2
Hello from rank 3 out of 4: iteration 2
:
Rank 3 has received message with data 4 from rank 2
Hello from rank 3 out of 4: iteration N
```

- 17.2. Repeat the previous exercise, but using nonblocking communications.
- 17.3. Write a program to broadcast an integer  $N$  to all processes, and have each process compute the variable  $r = N - \text{rank}$ . Then use **MPI\_Reduce** to find the sum, the maximum, and the minimum of the values of  $r$  over all processes. Verify the results are  $P*N - (P-1)*(P-2)/2$ ,  $N$ , and  $N-P$ , where  $P$  is the number of processes.
- 17.4. Write a program where each core creates an array of length  $N$  and populates it with integer values  $1000*\text{rank}+j$  for grid point  $x_j$ . Use **MPI\_Gather** to assemble the data into a single array. Use **MPI\_Scatter** to send the data back to the cores and then verify that the scattered data matches the data sent through the gather.
- 17.5. Repeat the task in Problem 17.4 using **MPI\_Send** and **MPI\_Recv** to do the communications. Compare the time required to complete the communications in both cases.
- 17.6. Write a program that constructs an  $N \times N$  array of type **double complex** where the entries are  $x_{j,k} = j + ik$ . Use **MPI\_Scatter** to distribute the array into  $N/P \times N$  subarrays spread across  $P$  processes. Each process should then print the values of the four corners of its subarray. Then use **MPI\_Gather** to reassemble the array.
- 17.7. Repeat the above exercise in the case where  $N$  is not evenly divisible by  $P$  and use the functions **MPI\_Scatterv** and **MPI\_Gatherv** to do the communications.
- 17.8. Write a program to generate an  $N \times N$  array of random values using a uniform distribution in the range  $[0, 1]$ . Each process should generate its own subarray of values of dimension  $N/P \times N$ , where  $P$  is the number of processes. The same initial seed should be broadcast to all processes, and then each process should add its rank to the seed to seed the generator for that process. Use the function **MPI\_Allgather** to gather the fully assembled array into each process. Verify that each subarray is distinct, and that if the same seed is used twice, then the full array is exactly the same.
- 17.9. Update Example 17.2 to use an error handler as described in Section 17.6. Try modifying the code to generate errors to verify the error checking is working. For example, delete “rank” from Line 85 so that it attempts to send to rank  $-1$  and see what error is generated.

# Chapter 18

# Groups and Communicators

For some applications, it is useful, or sometimes necessary, to subdivide the processes into functional groups. For example, suppose a grid is to be broken down into a two-dimensional array of subgrids, and communications are to be shared within each row or column. This isn't possible with the communications patterns so far because the communications group `MPI_COMM_WORLD` has been used throughout. Fortunately, MPI provides some support to create more specific communication patterns which are often useful. In this chapter, we discuss how to create subgroups of processes and how to organize communications within the subgroups.

## 18.1 • Subgroups and Communicators

To illustrate the use of different methods of communicating, three examples are presented showing different ways in which a  $2 \times 3$  grid of processes can be organized into rows and columns so that each column or each row can do shared communications. There are two new types that are important in organizing processes for communications: `MPI_Group` and `MPI_Comm`. An `MPI_Group` is simply a collection of some or all of the processes in a given implementation. Up to this point, every MPI function has used the `MPI_Group` corresponding to the entire list of available processes, but subgroups can be created. On top of a given `MPI_Group` there can be different communication patterns, and that is determined by the type `MPI_Comm`. So far `MPI_COMM_WORLD` is the only `MPI_Comm` that's been used, and this section covers how to create new communicators.

There are two components to setting up group communications for a subpopulation of the processes. First, the subpopulation must be identified, and second, a communicator must be created to work within that group. Groups need not be mutually exclusive, they can be overlapping or not, and they need not span all the available processes. This way, communications can flow in different directions among the same processes depending on the needs of the algorithm.

In the following series of examples, different ways of creating group communicators that will enable group communications among a subpopulation of the processes will be shown. In all cases, it is assumed that six processes will be organized into a  $2 \times 3$  array of processes, and group communications will be set up so that either rows or columns can have their communications be isolated from the rest of the array.

The first example is fairly straightforward where the different rows and groups are created explicitly.

### Example 18.1.

```

1 #include <stdio.h>
2 #include <mpi.h>
3
4 /*
5   int main( int argc , char* argv[] )
6
7   Demonstration of creating subgroup communications. This program
8       creates a 2 x 3 array of processes and then creates a group
9       communication, one for each row, and one for each column.
10      Does one broadcast to show it works.
11
12  Inputs: none, but requires at least 6 processes to run properly.
13
14  Outputs: For each subgroup, show the global and local rank of the
15          process within its corresponding groups
16
17 */
18
19 int main(int argc , char* argv [])
20 {
21     // Initialize the MPI interface .
22     MPI_Init(&argc , &argv );
23
24     // Get the rank and size in the world communicator
25     int world_rank , world_size;
26     MPI_Comm_rank(MPI_COMM_WORLD, &world_rank );
27     MPI_Comm_size(MPI_COMM_WORLD, &world_size );
28
29     // Set the dimensions of the process grid
30     int dim[2] = {2,3};
31
32     // Create the world_group so that we can choose from a subgroup of it
33     MPI_Group world_group ;
34     MPI_Comm_group(MPI_COMM_WORLD, &world_group );
35
36     // Organize the processes into an array like this :
37     //
38     //      3 4 5
39     //      0 1 2
40     //
41     // where these are the ranks of the processes
42
43     // Create two row communicators and then do a sample broadcast
44     int row_ranks[2][3] = {{0,1,2},{3,4,5}};
45     MPI_Group row_group[2];
46     MPI_Comm row_comm[2];
47
48     int row;
49     int value;
50     for (row=0; row<2; ++row) {
51         value = world_rank ;
52
53         // For each row, set the members of the row as a new group
54         MPI_Group_incl(world_group , 3, row_ranks[row] , &row_group[row]);
55
56         // Build a communicator for the new group
57         MPI_Comm_create(MPI_COMM_WORLD, row_group[row] , &row_comm[row]);
58

```

```

59 // Not all processes are members of the group, nonmembers will
60 // report row_comm[row] == MPI_COMM_NULL, so they should skip the
61 // broadcast
62 if (row_comm[row] != MPI_COMM_NULL) {
63     int row_rank, row_size;
64
65     // Retrieve the rank and size of this subgroup communicator
66     MPI_Comm_rank(row_comm[row], &row_rank);
67     MPI_Comm_size(row_comm[row], &row_size);
68
69     // Do a broadcast of the subgroup rank 0 process value of its
70     // world rank and then print the results.
71     MPI_Bcast(&value, 1, MPI_INT, 0, row_comm[row]);
72     printf("World rank: %d, row: %d, row_rank: %d/%d, value: %d\n",
73             world_rank, row, row_rank, row_size, value);
74 }
75
76 // Create three column communicators and then do a sample broadcast
77 int col_ranks[3][2] = {{0,3},{1,4},{2,5}};
78 MPI_Group col_group[3];
79 MPI_Comm col_comm[3];
80
81 int col;
82 for (col=0; col<3; ++col) {
83     value = world_rank;
84
85     // For each column, set the members of the column as a new group
86     // and then build the corresponding communicator
87     MPI_Group_incl(world_group, 2, col_ranks[col], &col_group[col]);
88     MPI_Comm_create(MPI_COMM_WORLD, col_group[col], &col_comm[col]);
89
90     // Only do the broadcast among processes that are group members
91     if (col_comm[col] != MPI_COMM_NULL) {
92         int col_rank, col_size;
93
94         // Get the local rank and size and broadcast the local rank 0
95         // value of the world_rank
96         MPI_Comm_rank(col_comm[col], &col_rank);
97         MPI_Comm_size(col_comm[col], &col_size);
98         MPI_Bcast(&value, 1, MPI_INT, 0, col_comm[col]);
99         printf("World rank: %d, col: %d, col_rank: %d/%d, value: %d\n",
100                world_rank, col, col_rank, col_size, value);
101     }
102 }
103
104 // Clean up all the created groups and communicators
105 for (row=0; row<2; ++row)
106     if (row_comm[row] != MPI_COMM_NULL) {
107         MPI_Comm_free(&row_comm[row]);
108         MPI_Group_free(&row_group[row]);
109     }
110
111
112 for (col=0; col<3; ++col)
113     if (col_comm[col] != MPI_COMM_NULL) {
114         MPI_Comm_free(&col_comm[col]);
115         MPI_Group_free(&col_group[col]);
116     }
117
118 MPI_Finalize();
119 return 0;
120 }
```

In order to create a subgroup, there has to be a group in which it is a subset. For this purpose, an `MPI_Group` is created on Line 34 to create a group from the built-in all-inclusive communicator `MPI_COMM_WORLD`. The subgroup is created on Line 54 by sending an array of the process ranks out of the `world_group` that should be included in the new row group. The identification of the processes is sent as an array, enumerated on Line 44. Once the subgroup is created, a communicator designed to work on that group can be created on Line 57.

This example is such that the communicators are only for a subgroup and do not address all the processes. Thus, if a process with rank that is not in the list tries to use the communicator, MPI will throw an error and kill the program. To avoid such events, one must use a guard to check that the communicator does not have the value `MPI_COMM_NULL` as is done on Line 62.

One can also ask for the size of the group and also ask for a process rank within that group using the same `MPI_Comm_rank` and `MPI_Comm_size` functions as before, but now using the new communicator groups. Similarly, group communications, for example, `MPI_Bcast` as on Line 71, are now done only within that subgroup and not with the whole set of processes.

Finally, once the communicators and groups are no longer needed, the memory that was allocated in the creation of the `MPI_Comm` and `MPI_Group` objects must also be freed as shown on Lines 108, 109. Again, because those groups and communicators do not apply to all processes, a guard checking for `MPI_COMM_NULL` must be used as on Line 107.

The rest of Example 18.1 shows how the columns have group communicators as well. Each process can be a member of one or more subgroups, and communicators can overlap in different ways. The key is to make sure it's clear to which group a group communicator is applied.

Try to predict what Example 18.1 will print out and then compare with this sample output:

```
World rank: 0, row: 0, row_rank: 0/3, value: 0
World rank: 1, row: 0, row_rank: 1/3, value: 0
World rank: 2, row: 0, row_rank: 2/3, value: 0
World rank: 3, row: 1, row_rank: 0/3, value: 3
World rank: 4, row: 1, row_rank: 1/3, value: 3
World rank: 5, row: 1, row_rank: 2/3, value: 3
World rank: 0, col: 0, col_rank: 0/2, value: 0
World rank: 3, col: 0, col_rank: 1/2, value: 0
World rank: 1, col: 1, col_rank: 0/2, value: 1
World rank: 4, col: 1, col_rank: 1/2, value: 1
World rank: 5, col: 2, col_rank: 1/2, value: 2
World rank: 2, col: 2, col_rank: 0/2, value: 2
```

## 18.2 • Communicators Using Split

Example 18.1 built communication groups by building them one at a time for each of the rows and columns. The method works but would be very cumbersome should there be a large number of processes. Thus, there are some streamlined methods for some common communication patterns that arise in numerical methods. One such method is by the function `MPI_Comm_split`. The communications split method takes an existing

communication and breaks it into subgroups each with the same *color*, or integer value. For the  $2 \times 3$  example, it will color the first row with the color value 0 and the second row with the color value 1, as illustrated in Figure 18.1.



**Figure 18.1.** Illustration of the color scheme used for the communicators generated with the `MPI_Comm_split` function in Example 18.2. (a) Color scheme used for the row communicator; (b) color scheme used for the column communicator.

The function declaration is

```
int MPI_Comm_split( MPI_Comm orig_comm,
                    int      color,
                    int      rank,
                    MPI_Comm* new_comm )
```

The first argument is the communication that will be split into subgroups. Each process will then give itself a color value, which is the second argument of the function. Processes with the same color value will be placed in the same communicator group. Next, the rank within the original communicator is given, and finally, the new communicator is passed back through the last argument.

### Example 18.2.

```

1 #include <stdio.h>
2 #include <mpi.h>
3
4 /*
5  int main(int argc, char* argv[])
6
7  Demonstration of creating subgroup communications .
8  This program creates a 2 x 3 array of processes and then creates a
9  group communication using the split operation . Does one broadcast
10 to show it works .
11
12 Inputs: none , but requires at least 6 processes to run properly .
13
14 Outputs: For each subgroup , show the global and local rank of the
15 process within its corresponding groups
16
17 */
18
19 int main(int argc, char* argv[])
20 {
21     // Initialize the MPI interface .
22     MPI_Init(&argc , &argv );
23
24     // Get the rank and size in the world communicator
25     int world_rank , world_size ;
26     MPI_Comm_rank(MPI_COMM_WORLD, &world_rank );
27     MPI_Comm_size(MPI_COMM_WORLD, &world_size );
28
29     // Organize the processes into an array like this :
```

```

30  //
31  //      3 4 5
32  //      0 1 2
33  //
34  // where these are the ranks of the processes
35
36  // Set the color for both the row and the column
37  int row_color = world_rank / 3;
38
39  // Create a communicator that puts like colors in the same group
40  MPI_Comm row_comm;
41  MPI_Comm_split(MPI_COMM_WORLD, row_color, world_rank, &row_comm);
42
43  // Retrieve the rank and size for this split communicator
44  int row_rank, row_size;
45  MPI_Comm_rank(row_comm, &row_rank);
46  MPI_Comm_size(row_comm, &row_size);
47
48  int value = world_rank;
49
50  // Do a broadcast of the subgroup rank 0 process value of its
51  // world rank and then print the results.
52  MPI_Bcast(&value, 1, MPI_INT, 0, row_comm);
53  printf("World rank: %d, row: %d, row_rank: %d/%d, value: %d\n",
54  world_rank, world_rank / 3, row_rank, row_size, value);
55
56  // Set the color for both the row and the column
57  int col_color = world_rank % 3;
58
59  // Create a communicator that puts like colors in the same group
60  MPI_Comm col_comm;
61  MPI_Comm_split(MPI_COMM_WORLD, col_color, world_rank, &col_comm);
62
63  // Retrieve the rank and size for this split communicator
64  int col_rank, col_size;
65  MPI_Comm_rank(col_comm, &col_rank);
66  MPI_Comm_size(col_comm, &col_size);
67
68  value = world_rank;
69
70  // Do a broadcast of the subgroup rank 0 process value of its
71  // world rank and then print the results.
72  MPI_Bcast(&value, 1, MPI_INT, 0, col_comm);
73  printf("World rank: %d, col: %d, col_rank: %d/%d, value: %d\n",
74  world_rank, world_rank % 3, col_rank, col_size, value);
75
76  MPI_Comm_free(&row_comm);
77  MPI_Comm_free(&col_comm);
78
79  MPI_Finalize();
80  return 0;
81 }

```

The output of Example 18.2 is the same as for Example 18.1. One key difference between Example 18.1 and Example 18.2 is that in Example 18.1, the groups have to be individually, and explicitly, defined. Using `split`, all the different row communicators are lumped into a single `MPI_Comm` object created on Line 41. Consequently, only one `MPI_Bcast` on Line 52 is required to broadcast to both rows. Similarly, only one `MPI_Bcast` is used for the column communicator. It makes sense that the type of operations on one row are likely to be repeated in the rest of the rows, so this can be very

helpful, particularly when the number of processes gets much larger than the examples presented here.

### 18.3 • Grid Communicators

Another convenient way to build communicators and to organize point-to-point communications is by using a grid communicator. Grid communicators organize processes into grids of any number of dimensions. Each dimension can be made into a loop by designating it as periodic so that when the rank of a neighboring process is requested it will loop around rather than give a nonexistent process number. Group communications can also be set up so that they work within one or more dimensions. An example of creating a grid communicator is shown in Example 18.3.

#### Example 18.3.

```
1 #include <stdio.h>
2 #include <mpi.h>
3
4 /*
5   int main(int argc, char* argv[])
6
7   Demonstration of creating subgroup communications. This program
8   creates a 2 x 3 array of processes and then creates a group
9   communication using the cartesian grid. Does one broadcast to show it
10  works.
11
12 Inputs: none, but requires at least 6 processes to run properly.
13
14 Outputs: For each subgroup, show the global and local rank of the
15  process within its corresponding groups
16
17 */
18
19 int main(int argc, char* argv[])
20 {
21   // Initialize the MPI interface.
22   MPI_Init(&argc, &argv);
23
24   // Get the rank and size in the world communicator
25   int world_rank, world_size;
26   MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
27   MPI_Comm_size(MPI_COMM_WORLD, &world_size);
28
29   // Organize the processes into an array like this:
30   //
31   //      3 4 5
32   //      0 1 2
33   //
34   // where these are the ranks of the processes
35
36   // Set the dimensions of the Cartesian grid
37   int dims[2] = {2,3};
38   // Set periodicity for each dimension, 1=yes, 0=no
39   int periodic[2] = {0,1};
40   // Is it OK to reorder rank? 1=yes, 0=no
41   int reorder = 0;
42
43   // Create a communicator based on a Cartesian grid
44   MPI_Comm grid_comm;
45   MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periodic, reorder,
```

```

46                                         &grid_comm);
47
48 // Retrieve the coordinates for this process
49 int grid_coords[2];
50 MPI_Cart_coords(grid_comm, world_rank, 2, grid_coords);
51
52 // Retrieve the ranks for neighbors in the grid
53 int rank_left, rank_right, rank_up, rank_down;
54 MPI_Cart_shift(grid_comm, 0, 1, &rank_down, &rank_up);
55 MPI_Cart_shift(grid_comm, 1, 1, &rank_left, &rank_right);
56
57 printf("World rank: %d, Grid coord.: (%d,%d), ", world_rank,
58                               grid_coords[0], grid_coords[1]);
59
60 printf("left:%d, right:%d, down:%d, up:%d\n", rank_left, rank_right,
61                               rank_down, rank_up);
62
63 // Create a communicator to link the rows
64 int dims_together[2] = {0,1};
65 MPI_Comm row_comm;
66 MPI_Cart_sub(grid_comm, dims_together, &row_comm);
67
68 // Test row communicator by doing a broadcast
69 int value = world_rank;
70 MPI_Bcast(&value, 1, MPI_INT, 0, row_comm);
71 printf("Row test: Coordinate: (%d,%d), value = %d\n",
72                               grid_coords[0],
73                               grid_coords[1], value);
74
75 // Create a communicator to link the columns
76 dims_together[0] = 1; dims_together[1] = 0;
77 MPI_Comm col_comm;
78 MPI_Cart_sub(grid_comm, dims_together, &col_comm);
79
80 // Test row communicator by doing a broadcast
81 value = world_rank;
82 MPI_Bcast(&value, 1, MPI_INT, 0, col_comm);
83 printf("Col test: Coordinate: (%d,%d), value = %d\n",
84                               grid_coords[0],
85                               grid_coords[1], value);
86
87 MPI_Comm_free(&grid_comm);
88 MPI_Comm_free(&row_comm);
89 MPI_Comm_free(&col_comm);
90
91 MPI_Finalize();
92 return 0;
93 }
```

The output of Example 18.3 is given here, where it's been sorted so it's easier to see the results of the example.

```

World rank: 0, Grid coord.: (0,0), left:2, right:1, down:-2, up:3
World rank: 1, Grid coord.: (0,1), left:0, right:2, down:-2, up:4
World rank: 2, Grid coord.: (0,2), left:1, right:0, down:-2, up:5
World rank: 3, Grid coord.: (1,0), left:5, right:4, down:0, up:-2
World rank: 4, Grid coord.: (1,1), left:3, right:5, down:1, up:-2
World rank: 5, Grid coord.: (1,2), left:4, right:3, down:2, up:-2
Row test: Coordinate: (0,0), value = 0
Row test: Coordinate: (0,1), value = 0
Row test: Coordinate: (0,2), value = 0
```

```

Row test: Coordinate: (1,0), value = 3
Row test: Coordinate: (1,1), value = 3
Row test: Coordinate: (1,2), value = 3
Col test: Coordinate: (0,0), value = 0
Col test: Coordinate: (0,1), value = 1
Col test: Coordinate: (0,2), value = 2
Col test: Coordinate: (1,0), value = 0
Col test: Coordinate: (1,1), value = 1
Col test: Coordinate: (1,2), value = 2

```

In Example 18.3, the Cartesian grid coordinates are created on top of the default `MPI_COMM_WORLD` on Line 45. The declaration for the function is

```

int MPI_Cart_create( MPI_Comm orig_comm,
                     int      num_dims,
                     int*    dims,
                     int*    isPeriodic,
                     int      mayReorder,
                     MPI_Comm* grid_comm )

```

The first argument is the communicator upon which the grid will be overlaid, for example, `MPI_COMM_WORLD`. Next the number of dimensions of the grid is specified, followed by the number of elements within each dimension. In Example 18.3, the number of dimensions is 2, and the grid dimensions are  $2 \times 3$  as specified on Line 37. The geometry of the grid can also be made to be periodic. A periodic grid will be connected in such a way that incrementing along the grid beyond the bounds will cause it to wrap around. In this example, the second dimension is designated as periodic. To see that this is the case, look at the output and note that beginning from grid coordinate  $(0, 0)$ , going to the left, which in this layout means decrementing the column index, would give grid coordinate  $(0, -1)$ . Of course there is no grid coordinate  $(0, -1)$ , but the second dimension was designated as periodic, so instead the coordinate to the left is  $(0, 2)$ . This is confirmed by seeing that Rank 0, corresponding to grid coordinate  $(0, 0)$ , reports that the rank to the left is Rank 2, and the grid coordinate with Rank 2 is in fact  $(0, 2)$ . By contrast, the first coordinate was designated as not periodic, so grid coordinate  $(0, 0)$  reports that the rank of the coordinate going down is  $-2$ , which does not correspond to any process. The next argument in `MPI_Cart_create` indicates whether it's permissible to change the original rank of some processes in order to optimize the connections relative to the physical connectivity of the hardware. Setting this argument to true says that it's OK to reorder. That means that the original world rank may get changed. If a code is being set up exclusively on the Cartesian grid right at the beginning, then it makes perfect sense to permit the reordering, but if this is for a subproblem where the world rank is used elsewhere, then this should be set to false. The resulting grid communicator is returned through the last argument.

Once the Cartesian grid communicator is created, there are a number of useful operations that are possible. First, similar to obtaining the world rank for the current process, the grid coordinates of the current process can be obtained by calling `MPI_Cart_coords` as on Line 50. The ranks of neighbors within the grid are obtained by calling `MPI_Cart_shift` as on Line 54, which is declared as

```
int MPI_Cart_shift( MPI_Comm  grid_comm,
                    int        which_dim,
                    int        increment,
                    int*       lower_rank,
                    int*       upper_rank )
```

The first argument is the grid communicator set up using `MPI_Cart_create`. The second argument indicates along which dimension the increment should be taken, and the third argument is the number of increments to move. The second argument must be in the range  $0 \leq \text{which\_dim} < \text{num\_dims}$ , where `num_dims` is the value entered in the `MPI_Cart_create` function. The increment is not limited to  $\pm 1$ , but that is the most commonly used value. The function returns the ranks of the processes corresponding to the increment in both the positive and negative directions, respectively. Thus, when using grid communicators to handle point-to-point communications between neighboring grids which must share boundary data, the Cartesian grid communicator can assist in identifying which processes are neighboring.

Similar to Example 18.2, row and column communicators can also be created using the `MPI_Cart_sub` function, which has the declaration

```
int MPI_Cart_sub( MPI_Comm  grid_comm,
                  int        dims_included,
                  MPI_Comm* new_comm      )
```

On Line 66, a communicator collecting the rows is created. The first argument is the grid communicator, and the last argument is the resulting new communicator. The second argument is an array that indicates which dimensions should be lumped together. Thus, to create a row communicator, grid processes  $(0, 0)$ ,  $(0, 1)$ , and  $(0, 2)$  are lumped together; in other words, include all entries with a fixed first coordinate and any second coordinate. This is indicated in the `MPI_Cart_sub` function by setting the second argument to be the array  $\{0, 1\}$ , meaning that grid components with varying second index should be part of the same groups, but different first indices would be for different groups. This kind of flexibility means that in higher dimensions it's possible to create groups of varying dimensions such as a one-dimensional group by setting `dims_included = {1, 0, 0}` or two-dimensional groups such as `dims_included = {0, 1, 1}`. The row and column communicators are created on Lines 66 and 77 respectively. Checking the results of the following `MPI_Bcast` in the output shows that the Rank 0 process of each row or column is broadcasting its world rank to the corresponding rows and columns as expected, the same as the previous two examples.

---

## Exercises

- 18.1. Write a program to use `MPI_Comm_split` to create two communication groups on a  $4 \times 4$  grid of processes where one group is the red squares and the other group is the black squares of a red/black ordering of the grid. Demonstrate success by using `MPI_Bcast` to send the character “R” or “B” to the corresponding group.
- 18.2. Modify the previous exercise to also create a  $4 \times 4$  Cartesian grid communicator. Use this communicator to send the color character to the neighbor to the right in a loop so that as a result, the checkerboard square swaps red for black everywhere.

## Chapter 19

# Measuring Efficiency and Checkpointing

Running codes on a cluster presents unique challenges compared to the other types of parallelism presented in this book. Coordinating multiple distinct computers to talk to each other in synchrony while keeping them busy on the underlying computations without losing time somewhere is simply not possible, so the goal is to get as close as possible. To get there, some sort of measure is needed for how close an algorithm gets to the perfect efficiency of using  $N$  processes to solve a task  $N$  times faster.

Another aspect of computing on a large cluster is preparing for disruptions. Consider that a node may have roughly a mean time between failure of approximately 400,000 hours. This seems like a long time, but the size of a cluster must also be factored in. For the Sunway TaihuLight with its 40,960 nodes, the mean time between failure becomes  $400,000/40,960 \approx 10$  hours. This means that roughly every 10 hours a node fails and has to be repaired or replaced. If using a fraction of the cluster, maybe you are lucky and are using the cores not affected, or maybe you're not so lucky. In either case, high performance codes need to prepare for disruptions that are beyond the control of the programmer, and one way this is done is through checkpointing.

In this chapter, we will address both of these cluster-specific issues beginning with how to measure parallel efficiency and also how to safeguard computations against unexpected hardware failures.

### 19.1 • Efficiency Measures

Ultimately, the purpose of using a distributed architecture is to throw more computational power at an expensive calculation in order to speed it up. Ideally, if using twice as many cores, then the job should be done in half the time. Of course, that's not always possible. There can be many reasons for losing efficiency in a parallel computing environment. For example, there can be delays due to high volume of interprocess communications, there can be latency issues where processes are remaining idle while waiting for a synchronization point with other processes, there can be load balancing issues where one process ends up doing more work and hence delaying the others, and there can be additional limitations imposed by mesh sizes, cache sizes, and memory sizes. These various issues may be significant at one size of computation but not as significant as the problem grows, and vice versa.

In the end, there must be a way of measuring efficiency when using a parallel architecture. There are two fundamental measures commonly used that test algorithm efficiency: strong scaling and weak scaling. Strong scaling measures how much time is required to complete a fixed-size problem compared to the number of cores used. A perfectly efficient algorithm is strongly scalable if a problem that takes time  $T$  for one core requires time  $T/N$  for  $N$  cores. Weak scaling is a measure of how the time to completion changes when the size of the problem grows with the number of cores. In this case, suppose a problem has size  $M$ , where, for example,  $M$  could be the number of grid points when solving a PDE, or the number of entries in a matrix to be inverted. A perfectly efficient algorithm is weakly scalable if when the problem size increases by a factor, and the number of cores also increases by that factor, then the time to compute the solution remains constant. For example, if the total number of grid points in a solver for a PDE is doubled, and the number of cores used to do the computation is doubled, does the time  $T$  remain the same?

It's a fairly simple task to test for strong scaling. Given a code that solves a problem, run it once with, say, 10 cores, and then run it again with 20 cores, and see if the elapsed time is cut in half. To test this more systematically, collect data on the time elapsed for a fixed problem size solved on various numbers of cores. Let  $T_N$  be the time to completion using  $N$  cores. The strong scaling efficiency is then measured by

$$\text{strong efficiency} = \frac{T_1}{NT_N}.$$

To test for weak scaling, it requires the problem to be able to change in size. For example, if solving a PDE problem, the measure of problem size could be the number of grid points used in the problem. Let  $T_1$  be the time to solve a problem of size  $M$  on a single core, and let  $T_N$  be the time to solve a problem of size  $NM$  using  $N$  cores; then the weak scaling efficiency is measured by

$$\text{weak efficiency} = \frac{T_1}{T_N}.$$

For example, a PDE is solved on a three-dimensional  $M \times M \times M$  grid, hence  $M^3$  grid points, and the time required to solve the problem for a single process is  $T_1$ . Now double the number of grid points in each dimension so that the grid has  $8M^3$  grid points and solve that problem on eight cores and let  $T_8$  be the time to solution. The weak efficiency would then be  $T_1/T_8$ .

In order to take these measurements, an accurate measurement of the compute time is required. MPI provides functions for this purpose: `MPI_Wtime` and `MPI_Wtick`. To measure the efficiency of Example 17.2, insert these lines at Line 36:

```
36  double precision = MPI_Wtick();
37  double starttime = MPI_Wtime();
```

then insert these lines at Line 222:

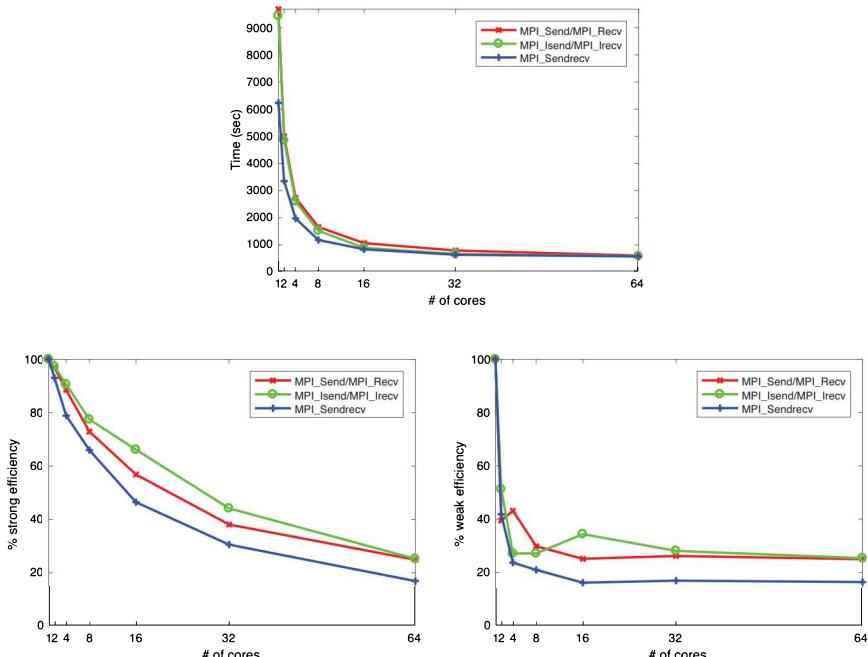
```
223  double time_elapsed = MPI_Wtime() - starttime;
224  printf("Execution time = %le seconds, with precision %e seconds\n",
225          time_elapsed, precision);
```

When timing programs while using MPI be sure to use `MPI_Wtime` and *not* the timing methods discussed in Section 6.3. Those methods are appropriate for serial codes, but for MPI different components of the program may be run on entirely distinct hardware, hence those other functions will likely produce unreliable results.

Figure 19.1 shows the strong and weak scaling results using Example 17.2 with three different methods of communication:

- blocking MPI\_Send/MPI\_Recv,
- nonblocking MPI\_Isend/MPI\_Irecv,
- blocking MPI\_Sendrecv.

Note that using MPI\_Sendrecv for communications proved to be the fastest but also reports the lowest both strong and weak efficiency. Based on that observation it is expected that MPI\_Sendrecv is the best choice for small problems, but when the problem gets bigger, using the nonblocking communications may prove to be advantageous.



**Figure 19.1.** For each plot, three different methods for communication are used in Example 10.5: (a) MPI\_Send/MPI\_Recv, (b) MPI\_Isend/MPI\_Irecv, (c) MPI\_Sendrecv. Top: Raw time measurements for  $N = 6400$  grid points and 1, 2, 4, 8, 16, 32, and 64 cores. Bottom left: Strong efficiency for  $N = 6400$ . Bottom right: Weak efficiency for  $N = 100, 200, 400, 800, 1600, 3200$ , and 6400 grid points and 1, 2, 4, 8, 16, 32, and 64 cores. Time step size used for weak scaling was constant for all cases so that the same number of iterations were calculated.

## 19.2 • Checkpointing

Inevitably, computer codes fail at some point due to programmer error, user error, or hardware error. It could be as simple as an off-by-one bug that terminates the program or as unexpected as a power outage or disk failure. On a small computer, running small jobs, there's no great loss—the problem is fixed and the program is restarted from the beginning.

When doing high performance computing, the size of the resources makes mistakes more expensive. A single compute-core-hour may cost only approximately \$0.05.

That may sound cheap, but use 512 cores for 7 days and the bill would be just over \$4,300 if the job runs to termination. That is still a tiny job compared to the large high performance systems that are presently in use. **Translate that cost to the Sunway TaihuLight, and that same error costs almost \$90,000,000!**

The dollar figures are meant to impress upon the reader the importance of taking greater care in conserving high performance computing resources. It goes without saying that algorithms should be thoroughly tested on smaller systems and shorter times before making a large request for resources. Even the least expensive laptops have at least two to four cores, so testing algorithms can be done quite easily on just about any machine. This much should be obvious. A less obvious precaution that experienced programmers use is the process known as checkpointing.

Checkpointing is where a program periodically stores enough information about its state that it could restart from where it left off by reading a checkpoint file. The purpose for this technique is to ensure that if a computation is interrupted or terminated before completion, it could be restarted from the last checkpoint so that the work up to that point is not lost. How this is accomplished is far too application dependent to be detailed here, but the basic principle is that it is not sufficient to simply dump output to a file without also storing any other data that may be relevant to continuing. In addition to storing the data necessary to resume, the program must also have a capability to read checkpoint data and then resume the computation from where the last checkpoint left off. Of course this restarting ability must also be tested before it's needed to ensure that all the data needed to resume is recoverable. Checkpointing is an important part of high performance computing and must not be disregarded out of inconvenience; the stakes are too high!

As an example of checkpointing, consider the one-dimensional heat equation in Example 17.2. It is customary to store intermediate time points as part of the output. This means using `MPI_Gather` to assemble the data for output and to save the output to a file. The file itself should have a temporal designation. In addition to the values of the function  $u$ , the time step size, the current time step, the spatial discretization, and the terminal time must also be stored. The extra data can be stored in the same file as the  $u$  values, or it could be stored in a separate parameters file that only gets stored once. In addition, the input parameter file should be such that it can use a checkpoint file as a means of retrieving the necessary parameters to run the problem over again.

# Chapter 20

# MPI Libraries

The libraries covered in Chapter 7 can also be used within a single process on a cluster, so those tools are all available. However, serial versions of libraries are not able to handle problems if they are too big for a single computer to hold or if multiple processes are needed to speed a computation. That is where specialized libraries for doing tasks on clusters come into play. In this chapter, we'll look at two such libraries: ScaLAPACK, which is the cluster implementation of LAPACK, and FFTW, which has a distributed version for computing Fourier transforms.

## 20.1 • ScaLAPACK

The same group that developed the LAPACK library of linear algebra routines has also developed a distributed architecture version, called ScaLAPACK. Documentation can be found at

<http://www.netlib.org/scalapack/slug/>

While not all LAPACK functions have been recast in a parallel framework, most have. For the most part, the naming scheme devised for the LAPACK library has been preserved except for ScaLAPACK, where the function names are preceded by the letter “p.” In Chapter 7, we looked at a sample code for solving a linear system using the LAPACK driver function `dgesv_`. Example 20.1 is the parallel analogue to Example 7.2 using ScaLAPACK with `pdgesv_` replacing `dgesv_`.

There are four basic steps to calling a ScaLAPACK routine:

1. Initialize the process grid.
2. Distribute the matrix onto the process grid.
3. Call the ScaLAPACK routine.
4. Release the process grid.

These steps are discussed in order.

### 20.1.1 • Setting Up the Process Grid

For ScaLAPACK, the process grid is a two-dimensional array of the available processes. Hence, the number of entries in the process grid should correspond to the number of cores requested to handle the task. In the example below, a  $2 \times 3$  process grid is requested, so it will require a minimum request of 6 cores when running with MPI. Setting up the process grid initializes the MPI system (i.e., calls `MPI_Init`), and then sets up a communication protocol so that the processes are indexed as in a matrix rather than linearly as seen previously (e.g., Ranks 0–5). When setting up the process grid, distribute the matrices to be operated on as evenly as possible across the process grid.

The process grid is set up by calling

```
sl_init_(int* context, int* pg_rows, int* pg_columns);
```

Upon return, the scalar `context` variable will contain the MPI context that identifies the process grid that was created. The second two arguments are input arguments for the number of rows and columns the process grid will contain. Roughly speaking, if given  $N$  cores, then set up the process grid to have  $r = \lfloor \sqrt{N} \rfloor$  rows and  $c = \lfloor N/r \rfloor$  columns. Multiple process grids can be set up at the same time and then use different contexts for different problems.

Next, a process can determine its location within the process grid by calling

```
blacs_gridinfo_(int* context, int* pg_rows, int* pg_columns,
                int* this_row, int* this_col);
```

The first argument is an input argument giving the integer context identifier obtained from `sl_init_`. The remaining arguments are output arguments, where `pg_rows` and `pg_columns` are the number of process rows and columns on the specified grid, and `this_row`, `this_col` are the row and column of this process within the process grid. The arguments `this_row` and `this_col` play a role similar to that of `rank` in more general MPI programs, while `pg_rows` and `pg_columns` corresponds to the `size` variable. This information will be used to distribute an initial matrix contained in one process to distribute it to the remaining parts of the process grid, and then to reassemble the solution afterward.

### 20.1.2 • Distributing the Matrix

Once the process grid is set up, the pieces of the matrix must be put in block-cyclic form onto the process grid. Before discussing the details of the distribution, we introduce the block-cyclic format. To begin, consider a vector of numbers of length 9 that is to be cut into blocks of length 2 and distributed among three processes. The initial vector is given here:

$$[1, 2, 3, 4, 5, 6, 7, 8, 9].$$

Break it into blocks of length 2 to get

$$[1, 2], [3, 4], [5, 6], [7, 8], [9],$$

where the last block may or may not have a full length. These blocks are then distributed among the three processes like dealing cards from a deck. The first process receives the first block, the second process receives the second block, the third process receives

the third block, and then it begins again with the first process so that the first process receives the fourth block, and so on. After following this procedure, the data is divided into three pieces according to block-cyclic form:

$$\{[1, 2], [7, 8]\}, \{[3, 4], [9]\}, \{[5, 6]\}.$$

Finally, the actual block lengths are understood, and in storage the data will be contiguous, so the three processes have the following data:

$$\{1, 2, 7, 8\}, \{3, 4, 9\}, \{5, 6\}.$$

To distribute a matrix, the same steps are followed, but independently for the rows and columns. For example, suppose a  $9 \times 9$  matrix is to be distributed onto a process grid of dimensions  $2 \times 3$  with block size  $2 \times 2$ . The original matrix is depicted here:

11	12	13	14	15	16	17	18	19
21	22	23	24	25	26	27	28	29
31	32	33	34	35	36	37	38	39
41	42	43	44	45	46	47	48	49
51	52	53	54	55	56	57	58	59
61	62	63	64	65	66	67	68	69
71	72	73	74	75	76	77	78	79
81	82	83	84	85	86	87	88	89
91	92	93	94	95	96	97	98	99

Following the vector example, the matrix is broken into blocks of size  $2 \times 2$ :

11	12	13	14	15	16	17	18	19
21	22	23	24	25	26	27	28	29
31	32	33	34	35	36	37	38	39
41	42	43	44	45	46	47	48	49
51	52	53	54	55	56	57	58	59
61	62	63	64	65	66	67	68	69
71	72	73	74	75	76	77	78	79
81	82	83	84	85	86	87	88	89
91	92	93	94	95	96	97	98	99

The blocks are then distributed among the three grid columns in the same way the single vector was done where the double lines separate the grid columns:

11	12	17	18	13	14	19	15	16
21	22	27	28	23	24	29	25	26
31	32	37	38	33	34	39	35	36
41	42	47	48	43	44	49	45	46
51	52	57	58	53	54	59	55	56
61	62	67	68	63	64	69	65	66
71	72	77	78	73	74	79	75	76
81	82	87	88	83	84	89	85	86
91	92	97	98	93	94	99	95	96

Finally, the blocks are distributed among the rows as well to get

11	12	17	18	13	14	19	15	16
21	22	27	28	23	24	29	25	26
51	52	57	58	53	54	59	55	56
61	62	67	68	63	64	69	65	66
91	92	97	98	93	94	99	95	96
31	32	37	38	33	34	39	35	36
41	42	47	48	43	44	49	45	46
71	72	77	78	73	74	79	75	76
81	82	87	88	83	84	89	85	86

where again the double lines indicate breaks between process grid cells.

There are occasions where one must map between global matrix coordinates and the local coordinate information in the process grids and indexing within the process grids. To go from the global to local coordinates is straightforward. If *i* is the global index, *b* is the block size, and *g* is the number of process grids in that direction (row or column), then the process number, *p*, that contains element *i*, and the local index *i\_loc* within that process are given by

```
p = (i/b)%g;
i_loc = i%b + b*(i/(b*g));
```

It is common that a given system to be solved may initially be on one process, but the solution is to be computed on a parallel architecture. Thus, the matrix must be distributed to the processes within the process grid. Point to point communications available on the grid can be used to do this. The standard blocking send and receive are given by

<pre>dgesd2d_(     int*      context,     int*      m,     int*      n,     double*   Asrc,     int*      nrowssrc,     int*      prowdest,     int*      pcoldest )</pre>	<pre>dgerv2d_(     int*      context,     int*      m,     int*      n,     double*   Adest,     int       nrowsdest,     int*      prowsrc,     int*      pcolsrc )</pre>
--	--

The first argument is the context associated with the process grid. The next two variables are the dimensions of the matrix to be transferred. The fourth argument is a pointer to the array data, which is assumed to have a leading dimension of *nrowssrc* on the source side and *nrowsdest* on the receiving side. The last two arguments are the coordinates to/from which the data is sent/received.

The communications protocol follows a naming scheme similar to other LAPACK functions. In this case, other types of matrices can be communicated by using other function calls. For example, *sgesd2d*/*sgerv2d* are the equivalent functions for single precision floating point. There are also versions for single and double precision complex values as well. The “ge” part of the name refers to a general matrix. Other types of matrices such as upper or lower triangular can also be sent with this method, but there are a few more arguments.

Besides allocating and initializing the data for the matrix on the process grid, there is an additional descriptor that must be created that is used for many of the ScaLAPACK functions to describe a matrix. The descriptor is an integer array that must have length the same size as the full dimension of the matrix. The data in the descriptor is initialized by calling the function

```
void descinit_( int* desc,
                int* m,
                int* n,
                int* mb,
                int* nb,
                int* irsrc,
                int* icsrc,
                int* context,
                int* lld,
                int* info )
```

where the arguments are, in order, a pointer to an array of length **m**, the dimensions of the full  $m \times n$  matrix, the  $mb \times nb$  dimensions of the blocks, the coordinates **irsrc**, **icsrc** of the top left corner of the process grid where the matrix is stored (usually  $(0, 0)$ ), the process grid context, the leading dimension of the local matrix size which is usually the maximum dimensions of the submatrix located in process grid node **irsrc**, **icsrc**, and finally, an output value that indicates any error messages for the call of this function.

### 20.1.3 • Calling the ScaLAPACK Routine

The ScaLAPACK package is intended to be a distributed memory equivalent to the LAPACK routines for serial computers. The discussion here has been limited to solving  $\mathbf{Ax} = \mathbf{b}$ , so, similar to Example 7.2, where the one time driver function **dgesv\_** was used, this parallel implementation uses the comparable function **pdgesv\_**. In general, the attempt is to keep the naming scheme consistent with LAPACK except the additional “p” prefix.

Example 20.1 shows how to use the solver **pdgesv\_**. The function is given by

```
void pdgesv_( int*      m,
               int*      nrhs,
               double*   A,
               int*      ia,
               int*      ja,
               int*      desca,
               int*      ipiv,
               double*   b,
               int*      ib,
               int*      jb,
               int*      descb,
               int*      info )
```

where the arguments are, in order, the dimension of the matrix **A**, the number of columns in the right-hand-side vector **b**, the local data for the matrix stored in block-cyclic order, the row and column of the top left corner of the matrix **A** (typically 1, 1),

the matrix descriptor for **A** created using `descinit_`, a temporary array used for maintaining pivots that has length at least equal to the number of rows in the local matrix storage plus the number of rows in the block size, the data for the right-hand-side data, the row and column index for the top left corner of the right-hand-side data **b** (typically  $(1, 1)$ ), the matrix descriptor for the right-hand side, and the error message data reporter.

#### 20.1.4 ■ Release the Process Grid

Similar to calling `MPI_Finalize()` for a standard MPI code, one must also shut down the process grid when finished with ScaLAPACK. This is accomplished with the function

```
blacs_exit_(int* notdone);
```

where the argument is a statement about whether you are still using the MPI environment. If, which is often the case, the solution of the linear system is but one step in a larger calculation, the process grid can be released without terminating the MPI framework by setting `notdone` to true (or 1). If `notdone` is false (or 0), the call to this function also calls `MPI_Finalize` to shut down MPI as well.

#### Example 20.1.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 // ScaLAPACK functions needed from the library
5 extern void sl_init_();
6 extern void blacs_gridinfo_();
7 extern void blacs_exit_();
8 extern void descinit_();
9 extern void pdlacpy_();
10 extern void pdgesv_();
11 extern double pdlange_();
12 extern double pdlamch_();
13 extern void pdgemm_();
14 extern void pdlaprnt_();
15 extern void dgesd2d_();
16 extern void dgerv2d_();
17
18 // function to initialize the matrix when solving Ax=b
19 void matinit(int context, double** A, int** desca, double** b,
20              int** descb, double** work, int** pivot);
21 // function to compute the dimensions of the local matrix
22 int numroc(int dimen, int bsize, int nprowcol);
23 // function for gathering the block-cyclic scattered solution vector
24 //      onto the 0,0 process.
25 void gathervec(int ictxt, double* bfull, double* blocal, int* descb);
26
27 /*
28  int main(int argc, char* argv[])
29
30  Solve Ax=b on a distributed architecture using ScaLAPACK
31
32  Inputs: none
33
34  Outputs: Prints out the final values.
35
36 */
37
```

```

38 int main(int argc, char* argv[])
39 {
40     int ictxt = 0; // The handle to the system context
41     int nprow = 2; // Number of process rows in the grid
42     int npcol = 3; // Number of process columns in the grid
43     int zero = 0;
44     int one = 1;
45     int six = 6;
46     double done = 1.0;
47
48     int myrow;
49     int mycol;
50
51     int* desca = NULL;
52     int* descb = NULL;
53     int* ipiv = NULL;
54     double* work = NULL;
55     int info; // information storage for the scalapack calls
56
57     double* A = NULL; // local matrix storage
58     double* b = NULL; // local rhs vector storage
59
60     // Initialize the process grid
61     sl_init_(&ictxt, &nprow, &npcol);
62     blacs_gridinfo_(&ictxt, &nprow, &npcol, &myrow, &mycol);
63
64     // Check whether this process is in the grid
65     if (myrow != -1) {
66
67         // Initialize and distribute the matrix, rhs, and create workspace
68         matinit(ictxt, &A, &desca, &b, &descb, &work, &ipiv);
69
70         // Make a copy for checking the answer
71         double* A0 = (double*)malloc(sizeof(double)
72                                     *numroc(desca[2], desca[4], nprow)
73                                     *numroc(desca[3], desca[5], npcol));
74         double* b0 = (double*)malloc(sizeof(double)
75                                     *numroc(descb[2], descb[4], nprow));
76         pdlacpy_("All", &desca[2], &desca[2], A, &one, &one, desca, A0,
77                                              &one, &one, desca);
78         pdlacpy_("All", &desca[2], &one, b, &one, &one, descb, b0,
79                                              &one, &one, descb);
80
81         // Do the solve
82         pdgesv_(&desca[2], &one, A, &one, &one, desca, ipiv, b,
83                                              &one, &one, descb, &info);
84
85         // Gather the result and display it
86         double* soln = (double*)malloc(sizeof(double)*desca[2]);
87         gathervec(ictxt, soln, b, descb);
88         if (myrow == 0 && mycol == 0) {
89             printf("Solution :\n");
90             for (int i=0; i<desca[2]; ++i)
91                 printf("%g\n", soln[i]);
92         }
93
94         double eps = pdlamch_(&ictxt, "Epsilon");
95         double anorm = pdlange_("I", &desca[2], &desca[2], A,
96                                              &one, &one, desca, work);
97         double bnorm = pdlange_("I", &desca[2], &one, b, &one, &one,
98                                              descb, work);
99         double negone = -1.0;
100        pdgemm_("N", "N", &desca[2], &one, &desca[2], &done, A0, &one, &one,
```

```

101             desca , b , &one , &one , descb , &negone , b0 ,
102                                         &one , &one , descb );
103     double xnorm = pdlange_( "I" , &desca[2] , &one , b0 , &one , &one ,
104                                         &descb , work );
105     double resid = xnorm/( anorm*bnorm*eps*desca[2] );
106     if (myrow == 0 && mycol == 0)
107         printf("Residual = %e\n" , resid );
108
109 // Free up the memory created in matinit and above
110 free(A);
111 free(b);
112 free(desca);
113 free(descb);
114 free(work);
115 free(ipiv);
116 free(A0);
117 free(b0);
118 }
119
120 // Exit BLACS
121 blacs_exit_(&zero);
122
123 return 0;
124 }
125
126 /*
127 int numroc (int dimen, int bsize, int nprowcol)
128
129 This calculates the maximum row or column size required to store
130 a local piece of a block-cyclic distributed matrix for use in
131 scalapack. This function works the same for both rows and
132 columns.
133
134 Inputs:
135 int dimen: Dimension of the full matrix
136
137 int bsize: block size of the row or column
138
139 int nprowcol: number of process rows or columns
140
141 Output:
142 returns the maximum row or column dimension size
143 */
144 int numroc(int dimen, int bsize, int nprowcol)
145 {
146 // how many blocks will there be in total?
147 int numblocks = (dimen+bsize-1)/bsize;
148 // how many rows/cols are in the last block?
149 int lastblock = dimen%bsize == 0 ? bsize : dimen%bsize;
150 // the number of rows/cols as a result of a full array of blocks
151 int ans = ((dimen-1)/(nprowcol*bsize))*bsize;
152 // adjustment for the last partial array of blocks if any
153 switch(numblocks%nprowcol) {
154 case 0: break; // number of blocks divided evenly, no partial row/col
155 case 1: ans += lastblock; break; // last block is in the first process
156 default: ans += bsize; // last block is in a later process,
157                         // so get a whole block
158 }
159 return ans;
160 }
161
162 /*
163 void matinit(int context, double** A, int** desca, double** b,

```

```

164         int** descb, double** work, int** pivot)
165
166 This is an example of taking a matrix stored as a full matrix in process
167 grid 0,0 and breaking it into block-cyclic pieces and distributing to
168 the process grid for use in scalapack. Be sure to free the memory when
169 finished.
170
171 Inputs:
172 int ictxt: the communications context that will be using this matrix
173
174 Outputs:
175 double** A: Allocates and populates the local storage of the
176     block-cyclic stored matrix
177
178 int** desca: Allocates and initializes the A matrix descriptor
179
180 double** b: Allocates and populates the local storage of the rhs vector
181     in block-cyclic form
182
183 int** descb: Allocates and initializes the b rhs vector descriptor
184
185 double** work: Create the temp double space required for
186     the linear solve
187
188 int** piv: Create the temp int space required for the linear solve
189
190 */
191 void matinit(int ictxt, double** A, int** desca, double** b,
192             int** descb, double** work, int** piv)
193 {
    // Build matrix
195 // [ 19 3 1 12 1 16 1 3 11 ] [ ] [ 0 ]
196 // [ -19 3 1 12 1 16 1 3 11 ] [ ] [ 0 ]
197 // [ -19 -3 1 12 1 16 1 3 11 ] [ ] [ 1 ]
198 // [ -19 -3 -1 12 1 16 1 3 11 ] [ ] [ 0 ]
199 // [ -19 -3 -1 -12 1 16 1 3 11 ] [ x ] [ 0 ]
200 // [ -19 -3 -1 -12 -1 16 1 3 11 ] [ ] [ 0 ]
201 // [ -19 -3 -1 -12 -1 -16 1 3 11 ] [ ] [ 0 ]
202 // [ -19 -3 -1 -12 -1 -16 -1 3 11 ] [ ] [ 0 ]
203 // [ -19 -3 -1 -12 -1 -16 -1 -3 11 ] [ ] [ 0 ]
204 //
205 int nprow, npcol; // number of process rows and columns
206 int myrow, mycol; // row and column of this process
207
208 blacs_gridinfo_(&ictxt, &nprow, &npcol, &myrow, &mycol);
209
210 int dlen = 9; // dimension of the square matrix A
211 int blksize = 2; // block size will be 2 for rows and columns
212 // number of rows in the process grid to represent the matrix:
213 //     mxllda = ceil(dlen/blksize)
214 int mxllda = numroc(dlen, blksize, nprow);
215 // number of cols in the process grid to represent the matrix:
216 //     mxllda = ceil(dlen/blksize)
217 int mxlocc = numroc(dlen, blksize, npcol);
218
219 // convenience variables for Fortran calls
220 int zero = 0;
221 int one = 1;
222
223 // error message info
224 int info;
225
226 // work array must have length mxlda

```

```

227 *work = (double*)malloc(sizeof(double)*mxllda);
228 // pivot array must have length mxllda+blksize
229 *piv = (int*)malloc(sizeof(int)*(mxllda+blksize));
230
231 // Create a matrix descriptor for the matrix and the rhs
232 *descA = (int*)malloc(sizeof(int)*dlen);
233 *descB = (int*)malloc(sizeof(int)*dlen);
234 descinit_(*descA, &dlen, &dlen, &blksize, &zero, &zero,
235 &ictxt, &mxllda, &info);
236 descinit_(*descB, &dlen, &one, &blksize, &one, &zero, &zero,
237 &ictxt, &mxllda, &info);
238
239 // Create the local storage space
240 *A = (double*)malloc(sizeof(double)*mxllda*mxlocc);
241 *B = (double*)malloc(sizeof(double)*mxllda);
242
243 // actual matrix creation, the 0,0 process will create the
244 // full matrix, and then distribute the pieces to the others
245 double data[9] = {19,3,1,12,1,16,1,3,11};
246 double rhsdata[9] = {0,0,1,0,0,0,0,0,0};
247 if (myrow == 0 && mycol == 0) {
248
249 // Create the full matrix and rhs in process 0,0
250 double Afull[9][9], bfull[9];
251 for (int i=0; i<9; ++i) {
252     for (int j=0; j<9; ++j)
253         Afull[i][j] = data[j]*(j < i ? -1 : 1);
254     bfull[i] = rhsdata[i];
255 }
256
257 // Create all the blocks that will get distributed
258 // The transpose from row-major to column-major is done here
259 double Ablock[nproc][nproc][mxlocc][mxllda];
260 double bblock[nproc][mxllda];
261 for (int i=0; i<9; ++i) {
262     int pr = (i/blksize)%nproc; // the process row for this row of A
263     // the local offset for this process row
264     int loci = i%blksize + blksize*(i/(blksize*nproc));
265     for (int j=0; j<9; ++j) {
266         int pc = (j/blksize)%nproc; // the process col for this col of A
267         // the local offset for this process col
268         int locj = j%blksize + blksize*(j/(blksize*nproc));
269         // Assign the global matrix to the local pieces
270         Ablock[pr][pc][locj][loci] = Afull[i][j];
271     }
272     bblock[pr][loci] = bfull[i];
273 }
274
275 // Temporary pointer to the local storage part
276 double* Aptr = &(Ablock[0][0][0][0]);
277 for (int pr=0; pr<nproc; ++pr)
278     for (int pc=0; pc<nproc; ++pc)
279         if (pr == 0 && pc == 0) {
280             // if in process 0,0, then just copy the data
281             // rather than Send/Recv
282             for (int i=0; i<mxllda*mxlocc; ++i)
283                 (*A)[i] = Aptr[i];
284             for (int i=0; i<mxllda; ++i)
285                 (*B)[i] = bblock[0][i];
286         } else {
287             // send the local piece of the matrix to the corresponding
288             // member of the process grid
289             dgesd2d_(&ictxt, &mxllda, &mxlocc, &(Ablock[pr][pc][0][0]),

```

```

290                                         &mxllda , &pr , &pc);
291     if (pc == 0)
292         dgesd2d_(&ictxt , &mxllda , &one , bblock[pr] , &mxllda ,
293                               &pr , &pc);
294 }
295 } else if (myrow != -1) {
296     // get the local matrix storage from process 0,0
297     dgerv2d_(&ictxt , &mxllda , &mxloc , *A , &mxllda , &zero , &zero);
298     if (mycol == 0)
299         dgerv2d_(&ictxt , &mxllda , &one , *b , &mxllda , &zero , &zero);
300 }
301 }
302 */
303 void gathervec(int ictxt , double* bfull , double* blocal , int* descb)
304
305 This function uses point to point communication to gather the solution
306 vector stored in block-cyclic fashion, into a contiguous single vector
307 in process grid 0,0.
308
309
310 Inputs:
311     int ictxt: The context id for the process grid the vector is
312             stored on
313
314     double* blocal: The local storage for the solution vector.
315             Is only expected in process grid nodes where mycol == 0
316
317     double* descb: The matrix description label for the blocal
318             vector.
319
320 Outputs:
321     double* bfull: The full size vector to be created in
322             process grid 0,0
323 */
324 void gathervec(int ictxt , double* bfull , double* blocal , int* descb)
325 {
326     // Get the information about the process grid from the context id
327     int nprow , npcol ;
328     int myrow , mycol ;
329     blacs_gridinfo_(&ictxt , &nprow , &npcol , &myrow , &mycol );
330
331     // Constants for Fortran calls
332     int one = 1;
333     int zero = 0;
334
335     if (myrow == 0 && mycol == 0) {
336
337         // Create copies of the local blocks so that they can be reassembled
338         // in the right order
339         double bblock [nprow][descb[8]];
340         for (int i=0; i<descb[8]; ++i)
341             bblock [0][i] = blocal [i];
342
343         // Get the data from the other processes with mycol == 0
344         for (int pr=1; pr<nprow; ++pr) {
345             dgerv2d_(&ictxt , &descb[8], &one , bblock[pr] , &descb[8],
346                               &pr , &zero);
347         }
348
349         // Map the local data into the full data array
350         for (int i=0; i<descb[2]; ++i) {
351             int pr = (i/descb[4])%nprow;
352             int loci = i%descb[4] + descb[4]*(i/(descb[4]*nprow));

```

```

353     bfull[i] = bblock[pr][loc];
354 }
355
356 } else if (myrow != -1 && mycol == 0) {
357 // Send the column data to the 0,0 grid process
358 dgesd2d_(&ictxt, &descb[8], &one, blocal, &descb[8], &zero, &zero);
359 }
360 }
```

### 20.1.5 • Description of Example 20.1

Example 20.1 illustrates the use of ScaLAPACK and the steps involved in performing a linear solve for a dense matrix **A** and a single right-hand-side vector **b**. Below is a description of the steps in the program:

- Lines 5–16: The standard ScaLAPACK package is written in Fortran, while the communications library (BLACS, or basic linear algebra communication system) is written in C (but in a Fortran-compatible way). This means that there is no header file with all the ScaLAPACK functions enumerated for purposes of declaring the functions. Therefore, every ScaLAPACK function to be used must be declared at the top of the file. The keyword `extern` means that the function will be declared elsewhere and is not in this file. Note that the function argument lists are all empty; the linker will check the arguments match when the program is linked to the library.
- Lines 19–20: This function is not a ScaLAPACK function but a function where the matrix **A** and the vector **b** will be defined and distributed across the process grid.
- Line 22: This function is defined to help calculate the maximum dimensions of the local storage for the matrix using the block-cyclic format.
- Line 25: This is a function for gathering the block-cyclic scattered solution vector into a single array on the (0, 0) process.
- Lines 57–58: The arrays **A** and **b** hold only the local submatrix, not the full matrix.
- Lines 61–62: The process grid is set up with specified dimensions, and the information about where the current node is in the process grid is retrieved on Line 61.
- Line 65: There may be more active processes than are used for this process grid. Those processes not involved in this process grid will have `myrow` set to `-1` on Line 61.
- Lines 71–78: Like LAPACK, the solver will alter the matrix and the solution vector will replace the right-hand-side vector **b**. In order to check for the error in the solution, the original matrix and right-hand side must be preserved by copying it.
- Line 82: This is where the solution to the linear system is actually computed. Upon return, the solution is stored in the vector **b**.
- Lines 86–92: The function `gathervc()` is designed to collect the different pieces of the solution vector and store them in the expected order on process (0, 0).

- Lines 94–107: The relative error in the residual is computed and displayed using node (0, 0) in the process grid.
- Lines 110–117: As always, be sure to clean up when finished to avoid memory leaks.
- Line 121: The process grid is shut down here. Because zero is passed as the argument, the function will call `MPI_Finalize` to shut down MPI as well.
- Lines 127–160: This function computes the maximum number of rows or columns of storage necessary to store a piece of the matrix divided according to the block-cyclic distribution.
- Lines 210–217: The matrix dimensions, the block size, and the storage size for the local part of the matrix are determined.
- Lines 227–229: The temporary work arrays required for the solver are created here. Their size depends on the matrix and block size dimensions, so they can't be allocated until the matrix dimensions are known.
- Lines 232–236: The matrix description data is stored in an integer array of length equal to the dimensions of the matrix. The memory is allocated and initialized here.
- Lines 240–241: The local part of the matrix and right-hand side are allocated.
- Lines 245–255: The full matrix is created in the (0,0) node of the process grid. It is not required to do this; each node in the process grid can construct its own part of the full matrix if that is what is desired. In fact, that is optimal if appropriate for the application. However, here it is constructed in one place so that point-to-point communications between process grid nodes can be demonstrated. The linear system constructed in this example is

$$\begin{bmatrix} s & c & a & l & a & p & a & c & k \\ -s & c & a & l & a & p & a & c & k \\ -s & -c & a & l & a & p & a & c & k \\ -s & -c & -a & l & a & p & a & c & k \\ -s & -c & -a & -l & a & p & a & c & k \\ -s & -c & -a & -l & -a & p & a & c & k \\ -s & -c & -a & -l & -a & -p & a & c & k \\ -s & -c & -a & -l & -a & -p & -a & c & k \\ -s & -c & -a & -l & -a & -p & -a & -c & k \end{bmatrix} \mathbf{x} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix},$$

where the letters are replaced with their numerical position in the alphabet, e.g.,  $a = 1$ ,  $c = 3$ , etc.

- Lines 259–273: In this section, the full matrix is broken down into an array of blocks according to the block-cyclic distribution. Given matrix row and column  $(i, j)$ , the corresponding process grid node (`pr,pc`) and local row and column indices (`loci,locj`) are computed and then the full matrix entry is assigned to the corresponding block. The same is done for the right-hand side, which is confined to the first column of the process grid.

- Lines 282–285: The full matrix  $\mathbf{A}$  and the right-hand-side vector  $\mathbf{b}$  are constructed in the  $(0, 0)$  node of the process grid, so the local submatrices of  $\mathbf{A}$  and  $\mathbf{b}$  can simply be copied rather than using point-to-point communication.
- Lines 289–292: The  $(0, 0)$  node sends the local block data to the other nodes in the process grid.
- Lines 297–299: The other nodes wait to receive their part of the matrix and right-hand side from the  $(0, 0)$  node of the process grid.
- Lines 339–359: The reverse process of the scatter on Lines 259–285 is applied to gather the solution vector from the distributed block-cyclic form into a single solution vector.

### 20.1.6 • Compiling and Linking with ScaLAPACK

To compile code that uses ScaLAPACK, both the ScaLAPACK and the LAPACK and BLAS libraries are needed:

```
$ mpicc -c mympiprog.c
$ mpicc -o mympiprog mympiprog.o -lscalapack -llapack -lblas
```

## 20.2 • FFTW

The serial version of FFTW as described in Section 7.2 can certainly be used in a distributed memory architecture by using domain decomposition. Because doing this presents a nice illustration of combining MPI communications with a different form of domain decomposition, it's instructive to look at using a serial one-dimensional FFT function to do higher-dimensional FFTs in parallel using MPI. However, in practice the MPI implementation of FFTs provided by FFTW is much easier. For more information about the FFT and how it is used, see Chapter 37.

### 20.2.1 • Using Serial FFTs in Parallel

A multidimensional FFT is actually a one-dimensional FFT applied first in one dimension, applied again in a second dimension, and so forth until all dimensions have been computed. Furthermore, when the one-dimensional FFT is applied in a particular direction, the multiple FFTs in that direction are completely independent and can be done in parallel without any communications. For simplicity, only the two-dimensional case is described here, but higher dimensions are analogous.

Consider a large two-dimensional  $N \times N$  array for which a two-dimensional FFT is desired. If the array isn't too large to fit within each process, then a simple algorithm can be devised such as

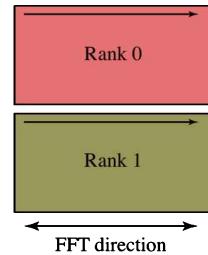
Rank 0	Rank 1
Apply FFT to rows 0 to $N/2 - 1$	Apply FFT to rows $N/2$ to $N - 1$
Use MPI_Allgather	Use MPI_Allgather
Apply FFT to columns 0 to $N/2 - 1$	Apply FFT to columns $N/2$ to $N - 1$
Use MPI_Allgather	Use MPI_Allgather

For larger arrays, where the full array cannot be stored in each process, the algorithm is a little more complicated. For example, the algorithm could look like

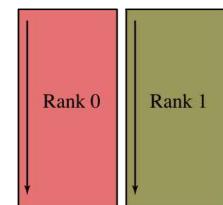
Rank 0	Rank 1
Send rows $N/2$ to $N - 1$ to Rank 1	Receive rows $N/2$ to $N - 1$ from Rank 0
Apply FFT to local rows	Apply FFT to local rows
Transpose local array	Transpose local array
Use MPI_Scatter	Use MPI_Scatter
Apply FFT to local columns	Apply FFT to local columns
Transpose local array	Transpose local array
Use MPI_Scatter	Use MPI_Scatter

These steps are illustrated in more detail here:

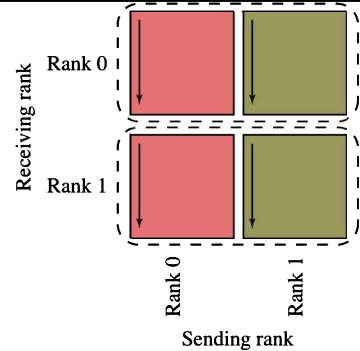
To start, the  $N \times N$  data is arranged so that each rank has an  $N/2 \times N$  part of the data. Rank 0 has the first part, i.e.,  $u[0][0], \dots, u[N/2-1][N-1]$ , and Rank 1 has the second part, i.e.,  $u[N/2][0], \dots, u[N-1][N-1]$ . The data in the columns are contiguous, so the FFT can be applied to each of the  $N/2$  rows in parallel.



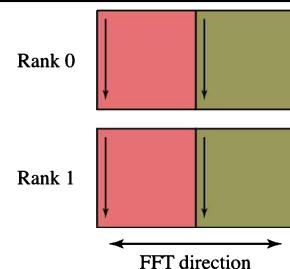
To organize the data for an FFT in the other direction, each rank first transposes its part of the data so that the short dimension is contiguous. This is done to take advantage of the **MPI\_Scatter** command. Without transposing first, the data would be split the long way, which is not what is desired.



The data is distributed using **MPI\_Scatter**. In this case, Rank 0 keeps the first half of its local data and sends the second half to Rank 1. Similarly, Rank 1 sends its first half to Rank 0 and keeps the second half. At this point, Rank 0 has two  $N/2 \times N/2$  arrays, the first is  $u[0][0], \dots, u[N/2-1][N/2-1]$ , and the second is  $u[N/2][0], \dots, u[N-1][N/2-1]$ . Rank 1 has the other two parts.



The two partial arrays cannot be directly appended because of the order of the data, so each rank copies the partial arrays into an  $N/2 \times N$  contiguous array so that the original first index is now contiguous. The FFT can now be applied to each of the rows in parallel. The transpose and scatter operations are repeated a second time to get back to the original arrangement of the data.



See Section 21.2.1 for more details of how the arrangement of data in subdomains impacts the use of `MPI_Gather` and `MPI_Scatter`.

For a three-dimensional array, a similar strategy can be employed but where, alternatively, the serial version of the two-dimensional FFT is used and each process has a subset of contiguous two-dimensional slices of the data. The data is then transposed and scattered as before, always ensuring that the dimensions to which the two-dimensional transform will be applied are contiguous within each process.

### 20.2.2 • Distributed FFTW

The previous strategy is of use for some applications, but for FFTW on a distributed architecture, it is much easier to use the MPI version of the software. The general organization of using plans to execute transforms and the assumption that data is stored in column-major order is largely the same, but there are some differences worth noting. Example 20.2 illustrates how to execute a forward and backward transform on a two-dimensional array of data.

#### Example 20.2.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4 #include <mpi.h>
5 #include <fftw3-mpi.h>
6
7 /*
8   int main(int argc, char* argv[])
9
10  Do a forward and reverse Fourier transform on a 2D array
11  where the array is distributed.
12
13  Inputs:
14    argv[1]: Length of first dimension
15    argv[2]: Length of second dimension
16
17  Outputs: Prints out the final values.
18
19 */
20
21 int main(int argc, char* argv[])
22 {
23   // Initialize MPI
24   MPI_Init(&argc, &argv);
25
26   // Initialize FFTW for MPI
27   fftw_mpi_init();
28
29   // Get dimensions of the domain
30   const ptrdiff_t M = atoi(argv[1]);
31   const ptrdiff_t N = atoi(argv[2]);
32   ptrdiff_t localM, localN;
33
34   // Determine the amount of local memory required
35   ptrdiff_t alloc_local = fftw_mpi_local_size_2d(M, N, MPI_COMM_WORLD,
36                                                 &localM, &localN);
37
38   // Allocate the local memory
39   fftw_complex* datain = fftw_alloc_complex(alloc_local);
40   fftw_complex* dataout = fftw_alloc_complex(alloc_local);

```

```

41 // Set up the transform plans
42 fftw_plan pf, pb;
43 pf = fftw_mpi_plan_dft_2d(M, N, datain, dataout, MPI_COMM_WORLD,
44     FFTW_FORWARD, FFTW_ESTIMATE);
45 pb = fftw_mpi_plan_dft_2d(M, N, dataout, dataout, MPI_COMM_WORLD,
46     FFTW_BACKWARD, FFTW_ESTIMATE);
47
48 #ifndef M_PI
49     const double M_PI = 4.0*atan(1.0);
50 #endif
51
52
53 // Set up the initial data on the local allocation
54 double dx = 2*M_PI/M;
55 double dy = 2*M_PI/N;
56 for (int i=0; i<localM; ++i) {
57     double x = dx*(local0+i);
58     for (int j=0; j<N; ++j) {
59         double y = dy*j;
60         datain[i*N+j][0] = cos(2*x+y);
61         datain[i*N+j][1] = sin(2*x+y);
62     }
63 }
64
65 // Execute the forward transform
66 fftw_execute(pf);
67
68 // Set up rank 0 to receive the full array upon completion
69 double* fulldatain = NULL;
70 double* fulldataout = NULL;
71 int rank;
72 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
73 if (rank == 0) {
74     fulldatain = (double*)malloc(2*M*N*sizeof(double));
75     fulldataout = (double*)malloc(2*M*N*sizeof(double));
76 }
77
78 // Assemble the results into the full array on rank 0 process
79 MPI_Gather(dataout, 2*localM*N, MPI_DOUBLE, fulldataout, 2*localM*N,
80             MPI_DOUBLE, 0, MPI_COMM_WORLD);
81
82 // Print the results
83 if (rank == 0) {
84     for (int i=0; i<M; ++i) {
85         for (int j=0; j<N; ++j) {
86             double real = fulldataout[i*2*N+2*j];
87             double imag = fulldataout[i*2*N+2*j+1];
88             printf("%4.2f%cc%4.2fi\t", real/M/N,
89                               imag >= 0 ? '+' : '-',
90                               fabs(imag/M/N));
91         }
92         printf("\n");
93     }
94 }
95
96 // Execute the backward transform
97 fftw_execute(pb);
98
99 MPI_Gather(dataout, 2*localM*N, MPI_DOUBLE, fulldataout, 2*localM*N,
100            MPI_DOUBLE, 0, MPI_COMM_WORLD);
101
102 fftw_destroy_plan(pf);
103 fftw_destroy_plan(pb);
104 fftw_free(datain);

```

```

104     fftw_free(dataout);
105     if (rank == 0) {
106         free(fulldatain);
107         free(fulldataout);
108     }
109     MPI_Finalize();
110 }
```

For the MPI version of FFTW the library must be initialized to prepare the communications, which is done on Line 27.

The data is organized so that the first subscript is broken into  $P$  subdomains subdivided as equally as possible among the  $P$  processes. It is not necessary for the first dimension to be a multiple of  $P$ , but if not, then the subdomains will not all have the same local length in the first dimension. It doesn't impact computing the transform, but it would alter the way the data is reassembled upon completion. In this example, since the regular MPI\_Gather is used on Lines 79, 98, there is an implicit assumption that all the subdomains are equal in size, i.e.,  $M$  is a multiple of  $P$ . The size of the local domain is determined on Line 35. The last two arguments are where the size of the first dimension, `localM`, and the first index within the global array, `local0`, are set. The local memory is allocated on Lines 39, 40.

Setting up the transform plan is very similar to the serial version with the exception that the plan function has a different name, `fftw_mpi_plan_dft_2d`, and requires the communication group that will be performing the calculation, in this case `MPI_COMM_WORLD`. The rest of the arguments are the same, noting that the domain dimensions given as the first argument are the dimensions of the full array, not the local dimensions. The input and output arrays may be the same, and if so, the input array is overwritten by the output results. The forward plan is set up on Line 44 and the backward plan is set up on Line 46. Execution of the plans is done on Lines 66, 96.

When accessing or setting the values in the local array, the local coordinate system must incorporate the position within the full array. To do that, the first local index will only have `localM` length starting at zero for each subdomain as evidenced by the limits of the loop on Line 56. The global index for the first dimension is the sum of the local index with `local0` that was obtained on Line 35. Line 57 illustrates how to use `local0` to determine the global coordinate.

Finally, the plans and the data allocated by the library must also be freed as on Lines 101–104.

### 20.2.3 • Transforming Real-Valued Data

The distributed FFTW also has specialized transforms for real-valued data. For example, the plans for transforming two-dimensional real data are generated using the functions

```

fftw_mpi_plan_dft_r2c_2d (M, N, r_in, c_out, MPI_COMM_WORLD,
                           FFTW_ESTIMATE);
fftw_mpi_plan_dft_c2r_2d (M, N, c_in, r_out, MPI_COMM_WORLD,
                           FFTW_ESTIMATE);
```

Execution of the plans is done the same as described in the previous section.

The organization of the spectral coefficients that result from executing these plans is the same as described for the serial version in Section 7.2.4, the main difference being that the coefficients are again distributed among the processes by splitting in the first

dimension as described in the previous section. Thus, for two processes and assuming  $M$  is divisible by 2, the data is organized as below:

Rank 0	$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$\cdots$	$a_{0,\frac{N}{2}}$
	$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$\cdots$	$a_{1,\frac{N}{2}}$
	$\vdots$	$\vdots$	$\vdots$		$\vdots$
	$a_{\frac{M}{2}-1,0}$	$a_{\frac{M}{2}-1,1}$	$a_{\frac{M}{2}-1,2}$	$\cdots$	$a_{\frac{M}{2}-1,\frac{N}{2}}$
Rank 1	$a_{\frac{M}{2},0}$	$a_{\frac{M}{2},1}$	$a_{\frac{M}{2},2}$	$\cdots$	$a_{\frac{M}{2},\frac{N}{2}}$
	$a_{1-\frac{M}{2},0}$	$a_{1-\frac{M}{2},1}$	$a_{1-\frac{M}{2},2}$	$\cdots$	$a_{1-\frac{M}{2},\frac{N}{2}}$
	$a_{2-\frac{M}{2},0}$	$a_{2-\frac{M}{2},1}$	$a_{2-\frac{M}{2},2}$	$\cdots$	$a_{2-\frac{M}{2},\frac{N}{2}}$
	$\vdots$	$\vdots$	$\vdots$		$\vdots$
	$a_{-1,0}$	$a_{-1,1}$	$a_{-1,2}$	$\cdots$	$a_{-1,\frac{N}{2}}$

For these transforms, note that the real data must allocate enough space to hold the complex data as temporary storage. Thus, the memory allocation for each process is handled as before using the function `fftw_mpi_local_size_2d`; however, when allocating memory for the real array, the allocation should be doubled as shown in the following code:

```
ptrdiff_t alloc_local = fftw_mpi_local_size_2d(M, N, MPI_COMM_WORLD,
                                              &localM, &local0);
double* realdata = fftw_alloc_real(2*alloc_local);
fftw_complex* compdata = fftw_alloc_complex(alloc_local);
```

The memory would still be utilized the same as if the extra memory were not added, i.e., `realdata[i*N+j]` is still used to access the  $(i, j)$  entry in the array.

## Exercises

- 20.1. Modify Example 20.2 so that the integer wave modes  $k1, k2$  are part of the input and Lines 60, 61 are replaced with

```
58     datain [ i*N+j ][ 0 ] = cos( k1*x+k2*y );
59     datain [ i*N+j ][ 1 ] = sin( k1*x+k2*y );
```

Try different values of  $k1, k2$  to see how the coefficients change.

- 20.2. Modify Example 20.2 so that the input data is real-valued with initial value  $\cos(k1*x+k2*y)$  as in the previous example. Use the two plans discussed in Section 20.2.3 to do the forward and reverse transforms. Try different values of  $k1, k2$  to see how the coefficients change.
- 20.3. Implement the two-dimensional FFT following the algorithm in Section 20.2.1.  
**Extra challenge:** Implement it in the case where the dimensions of the domain are not evenly divisible by the number of processes.



## Chapter 21

# Projects for Distributed Programming

The background for the projects below is in Part VI of this book. You should be able to build each of these projects based on the information provided in this text and utilize MPI to implement parallel versions of these projects for use on a cluster. Because the algorithmic design in the context of cluster computing may differ significantly from the other types of parallelism in this book, there are some additional comments specific to MPI to give hints on how to develop your algorithms.

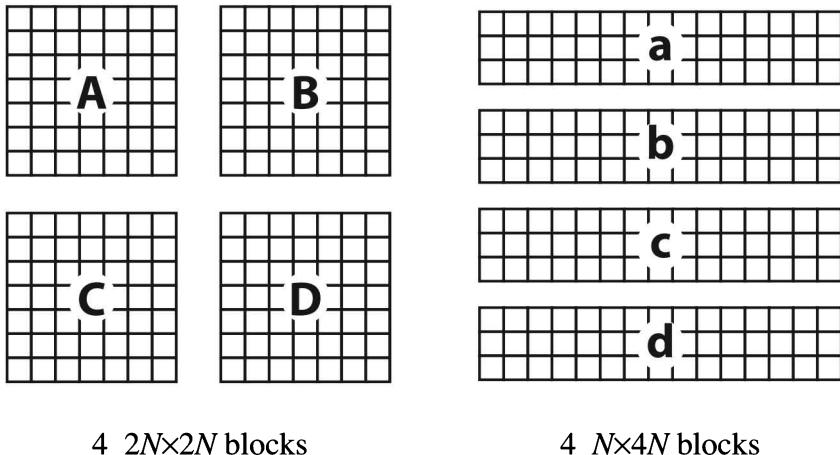
Before getting into the details of each application, there are some general comments to make concerning all the projects. First, when reading the parameter values from the command line, use the Rank 0 process to read the parameter values, and then use `MPI_Bcast` to distribute the information to the other processes as needed. Similarly, only the Rank 0 process should write data out to a file. This means using group communications such as `MPI_Gather` to collect the data in one place in order for it to be properly written. Finally, be sure to use the function `MPI_Wtime` to measure the execution time.

## 21.1 • Random Processes

Random processes are easily parallelizable because each individual calculation is independent of the others. The random number generators in each process must be initialized with different values, and then the various processes can be computed independently. One way to ensure that each process has a unique seed is to use a single seed for all processes, and then each process adds its rank to that seed. The only communications required are to use `MPI_Bcast` to share the parameters among the processes and `MPI_Reduce` to combine the results from the various processes at the end.

### 21.1.1 • Monte Carlo Integration

Program the assignment in Section 34.3.1 using  $N$  samples, where  $N$  is an input parameter. Measure the time required to complete the calculation by using the `MPI_Wtime` function at the beginning and end of the program and print the elapsed time in seconds. Use the plot to estimate the cost to compute a solution with error  $10^{-5}$  with 99.5% confidence. Generate a plot to show both the weak and the strong scaling of the algorithm.



**Figure 21.1.** Different choices for a two-dimensional spatial domain decomposition. Compare the number of grid points that will require communications and to how many different processes.

### 21.1.2 • Duffing–Van der Pol Oscillator

Program the assignment in Section 34.3.2 using  $N$  samples and using  $M$  time steps, i.e., take  $\Delta t = T/M$  in the Euler–Maruyama method. Measure the time required to complete the calculation by using the `MPI_Wtime` function at the beginning and end of the program and print the elapsed time in seconds. Plot an estimate for  $p(t)$  for  $0 \leq t \leq T = 10$ . Generate a plot to show both the weak and the strong scaling of the algorithm.

## 21.2 • Finite Difference Methods

### 21.2.1 • Domain Decomposition and MPI

A very common and useful strategy for distributed systems is domain decomposition. This is where the full problem domain is broken down into roughly equally sized sub-domains. Examples of this concept include the one-dimensional heat equation from Section 17.2, where the domain was subdivided into equal segments, and each segment was updated in parallel, and the implementation of a two-dimensional FFT in Section 20.2.1. For finite difference methods, domain decomposition requires neighboring subdomains to communicate enough boundary data so that each subdomain can complete a full update.

In one dimension, it is fairly straightforward to subdivide the domain, but consider a two-dimensional domain that has dimensions  $4N \times 4N$  grid points, and compare two different decompositions as depicted in Figure 21.1. In each case the domain is broken into four equal subdomains, but of differently sized shapes. In both cases, each processor is responsible for updating  $4N^2$  grid points; however, the communications demands will be quite different.

For the four square subdomains, each subdomain must communicate with each other subdomain. For example, the subdomain  $A$  shares a side of length  $2N$  with subdomain  $B$  and also  $C$ . It also shares a corner with subdomain  $D$ . The other subdomains are

similar. That means that roughly  $16N + 4$  numbers must be communicated each time step in 12 send/receive pairs, and they are heavily interdependent in order to advance in time.

For the case of four rectangular subdomains, subdomain  $a$  only shares a common side with subdomain  $b$ ,  $b$  only connects with  $a$  and  $c$ , and so forth. Each of those shared sides are of length  $4N$ . Adding up all the communications, there are  $24N$  numbers that must be communicated in 6 send/receive pairs, but there is less interdependency. Each subdomain has only one or two neighbors compared with three neighbors for everyone in the first case.

Which case is better? That depends on your application and the method of communications. Point-to-point communications will work with any configuration but are more difficult to coordinate in an efficient manner. Group communications rely on data being contiguous in memory to be efficient and so the  $2 \times 2$  arrangement of subdomains is not ideal. This point arises in many applications, so a more careful discussion of the arrangement of the data in relation to group communications is warranted.

Suppose  $u$  is an  $M \times N$  array declared statically as

```
double u[M][N];
```

or dynamically as

```
double (*u)[N] = malloc(M * sizeof(u));
```

where  $M$  and  $N$  are both evenly divisible by 2. Recall that data in C is stored in row-major form, so it is stored in the order indicated by the arrows in Figure 21.2. Thus, if the data is split along rows, contiguous data will remain contiguous after calling `MPI_Scatter` and will be reassembled back into the full array correctly after calling `MPI_Gather`, as illustrated in Figure 21.3. On the other hand, if the data is split along columns, then the data will no longer be contiguous, as illustrated in Figure 21.4. That means that a call to `MPI_Scatter` will not produce the desired results, and `MPI_Gather` will reassemble the data in an undesirable order. Thus, subdividing the domain into contiguous rows is most often the easiest way to manage interprocess group communications.

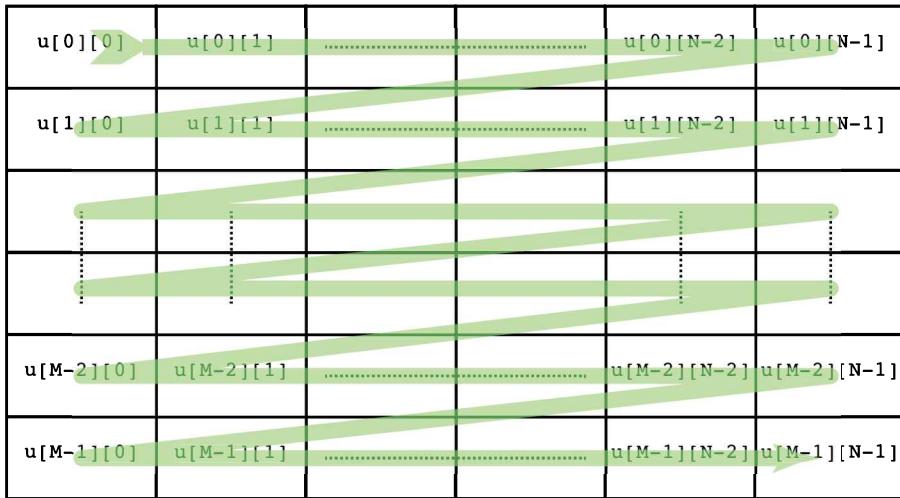
Before discarding the latter case completely, it is still possible to make such an arrangement of subdomains work—it just requires writing a transpose operation into the algorithm. By simply transposing the data, the data can be kept contiguous after all as illustrated in Figure 21.5.

There are instances where the  $2 \times 2$  style of subdomain arrangement does make sense, and then it is worth the effort to better understand the grid communicators discussed in Section 18.3. However, communications are only part of the algorithm, and data management as discussed above is still an important consideration in order to ensure that the layout of the grid remains consistent with the algorithm.

### 21.2.2 • ADI and MPI

Before studying this section, the reader may wish to review Section 35.3.2 to understand the background of how the ADI method works and the equations that must be solved to advance in time.

As a rough sketch of the ADI algorithm for MPI, suppose the full domain is  $4N \times 4N$  grid points and four processes will share the work, where the domain is subdivided as shown for Stage 1 in Figure 21.6. Stage 1 will have each of the four processes doing

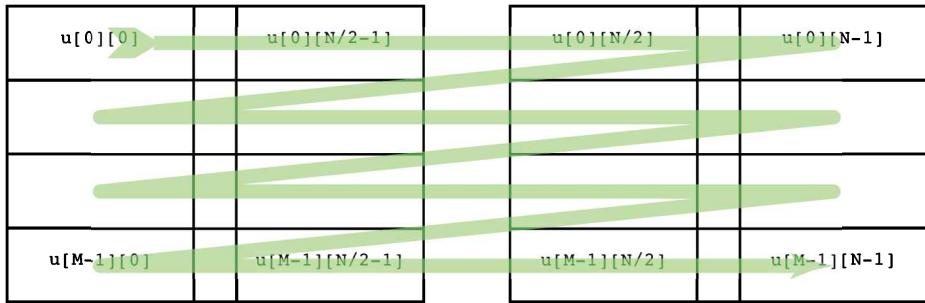


**Figure 21.2.** Illustration of the storage for a two-dimensional array. Data in  $C$  is stored in row-major form. The orientation of the data is important to understand in order to take advantage of group communications.

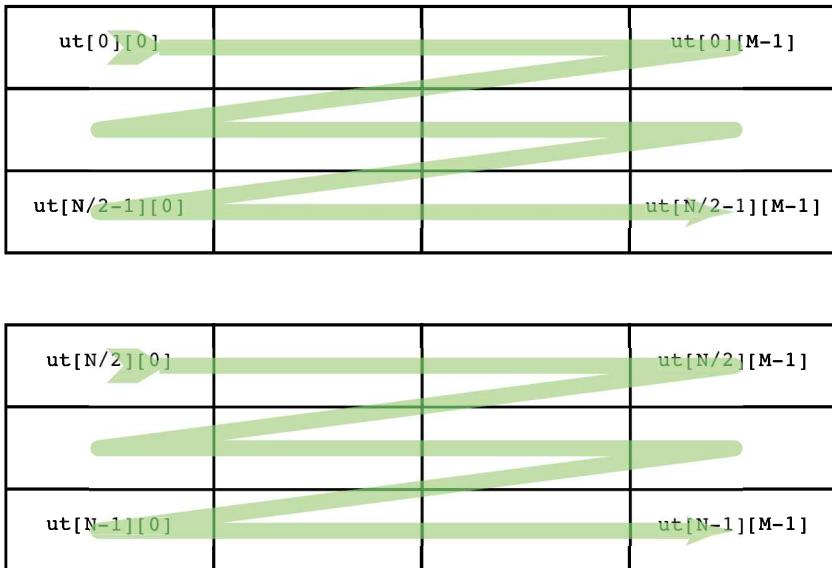


**Figure 21.3.** Illustration of how storage remains contiguous after a call to `MPI_Gather` or `MPI_Scatter` if the data is oriented in the same direction as the full array.

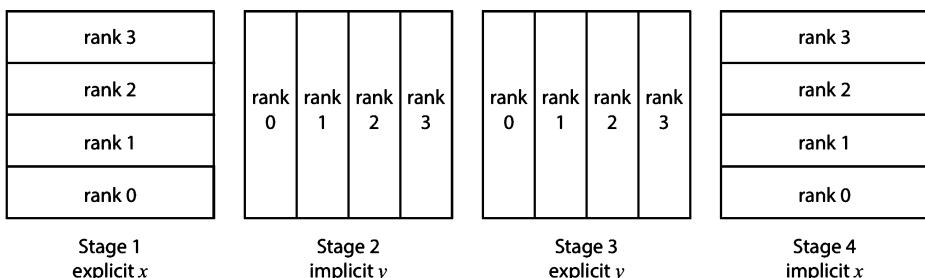
1/4 of the explicit update in the  $x$ -direction, which is not dependent on data in the  $y$ -direction, hence it can be done in parallel with no communication between processes. In Stage 2, the update is implicit in the  $y$ -direction. In order to do this update, the data must be organized along columns rather than rows. There are a few choices for how to handle this communication, which will be discussed below. Once the data is reoriented into vertical strips, then Stage 2 and Stage 3 can be done in order. After Stage 3, the data



**Figure 21.4.** Illustration of how storage is noncontiguous after a call to `MPI_Gather` or `MPI_Scatter` if the data is oriented in the opposite direction as the full array.



**Figure 21.5.** Illustration of how to arrange data when the desired subdomains are columns. Here, the original data  $u[M][N]$  is transposed to get  $ut[N][M]$  and then subdivided as in Figure 21.3.



**Figure 21.6.** Illustration of the four stages of the ADI algorithm for four processes. Each process gets a strip of the domain aligned with the  $x$ -direction to update. Between Stages 1 and 2 data must be transferred and transposed between the processes so that the strips can be updated in the  $y$ -direction. The transfer/transpose operation is repeated between Stages 3 and 4 to return to the  $x$ -direction.

must be put back in horizontal strips so that the final implicit update in the  $x$ -direction can be completed.

For this algorithm, the rough sketch sounds simple enough, but the devil is in the details. Before exploring the two approaches, there is one additional point that should be understood ahead of time. The implicit step updates will require the data in the long dimension to be contiguous in memory. This is a requirement because that data will be fed to a linear solver as a vector. If done right, the entire subdomain can be fed as the right-hand vector using the number of right-hand sides parameter for the linear solver so that they are all solved simultaneously in one LAPACK function call.

Suppose the data is arranged originally in horizontal strips. Recalling that  $C$  is stored as row-major, it means that if  $i$  is designated as the increment in the  $x$ -direction, and  $j$  is the increment in the  $y$ -direction, then the subdomain would be accessed by an expression such as  $u[j][i]$  because the  $x$ -direction needs to be contiguous in memory. That means that when doing updates in the  $y$ -direction, the data must also be transposed so that the columns become rows and the data can again be contiguous, and the data is accessed using  $u[i][j]$ . This issue will also be relevant when using gather/scatter operations because these operations work on linear arrays.

### ADI Using MPI\_Allgather

The first approach for doing ADI is where every process contains a copy of the full grid. This is the simplest to implement, but also the least efficient in terms of the amount of data that must be communicated. In addition, the size of the problem that can be solved in this method will necessarily be smaller due to memory constraints. With that in mind the algorithm stages and communications are below. Assume that each process begins with the current version of the data for the full grid laid out so that the  $x$ -direction is contiguous as illustrated in Figure 21.2.

**Stage 1:** The first step is to do the explicit update in the  $x$ -direction corresponding to the right-hand side of Equation (35.32). Each process does an update on its own subdomain leaving the other subdomains untouched. There is no dependency in the  $y$ -direction, so the updates can be done without any additional communications.

**Communications:** The data is laid out correctly for a gather operation, so all processes can be brought up to date using `MPI_Allgather`. The data will need to be in the  $y$ -direction next, so after the communication transpose the matrix in memory.

**Stage 2:** The implicit update in the  $y$ -direction is computed corresponding to the left-hand side of Equation (35.33). Apply the LAPACK solver `dgetrs_` or `dgttrs_` to the contiguous data in the  $y$ -direction, which is a result of the transpose completed in the communications step. Now each process applies the LAPACK routine to its portion of the data with the `nrhs` variable set to the number of rows the process must update in its subdomain.

**Stage 3:** The explicit update in the  $y$ -direction is next corresponding to the right-hand side of Equation (35.34). The update does not require the data to be in any particular orientation when each process has the full grid, so the data can be left in its current state.

Communications: Same as the previous communications task. Update all processes using `MPI_Allgather`, and then transpose the array in memory to get the  $x$ -direction to be contiguous.

Stage 4: This final step is an implicit step in the  $x$ -direction corresponding to the left-hand side of Equation (35.35). After the transpose in the communications step, the data is laid out as before Stage 1, and the LAPACK linear solver can be applied to the subset of rows each subdomain must update similar to Stage 2.

It should be readily apparent that this algorithm is pushing around a large amount of data during the communications stages. Nearly the entire grid is being sent from all processes to all the other processes twice each time step. So this is inefficient in terms of both memory storage and communication costs.

### ADI Using Gather and Scatter

This approach is a little more difficult to implement but has a much smaller memory footprint and communications cost. Assume to begin that each process has a copy of only its local subdomain oriented so that the  $x$ -direction is contiguous. The trick to this algorithm is distributing data correctly during the communications phases. The actual compute stages are the same as the previous approach.

Stage 1: The first step is to do the explicit update in the  $x$ -direction corresponding to the right-hand side of Equation (35.32). Each process does an update on its own subdomain.

Communications: Each process must send a portion of its local data to each of the other processes, so this is a natural place to use `MPI_Scatter`, but the data is not laid out in the correct way for this to work. As described in Sections 20.2.1 and 21.2.1, the data must first be transposed before it is sent via `MPI_Scatter`. Each process must receive this data from the scatter into the corresponding part of the subdomain array. At the end of this operation, each process will have a subdomain of the data with contiguous data running in the  $y$ -direction ready for Stage 2.

Stage 2: Do the implicit  $y$ -direction update on the subdomain data as in the previous algorithm.

Stage 3: Do the explicit  $y$ -direction update on the subdomain data as in the previous algorithm.

Communications: The previous communications method is repeated, where the data is transposed and then scattered using `MPI_Scatter`.

Stage 4: The previous communications phase reoriented the data so that the  $x$ -direction is now contiguous, so the implicit  $x$ -direction update is done similarly to the previous algorithm to complete the time step.

It should be evident now that if there are  $P$  processes, and the memory requirements for a given process in the first approach were  $M$ , then this second approach would require  $M/P$  for memory because each process only has to manage its own subdomain, not the full grid. Furthermore, where the first approach used communications that required

transmitting  $M(P - 1)P$  data during the communications phase, this second approach only transmits  $M(P - 1)$ . So the memory footprint is smaller and the communications costs are smaller. This can result in a significant performance improvement at the cost of a little more complicated algorithm.

The two approaches for this type of problem serve as an excellent case study showing that the algorithms in this parallel world can vary widely and result in significant differences in performance. Taking time to carefully consider the different options of how to implement these algorithms is well worth the effort.

### 21.2.3 • Brusselator Reaction

Program the assignment in Section 35.4.1 using an  $N \times N$  grid. Measure the time required to complete the calculation by using the `MPI_Wtime` function at the beginning and end of the program and print the elapsed time in seconds. Generate a plot to show both the weak and the strong scaling of the algorithm.

### 21.2.4 • Linearized Euler Equations

Program the assignment in Section 35.4.2 using an  $N \times N$  grid. Measure the time required to complete the calculation by using the `MPI_Wtime` function at the beginning and end of the program and print the elapsed time in seconds. Generate a plot to show both the weak and the strong scaling of the algorithm.

## 21.3 • Elliptic Equations and SOR

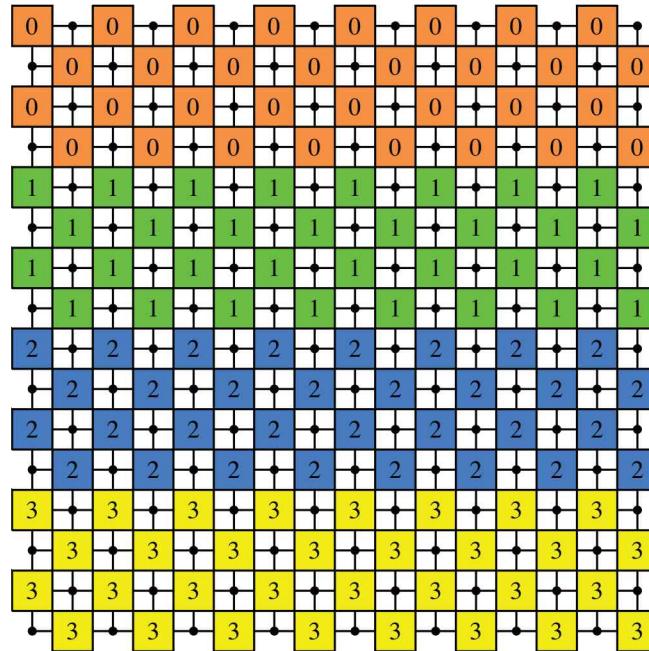
Red/black ordering is used on SOR type problems so that the update can be done in parallel. The grid is updated in two separate parallel passes, one for red and one for black, with communications between each pass. The data can be organized either so that each process contains a complete grid or so that each process contains a subdomain organized by rows with one row of grid points overlapping between subdomains. On the first pass, the red grid points are divided equally among all the processes and then updated using the SOR formula. After the red grid points are updated, the black points are similarly divided among the processes and updated. For simplicity, it is best to subdivide the domains by rows as discussed in Section 21.2.1.

In terms of communications, there are again two strategies, one where each process keeps a complete updated grid and one where each process has only a subdomain. If each process keeps a complete grid, then use `MPI_Allgather`, or `MPI_Allgatherv` if needed, to share the updated values on all the processes after each pass over the data. This is a rather wasteful way to manage communications since only the grid points immediately adjacent to a subdomain need to be updated.

The second communications strategy would be to send and receive only the one row immediately adjacent to the neighboring subdomains. This is a similar strategy to that described for the heat equation in Sections 17.2 and 17.3. In this way, storage for the domain is broken into pieces, thus using less memory per process. Also, for an  $N \times N$  grid distributed over  $P$  processes, the amount of data communicated between passes over the grid is  $O(PN)$  compared to  $O(PN^2)$  for the method using `MPI_Allgather`.

Whichever communications method is used, the subdivision of red/black grid points is illustrated in Figure 21.7.

During each iteration, each process will compute its own local value of the maximum residual, but no process should terminate until all processes compute a maximum



**Figure 21.7.** Illustration of updating red points in a red/black ordering for SOR. The numbered boxes correspond to the red points and the dots correspond to the black points. Each color/number represents the data that each of four cores would be responsible for computing. When all red points are updated, use `MPI_Allgather` to share the results with all neighbors or `MPI_Sendrecv` to share boundary data with neighbors. Data is cut along rows to keep the local data contiguous.

residual below the convergence threshold. To do this, at the end of each complete pass through the grid use the `MPI_Allreduce` function with the `MPI_MAX` operation so that each process receives the maximum residual among all the processes.

### 21.3.1 • Diffusion with Sinks and Sources

Program the assignment in Section 36.1.1 using a  $(2N - 1) \times N$  grid. Experiment to find the optimal  $\omega$  that requires the fewest iterations to reach a tolerance of  $10^{-9}$ . Compute the time required to complete the calculation, then divide by the number of iterations to give the average time used for a single pass through the grid. Generate a plot to show both the weak and the strong scaling of the algorithm.

### 21.3.2 • Stokes Flow 2D

Program the two-dimensional assignment in Section 36.1.2 using a MAC grid that is approximately  $N \times N$ , i.e., the grid for  $u$  should be  $N \times N - 1$ , the grid for  $v$  should be  $N - 1 \times N$ , and the grid for  $p$  should be  $N - 1 \times N - 1$ . Note that since the grids for  $u$ ,  $v$ , and  $p$  are all different dimensions, it won't be possible to divide them all equally among  $P$  processes, so you will need to use `MPI_Gatherv` to collect the final solution. Experiment to find the optimal  $\omega$  that requires the fewest iterations to reach a tolerance of  $10^{-9}$ . Compute the time required to complete the calculation, then divide by the

number of iterations to give the average time used for a single pass through the grid. Verify that you get a parabolic profile for the velocity field in the  $x$ -direction. Generate a plot to show both the weak and the strong scaling of the algorithm.

### 21.3.3 • Stokes Flow 3D

Program the three-dimensional assignment in Section 36.1.2 using a MAC grid that is approximately  $N \times N \times N$ , i.e., the grid for  $u$  should be  $N \times N - 1 \times N - 1$ , the grid for  $v$  should be  $N - 1 \times N \times N - 1$ , the grid for  $w$  should be  $N - 1 \times N - 1 \times N$ , and the grid for  $p$  should be  $N - 1 \times N - 1 \times N - 1$ . Note that since the grids for  $u$ ,  $v$ ,  $w$ , and  $p$  are all different dimensions, it won't be possible to divide them all equally among  $P$  processes, so you will need to use `MPI_Gatherv` to collect the final solution. Experiment to find the optimal  $\omega$  that requires the fewest iterations to reach a tolerance of  $10^{-9}$ . Compute the time required to complete the calculation, then divide by the number of iterations to give the average time used for a single pass through the grid. Verify that you get a roughly parabolic profile for the velocity field in the  $x$ -direction. Generate a plot to show both the weak and the strong scaling of the algorithm.

## 21.4 • Pseudospectral Methods

The FFTW package for computing the Fourier transform is already capable of using multiple processes, so it is best to use the MPI versions of the Fourier transforms directly. The spectral coefficients are also distributed between the processes, and hence when computing the derivatives in spectral space, each process can modify its locally stored coefficients without requiring any additional communications. Similarly, the update using the Runge–Kutta algorithm is also entirely local and does not require additional communications. Thus, the only explicit MPI communications needed to implement the pseudospectral algorithms will be to broadcast the parameter values at the beginning and using `MPI_Gather` or `MPI_Gatherv` to collect the data when saving it to a file.

### 21.4.1 • Complex Ginsburg–Landau Equation

Program the assignment in Section 37.5.1 using an  $N \times N$  grid. Measure the time required to complete the calculation by using the `MPI_Wtime` function at the beginning and end of the program and print the elapsed time in seconds. Plot the results for the phase and identify where the spiral waves have emerged through pattern formation. Generate a plot to show both the weak and the strong scaling of the algorithm.

### 21.4.2 • Allen–Cahn Equation

Program the assignment in Section 37.5.2 in two dimensions using an  $N \times N$  grid. You should use the real to complex plans as described in Section 20.2.3 so that the function values  $\phi$  can be kept as real-valued. Measure the time required to complete the calculation by using the `MPI_Wtime` function at the beginning and end of the program and print the elapsed time in seconds. Plot the results and verify that phase separation is occurring. Generate a plot to show both the weak and the strong scaling of the algorithm.

## **Part IV**

# **GPU Programming and CUDA**



In Parts II and III, parallelism using many processors, all of the same or a very similar type, are harnessed to solve a problem. Programming with GPUs is quite different and much more resembles the old vector processors popularized in the 1970s and 1980s. The idea was to have many processors that all do the same task simultaneously. On those computers, there were specialized compilers that could automatically convert a loop into a single command that would execute every iteration simultaneously in a process called loop unrolling. Unfortunately, programming GPUs is not nearly so automated, but that will likely change and evolve as the field matures.<sup>8</sup> In the meantime, GPUs offer substantial speed-ups over serial programs, or even codes written for distributed architectures, especially for the types of problems often solved in applied mathematics.

One of the leaders in the GPU development area is clearly NVIDIA. All modern NVIDIA graphics cards and many older model cards can be programmed using the tools covered in this book. The cards can be graphics cards that are used for driving the computer monitor, or they can be dedicated computational tools, not hooked up to any screen. To facilitate programming their GPU cards, NVIDIA developed an extension to the C programming language called CUDA. The CUDA language is specific to the NVIDIA cards, is tuned to take advantage of any special capabilities or features of their cards, and on the surface works very similarly to the `gcc` compiler used for this text.

In fact, the CUDA compiler is actually a wrapper that couples two distinct compilers into one. In CUDA, there will be special keywords and syntax that are not part of the standard C language. When the CUDA compiler, `nvcc`, is invoked, a preprocessor separates out the parts marked by the extra keywords and syntax and compiles that code to produce object code to be run on the GPU. The remaining code is then passed to the `gcc` compiler. So using `nvcc` actually invokes two distinct compilers to produce the final product, while all being transparent to the user.

The `nvcc` compiler is capable of compiling both C and C++ code and for both cases the suffix on the filenames is “`.cu`”.

Before getting started, it is worth pointing out that graphics cards made by other manufacturers, e.g., Intel or AMD, also have programming abilities. However, the software infrastructure to take advantage of those cards is not quite as well developed as CUDA. There is one notable exception, namely, openCL, which was designed to be agnostic toward the graphics card manufacturers and was proposed as a standard for graphics card developers to follow. OpenCL has fallen behind more recently but is still a viable alternative for non-NVIDIA cards and so is covered in Part V.

Almost any computer with an NVIDIA graphics card can be used for programming in CUDA. Before using CUDA, a special driver as well as the compiler and tools must be installed. To obtain the driver, go to the website <http://www.nvidia.com/cuda> and click on the download drivers link to find the correct driver for your computer, operat-

<sup>8</sup>OpenACC is an effort in this direction that utilizes GPUs through compiler directives in a very similar manner to OpenMP as described in Part II.

ing system, and graphics card model. The compiler and tools can be obtained by going to <http://developer.nvidia.com/object/gpucomputing.html> and looking for the CUDA toolkit. For both packages follow the installation instructions provided by NVIDIA.

In Chapter 22 the CUDA framework is introduced, including how to connect with and select the GPU device to be used and how to detect its relevant properties. Similar to the wrapper compiler `mpicc` introduced in Part III, the CUDA language compiler `nvcc` is introduced in this chapter.

CUDA is designed for massively parallel programs that are based on the device structure of blocks and threads. Chapter 23 introduces the concepts of blocks and threads and how they can be organized into one or more dimensions of coordinated processes each doing the same task simultaneously. At the end of Chapter 23 there is also a discussion about how to handle error messages from CUDA functions in a consistent manner.

GPU devices contain multiple types of memory each with different access speed to the cores and some having specialized properties and purposes. Chapter 24 discusses the various types of memory and their structure and speed, and then compares them for purposes of doing a simple finite difference calculation. The chapter also covers warp shuffles, which are a means for threads to read memory from other threads within the same warp.

In Chapter 25, streams are used to subdivide blocks and threads into parallel tasks. One advantage of using streams is that on NVIDIA graphics cards computation and device/host communications can be made to overlap to improve performance. The chapter concludes with information about how to use events in streams to measure the computational time required to run kernel functions.

Chapter 26 discusses some libraries provided by the CUDA environment that are similar to the libraries discussed in other parts of this book. In this case, the library MAGMA is comparable to LAPACK and written by many of the same authors, while the library cuFFT is comparable to FFTW. The chapter also introduces the cuRAND library, which is critical because random number generators like `drand48` are not available on the GPU hardware.

Chapter 27 concludes this part of the text with a discussion on how to solve the application problems from Part VI using the CUDA language and any specialized information related to a GPU implementation.

For more information on CUDA, the interested reader is referred to the texts by Sanders and Kandrot [18] and Cheng, Grossman, and McKercher [19].

# Chapter 22

# Intro to CUDA

When programming in CUDA, we must be careful to distinguish between operations being done by the CPU and those on the GPU. In keeping with the CUDA terminology, operations performed in main memory or by the CPU will be said to be on the host. Those operations or memory addresses located on the GPU are designated as being on the device. This chapter covers the first steps toward computing on the device. Using `printf` from within a kernel function is possible, but it should be done only for debugging purposes. Because of that, the introductory program will be a simple addition of two numbers. We will see how to compile the program using the CUDA compiler `nvcc` and how to select a particular device when given multiple choices.

## 22.1 • First CUDA Program

CUDA is more than simply linking in some useful libraries or compiler directives like those used in earlier parts of this text—it contains extensions to the C language that a regular C compiler does not understand. Therefore, compiling CUDA code will require using the CUDA compiler, `nvcc`, in order to handle the CUDA language extensions. Files with CUDA code embedded in them will have the suffix “`.cu`” instead of the typical “`.c`” for C code so that the compiler knows when to look for CUDA specific content. Note that “`.cu`” files compiled with `nvcc` and “`.c`” files compiled with `gcc` can be linked together at the linking stage. Example 22.1 is a simple CUDA program which would be stored in a file named, for example, `hello.cu`:

### Example 22.1.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 /*
5  void add(int a, int b, int* c)
6
7  adds the first two input arguments and places the result in the third
8  argument
9
10 Inputs:
11   int a, b: two addends
12
13 Output:
```

```

14  int* c: pointer to the address where the result will be stored.
15  The address is assumed to be a GPU memory address.
16 */
17 __global__ void add(int a, int b, int* c) {
18     *c = a + b;
19 }
20 /*
21 int main(int argc, char* argv[])
22
23 Demonstration of a simple program that uses a GPU kernel function.
24 It takes two input values, adds them together on the GPU, and then
25 brings the results from the GPU back into the CPU for output.
26
27 Inputs: argc should be 3
28 argv[1]: first addend
29 argv[2]: second addend
30
31 Outputs:
32 Displays the resulting sum
33 */
34
35 int main(int argc, char* argv[]) {
36
37 // Read the input values
38 int a = atoi(argv[1]);
39 int b = atoi(argv[2]);
40
41 // c is the storage place for the main memory result
42 int c;
43
44 // dev_c is the storage place for the result on the GPU (device)
45 int *dev_c;
46
47 // Allocate memory on the GPU to store the result
48 cudaMalloc((void**)&dev_c, sizeof(int));
49
50 // Use one GPU unit to perform the addition and store the result
51 // in dev_c
52 add<<<1,1>>>(a, b, dev_c);
53
54 // Move the result into main memory from the GPU
55 cudaMemcpy(&c, dev_c, sizeof(int), cudaMemcpyDeviceToHost);
56
57 printf("%d + %d = %d\n", a, b, c);
58
59 // Free the allocated memory on the GPU.
60 cudaFree(dev_c);
61
62 return 0;
63 }
```

The language extensions that are most obvious are on Lines 17, 52, combined with some new functions on Lines 48, 55, and 60. Recall that multiple compilers are required for GPU programming, there's the regular compiler, e.g., `gcc`, that takes C code and turns it into object code for the CPU, and there's the NVIDIA compiler that takes C code and turns it into object code for the GPU. The object codes are different, and so they require different compilers. The keyword `__global__` on Line 17 is one way to demarcate functions that will execute on the GPU instead of the CPU. This kind of function is often called a kernel function. On Line 52 the kernel function is actually

called. The triple angle brackets are another C language extension added for CUDA to identify a kernel function from a regular C function. The numbers in the brackets are for identifying the amount of GPU resources that will be allocated for the kernel task.

Memory on the device is also allocated in a different way on Line 48. It is important to remember that there are now two sets of memory addresses, conventional or CPU memory addresses on the host, and device or GPU memory addresses. They are distinct and cannot be mixed. Attempting to access a GPU pointer in CPU memory will most often result in a segmentation fault, and vice versa. While the two types are both pointers to `int` in this case, the actual addresses of the memory on the host and the device are not likely to be in the same range. It's akin to taking an address like 123 Main St., Seattle, WA, and then looking for 123 Main St., New York, NY; you're not likely to end up where you meant to go.

On Line 48, space for an integer is allocated in device memory, not CPU memory. This is required because when the kernel function `add` is executed on the GPU, it must store its result in GPU memory space. In order to use the result in the main code, it has to be copied back into CPU memory, and that is done on Line 55. Finally, as has been emphasized often in this text, always clean up when done, and that includes freeing memory allocated on the GPU card, which is done on Line 60.

## 22.2 • Selecting the Correct GPU

Not all GPUs are the same, and for some of them the difference can be quite important. For example, not all GPUs are capable of doing double precision floating point calculations. If the desired device is not explicitly set, then the system will choose, which may lead to trouble or get less than optimal results. Table 22.1 shows some data about the GPUs for the machine used to generate the examples. There are two Tesla K20c cards, which are cards specifically designed for GPU computing, while the on-board GPU is the Quadro K600 that is meant to drive the terminal screen. At the beginning of a CUDA program, the device to be used can be specified by setting the characteristics the device must have, and then asking the CUDA runtime environment to select the closest match. The call to `cudaSetDevice` must occur before any direct access to the GPU is initiated, e.g., by allocating memory.

**Table 22.1.** Comparison of two different GPU units on a representative system. There are two Tesla K20c units and one Quadro K600. (a) Compute capability currently range from 1.0 to 7.0. (b) Compute capability 7.0 has 98,304 bytes.

Device Name	Quadro K600	Tesla K20c
Compute capability <sup>a</sup>	3.0	3.5
Clock rate	876MHz	706MHz
Multiprocessors	1	13
Cores per multiprocessor	192	192
Total cores	192	2496
Warp size	32	32
Shared memory per block <sup>b</sup>	49,152	49,152
Constant memory	65,536	65,536
Texture memory	65,536	65,536
Global memory	1Gb	5Gb

**Example 22.2.**

```

1 #include <stdio.h>
2 #include <cuda.h>
3 #include <cuda_runtime.h>
4
5 int main(int argc, char* argv[]) {
6
7     cudaDeviceProp prop;
8     int dev;
9
10    // set all entries to 0 to mean no preference
11    memset( &prop, 0, sizeof(cudaDeviceProp));
12    // look for GPU with at least 13 cores
13    prop.multiProcessorCount = 13;
14
15    cudaChooseDevice(&dev, &prop);
16    cudaSetDevice(dev);
17
18    printf("Chose device #%d\n", dev);
19    return 0;
20 }
```

Example 22.2 demonstrates the process of choosing the GPU device. On Line 7, a `cudaDeviceProp` variable is declared, which is a C struct with many variables such as those in Table 22.1 contained in it. The function `memset` on Line 11 sets all the values in that struct to zero, indicating there are no preferences for any of the values. One way to distinguish between the Quadro K600 and the Tesla K20c is to look at the number of multiprocessors. By specifying the `multiProcessorCount` on Line 13 to be 13, the Quadro K600 is eliminated from consideration. The device properties in the variable `prop` are compared against the available devices and an integer device ID is returned in the variable `dev` on Line 15. Finally, the choice of the device is set on Line 16.

Once a GPU is selected, it can be useful to determine some key characteristics for the device selected, such as the shared memory size, or the maximum number of threads per block, or other quantities for that particular GPU. All of that information can be retrieved using the same `cudaDeviceProp` variable by calling

```
cudaGetDeviceProperties(&prop, dev);
```

which was how Table 22.1 was assembled. Check the CUDA documentation for a complete list of the components of the `cudaDeviceProp` structure. Specific information such as the amount of constant memory on a device can also be requested using the `cudaDeviceGetAttribute` function like this:

```
int value;
cudaDeviceGetAttribute(&value, cudaDevAttrTotalConstantMemory, dev);
```

Check the documentation for `cudaDeviceGetAttribute` for a complete list of information that can be requested.

Note that for brevity the subsequent examples will skip past the device selection step because the default device is the one desired, but it should normally be included.

## 22.3 • CUDA Development Tools

There are a few CUDA development tools that are worth mentioning right up front. To take full advantage of these tools, compile with the flags `-g -G` to encode debugging

information into the binary. Those flags should be removed when the code is ready for full usage.

### 22.3.1 • Debugging with `cuda-gdb`

The tool `cuda-gdb` is useful for debugging purposes where kernel functions can be stepped through to locate errors. It has the same commands as the `gdb` tool described in Section 1.5, but `cuda-gdb` is also able to debug inside kernels. There are additional commands for selecting the specific block or thread that is to be observed via the commands `cuda block` and `cuda thread`. For help with the CUDA-specific features of this tool, type `help cuda` at the (`cuda-gdb`) command prompt for a list of available options. For a more extensive description, the interested reader is referred to the NVIDIA documentation at <https://docs.nvidia.com/cuda/cuda-gdb/index.html>

### 22.3.2 • Profiling with `nvvvp` and `nvprof`

The CUDA tools `nvvvp` and `nvprof` are useful for checking the efficiency of code and looking for bottlenecks in performance. The tool `nvvvp` is a graphical interface for exploring the GPU and CPU usage for a program, while `nvprof` provides text output of the data.

Suppose the code in Example 22.1 were compiled into an executable named `add`; then to analyze the performance of the program using `nvvvp`, simply insert `nvvvp` before invoking the command to execute the program. In other words, use the command

```
$ nvvp add 5 6
```

Figure 22.1 shows a view of the graphical interface that appears as a result. In the figure, the top part of the display shows time bars indicating when certain tasks started and stopped to check for concurrency of kernels, memory transfers, and other activity so that it is easy to identify where bottlenecks arise. On the left, the different threads within the system are identified. The host program is shown first where the runtime CUDA functions such as `cudaMalloc` that are called from the host are shown. The next line concerns CUDA functions that are not called directly but are invoked in order to make CUDA work, and for the most part these are beyond direct control by the programmer. The profiling overhead are operations that are there only because the `nvvvp` tool was invoked.

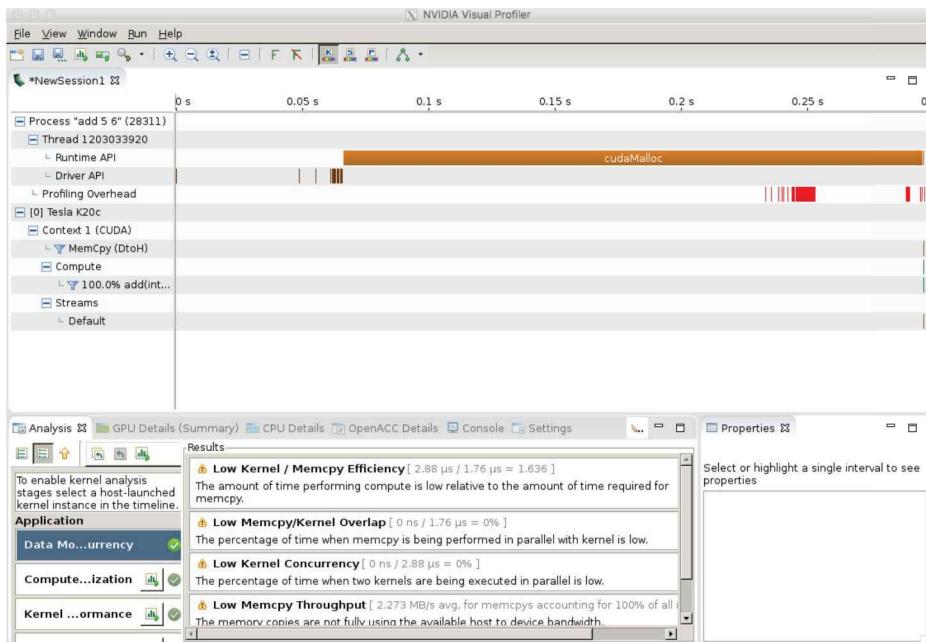
The next set of lines concerns operations performed on the device and they are broken down into memory transfers and kernel operations. The distinctions between these two types of operations and how they interact with streams will be discussed in Chapter 25. For this example, there is one memory transfer when the sum is transferred back to the host for printing, and there is one kernel function called `add`, and they are all handled on the default stream.

At the bottom of the window are a number of tabs where tools for analyzing the results can be found. The Analysis tab gives recommendations for how to improve performance. For example, in the tool for checking data movement and concurrency the analysis mentions that the compute time is low relative to the time required to transfer the sum from the device back to the host.

It should be glaringly obvious in this chart that memory allocation on the device is by far the most expensive operation. For this extremely simple example, this is more exaggerated than would be for a more practical application. Nonetheless, it is an obser-

vation worth remembering when developing code that memory allocation is expensive.

There are many more features offered by this tool than can be addressed in this book. The interested reader is referred to the extensive documentation provided on the NVIDIA development site [20].



**Figure 22.1.** Example use of nvvp to analyze the program in Example 22.1. The chart in the upper right shows bars indicating when tasks run and for how long. For this simple example, the cost of `cudaMalloc` dominates the time compared to the time used by the kernel indicated by the bar at the far right on the line showing “100.0% add(int...). The bottom of the display provides many details including an analysis tool that can make suggestions for improvement.

If a graphical front end is unavailable or when working remotely and the bandwidth to run the window is prohibitive, the text-only tool `nvprof` is used for a simple dump of GPU usage. The program is invoked in the same manner as for `nvvp` by prepending `nvprof` before the execution. A sample of the resulting output is shown below:

```
--=30665== NVPROF is profiling process 30665, command: ./add 5 6
5 + 6 = 11
--=30665== Profiling application: ./add 5 6
--=30665== Profiling result:
      Type  Time(%)    Time    Calls      Avg      Min      Max   Name
GPU activities :  70.49%  4.1280us     1  4.1280us  4.1280us  4.1280us  add(int, int, int*)
                29.51%  1.7280us     1  1.7280us  1.7280us  1.7280us  [CUDA memcpy DtoH]
API calls :  98.17% 270.44ms     1  270.44ms 270.44ms 270.44ms  cudaMalloc
                1.34% 3.6814ms  288 12.782us   356ns 793.91us  cuDeviceGetAttribute
                0.19% 518.05us     3 172.68us  89.386us 217.64us  cuDeviceTotalMem
                0.10% 270.26us     3 90.086us  83.259us 102.41us  cuDeviceGetName
                0.10% 264.22us     1 264.22us  264.22us 264.22us  cudaFree
                0.08% 222.71us     1 222.71us  222.71us 222.71us  cudaLaunchKernel
                0.01% 38.401us     1 38.401us  38.401us 38.401us  cudaMemcpy
                0.01% 30.792us     3 10.264us  3.8900us 21.658us  cuDeviceGetPCIBusId
                0.00% 8.1300us     6 1.3550us  401ns 2.6090us  cuDeviceGet
                0.00% 4.0740us     3 1.3580us  358ns 2.3000us  cuDeviceGetCount
```

Upon completion of the program it shows how much time is spent in each kernel, doing memory transfers between device and host, and calling various CUDA functions. The lines listed for GPU activities are where computational time is taken on the device, while the API calls are for time taken on the host. Some of these functions will be discussed later, such as `cudaMalloc`, `cudaFree`, and `cudaMemcpy`, while the function `cudaLaunchKernel` is the equivalent of the “`add<<< >>>`” notation. Here too `cudaMalloc` dominates the overall cost of the program.

### 22.3.3 • Memory Checking with `cuda-memcheck`

The CUDA equivalent to the `valgrind` tool discussed in Section 1.6 for checking for memory bounds and leaks is `cuda-memcheck`. Suppose that Line 18 is changed to

```
18 c[1] = a + b;
```

so that the value is stored in the wrong, and unallocated, location. This problem is easily found using `cuda-memcheck` by using the command

```
$ cuda-memcheck add 5 6
```

which produces the output

```
===== CUDA-MEMCHECK
===== Invalid __global__ write of size 4
===== at 0x000000e0 in /piggy/home/chopp/book_examples/1D/add.cu:18:add(int, int, int*)
===== by thread (0,0,0) in block (0,0,0)
===== Address 0xb03d000004 is out of bounds
===== Saved host backtrace up to driver entry point at kernel launch time
===== Host Frame:/usr/lib/x86_64-linux-gnu/libcuda.so.1 (cuLaunchKernel + 0x2cd) [0x24881d]
===== Host Frame:./add [0x18802]
===== Host Frame:./add [0x189f7]
===== Host Frame:./add [0x47785]
===== Host Frame:./add [0x367a]
===== Host Frame:./add [0x3549]
===== Host Frame:./add [0x3592]
===== Host Frame:./add [0x335e]
===== Host Frame:/lib/x86_64-linux-gnu/libc.so.6 (__libc_start_main + 0xf0) [0x20830]
===== Host Frame:./add [0x3199]

=====
===== Program hit cudaErrorLaunchFailure (error 4) due to "unspecified launch failure" on CUDA API call to cudaMemcpy.
===== Saved host backtrace up to driver entry point at error
===== Host Frame:/usr/lib/x86_64-linux-gnu/libcuda.so.1 [0x3453b3]
===== Host Frame:./add [0x3a32f]
===== Host Frame:./add [0x3378]
===== Host Frame:/lib/x86_64-linux-gnu/libc.so.6 (__libc_start_main + 0xf0) [0x20830]
===== Host Frame:./add [0x3199]

=====
5 + 6 = 1
=====
Program hit cudaErrorLaunchFailure (error 4) due to "unspecified launch failure" on CUDA API call to cudaFree.
===== Saved host backtrace up to driver entry point at error
===== Host Frame:/usr/lib/x86_64-linux-gnu/libcuda.so.1 [0x3453b3]
===== Host Frame:./add [0x422a6]
===== Host Frame:./add [0x339e]
===== Host Frame:/lib/x86_64-linux-gnu/libc.so.6 (__libc_start_main + 0xf0) [0x20830]
===== Host Frame:./add [0x3199]

=====
ERROR SUMMARY: 3 errors
```

The first thing `cuda-memcheck` catches is the “`Invalid __global__ write of size 4`” on Line 18. This is exactly the error that was created where the sum was attempted to be saved to a memory address that wasn’t allocated. The other errors are fallout triggered from that original error.

The `cuda-memcheck` tool has other tools built into it as well, namely, `racecheck`, `synccheck`, and `initcheck`. To access these tools, use the `--tool` option, e.g.,

```
$ cuda-memcheck --tool racecheck add 5 6
```

The `memcheck` and `racecheck` tools are often the most useful for CUDA beginners.

## Exercises

- 22.1. Build and run Example 22.1 to make sure you are able to access CUDA and a suitable GPU.
- 22.2. Modify Example 22.2 to report the following information about your GPU device. The property names listed in the table below are from the `cudaDeviceProp` structure, e.g., use `prop.totalGlobalMem` to get the total available global memory on the device.

Device Information	Property name	Type
Global memory	<code>totalGlobalMem</code>	<code>size_t</code>
Shared memory per block	<code>shareMemPerBlock</code>	<code>size_t</code>
Warp size	<code>warpSize</code>	<code>int</code>
Max block dimensions	<code>maxGridSize</code>	<code>int[3]</code>
Max thread dimensions	<code>maxThreadsDim</code>	<code>int[3]</code>
Max threads per block	<code>maxThreadsPerBlock</code>	<code>int</code>
Can execute concurrent kernels	<code>concurrentKernels</code>	<code>int</code>
Can execute concurrent memory transfer and kernels	<code>deviceOverlap</code>	<code>int</code>

- 22.3. Compile and debug Example 22.1 using the commands

```
$ nvcc -g -G add.cu -o add
$ cuda-gdb add
(cuda-gdb) break main
(cuda-gdb) run 5 6
```

Try using the `step`, `next`, and `continue` commands in the debugger to step through the program. The `step` and `next` commands are similar except that `next` does not descend into any subroutines, while `step` does.

- 22.4. Try running the `nvprof` tool on Example 22.1.
- 22.5. Try running the `cuda-memcheck` tool on Example 22.1. Alter the program to generate an error that the tool will detect.

# Chapter 23

# Parallel CUDA Using Blocks

Adding two single numbers is not the best use of a GPU device; it's time to take advantage of the high level of parallelism GPU devices have to offer. To that end, this chapter covers the organizational layout of the device into blocks of threads and to build kernel functions that can use this organization to do simultaneous computations on arrays of data.

## 23.1 • Running Kernels in Parallel

Example 23.1 is an example of a parallel code, where a simple finite difference approximation is calculated on a one-dimensional data set. For more information about finite difference approximations, see Chapter 35.

### Example 23.1.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #ifndef M_PI
5 #define M_PI 3.1415926535897932384626433832795
6 #endif
7
8 /*
9  void diff(double* u, int* N, double* dx, double* du)
10
11 Compute the central difference operator on periodic data
12
13 Inputs :
14     double* u: Function data, assumed periodic
15     int* N: pointer to the length of the data array
16     double* dx: pointer to the space step size
17
18 Outputs :
19     double* du: central difference of the u data
20 */
21 __global__ void diff(double* u, int* N, double* dx, double* du) {
22     // blockIdx is a CUDA provided constant that tells
23     // the block index within the grid
24     int tid = blockIdx.x;
25     // Notice there's no loop, each core will perform its operation on
```

```

26 //      its own entry but some cores should not participate if they
27 //      are outside the range.
28 if (tid < *N) {
29     int ip = (tid+1)%(*N);
30     int im = (tid+*N-1)%(*N);
31     du[tid] = (u[ip]-u[im])/(*dx)/2.;
32 }
33 }
34 */
35 /*
36 int main(int argc, char* argv[])
37 Demonstrate a simple example for implementing a
38 parallel finite difference operator
39
40 Inputs: argc should be 2
41 argv[1]: Length of the vector of data
42
43 Outputs: the initial data and its derivative.
44 */
45 int main(int argc, char* argv[]) {
46     int N = atoi(argv[1]); // Get the length of the vector from input
47
48 // These are addresses into host memory
49 double* u = (double*)malloc(N*sizeof(double)); // function data
50 double* du = (double*)malloc(N*sizeof(double)); // derivative data
51
52 // These are addresses into device memory, the "dev_" is optional
53 double* dev_u; // function data
54 double* dev_du; // derivative data
55 double* dev_dx; // space step size
56 int* dev_N; // array length
57
58 // Allocate memory on the device
59 cudaMalloc((void**)&dev_u, N*sizeof(double));
60 cudaMalloc((void**)&dev_du, N*sizeof(double));
61 cudaMalloc((void**)&dev_dx, sizeof(double));
62 cudaMalloc((void**)&dev_N, sizeof(int));
63
64 // Initialize the function data on the host
65 double dx = 2*M_PI/N;
66 for (int i=0; i<N; ++i)
67     u[i] = sin(i*dx);
68
69 // copy the input data from the host to the device
70 cudaMemcpy(dev_dx, &dx, sizeof(double), cudaMemcpyHostToDevice);
71 cudaMemcpy(dev_u, u, N*sizeof(double), cudaMemcpyHostToDevice);
72 cudaMemcpy(dev_N, &N, sizeof(int), cudaMemcpyHostToDevice);
73
74 // execute the finite difference kernel using N blocks
75 diff<<<N, 1>>>(dev_u, dev_N, dev_dx, dev_du);
76
77 // copy the result from the device back to the host.
78 cudaMemcpy(du, dev_du, N*sizeof(double), cudaMemcpyDeviceToHost);
79
80 for (int i=0; i<N; ++i)
81     printf("%lf\t%lf\n", u[i], du[i]);
82
83 // clean up all the allocated memory
84 cudaFree(dev_u);
85 cudaFree(dev_du);
86 cudaFree(dev_dx);
87 cudaFree(dev_N);
88 free(u);

```

```

89     free(du);
90     return 0;
91 }
```

Example 23.1 takes as input the length of the initial data vector and then computes the central difference operator on the data, which is assumed periodic, in parallel on the GPU. The basic outline of the algorithm here is that the data is initialized in host memory and copied to the device memory where the finite difference is computed, and the result is copied back to host memory to be written to the terminal.

On Lines 49–62, memory is allocated on both the host (Lines 49–50) and the device (Lines 59–62). It's important to remember that pointers are nothing more than long integers that give the index into available memory and do not have any hard connection to their allocation. That means that there are no safeguards for checking that a given memory address is a host memory address or a device memory address. The C types are the same, so the compiler can't distinguish between them. That means it is the programmer's responsibility to ensure the two do not get confused. One crutch for ensuring this is to use a label on host and/or device pointers that indicates their use. Here, the prefix “`dev_`” is used to label memory allocated on the device; it is not a language requirement to use such labels.

The memory allocation for host memory has been discussed in Part I and should be familiar by now. Memory allocation on the device is similar but has a slightly different form:

```
cudaMalloc(void** dev_ptr, size_t length);
```

Here, a reference to the device memory pointer is passed as the first argument (note the “`&`” in the first argument on Lines 59–62) and the number of bytes to be allocated in the second argument. As has been discussed many times, any memory that is allocated must also be deallocated, and as expected, there is a separate function for releasing device memory as compared to host memory. To release device memory, use the function `cudaFree` in the same manner that `free` is used for host memory. In this example, both types of memory are released on Lines 84–89.

Data is transferred between the host and the device through the `cudaMemcpy` function:

```
int cudaMemcpy( void* dest_ptr,
               void* src_ptr,
               size_t length,
               enum cudaMemcpyKind direction)
```

On Lines 70–72, the input data is copied from the host to the device, and on Line 78, the solution data is copied from the device back to the host. The direction of the operation is determined by the last argument, which can be any of

- `cudaMemcpyHostToDevice`
- `cudaMemcpyDeviceToHost`
- `cudaMemcpyDeviceToDevice`
- `cudaMemcpyHostToHost`

Again, it is up to the programmer to ensure that the pointers in the first two arguments match the destination and source locations as listed in the direction. Failure to do so won't be caught by the compiler, but using the wrong type of pointer will give unpredictable results. Note also that of all the operations in this example, by far the most expensive is the transfer of data to and from the device. For this reason, it is generally best to keep calculations on the device as much as possible.

The kernel function `diff`, which is defined on Lines 21–33 and called on Line 75, is where the finite difference is computed. Only device memory is used inside the kernel function. Device operations do not have access to host memory directly, which is why data must be transferred back and forth before calling the kernel. When the kernel is called, the values in the angle brackets mean the kernel requests  $N$  blocks with one thread per block. The relationship between blocks, threads, and warps will be discussed in the next section. There is no loop in the kernel because each block will compute one entry in the array. It's similar to how this could be done in MPI, where each process would compute one value in the result vector, and then use `MPI_Gather` to reassemble the output into a single array. In this case, the equivalent of the MPI rank is the `blockIdx` structure that is supplied by the CUDA compiler. Blocks can be multi-dimensional, but here only one dimension is used, so `blockIdx.x` gives the equivalent of the rank. If there are more cores than the length of the array, then `blockIdx.x` is tested to make sure that the index into the block is within the range of the array on Line 28.

This is not the most efficient way to implement this code, but it is a start. To see how it scales compared to a serial code, see Figure 25.5.

## 23.2 • Organization of Blocks

In the preceding example, the blocks were organized into a linear fashion, namely, a single linear list of  $N$  blocks, but that is not a requirement. Blocks can be organized into one-, two-, or three-dimensional grids, which facilitates mapping computational grids onto GPU processors by organizing the layout of the domain to match the layout of the GPU blocks.

A grid of blocks is created by using the CUDA-supplied `dim3` type. To create an  $N_1 \times N_2$  two-dimensional grid of blocks Line 75 would be replaced with

```
75 dim3 blockdim( N1,N2 );
76 diff<<<blockdim , 1>>>(dev_u , dev_N , dev_dx , dev_du );
```

When a third argument is not supplied, the CUDA compiler automatically fills in the third dimension as 1. For the Tesla K20c GPUs, the maximum block dimensions are (2147483647, 65535, 65535). That does not mean that it's possible to run  $2,147,483,647 \times 65,535 \times 65,535 = 9,223,090,559,730,712,575$  parallel blocks! The number of parallel computations is still limited to the number of cores listed in Table 22.1.

Inside an invocation of a kernel, the `blockIdx` structure gives the coordinates within the block grid given by (`blockIdx.x`, `blockIdx.y`, `blockIdx.z`). The dimensions of the grid are also given by a CUDA supplied structure (`gridDim.x`, `gridDim.y`, `gridDim.z`). The two-dimensional Laplacian is computed in Example 23.2 to illustrate the use of multiple block dimensions.

**Example 23.2.**

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4
5 #ifndef M_PI
6 #define M_PI 3.1415926535897932384626433832795
7 #endif
8
9 // declare the kernel here, the definition is at the end of the file
10 __global__ void diff(double* u, int* M, int* N, double* dx, double* dy,
11                      double* du);
12
13 /*
14  int main(int argc, char* argv[])
15 Demonstrate a simple example for implementing a
16 parallel finite difference operator on a 2D array
17
18 Inputs: argc should be 3
19 argv[1]: Size of first dimension
20 argv[2]: Size of second dimension
21
22 Outputs: the initial data and its derivative.
23 */
24
25 int main(int argc, char* argv[]) {
26     int M = atol(argv[1]);
27     int N = atol(argv[2]);
28     // Allocate the memory on the host
29     double *u = (double*)malloc(M*N*sizeof(double));
30     double *du = (double*)malloc(M*N*sizeof(double));
31
32     double* dev_u;
33     double* dev_du;
34     double* dev_dx;
35     double* dev_dy;
36     int* dev_M;
37     int* dev_N;
38
39     // Allocate the memory on the device
40     cudaMalloc((void**)&dev_M, sizeof(int));
41     cudaMalloc((void**)&dev_N, sizeof(int));
42     cudaMalloc((void**)&dev_u, M*N*sizeof(double));
43     cudaMalloc((void**)&dev_du, M*N*sizeof(double));
44     cudaMalloc((void**)&dev_dx, sizeof(double));
45     cudaMalloc((void**)&dev_dy, sizeof(double));
46
47     // Initialize the data on the host.
48     double dx = 2*M_PI/M;
49     double dy = 2*M_PI/N;
50     for (int i=0; i<M; ++i)
51         for (int j=0; j<N; ++j)
52             u[i+j*M] = sin(i*dx)*cos(j*dy);
53
54     // Copy the parameters and data to the device
55     cudaMemcpy(dev_dx, &dx, sizeof(double), cudaMemcpyHostToDevice);
56     cudaMemcpy(dev_dy, &dy, sizeof(double), cudaMemcpyHostToDevice);
57     cudaMemcpy(dev_u, u, M*N*sizeof(double), cudaMemcpyHostToDevice);
58     cudaMemcpy(dev_M, &M, sizeof(int), cudaMemcpyHostToDevice);
59     cudaMemcpy(dev_N, &N, sizeof(int), cudaMemcpyHostToDevice);
60
61     // Construct a 2D grid of blocks

```

```

62    dim3 meshDim(M,N) ;
63
64    // Invoke the kernel using the 2D array of blocks
65    diff<<<meshDim,1>>>(dev_u , dev_M, dev_N, dev_dx , dev_dy , dev_du);
66
67    // Copy the results back to the host
68    cudaMemcpy(du , dev_du , M*N*sizeof(double) , cudaMemcpyDeviceToHost);
69
70    // Release the memory allocated on the device
71    cudaFree(dev_u);
72    cudaFree(dev_du);
73    cudaFree(dev_dx);
74    cudaFree(dev_dy);
75    cudaFree(dev_M);
76    cudaFree(dev_N);
77
78    // Release the memory allocated on the host
79    free(u);
80    free(du);
81    return 0;
82 }
83
84 /*
85 void diff(double* u, int* M, int* N, double* dx, double* dy,
86           double* du)
87
88 Compute the central difference operator on periodic data
89
90 Inputs:
91   double* u: Function data, assumed periodic
92   int* M: pointer to the x length of the data array
93   int* N: pointer to the y length of the data array
94   double* dx: pointer to the x space step size
95   double* dy: pointer to the y space step size
96
97 Outputs:
98   double* du: Laplacian of the u data
99 */
100 __global__ void diff(double* u, int* M, int* N, double* dx, double* dy,
101                      double* du) {
102     if (blockIdx.x < *M && blockIdx.y < *N) {
103         int ij00 = blockIdx.x + blockIdx.y * gridDim.x;
104         int ijp0 = (blockIdx.x + 1)%gridDim.x + blockIdx.y * gridDim.x;
105         int ijm0 = (blockIdx.x + gridDim.x - 1)%gridDim.x
106                                         + blockIdx.y * gridDim.x;
107         int ij0p = blockIdx.x + ((blockIdx.y + 1)%gridDim.y) * gridDim.x;
108         int ij0m = blockIdx.x + ((blockIdx.y + gridDim.y - 1)%gridDim.y)
109                                         * gridDim.x;
110         du[ij00] = (u[ijp0]-2.*u[ij00]+u[ijm0])/ *dx/ *dx
111                         + (u[ij0p]-2.*u[ij00]+u[ij0m])/ *dy/ *dy;
112     }
113 }
```

On Line 29, the two-dimensional array is allocated in the manner of a one-dimensional array. Unfortunately, the CUDA compiler does not allow for the two-dimensional contiguous array allocation as has been used elsewhere in this text. To compensate for this, the two-dimensional indices are adjusted to access a one-dimensional array. This is accomplished on Line 52, where the translation from  $(i, j)$  coordinates to linear coordinates is  $i+j*M$ , where  $M$  is the length of each row in the data set.

On Line 62, the dimensions of the block array are specified, and then the array of blocks is requested for the call to the kernel function on Line 65.

The kernel function is similar to the one-dimensional version, where the data is assumed to be periodic in both the  $x$ - and  $y$ -directions. Since this is in two dimensions, both `blockIdx.x` and `blockIdx.y` are used to determine the array index values. The dimensions of the grid array requested for the kernel are kept in another CUDA-supplied struct `gridDim`. The way this program is set up, it sets the grid dimensions to be the same as the mesh dimensions. Using blocks alone is not a typical arrangement because it does not provide much improved efficiency over a CPU implementation. To get more efficiency requires using threads.

### 23.3 • Threads

Each block on the device is further subdivided into threads, where the threads are able to share a small amount of memory per block. The thread count is specified by the second number in the angle bracket notation for executing kernel functions. Therefore, to execute Example 23.1 using  $N$  threads in one block, Line 75 is changed to

```
75    diff <<<1, N >>>(dev_u, dev_N, dev_dx, dev_du);
```

The indexing for blocks used the CUDA built-in struct `blockIdx` to give the particular block index for which the kernel is executing, and the indexing for threads is very similar, using `threadIdx` to give the thread index. Thus, to change Example 23.1 to use one block with multiple threads, Line 24 is also changed to

```
24    int tid = threadIdx.x;
```

Threads can also be arranged in two and three dimensions using the same CUDA type `dim3`. For the Tesla K20c, the maximum thread dimensions are (1024, 1024, 64). The built-in struct (`blockDim.x`, `blockDim.y`, `blockDim.z`) gives the thread dimension information, equivalent to the variable `gridDim` for block dimensions.

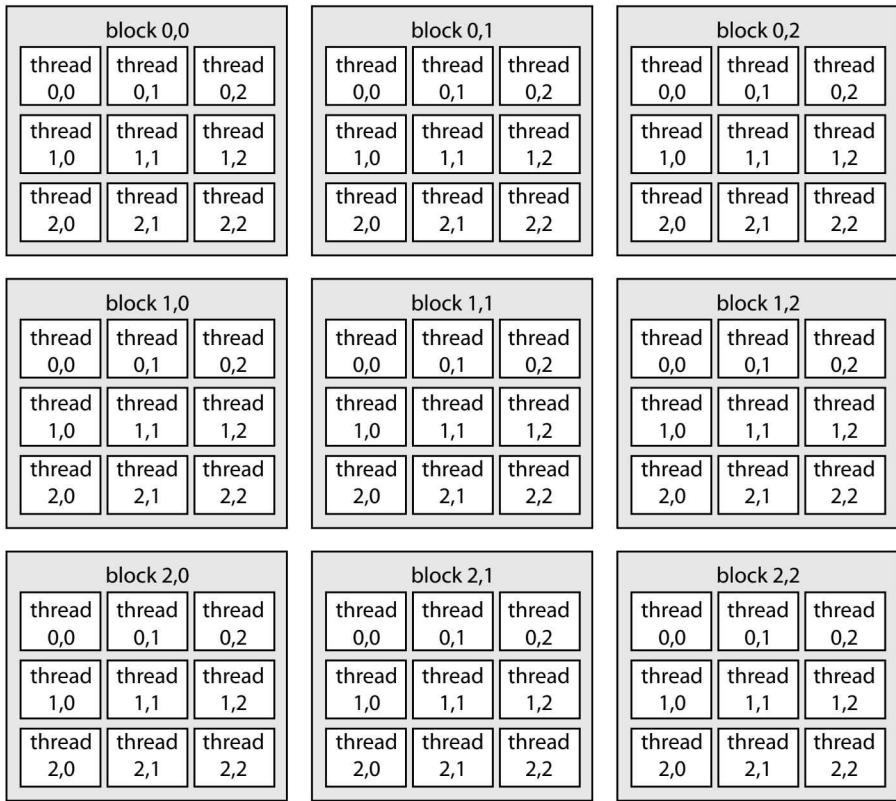
Putting it together, the picture for organizing computations is illustrated in Figure 23.1, where the task is broken down into groups of blocks, and each block is further subdivided into threads working in tandem.

### 23.4 • Error Handling

In an effort to keep the discussion as simple as possible without undue distractions, one of the topics skipped over so far is error detection. The code will be slightly faster if these matters are ignored, but on the other hand, there's a risk that an error would go undetected and negatively impact the final computed results. To build solid code, the error codes should be checked and dealt with appropriately. The low-level nature of coding a GPU makes it even more important to pay close attention to this detail. Fortunately, CUDA provides an easy mechanism for checking the error status for most CUDA calls from the host computer. Every CUDA function (easily identified by the leading “cuda” in the function name) returns a value of type `cudaError_t`. If it returns `cudaSuccess`, then all is good. Otherwise, the value can be passed to the function `cudaGetErrorString` to get a string message for what the error is.

In sample code of the CUDA installation is a function called `checkCudaErrors` that can be used to check for errors. Whenever a CUDA function is called, simply wrap it inside a function call to `checkCudaErrors`:

```
checkCudaErrors( cudaMalloc((void**) &dev_u, N * sizeof(double)) );
```



**Figure 23.1.** Diagram of the organization of threads within blocks with both in arrays of different dimensions. The dimensions of the thread arrays need not be the same as the blocks.

If, for example, an error occurs with the call to `cudaMalloc`, the line number, the specific error message, and the line of code itself will be printed to the terminal before the program is terminated. The function is defined in a header file called `helper_cuda.h`, which is not in the usual CUDA include directory. Thus, to use `checkCudaErrors` add this extra include path when compiling:

-I/usr/local/cuda/samples/common/inc

where it's assumed that CUDA is installed in `/usr/local`. Adjust this path accordingly if your installation is in a different directory.

To write your own error handler that can be turned on and off, create a header file for your CUDA-specific files, and include the following code into that header file:

### Example 23.3.

```

9 }
10 #define CheckError( err ) (CHECK_ERROR(err , __FILE__ , __LINE__))
11
12 #else
13
14 #define CheckError( err ) err
15
16 #endif

```

To use this code, whenever a CUDA function is called, simply wrap it inside a function call to `CheckError` in the same manner described for `checkCudaErrors`.

Since the C preprocessor has not been presented in any great detail, this usage will require some discussion. The first preprocessor directive in this example is `#ifdef DO_ERROR_CHECKING`, which tells the compiler that if the macro `DO_ERROR_CHECKING` is defined as a macro, then compile Lines 2–11. If the macro is *not* defined, then compile the code after the `#else` clause, which would be Lines 13–15, where the conditional compiling is terminated with `#endif`. Thus, to turn on error checking, either insert the macro definition `#define DO_ERROR_CHECKING` at the top of the file or use the compiler flag `-D` to define the macro when compiling like this:

```
$ nvcc -o myprog myprog.cu -DDO_ERROR_CHECKING
```

Returning to Line 10, the “function” `CheckError` is in fact not a function at all, but simply a macro that can also take an argument. If `DO_ERROR_CHECKING` is defined, then when the preprocessor sees the text “`CheckError(X)`”, it will replace it with “`(CHECK_ERROR(X, __FILE__, __LINE__))`”. The preprocessor also identifies the strings “`__FILE__`” and “`__LINE__`” and will substitute the name of the current file for `__FILE__` and the line number within that file in place of `__LINE__`. Therefore, when the preprocessor is done, it will replace

```
1 CheckError( cudaMalloc((void**)&dev_u , N*sizeof(double)) );
```

with

```
1 CHECK_ERROR( cudaMalloc((void**)&dev_u , N*sizeof(double)),
2 "filename.cu" , 17 );
```

Now, `CHECK_ERROR` is a regular function, and if the value of the first argument is not `cudaSuccess`, then it prints an error message using `cudaGetErrorString` and exits the program. Which is at a minimum what an error handler should do.

Example 24.3 shows how to use the `cudaCheckErrors` function in practice, but for space considerations, the remaining examples will not. The best practice is to use error checking for a more robust code.

## Exercises

- 23.1. Write a kernel that fills out a two-dimensional array of integers of dimension  $100 \times 100$ . The kernel should save the value of the function

```
value = 100*(100*(blockIdx.x+blockIdx.y)+threadIdx.x) +
        threadIdx.y;
```

Experiment with different size numbers of threads per block and verify the values returned are as expected.

- 23.2. Modify Example 23.1 to add all the error checking as described in Section 23.4.  
Try modifying the code to generate errors to verify the error checking is working.  
For example, try commenting out Line 62 and see what happens.

# Chapter 24

# GPU Memory

Up to this point, the memory used in the examples has all been global memory on the GPU, which is the slowest kind of memory. There is higher speed memory that can significantly boost performance. Choices include using shared memory, constant memory, and texture memory. Shared memory is the easiest to use since it behaves the most like global memory, but it still requires some thought to use it effectively. Constant memory is fast but not as plentiful and limited in its uses, and texture memory is also fast but requires extra steps to use for double precision. In this chapter, the various memory types and how to use them will be discussed.

## 24.1 • Shared Memory

Simply switching Example 23.1 from a list of blocks to a single block with a list of threads is not always sufficient to produce performance improvements. The problem is that the bottleneck in the problem is not the number of cores being applied to the task, or how those cores are organized, but actually in the time required to retrieve the necessary data from memory and to store the result back into memory. So far all examples are using global memory throughout. A latency problem can arise if multiple threads or blocks make requests from the same global memory simultaneously because they are effectively forced to queue up sequentially resulting in delays.

Shared memory is a way to have threads cooperate when they require the same global memory. Each block has an allocation of shared memory assigned to it for which read/write times are significantly shorter than read/write to global memory. Shared memory has limited capacity compared to global memory so it must be used wisely as well. See Table 22.1 for a comparison of memory capacities for the Tesla K20c and Quadro K600.

Shared memory in a kernel is identified by the attribute label `__shared__` in front of the variable declaration. The amount of data used in shared memory can be determined either statically or dynamically. For static allocation, the variable would be declared in the kernel with a fixed length:

```
__shared__ float localmem [256];
```

where this creates an array of 256 floating point numbers. To create the array dynamically, the variable is declared in the kernel very similarly except the number in the

square brackets is omitted and the variable is declared as `extern`:

```
extern __shared__ float localmem [];
```

When the kernel is called, the amount of shared memory per block is provided in an optional third argument in the angle brackets:

```
kernel <<<numBlocks, numThreads, 256*sizeof(float)>>>( ... );
```

The whole purpose of shared memory is to take advantage of faster memory access, and to do this, the transfer of data to/from global memory will be done in parallel as well. Consider the following finite difference kernel `diff` that is a modified version of the kernel in Example 23.1:

### Example 24.1.

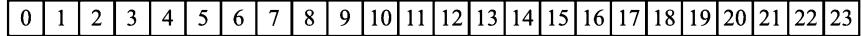
```

1 --global__ void diff(double* u, int* N, double* dx, double* du) {
2
3     // Shared memory declared statically
4     --shared__ double localu[threadsPerBlock+2];
5     --shared__ double localdu[threadsPerBlock];
6
7     // Set up global and shared memory indices
8     int g_i = (threadIdx.x + blockIdx.x * blockDim.x) % *N;
9     int l_i = threadIdx.x + 1;
10    int g_im = (g_i + *N - 1) % *N;
11    int g_ip = (g_i + 1) % *N;
12
13    // Transfer global memory to shared memory
14    localu[l_i] = u[g_i];
15    if (threadIdx.x == 0)
16        localu[0] = u[g_im];
17    if (threadIdx.x == threadsPerBlock - 1)
18        localu[l_i+1] = u[g_ip];
19
20    // Compute the finite difference for this thread and store
21    // in shared memory
22    localdu[threadIdx.x] = (localu[threadIdx.x+2] - localu[threadIdx.x])
23                                /*dx*/ / 2.;
24
25    // Transfer the results from shared memory to global memory
26    du[g_i] = localdu[threadIdx.x];
27 }
```

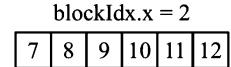
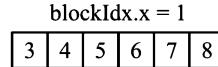
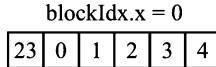
In this example, suppose there are six blocks each with four threads; then the local and global memory layout for a data array of length 24 is illustrated in Figure 24.1. The logic in this kernel can be broken down into three stages: (1) transfer a piece of global memory to local shared memory, (2) do the local computation for this thread, (3) move the result from local memory to global memory. In Example 24.1, the local and global indices into the data array are computed beginning on Line 8.

Stage 1 of the kernel, where the global memory is copied to the shared memory is done on Lines 14–18. Here, each thread copies its corresponding global memory location to the local shared memory address on Line 14. However, the finite difference approximation uses a three-point stencil, so an additional data point on each end is also needed. Therefore, the first and last threads in the block additionally copy the

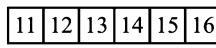
global u, du



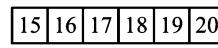
local u



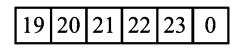
blockIdx.x = 3



blockIdx.x = 4

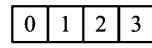


blockIdx.x = 5

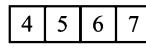


local du

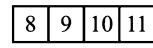
blockIdx.x = 0



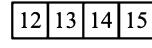
blockIdx.x = 1



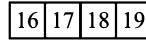
blockIdx.x = 2



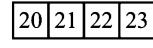
blockIdx.x = 3



blockIdx.x = 4



blockIdx.x = 5



**Figure 24.1.** Illustration of the memory layout for shared memory in Example 24.1.

neighboring value from the global data so that the finite difference calculation within any thread is confined entirely to shared memory. For example, using Figure 24.1, thread 0 of block 1 will copy both global data points  $u[4]$  and  $u[3]$  and put them in local data with indices 1 and 0, respectively.

For stage 2, the finite difference calculation is done entirely on local shared memory on Line 22.

Finally, stage 3 is done on Line 26, where the locally stored solution is copied back to global memory so it can be accessed by the host or by subsequent kernel functions. In this direction, the overlap points are not computed, so the extra copies set up in stage 1 are not used here.

Unfortunately, the kernel in Example 24.1 will not necessarily work correctly. Suppose thread 1 starts to do its finite difference computation before thread 0 finishes doing its copy from global to local; then the finite difference computed in thread 1 may be using data from the local memory that hasn't been initialized yet and hence have random data in it. In order to guarantee that the local shared memory is ready before computing the finite difference, the threads have to certify that they've all completed their global to local memory copy. This is accomplished by syncing the threads using the `__syncthreads` function. The `__syncthreads` command is similar in spirit to the `MPI_Wait` command from MPI, where all threads of the given block will stop and wait at the synchronization point until all threads reach that point.

The `__syncthreads` function must be used carefully. For example, consider the following use:

```

1  if (threadIdx.x < blockDim.x/2) {
2      // do some special work here
3      __syncthreads();
4 }
```

Here `__syncthreads` is inside an `if` statement where only some of the threads reach the synchronization point, and there is no other synchronization point for the other threads. The program will lock up waiting for the threads that can never reach the synchronization point. To fix this, there must be a corresponding `__syncthreads` for the rest of the threads:

```

1  if (threadIdx.x < blockDim.x/2) {
2      // do some special work here
3      __syncthreads();
4  } else {
5      __syncthreads();
6 }
```

The kernel in Example 24.1 is corrected in Example 24.2.

### Example 24.2.

```

1 __global__ void diff(double* u, int* N, double* dx, double* du) {
2
3     // Shared memory declared statically
4     __shared__ double localu[threadsPerBlock+2];
5     __shared__ double localdu[threadsPerBlock];
6
7     // Set up global and shared memory indices
8     int g_i = (threadIdx.x + blockIdx.x * blockDim.x) % *N;
9     int l_i = threadIdx.x + 1;
10    int g_im = (g_i + *N - 1) % *N;
11    int g_ip = (g_i + 1) % *N;
12
13    // Transfer global memory to shared memory
14    localu[l_i] = u[g_i];
15    if (threadIdx.x == 0)
16        localu[0] = u[g_im];
17    if (threadIdx.x == threadsPerBlock - 1)
18        localu[l_i+1] = u[g_ip];
19
20    __syncthreads();
21
22    // Compute the finite difference for this thread and store
23    // in shared memory
24    localdu[threadIdx.x] = (localu[threadIdx.x+2]-localu[threadIdx.x])
25                                /* dx */ / 2. ;
26
27    // Transfer the results from shared memory to global memory
28    du[g_i] = localdu[threadIdx.x];
29 }
```

## 24.1.1 • Warps

A final note about shared memory is that it is retrieved synchronously in groups of requests called a half-warp, and the standard warp size is 32, so that means that memory access is done in groups of 16 at a time. The warp access works best if sequential portions of the data are accessed as opposed to scattered data. This group memory access is called coalesced memory access.

To illustrate how warps can affect performance, consider a large  $N \times N$  array, stored in row-major order. The following kernel would add one to each entry in a specified row of the matrix:

```
__global__ void addone(double* u, int row)
{
    int i = gridDim.x*blockDim.x*row + threadIdx.x + blockDim.x*blockIdx.x;
    u[i] = u[i] + 1.;
```

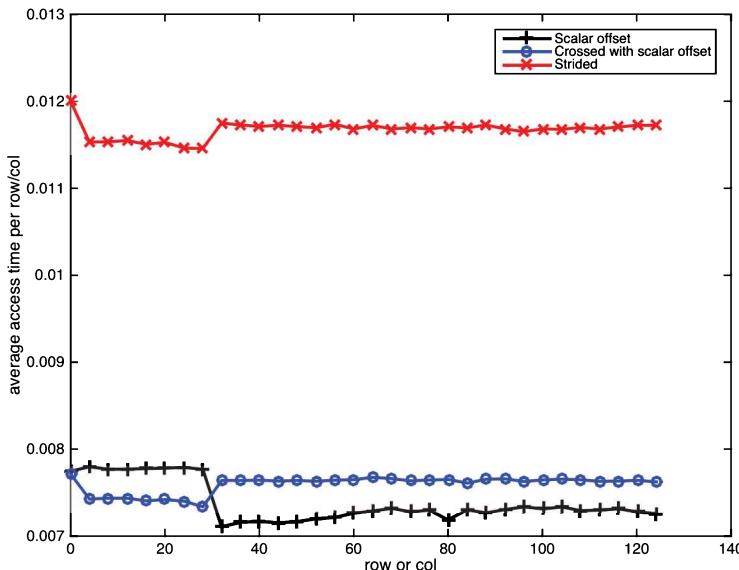
On the other hand, to access the rows for a fixed column, the data is strided, i.e., the index  $j$  would be given by

```
int j = gridDim.x*blockDim.x*(threadIdx.x + blockDim.x*blockIdx.x)
      + column;
```

What is important is how large the stride is. For the fixed row, the stride is one, while for the fixed column, the stride is  $N$ . To show that it's only memory address proximity, consider a third case where the index  $i$  exchanges even columns with odd:

```
int i = gridDim.x*blockDim.x*row + threadIdx.x + blockDim.x*blockIdx.x
      + (threadIdx.x%2 ? -1 : 1);
```

The results of simply loading data using these indices are illustrated in Figure 24.2. Thus, if using  $N$  blocks with  $N$  threads to access a two-dimensional matrix, the blocks should correspond to rows, and the threads correspond to columns whenever possible to improve performance.



**Figure 24.2.** Comparison of efficiency of memory access with and without coalesced memory access. An entire row or column of a  $2048 \times 2048$  matrix is accessed. Scalar offset corresponds to accessing a fixed row with a stride of one. Crossed with scalar offset has stride one but where odd indices access even addresses and vice versa. Strided accesses a fixed column so that  $\text{threadIdx.x}$  is multiplied by  $N$ . When stride is one, memory access is coalesced, resulting in 50% faster memory access.

## 24.2 • Constant Memory

Constant memory is another form of fast-access, low-latency memory available on the GPU. Table 22.1 shows that a Tesla K20c has about 65Kb of constant memory. This memory space is useful for values that are common within a block. For example, in all the sample code so far in this section, the length of the array and the space step size have been passed as arguments to the kernel function. In reality, they should be treated as constants within the kernel.

Of course, constants can be defined statically at compile time, such as the constant `threadsPerBlock`, but for the finite difference operation, the number of nodes in the array is not determined until runtime. Example 24.3 shows how the variables `dev_N` and `dev_dx` are set up as constant memory and used within the kernel function.

### Example 24.3.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include "checkerror.h"
4
5 #ifndef M_PI
6 #define M_PI 3.1415926535897932384626433832795
7 #endif
8
9 // set the threads per block as a constant multiple of the warp size
10 const int threadsPerBlock = 256;
11
12 // declare constant memory on the device
13 __device__ static int dev_N;
14 __device__ static double dev_dx;
15
16 // declare the kernel function
17 __global__ void diff(double* u, double* du);
18
19 /*
20  int main(int argc, char* argv[])
21  Demonstrate a simple example for implementing a
22  parallel finite difference operator
23
24  Inputs: argc should be 2
25  argv[1]: Length of the vector of data
26
27  Outputs: the initial data and its derivative.
28 */
29 int main(int argc, char* argv[]) {
30
31 // read in the number of grid points
32 int N = atoi(argv[1]);
33
34 // determine how many blocks are needed for the whole grid
35 const int blocksPerGrid = N/threadsPerBlock
36                                     + (N%threadsPerBlock > 0 ? 1 : 0);
37
38 // allocate host memory
39 double* u = (double*)malloc(N*sizeof(double));
40 double* du = (double*)malloc(N*sizeof(double));
41
42 double* dev_u;
43 double* dev_du;
44
45 // allocate device memory

```

```
46  CheckError( cudaMalloc((void**)&dev_u , N*sizeof(double)) );
47  CheckError( cudaMalloc((void**)&dev_du , N*sizeof(double)) );
48
49 // initialize the data on the host
50 double dx = 2*M_PI/N;
51 for (int i=0; i<N; ++i)
52     u[i] = sin(i*dx);
53
54 // cudaMemcpyToSymbol copies data into the constant memory space
55 // on the GPU
56 CheckError( cudaMemcpyToSymbol(dev_N, &N, sizeof(int)) );
57 CheckError( cudaMemcpyToSymbol(dev_dx, &dx, sizeof(double)) );
58 CheckError( cudaMemcpy(dev_u, u, N*sizeof(double),
59                         cudaMemcpyHostToDevice) );
60
61 // The kernel call no longer needs to send dev_N or dev_dx
62 // to the kernel
63 diff<<<blocksPerGrid, threadsPerBlock>>>(dev_u, dev_du);
64
65 // Copy the results back to the host
66 CheckError( cudaMemcpy(du, dev_du, N*sizeof(double),
67                         cudaMemcpyDeviceToHost) );
68
69 // Notice that we don't need to free the dev_N and dev_dx pointers
70 // anymore.
71 CheckError( cudaFree(dev_u) );
72 CheckError( cudaFree(dev_du) );
73 free(u);
74 free(du);
75 return 0;
76 }
77
78 // The constant variables are not passed through
79 // the kernel argument list.
80 __global__ void diff(double* u, double* du) {
81
82     __shared__ double localu[threadsPerBlock+2];
83     __shared__ double localdu[threadsPerBlock];
84
85     // copy data from global to local memory
86     // dev_N is now in constant memory
87     int g_i = (threadIdx.x + blockIdx.x * blockDim.x) % dev_N;
88     int l_i = threadIdx.x + 1;
89     int g_im = (g_i + dev_N - 1) % dev_N;
90     int g_ip = (g_i + 1) % dev_N;
91     localu[l_i] = u[g_i];
92     if (threadIdx.x == 0)
93         localu[0] = u[g_im];
94     if (threadIdx.x == threadsPerBlock - 1)
95         localu[l_i+1] = u[g_ip];
96
97     __syncthreads();
98
99     // compute the finite difference
100    localdu[threadIdx.x] = (localu[threadIdx.x+2]-localu[threadIdx.x])
101                                / dev_dx / 2. ;
102
103    // save the results back into global memory
104    du[g_i] = localdu[threadIdx.x];
105 }
```

The first difference is that the variables `dev_N` and `dev_dx` are no longer declared inside the main program but instead are declared in global scope on Lines 13–14. On those lines, the variables are identified as `__device__ static`, but they are not given constant values yet. When the values of the constants are determined at runtime, the data for the number of nodes and the space step size are loaded onto the device using `cudaMemcpyToSymbol` on Lines 56–57. Instead of loading the data into global memory, it gets loaded into constant memory. Consequently, it does not need to be added to the argument list; hence when the kernel is called, on Line 63, the `dev_N` and `dev_dx` variables are no longer in the argument list. Finally, the variables can be read within the kernel as normal, such as on Line 87.

## 24.3 • Texture Memory

Texture memory is yet another way to store data on the GPU that can provide a boost to performance. Texture memory is different from the other types seen so far in that texture memory has the ability to provide interpolations between data points as well as the data itself. For example, data can be queried between integral grid points by providing a floating point index instead of an integer. Texture memory is not able to provide double precision storage like the other types directly, but there is a way around that. Finally, texture memory is read-only, so it can only be used for input data, not for the output, which limits its applicability.

To begin, suppose for simplicity the data is single precision floating point rather than double precision. Texture data must be declared in the global scope, meaning it's declared outside the main program similar to how constant memory is declared, but it is declared in a different way:

```
texture<float, cudaTextureType1D, cudaMemcpyKind> tex_u;
```

The texture type with angle brackets is a CUDA construct. The first argument in the angle brackets is the data type to be stored in the texture memory. Texture data can be stored in multiple dimensions and multiple layers, but since the present task is to do a one-dimensional finite difference operation, a simple one-dimensional arrangement is used.

Inside the main program, the texture memory is allocated using `cudaMalloc`, but an additional function call is required to bind the allocated memory to texture memory as opposed to global memory:

```
cudaMalloc((void**)&dev_u, N * sizeof(float));
cudaBindTexture(NULL, tex_u, dev_u, N * sizeof(float));
```

Since the texture memory is declared globally, like a constant, it does not get passed through the argument list to the kernel function like it did before, so it is taken out of the argument list.

Texture memory is also accessed differently inside the kernel function. A special fetch function must be used to retrieve data from the texture. To retrieve the data at global index `i`, the data is retrieved by

```
value = tex1Dfetch(tex_u, i);
```

Finally, there is an additional clean-up step to unbind the memory before it is freed:

```
cudaUnbindTexture(tex_u);
```

Example 24.4 gives the new version of the finite difference program using texture memory.

#### Example 24.4.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <cuda.h>
4 #include <cuda_runtime.h>
5
6 #ifndef M_PI
7 #define M_PI 3.1415926535897932384626433832795
8 #endif
9
10 const int threadsPerBlock = 256;
11
12 // declare constant memory on the device
13 __device__ static int dev_N;
14 __device__ static float dev_dx;
15 // Declare the texture memory in global scope like the constant memory
16 texture<float, cudaTextureType1D, cudaReadModeElementType> tex_u;
17
18 // Note that the kernel function now only has the output for an argument
19 __global__ void diff(float* du);
20
21 /*
22  int main(int argc, char* argv[])
23  Demonstrate a simple example for implementing a
24  parallel finite difference operator
25
26  Inputs: argc should be 2
27  argv[1]: Length of the vector of data
28
29  Outputs: the initial data and its derivative.
30 */
31
32 int main(int argc, char* argv[]) {
33
34  // read in the number of grid points
35  int N = atoi(argv[1]);
36
37  // determine how many blocks are needed for the whole grid
38  const int blocksPerGrid = N/threadsPerBlock
39  + (N%threadsPerBlock > 0 ? 1 : 0);
40
41  // allocate host memory
42  float* u = (float*)malloc(N*sizeof(float));
43  float* du = (float*)malloc(N*sizeof(float));
44
45  float* dev_u;
46  float* dev_du;
47
48  // allocate device memory
49  cudaMalloc((void**)&dev_u, N*sizeof(float));
50  cudaMalloc((void**)&dev_du, N*sizeof(float));
51
52  // The device memory is bound to the texture memory here.
53  cudaBindTexture(NULL, tex_u, dev_u, N*sizeof(float));
54
55  // initialize the data on the host
56  float dx = 2*M_PI/N;
57  for (int i=0; i<N; ++i)
58    u[i] = sin(i*dx);

```

```

59
60 // set the values of N and dx in constant memory
61 cudaMemcpyToSymbol(dev_N, &N, sizeof(int));
62 cudaMemcpyToSymbol(dev_dx, &dx, sizeof(float));
63
64 // copy the input data to the device
65 cudaMemcpy(dev_u, u, N*sizeof(float), cudaMemcpyHostToDevice);
66
67 // call the kernel
68 diff<<<blocksPerGrid , threadsPerBlock>>>(dev_du);
69
70 // Copy the results back to the host
71 cudaMemcpy(du, dev_du, N*sizeof(float), cudaMemcpyDeviceToHost);
72
73 // The clean-up phase also requires the texture memory to be unbound.
74 cudaUnbindTexture(tex_u);
75 cudaFree(dev_u);
76 cudaFree(dev_du);
77 free(u);
78 free(du);
79 return 0;
80 }
81
82 __global__ void diff(float* du) {
83
84 int g_i = (threadIdx.x + blockIdx.x * blockDim.x) % dev_N;
85 int g_im = (g_i + dev_N - 1) % dev_N;
86 int g_ip = (g_i + 1) % dev_N;
87
88 // Texture memory is accessed through specialized fetch functions
89 // such as tex1Dfetch()
90 du[g_i] = (tex1Dfetch(tex_u, g_ip) - tex1Dfetch(tex_u, g_im)) / dev_dx / 2. ;
91 }
```

### 24.3.1 • Texture Memory and Double Precision

Example 24.4 only used single precision because texture memory is not designed to directly handle double precision data. However, texture memory can be used for double precision data; it just requires a little extra effort to make it work. Double precision values require 8 bytes, or twice what an `int` or `float` requires. The CUDA language provides a number of other numeric types for various purposes, and one of those types is the `int2` type. It's a struct with integer members `x` and `y` and hence has the same storage size of 8 bytes. Therefore, to make the double precision version of the texture memory finite difference kernel, the first step is to change the type of the texture memory from `float` to `int2` on Line 16:

```
17 texture<int2, cudaTextureType1D, cudaReadModeElementType> tex_u;
```

All other floats inside the main program are converted to `double` like the previous examples. The kernel function does require a little extra work, and it's shown here:

```

1 __global__ void diff(double* du) {
2
3     int g_i = (threadIdx.x + blockIdx.x * blockDim.x) % dev_N;
4     int g_im = (g_i + dev_N - 1) % dev_N;
5     int g_ip = (g_i + 1) % dev_N;
6
7     // Fetch the int2 data from texture memory
```

```

8 int2 up_int2 = tex1Dfetch(tex_u, g_ip);
9 int2 um_int2 = tex1Dfetch(tex_u, g_im);
10
11 // Convert the int2 data back into doubles.
12 double up = __hiloint2double(up_int2.y, up_int2.x);
13 double um = __hiloint2double(um_int2.y, um_int2.x);
14
15 du[g_i] = (up-um)/dev_dx/2.;
16 }

```

When retrieving data from texture memory, it's assumed the texture memory is still in `int2` format, so the data must be retrieved as if it were an `int2`, and then converted back into double. The CUDA environment provides a helper function to accomplish the task. On Lines 8–9 the data is retrieved, but it's converted back into doubles on Lines 12–13. The order in which double precision numbers are stored is slightly different, so the two components of the `int2` type are fed in reverse order to the function that converts the values back into double on Lines 12–13. The full double precision version is shown below in Example 24.5.

### Example 24.5.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <cuda.h>
4 #include <cuda_runtime.h>
5
6 #ifndef M_PI
7 #define M_PI 3.1415926535897932384626433832795
8 #endif
9
10 const int threadsPerBlock = 256;
11
12 // declare constant memory on the device
13 __device__ static int dev_N;
14 __device__ static double dev_dx;
15 // Declare the texture memory using int2 type
16 texture<int2, cudaTextureType1D, cudaReadModeElementType> tex_u;
17
18 // Declare the kernel function
19 __global__ void diff(double* du);
20
21 /*
22  int main(int argc, char* argv[])
23  Demonstrate a simple example for implementing a
24  parallel finite difference operator
25
26  Inputs: argc should be 2
27  argv[1]: Length of the vector of data
28
29  Outputs: the initial data and its derivative.
30 */
31
32 int main(int argc, char* argv[]) {
33
34  // read in the number of grid points
35  int N = atoi(argv[1]);
36
37  // determine how many blocks are needed for the whole grid
38  const int blocksPerGrid = N/threadsPerBlock
39  + (N%threadsPerBlock > 0 ? 1 : 0);
40

```

```
41 // allocate host memory
42 double* u = (double*)malloc(N*sizeof(double));
43 double* du = (double*)malloc(N*sizeof(double));
44
45 double* dev_u;
46 double* dev_du;
47
48 // allocate device memory
49 cudaMalloc((void**)&dev_u, N*sizeof(double));
50 cudaMalloc((void**)&dev_du, N*sizeof(double));
51
52 // initialize the data on the host
53 double dx = 2*M_PI/N;
54 for (int i=0; i<N; ++i)
55     u[i] = sin(i*dx);
56
57 // set the values of N and dx in constant memory
58 cudaMemcpyToSymbol(dev_N, &N, sizeof(int));
59 cudaMemcpyToSymbol(dev_dx, &dx, sizeof(double));
60
61 // copy the input data to the device
62 cudaMemcpy(dev_u, u, N*sizeof(double), cudaMemcpyHostToDevice);
63
64 // The device memory is bound to the texture memory here.
65 cudaBindTexture(NULL, tex_u, dev_u, N*sizeof(double));
66
67 // call the kernel
68 diff<<<blocksPerGrid, threadsPerBlock>>>(dev_du);
69
70 // Copy the results back to the host
71 cudaMemcpy(du, dev_du, N*sizeof(double), cudaMemcpyDeviceToHost);
72
73 // The clean-up phase also requires the texture memory to be unbound.
74 cudaUnbindTexture(tex_u);
75 cudaFree(dev_u);
76 cudaFree(dev_du);
77 free(u);
78 free(du);
79 return 0;
80 }
81
82 __global__ void diff(double* du) {
83
84     int g_i = (threadIdx.x + blockIdx.x * blockDim.x) % dev_N;
85     int g_im = (g_i + dev_N - 1) % dev_N;
86     int g_ip = (g_i + 1) % dev_N;
87
88     // Fetch the data masquerading as int2
89     int2 up_int2 = tex1Dfetch(tex_u, g_ip);
90     int2 um_int2 = tex1Dfetch(tex_u, g_im);
91
92     // Convert the int2 data into double
93     double up = __hiloint2double(up_int2.y, up_int2.x);
94     double um = __hiloint2double(um_int2.y, um_int2.x);
95
96     du[g_i] = (up-um)/dev_dx/2.;
97 }
```

## 24.4 • Warp Shuffles

Shared memory is high speed, but it is also limited in terms of capacity, so sometimes it can be advantageous to look for alternative ways to communicate between threads. At the same time, the process of having each thread loading a piece of the shared memory from global memory, synchronizing the threads, and then subsequently reading from shared memory for a local calculation within a given thread seems unnecessarily burdensome. Warp shuffles are a means of streamlining the interthread communication by directly sampling variables from other threads within the same warp.

Up to this point, the organization of calculations has been in terms of blocks and threads. Threads within a block are implicitly grouped into intermediate collections called warps. According to Table 22.1, the warp size is 32, which means that if a block contains 512 threads, then those threads are organized into 16 warps of 32 threads each. Thus, warps present an additional level of structure inside blocks. The index of a given thread within a warp, also called the lane, is given by `threadIdx.x%32`.

Suppose that each thread within a warp retrieves a value of a variable into its non-shared local memory and stores it in a variable `u`. If the thread with lane `j` wants to read the value of the variable `u` in lane `j+1` and store it in a new variable `uplus`, then it can be accomplished via a shuffle command like this:

```
uplus = __shfl_down_sync(0xFFFFFFFF, u, 1);
```

The first argument is a flag that indicates which of the 32 threads, also called lanes, in the warp will participate in the shuffle given as a hexadecimal number. This forces all threads to be synchronized before executing the shuffle. If, for example, only the first 8 threads will use the shuffle, then only the first 8 threads need to be synchronized, so the proper flag would be `0x000000FF`. For the simple examples in this text, the full mask `0xFFFFFFFF` is used throughout. For more complex algorithms, the reader should consult the CUDA documentation for setting the mask. The second argument is the variable to be shuffled, in this case the variable `u`. The third argument is how many lanes to shift, in this case one, because lane `j` wanted to look one lane higher.

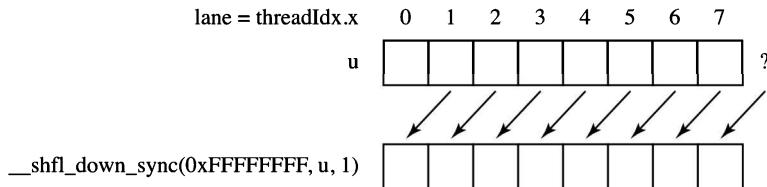
This operation is illustrated in Figure 24.3, where the boxes indicate the values of the variable `u` for each of the threads. The thread number corresponds to the value of `threadIdx.x` inside the kernel. Thus, if `threadIdx.x` is 3, for example, then the expression `__shfl_down_sync(0xFFFFFFFF, u, 1)` will contain the value of the variable `u` from the thread with `threadIdx.x` equal to 4. Note that in this operation, the value of the expression for thread 31, the last thread in a warp, is undefined because there is no thread with `threadIdx.x` equal to 32 within a single warp. The `__shfl_up_sync` function is the companion to the `__shfl_down_sync` with the arrows going the opposite direction, as illustrated in Figure 24.4.

The shuffle function `__shfl_xor_sync` effectively swaps the values with neighboring threads depending upon the value of the mask used. If the mask is a power of two, i.e.,  $2^p$  for some  $p$ , then  $p$  threads swap with their neighboring  $p$  threads. The cases for  $p = 0, 1$  are illustrated in Figure 24.5.

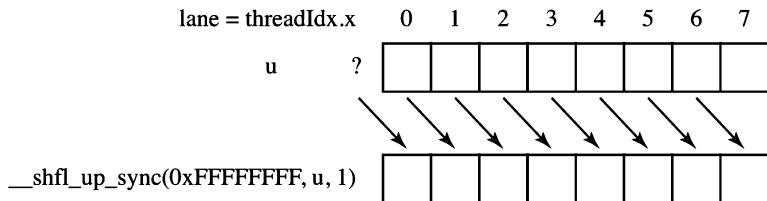
The function `__shfl_sync` simply does a direct copy of the value from the specified lane. Thus, if the value from thread 7 is desired, then use

```
__shfl_sync(0xFFFFFFFF, u, 7);
```

This is one way in which a value can be broadcast to all threads. There are other shuffles available; the ones covered here are the most useful in the context of numerical methods.



**Figure 24.3.** Illustration of using the shuffle down operation for sharing a variable named “u” with a neighboring thread. The arrows in the diagram indicate that the source data in the top row will appear in the shuffle command in the lane in the bottom row. Thus, when lane  $j$  calls `__shfl_down_sync(0xFFFFFFFF, u, 1)`, it will return the value contained in the variable  $u$  in lane  $j + 1$ . For lane  $j = 31$ , which is  $\text{warpSize} - 1$ , lane  $\text{warpSize}$  doesn’t exist, and so the value returned by the shuffle down command is undetermined as indicated by the question mark.



**Figure 24.4.** Illustration of using the shuffle up operation for sharing a variable named “u” with a neighboring thread. The arrows in the diagram indicate that the source data in the top row will appear in the shuffle command in the lane in the bottom row. Thus, when lane  $j$  calls `__shfl_up_sync(0xFFFFFFFF, u, 1)`, it will return the value contained in the variable  $u$  in lane  $j - 1$ . For lane  $j = 0$ , lane  $-1$  doesn’t exist and so the value returned by the shuffle up command is undetermined as indicated by the question mark.

Example 24.6 uses the shuffle down operation to do a reduction operation such as summing all the values in a warp. Figure 24.6 illustrates the strategy. At each step in the loop, the value of  $v$  in thread  $i$  is added to the value of  $v$  in thread  $i + j$ . When done, the sum of the 32 values will reside in  $v$  in the thread with index 0, so that value is stored into global memory before the kernel finishes.

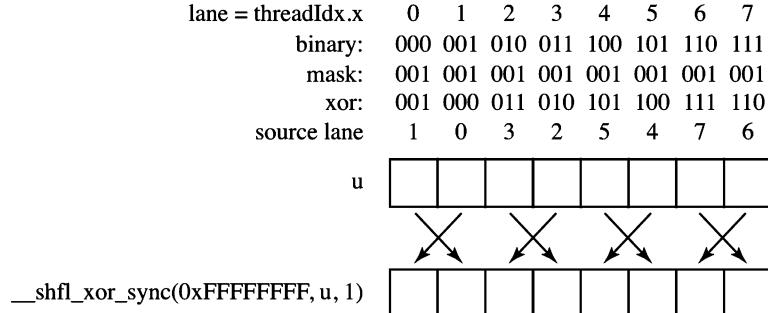
#### Example 24.6.

```

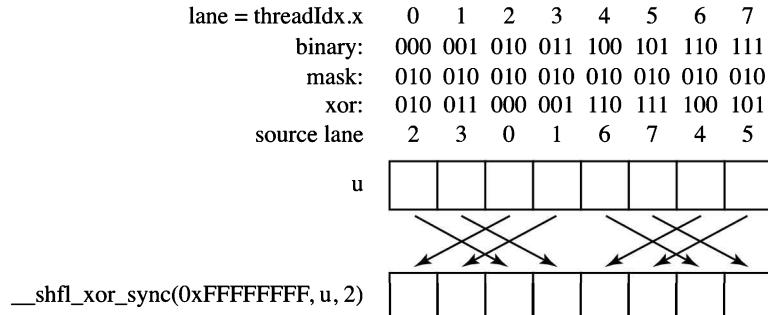
1  __global__ void warpsum(double* u, double* sum)
2 {
3     // Sum all values in a warp
4
5     double v = u[threadIdx.x];
6     int j;
7     for (j=16; j > 0; j/=2)
8         v += __shfl_down_sync(0xFFFFFFFF, v, j);
9     if (threadIdx.x == 0)
10        *sum = v;
11 }
```

To get the maximum value from among all the threads in a warp, the function `__shfl_xor_sync` can be used to combine all the comparisons. Example 24.7 and the corresponding Figure 24.7 illustrate how that would work.

(a)



(b)



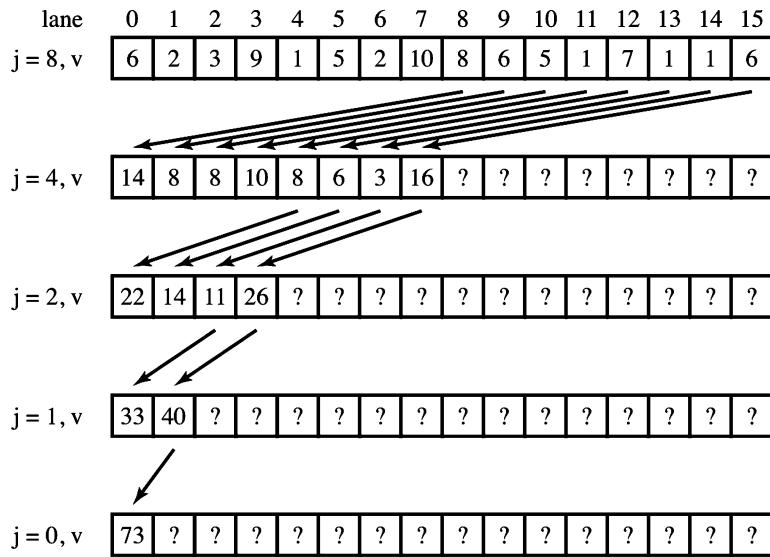
**Figure 24.5.** Illustration of using the shuffle exclusive-or operation for sharing a variable named “*u*” with a neighboring thread. In (a), the mask is 1, so each even numbered thread trades with the odd thread next to it. In (b), the mask is 2, which is binary 10, so the swaps occur pairwise. For the given lane, the source lane is the thread whose value will be sampled when calling the shuffle functions with the indicated mask values.

### Example 24.7.

```

1  __global__ void warpmax(double* u)
2  {
3      // replace all the values in u with the max(u)
4
5      double v = u[threadIdx.x];
6      double w;
7      int j;
8      for (j=1; j < warpSize; j*=2) {
9          w = __shfl_xor_sync(0xFFFFFFFF, v, j);
10         v = v > w ? v : w;
11     }
12     u[threadIdx.x] = v;
13 }
```

As a final example, Example 24.8 does a finite difference calculation similar to the other examples in the text. In this case, the `__shfl_up_sync` and `__shfl_down_sync` functions are used to compute the central finite difference operator on an array.



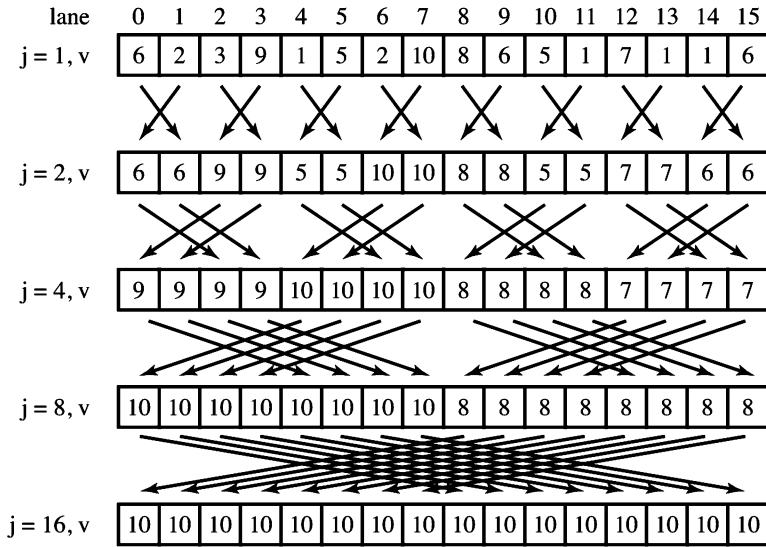
**Figure 24.6.** Illustration of using shuffle down to do a fast sum of the values in an array of length 32, i.e., the size of a warp. Each row corresponds to an iteration in the `for` loop in Example 24.6. Here, the step with  $j = 16$  is excluded for simplicity. Notice that each lane adds its own value to the value in the lane indicated by the arrow. Thus, when  $j = 8$ , then lane 5 adds the value of  $v$  in lane 5 with the value of  $v$  in lane 5 + 8 = 13 to get a sum of 6, and the value is saved in  $v$  in lane 5.

### Example 24.8.

```

1 #include <stdio.h>
2 #include <cuda.h>
3 #include <math.h>
4
5 #ifndef M_PI
6 #define M_PI 3.1415926535897932384626433832795
7 #endif
8
9 // Setting the block size to 30, 2 less than the warpSize
10 #define BSIZE 30
11
12 // declare constant memory on the device
13 __device__ static int dev_N;
14 __device__ static double dev_dx;
15 /*
16  * void diff(double* u, int N, double dx, double* du)
17  *
18  * Compute the central difference operator on periodic data
19  * using warp shuffles.
20  */
21
22 Inputs:
23   double* u: Function data, assumed periodic
24   int* N: pointer to the x length of the data array
25   double dx: the space step size
26
27 Outputs:
28   double* du: derivative of the u data
29 */

```



**Figure 24.7.** Illustration of using shuffle xor to do a fast maximum of the values in an array of length 32, i.e., the size of a warp. Each row corresponds to an iteration in the `for` loop in the kernel. Here, the step with  $j = 16$  is excluded for simplicity. Notice that each lane computes the max of its own value for  $v$  and the value of  $v$  in the lane indicated in the arrow. Thus, when  $j = 1$ , then lane 5 computes the maximum of the value of  $v$  in lane 5 and lane 4 and stores it back in  $v$  in lane 5. Lane 4 does the same comparison and stores the result in  $v$  in lane 4. This means the max in lanes 4 and 5 is stored in both lanes.

```

30 __global__ void diff(double* u, int N, double dx, double* du)
31 {
32     // Let j be the index in the global array
33     int j = threadIdx.x + blockIdx.x*BSIZE - 1;
34     int i = (j + N)%N;
35
36     // Each thread in the warp loads just one grid value
37     double myu = u[i];
38
39     // Compute the finite difference.
40     // Shuffle down is the point to the right, shuffle up is to the left
41     double mydu = __shfl_down_sync(0xFFFFFFFF, myu, 1)
42             - __shfl_up_sync(0xFFFFFFFF, myu, 1);
43
44     // Only store the finite differences in the interior of the warp,
45     // the end values are not valid
46     if (j>=0 && j<N && threadIdx.x > 0 && threadIdx.x < 31) {
47         du[j] = mydu/2./dx;
48     }
49 }
50 /*
51 int main(int argc, char* argv[])
52 Demonstrate a simple example for implementing a
53 parallel finite difference operator
54
55 Inputs: argc should be 2
56 argv[1]: Length of the vector of data
57
58 Outputs: the initial data and its derivative.
59

```

```
60 */
61 int main(int argc, char* argv[])
62 {
63     cudaDeviceProp prop;
64     int dev;
65     memset(&prop, 0, sizeof(cudaDeviceProp));
66     prop.multiProcessorCount = 13;
67     cudaChooseDevice(&dev, &prop);
68     cudaSetDevice(dev);
69
70     // read in the number of grid points
71     int N = atoi(argv[1]);
72
73     // determine how many blocks are needed for the whole grid
74     const int blocksPerGrid = (N+BSIZE-1)/BSIZE;
75
76     // allocate host memory
77     double* u = (double*)malloc(N*sizeof(double));
78     double* du = (double*)malloc(N*sizeof(double));
79
80     double* dev_u;
81     double* dev_du;
82
83     // allocate device memory
84     cudaMalloc((void**)&dev_u, N*sizeof(double));
85     cudaMalloc((void**)&dev_du, N*sizeof(double));
86
87     // initialize the data on the host
88     double dx = 2*M_PI/N;
89     int i;
90     for (i=0; i<N; ++i)
91         u[i] = sin(i*dx);
92
93     // set the values of N and dx in constant memory
94     cudaMemcpyToSymbol(dev_N, &N, sizeof(int));
95     cudaMemcpyToSymbol(dev_dx, &dx, sizeof(double));
96
97     // copy the input data to the device
98     cudaMemcpy(dev_u, u, N*sizeof(double), cudaMemcpyHostToDevice);
99
100    // call the kernel
101    diff<<<blocksPerGrid, 32>>>(dev_u, N, dx, dev_du);
102
103    // copy the results back to the host
104    cudaDeviceSynchronize();
105    cudaMemcpy(du, dev_du, N*sizeof(double), cudaMemcpyDeviceToHost);
106
107    // clean up
108    free(u);
109    free(du);
110    cudaFree(dev_u);
111    cudaFree(dev_du);
112
113    return 0;
114}
```

## Exercises

- 24.1. Try running the kernel `diff` from Example 24.1, which does not have the call to `__syncthreads`. Run the `racecheck` tool described in Section 22.3.3 and see if it catches the problem. Insert the `__syncthreads` and verify that it solves the problem.
- 24.2. Try writing a constant memory version of the `diff` kernel where the array `dev_u` is put in constant memory. Use `cudaGetDeviceProperties` or the function `cudaDeviceGetAttribute` to determine the maximum size of the array `dev_u` that will fit in constant memory.
- 24.3. Write a kernel that when combined with the `warpsum` kernel of Example 24.6 will compute the sum of an array of double precision values of any length. Write a corresponding loop without using CUDA; verify they get the same answer and compare the time required to compute the solution.
- 24.4. Write a kernel that when combined with the `warpmax` kernel of Example 24.7 will compute the maximum of an array of double precision values of any length. Write a corresponding loop without using CUDA; verify they get the same answer and compare the time required to compute the solution.
- 24.5. Write a new kernel `warpmax` that uses `__shfl_down_sync` to compute the maximum value within a warp.



# Chapter 25

# Streams

Modern GPU cards are not strictly required to perform a single task in unison, although that is the basis for much of the speed benefits gained from utilizing such cards. They are capable of handling more than one task depending on the amount of memory usage and type of tasks they are. The Tesla K20c is capable of up to 32 concurrent kernels.<sup>9</sup> Devices that return 1 for the `cudaDevAttrGpuOverlap` attribute are also able to do memory transfers at the same time as computing. Streams are required to take advantage of this capability. Streams have been used implicitly in all the examples so far, but just using one default stream.

One way to envisage streams is to think of a stream as a queue of CUDA kernel calls, memory transfers, and events, e.g., timing events, that will execute these tasks in sequence. Multiple streams can be active at any given time, sending their sequence of tasks to the device. The device handles the load balancing between the streams. This chapter focuses on how to set up streams, how to stack computation and device/host memory transfers to reduce latency, and how to measure the performance of kernels by inserting timing events into streams.

## 25.1 • Stream Creation and Destruction

A stream is created using the code

```
cudaStream_t stream;  
cudaStreamCreate( &stream );
```

and is disposed of in the clean-up phase at the end of the program using the function

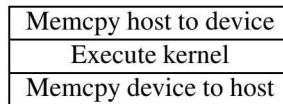
```
cudaStreamDestroy( stream );
```

Once the stream is created, tasks can begin to be assigned to it. For example, to assign the kernel `diff` to the stream named “`stream`”, it is inserted as the fourth argument inside the angle brackets:

```
diff<<< blocks , threads , shared_memory , stream>>>(dev_u , dev_du);
```

<sup>9</sup>Different compute capabilities allow for a different number of concurrent kernels. Compute capability 3.2 is limited to 4 concurrent kernels, and compute capability 7.0 allows for 128.

The examples presented so far will require some changes in order to take advantage of streams. Example 24.2 will be used as a starting point and will be modified to work with multiple streams. Recall that in order to execute a kernel the data it needs must first be copied from the host to the device, and then back again to retrieve the results. The process can be depicted graphically as follows:



The purpose of using multiple streams is to allow tasks to be done asynchronously. One place where that task requires extra care is in the memory transfers between the host and the device.

## 25.2 • Asynchronous Memory Copies

In prior examples, when memory was copied, say, from the host to the device, the memory being copied would be locked down in place temporarily so that the data could be transferred. However, even though from a programmer's point of view pointers seem to always point to the same location, there's actually a bit of misdirection going on under the hood, and memory that you think is there when you created the pointer may not actually be in that location at a later time unless you access it. This is a process called virtual memory, and it is a core principle that allows modern computer systems to handle many tasks seemingly simultaneously. However, if memory transfers are done asynchronously, then the memory must be pinned down so that it remains in place in the time between when the memory copy is requested and when the transfer is actually executed.

To make the memory transfer asynchronous, there are two changes required: the memory on the host must be allocated in a different way, and the function for copying the memory is different. Memory to be copied to/from the device asynchronously must be allocated using the function `cudaHostAlloc` instead of the usual `malloc`:

```
double* u;
cudaHostAlloc((void**)&u, N*sizeof(double), cudaHostAllocDefault);
```

This creates page-locked memory that will not move while it is active. When memory is locked, other programs or tasks on the same computer have less memory to use, so allocating large chunks of data this way is not recommended for general use and should be freed as soon as possible so that computer performance doesn't degrade. Freeing this new kind of allocated memory requires a corresponding free function:

```
cudaFreeHost(u);
```

The second change required to make the memory transfer asynchronous is to use a different copy function:

```
cudaMemcpyAsync(dev_u, u, N*sizeof(double), cudaMemcpyHostToDevice,
                stream);
```

This function looks very similar to the `cudaMemcpy` from before except that it takes an extra argument for the stream to which this task will be assigned, and it works asynchronously. That means that after calling this function, control returns immediately and

doesn't wait for the memory transfer to be completed. However, any subsequent task submitted to this particular stream will wait until it's completed.

If the last task in the stream queue is an asynchronous memory copy from device to host to store the results of the computation, then before using the data, the stream must be synchronized to ensure the memory copy is completed. The function `cudaStreamSynchronize` serves the same purpose on the host side as the function `__syncthreads`:

```
cudaStreamSynchronize(stream);
```

The stream will stop at this command until all the tasks assigned to `stream` are completed.

## 25.3 • Single Stream Example

To facilitate the discussion for multiple streams later, the structure of the program has changed a little bit as well to accommodate the periodic boundary conditions. The initial data will now have length `N+2` so that the kernel doesn't have to do the modular arithmetic that has been used up to now. This will necessitate some changes in the indexing in the kernel. Otherwise, the program is very similar to Example 24.2. Example 25.1 also includes code for events to compute timing tasks, which will be discussed in Section 25.6. Lines that differ from previous examples and the new functions are commented in the code.

### Example 25.1.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4 #include "checkerror.h"
5
6 #ifndef M_PI
7 #define M_PI 3.1415926535897932384626433832795
8 #endif
9
10 const int threadsPerBlock = 256;
11 __device__ __constant__ int dev_N;
12 __device__ __constant__ double dev_dx;
13
14 __global__ void diff(double* u, double* du);
15
16 /*
17  int main(int argc, char* argv[])
18  Demonstrate a simple example for implementing a
19  parallel finite difference operator
20
21  Inputs: argc should be 2
22  argv[1]: Length of the vector of data
23
24  Outputs: the initial data and its derivative.
25 */
26
27 int main(int argc, char* argv[]) {
28
29 // Get the length of the data from the argument list
30 int N = atol(argv[1]);
31 double dx = 2*M_PI/N;
32 const int blocksPerGrid = N/threadsPerBlock
```

```

33                                     + (N%threadsPerBlock > 0 ? 1 : 0);
34 double *u, *du;
35
36 double* dev_u;
37 double* dev_du;
38
39 // Set the constant values for N and dx on the device
40 cudaMemcpyToSymbol(dev_dx, &dx, sizeof(double));
41 cudaMemcpyToSymbol(dev_N, &N, sizeof(int));
42
43 // Replace regular malloc with cudaHostAlloc so that memory is pinned
44 cudaHostAlloc((void**)&u, (N+2)*sizeof(double), cudaHostAllocDefault);
45 cudaHostAlloc((void**)&du, N*sizeof(double), cudaHostAllocDefault);
46
47 // Allocate memory on the device
48 cudaMalloc((void**)&dev_u, (N+2)*sizeof(double));
49 cudaMalloc((void**)&dev_du, N*sizeof(double));
50
51 // Create a new stream
52 cudaStream_t stream;
53 cudaStreamCreate( &stream );
54
55 // Initialize the data
56 for (int i=0; i<N; ++i)
57     u[i+1] = sin(i*dx);
58 u[0] = u[N];
59 u[N+1] = u[1];
60
61 // The memory copy is performed asynchronously,
62 // control returns immediately
63 cudaMemcpyAsync(dev_u, u, (N+2)*sizeof(double), cudaMemcpyHostToDevice,
64                                         stream);
65
66 // Create the CUDA events that will be used for timing
67 // the kernel function
68 cudaEvent_t start, stop;
69 cudaEventCreate(&start);
70 cudaEventCreate(&stop);
71
72 // Click, the timer has started running
73 cudaEventRecord(start, 0);
74
75 // The kernel call is also issued asynchronously because we've
76 // specified a stream which is the 4th argument
77 diff<<<blocksPerGrid, threadsPerBlock, 0, stream>>>(dev_u, dev_du);
78
79 // Click, the timer event stops when all threads synchronize.
80 cudaEventRecord(stop, 0);
81 cudaEventSynchronize(stop);
82
83 // The elapsed time is computed by taking the difference between
84 // start and stop
85 float elapsedTime;
86 cudaEventElapsedTime(&elapsedTime, start, stop);
87
88 // The solution data can also be copied asynchronously from the device
89 // back to the host
90 cudaMemcpyAsync(du, dev_du, N*sizeof(double), cudaMemcpyDeviceToHost,
91                                         stream);
92
93 // Because the data is copied asynchronously, we can't use it
94 // until we're sure the stream has finished
95 cudaStreamSynchronize(stream);

```

```

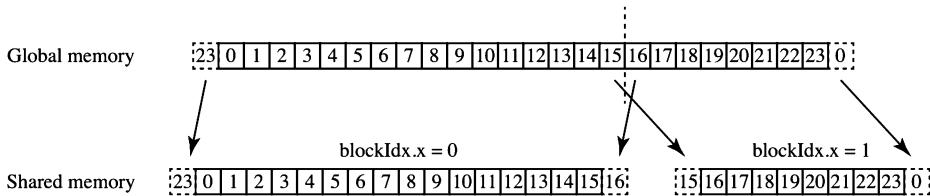
96
97 // The CUDA events have to be disposed of properly
98 cudaEventDestroy(start);
99 cudaEventDestroy(stop);
100
101 cudaFree(dev_u);
102 cudaFree(dev_du);
103
104 // The pinned memory must also be freed using a CUDA function
105 cudaFreeHost(u);
106 cudaFreeHost(du);
107
108 // The stream must be disposed of as well.
109 cudaStreamDestroy(stream);
110
111 printf("Time: %gms\n", elapsedTime);
112 return 0;
113 }

115 __global__ void diff(double* u, double* du) {
116     __shared__ double localu[threadsPerBlock+2];
117     int g_i = threadIdx.x + blockIdx.x * blockDim.x + 1;
118
119     // The length of the data vector may be shorter than
120     //      the number of threads
121     if (g_i <= dev_N) {
122         int l_i = threadIdx.x + 1;
123         localu[l_i] = u[g_i];
124
125         // The first thread must pick up the boundary data at the left
126         if (threadIdx.x == 0)
127             localu[0] = u[g_i-1];
128
129         // The last thread in the range needs to also double up for the
130         //      boundary data on the right
131         if (g_i == dev_N
132             || (g_i < dev_N && threadIdx.x == threadsPerBlock - 1))
133             localu[l_i+1] = u[g_i+1];
134
135         // The threads must complete the copy from global to shared memory
136         // before the calculation
137         __syncthreads();
138
139         du[g_i-1] = (localu[threadIdx.x+2]-localu[threadIdx.x])/dev_dx/2.;
140     } else {
141
142         // All threads must issue this command whether involved
143         //      in the computation or not
144         __syncthreads();
145     }
146 }

```

The indexing used in the kernel function is illustrated in Figure 25.1, where it assumes 16 threads per block. In the second block, the threads with indices too large are excluded on Line 121. Additionally, the first and last threads with a role in computing the finite difference are also used to copy the boundary data on Lines 126–133. The threads copying data must be synchronized before performing the finite difference as was discussed before. This time, because some of the threads are excluded from Line 137, the rest of the threads also have to synchronize or the program will block here, hence the second synchronization point on Line 144.

Note that this is not the only way to organize the threads and data. An alternative would be to use the 16 threads to compute 14 finite differences with the extra 2 threads used just to copy the boundary data but not compute the finite difference. In this case, the transfer from global to shared memory may be a bit faster because no threads must do a second load, but then fewer threads perform the finite difference operator, hence requiring more blocks/threads to complete the task. These kinds of trade-offs are important to consider when designing a fast code because they can have a significant effect on the final performance. Comparing the two strategies in Figures 25.5 and 25.6 shows that this latter strategy is a little less efficient than having the end threads load the boundary data as is done in Example 25.1.



**Figure 25.1.** Illustration of the global and shared memory arrangement for Example 25.1 assuming 16 threads per block. The first block gets a complete set of shared memory data, but the second block is incomplete because the vector length is too short.

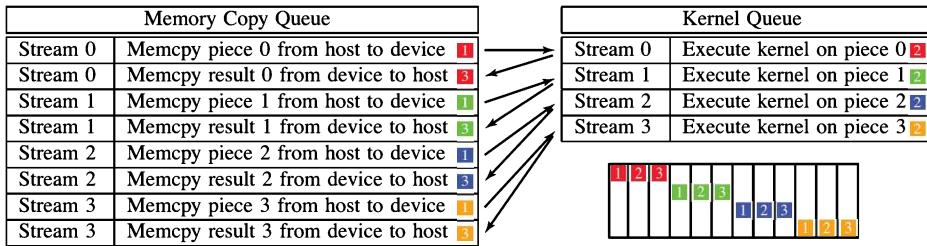
## 25.4 • Multiple Streams

The power of streams doesn't really kick in until they work in tandem to take full advantage of the various computing components. For example, the key is to get memory transfers going while a kernel is running. The dance steps of the various streams and operations in the streams are critical for optimal performance. Unfortunately, the sequential nature of serial programming is not quite in line with how the GPU works.

Suppose the finite difference calculation is broken into four pieces, one piece for each of four streams. One way to queue up the commands for the streams would be to post them with a loop through each stream:

Stream 0	Memcpy piece 0 from host to device
Stream 0	Execute kernel on piece 0
Stream 0	Memcpy result 0 from device to host
Stream 1	Memcpy piece 1 from host to device
Stream 1	Execute kernel on piece 1
Stream 1	Memcpy result 1 from device to host
Stream 2	Memcpy piece 2 from host to device
Stream 2	Execute kernel on piece 2
Stream 2	Memcpy result 2 from device to host
Stream 3	Memcpy piece 3 from host to device
Stream 3	Execute kernel on piece 3
Stream 3	Memcpy result 3 from device to host

where the loop is unrolled to see the sequence of instructions issued. The GPU will now queue these instructions in the order received into two queues: the Memory Copy Queue and the Kernel Queue. The two queues end up as shown in Figure 25.2. The Memory Copy Queue and the Kernel Queue can operate simultaneously in this model, so items



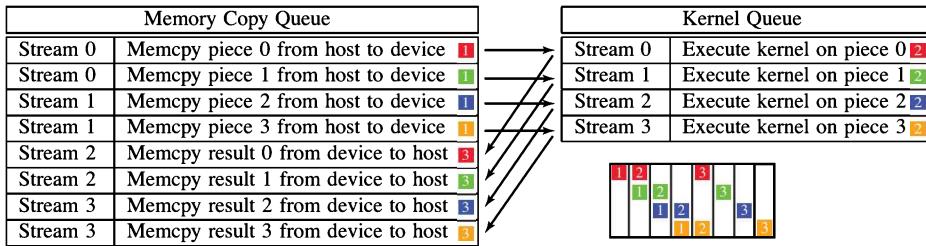
**Figure 25.2.** Order of events for the Memory Copy and Kernel Queues when events are submitted stream by stream. The arrows indicate the dependencies according to the order of operations within each stream, which is color-coded. The final timing of the events is shown in the lower right. They are sequential because [3] was submitted before [1].

labeled with a 2 can be executed simultaneously with items labeled 1 or 3 in Figure 25.2. The problem with this arrangement is that the device to host memory copy for stream 0 depends on the stream 0 kernel, and since that is in the queue ahead of the stream 1 memory copy from host to device, the stream 1 memory copy must wait until stream 0 is completely finished. This effectively forces the operations to work sequentially as if they are all in the same stream rather than simultaneously as illustrated in the sequence in the bottom right.

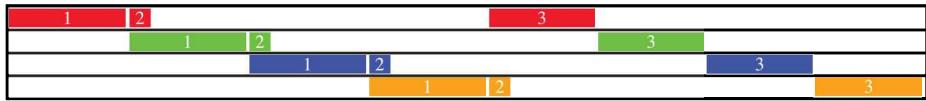
Compare the previous results to the following sequence for issuing commands:

Stream 0	Memcpy piece 0 from host to device
Stream 1	Memcpy piece 1 from host to device
Stream 2	Memcpy piece 2 from host to device
Stream 3	Memcpy piece 3 from host to device
Stream 0	Execute kernel on piece 0
Stream 1	Execute kernel on piece 1
Stream 2	Execute kernel on piece 2
Stream 3	Execute kernel on piece 3
Stream 0	Memcpy result 0 from device to host
Stream 1	Memcpy result 1 from device to host
Stream 2	Memcpy result 2 from device to host
Stream 3	Memcpy result 3 from device to host

The resulting sequences in the queues are shown in Figure 25.3. By ordering the operations in this way, the Memory Copy Queue and the Kernel Queue can now work simultaneously to complete operations. The result is that the number of steps is cut down from 12 to 8, resulting in some savings. This kind of stacking is very easy to implement, but for the best results, consideration of the length of different operations is required. For this simple example, there isn't much flexibility, but using this same sequence of operations, and using the timings provided by a test run of Example 25.1, the scheduling figure would actually look more like that in Figure 25.4. It should be evident from this illustration that there may not be much savings in using multiple streams for this particular task. It should also be evident from this illustration that the largest share of the effort is going into memory transfers between the host and the device. Consequently, it will be strongly advised to try and keep computations on the GPU as much as possible without moving data between the host and the GPU.



**Figure 25.3.** Order of events for the Memory Copy and Kernel Queues when events are submitted task by task instead of stream by stream. The arrows indicate the dependencies according to the order of operations within each stream, which is color-coded. The final timing of the events is shown in the lower right. They overlap because [3] was submitted after the last [1].



**Figure 25.4.** Illustration of the actual size of the boxes from Figure 25.3 based on timing data. Overlap is helpful, but clearly the data transfers are far more expensive than the cost of the kernel.

The implementation of four streams in this manner is not much different from how the single stream was written, but the example code is presented below for reference. The difference in timing can be seen in Figures 25.5 and 25.6 with the labels (1-stream) and (2-streams).

### Example 25.2.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include "checkerror.h"
4
5 #ifndef M_PI
6 #define M_PI 3.1415926535897932384626433832795
7 #endif
8
9 // This is the number of streams that will be used.
10 // Does not have to be a constant.
11 const int numStreams = 4;
12 const int threadsPerBlock = 256;
13 __device__ __constant__ int dev_N;
14 __device__ __constant__ double dev_dx;
15
16 __global__ void diff(double* u, double* du);
17
18 /*
19  int main(int argc, char* argv[])
20  Demonstrate a simple example for implementing a
21  parallel finite difference operator
22
23  Inputs: argc should be 2
24  argv[1]: Length of the vector of data
25
26  Outputs: the initial data and its derivative.

```

```

27 */
28
29 int main(int argc, char* argv[]) {
30
31     int N = atol(argv[1]);
32
33     // The length of each subsection to be handled by each stream.
34     // Assumes here that N is evenly divisible by numStreams so that
35     // all pieces have equal length
36     int localN = N/numStreams;
37     double dx = 2*M_PI/N;
38
39     // The grid is cut into numStreams pieces,
40     // the number of blocks are determined by the local length
41     const int blocksPerGrid = localN/threadsPerBlock
42                                     + (localN%threadsPerBlock > 0 ? 1 : 0);
43     double *u, *du;
44
45     // The global data on the device will be cut into equal pieces
46     // for each stream
47     double* dev_u[numStreams];
48     double* dev_du[numStreams];
49
50     cudaMemcpyToSymbol(dev_dx, &dx, sizeof(double));
51     cudaMemcpyToSymbol(dev_N, &localN, sizeof(int));
52
53     // Only one copy of the data on the host, and that is pinned
54     // for asynchronous transfer
55     cudaHostAlloc((void**)&u, (N+2)*sizeof(double), cudaHostAllocDefault);
56     cudaHostAlloc((void**)&du, N*sizeof(double), cudaHostAllocDefault);
57
58     // Create an array of streams, and allocate device memory
59     // for each of them.
60     cudaStream_t stream[numStreams];
61     for (int i=0; i<numStreams; ++i) {
62         cudaMalloc((void**)&dev_u[i], (N/numStreams+2)*sizeof(double));
63         cudaMalloc((void**)&dev_du[i], N/numStreams*sizeof(double));
64         cudaStreamCreate(&(stream[i]));
65     }
66
67     for (int i=0; i<N; ++i)
68         u[i+1] = sin(i*dx);
69     u[0] = u[N];
70     u[N+1] = u[1];
71
72     // First we queue the memcpy from host to device for all streams
73     for (int i=0; i<numStreams; ++i)
74         cudaMemcpyAsync(dev_u[i], u+(i*N/numStreams),
75                         (N/numStreams+2)*sizeof(double), cudaMemcpyHostToDevice,
76                         stream[i]);
77
78     cudaEvent_t start, stop;
79     cudaEventCreate(&start);
80     cudaEventCreate(&stop);
81     cudaEventRecord(start, 0);
82
83     // Next queue the kernel functions for all streams
84     for (int i=0; i<numStreams; ++i)
85         diff<<<blocksPerGrid, threadsPerBlock, 0, stream[i]>>>(dev_u[i],
86                                                               dev_du[i]);
87
88     cudaEventRecord(stop, 0);
89     cudaEventSynchronize(stop);

```

```

90    float elapsedTime;
91    cudaEventElapsedTime(&elapsedTime, start, stop);
92
93    // Finally queue the memcpy to get the results off the device and
94    // back in the host
95    for (int i=0; i<numStreams; ++i)
96        cudaMemcpyAsync(du+i*N/numStreams, dev_du[i],
97                        N/numStreams*sizeof(double), cudaMemcpyDeviceToHost, stream[i]);
98
99    // Each stream must be synchronized before using the results.
100   for (int i=0; i<numStreams; ++i)
101       cudaStreamSynchronize(stream[i]);
102
103   // Time to clean up
104   cudaEventDestroy(start);
105   cudaEventDestroy(stop);
106
107   for (int i=0; i<numStreams; ++i) {
108       cudaFree(dev_u[i]);
109       cudaFree(dev_du[i]);
110   }
111
112   cudaFreeHost(u);
113   cudaFreeHost(du);
114
115   for (int i=0; i<numStreams; ++i)
116       cudaStreamDestroy(stream[i]);
117
118   printf("Time: %gms\n", elapsedTime);
119   return 0;
120 }
121
122 __global__ void diff(double* u, double* du) {
123     __shared__ double localu[threadsPerBlock+2];
124     int g_i = threadIdx.x + blockIdx.x * blockDim.x + 1;
125     if (g_i <= dev_N) {
126         int l_i = threadIdx.x + 1;
127         localu[l_i] = u[g_i];
128         if (threadIdx.x == 0)
129             localu[0] = u[g_i-1];
130         if (g_i == dev_N || (g_i < dev_N
131                             && threadIdx.x == threadsPerBlock - 1))
132             localu[l_i+1] = u[g_i+1];
133
134         __syncthreads();
135
136         du[g_i-1] = (localu[threadIdx.x+2]-localu[threadIdx.x])/dev_dx/2.;
137     } else {
138         __syncthreads();
139     }
140 }
```

### 25.4.1 • Multiple Devices

It is commonplace for machines to have multiple GPU devices, so it makes sense to determine how to use them for the same computation and when doing so is a good idea. There are different methods for managing multiple GPUs for a calculation; Example 25.3 is a modification of Example 25.2 that assigns each device a single stream.

**Example 25.3.**

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #ifndef M_PI
5 #define M_PI 3.1415926535897932384626433832795
6 #endif
7
8 const int threadsPerBlock = 256;
9 __device__ __constant__ int dev_N;
10 __device__ __constant__ double dev_dx;
11
12 __global__ void diff(double* u, double* du);
13
14 /*
15  * int main(int argc, char* argv[])
16  * Demonstrate a simple example for implementing a
17  * parallel finite difference operator
18  *
19  * Inputs: argc should be 2
20  * argv[1]: Length of the vector of data
21  *
22  * Outputs: the initial data and its derivative.
23 */
24
25 int main(int argc, char* argv[]) {
26
27     int N = atol(argv[1]);
28
29     int numDevices;
30     cudaGetDeviceCount(&numDevices);
31     printf("Found %d devices\n", numDevices);
32
33     // The length of each subsection to be handled by each stream.
34     // Assumes here that N is evenly divisible by numDevices so that
35     // all pieces have equal length
36     int localN = N/numDevices;
37     double dx = 2*M_PI/N;
38
39     // The grid is cut into numDevices pieces,
40     // the number of blocks are determined by the local length
41     const int blocksPerGrid = localN/threadsPerBlock
42                                     + (localN%threadsPerBlock > 0 ? 1 : 0);
43     double *u, *du;
44
45     // The global data on the device will be cut into equal pieces
46     // for each stream
47     double* dev_u[numDevices];
48     double* dev_du[numDevices];
49
50     // Only one copy of the data on the host, and that is pinned
51     // for asynchronous transfer
52     cudaHostAlloc((void**)&u, (N+2)*sizeof(double), cudaHostAllocDefault);
53     cudaHostAlloc((void**)&du, N*sizeof(double), cudaHostAllocDefault);
54
55     // Create an array of streams, and allocate device memory
56     // for each of them.
57     cudaStream_t stream[numDevices];
58     for (int i=0; i<numDevices; ++i) {
59         cudaSetDevice(i);
60         cudaMemcpyToSymbol(dev_dx, &dx, sizeof(double));
61         cudaMemcpyToSymbol(dev_N, &localN, sizeof(int));

```

```

62     cudaMalloc (( void **) &dev_u [ i ] , ( N / numDevices + 2 ) * sizeof ( double ) );
63     cudaMalloc (( void **) &dev_du [ i ] , N / numDevices * sizeof ( double ) );
64     cudaStreamCreate ( &( stream [ i ] ) );
65   }
66
67   for ( int i = 0; i < N; ++i )
68     u [ i + 1 ] = sin ( i * dx );
69   u [ 0 ] = u [ N ];
70   u [ N + 1 ] = u [ 1 ];
71
72 // First we queue the memcpy from host to device for all streams
73 for ( int i = 0; i < numDevices; ++i ) {
74   cudaSetDevice ( i );
75   cudaMemcpyAsync ( dev_u [ i ] , u + ( i * N / numDevices ) ,
76                   ( N / numDevices + 2 ) * sizeof ( double ) , cudaMemcpyHostToDevice ,
77                                         stream [ i ] );
78 }
79
80 cudaEvent_t start [ numDevices ] , stop [ numDevices ];
81 // Next queue the kernel functions for all streams
82 for ( int i = 0; i < numDevices; ++i ) {
83   cudaSetDevice ( i );
84   cudaEventCreate ( &start [ i ] );
85   cudaEventCreate ( &stop [ i ] );
86   cudaEventRecord ( start [ i ] , stream [ i ] );
87   diff <<< blocksPerGrid , threadsPerBlock , 0 , stream [ i ] >>> ( dev_u [ i ] ,
88                                                               dev_du [ i ] );
89 }
90
91 float elapsedTime [ numDevices ];
92 for ( int i = 0; i < numDevices; ++i ) {
93   cudaSetDevice ( i );
94   cudaEventRecord ( stop [ i ] , stream [ i ] );
95   cudaEventSynchronize ( stop [ i ] );
96   cudaEventElapsedTime ( &elapsedTime [ i ] , start [ i ] , stop [ i ] );
97 }
98
99 // Finally queue the memcpy to get the results off the device and
100 // back in the host
101 for ( int i = 0; i < numDevices; ++i ) {
102   cudaSetDevice ( i );
103   cudaMemcpyAsync ( du + i * N / numDevices , dev_du [ i ] ,
104                   N / numDevices * sizeof ( double ) , cudaMemcpyDeviceToHost ,
105                                         stream [ i ] );
106 }
107
108 // Each stream must be synchronized before using the results .
109 for ( int i = 0; i < numDevices; ++i ) {
110   cudaSetDevice ( i );
111   cudaStreamSynchronize ( stream [ i ] );
112 }
113
114 // Time to clean up
115 for ( int i = 0; i < numDevices; ++i ) {
116   cudaEventDestroy ( start [ i ] );
117   cudaEventDestroy ( stop [ i ] );
118 }
119
120 cudaFreeHost ( u );
121 cudaFreeHost ( du );
122
123 for ( int i = 0; i < numDevices; ++i ) {
124

```

```

125     cudaSetDevice(i);
126     cudaFree(dev_u[i]);
127     cudaFree(dev_du[i]);
128     cudaStreamDestroy(stream[i]);
129 }
130
131 return 0;
132 }

__global__ void diff(double* u, double* du) {
    __shared__ double localu[threadsPerBlock+2];
    int g_i = threadIdx.x + blockIdx.x * blockDim.x + 1;
    if (g_i <= dev_N) {
        int l_i = threadIdx.x + 1;
        localu[l_i] = u[g_i];
        if (threadIdx.x == 0)
            localu[0] = u[g_i-1];
        if (g_i == dev_N || (g_i < dev_N
            && threadIdx.x == threadsPerBlock -1))
            localu[l_i+1] = u[g_i+1];
        __syncthreads();
        du[g_i-1] = (localu[threadIdx.x+2]-localu[threadIdx.x])/dev_dx/2.;
    } else {
        __syncthreads();
    }
}

```

There are relatively few changes to convert the multiple stream example into a code that utilizes multiple GPUs. When using multiple devices, it is useful to know what resources are available. The number of CUDA-capable devices on the system can be obtained using the function `cudaGetDeviceCount` as shown on Line 30. For simplicity, this example assumes that all devices will be used for equal pieces of the input data, but in practice if the available devices are not all the same, the size of the data given to each device should be proportional to its speed and capabilities. Next, recall that the function `cudaSetDevice` sets the device used for subsequent device operations. It can be called multiple times to switch between devices as needed, as shown on Lines 59, 75, 84, 94, 103, 111, and 125.

If the computation is to be distributed to multiple GPUs, then each GPU device will require the values for the constant memory, separate memory buffers need to be established for each device, and each device will be assigned a separate stream. This is done on Lines 59–65. The kernel is then executed on different devices by choosing a different device for each stream, as seen on Lines 84–88. The rest of the instances of using `cudaSetDevice` are similar.

## 25.5 • General Strategies

As shown in Figure 25.4, the transfer of data between the host and the device can be a significant cost even when overlapped with kernel calls, so minimizing data transfers is important. Therefore, keep the data on the GPU as much as possible and just issue sequences of kernel functions. For example, suppose there is a kernel function `onestep` that does one time step for one grid point of a time-dependent problem. There's no reason to ship data back and forth between the host and the device every time step;

instead transfer the data only when needed, e.g., every 100th step:

```

1 // Set up the initial data
2 cudaMemcpyAsync(dev_u0, u0, N*sizeof(double), cudaMemcpyHostToDevice,
3                                         stream);
4
5 for (int bigstep=0; bigstep*100*dt<Tfinal; ++bigstep) {
6
7     for (int step=0; step<50; ++step) {
8         diff <<<blocksPerGrid, threadsPerBlock, 0, stream>>>(dev_u0,
9                                         dev_u1);
10        diff <<<blocksPerGrid, threadsPerBlock, 0, stream>>>(dev_u1,
11                                         dev_u0);
12    }
13
14    // Get the current output data
15    cudaMemcpyAsync(u1, dev_u0, N*sizeof(double),
16                                         cudaMemcpyDeviceToHost, stream);
17    cudaStreamSynchronize(stream);
18    saveDataToFile(100*bigstep, u1);
19 }
```

By putting the kernel in a loop like this, the data is kept on the GPU for faster access.

There are additional tricks one can play. For example, the above loop requires that each kernel move data between global memory and shared memory each iteration. If the overlap between neighboring grids is made wider, then more steps can be executed in a single kernel, thus eliminating some data transfers between global and shared memory, further improving performance. Of course, these strategies are very application dependent, and there is a threshold where they make things worse rather than better, but that is part of the tuning process.

## 25.6 • Measuring Performance

In order to evaluate the speed of individual kernels on GPUs, CUDA events will be inserted into the stream at the beginning and the end of the code to be timed, and the difference in time between the events will determine the time of execution. A CUDA event is a variable of type `cudaEvent_t`:

```
cudaEvent_t start;
cudaEventCreate(&start);
```

The time is recorded into the event by using

```
cudaEventRecord(start, 0);
```

where the second argument is the stream number, in this case the default stream. When explicitly using streams, the stream is given as the second argument.

The kernel function is run asynchronously with the host code, so the time to finish the kernel must be after all the blocks and threads have completed, so another synchronization point on the host side is needed. This is accomplished with the function `cudaEventSynchronize` that takes as an argument the event record that requires synchronization. Therefore, to record the ending time of a kernel call, use

```
cudaEvent_t stop;
cudaEventCreate(&stop);
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
```

The call to `cudaEventRecord` inserts the `stop` event into the stream, and the function `cudaEventSynchronize` pauses the host until the `stop` event is at the top of the queue.

Finally, use the function `cudaEventElapsedTime` to get the elapsed time between two events. The function takes a pointer to a `float` as the first argument where the time will be recorded, and the two CUDA events between which the time is measured in milliseconds. The example below shows how to record the time required to execute the kernel `diff` on the GPU:

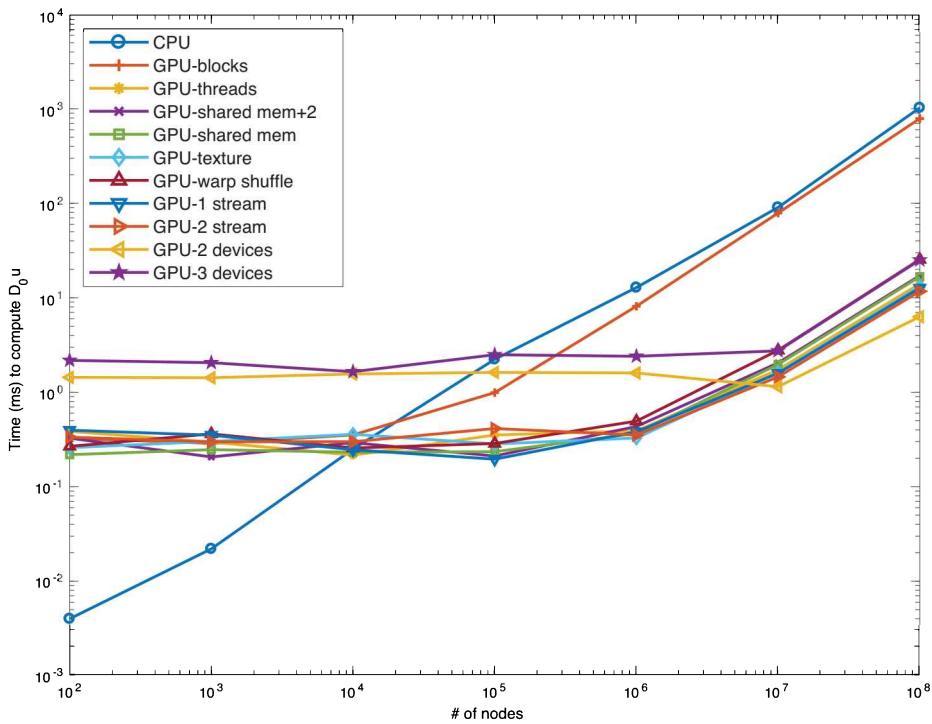
#### Example 25.4.

```
1  cudaEvent_t start , stop ;
2  cudaEventCreate(& start );
3  cudaEventCreate(& stop );
4  cudaEventRecord( start , 0 );
5
6  diff <<<blocksPerGrid , threadsPerBlock >>>( dev _du );
7
8  cudaEventRecord( stop , 0 );
9  cudaEventSynchronize( stop );
10 float elapsedTime ;
11 cudaEventElapsedTime(&elapsedTime , start , stop );
```

Using this timing method, Figure 25.5 shows the time required for each of the various methods for computing the finite difference discussed as a function of the length,  $N$ , of the data. It clearly shows that simply using a kernel with many blocks does not achieve much improvement over a regular serial C code. This is because the blocks are not able to coordinate their instructions with each other, and hence they essentially serialize the code, just on the GPU instead of the CPU. The limitations on the number of threads means that for large problems, using only one block with many threads is also not feasible. At the same time, taking advantage of cached shared memory or texture memory can result in almost 200 times faster code. Using texture memory is also fast, but it has size limitations. In short, it is well worth the effort to learn all these implementations on the GPU to improve speed, and failure to do so may very well result in no improvement in performance at all.

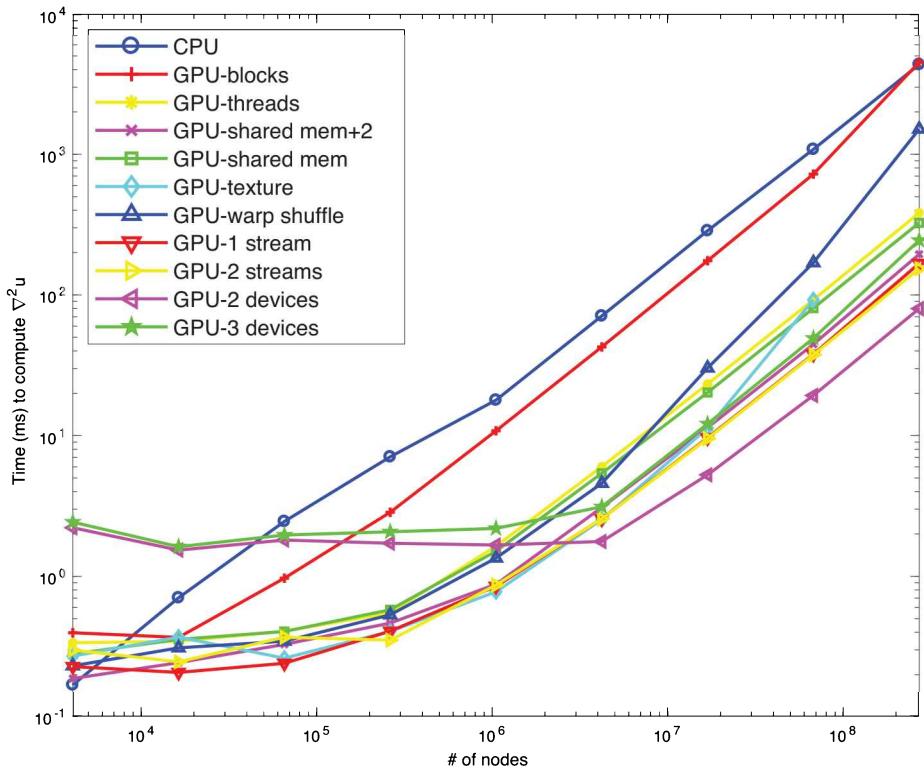
For a one-dimensional array, it's difficult to distinguish between many of the options in Figure 25.5, but they do separate when comparing two-dimensional arrays and computing the Laplacian using finite differences. Figure 25.6 shows the same algorithms adapted to computing the two-dimensional Laplacian using a finite difference approximation. It clearly shows that using streams and shared memory are a solid strategy for this application. However, different applications can have very different results, so all methods should be considered for their merits when choosing a particular strategy.

A few comments about Figure 25.6: (1) In the context of these timings, which were used only to assess the speed of the finite difference kernel itself and not including the time required to transfer the data to/from the device, using one or more streams with asynchronous data transfers will not see any improvements in speed since the data transfers were not included. (2) Warp shuffles work well in one dimension, but two dimensions is more challenging. In this implementation two separate kernels, one for the  $x$  derivative and one for the  $y$  derivative, were used so that the shuffle could be done in lengths of 32, which is the maximum for the warp. Warps work best when



**Figure 25.5.** Scaling properties of one-dimensional finite central difference using various versions of parallelism in CUDA compared to a serial CPU implementation. The horizontal axis shows the number of nodes in the grid, and the vertical axis shows the time to complete the finite difference kernel function. The legend refers to (CPU) Serial CPU code, no CUDA code; (GPU-blocks) Results from using Example 23.1; (GPU-threads) Results from changing the kernel call in Example 23.1 to have  $\lceil N/1024 \rceil$  blocks and 1024 threads per block; (GPU-shared mem+2) Results from using 1024 threads per block and shared memory as in Example 24.2; (GPU-shared mem) Results from using 1024 threads per block and shared memory as in Example 24.2 but where end threads do not do an extra load (see Section 27.2); (GPU-texture) Results from using texture memory instead of shared memory as in Example 24.5, noting that  $N = 10^8$  is not included due to texture memory limitations; (GPU-warp shuffle) Results from using warp shuffles to compute the finite difference; (GPU-1 stream) Results from explicitly using 1 stream and shared memory; (GPU-2 stream) Results from dividing up the computation between two streams; (GPU-2 devices) Results from dividing up the computation between two identical devices; (GPU-3 devices) Results from dividing up the computation equally between two identical devices and one additional slower device.

loading contiguous data, which is true for one dimension, but not for the other, hence some performance loss. (3) For the system where this data was generated, there was insufficient texture memory to generate timing data on the  $16,384 \times 16,384$  grid. (4) There is additional cost to using multiple GPUs, so it only makes sense to use multiple GPUs for a sufficiently large grid, but for that larger grid there is a significant advantage. (5) Using heterogeneous GPUs requires special care to make sure the load is balanced properly. If not, then the computational speed is limited by the most loaded device, leading to less efficiency.



**Figure 25.6.** Scaling properties of two-dimensional finite difference Laplacian using various versions of parallelism in CUDA compared to a serial CPU implementation. The horizontal axis shows the number of nodes in the grid, and the vertical axis shows the time to complete the finite difference kernel function. The legend refers to (CPU) Serial CPU code, no CUDA code; (GPU-blocks) Results from using  $N \times N$  block dimensions and 1 thread per block; (GPU-threads) Results from using  $\lceil N/32 \rceil \times \lceil N/32 \rceil$  blocks and  $32 \times 32$  threads; (GPU-shared mem+2) Results from using  $\lceil N/32 \rceil \times \lceil N/32 \rceil$  blocks  $32 \times 32$  threads per block; (GPU-shared mem) Same as previous except side threads do not load extra value (see Section 27.2); (GPU-texture) Same as (GPU-threads) except using texture memory instead of shared memory, noting that  $N = 10^8$  is not included due to texture memory limitations; (GPU-warp shuffle) Results from using warp shuffles to compute the finite difference; (GPU-1 stream) Same as (GPU-shared mem+2) with explicitly using 1 stream; (GPU-2 stream) Same as (GPU-1 stream) except computation divided between two streams; (GPU-2 devices) Results from using 2 identical GPU devices; (GPU-3 devices) Results from using 2 identical GPU devices plus an additional slower GPU.

## Exercises

- 25.1. Modify Example 23.2 to subdivide the domain in such a way that a maximal number of threads is used per block to cover the given dimensions  $M, N$ . Use the function `cudaGetDeviceProperties` to obtain what the maximum number of threads is for your device. Use shared memory for the local array to improve efficiency.

- 25.2. Write a program that uses streams to load a three-dimensional array from host memory onto the device, compute the Laplacian on the array, and then transfer the results back to host memory. Use the timing tools to estimate how much time is consumed doing memory transfers versus computing the finite difference.
- 25.3. If you have two different GPU devices in your computer system, modify Example 25.3 to do load balancing between the two devices. You will need to measure the time required for each device to compute the solution for different values of  $N$  and then divide the array accordingly so they each spend approximately the same amount of time to complete the task assigned them. Verify the timing by using events.

# Chapter 26

# CUDA Libraries

The CUDA package includes some useful libraries, including a version of the BLAS library, an FFT package based on FFTW called cuFFT, and a sparse matrix library called cuSPARSE. In addition, there are a few CUDA-based implementations of LAPACK available on the web. The library MAGMA is developed by the same group that created LAPACK, so much of the terminology and solution steps in MAGMA are very similar to that in LAPACK. This chapter goes through the basics of using these packages.

## 26.1 • MAGMA

MAGMA is an acronym for matrix algebra for GPU and multicore architectures. The MAGMA library documentation is located at

<http://icl.cs.utk.edu/magma/software/>

and there is a very helpful document (see [21]) that details many examples of usage. The package is a successor to the LAPACK package, and many of the arguments and calling conventions used for those packages are also used for MAGMA so that the transition is straightforward.

The CUDA language calls in the MAGMA library are all embedded so it is possible to use MAGMA solely as a linear algebra engine and not use any CUDA code. If that is done, then the program can be saved with the “.c” suffix and compiled with `gcc` rather than using `nvcc`. However, it can also be compiled with `nvcc` without modification.

### 26.1.1 • Simple Example of a Linear Solve

As a simple test case for the MAGMA package, Example 26.1 solves a linear system of equations using the function `magma_dgesv`, which is the MAGMA equivalent of the LAPACK function `dgesv_`. The matrix `A` in this example is stored in column-major format, the same way as was done for LAPACK in Part I.

#### Example 26.1.

```
1 #include <stdio.h>
2 #include <cuda.h>
3 #include <magma.h>
```

```

4 #include <magma_types.h>
5 #include <magma_lapack.h>
6
7 /*
8 int main( int argc , char* argv[] )
9
10 The main program takes no arguments and prints nothing. A matrix A is
11 created in column-major format and a right-hand side is created.
12 The system of equations is then solved. The solution is stored in
13 the vector b upon return .
14
15 Inputs: none
16
17 Outputs: none
18 */
19 int main( int argc , char* argv[] )
20 {
21     // Initialize the MAGMA system
22     magma_init();
23
24     // temporary data required by dgesv similar to LAPACK
25     magma_int_t *piv, info;
26
27     // the matrix A is m x m.
28     magma_int_t m = 9;
29
30     // the right-hand side has dimension m x n
31     magma_int_t n = 1;
32
33     // error code for MAGMA functions
34     magma_int_t err;
35
36     // The matrix A and right-hand-side b
37     double* A;
38     double* b;
39
40     // Allocate matrices on the host using pinned memory
41     err = magma_dmalloc_pinned(&A, m*m);
42     err = magma_dmalloc_pinned(&b, m);
43     if (err) {
44         printf("oops, an error in memory allocation!\n");
45         exit(1);
46     }
47     // Create temporary storage for the pivots.
48     // Does not have to be pinned.
49     piv = (magma_int_t*)malloc(m*sizeof(magma_int_t));
50
51     // Generate matrix A
52     double row[9] = {3., 21., 4., 1., 13., 1., 7., 13., 1.};
53     int i, j, index = 0;
54     for (j=0; j<9; ++j)
55         for (i=0; i<9; ++i)
56             A[index++] = (i > j ? -1 : 1) * row[j];
57
58     // Generate b
59     b[0] = 17;
60     for (i=0; i<4; ++i) {
61         b[i+1] = 11;
62         b[i+5] = -15;
63     }
64
65     // MAGMA solve
66     magma_dgesv(m, n, A, m, piv, b, m, &info);

```

```

67 // Solution is stored in b
68
69 magma_free_pinned (A) ;
70 magma_free_pinned (b) ;
71 free (piv) ;
72 magma_finalize () ;
73 return 0;
74 }
```

The MAGMA system is set up on Line 22. This function sets up the system context that will be used to solve the equation. The MAGMA package is capable of using multiple CPUs and multiple GPUs and will balance the work depending on the context. Because the system will use multiple streams to solve the problem, use pinned memory for the input data. MAGMA takes care of the arguments necessary to create the pinned memory so that the program need only provide a pointer and the dimensions to the array, which is done on Lines 41–42. The temporary pivot data does not require pinning, and hence it can be allocated using the usual `malloc` function on Line 49.

The function `magma_dgesv` on Line 66 takes the same arguments as the LAPACK `dgesv_` function and in the same order, so that information will not be repeated here. See Section 7.1 for more information. Upon return, the solution is stored in the vector `b` and any error information is stored in the variable `info`.

For a matrix `A` with dimensions  $16,384 \times 16,384$ , a test comparison between the MAGMA package and standard LAPACK produced a solution approximately 18 times faster.

### 26.1.2 • Compiling and Linking MAGMA Code

In order to compile and link the program listed in Example 26.1, a number of libraries must be included. For the compile line, use the following command:

```
$ nvcc -O3 -DADD_ -DHAVE_CUBLAS -c program.c
```

The flag `-O3` is an optimization flag and asks for level 3 optimizations. Review Section 1.3.1 for more information on the optimization flag. The `-D` flag is equivalent to putting `#define` at the top of your file. These flags may vary for your installation version of MAGMA, so be sure to check with your system administrator or documentation.

The link line for the program also has a few bells and whistles to add so that all the libraries are linked properly. The link line for this program is

```
$ nvcc -o program program.o -lmagma -llapack -lcublas -lcudart
                                         -lm -lgfortran
```

where the MAGMA and CUDA libraries are explicitly linked. MAGMA is designed to be a multicore multi-GPU system, so for CPU implementations it still requires linking the LAPACK library and the corresponding `gfortran` library.

### 26.1.3 • CPU Interface Versus GPU Interface

In Example 26.1, the results of the computation were stored on the host. That may or may not be what is desired. In fact, it is more likely that it would be better to keep the

solution on the device rather than shifting it back and forth to the host. For that reason, MAGMA also has a GPU interface, where the inputs and the results are kept on the GPU. The GPU interface version of Example 26.1 is shown below in Example 26.2.

### Example 26.2.

```

1 #include <stdio.h>
2 #include <cuda.h>
3 #include <magma.h>
4 #include <magma_types.h>
5 #include <magma_lapack.h>
6
7 /*
8 int main(int argc, char* argv[])
9
10 The main program takes no arguments and prints nothing. A matrix A is
11 created in column-major format and a right-hand side is created.
12 The system of equations is then solved. The solution is stored in
13 the vector b upon return.
14
15 Inputs: none
16
17 Outputs: none
18 */
19 int main(int argc, char* argv[])
20 {
21     // Initialize the MAGMA system
22     magma_init();
23
24     // temporary data required by dgesv similar to LAPACK
25     magma_int_t *piv, info;
26
27     // the matrix A is m x m.
28     magma_int_t m = 9;
29
30     // the right-hand side has dimension m x n
31     magma_int_t n = 1;
32
33     // error code for MAGMA functions
34     magma_int_t err;
35
36     // The matrix A and right hand side b
37     double* A;
38     double* b;
39     double* dev_A;
40     double* dev_b;
41
42     // Allocate matrices on the host using pinned memory
43     err = magma_dmalloc_cpu(&A, m*m);
44     err = magma_dmalloc_cpu(&b, m);
45     err = magma_dmalloc(&dev_A, m*m);
46     err = magma_dmalloc(&dev_b, m);
47     if (err) {
48         printf("oops, an error in memory allocation!\n");
49         exit(1);
50     }
51     // Create temporary storage for the pivots.
52     // Does not have to be pinned.
53     piv = (magma_int_t*)malloc(m*sizeof(magma_int_t));
54
55     // Generate matrix A
56     double row[9] = {3., 21., 4., 1., 13., 1., 7., 13., 1.};

```

```

57  int i, j, index = 0;
58  for (j=0; j<9; ++j)
59    for (i=0; i<9; ++i)
60      A[index++] = (i > j ? -1 : 1) * row[j];
61
62  // Generate b
63  b[0] = 17;
64  for (i=0; i<4; ++i) {
65    b[i+1] = 11;
66    b[i+5] = -15;
67  }
68
69  // copy matrix on host onto device
70  magma_dsetmatrix(m, m, A, m, dev_A, m);
71  magma_dsetmatrix(m, n, b, m, dev_b, m);
72
73  // MAGMA solve
74 #ifdef USE_DGETRF
75  magma_dgetrf_gpu(m, m, dev_A, m, piv, &info);
76  magma_dgetrs_gpu(MagmaNoTrans, m, n, dev_A, m, piv, dev_b, m, &info);
77 #else
78  magma_dgesv_gpu(m, n, dev_A, m, piv, dev_b, m, &info);
79 #endif
80
81  // copy solution in dev_b back onto host
82  magma_dgetmatrix(m, n, dev_b, m, b, m);
83
84  free(A);
85  free(b);
86  magma_free(dev_A);
87  magma_free(dev_b);
88  free(piv);
89  magma_finalize();
90  return 0;
91 }
```

In order to take advantage of the GPU interface, the matrix **A** and the right-hand-side **b** must be on the device before calling the solver. Since the solver assumes the data is already on the device, there is no reason to set up pinned memory; hence the equivalent of a regular `malloc` is set up to create the matrix and right-hand-side vector on the host on Lines 43–44 using the `magma_dmalloc_cpu` function. Device memory to hold the input data and the results is done using `magma_dmalloc` on Lines 45–46.

Once the data is created on the host, it can be copied to the device using the function `magma_dsetmatrix` as shown on Lines 70–71. The linear solver can now be called, but this time the function is `magma_dgesv_gpu` as shown on Line 78, where the other change of note is that device memory is used as input rather than host memory. The result is then stored in the `dev_b` vector. To retrieve the results from the device, use `magma_dgetmatrix` as shown on Line 82.

Note that on Line 74, there is another compiler directive. If this example is compiled as is, the macro `USE_DGETRF` is undefined, and so Line 78 will be compiled and Lines 75, 76 will be skipped. However, if on the compile line, the flag “`-DUSE_DGETRF`” is used, or the line

```
#define USE_DGETRF
```

is added to the top of the file, then Lines 75, 76 will be included in the compile and Line 78 will be skipped. This allows for both solvers to be present in the file, with the

choice made at compile time, and it also shows how the single solver driver is called compared to separating the factorization from the solver.

If the `USE_DGETRF` macro is undefined, then the primary difference between Examples 26.1 and 26.2 is the use of the GPU interface solver `magma_dgesv_gpu`. Both the matrix `dev_A` and right-hand-side matrix `dev_b` are expected to be in device memory for this solver. Because device memory is specified, the function also requests offsets, hence the zeros in the fourth and eighth arguments of `magma_dgesv_gpu` as shown on Line 78. Otherwise, the arguments are the same as in the previous example.

If the `USE_DGETRF` macro is defined, then the functions `magma_dgetrf_gpu` and `magma_dgetrs_gpu` are used to solve the system. The primary purpose for separating the solution into two steps is because it is often the case that the matrix to be inverted remains constant, while the right-hand side may vary across multiple solutions. In that case, `magma_dgetrf_gpu` is called *once* to factor the matrix in `dev_A`, and the resulting factorization can be reused for multiple calls to `magma_dgetrs_gpu` leading to reduced computational cost. Another difference between the methods is that `magma_dgetrs_gpu` also takes an additional first argument that is either `MagmaTrans` or `MagmaNoTrans`. For `MagmaNoTrans`, the linear system solved is  $\mathbf{Ax} = \mathbf{b}$ , while for `MagmaTrans`, the system  $\mathbf{A}^T\mathbf{x} = \mathbf{b}$  is solved. That means that if the matrix  $\mathbf{A}$  is stored in row-major order, then using `MagmaTrans` would mean solving the linear system without having to transpose the matrix first. If `dev_b` is a matrix with more than one column, then it still must be stored in column-major order regardless. Note that for complex matrices, there is the additional `MagmaConjTrans` corresponding to solving  $\mathbf{A}^H\mathbf{x} = \mathbf{b}$ .

For either method, upon completion, the result is stored in the `dev_b` vector on the device. To retrieve the results from the device, use `magma_dgetmatrix` as shown on Line 82.

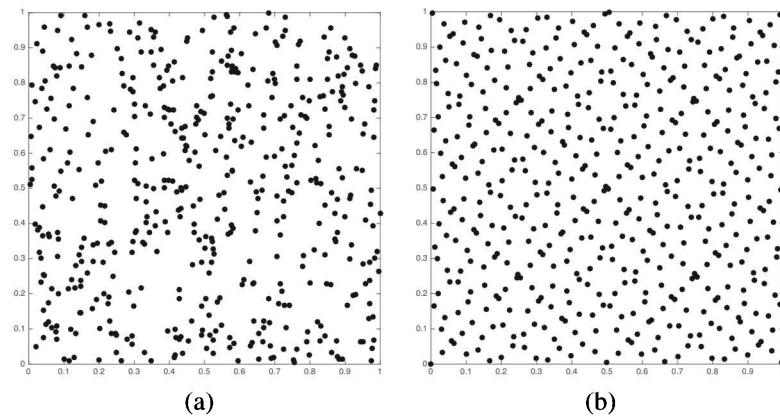
The last difference between Example 26.2 and Example 26.1 is that the host memory, since it's allocated with a regular `malloc`, can be released using a regular `free` as on Line 84. Meanwhile, the device memory allocated uses the `magma_free` function on Line 86.

## 26.2 • cuRAND

Generating random numbers in many threads requires attention to details for it to be done properly. For example, when 1,000 threads are all trying to generate random numbers, how can the threads be seeded uniquely? Failing to do so would cause distributions to have correlations, which could result in skewed results. The cuRAND library is a package to help deal with these issues.

Similar to the MAGMA package, one can use the package as a black box where an array of random numbers are produced in parallel by a call from the host, i.e., no kernel function is written, or the random number generator can be called from within a kernel function to produce a single random value. The library also has other distributions to sample from besides the uniform distribution as is produced by functions like `drand48`.

The package provides two different styles of random numbers: pseudorandom numbers and quasi-random numbers. Pseudorandom numbers are numbers generated by a deterministic linear congruential generator algorithm initialized with a seed. Quasi-random numbers are also random, but they are specially designed to be more uniformly spaced over the domain. That means that if, for example, a set of uniform random numbers are generated over the interval  $[0, 1]$ , the collection of random samples will be roughly uniformly spaced throughout the interval so that a histogram of the results



**Figure 26.1.** (a) Random points in a box generated with a pseudorandom number generator. (b) Random points in a box generated with a quasi-random number generator. Quasi-random numbers are intentionally more evenly spaced to improve accuracy of some algorithms such as Monte Carlo integration.

will be reasonably flat with not much variation. To see how the two cases look different, Figure 26.1 shows two sets of random points in the box  $[0, 1] \times [0, 1]$  illustrating how the quasi-random is more uniformly spaced compared to the pseudorandom values. Within these two general classifications of generators, the library offers multiple random sequence algorithms. For more information on this topic, check the documentation in

<http://docs.nvidia.com/cuda/curand>

### **26.2.1 ▪ Host Interface**

There are four basic steps to generating a list of random values using the host interface: (1) create the generator, (2) seed the generator, (3) use the generator to produce a list of random samples, and (4) release the memory allocated for the generator. The steps are illustrated in Example 26.3.

### **Example 26.3.**

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <cuda.h>
4 #include <curand.h>
5 /*
6    int main( int argc , char* argv[])
7
8    Example generating random numbers
9
10   Inputs: argc should be 2
11   argv[1]: Number of samples to generate
12
13   Outputs: the initial data and its derivative.
14 */
15
16

```

```

17 int main(int argc, char* argv[]) {
18
19     // Choose the GPU card
20     cudaDeviceProp prop;
21     int dev;
22     memset(&prop, 0, sizeof(cudaDeviceProp));
23     prop.multiProcessorCount = 13;
24     cudaChooseDevice(&dev, &prop);
25     cudaSetDevice(dev);
26
27     // Get the number of samples to take
28     int num_samples = atoi(argv[1]);
29
30     // Allocate space for the output and also for the GPU
31     double *x = (double*)malloc(num_samples*sizeof(double));
32     double *dev_x;
33     cudaMalloc((void**)&dev_x, num_samples*sizeof(double));
34
35     // Set up random number generator, use default pseudorandom
36     // number generator
37     curandGenerator_t gen;
38     curandCreateGenerator(&gen, CURAND_RNG_PSEUDO_DEFAULT);
39
40     // Seed the random number generator using the time
41     curandSetPseudoRandomGeneratorSeed(gen, time(NULL));
42
43     // Generate num_samples on the device using a uniform distribution
44     curandGenerateUniformDouble(gen, dev_x, num_samples);
45
46     // Copy results back to the host
47     cudaMemcpy(x, dev_x, num_samples*sizeof(double),
48                           cudaMemcpyDeviceToHost);
49
50     // Store result
51     FILE* outfile = fopen("Example26.3.out", "w");
52     fwrite(x, sizeof(double), num_samples, outfile);
53     fclose(outfile);
54
55     // Free the memory allocated including the generator
56     curandDestroyGenerator(gen);
57     cudaFree(dev_x);
58     free(x);
59     return 0;
60 }
```

The random number generator is created on Line 38, which is the default pseudorandom number algorithm. There are a number of other algorithm choices including CURAND\_RNG\_PSEUDO\_MT19937 for the Mersenne Twister generator and the Sobol quasi-random generator CURAND\_RNG\_QUASI\_SOBOLE64. See the online documentation for other choices and more information.

The random number generator is seeded on Line 41. The first argument is the created generator, and the second argument is an unsigned long long int value. For the sake of brevity, the output of the `time` function is used to create a new seed, but other methods such as sampling the file `/dev/urandom` as discussed in Section 7.3 could be used. See Section 7.3 for additional discussion about the importance of printing out the seed value and reproducibility of results.

A set of random values can be generated for a handful of different distributions. In this example, a uniform distribution on the interval [0, 1] is generated on Line 44. The

first argument is the generator previously set up, the second argument is the address *in device memory* where the values will be stored, and the third argument is the number of samples requested.

If the list of random values are needed on the host, then the array must be copied to the host as is done on Line 47. However, if further operations are to be performed on the array, then it is best to leave the data on the device and write kernels to manipulate the data as desired. For example, Line 44 might be followed by a transform that will change the distribution of the random values, or a reduction kernel to sum the values for purposes of computing the mean and variance.

Other distributions are available such as random integers similar to the `random` function in standard C, normal distribution, log-normal distribution, and Poisson distribution. The uniform, normal, and log-normal distributions are available in both single and double precision. For example, the single precision uniform distribution is called `curandGenerateUniform`.

### 26.2.2 • Device Interface

For a more refined control of the random number generation, or to incorporate the generator into a larger kernel function, there is also a device interface so that random numbers can be generated within kernel functions. For example, to have each thread solve a stochastic differential equation requiring many random values, each thread should have its own initial seed. Rather than repeatedly calling the host random generator as in Example 26.3, it's better to request a random generator from within the kernel.

To use the device interface, there is again a four-step process, where first a collection of generator states must be created, one for each thread; second, each of those states must be seeded within a setup kernel; third, the kernel can use the generator for the desired distribution; and finally there is a clean-up step. The sequence is illustrated in Example 26.4, where the kernel function generates 100 log-normal distributed random variables and sums them up.

#### Example 26.4.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <cuda.h>
4 #include <curand.h>
5 #include <curand_kernel.h>
6
7 __global__ void setup_kernel(int N, long int seed, curandState_t *state)
8 {
9     int id = threadIdx.x + blockIdx.x * blockDim.x;
10    if (id < N)
11        curand_init(seed, id, 0, &state[id]);
12 }
13
14 __global__ void do_samples(int N, double mean, double stddev,
15                           double* sum, curandState_t* state)
16 {
17     int id = threadIdx.x + blockIdx.x * blockDim.x;
18     if (id < N) {
19         curandState_t localState = state[id];
20         sum[id] = 0.;
21         for (int i=0; i<100; ++i)
22             sum[id] += curand_log_normal_double(&localState, mean, stddev);
23         state[id] = localState;

```

```
24    }
25 }
26 /*
27 int main( int argc , char* argv[] )
28 Example generating random numbers with exponential
29 distribution
30
31 Inputs: argc should be 3
32 argv[1]: Number of samples to generate
33 argv[2]: Mean of the distribution
34 argv[3]: Standard deviation of the distribution
35
36 Outputs: the initial data and its derivative .
37 */
38
39 int main(int argc , char* argv [] ) {
40
41 // Choose the GPU card
42 cudaDeviceProp prop;
43 int dev;
44 memset(&prop , 0 , sizeof(cudaDeviceProp));
45 prop.multiProcessorCount = 13;
46 cudaChooseDevice(&dev , &prop);
47 cudaSetDevice(dev);
48
49 // Get maximum thread count for the device
50 cudaGetDeviceProperties(&prop , dev);
51 int num_threads = prop.maxThreadsPerBlock ;
52
53 // Get number of samples
54 int N = atoi(argv [1]);
55 int num_blocks = N/num_threads + (N%num_threads ? 1 : 0);
56
57 // Get parameters for the log-normal distribution
58 double mean = atof(argv [2]);
59 double stddev = atof(argv [3]);
60
61 // Allocate space for the kernel functions
62 double *x = (double*)malloc(N*sizeof(double));
63 double *dev_x;
64 cudaMalloc((void**)&dev_x , N*sizeof(double));
65
66 // Set up a random number generator state for each thread
67 curandState_t* dev_state;
68 cudaMalloc((void**)&dev_state , N*sizeof(curandState_t));
69
70 // Seed the generators
71 setup_kernel<<<num_blocks , num_threads>>>(N, time(NULL) , dev_state);
72
73 // Convert to exponential distribution
74 do_samples<<<num_blocks , num_threads>>>(N, mean , stddev , dev_x ,
75 dev_state);
76
77 // Copy device data to host
78 cudaMemcpy(x , dev_x , N*sizeof(double) , cudaMemcpyDeviceToHost);
79
80 // Store result
81 FILE* outfile = fopen("Example26.4.out" , "w");
82 fwrite(x , sizeof(double) , N , outfile);
83 fclose(outfile);
84
85
86
```

```

87 // Clean up the allocated memory
88 cudaFree( dev_state );
89 cudaFree( dev_x );
90 free( x );
91 return 0;
92 }
```

To begin, the maximum number of threads available per block is retrieved on Line 52 using the function `cudaGetDeviceProperties` introduced in Section 22.2. If  $N$  sums are to be generated in this example, then the number of blocks and threads necessary to compute  $N$  values is determined on Line 57.

The setup kernel begins on Line 7, where an array of random generator states, of type `curandState_t`, are initialized, one for each block/thread. The kernel function

```
curand_init( long long int seed,
             unsigned long long int index,
             unsigned long long int offset,
             curandState_t* state )
```

is used to initialize the random number generator state so that every thread will have a unique starting state. The seed value will be the same for each process, but this function will automatically increment the seed by the index so that each thread will have a different resulting sequence. To make this possible, a different generator state of type `curandState_t` is created for each index, which are allocated on Line 70. The offset is used to first run off `offset` random values before using the generator. Once all the data is allocated, the setup kernel function is called on Line 73.

Once the generator states are initialized, the kernel function can call the generator as on Line 22. In this example a log-normal distributed random variable is requested, and the log-normal distribution requires two parameters, the mean and the standard deviation. The parameter values are passed through the argument list to the kernel function. For purposes of speed, the state variable for the given thread is copied from global into local memory space on Line 19. If this is the only time that this kernel will be called with this thread, then no further action would be necessary, but if this kernel is to be used again later, then `localState` is saved back into the global `state` array as on Line 23. If the local state is not saved back into global space, then the kernel will generate the same sequence of values every time it is called because the random number generator state didn't get updated.

Finally, the array of states are released on Line 88.

## 26.3 • cuFFT

The standard CUDA package also includes an FFT library that is based on the FFTW library discussed in other parts of this book. There are some substantive differences, but for the most part it works in the same way. One key difference is that it assumes that the input and output will be on the device; there is no CPU interface like the MAGMA package. Example 26.5 demonstrates the basic use of the library to generate an FFT.

### Example 26.5.

```

1 #include <stdio.h>
2 #include <cufft.h> // This is the header file for the CUDA FFT library
3
```

```

4 #ifndef M_PI
5 #define M_PI 3.1415926535897932384626433832795
6 #endif
7
8 __global__ void rescale(cufftDoubleComplex* a, double f);
9
10 /*
11 int main(int argc, char* argv[])
12
13 The main program takes no arguments and creates a data set to run
14 through the FFT server and outputs the results from doing a forward
15 and inverse transform.
16
17 Inputs: none
18
19 Output: Sample output of the FFT server
20 */
21
22 int main(int argc, char* argv[]) {
23
24 // Number of collocation points in the data set
25 const int N = 256;
26 // Host memory for initializing the input data and retrieving the
27 // output
28 cufftDoubleComplex *u
29     = (cufftDoubleComplex *)malloc(N*sizeof(cufftDoubleComplex));
30 cufftDoubleComplex *a
31     = (cufftDoubleComplex *)malloc(N*sizeof(cufftDoubleComplex));
32
33 double dx = 2*M_PI/N; // space step size
34
35 // Initialize the data
36 int i;
37 for (int i=0; i<N; ++i) {
38     u[i].x = sin(i*dx);
39     u[i].y = 0.;
40 }
41
42 // Create the device memory for running the transform
43 cufftDoubleComplex* dev_u;
44 cufftDoubleComplex* dev_a;
45 cudaMalloc((void**)&dev_u, sizeof(cufftDoubleComplex)*N);
46 cudaMalloc((void**)&dev_a, sizeof(cufftDoubleComplex)*N);
47
48 // Copy the input data to device global memory
49 cudaMemcpy(dev_u, u, N*sizeof(cufftDoubleComplex),
50           cudaMemcpyHostToDevice);
51
52 // Create a FFT plan, this is similar to the FFTW plan discussed in
53 // the MPI section.
54 // This plan is for a 1D double complex to double complex transform
55 // (Z = double complex)
56 cufftHandle plan;
57 cufftPlan1d(&plan, N, CUFFT_Z2Z, 1);
58
59 // Carry out the forward FFT. Everything is done in device memory
60 cufftExecZ2Z(plan, dev_u, dev_a, CUFFT_FORWARD);
61
62 // The forward transform rescales the data so that it's multiplied by
63 // N/2, correct it with a rescaling kernel
64 rescale<<<1, N>>>(dev_a, 2./N);
65
66 // Wait for the stream to be completed before retrieving the data

```

```

67    cudaDeviceSynchronize();
68
69    // Retrieve the data from the device
70    cudaMemcpy(a, dev_a, N*sizeof(cufftDoubleComplex),
71                           cudaMemcpyDeviceToHost);
72
73    // print out the results
74    for (i=0; i<N; ++i) {
75        printf ("%d: %5.1f + %5.1fi\n", i, a[i].x, a[i].y);
76    }
77
78    printf ("----\n");
79
80    // Now do the inverse transform
81    cufftExecZ2Z(plan, dev_a, dev_u, CUFFT_INVERSE);
82    // The inverse transform does a different scaling
83    rescale<<<1, N>>>(dev_u, 0.5);
84
85    // Again wait for the stream to finish
86    cudaDeviceSynchronize();
87
88    // retrieve the results of the inverse transform
89    cudaMemcpy(u, dev_u, N*sizeof(cufftDoubleComplex),
90                           cudaMemcpyDeviceToHost);
91
92    // print the results
93    for (i=0; i<N; ++i)
94        printf ("%d: %10.5f + %10.5fi == %10.5f\n", i, u[i].x, u[i].y,
95                                         sin(i*dx));
96
97    // destroy the plan
98    cufftDestroy(plan);
99    // release the memory
100   cudaFree(dev_u);
101   cudaFree(dev_a);
102   free(u);
103   free(a);
104   return 0;
105 }
106
107 // kernel for rescaling the results
108 __global__ void rescale(cufftDoubleComplex* dev_u, double f)
109 {
110     dev_u[threadIdx.x].x = dev_u[threadIdx.x].x*f;
111     dev_u[threadIdx.x].y = dev_u[threadIdx.x].y*f;
112 }
```

The CUDA FFT library defines two complex number types, `cufftComplex` and `cufftDoubleComplex`, for single and double precision, respectively. These are the same as the CUDA library-defined types `cuComplex` and `cuDoubleComplex`. All these CUDA-defined complex-valued variable types use a struct to define the real and imaginary parts so that a variable `u` has real part `u.x` and imaginary part `u.y` as on Line 94 in Example 26.5.

The strategy for building the FFT is similar to how FFTW works in Section 7.2. The first step is to create a plan that will optimize the computation of the FFT for the given computing resources, in this case the GPU. The plan is of type `cufftHandle_t` and a one-dimensional plan is created on Line 57. The type of plan is a CUFFT\_Z2Z plan. The Z2Z means a transform that takes as input the type `cufftDoubleComplex` and returns

`cufftDoubleComplex` as output. The naming convention follows the LAPACK-type conventions where

Char	Type
S	float
D	double
C	<code>cufftComplex</code>
Z	<code>cufftDoubleComplex</code>

In this example, there are  $N$  collocation points of type `cufftDoubleComplex` as input and it will produce coefficients with `cufftDoubleComplex` output. The last argument is for batch processing. In this example, there is only one data set of length  $N$ , but if there were  $M$  data sets of length  $N$ , then the data would be given as contiguous  $MN$  data points and the last argument would be  $M$ . This way one call can do multiple FFTs. This is very handy when, for example, each row of a grid is to have the FFT applied; then the full array is given as data and the batch number would be the number of rows.

Once the plan is set up, it can be executed either forward or backward as is done on Lines 60 and 81. The first argument is the `plan`, followed by the input data, the output data, and the direction of the transform. The input and output arrays can be the same, in which case the output overwrites the input. This illustrates one significant difference between cuFFT and the FFTW library in Section 7.2, where in the latter case separate plans were required for the forward and backward transforms.

Like the FFTW package, the scaling is not such that the forward and inverse transforms are inverses of each other. After a forward transform, the data is multiplied by a factor of  $N/2$ . The inverse transform multiplies the result by an additional factor of 2. Therefore, if using this package to do a pseudospectral method calculation, the process of doing a forward followed by a backward transform will require the result to be divided by  $N$ . To correct for these, the rescaling could be done at the end, but here the rescaling is done in two stages using a simple kernel on Lines 64 and 83.

The package operates asynchronously using multiple streams, so before using the data, the function `cudaDeviceSynchronize` should be called to make sure the FFT is completed.

Finally, the plan is disposed of using the `cufftDestroy` function as shown on Line 98.

### 26.3.1 • Two-Dimensional Transforms

Transforming data in two and higher dimensions is accomplished in very much the same way as described for one dimension above. It is important to remember that like the FFTW libraries described elsewhere in this text, data is stored in column-major order. This will be particularly important when computing spectral derivatives in the pseudospectral method applications in Chapter 37. A two-dimensional plan `p` for an  $M \times N$  grid to compute the forward transform is constructed using the function `cufftPlan2d`:

```
cufftHandle plan;
cufftPlan2d(&plan, M, N, CUFFT_Z2Z);
```

The plan is executed using the same function `cufftExecZ2Z` as before:

```
cufftExecZ2Z(plan, dev_input, dev_output, CUFFT_FORWARD);
```

Here the input data is `dev_input` and the output data is in `dev_output`.

The order of the coefficients will have the same structure in each dimension as for the one-dimensional results so that the array is organized as below:

$a_{0,0}$	$a_{0,1}$	$\cdots$	$a_{0,\frac{N}{2}}$	$a_{0,1-\frac{N}{2}}$	$a_{0,2-\frac{N}{2}}$	$\cdots$	$a_{0,-1}$
$a_{1,0}$	$a_{1,1}$	$\cdots$	$a_{1,\frac{N}{2}}$	$a_{1,1-\frac{N}{2}}$	$a_{1,2-\frac{N}{2}}$	$\cdots$	$a_{1,-1}$
$\vdots$	$\vdots$		$\vdots$	$\vdots$	$\vdots$		$\vdots$
$a_{\frac{M}{2},0}$	$a_{\frac{M}{2},1}$	$\cdots$	$a_{\frac{M}{2},\frac{N}{2}}$	$a_{\frac{M}{2},1-\frac{N}{2}}$	$a_{\frac{M}{2},2-\frac{N}{2}}$	$\cdots$	$a_{\frac{M}{2},-1}$
$a_{1-\frac{M}{2},0}$	$a_{1-\frac{M}{2},1}$	$\cdots$	$a_{1-\frac{M}{2},\frac{N}{2}}$	$a_{1-\frac{M}{2},1-\frac{N}{2}}$	$a_{1-\frac{M}{2},2-\frac{N}{2}}$	$\cdots$	$a_{1-\frac{M}{2},-1}$
$a_{2-\frac{M}{2},0}$	$a_{2-\frac{M}{2},1}$	$\cdots$	$a_{2-\frac{M}{2},\frac{N}{2}}$	$a_{2-\frac{M}{2},1-\frac{N}{2}}$	$a_{2-\frac{M}{2},2-\frac{N}{2}}$	$\cdots$	$a_{2-\frac{M}{2},-1}$
$\vdots$	$\vdots$		$\vdots$	$\vdots$	$\vdots$		$\vdots$
$a_{-1,0}$	$a_{-1,1}$	$\cdots$	$a_{-1,\frac{N}{2}}$	$a_{-1,1-\frac{N}{2}}$	$a_{-1,2-\frac{N}{2}}$	$\cdots$	$a_{-1,-1}$

### 26.3.2 • Transforming Real-Valued Data

When using this package to transform real-valued data, some savings can be gained by using the plan functions that transform between real- and complex-valued data. In this case, the forward and backward plans are created like this:

```
cufftHandle fplan;
cufftPlan2d(&fplan , M, N, CUFFT_D2Z);

cufftHandle bplan;
cufftPlan2d(&bplan , M, N, CUFFT_Z2D);
```

Here the first plan, `fplan`, is for a forward transform, and the second plan, `bplan`, is for a backward transform. The plans are executed using the corresponding functions `cufftExecD2Z` and `cufftExecZ2D`, respectively:

```
cufftExecD2Z(fplan , dev_input , dev_output);
cufftExecZ2D(bplan , dev_input , dev_output);
```

For the one-dimensional transform, only the complex coefficients  $a_0, \dots, a_{N/2}$  are computed because the remaining coefficients are the complex conjugates of these. In higher dimensions, only the last dimension is cut in half, so that the coefficients are arrayed as shown below:

$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$\cdots$	$a_{0,\frac{N}{2}}$
$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$\cdots$	$a_{1,\frac{N}{2}}$
$\vdots$	$\vdots$	$\vdots$		$\vdots$
$a_{\frac{M}{2},0}$	$a_{\frac{M}{2},1}$	$a_{\frac{M}{2},2}$	$\cdots$	$a_{\frac{M}{2},\frac{N}{2}}$
$a_{1-\frac{M}{2},0}$	$a_{1-\frac{M}{2},1}$	$a_{1-\frac{M}{2},2}$	$\cdots$	$a_{1-\frac{M}{2},\frac{N}{2}}$
$a_{2-\frac{M}{2},0}$	$a_{2-\frac{M}{2},1}$	$a_{2-\frac{M}{2},2}$	$\cdots$	$a_{2-\frac{M}{2},\frac{N}{2}}$
$\vdots$	$\vdots$	$\vdots$		$\vdots$
$a_{-1,0}$	$a_{-1,1}$	$a_{-1,2}$	$\cdots$	$a_{-1,\frac{N}{2}}$

### 26.3.3 • Compiling and Linking with cufft Library

When building code to use the `cufft` library, use the NVIDIA compiler as usual with the library `-lcufft`.

## 26.4 • cuSPARSE

The cuSPARSE library provides some useful basic linear algebra operations that are for matrices and vectors stored in a sparse format. It does not provide a full-blown linear algebra package like LAPACK but nonetheless can be useful for large linear systems.

The library provides a collection of functions that follow a naming scheme given by

```
cusparse<t><matrix format><operation><output matrix format>
```

where `<t>` is the data type, `<matrix format>` is the type of sparse format of the matrix, and `<operation>` is the operation to be performed. The options for these labels are shown in Table 26.1. For example, the function `cusparseDcsrsv` is a matrix vector product using double precision floating point data and where the matrix is stored in compressed sparse row matrix form. Example 26.6 illustrates the use of this function.

**Table 26.1.** Options for the naming convention for (a) data type, (b) matrix format, and (c) operation to be performed.

<code>&lt;t&gt;</code>	Type
S	float
D	double
C	cuComplex
Z	cuDoubleComplex

(a) Data formats

Label	Matrix Format
coo	Coordinate format
csr	Compressed sparse row
csc	Compressed sparse column
hyb	Hybrid format

(b) Available matrix formats

Label	Operation
axpyi	$y \leftarrow y + ax$
doti	$\text{result} \leftarrow y^T x$
dotci	$\text{result} \leftarrow y^H x$
gthr	convert dense vector to sparse vector
gthrz	convert dense vector to sparse vector
roti	apply Givens rotation
sctr	convert sparse vector to dense vector
mv	$y \leftarrow aAx + by$
sv	solve triangular system $Ay = ax$
mm	$C \leftarrow aAB + bC$
sm	solve triangular system $AY = aX$

(c) Available operations

The functions work exclusively on device memory; there is no CPU interface as for MAGMA. Furthermore, all the functions operate asynchronously, so to ensure completion, use `cudaDeviceSynchronize`.

### 26.4.1 • Sparse Formats

#### Vector Sparse Format

The vector sparse format is pretty simple. It comprises a data vector and an index vector. The data vector contains only the nonzero entries in the vector, and the index vector gives the indices of those nonzero entries in the full vector. For example,

Dense format: 

1	0	2	3	0	0	4	0	5
---	---	---	---	---	---	---	---	---

Data vector: 

1	2	3	4	5
---	---	---	---	---

Index vector: 

0	2	3	6	8
---	---	---	---	---

#### Coordinate Matrix Format

The coordinate matrix format (coo label) consists of three vectors. The first vector contains the nonzero entries in the matrix by rows. The second two vectors are integer vectors of the same length as the data vector. One contains the row coordinate (zero-based), and the other contains the column coordinate (zero-based) for each of the non-zero entries in the data vector. For example:

Dense format: 

1	0	0	2	0	0	3	0	0
0	4	0	0	5	0	0	0	0
0	0	6	0	0	7	0	0	8
9	0	0	1	0	0	0	0	0
0	2	0	0	3	0	0	0	4

Data vector: 

1	2	3	4	5	6	7	8	9	1	2	3	4
---	---	---	---	---	---	---	---	---	---	---	---	---

Row Index vector: 

0	0	0	1	1	2	2	2	3	3	4	4	4
---	---	---	---	---	---	---	---	---	---	---	---	---

Column Index vector: 

0	3	6	1	4	2	5	8	0	3	1	4	8
---	---	---	---	---	---	---	---	---	---	---	---	---

#### Compressed Sparse Row Matrix Format

The compressed sparse row matrix format (csr label) consists of three vectors. The first vector contains the nonzero entries in the matrix by rows. The second two vectors are integer vectors: the row index vector and the column index vector. The row index vector will have length one more than the number of rows of the matrix. The column index vector will have the same length as the data vector. The row index vector gives the index in the data vector where each row's data begins (zero-based) and the last entry in

this vector contains the total number of nonzero entries. The column index vector gives the column index within the row where the nonzero entries are listed. For example:

Dense format:	<table border="1"><tr><td>1</td><td>0</td><td>0</td><td>2</td><td>0</td><td>0</td><td>3</td><td>0</td><td>0</td></tr><tr><td>0</td><td>4</td><td>0</td><td>0</td><td>5</td><td>0</td><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>0</td><td>6</td><td>0</td><td>0</td><td>7</td><td>0</td><td>0</td><td>8</td></tr><tr><td>9</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>2</td><td>0</td><td>0</td><td>3</td><td>0</td><td>0</td><td>0</td><td>4</td></tr></table>	1	0	0	2	0	0	3	0	0	0	4	0	0	5	0	0	0	0	0	0	6	0	0	7	0	0	8	9	0	0	1	0	0	0	0	0	0	2	0	0	3	0	0	0	4
1	0	0	2	0	0	3	0	0																																						
0	4	0	0	5	0	0	0	0																																						
0	0	6	0	0	7	0	0	8																																						
9	0	0	1	0	0	0	0	0																																						
0	2	0	0	3	0	0	0	4																																						
Data vector:	<table border="1"><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>1</td><td>2</td><td>3</td><td>4</td></tr></table>	1	2	3	4	5	6	7	8	9	1	2	3	4																																
1	2	3	4	5	6	7	8	9	1	2	3	4																																		
Row Index vector:	<table border="1"><tr><td>0</td><td>3</td><td>5</td><td>8</td><td>10</td><td>13</td></tr></table>	0	3	5	8	10	13																																							
0	3	5	8	10	13																																									
Column Index vector:	<table border="1"><tr><td>0</td><td>3</td><td>6</td><td>1</td><td>4</td><td>2</td><td>5</td><td>8</td><td>0</td><td>3</td><td>1</td><td>4</td><td>8</td></tr></table>	0	3	6	1	4	2	5	8	0	3	1	4	8																																
0	3	6	1	4	2	5	8	0	3	1	4	8																																		

### Compressed Sparse Column Matrix Format

The compressed sparse column matrix format (csc label) is the same as the compressed sparse row matrix format except the roles of the row and column index vectors are reversed. The column index vector gives the starting point in the data vector for each column's data, and the row index gives the row for each entry. For example:

Dense format:	<table border="1"><tr><td>1</td><td>0</td><td>0</td><td>2</td><td>0</td><td>0</td><td>3</td><td>0</td><td>0</td></tr><tr><td>0</td><td>4</td><td>0</td><td>0</td><td>5</td><td>0</td><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>0</td><td>6</td><td>0</td><td>0</td><td>7</td><td>0</td><td>0</td><td>8</td></tr><tr><td>9</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>2</td><td>0</td><td>0</td><td>3</td><td>0</td><td>0</td><td>0</td><td>4</td></tr></table>	1	0	0	2	0	0	3	0	0	0	4	0	0	5	0	0	0	0	0	0	6	0	0	7	0	0	8	9	0	0	1	0	0	0	0	0	0	2	0	0	3	0	0	0	4
1	0	0	2	0	0	3	0	0																																						
0	4	0	0	5	0	0	0	0																																						
0	0	6	0	0	7	0	0	8																																						
9	0	0	1	0	0	0	0	0																																						
0	2	0	0	3	0	0	0	4																																						
Data vector:	<table border="1"><tr><td>1</td><td>9</td><td>4</td><td>2</td><td>6</td><td>2</td><td>1</td><td>5</td><td>3</td><td>7</td><td>3</td><td>8</td><td>4</td></tr></table>	1	9	4	2	6	2	1	5	3	7	3	8	4																																
1	9	4	2	6	2	1	5	3	7	3	8	4																																		
Row Index vector:	<table border="1"><tr><td>0</td><td>3</td><td>1</td><td>4</td><td>2</td><td>0</td><td>3</td><td>1</td><td>4</td><td>2</td><td>0</td><td>2</td><td>4</td></tr></table>	0	3	1	4	2	0	3	1	4	2	0	2	4																																
0	3	1	4	2	0	3	1	4	2	0	2	4																																		
Column Index vector:	<table border="1"><tr><td>0</td><td>2</td><td>4</td><td>5</td><td>7</td><td>9</td><td>10</td><td>10</td><td>11</td><td>13</td></tr></table>	0	2	4	5	7	9	10	10	11	13																																			
0	2	4	5	7	9	10	10	11	13																																					

### 26.4.2 • Matrix Vector Multiplication

Example 26.6 gives a short demonstration of how to do sparse matrix vector multiplication using the package.

#### Example 26.6.

```

1 #include <stdio.h>
2 #include <cusparse.h> // this is the header file for the cuSPARSE
3           library
4 /*
5 int main( int argc , char* argv[] )
6 {
7     The main program takes no arguments and creates a tridiagonal matrix and
8     then multiplies it by a simple sparse vector .

```

```
9
10 Inputs: none
11
12 Output: prints the expression computed in matrix format.
13 */
14
15 int main(int argc, char* argv[]) {
16
17     // Dimensions of the matrix for this example
18     const int N = 9;
19
20     // The data vector for compressed sparse row format
21     double* Aval = (double*)malloc(3*N*sizeof(double));
22
23     // The row index and column index vectors for the compressed sparse
24     // row format
25     int* Arowptr = (int*)malloc((N+1)*sizeof(int));
26     int* Acolind = (int*)malloc(3*N*sizeof(int));
27
28     // The data vector for a sparse vector
29     double* xval = (double*)malloc(N*sizeof(double));
30
31     // The index vector for a sparse vector
32     int* xind = (int*)malloc(N*sizeof(int));
33
34     // A dense solution vector.
35     double* y = (double*)malloc(N*sizeof(double));
36
37     // build the matrix A with -2 down the diagonal and ones on the
38     // off diagonals in compressed sparse row format
39     int row;
40     int index = 0;
41     for (row=0; row<N; ++row) {
42         Arowptr[row] = index;
43         if (row > 0) {
44             Aval[index] = 1.;
45             Acolind[index++] = row-1;
46         }
47         Aval[index] = -2.;
48         Acolind[index++] = row;
49         if (row < N-1) {
50             Aval[index] = 1.;
51             Acolind[index++] = row+1;
52         }
53     }
54     Arowptr[N] = index;
55     int nnz = index;
56
57     // build the vector x as a sparse vector with
58     // alternating 1 0 2 0 3 ...
59     index = 0;
60     for (row=0; row<N; row += 2) {
61         xval[index] = (row/2)+1;
62         xind[index++] = row;
63     }
64
65     // All operations take place on the device, so must create versions
66     // of all the input and output data on the device and copy the
67     // input data to the device.
68     // Names correspond to the host variable names
69     double* dev_Aval;
70     int* dev_Arowptr;
71     int* dev_Acolind;
```

```

72  double* dev_xval ;
73  int* dev_xind ;
74  double* dev_x ;
75  double* dev_y ;
76
77  // Allocate device memory
78  cudaMalloc((void**)&dev_Aval , 3*N*sizeof(double));
79  cudaMalloc((void**)&dev_Arowptr , (N+1)*sizeof(int));
80  cudaMalloc((void**)&dev_Acolind , 3*N*sizeof(int));
81  cudaMalloc((void**)&dev_xval , N*sizeof(double));
82  cudaMalloc((void**)&dev_xind , N*sizeof(int));
83  cudaMalloc((void**)&dev_x , N*sizeof(double));
84  cudaMalloc((void**)&dev_y , N*sizeof(double));
85
86  // Copy host data to the device
87  cudaMemcpy(dev_Aval , Aval , 3*N*sizeof(double) ,
88             cudaMemcpyHostToDevice);
89  cudaMemcpy(dev_Arowptr , Arowptr , (N+1)*sizeof(int) ,
90             cudaMemcpyHostToDevice);
91  cudaMemcpy(dev_Acolind , Acolind , 3*N*sizeof(int) ,
92             cudaMemcpyHostToDevice);
93  cudaMemcpy(dev_xval , xval , N*sizeof(double) , cudaMemcpyHostToDevice);
94  cudaMemcpy(dev_xind , xind , N*sizeof(int) , cudaMemcpyHostToDevice);
95
96  // Initialize the cuSPARSE library and get the environment context
97  cusparseHandle_t handle = NULL;
98  cusparseCreate(&handle);
99
100 // Create a matrix description for our sparse matrix
101 cusparseMatDescr_t descr = NULL;
102 cusparseCreateMatDescr(&descr);
103 cusparseSetMatType(descr , CUSPARSE_MATRIX_TYPE_GENERAL);
104 cusparseSetMatIndexBase(descr ,CUSPARSE_INDEX_BASE_ZERO);
105
106 // Convert the sparse vector into a dense vector
107 cusparseDscctr(handle , (N+1)/2 , dev_xval , dev_xind , dev_x ,
108                 CUSPARSE_INDEX_BASE_ZERO);
109
110 // Multiple the matrix A by the vector x
111 double one = 1.;
112 double zero = 0.;
113 cusparseDcsrsv(handle , CUSPARSE_OPERATION_NON_TRANSPOSE , N , N , nnz ,
114                  &one , descr , dev_Aval , dev_Arowptr , dev_Acolind ,
115                  dev_x , &zero , dev_y);
116
117 // Ensure the operation is complete.
118 cudaDeviceSynchronize();
119
120 // Copy the results on the device back to the host
121 cudaMemcpy(y , dev_y , N*sizeof(double) , cudaMemcpyDeviceToHost);
122
123 // Print the results
124 int i , j , mindex = 0 , vindex = 0;
125 for (i=0; i<N; ++i) {
126     printf("[ ");
127     for (j=0; j<N; ++j) {
128         if (mindex < Arowptr[i+1] && j==Acolind[mindx]) {
129             printf("%5.1f" , Aval[mindx++]);
130         } else {
131             printf("%5.1f" , 0.);
132         }
133     }
134     printf("] [");

```

```

135     if (i == xind[vindex]) {
136         printf("%5.1f]", xval[vindex++]);
137     } else {
138         printf("%5.1f]", 0.);
139     }
140     if (i == N/2) {
141         printf(" = [");
142     } else {
143         printf("    [");
144     }
145     printf("%5.1f]\n", y[i]);
146 }
147
148 return 0;
149 }
```

Compare the format description above to the construction of the compressed sparse row matrix format on Lines 39–54. In addition to the raw data for the matrix storage, a matrix description object of type `cusparseMatDescr_t` is required for the functions in the library. The matrix in this example has no special structure to be exploited and uses zero-based indexing. (One-based indexing is also permitted.) The matrix description is created and set on Lines 101–104.

The library itself must also be initialized on Line 98, which is done by creating the environment context of type `cusparseHandle_t` and using `cusparseCreate`.

The matrix vector multiply requires the vector to be in dense format, so the function `cusparseDscstr` is used to convert the sparse vector into a dense vector on Line 107. The matrix vector multiplication is computed with the function `cusparseDcsrsv` on Line 113 and followed by `cudaDeviceSynchronize` to ensure that all operations are complete.

The output of this program is shown below, so that the structure of data is easier to understand:

```

[ -2.0  1.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0] [ 1.0]   [ -2.0]
[ 1.0 -2.0  1.0  0.0  0.0  0.0  0.0  0.0  0.0] [ 0.0]   [ 3.0]
[ 0.0  1.0 -2.0  1.0  0.0  0.0  0.0  0.0  0.0] [ 2.0]   [ -4.0]
[ 0.0  0.0  1.0 -2.0  1.0  0.0  0.0  0.0  0.0] [ 0.0]   [ 5.0]
[ 0.0  0.0  0.0  1.0 -2.0  1.0  0.0  0.0  0.0] [ 3.0] = [ -6.0]
[ 0.0  0.0  0.0  0.0  1.0 -2.0  1.0  0.0  0.0] [ 0.0]   [ 7.0]
[ 0.0  0.0  0.0  0.0  0.0  1.0 -2.0  1.0  0.0] [ 4.0]   [ -8.0]
[ 0.0  0.0  0.0  0.0  0.0  0.0  1.0 -2.0  1.0] [ 0.0]   [ 9.0]
[ 0.0  0.0  0.0  0.0  0.0  0.0  0.0  1.0 -2.0] [ 5.0]   [ -10.0]
```

### 26.4.3 • Tridiagonal Solver

Another useful purpose of the sparse matrix package is to solve linear systems. It does not provide a general purpose solver, but it does offer a solver for upper or lower triangular and also tridiagonal matrices. Since tridiagonal systems arise in the solution of the projects in Part VI, the solver is discussed here.

Similar to the LAPACK tridiagonal solver, the matrix is specified by three arrays representing the lower, main, and upper diagonals of the matrix, but there is a small difference in the details. In the LAPACK solver, the three diagonals are stored so that

the first entry in the array for the lower diagonal corresponds to row 2, column 1, of the matrix, i.e., the first entry of the lower diagonal matrix. For the cuSPARSE solver, the index in the array must conform to the row of the matrix. That means the first entry of the array for the lower diagonal should be set to zero because it is outside the matrix. It may be easier to see the difference by comparing them side by side in Figure 26.2. The obvious difference is that the cuSPARSE version has the array index correspond to the row of the matrix, not the distance along the diagonal.

## LAPACK configuration:

$$\begin{array}{ccccccccc} d_m[0] & d_u[0] & & 0 & & & & & \\ d_i[0] & d_m[1] & d_u[1] & & & & & & \\ 0 & d_i[1] & & \ddots & & & & & \\ & & & & \ddots & & & & \\ & & & & & d_u[N-3] & 0 & & \\ & & & & d_i[N-3] & d_m[N-2] & d_u[N-2] & & \\ & & & & 0 & d_i[N-2] & d_m[N-1] & & \end{array}$$

### cuSPARSE configuration:

$$(0 =) d_l[0] \left| \begin{array}{cccccc} d_m[0] & d_u[0] & 0 & & & \\ d_l[1] & d_m[1] & d_u[1] & \ddots & & \\ 0 & d_l[2] & \ddots & \ddots & \ddots & \\ & & \ddots & \ddots & \ddots & \\ & & & d_l[N-2] & d_m[N-2] & d_u[N-2] \\ & & & 0 & d_l[N-1] & d_m[N-1] \end{array} \right| d_u[N-1] (= 0)$$

**Figure 26.2.** Comparison of arrangement of data for the tridiagonal solver in the LAPACK and cuSPARSE libraries. For the cuSPARSE library each diagonal has the same length and the first entry of the lower diagonal and the last entry of the upper diagonal must be zero. This means the indices for the lower diagonal are shifted by one compared to the solver in LAPACK.

Another difference between the two, which turns out to be fairly significant, is that the cuSPARSE solver does not alter the diagonals. This may sound convenient, but it is not ideal. The LAPACK solver can be used to separate the factorization from the solution so that repeated solves with the same matrix can be done quicker, but at the cost that the diagonals get altered during the factorization. The cuSPARSE solver requires it to redo the factorization every application, resulting in slower code. In some cases, the MAGMA dense matrix solver can be twice as fast as the cuSPARSE tridiagonal solver for the same problem, so it is not necessarily a better idea to use the tridiagonal solver from cuSPARSE. Example 26.7 shows how the tridiagonal solver is used.

### **Example 26.7.**

```
1 #include <stdio.h>
2 #include <cusparse.h>
3 #include <stdlib.h>
4 #include <cuda.h>
5
6 // Set up matrix diagonals
```

```

7  __global__ void init ( int N, double *dl, double *d, double *du)
8 {
9     int idx = threadIdx.x + blockIdx.x * blockDim.x;
10
11    if ( idx == 0 ) { //first row, dl[0] = 0
12        dl[idx] = 0.0;
13        du[idx] = -1.0 ;
14        d[idx] = 2.0;
15    } else if ( idx == N-1 ) { // last row, du[N-1]=0
16        dl[idx] = -1.0 ;
17        du[idx] = 0. ;
18        d[idx] = 2.0;
19    } else if ( idx < N-1 ) { // all the middle rows
20        dl[idx] = -1.0;
21        du[idx] = -1.0;
22        d[idx] = 2.0;
23    }
24 }
25
26 /*
27  int main( int argc , char* argv[])
28
29 Example program for using the tridiagonal solver in the
30 cuSPARSE library. This program solves the NxN problem Ax = b
31 where
32      [2   -1           ]      [ 1   ]
33      [-1   2   -1       ]      [ 2   ]
34 A =  [   -1   2   -1       ]  b = [ 3   ]
35      [       ...        ]      [   ...  ]
36      [           -1   2   -1]      [ N-1  ]
37      [                 -1   2]      [  N   ]
38
39 Input: argc should be 2
40 argv[1]: Dimensions of A, b
41
42 Output
43     solution vector is printed
44 */
45 int main( int argc , char* argv[])
46 {
47     // Pick the GPU device
48     cudaDeviceProp prop;
49     int dev;
50     memset(&prop , 0, sizeof(cudaDeviceProp));
51     prop.multiProcessorCount = 13;
52     cudaChooseDevice(&dev , &prop);
53     cudaSetDevice(dev);
54     cudaGetDeviceProperties(&prop , dev);
55
56     int N = atoi(argv[1]);
57
58     // Initialize cuSPARSE library
59     cusparseStatus_t status;
60     cusparseHandle_t handle=0;
61     status= cusparseCreate(&handle);
62
63     // Allocate Memory on host(CPU) and device (GPU)
64     double *x = (double*) malloc(N*sizeof(double));
65     double *b = (double*) malloc(N*sizeof(double));
66     double *dev_dl , *dev_d , *dev_du , *dev_b;
67     cudaMalloc((void**)&dev_dl , N*sizeof(double));
68     cudaMalloc((void**)&dev_d , N*sizeof(double));
69     cudaMalloc((void**)&dev_du , N*sizeof(double));

```

```

70    cudaMalloc((void**)&dev_b, N*sizeof(double));
71
72    // initialize the three diagonals
73    if (N > 256) {
74        int num_blocks = N/256 + (N%256 > 0 ? 1 : 0);
75        init<<<num_blocks, 256>>>(N, dev_dl, dev_d, dev_du);
76    } else
77        init<<<1, N>>>(N, dev_dl, dev_d, dev_du);
78
79    // initialize the B vector and put it on the device
80    int i;
81    for (i=0; i<N; ++i)
82        b[i] = i+1;
83    cudaMemcpy(dev_b, b, N*sizeof(double), cudaMemcpyHostToDevice);
84
85    // Solve the tridiagonal system for a single right-hand side
86    // Solution is placed in dev_b, diagonals are left unchanged
87    status = cusparseDgtsv(handle, N, 1, dev_dl, dev_d, dev_du, dev_b, N);
88    if (status != CUSPARSE_STATUS_SUCCESS) {
89        printf("CUSPARSE solver failed");
90    }
91
92    // Retrieve the solution from the device
93    cudaMemcpy(x, dev_b, N*sizeof(double), cudaMemcpyDeviceToHost);
94
95    // Print the solution
96    for (i=0; i<N; ++i)
97        printf("%f\n", x[i]);
98
99    /*Free the Memory*/
100   cusparseDestroy(handle);
101   cudaFree(dev_dl);
102   cudaFree(dev_d);
103   cudaFree(dev_du);
104   cudaFree(dev_b);
105   free(x);
106   free(b);
107   return 0;
108 }
```

In order to use the cuSPARSE tridiagonal solver, the library requires some setup, which is done on Line 61 with the `cusparseCreate` function. The `cusparseHandle_t` handle that is returned will be used to invoke the solver. The `cusparseStatus_t` type is for checking whether an operation was successful. The resources allocated on the host by creating the `cusparseHandle_t` must also be released using `cusparseDestroy` as on Line 100.

The tridiagonal matrix must be stored on the device. It can be built on the host and then sent to the device, or it can be created in a kernel on the device itself. Example 26.7 uses the latter approach with the `init` kernel listed on Line 7. There, the three diagonal arrays are filled out following the cuSPARSE tridiagonal format described earlier. To that end, note how on Line 12, the lower diagonal entry in the first row is set to zero, and on Line 17, the last entry of the upper diagonal is also set to zero. At the conclusion of this kernel, the three diagonals are ready to be fed to the solver.

The right-hand-side vector must also be in device memory, but in this example, the right-hand side is set on the host and then copied to the device on Line 83. The solver can then be called on Line 87. The solver function has the following declaration:

```
cusparseStatus_t cursparseDgtsv( cusparseHandle_t handle,
                                  unsigned int N,
                                  unsigned int nrhs,
                                  double* dev_dl,
                                  double* dev_d,
                                  double* dev_du,
                                  double* dev_b,
                                  unsigned int N )
```

Here, the sparse matrix system initialized earlier is stored in `handle`. The dimensions of the matrix to be inverted are  $N \times N$ . The solver can be applied to multiple right-hand-side vectors, and the number of vectors stored in `dev_b` is given by `nrhs`. Next, the three diagonals—lower, main, upper—are specified in device memory by `dev_dl`, `dev_d`, `dev_du`, respectively. Similarly the data for the right-hand-side vectors is stored consecutively in `dev_b`. Finally, the last argument should be set to the dimension of the right-hand side, which should also be the same as the matrix and hence should also be set to `N`.

The “D” in the `cusparseDgtsv` name means double precision; it can also handle single precision (“S”), single precision complex (“C”), and double precision complex (“Z”). For these latter two, the types `cuComplex` and `cuDoubleComplex` are provided by the library.

The solver returns a value of type `cusparseStatus_t`, which reports whether the operation was successful. Most functions in the library do the same thing. This is checked on Line 88 to make sure the solver finished the problem successfully. Finally, the solution vector is stored where the right-hand-side data was, and this is copied back to the host on Line 93.

#### 26.4.4 • Linking the Library

The cuSPARSE library is linked into by adding `-lcusparse` to the command when building the executable.

---

## Exercises

- 26.1. Write a kernel to generate an  $N \times N$  array of random values using a uniform distribution in the range  $[0, 1]$ . Verify that no two values are repeated, and if the same seed is used twice, then the full array is exactly the same.
- 26.2. Generate a set of  $N$  points sampled uniformly in the domain  $[-1, 1] \times [-1, 1]$ . The number of sampled points  $M$  that lie inside a circle of radius 1 should be approximately  $M \approx \frac{\pi}{4}N$ , i.e., the area of the circle divided by the area of the square times the number of samples. Thus, an estimate for  $\pi$  would be generated by the expression  $\frac{4M}{N}$ . Try estimating  $\pi$  using this method.
- 26.3. Modify Example 26.1 to solve a tridiagonal system of equations using the GPU interface.
- 26.4. Write a program to generate  $N$  random uniformly distributed values on the device, and then write a kernel that sums all the values. See the discussion in Section 27.1.1.

- 26.5. Add a kernel to Example 26.5 to compute the first derivative of the data in an array of length  $N$  following the discussion in Section 37.2.
- 26.6. Modify Example 26.5 to do a two-dimensional FFT transform. In this case, the plan is constructed by calling the function

```
cufftPlan2d(&plan , M, N, CUFFT_Z2Z);
```

where  $M, N$  are the dimensions of the domain. Aside from the data arrays `in` and `out` being pointers to two-dimensional arrays of dimensions  $M \times N$ , the remainder of the arguments are the same as before. Take as inputs to your program the integer wave modes  $k_1$  and  $k_2$  so that the complex variable  $\text{in}[i][j][0] = \cos((k_1*i+k_2*j)*dx)$  and  $\text{in}[i][j][1] = \sin((k_1*i+k_2*j)*dx)$ . Print the values of the spectral coefficients after the forward transform. Perform the reverse transform and verify that the initial data is recovered. Try different values of  $k_1$  and  $k_2$  and note how the spectral coefficients depend on the values.

## Chapter 27

# Projects for CUDA Programming

The background for the projects below is in Part VI of this book. You should be able to build each of these projects based on the information provided in this text and utilize CUDA to implement GPU versions of these projects. Where appropriate, there are some additional comments specific to CUDA that will help you to understand how to develop your algorithms.

## 27.1 • Random Processes

### 27.1.1 • Monte Carlo Integration

There are three ways to handle Monte Carlo integration using the GPU: (1) use the host interface for the cuRAND library and sum the results on the host, (2) use the device interface and then subdivide the array among the threads where each thread computes a subtotal for its part with the results added together afterward, or (3) use the device interface and create a reduction kernel. Case 1 is very straightforward and simple but really only uses the GPU to generate the random values and nothing else, so it is not of much interest here. The other two cases use the device interface for cuRAND to generate an array of length  $N$  of random values. Suppose also that there are  $B$  blocks with  $T$  threads each available for the calculation.

Case 2: The array is subdivided into  $BT$  subarrays of length  $N/(BT)$ . Each thread applies the function  $f(x)$  to its list of  $N/(BT)$  values and then sums the total, storing it in the first entry of its subarray. A second kernel is used to add the  $BT$  subtotals to get the final result. If  $N \leq (BT)^2$ , then this is definitely the right strategy to use. However, if  $N \gg (BT)^2$ , then case 3 is the better choice.

Case 3: Two kernels are used, one to apply the function, and one to do the final addition reduction. The first kernel is straightforward; it simply applies  $f(x)$  to each entry in the random array storing it back in place. Alternatively, it can also compute the local subtotal and store just the subtotal like for case 2. For simplicity of discussion, assume that the subtotal is not computed in the first kernel. The second kernel requires a bit more discussion and is important enough in various contexts to present here.

CUDA does not provide a general global reduction kernel, but it is easy enough to create one. Given an array of length  $N$  whose entries are to be summed, a reduction kernel is used to compute subtotals repeatedly until it is reduced to a single total. Ex-

ample 24.6 shows how to use a warp shuffle to quickly sum all the values of an array within a warp. A complete reduction kernel would use Example 24.6 to build a more general kernel to sum all the values in an array of length  $N$  by summing all the values within the warp and then storing the result in an array in global memory in the entry corresponding to the warp ID

Program the assignment in Section 34.3.1 using  $N$  samples, where  $N$  is an input parameter. You should write at least three kernels: one to set up the random number generator states so that each thread doesn't create the same random sequence, one to compute the values to be summed (and possibly to compute a subtotal for the values generated), and one summation reduction kernel to sum values from different threads.

Measure the time required to complete the calculation as a function of  $N$  by using the `gettimeofday` function at the beginning and end of your program. Use the plot to estimate the cost to compute a solution with error  $10^{-5}$  with 99.5% confidence and then validate the claim. **Optional:** Compare the pseudorandom number generator and the quasi-random number generator to see whether there is a difference in the accuracy of the results.

### 27.1.2 • Duffing–Van der Pol Oscillator

When solving many stochastic differential equations simultaneously, the various realizations of the solution are all completely independent, so it's simple enough to write a kernel so that each thread solves one SDE from beginning to end and then utilize as many threads as possible in each block. Since the output only involves tallying the hit counts along the way, everything should be done in local memory except for incrementing the hits in global memory so it can be retrieved later.

Program the assignment in Section 34.3.2 using  $N$  samples and using  $M$  time steps, i.e., take  $\Delta t = T/M$  in the Euler–Maruyama method. The values of  $N$ ,  $M$ , and  $\sigma$  should be given on the command line. Use  $\alpha = 1$ , which should be defined in your program. Measure the time required to complete the calculation as a function of  $NM$  by using the `gettimeofday` function at the beginning and end of your program. Plot an estimate for  $p(t)$  for  $0 \leq t \leq T = 10$ .

## 27.2 • Finite Difference Methods

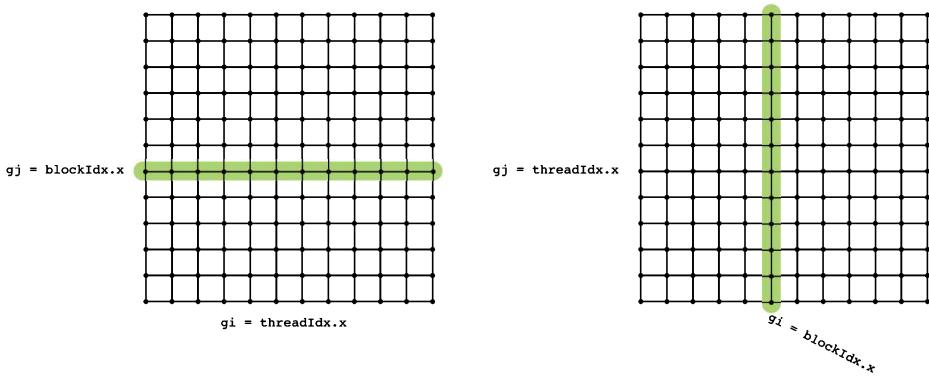
For codes that involve grids, there are many ways in which the blocks and threads can be mapped onto the grid, and the choice depends heavily on the dimensions of the grid being used and the exact calculation being performed. For purposes of this discussion, assume the maximum number of threads per block is  $1024 = 32 \times 32$ , and the size of a warp is 32.

For the ADI method being used for these projects, the explicit update steps are entirely one-dimensional, i.e., there is no interaction between parallel rows or columns of the grid. It makes sense to organize the threads so that each block handles one row and uses the number of available threads. Thus, for an  $N \times N$  grid, where  $N \leq 1024$ , the one-dimensional explicit update kernel could be called with  $N$  blocks each with  $N$  threads and the global indices  $gi$ ,  $gj$  can be computed as

```
int gi = threadIdx.x;
int gj = blockIdx.x;
```

or vice versa depending on the direction of the explicit update. The values of the grid  $u[i*N+j]$  for  $0 \leq i < N$  and for a fixed value for  $j$  would then be loaded into lo-

cal memory, the finite difference operation computed, and then saved back into global memory. This strategy is illustrated in Figure 27.1.



**Figure 27.1.** Illustration of having each block compute an operation on a single row or column. On the left, the block contains a row of the grid for a fixed  $gj$ . On the right, the block contains a column of the grid for a fixed  $gi$ .

If  $N > 1024$ , or greater than the maximum number of threads per block, then the number of blocks in the long dimension would have to be bigger than the current 1. Let  $b = \lceil N/1024 \rceil$ ; then for a fixed  $gj$  and varying  $gi$ , the dimensions of the blocks would be

```
dim3 blocks (b,N);
```

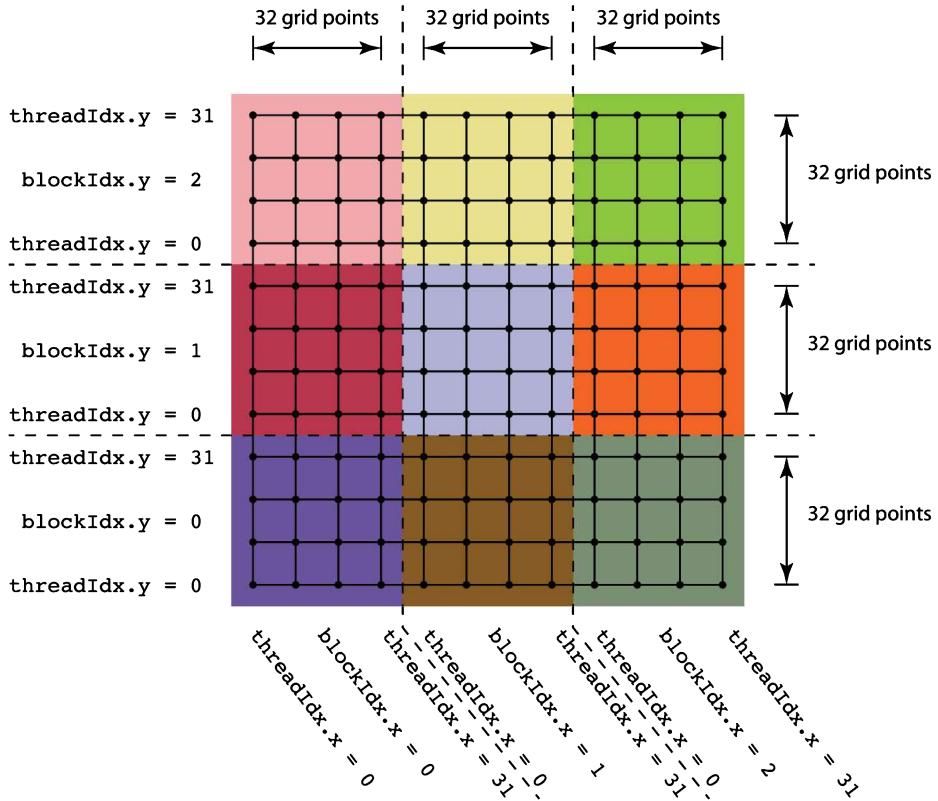
and the global indices  $gi$ ,  $gj$  would be

```
int gi = threadIdx.x + blockDim.x * blockIdx.x;
int gj = blockIdx.y;
```

Another advantage of this one-dimensional strategy is that it allows for the efficient use of either warp shuffles or local memory loaded a warp at a time to do the finite differences.

If the finite difference stencil is two-dimensional as it is for the explicit Runge–Kutta method, then the one-dimensional handling of blocks is not so favorable, and it is better to go with a square arrangement of threads. For the parameters assumed here, that would be a  $32 \times 32$  arrangement of threads. The grid is then tiled in a manner as illustrated in Figure 27.2. For the sake of argument, assume a standard five-point stencil is used for doing a finite difference operation, in other words, to update  $u_{i,j}$ , only the values  $u_{i,j}$ ,  $u_{i\pm 1,j}$ ,  $u_{i,j\pm 1}$  are needed. Within this context, there are two choices for how to manage the edges of the local memory space: (1) use a local memory array that is  $34 \times 34$  and use the threads at the edges to load the extra edge data, or (2) use a local memory array that is  $32 \times 32$  and have only the inner threads perform the finite difference calculation. The two options are illustrated in Figure 27.3.

For the local grid on the left in Figure 27.3, there is an extra cost for having some threads load two (or three points for the case of the corner threads) instead of one. The interior threads must wait until the extra loads into shared memory are completed. By comparison, the local grid on the right in the figure must load only one value into local memory. However, there is a trade-off, because the grid on the left can update  $32^2$  points, while the one on the right can update only  $30^2$  because the edge threads remain idle. There is no perfect solution, but for the case of the explicit finite difference

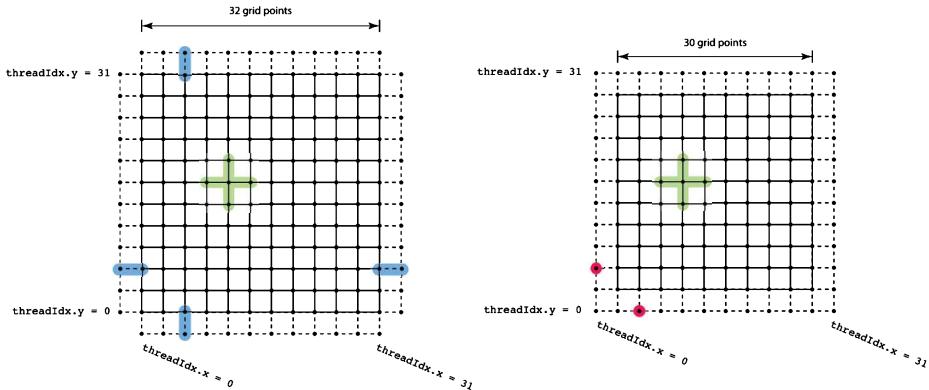


**Figure 27.2.** Illustration of tiling a complete grid with a  $3 \times 3$  array of blocks that are each  $32 \times 32$  grid points.

approximation of the Laplacian, these two are compared for speed, where the left grid in Figure 27.3 is represented as (GPU-shared mem+2) in Figure 25.6, and the right grid is represented as (GPU-shared mem). While it is not a dramatic difference, the left-hand strategy of using a larger local grid is a little faster than letting the edge threads go idle.

Once the numerical method and grid strategy are chosen, multiple kernels are needed to implement the algorithm. For the explicit Runge–Kutta method, just one kernel is needed to handle all stages of the Runge–Kutta method. That kernel should take the stage number or the fraction of the time step that is different for each stage as an argument.

For the ADI method, two kernels for the explicit steps are needed, one for the  $x$ -direction and one for the  $y$ -direction because the indexing in the data is different. Otherwise, these should be fairly simple adaptations of the `diff` kernel in the examples. The implicit steps may use either the tridiagonal solver from the cuSPARSE library as described in Section 26.4.3 or the general matrix solver functions `magma_dgetrf_gpu` and `magma_dgetrs_gpu` from the MAGMA library. Remember that the matrix to be inverted does not change with time, so the `magma_dgetrf_gpu` is called once to factor the matrix, and thereafter use `magma_dgetrs_gpu` to get the solutions, always leaving the data on the device. If the domain is square and the boundary conditions the same for both the  $x$ - and  $y$ -directions, then the factorization is done only once, but if the domain is not square or if the boundary conditions vary between the directions, then the



**Figure 27.3.** Illustration of the two options for using a local square grid for handling a two-dimensional finite difference stencil. The green highlight illustrates the shape of the stencil for a representative grid point. On the left, the local grid is  $34 \times 34$ ; each thread loads one grid value from global into local memory. Threads on the edge, e.g., `threadIdx.x=0` or `threadIdx.y=31`, must load two values as illustrated by the blue highlight. All threads participate in computing the finite difference. On the right, the local grid is  $32 \times 32$ , and each thread loads only one value from global to local memory. The threads on the edge, as illustrated by the red highlight, do not participate in the finite difference calculation.

factorization is done twice, once for each direction. For the solution, it only needs to be called once per iteration because it can be applied to the entire data set, i.e., it can be solved for multiple rows or columns of the array in one call.

No matter which linear solver is used, the right-hand side must still be ordered in the right direction so that the right-hand-side data is contiguous. To facilitate that, a simple kernel for transposing the grid data will be helpful.

### 27.2.1 • Brusselator Reaction

Program the assignment in Section 35.4.1 using an  $N \times N$  grid. In addition to the kernels discussed above, you will also need two more kernels, one to set up a random number generator and another to use the random number generator to initialize the grid with random values. Measure the time required to complete the calculation by using the `gettimeofday` function at the beginning and end of your program and print the elapsed time in seconds.

### 27.2.2 • Linearized Euler Equations

Program the assignment in Section 35.4.2 using an  $N \times N$  grid. In addition to the kernels discussed above, you will also need a kernel to initialize the grid. Measure the time required to complete the calculation by using the `gettimeofday` function at the beginning and end of your program and print the elapsed time in seconds.

## 27.3 • Elliptic Equations and SOR

Since both of the applications in this section involve the five-point diffusion stencil, review the comments concerning organizing the grid from Section 27.2. Use red/black

ordering for the updates. You may write either one kernel that takes the color as an argument or two separate kernels, one for the red update and one for the black. Store the residual in a separate array so that it can be tested for convergence. You will need an additional kernel for a reduction operation for the maximum absolute value in the residual. The reduction operation should use the same scheme as described in Section 27.1.1.

### 27.3.1 • Diffusion with Sinks and Sources

Program the assignment in Section 36.1.1 using a  $(2N - 1) \times N$  grid. Experiment to find the optimal  $\omega$  that requires the fewest iterations to reach a tolerance of  $10^{-9}$ . Compute the time required to complete the calculation using the `gettimeofday` function, then divide by the number of iterations to give the average time used for a single pass through the grid. Repeat these steps for a grid of dimensions  $(4N - 1) \times 2N$ . Does the optimal value of  $\omega$  remain the same? Verify that the time cost for a single pass through the grid is proportional to the number of grid points.

### 27.3.2 • Stokes Flow 2D

Considering that the boundary conditions are slightly different for the three different variables and that the grid dimensions are different, you will want to use specialized kernels for each.

Program the two-dimensional assignment in Section 36.1.2 using a MAC grid that is approximately  $N \times N$ , i.e., the grid for  $u$  should be  $N \times N - 1$ , the grid for  $v$  should be  $N - 1 \times N$ , and the grid for  $p$  should be  $N - 1 \times N - 1$ . Experiment to find the optimal  $\omega$  that requires the fewest iterations to reach a tolerance of  $10^{-9}$ . Verify that you get a parabolic profile for the velocity field in the  $x$  direction. Check that the time cost for a single pass through the grid is proportional to the number of grid points.

### 27.3.3 • Stokes Flow 3D

Program the three-dimensional assignment in Section 36.1.2 using a MAC grid that is approximately  $N \times N \times N$ , i.e., the grid for  $u$  should be  $N \times N - 1 \times N - 1$ , the grid for  $v$  should be  $N - 1 \times N \times N - 1$ , the grid for  $w$  should be  $N - 1 \times N - 1 \times N$ , and the grid for  $p$  should be  $N - 1 \times N - 1 \times N - 1$ . Note that the grid arrangements described in Section 27.2 would need to be modified for three dimensions. In this case, you would end up with a  $16 \times 8 \times 8$  arrangement of threads in each block. Experiment to find the optimal  $\omega$  that requires the fewest iterations to reach a tolerance of  $10^{-9}$ . Verify that you get a roughly parabolic profile for the velocity field in the  $x$ -direction. Check that the time cost for a single pass through the grid is proportional to the number of grid points.

## 27.4 • Pseudospectral Methods

The Fourier transforms will be handled using the cuFFT library keeping data on the device. Additionally, one or more separate kernels will be needed to perform the pseudospectral derivative operations and a kernel to implement the Runge–Kutta method update. Kernels will also be needed to set up for random number generation on the device and to initialize the random data on the grid.

### 27.4.1 • Complex Ginsburg–Landau Equation

Program the assignment in Section 37.5.1 using an  $N \times N$  grid. Because this is a complex-valued equation, use the `cufftExecZ2Z` plan execution. Compute the time required to execute the solution to the indicated terminal time. Plot the results for the phase and identify where the spiral waves have emerged through pattern formation. Measure the time required to complete the calculation by using the `gettimeofday` function at the beginning and end of the program and print the elapsed time in seconds.

### 27.4.2 • Allen–Cahn Equation

Program the assignment in Section 37.5.2 in two dimensions using an  $N \times N$  grid. Because this is a real-valued equation, some gain in efficiency can be made by using the pair of plan executions `cufftExecD2Z` and `cufftExecZ2D`. The former is used for the forward transform and the latter for the inverse transform. Compute the time required to execute the solution to the indicated terminal time. Plot the results and verify that phase separation is occurring. Measure the time required to complete the calculation by using the `gettimeofday` function at the beginning and end of your program and print the elapsed time in seconds.



## **Part V**

# **GPU Programming and OpenCL**



The CUDA language covered in Part IV is a proprietary language designed exclusively for NVIDIA graphics cards. Not all computers contain NVIDIA graphics cards, so an alternative is needed. An open source cross-platform alternative to CUDA is called OpenCL [22]. Most of the high-end graphics card makers, including NVIDIA, have an OpenCL backend so that a generic OpenCL interface can be used to access lots of subsystems on a computer including not just graphics cards, but also CPUs, digital signal processors (DSPs), and so on. Each subsystem has a unique set of characteristics and capabilities and can be run in parallel. OpenCL makes all of these devices available to the programmer by letting the code dictate the parameters for optimal operation. In this text, the focus will be just on using the GPUs since that is the most useful for the type of programs required for scientific computing. For a reference to the OpenCL functions and more information, check out the website

<https://www.khronos.org>

There are also helpful reference books [23].

In Chapter 28, the basic steps to create an OpenCL application are discussed, including how to initiate the connection with the GPU devices, how to retrieve key functionality of the device, and how to select an individual device for computation.

OpenCL kernel functions are introduced in Chapter 29 along with a discussion about how kernels are stored, whether as strings in a “.c” file or as a separate file that is read through a supporting function. The organization of the computational units into work-groups and work-items is also presented.

For OpenCL, the memory space on the device is broken down into global, local, and constant. Chapter 30 describes how to access and utilize the different types of memory within the OpenCL framework.

Kernel functions are executed by submitting them to command queues. Chapter 31 explores the role of command queues and how to insert events into command queues to control the order in which kernels are executed and to measure the cost of computing a kernel function. Using multiple command queues makes it possible for concurrent kernel execution within a single GPU and also makes it easy to utilize multiple devices, including the CPU itself, as part of a computation.



# Chapter 28

# Intro to OpenCL

When comparing OpenCL with CUDA, there is one noticeable difference, which is that CUDA is an extension of the C language that incorporates a special compiler, nvcc, while OpenCL uses plain C and access to the GPU, including executing kernel functions, is handled by an external OpenCL library that is linked in. Since both CUDA and OpenCL are utilizing the same hardware, there are some obvious parallels in terms of language. Both refer to the programs written for the GPU as kernel functions. For CUDA the terms blocks, threads, and streams correspond to the OpenCL terms of work-groups, work-items, and command queues, respectively.

Undoubtedly, the biggest difference between CUDA and OpenCL is how and when the kernel functions are compiled and invoked. Where CUDA has a streamlined specialized notation that hides the details from the user, OpenCL has it all on display. This chapter begins with the equivalent of “Hello World!” by using the GPU to add two numbers together. The basic steps of using OpenCL including choosing a device, setting up the environment, and introducing the compiling process are also covered.

## 28.1 • First OpenCL Program

The basic structure of an OpenCL program includes a `main` program that runs on the host and kernel functions that run on the GPU and other devices. Kernel functions can be written in separate files using the suffix “.cl” or by explicitly writing them as text strings depending on the implementation of OpenCL. This means that the code for the kernel functions is *compiled at runtime*. This can be expensive if the same kernel function is compiled repeatedly, so kernels should be compiled once and then reused as needed.

### Example 28.1.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <CL/opencl.h>
4
5 char addkernel[256] =
6 "kernel void add(int a, int b, global int* c)      \
7 {          *c = a + b;                                \
8 }";
```

```
10  /*
11   * int main(int argc, const char * argv[])
12   *
13   * Example generating random numbers
14   *
15   * Inputs: argc should be 3
16   * argv[1]: first addend
17   * argv[2]: second addend
18   *
19   * Outputs: the sum
20   */
21
22
23 int main(int argc, const char * argv[]) {
24
25     // Read the input values
26     int a = atoi(argv[1]);
27     int b = atoi(argv[2]);
28
29     // c is the place for the result
30     int c;
31
32     // err will store error info for OpenCL functions
33     int err;
34
35     // Request a GPU device
36     cl_platform_id platform;
37     err = clGetPlatformIDs(1, &platform, NULL);
38     cl_device_id device;
39     err = clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device, NULL);
40
41     // Setup the context for the GPU
42     cl_context context = clCreateContext(0, 1, &device, NULL, NULL, &err);
43
44     // Before accessing the GPU, we must create a dispatch queue,
45     // which is the sequence of commands that will be sent to the GPU.
46     cl_command_queue queue = clCreateCommandQueueWithProperties(context,
47                     device, NULL, &err);
48
49     // In OpenCL, the kernel is compiled at runtime
50     const char* srccode = addkernel;
51     cl_program program = clCreateProgramWithSource(context, 1, &srccode,
52                                         NULL, &err);
53
54     // Compile the kernel code
55     err = clBuildProgram(program, 0, NULL, NULL, NULL);
56
57     // Create the kernel function from the compiled OpenCL code
58     cl_kernel kernel = clCreateKernel(program, "add", &err);
59
60     // Set up the memory on the GPU where the answer will be stored
61     cl_mem dev_c = clCreateBuffer(context, CL_MEM_WRITE_ONLY, sizeof(int),
62                               NULL, NULL);
63
64     // Define all the arguments for the kernel function
65     err = clSetKernelArg(kernel, 0, sizeof(int), &a);
66     err = clSetKernelArg(kernel, 1, sizeof(int), &b);
67     err = clSetKernelArg(kernel, 2, sizeof(cl_mem), &dev_c);
68
69     const unsigned long one = 1;
70
71     // Execute the kernel
72     err = clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &one, &one, 0,
```

```
73     clFinish(queue);                                     NULL, NULL);  
74  
75     // Retrieve the answer from the GPU memory  
76     err = clEnqueueReadBuffer(queue, dev_c, 1, 0, sizeof(int), &c, 0,  
77                               NULL, NULL);  
78  
79     printf("%d + %d = %d\n", a, b, c);  
80  
81     // Free the memory allocated on the GPU  
82     clReleaseMemObject(dev_c);  
83     clReleaseKernel(kernel);  
84     clReleaseProgram(program);  
85     clReleaseCommandQueue(queue);  
86     clReleaseContext(context);  
87  
88     return 0;  
89 }  
90 }
```

Example 28.1 demonstrates the basic steps for creating and running a kernel function to add two integers. The basic pattern is that memory is allocated both on the host and the GPU, data is transported to the GPU, a kernel is executed on that data, and the results are transported back to the host. And of course there is clean-up at the end. However, before a kernel can be run, the device must be prepared, which entails the sequence of several steps: (1) getting the platform ID, (2) requesting an available device, (3) setting up the device context, (4) setting up a command queue, (5) compiling the kernel, (6) allocating device memory, and (7) defining the arguments for the kernel. This is all done on Lines 37–67.

### 28.1.1 • Preparing the Device

The first step is to get a platform ID. A platform in OpenCL is information on what devices are available for the programmer to use. This includes any GPU devices and may also include the CPU, digital signal processors (DSPs), or other devices. When OpenCL is installed, one or more platforms are created that contain the hardware specific information available to the programmer. On Line 37 of Example 28.1, the default platform is requested. Section 28.2 will discuss the function `clGetPlatformIDs` in greater detail. This function, like most OpenCL functions, returns an error status value either as a return value or through the last argument in the argument list. Error handling will be discussed in Section 29.3; for now it is ignored.

Next, a GPU device is requested from the platform using `clGetDeviceIDs` on Line 39. A CPU device can be requested using the argument `CL_DEVICE_TYPE_CPU` instead. Here, only one GPU device is requested. Choosing a particular GPU device will be covered in Section 28.2, while using multiple GPU devices will be discussed later in Chapter 31.

Once the device is chosen, a context for the device must be set up. This includes properties such as available memory, command queues, and other things for the chosen device. The context is set up on Line 42 using the function `clCreateContext`. The only arguments of relevance in this text are the second, third, and last arguments. The second and third arguments are the number of devices to be used and the device ID obtained from `clGetDeviceIDs`, and the last argument is where any error codes are stored.

Kernels are entered into command queues to do their calculations, so the next step is to set up a command queue for the device. The command queue is created with the function `clCreateCommandQueueWithProperties` for the selected context and device on Line 46. The default behavior is for the third argument to be `NULL`, but this is where a list of queue properties can be inserted. One property that will be encountered in Section 31.2 is `CL_QUEUE_PROFILING_ENABLE` for measuring the time required for kernels in the queue, but for now the default command queue is sufficient. Command queues in general will be discussed in greater detail in Chapter 31. The device is now ready to begin accepting kernel functions for computing tasks.

### 28.1.2 • Compiling a Kernel Function

Before a kernel can be submitted to a command queue, the kernel code must be compiled. For OpenCL, compiling the kernel occurs at run-time. Like any other program that gets compiled, a kernel program starts as a string of text that gets fed to a compiler. In this case, the text of the kernel program is stored in a string variable on Line 5. String constants cannot be broken across lines of code without special treatment. Here, the “\” character is a continuation character to indicate that the string continues on the next line. Thus, the entire kernel function is stored in a single string variable. The source code is loaded into memory to be compiled on Line 51 using `clCreateProgramWithSource`.

Once the code for the kernel is loaded, it is compiled using `clBuildProgram` on Line 55. A more detailed discussion about the compilation process will be covered in Section 29.2 including alternative ways to store the kernel code and how to retrieve any error messages that arise as part of the compilation. The compiled code is then turned into an executable kernel ready for a command queue using `clCreateKernel` on Line 58.

### 28.1.3 • Running a Kernel

Before running a kernel, the memory on the device must be prepared. In this case, the only device memory needed is a single integer, `dev_c`, which is allocated using `clCreateBuffer` on Line 61. Here, the label “`dev_`” is not required but is a reminder that it is a memory buffer set up on the device, and not memory on the host. In this example, `dev_c` is the memory where the kernel will store the result of the sum.

In addition to the output variable `dev_c`, the kernel also has two other input arguments, integers `a` and `b`, on Line 6. All the arguments are set up using the function `clSetKernelArg` on Lines 65–67. The second argument gives the position in the argument list, the third is the size in bytes of the argument, and the last is a pointer to the data to be used.

The kernel is actually run by submitting the kernel to the device’s command queue with the function `clEnqueueNDRangeKernel` on Line 72. The main program will continue immediately after queuing the kernel, so the kernel may not be done before the host proceeds to attempt to read the output on Line 77. Thus, the `clFinish` function on Line 74 blocks execution until the command queue has finished performing all the tasks in its queue. More refined control of the command queue will be covered in Chapter 31.

After the kernel is executed, the output data is mapped back to the host using `clEnqueueReadBuffer` on Line 77. Reading a buffer back into the host is also a task that is put in the command queue. Here, the device memory, `dev_c`, will be copied into the host variable, `c`, and the amount of memory to be read will be the size of one `int`. Some of the other options for this command will be discussed later.

### 28.1.4 • Clean-Up

At the end, there is significant clean-up required. The items created in this example must all be released as shown on Lines 83–87. This includes `clReleaseMemObject` to release the device memory object, `dev_c`, `clReleaseKernel` to release the kernel used for execution, `clReleaseProgram` to release the program used to build the kernel, `clReleaseCommandQueue` to release the command queue set up for the device, and `clReleaseContext` to release the context.

### 28.1.5 • The Kernel Program

The kernel on Lines 5–9 is very simple. Line 6 contains a new keyword, `kernel`, which tells the compiler that this is meant to be a kernel function. It computes the sum of the first two arguments and stores the result in the third. The first two arguments are values that are passed from the host to the kernel through the `clSetKernelArg` function on Lines 65, 66. The third argument is based on device memory and the keyword `global` indicates it will be stored in global memory space. On the host side, the device memory is allocated on Line 61 and has type `cl_mem` and is not specified as having any particular data type, only the amount of memory allocated is indicated. In the kernel it is referenced as a pointer to an array, however, and can be used for any data type so long as it fits inside the amount of memory allocated. Thus, storing the sum in `*c` in the kernel places it in the buffer so that it can be read back later into the host on Line 77.

## 28.2 • Setting Up the Context

For the most part, setting up the GPU context will be the same each time, although hardware changes and portability may require some adaptation. This section covers setting up the context more carefully to make it more general. Example 28.2 shows a more detailed construction of the context after showing how to access information about the platforms and devices available.

#### Example 28.2.

```
1 #include <stdio.h>
2
3 // Header file for OpenCL functions
4 #include <CL/opencl.h>
5
6 // Function to make sure the text buffer is long enough
7 void CheckBuffer(char** buffer, size_t* bufferlen, size_t newbufferlen)
8 {
9     if (newbufferlen > *bufferlen) {
10         if (*buffer == NULL)
11             *buffer = (char*)malloc(newbufferlen*sizeof(char));
12         else
13             *buffer = (char*)realloc(*buffer, newbufferlen*sizeof(char));
14         *bufferlen = newbufferlen;
15     }
16 }
17
18 /*
19  int main(int argc, const char * argv[])
20
21 Set up the OpenCL environment and show available options
22
23 Inputs: none
```

```

24
25   Outputs: useful info about each device
26 */
27
28 int main(int argc, const char *argv[])
29 {
30   int i;
31   cl_uint num_platforms;
32   size_t size;
33   char* buffer = NULL;
34   size_t bufferlen = 0;
35
36   // Get the number of available platforms
37   cl_int err = clGetPlatformIDs(0, NULL, &num_platforms);
38
39   // Retrieve all the available platforms
40   cl_platform_id platform[num_platforms];
41   err = clGetPlatformIDs(num_platforms, platform, NULL);
42
43   // Retrieve information about each platform
44   for (i=0; i<num_platforms; ++i) {
45     printf("Platform %d info :\n", i);
46
47     // Get the OpenCL version
48     err = clGetPlatformInfo(platform[i], CL_PLATFORM_VERSION, 0, NULL,
49                           &size);
50     buffer = CheckBuffer(&buffer, &bufferlen, size);
51     err = clGetPlatformInfo(platform[i], CL_PLATFORM_VERSION, size,
52                           buffer, NULL);
53     printf("\tVersion: %s\n", buffer);
54
55     // Get the platform name
56     err = clGetPlatformInfo(platform[i], CL_PLATFORM_NAME, 0, NULL,
57                           &size);
58     buffer = CheckBuffer(&buffer, &bufferlen, size);
59     err = clGetPlatformInfo(platform[i], CL_PLATFORM_NAME, size, buffer,
60                           NULL);
61     printf("\tName: %s\n", buffer);
62
63     // Get the platform vendor
64     err = clGetPlatformInfo(platform[i], CL_PLATFORM_VENDOR, 0, NULL,
65                           &size);
66     buffer = CheckBuffer(&buffer, &bufferlen, size);
67     err = clGetPlatformInfo(platform[i], CL_PLATFORM_VENDOR, size,
68                           buffer, NULL);
69     printf("\tVendor: %s\n", buffer);
70
71     // Get the list of devices for this platform
72     cl_uint num_devices;
73     clGetDeviceIDs(platform[i], CL_DEVICE_TYPE_ALL, 0, NULL,
74                           &num_devices);
75     cl_device_id device[num_devices];
76     clGetDeviceIDs(platform[i], CL_DEVICE_TYPE_ALL, num_devices, device,
77                           NULL);
78
79     int j;
80     for (j=0; j<num_devices; ++j) {
81
82       // Get the device make and model
83       printf("\tDevice %d: ", j);
84       err = clGetDeviceInfo(device[j], CL_DEVICE_VENDOR, 0, NULL,
85                           &size);
86       buffer = CheckBuffer(&buffer, &bufferlen, size);

```

```
87     err = clGetDeviceInfo(device[j], CL_DEVICE_VENDOR, size, buffer,
88                           NULL);
89     printf("%s ", buffer);
90     err = clGetDeviceInfo(device[j], CL_DEVICE_NAME, 0, NULL, &size);
91     buffer = CheckBuffer(&buffer, &bufferlen, size);
92     err = clGetDeviceInfo(device[j], CL_DEVICE_NAME, size, buffer,
93                           NULL);
94     printf("%s\n", buffer);
95
96     // Get the device type
97     printf("\t\tType:");
98     cl_device_type dtype;
99     err = clGetDeviceInfo(device[j], CL_DEVICE_TYPE,
100                          sizeof(cl_device_type), &dtype, NULL);
101    if (dtype & CL_DEVICE_TYPE_CPU)
102        printf(" CPU");
103    if (dtype & CL_DEVICE_TYPE_GPU)
104        printf(" GPU");
105    if (dtype & CL_DEVICE_TYPE_ACCELERATOR)
106        printf(" ACCELERATOR");
107    if (dtype & CL_DEVICE_TYPE_DEFAULT)
108        printf(" DEFAULT");
109    if (dtype & CL_DEVICE_TYPE_CUSTOM)
110        printf(" CUSTOM");
111    printf("\n");
112
113    // Get device memory information
114    cl_ulong memsize;
115    err = clGetDeviceInfo(device[j], CL_DEVICE_GLOBAL_MEM_SIZE,
116                          sizeof(cl_long), &memsize, NULL);
117    printf("\t\tGlobal Memory Size: %ld\n", memsize);
118
119    err = clGetDeviceInfo(device[j], CL_DEVICE_LOCAL_MEM_SIZE,
120                          sizeof(cl_long), &memsize, NULL);
121    printf("\t\tLocal Memory Size: %ld\n", memsize);
122
123    err = clGetDeviceInfo(device[j],
124                          CL_DEVICE_MAX_CONSTANT_BUFFER_SIZE,
125                          sizeof(cl_long), &memsize, NULL);
126    printf("\t\tConstant Memory Size: %ld\n", memsize);
127
128    // Get the number of multiprocessors
129    cl_uint units;
130    err = clGetDeviceInfo(device[j], CL_DEVICE_MAX_COMPUTE_UNITS,
131                          sizeof(cl_uint), &units, NULL);
132    printf("\t\tMax Compute Units: %d\n", units);
133
134    // Get the maximum work-group size
135    size_t size;
136    err = clGetDeviceInfo(device[j], CL_DEVICE_MAX_WORK_GROUP_SIZE,
137                          sizeof(size_t), &size, NULL);
138    printf("\t\tMax Work Group Size: %ld\n", size);
139
140    // Get the maximum work-items per work-group
141    err = clGetDeviceInfo(device[j],
142                          CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS,
143                          sizeof(cl_uint), &units, NULL);
144    size_t widims[units];
145    err = clGetDeviceInfo(device[j], CL_DEVICE_MAX_WORK_ITEM_SIZES,
146                          sizeof(widims), widims, NULL);
147    printf("\t\tMax Work-item dimensions: (");
148    int k;
149    for (k=0; k<units; ++k) {
```

```

150     printf("%ld ", widims[k]);
151     if (k < units -1)
152         printf(", ");
153 }
154 printf("\n\n");
155
156 // Get the list of platform extensions
157 err = clGetDeviceInfo(device[j], CL_DEVICE_EXTENSIONS, 0, NULL,
158                         &size);
159 CheckBuffer(&buffer, &bufferlen, size);
160 err = clGetDeviceInfo(device[j], CL_DEVICE_EXTENSIONS, size,
161                         buffer, NULL);
162 printf("\tExtensions: %s\n\n", buffer);
163 }
164
165 // release the buffer
166 if (buffer != NULL)
167     free(buffer);
168
169 // Use GPU device[0] from platform[0]
170 cl_platform_id theplatform;
171 err = clGetPlatformIDs(1, &theprofile, NULL);
172 cl_uint numdev;
173 clGetDeviceIDs(theprofile, CL_DEVICE_TYPE_GPU, 0, NULL, &numdev);
174 cl_device_id devices[numdev];
175 clGetDeviceIDs(profile[0], CL_DEVICE_TYPE_GPU, numdev, devices,
176                         NULL);
177
178 // Now set up the context
179 cl_context context = clCreateContext(0, 1, &devices[0], NULL, NULL,
180                                         &err);
181
182
183 return 0;
184 }
```

The header file `CL/opencl.h` shown on Line 4 is where all the OpenCL functions are declared.<sup>10</sup> Lines 37–168 show how to retrieve information about the platforms and devices available. The functions `clGetPlatformInfo` and `clGetDeviceInfo` take many more options than are shown in this example, but the key ones relevant to beginning OpenCL programming are shown and the results for a few machines as illustrations appear in Tables 28.1–28.3. Check the OpenCL documentation for a complete list of options. Lines 172–180 are the lines that will appear at the beginning of every OpenCL code up to minor modifications. If multiple platforms or devices are available, then these functions would be used to ensure that the correct combination is chosen for the given application.

The first step in setting up the GPU context is to choose a platform, which is a collection of devices available. For most systems, there is only one. To get the number of available platforms, and to choose the platform, use the function

```

cl_int clGetPlatformIDs( cl_uint num_requested,
                        cl_platform_id* platforms,
                        cl_uint* num_platforms );
```

---

<sup>10</sup>For installations on OS X systems, the header is in a different directory so use `OpenCL/opencl.h` instead.

To explore all the available platforms, the function is called twice: once to get the number of available platforms, and a second time to get the actual list as on Lines 37 and 41. By setting the `num_requested` to zero and `platforms` to `NULL`, then only the number of available platforms is returned through the argument `num_platforms`. Once the number of platforms is known, then an array of platforms can be created so that they can all be loaded. This time `num_requested` uses the `num_platforms` from the first call, and `platforms` is an array of type `cl_platform_id`. If using the default platform, then it is sufficient to call `clGetPlatformIDs` once as on Line 172.

Once the platform is chosen, the next step is to choose the device to be used. The function

```
cl_int clGetDeviceIDs( cl_platform_id platform,
                      cl_device_type device_type,
                      cl_uint num_requested,
                      cl_device_id* devices,
                      cl_uint* num_devices );
```

is used in a very similar manner to `clGetPlatformIDs`. The chosen platform is used for the first argument. The `device_type` argument makes it possible to specify the type of devices desired. To include all devices, use device type `CL_DEVICE_TYPE_ALL`, but to get a specific type, any of the types listed in Lines 101–109 can be used individually or as a combination. For example, to get the number of devices that are either a CPU or a GPU, use

```
clGetDeviceInfo(platform, CL_DEVICE_TYPE_CPU | CL_DEVICE_TYPE_GPU,
               0, NULL, &num_devices);
```

To get the number of available devices, set `num_requested` to zero and `devices` to `NULL` and the number of devices will be put in the variable `num_devices`. To get a list of devices, use `num_devices` for the `num_requested` argument and an array large enough to store the list for `devices`. On Lines 174 and 176, the list of all available GPU devices is obtained.

On Line 158, a list of possible extensions is enumerated and printed as a text string. This is a string worth checking on your system to ensure that double precision calculations are possible. By default, OpenCL works with single precision, but double precision can be turned on for a kernel if the extension `cl_khr_fp64` is available. Check the list of extensions for this extension to ensure your device is capable of double precision calculations. This extension will be enabled in Example 29.1.

Once the device to be used for a calculation is chosen, the OpenCL context can be created with the function

```
cl_int clCreateContext( const cl_context_properties* props,
                      cl_uint num_devices,
                      cl_device_id* devices,
                      void (CL_CALLBACK* fcn)(...),
                      void* data,
                      cl_int* err );
```

The `props` argument is a zero-terminated list of properties which are beyond the scope of this text. For the purposes of this text, just use zero for `props`. Likewise, the `fcn` and `data` arguments are for advanced programmers; simply use the constant `NULL` for each of these. To set up the context to use one GPU device, set `num_devices` to one, and

put a pointer to the chosen device ID obtained from `clGetDeviceIDs` for the `devices` pointer. The `err` value can be set to `NULL` if no error checking is done; otherwise the `error` value can be obtained. Error checking will be covered in more detail in Section 29.3. Once the context is set up, it is used throughout the rest of the program.

The output of Example 28.2 was obtained on a few sample systems and the results are shown in Tables 28.1–28.3. Not all available OpenCL-compatible devices are the same, and for some of them the difference can be quite important. For example, not all devices are GPUs. Choosing the device poorly can cause trouble or make the code run slower. When using multiple devices, choosing two devices that are different will require effort to ensure that the computational load is balanced properly so that each finishes as close as possible to the same time even though they may have different speeds and capabilities.

**Table 28.1.** Info for the three available devices on a MacBook Pro Retina Display.

Device Name	Intel i7	Intel HD Graphics 4000	NVIDIA GeForce GT 650M
Device Type	CPU	GPU	GPU
Multiprocessors	8	16	2
Max. work-group size	1024	512	1024
Available global mem.	8.5Gb	1Gb	1Gb
Available local mem.	32,768 bytes	65,536 bytes	49,152 bytes
Available constant mem.	65,536 bytes	65,536 bytes	65,536 bytes

**Table 28.2.** Info for the three available devices on a Mac Pro.

Device Name	Intel Xeon	AMD Radeon HD D300	AMD Radeon HD D300
Device Type	CPU	GPU	GPU
Multiprocessors	8	20	20
Max. work-group size	1024	256	256
Available global mem.	12.8Gb	2.1Gb	2.1Gb
Available local mem.	32,768 bytes	32,768 bytes	32,768 bytes
Available constant mem.	65,536 bytes	65,536 bytes	65,536 bytes

**Table 28.3.** Info for the three available devices on a Dell workstation with dual Tesla K20c cards.

Device Name	NVIDIA Tesla K20c	NVIDIA Quadro K620	NVIDIA Tesla K20c
Device Type	GPU	GPU	GPU
Multiprocessors	13	3	13
Max. work-group size	1024	1024	1024
Available global memory	5.0Gb	2.1Gb	5.0Gb
Available local memory	49,152 bytes	49,152 bytes	49,152 bytes
Available constant memory	65,536 bytes	65,536 bytes	65,536 bytes

Once a device is selected, use the `clGetDeviceInfo` function as illustrated in Example 28.2 to get the key characteristics such as the shared memory size, or the maximum number of work-items per work-group to know how to optimize the code.

---

## Exercise

- 28.1. Compile and run Example 28.2 and assemble a table of data for your system. Make a note of which platform and device(s) are available and what their platform and device indices are.



## Chapter 29

# Parallel OpenCL Using Work-Groups

The first parallel OpenCL example program is presented in this chapter. As outlined in Section 28.1, there are a number of steps required to invoke a kernel from loading the source code to compiling and linking during runtime. One of the steps that will get extra scrutiny is how kernel functions are compiled. There are facilities for offline compiling, where the kernel is compiled using a shell program and saved to a file for later use, but that is a bit more subtle and beyond the scope of this text. Instead, steps for loading a kernel function from a file and then compiling it will be covered.

After the careful treatment of compiling a kernel, a finite difference kernel will be developed as a test case for doing parallel computations. The basic building blocks of parallel code using OpenCL are work-groups and work-items, which are functionally equivalent to blocks and threads in CUDA. The number of work-groups and the number of work-items per work-group are determined when the kernel is inserted into the command queue. The organization of work-groups and work-items will also be covered in this chapter.

### 29.1 • Parallel OpenCL Example

Example 29.1 is a first example of a parallel code using OpenCL, where a simple finite difference approximation is implemented for a one-dimensional data set.

#### Example 29.1.

```
1 char kernel[1024] =  
2 "#pragma OPENCL EXTENSION cl_khr_fp64 : enable \n \  
3  
4 kernel void diff(global double* u,           \  
5                 int N,                   \  
6                 double dx,               \  
7                 global double* du)           \  
8 {  
9     size_t i = get_global_id(0);           \  
10    int ip = (i+1)%N;                   \  
11    int im = (i+N-1)%N;                 \  
12    du[i] = (u[ip] - u[im])/dx / .;  
13 }";  
14  
15 #include <stdio.h>
```

```

16 #include <stdlib.h>
17 #include <math.h>
18 #include <CL/opencl.h>
19
20 #ifndef M_PI
21 #define M_PI 3.1415926535897932384626433832795
22 #endif
23
24 /*
25  int main(int argc, const char * argv[])
26
27 Demonstrate a simple example for implementing a
28 parallel finite difference operator
29
30 Inputs: argc should be 2
31 argv[1]: Length of the vector of data
32
33 Outputs: the initial data and its derivative.
34 */
35
36 int main(int argc, const char * argv[]) {
37
38 // Read the input values
39 int N = atoi(argv[1]); // Get the length of the vector from input
40
41 // Get platform info
42 cl_platform_id platform;
43 int err = clGetPlatformIDs(1, &platform, NULL);
44
45 // Request a GPU device
46 cl_device_id device_id;
47 err = clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device_id,
48 NULL);
49
50 // Setup the context for the GPU
51 cl_context context = clCreateContext(0, 1, &device_id, NULL, NULL,
52 &err);
53
54 // Before accessing the GPU, we must create a dispatch queue, which is
55 // the sequence of commands that will be sent to the GPU.
56 cl_command_queue queue = clCreateCommandQueueWithProperties(context,
57 device_id, NULL, &err);
58
59 // In OpenCL, the kernel is compiled at runtime.
60 const char* srccode = kernel;
61 cl_program program = clCreateProgramWithSource(context, 1, &srccode,
62 NULL, &err);
63
64 // Compile the kernel code
65 err = clBuildProgram(program, 0, NULL, NULL, NULL);
66
67 // Create the kernel function from the compiled OpenCL code
68 cl_kernel kernel = clCreateKernel(program, "diff", &err);
69
70 // Allocate memory for the input arguments, data will be copied from
71 // the inputs to the GPU at the same time.
72 double dx = 2*M_PI/N;
73 double* u;
74 u = (double*)malloc(N*sizeof(double));
75 int i;
76 for (i=0; i<N; ++i)
77     u[i] = sin(i*dx);
78

```

```

79  double* du = (double*)malloc(N*sizeof(double));
80
81  cl_mem dev_u = clCreateBuffer(context, CL_MEM_READ_ONLY,
82                                N*sizeof(double), NULL, &err);
83  err = clEnqueueWriteBuffer(queue, dev_u, 0, 0, N*sizeof(double), u, 0,
84                            NULL, NULL);
85
86  // Set up the memory on the GPU where the answer will be stored
87  cl_mem dev_du = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
88                                 N*sizeof(double), NULL, NULL);
89
90  // Define all the arguments for the kernel function
91  err = clSetKernelArg(kernel, 0, sizeof(cl_mem), &dev_u);
92  err = clSetKernelArg(kernel, 1, sizeof(int), &N);
93  err = clSetKernelArg(kernel, 2, sizeof(double), &dx);
94  err = clSetKernelArg(kernel, 3, sizeof(cl_mem), &dev_du);
95
96  size_t maxwgs = 1;
97  size_t N1 = N;
98  err = clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &N1, &maxwgs, 0,
99                               NULL, NULL);
100 clFinish(queue);
101
102 err = clEnqueueReadBuffer(queue, dev_du, 1, 0, N*sizeof(double), du,
103                           0, NULL, NULL);
104
105 // Free the memory allocated on the GPU
106 clReleaseMemObject(dev_du);
107 clReleaseMemObject(dev_u);
108 clReleaseKernel(kernel);
109 clReleaseProgram(program);
110 clReleaseCommandQueue(queue);
111 clReleaseContext(context);
112
113 // Free the memory on the CPU
114 free(u);
115 free(du);
116
117 return 0;
118 }

```

The program takes as input the length of the initial data vector, then computes the central difference operator on the data, which is assumed periodic, on the GPU. The basic outline of the algorithm is as follows: (1) the data is initialized in host memory, (2) it is copied to the device memory where (3) the kernel computes the finite difference, and (4) the result copied back to host memory to be written to the terminal.

Before the algorithm can be executed, the hardware context must be created. The context with a single GPU is set up on Lines 43–51 as was discussed in the previous chapter. Once the context is created, then a command queue is created using the function below so that kernels can be submitted to the GPU:

```

cl_command_queue clCreateCommandQueueWithProperties(
    cl_context context,
    cl_device_id device,
    cl_queue_properties* props,
    cl_int* err
);

```

The first two arguments are the `context` created and the specific `device` to be associ-

ated with this command queue. For now, the array `props` will be given the value `NULL`, and like some other functions, the value `err` is assigned an error value unless `NULL` is used for the last argument. The function returns the created command queue as shown on Line 56.

The next step is to actually create the binary code that the device will run, which requires a runtime compilation. Section 29.2 will cover this step in significant detail, so for now understand that the kernel is loaded and compiled in a similar manner as Example 28.1. In the present example, the compilation is done on Lines 59–68.

Returning to the algorithm, after the data on the host is initialized, the next step is to set up the memory allocation on the device using the function

```
cl_mem clCreateBuffer( cl_context context,
                      cl_mem_flags options,
                      size_t size,
                      void* host_ptr,
                      cl_int* err );
```

**Table 29.1.** List of available options for the `clCreateBuffer` function. The last option can be combined with others using the bitwise “or” operator “|”.

Option	Meaning
<code>CL_MEM_READ_ONLY</code>	The GPU will only read from this memory, not write to it.
<code>CL_MEM_WRITE_ONLY</code>	The GPU will only write to this memory, not read from it.
<code>CL_MEM_READ_WRITE</code>	The GPU will read and/or write to this memory.
<code>CL_MEM_COPY_HOST_PTR</code>	After allocating the memory, the data contained in the provided host pointer will be copied into the allocated space.

There are several choices for `options`; the ones used in this text are in Table 29.1. The `host_ptr` argument is assigned `NULL` unless the `CL_MEM_COPY_HOST_PTR` option is used, in which case `host_ptr` must point to memory on the host at least as large as the memory being allocated on the device. All memory stored on the device is arranged into objects of type `cl_mem`, which is returned from `clCreateBuffer`. The input buffer is called `dev_u`, and is marked as *read only* because the GPU will not write to that buffer, only read data from it.

Before the GPU can read the buffer, though, it must be populated with the data from the host. To do this, a command to write to a buffer is put in the command queue to copy data from the host to the device using the function

```
cl_int clEnqueueWriteBuffer( cl_command_queue queue,
                            cl_mem dev_dest,
                            cl_bool is_blocking,
                            size_t offset,
                            size_t size,
                            void* src,
                            cl_uint wait_list_num,
                            cl_event* wait_list,
                            cl_event* event );
```

This command puts in the command queue a task to copy data from the host pointer `src` to the device memory `dev_dest`. The `is_blocking` argument is used to indicate whether this command is blocking or nonblocking. By setting `is_blocking` to 1 or `CL_TRUE`, the main program will be paused until the memory transfer is complete. This may be desired if the host memory to be copied will be modified on the host shortly after this function call and there may be a race condition problem where the host modifies the data before the transfer is completed and the modified values are used in the transfer. Setting `is_blocking` to 0 or `CL_FALSE` means the main program can continue without waiting for the memory transfer to be completed. The `offset` allows the destination of the data to be somewhere else within the destination buffer. For example, if an array of type `double` of length 10 is allocated on the device and the data to be copied is to go in the second half of the array starting at index 5, then `offset` would be set to `5*sizeof(double)` corresponding to the fifth element in the array. The remaining arguments can be set to `NULL` for now. Events will be covered in Chapter 31.

On Line 83, the host data stored in `u` will be copied to the device memory `dev_u` upon completion. Since setting up a buffer with host data is a frequent task, Lines 81–84 can be combined into a single call using the extra flag `CL_MEM_COPY_HOST_PTR`:

```
81  cl_mem dev_u = clCreateBuffer(context,
82                      CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
83                      N*sizeof(double), u, &err);
```

The buffer for storing the result is also created in device memory in the variable `dev_du`. This time the memory is marked as *write only* because the GPU will not read from that buffer, only write the results out to it. Specifying the usage like this is a hint to the device that will help streamline the resulting code.

The kernel on Line 2 takes four arguments, namely, the input data, the number of entries in the data, the step size `dx`, and the location of the output data. Those four arguments are set up along with pointers to the memory space for each of the arguments using the function

```
cl_int clSetKernelArg( cl_kernel kernel,
                      cl_uint index,
                      size_t size,
                      void* value_ptr );
```

The `index` is the position in the argument list of the kernel, the `size` is the size of the data, and `value_ptr` is a pointer to either memory on the host or to a `cl_mem` object indicating device memory.

The kernel is now ready to be inserted into the command queue using the function

```
cl_int clEnqueueNDRangeKernel( cl_command_queue queue,
                               cl_kernel kernel,
                               cl_uint work_dim,
                               size_t* offset,
                               size_t* global_work_size,
                               size_t* local_work_size,
                               cl_uint wait_list_num,
                               cl_event* wait_list,
                               cl_event* event );
```

Here, `queue` is the command queue to be used and `kernel` is the kernel to be run on the GPU. Data can be arranged in up to three dimensions. The finite difference operation set here is naturally one-dimensional, so the `work_dim` is set to 1. The indexing of the global indices normally starts at zero, but the `global_work_offset` can be set to allow for indexing to start somewhere other than all zeros. The `global_work_size` describes the number of times the kernel is to be run. In this example, there are `N` grid points where the finite difference is computed, so the global work size is naturally set to be the array `{N, 0, 0}`. Since the `work_dim` is set to 1, only the first entry in the global work size is evaluated, and hence simply supplying a pointer to variable `N1` is sufficient. A pointer to `N` wouldn't work because it is an `int` and the argument is supposed to be of type `size_t`.

The next argument is important and is device dependent. Each device has a finite number of cores called work-items that it can simultaneously run on a kernel, and that number varies significantly between devices as shown in Tables 28.1–28.3. Because this is device dependent, it's wise to request the maximum work-group size from the device using the function `clGetDeviceInfo` as on Line 136 of Example 28.2 to set this value. This topic will be covered in Section 29.5, so for this example the work-group size is chosen to be 1, which will work but is not the most efficient. The remainder of the arguments for this function are concerned with events, which will be covered in Chapter 31. For now the last three arguments are set to `NULL`.

Running kernels on a GPU or other device is done asynchronously, so as soon as the kernel instructions are submitted, the program resumes with the next line. In this example, the program may try to read the results on Line 102 before they're ready, so the host execution is blocked while the kernel is running by using the `clFinish` function. It basically stops the program until the queue is emptied of instructions.

Once the data is ready, the next task is to retrieve the results back from the device. The companion function to `clEnqueueWriteBuffer` is the function

```
cl_int clEnqueueReadBuffer( cl_command_queue queue,
                           cl_mem dev_src,
                           cl_bool is_blocking,
                           size_t offset,
                           size_t size,
                           void* dest,
                           cl_uint wait_list_num,
                           cl_event* wait_list,
                           cl_event* event );
```

The arguments are the same as for `clEnqueueWriteBuffer` except this time `dev_src` is the device memory to be copied to the host memory at `dest`.

Finally, once the data is read from the device and the computation is finished, all the memory allocated must be released. The OpenCL objects, including all memory, kernel, program, queue, and context objects, must be released as is done on Lines 106–111. Finally, the memory allocated on the host is released in the usual manner.

The kernel function itself is on Lines 2–13. The first thing to notice about the kernel is the line

2	<code>#pragma OPENCL EXTENSION cl_khr_fp64 : enable \n</code>
---	---

Not all GPUs are capable of handling double precision, so by default double precision is turned off for OpenCL. To use double precision the `cl_khr_fp64` extension must be

enabled on Line 2. It must appear on a line by itself, which if this code were written in a file would be by default. Because this code is written in a string, the new line instruction must be embedded explicitly, hence the trailing “\n”. The body of the kernel itself begins by determining the index of the array that is to be computed by requesting the global index using the function `get_global_id` on Line 9. The argument for this function is the zero-based dimension in question, and since this is a one-dimensional problem, the argument is set to zero. The finite difference is computed for that index and stored in device memory on Line 12.

To see how this program scales compared to a serial code, see Figure 31.1.

## 29.2 • Compiling Kernel Functions

Returning to Lines 59–68, to compile a kernel, the program starts as a string variable that contains the code for the kernel. Constructing that string variable can be done in multiple ways. One option is to write the kernel as a string variable in a C code file as was done in Example 29.1. Note the use of quotation marks at the beginning and end of the kernel code and the “\\” character at the end of the lines, which is the continuation character for breaking string constants across multiple lines of a text file. If the string variable were inspected at runtime, those continuation characters would not appear. Nonetheless, they are required in this context because string literal constants are not permitted to have line breaks. It works well enough for this simple example but becomes cumbersome as the code gets more complex.

Alternatively, the code can be saved in a separate file, commonly using the “.cl” suffix to indicate it is an OpenCL kernel file. In this case, the kernel file would be written in very much the same manner as any other C function. Standard file I/O is then used to read the kernel file as a character string. If there are several kernels that require building, or have a kernel that is more substantial than a handful of lines, then this is a much more practical option.

Whichever way the kernel code is stored into a string variable, the code must be assembled into a program object using

```
cl_program clCreateProgramWithSource( cl_context context,
                                      cl_uint count,
                                      char* lines[],
                                      size_t* lengths,
                                      cl_int* err );
```

The kernel function may be broken into multiple strings, so `count` is the number of strings that define the complete kernel, `lines` is an array of string pointers, and `lengths` is an array of integers that give the lengths of each of the strings. In Example 29.1, there is only one string that defines the kernel. If that’s the case, then `NULL` can be passed for the `lengths` variable.

The program is then compiled using the function

```
cl_int clBuildProgram( cl_program program,
                       cl_uint num_devices,
                       cl_device_id* devices,
                       char* options,
                       void (CL_CALLBACK* func)(...),
                       void* data );
```

The program can be built for multiple devices, which will be covered in Section 31.4. Until then, `num_devices` is set to one and a pointer to the device in use is used for `devices`. A number of options can be added to the build using the `options` argument, but for now the default of using `NULL` is sufficient. The last two arguments are advanced topics not covered in this text and so both are set to `NULL` as well.

For robust code, it is good practice to check the error values returned by these OpenCL commands to make sure they completed without difficulties. It is sometimes convenient to ignore those values when writing code for speed, but think about how often your C code compiles without errors. Then consider that the compilation will be happening at runtime. With that in mind, it's important to check the error value and get feedback if there's a problem. Thus, Line 65 should be replaced with the following code in order to get any compiler error messages:

```

65 // Compile the kernel code
66 err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
67 if (err != CL_SUCCESS)
68 {
69     size_t len;
70
71     clGetProgramBuildInfo(program, device_id, CL_PROGRAM_BUILD_LOG,
72                           0, NULL, &len);
73     char* buffer = (char*)malloc(len*sizeof(char));
74
75     printf("Error: Failed to build program executable !\n");
76     clGetProgramBuildInfo(program, device_id, CL_PROGRAM_BUILD_LOG,
77                           len, buffer, NULL);
78     printf("%s\n", buffer);
79     free(buffer);
80     exit(1); // kernel didn't compile, stop the program
81 }
```

Should the kernel not compile successfully, then compiler messages indicating the problem can be recovered using the `clGetProgramBuildInfo` function. The first call to this function puts the length of the text buffer into the variable `len` so that the buffer can be allocated, and the second call retrieves the actual text and places it in the buffer. The messages are then printed to the screen before the code exits. This snippet of code will be your best friend in the early stages of learning OpenCL when the code doesn't compile.

Finally, the compiled code is turned into a kernel using the function

```
cl_kernel clCreateKernel( cl_program program,
                         char* name,
                         cl_int* err );
```

It is important to remember to set the `name` argument on Line 68 to the name of the kernel function on Line 2.

Example 29.2 shows how to put all these techniques together to modify Example 29.1. The kernel function is in a separate file called `diff.cl` and the compiler diagnostic steps have been added.

**Example 29.2.****File: diff.cl**

```
1 #pragma OPENCL EXTENSION cl_khr_fp64: enable
2
3 kernel void diff(global double* u,
4                   int N,
5                   double dx,
6                   global double* du)
7 {
8     size_t i = get_global_id(0);
9     int ip = (i+1)%N;
10    int im = (i+N-1)%N;
11    du[i] = (u[ip] - u[im])/dx/2.;
12 }
```

**File: main.c**

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4 #include <CL/opencl.h>
5
6 #ifndef M_PI
7 #define M_PI 3.1415926535897932384626433832795
8 #endif
9
10 char* GetKernelSource(char* filename)
11 {
12     // Open the OpenCL file
13     FILE* file = fopen(filename, "r");
14
15     // Move the file pointer to the end of the file
16     fseek(file, 0L, SEEK_END);
17
18     // Get the position of the file pointer relative to the beginning
19     size_t len = ftell(file);
20
21     // Move the file pointer back to the beginning of the file
22     rewind(file);
23
24     // Allocate space for the source file
25     char* srccode = (char*)malloc((len+1)*sizeof(char));
26
27     // Read the source code into the allocated memory
28     fread(srccode, sizeof(char), len, file);
29     // Be sure to terminate the string
30     srccode[len] = '\0';
31     fclose(file);
32     return srccode;
33 }
34
35 /*
36  int main(int argc, const char * argv[])
37
38 Demonstrate a simple example for implementing a
39 parallel finite difference operator
40 */
```

```

41  Inputs: argc should be 2
42  argv[1]: Length of the vector of data
43
44  Outputs: the initial data and its derivative.
45 */
46
47 int main(int argc, const char * argv[]) {
48
49  // Read the input values
50  int N = atoi(argv[1]); // Get the length of the vector from input
51
52  // Get platform info
53  cl_platform_id platform;
54  int err = clGetPlatformIDs(1, &platform, NULL);
55
56  // Request a GPU device
57  cl_device_id device_id;
58  err = clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device_id,
59                      NULL);
60
61  // Setup the context for the GPU
62  cl_context context = clCreateContext(0, 1, &device_id, NULL, NULL,
63                                      &err);
64
65  // Before accessing the GPU, we must create a dispatch queue, which is
66  // the sequence of commands that will be sent to the GPU.
67  cl_command_queue queue = clCreateCommandQueueWithProperties(context,
68  device_id, NULL, &err);
69
70  // In OpenCL, the kernel is compiled at runtime.
71  char* srccode = GetKernelSource("diff.cl");
72  cl_program program = clCreateProgramWithSource(context, 1,
73  (const char**)&srccode, NULL, &err);
74
75  // Compile the kernel code
76  err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
77  if (err != CL_SUCCESS)
78  {
79    size_t len;
80
81    clGetProgramBuildInfo(program, device_id, CL_PROGRAM_BUILD_LOG,
82                           0, NULL, &len);
83    char* buffer = (char*)malloc(len*sizeof(char));
84
85    printf("Error: Failed to build program executable !\n");
86    clGetProgramBuildInfo(program, device_id, CL_PROGRAM_BUILD_LOG,
87                           len, buffer, NULL);
88    printf("%s\n", buffer);
89    free(buffer);
90    exit(1); // kernel didn't compile, stop the program
91  }
92
93  // Create the kernel function from the compiled OpenCL code
94  cl_kernel kernel = clCreateKernel(program, "diff", &err);
95
96  // Allocate memory for the input arguments, data will be copied from
97  // the inputs to the GPU at the same time.
98  double dx = 2*M_PI/N;
99  double* u;
100 u = (double*)malloc(N*sizeof(double));
101 int i;
102 for (i=0; i<N; ++i)
103   u[i] = sin(i*dx);

```

```

104
105     double* du = (double*)malloc(N*sizeof(double));
106
107     // Transfer the input data to the device
108     cl_mem dev_u = clCreateBuffer(context, CL_MEM_READ_ONLY,
109                                     N*sizeof(double), NULL, NULL);
110     err = clEnqueueWriteBuffer(queue, dev_u, 1, 0, N*sizeof(double), u, 0,
111                               NULL, NULL);
112
113     // Set up the memory on the GPU where the answer will be stored
114     cl_mem dev_du = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
115                                     N*sizeof(double), NULL, NULL);
116
117     // Define all the arguments for the kernel function
118     err = clSetKernelArg(kernel, 0, sizeof(cl_mem), &dev_u);
119     err = clSetKernelArg(kernel, 1, sizeof(int), &N);
120     err = clSetKernelArg(kernel, 2, sizeof(double), &dx);
121     err = clSetKernelArg(kernel, 3, sizeof(cl_mem), &dev_du);
122
123     size_t maxwgs = 1;
124     size_t N1 = N;
125     err = clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &N1, &maxwgs, 0,
126                                  NULL, NULL);
127     clFinish(queue);
128
129     err = clEnqueueReadBuffer(queue, dev_du, 1, 0, N*sizeof(double), du,
130                               0, NULL, NULL);
131
132     // Free the memory allocated on the GPU
133     clReleaseMemObject(dev_du);
134     clReleaseMemObject(dev_u);
135     clReleaseKernel(kernel);
136     clReleaseProgram(program);
137     clReleaseCommandQueue(queue);
138     clReleaseContext(context);
139
140     // Free the memory on the CPU
141     free(u);
142     free(du);
143     free(srccode);
144
145     return 0;
146 }
```

Now that the kernel is moved into its own file, it has the same formatting as any other C function, which is far more intuitive and manageable than squeezing the program into a single character string. To get the kernel back as a string, a new function `GetKernelSource` on Line 10 is added to read a text file and store it into a string. It introduces some additional functions not covered in Section 4.2. In order to read the whole file, the length of the file is needed in order to allocate enough memory for its contents. When a file is opened, an implicit pointer to a location in the file is created, normally at the beginning of the file when it is opened. This implicit pointer points to the next character that will be read in from the file when a file is opened for reading. That implicit pointer is moved to the end of the file using the `fseek` function on Line 16. Once at the end, the `ftell` function gives the position of the implicit pointer relative to the beginning of the file on Line 19, so in other words, if the pointer is at the end, then `ftell` will return the length of the file. With that information in hand, memory is allocated to store the text of the file on Line 25. Note that one byte is added to the length

because space is needed for the terminating “\0” character at the end. Be sure to add the “\0” character at the end; forgetting this easily overlooked detail can lead to many inexplicable errors that can be very difficult to track down. On Line 22, the implicit pointer is moved back to the beginning of the file so that the subsequent `fread` gets the entire contents of the file.

It’s recommended that the function `GetKernelSource` be put in its own file with a corresponding header so that it can be reused in other OpenCL applications. From here on, it will be assumed that the function declaration will be in the header file `getkernel.h` and that the file name of the kernel will be the kernel function name with the “.cl” suffix.

## 29.3 • Error Handling

To build solid code, the error codes created by the various OpenCL functions should be checked and dealt with appropriately. Because of the low-level nature of the coding required to utilize a GPU, it is even more important to pay close attention to this detail. Fortunately, the OpenCL functions have a means of returning an error value either as a returned value or as an additional argument in the function argument list.

When an error value is returned as zero, it means there were no errors. Equivalently, the error value can be compared to the macro `CL_SUCCESS`. When the error value is nonzero, it means a problem occurred, and the value of the error will give a clue to the problem. To facilitate consistent use of error checking, a function `CHECK_ERROR` can be added that will handle error codes in the background. A header file `cl_handle_error.h` containing the following code makes it easier to check for errors during the execution of a program and allows for error checking to be turned on or off at compile time.

### Example 29.3.

#### `cl_handle_error.h`

```

1 #ifdef DO_ERROR_CHECKING
2 static void CHECK_ERROR( int err, const char* file, int line ) {
3
4     static const char* errstrings[] =
5     {
6         // Error Codes
7         "CL_SUCCESS",                                //  0
8         "CL_DEVICE_NOT_FOUND",                      // -1
9         "CL_DEVICE_NOT_AVAILABLE",                  // -2
10        "CL_COMPILER_NOT_AVAILABLE",                // -3
11        "CL_MEM_OBJECT_ALLOCATION_FAILURE",        // -4
12        "CL_OUT_OF_RESOURCES",                     // -5
13        "CL_OUT_OF_HOST_MEMORY",                   // -6
14        "CL_PROFILING_INFO_NOT_AVAILABLE",         // -7
15        "CL_MEM_COPY_OVERLAP",                    // -8
16        "CL_IMAGE_FORMAT_MISMATCH",                // -9
17        "CL_IMAGE_FORMAT_NOT_SUPPORTED",          // -10
18        "CL_BUILD_PROGRAM_FAILURE",               // -11
19        "CL_MAP_FAILURE",                        // -12
20        "",                                     // -13
21        "",                                     // -14
22        "",                                     // -15
23        "",                                     // -16
24        "",                                     // -17
25        ""                                     // -18

```

```
26      " " , // -19
27      " " , // -20
28      " " , // -21
29      " " , // -22
30      " " , // -23
31      " " , // -24
32      " " , // -25
33      " " , // -26
34      " " , // -27
35      " " , // -28
36      " " , // -29
37      "CL_INVALID_VALUE" , // -30
38      "CL_INVALID_DEVICE_TYPE" , // -31
39      "CL_INVALID_PLATFORM" , // -32
40      "CL_INVALID_DEVICE" , // -33
41      "CL_INVALID_CONTEXT" , // -34
42      "CL_INVALID_QUEUE_PROPERTIES" , // -35
43      "CL_INVALID_COMMAND_QUEUE" , // -36
44      "CL_INVALID_HOST_PTR" , // -37
45      "CL_INVALID_MEM_OBJECT" , // -38
46      "CL_INVALID_IMAGE_FORMAT_DESCRIPTOR" , // -39
47      "CL_INVALID_IMAGE_SIZE" , // -40
48      "CL_INVALID_SAMPLER" , // -41
49      "CL_INVALID_BINARY" , // -42
50      "CL_INVALID_BUILD_OPTIONS" , // -43
51      "CL_INVALID_PROGRAM" , // -44
52      "CL_INVALID_PROGRAM_EXECUTABLE" , // -45
53      "CL_INVALID_KERNEL_NAME" , // -46
54      "CL_INVALID_KERNEL_DEFINITION" , // -47
55      "CL_INVALID_KERNEL" , // -48
56      "CL_INVALID_ARG_INDEX" , // -49
57      "CL_INVALID_ARG_VALUE" , // -50
58      "CL_INVALID_ARG_SIZE" , // -51
59      "CL_INVALID_KERNEL_ARGS" , // -52
60      "CL_INVALID_WORK_DIMENSION" , // -53
61      "CL_INVALID_WORK_GROUP_SIZE" , // -54
62      "CL_INVALID_WORK_ITEM_SIZE" , // -55
63      "CL_INVALID_GLOBAL_OFFSET" , // -56
64      "CL_INVALID_EVENT_WAIT_LIST" , // -57
65      "CL_INVALID_EVENT" , // -58
66      "CL_INVALID_OPERATION" , // -59
67      "CL_INVALID_GL_OBJECT" , // -60
68      "CL_INVALID_BUFFER_SIZE" , // -61
69      "CL_INVALID_MIP_LEVEL" , // -62
70      "CL_INVALID_GLOBAL_WORK_SIZE" , // -63
71      "CL_UNKNOWN_ERROR_CODE"
72  };
73
74  if (err != CL_SUCCESS) {
75      int errnum = err >= -63 && err <= 0 ? -err : 64;
76      printf("Error %s in %s at line %d\n", errstrings[errnum],
77             file , line);
78      exit(1);
79  }
80 }
81 #define CheckError( err ) (CHECK_ERROR(err , __FILE__ , __LINE__))
82 #define CheckErrorValue( err ) (CHECK_ERROR(err , __FILE__ , __LINE__))
83 #else
84 #define CheckError( err ) ( err )
85 #define CheckErrorValue( err )
86 #endif
```

In the examples thus far, the variable name `err` has been used to collect the error values. Some functions output `err` as a return value, and some functions assign the error value through the argument list. To handle both cases, two macros are created on Lines 81 and 82. Use `CheckError` when it is a function that returns an error value, and use `CheckErrorValue` when it is passed back through the argument list. For example,

```
CheckError( clBuildProgram(program, 0, NULL, NULL, NULL, NULL) );
cl_kernel kernel = clCreateKernel(program, "mykernel", &err);
CheckErrorValue(err);
```

When an error is detected, the function in this header prints out the error code and the file and line number where it occurred and then terminates the program. In practice, the proper thing to do is to respond to the problem according to the situation by taking care of cleaning up any allocated memory, for example, but that would require a specialized error handler. This function will suffice for the purposes of this text.

Since the C preprocessor has not been discussed in any great detail, the usage in `cl_handle_error.h` requires some discussion. The first directive in this example is an `#ifdef` on Line 1, which tells the compiler that if the macro `DO_ERROR_CHECKING` is defined as a macro, then compile Lines 2–82. If the macro is *not* defined, then compile the code after the `#else` clause, which would be Lines 84 and 85 where the conditional compiling is terminated with `#endif`. Thus, there are two ways to turn on error checking; the first is to insert the macro definition `#define DO_ERROR_CHECKING` before the `cl_handle_error.h` file is included:

```
#define DO_ERROR_CHECKING
#include "cl_handle_error.h"
```

The alternative, and preferred, method is to add the option `-DDO_ERROR_CHECKING` to the compile line with the “`-D`” option:

```
$ gcc -o myprog myprog.c -DDO_ERROR_CHECKING
```

Returning to Line 81, the “function” `CheckError` is in fact not a function at all, but a macro that can also take an argument. If `DO_ERROR_CHECKING` is defined, then when the preprocessor sees the text

```
CheckError(X);
```

it is replaced with

```
(CHECK_ERROR(X, __FILE__, __LINE__));
```

The compiler-defined strings “`__FILE__`” and “`__LINE__`” are also identified by the preprocessor and are substituted for the name of the current file and the line number within that file, respectively. For example, the preprocessor will replace

```
17 CheckError( clBuildProgram(program, 0, NULL, NULL, NULL, NULL) );
```

with

```
( CHECK_ERROR( clBuildProgram(program, 0, NULL, NULL, NULL, NULL),
              "filename.c", 17 ));
```

Now, `CHECK_ERROR` is a regular function defined on Line 2. If the value of the first argument is not `CL_SUCCESS`, then it prints an error message from the list of error strings and exits the program, which is at a minimum what an error handler should do.

Example 30.1 shows how to use the `CheckError` and `CheckErrorValue` functions in practice, but for space considerations, the remaining examples will not.

## 29.4 • Organization of Work-Groups

Example 29.4 illustrates the use of multiple dimensions by computing the Laplacian on a two-dimensional grid. It is similar to the one-dimensional version of Example 29.2, where the data is assumed to be periodic in both the  $x$ - and  $y$ -directions.

### Example 29.4.

#### File: diff.cl

```

1 #pragma OPENCL EXTENSION cl_khr_fp64: enable
2
3 kernel void diff(global double* u,
4                   double dx,
5                   double dy,
6                   global double* du)
7 {
8     size_t M = get_global_size(0);
9     size_t N = get_global_size(1);
10    size_t i = get_global_id(0);
11    size_t j = get_global_id(1);
12    int ij00 = i+j*M;
13    int ijp0 = (i+1)%M+j*M;
14    int ijm0 = (i+N-1)%M+j*M;
15    int ij0p = i+((j+1)%N)*M;
16    int ij0m = i+((j+N-1)%N)*M;
17    du[ij00] = (u[ijp0] - 2*u[ij00] + u[ijm0])/dx/dx
18                  + (u[ij0p] - 2*u[ij00] + u[ij0m])/dy/dy;
19 }
```

#### File: main.c

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4 #include <CL/opencl.h>
5 #include "getkernel.h"
6
7 #ifndef M_PI
8 #define M_PI 3.1415926535897932384626433832795
9 #endif
10 /*
11  * int main(int argc, const char * argv[])
12  * Demonstrate a simple example for implementing a
13  * parallel finite difference operator on a 2D array
14  *
15  * Inputs: argc should be 3
16  * argv[1]: Size of first dimension
17  * argv[2]: Size of second dimension
18  *
19  * Outputs: the initial data and its derivative.
20  */
21
22
```

```

23 int main(int argc, const char * argv[]) {
24
25     // Read the input values
26     int M = atoi(argv[1]); // Get the length of the vector from input
27     int N = atoi(argv[2]);
28
29     // Request platform ID
30     cl_platform_id platform;
31     int err = clGetPlatformIDs(1, &platform, NULL);
32
33     // Request a GPU device
34     cl_device_id device_id;
35     err = clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device_id,
36                         NULL);
37
38     // Setup the context for the GPU
39     cl_context context = clCreateContext(0, 1, &device_id, NULL, NULL,
40                                         &err);
41
42     // Set up the command queue
43     cl_command_queue queue = clCreateCommandQueueWithProperties(context,
44                                                               device_id, NULL, &err);
45
46     // Load the source code from a file
47     char* srccode = GetKernelSource("diff.cl");
48     cl_program program = clCreateProgramWithSource(context, 1,
49                                                   (const char**)(&srccode), NULL, &err);
50
51     // Compile the kernel code
52     err = clBuildProgram(program, 0, NULL, NULL, NULL);
53
54     // Create the kernel function from the compiled OpenCL code
55     cl_kernel kernel = clCreateKernel(program, "diff", &err);
56
57     // Allocate memory for the input arguments and initialize
58     double dx = 2*M_PI/M;
59     double dy = 2*M_PI/N;
60     double* u = (double*)malloc(M*N*sizeof(double));
61     int i, j;
62     for (i=0; i<M; ++i)
63         for (j=0; j<N; ++j)
64             u[i+j*M] = sin(i*dx)*cos(j*dy);
65
66     double* du = (double*)malloc(M*N*sizeof(double));
67     cl_mem dev_u = clCreateBuffer(context, CL_MEM_READ_ONLY,
68                                   M*N*sizeof(double), NULL, NULL);
69     err = clEnqueueWriteBuffer(queue, dev_u, 1, 0, M*N*sizeof(double), u,
70                               0, NULL, NULL);
71
72     // Set up the memory on the GPU where the answer will be stored
73     cl_mem dev_du = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
74                                   M*N*sizeof(double), NULL, NULL);
75
76     // Define all the arguments for the kernel function
77     err = clSetKernelArg(kernel, 0, sizeof(cl_mem), &dev_u);
78     err = clSetKernelArg(kernel, 1, sizeof(double), &dx);
79     err = clSetKernelArg(kernel, 2, sizeof(double), &dy);
80     err = clSetKernelArg(kernel, 3, sizeof(cl_mem), &dev_du);
81
82     // Execute the kernel
83     size_t dims[2] = {M, N};
84     size_t wgs[2] = {1, 1};
85     err = clEnqueueNDRangeKernel(queue, kernel, 2, NULL, dims, wgs, 0,
```

```

86                                     NULL, NULL);
87     clFinish(queue);
88
89     // Retrieve the results from the device
90     err = clEnqueueReadBuffer(queue, dev_du, 1, 0, M*N*sizeof(double), du,
91                               0, NULL, NULL);
92
93     // Free the memory allocated on the GPU
94     clReleaseMemObject(dev_du);
95     clReleaseMemObject(dev_u);
96     clReleaseKernel(kernel);
97     clReleaseProgram(program);
98     clReleaseCommandQueue(queue);
99     clReleaseContext(context);
100
101    // Free the memory on the CPU
102    free(u);
103    free(du);
104    free(srccode);
105
106    return 0;
107 }

```

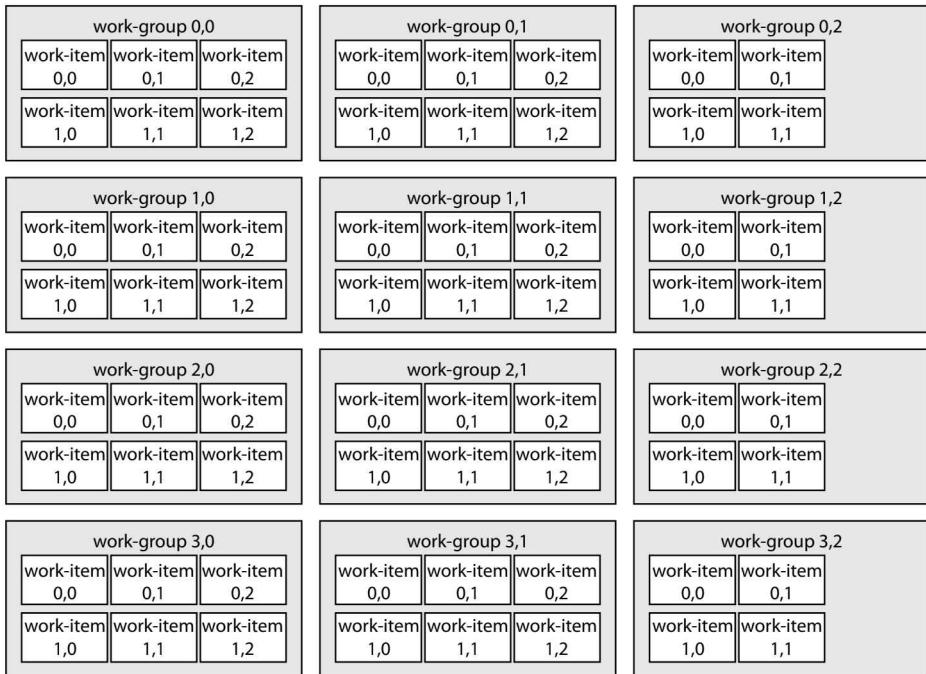
Even though a two-dimensional array is desired for an application like this, the array is still being allocated on Line 67 in the manner of a one-dimensional array. Unfortunately, even though the device can be divided into different dimensions, the data must still be linear. The indexing of the array can be seen on Line 12 where the  $(i, j)$  coordinates convert to the linear coordinate  $i+j*M$ , where  $M$  is the length of the first dimension in the dataset. This means that the data is saved in column-major order. The data could also be saved in row-major order, and in that case the linear coordinate would be  $i*N+j$ . Either method works, so long as the usage is consistent.

On Line 83, the dimensions of the work-group are specified and they conform to the number of grid points in the domain, namely,  $M \times N$ . It is not necessary to pass those dimensions to the kernel because the kernel can retrieve the dimensions using the function `get_global_size` on Line 8 where the argument is the dimension. The indices for this particular invocation of the kernel are retrieved using the `get_global_id` function on Line 10, again the argument corresponding to the dimension.

## 29.5 • Work-Items

For the key examples up to this point, Examples 29.1 and 29.4, the kernel was executed the number of times specified by the `N1` argument on Line 97 of Example 29.1 or by the `dims` argument on Line 83 of Example 29.4. In each case, the next argument was specified as 1 or the array `{1, 1}`. This means that the work-groups were all done separately, which is not very efficient.

It is far more efficient to group the work into work-groups, which are collections of work-items, where the work-items are able to share a small amount of memory within a work-group. The organization of work-groups and work-items is illustrated in Figure 29.1. The work-item dimension information is specified by the work-group size, which is the sixth argument of the `clEnqueueNDRangeKernel` function. Figure 29.1 illustrates the organization of the global work size subdivided into work-groups and work-items for a two-dimensional domain. If the number of work-items in a work-group does not divide evenly into the global work size, then a final work-group of a



**Figure 29.1.** Diagram of the organization of work-items within work-groups with both in arrays of different dimensions. The dimensions of the work-items arrays need not be the same as the work-groups. For the shown organization, the global work size submitted is  $8 \times 8$  with each work-group consisting of  $2 \times 3$  work-items. Thus, the work-groups must have dimensions  $4 \times 3$  to cover the full  $8 \times 8$  grid. The work-groups in the third column are not full sized because 8 is not evenly divisible by 3.

smaller size is used to complete the global work size. The most efficiency is gained when the work-group sizes are as large as possible within the constraints of the hardware and when the number of work-items in the work-group divides evenly into the global work size so that there are no stray work-groups.

If the dimensions of the work-group size are not important for the kernel, then specifying the work-group size as `NULL` will let OpenCL choose the work-group size. For many kernels, the actual arrangement of work-groups and work-items is important for algorithm development, so when it matters the work-group size must be specified. Different GPU cards have different work-group size limitations. For example, the AMD Radeon HD D300 has maximum work-group size 256 ( $= 16 \times 16$ ), while the NVIDIA Tesla K20c has 1024 ( $= 32 \times 32$ ). When manually choosing the dimensions of the work-group size, there are two parameters that should be checked using the `clGetDeviceInfo` function. The first is the maximum work-group size, which can be obtained using the `CL_DEVICE_MAX_WORK_GROUP_SIZE` option as on Line 136 of Example 28.2, and the second is the `CL_DEVICE_MAX_WORK_ITEM_SIZES` on Line 145 of the same example. Suppose the work-group size chosen has three dimensions of size  $\ell \times m \times n$ . Suppose also the maximum work-group size is determined to be `maxwgs`, and the maximum work-group dimensions are in the array `wdim[3]` as obtained from `clGetDeviceInfo`. Then to be a valid work-group size, the product must satisfy  $\ell mn \leq \text{maxwgs}$  and each dimension must also be within the constraints, in other words,  $\ell \leq \text{wdim}[0]$ ,  $m \leq \text{wdim}[1]$ , and  $n \leq \text{wdim}[2]$ .

For example, to specify the number of work-items to be a  $16 \times 16$  array per work-group, change Lines 82–86 of Example 29.4 to

```
82 // Execute the kernel
83 size_t dims[2] = {M, N};
84 size_t wgs[2] = {16,16};
85 err = clEnqueueNDRangeKernel(queue, kernel, 2, NULL, dims, wgs, 0,
86                               NULL, NULL);
```

where `wgs` is the number of work-items per work-group.

To get the local information within a kernel, the local work-group size can be obtained by calling `get_local_size`. The local work-item index is obtained with the function `get_local_id`. The function `get_num_groups` returns the number of work-groups, and the index of a work-group is obtained by the function `get_group_id`. All these functions take the dimension of the argument similar to how `get_global_id` is used.

---

## Exercises

- 29.1. Write a kernel that fills out a two-dimensional array of integers of dimension  $100 \times 100$ . The kernel should save the value of the function

```
value = 100*(100*(100* get_global_id(0)+get_global_id(1))+  
           get_local_id(0))+get_local_id(1);
```

Experiment with different size work-groups and verify that the values returned are as expected. Also try setting the work-group size to `NULL` to see how the OpenCL system decides to divide up the work-groups.

- 29.2. Modify Example 29.4 to add all the error checking. Try turning the error checking on and off by using or not using the “-D” compiler flag. Modify an argument to one of the functions to generate an error to see how the error checking works.



# Chapter 30

# GPU Memory

The examples up to this point have exclusively used global memory on the GPU to do the computations, which is the slowest kind of memory. There are other choices for higher speed memory that will improve performance. OpenCL breaks down memory into global, local, and constant. Local and constant memory are the fastest, so it is important to take advantage of that memory where appropriate. This chapter explores the use of local and constant memory.

## 30.1 • Local Memory

Though it is highly device dependent, one bottleneck in the codes is not the number of work-items being applied to the task, nor how those work-items are organized, but the time required to retrieve the necessary data from memory and to store the result back into memory. When the work-items are all trying to access global memory latency arises due to all the requests happening simultaneously. As a result they are effectively forced to queue up sequentially, resulting in less than optimal performance. For the simple finite difference kernel used as a base example in this text, the gain from using local memory is not dramatic. For example, compare the results for GPU-shared mem that uses local memory versus GPU-work-items that doesn't in Figures 31.1 and 31.2. For more complex kernels the advantage could be important.

Each work-group has an allocation of local memory assigned to it for which read and write times are significantly shorter than read/write to global memory. Checking Tables 28.1–28.3, the amount of available local memory is much smaller than the available global memory. Local memory in a kernel is identified by the attribute label `local` in front of the variable declaration. The amount of data used in local memory can be determined either statically or dynamically. For static allocation, the variable would be declared in the kernel like this:

```
local float localmem [256];
```

where this creates an array of 256 floating point numbers. To create the array dynamically, the variable is declared in the argument list of the kernel and the memory is allocated using the `clSetKernelArg` function using `NULL` for the last argument.

For example, a kernel with a dynamically allocated array of `double` would be declared in the kernel as

```
kernel void myfunc(local double* data);
```

and then the memory is allocated in the host code,

```
clSetKernelArg(kernel, 0, N*sizeof(double), NULL);
```

where an array of `N` double precision values is created in the local memory space. Examples of this type of usage can be found in Example 30.1.

The whole purpose of local memory is to take advantage of faster memory access, and to do this, the transfer of data to/from global memory will be done in parallel as well. To see how this is done, consider the following finite difference kernel `diff` that is a modified version of the kernel in Example 29.2.

### Example 30.1.

#### File: diff.cl

```

1 #pragma OPENCL EXTENSION cl_khr_fp64: enable
2 /*
3  * kernel diff
4  * Finite difference operator on a 1D array
5  *
6  * Inputs:
7  *   double* u: grid data
8  *   int N: Size of grid
9  *
10 * Outputs:
11 *   double* du: finite difference of u
12 */
13
14
15 kernel void diff(global double* u,
16                   int N,
17                   double dx,
18                   local double* localu,
19                   local double* localdu,
20                   global double* du)
21 {
22     size_t g_i = get_global_id(0);
23     size_t l_i = get_local_id(0)+1;
24     int g_ip = (g_i+1)%N;
25     int g_im = (g_i+N-1)%N;
26     localu[l_i] = u[g_i];
27     if (l_i == 1)
28         localu[0] = u[g_im];
29     if (l_i == get_local_size(0))
30         localu[l_i+1] = u[g_ip];
31     localdu[l_i-1] = (localu[l_i+1] - localu[l_i-1])/dx/2.;
32     du[g_i] = localdu[l_i-1];
33 }
```

**File: main.c**

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4 #include <CL/opencl.h>
5 #include "getkernel.h"
6 #include "cl_handle_error.h"
7
8 #ifndef M_PI
9 #define M_PI 3.1415926535897932384626433832795
10#endif
11
12/*
13 Demonstrate a simple example for implementing a
14 parallel finite difference operator
15
16 Inputs: argc should be 2
17 argv[1]: Length of the domain
18
19 Outputs: the initial data and its derivative.
20*/
21
22int main(int argc, const char * argv[]) {
23
24    // Read the input values
25    int N = atoi(argv[1]); // Get the length of the vector from input
26    int err;
27
28    // Request the platform info
29    cl_platform_id platform;
30    CheckError( clGetPlatformIDs(1, &platform, NULL) );
31
32    // Request a GPU device
33    cl_device_id device_id;
34    CheckError( clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1,
35                                &device_id, NULL) );
36
37    // Setup the context for the GPU
38    cl_context context = clCreateContext(0, 1, &device_id, NULL, NULL,
39                                         &err);
40    CheckErrorValue(err);
41
42    // Before accessing the GPU, we must create a dispatch queue, which is
43    // the sequence of commands that will be sent to the GPU.
44    cl_command_queue queue = clCreateCommandQueueWithProperties(context,
45                                                                device_id, NULL, &err);
46    CheckErrorValue(err);
47
48    // In OpenCL, the kernel is compiled at runtime,
49    // try to do this only once and then save it.
50    char* srccode = GetKernelSource("diff.cl");
51    cl_program program = clCreateProgramWithSource(context, 1,
52                                                   (const char**)&srccode, NULL, &err);
53    CheckErrorValue(err);
54
55    // Compile the kernel code
56    CheckError( clBuildProgram(program, 0, NULL, NULL, NULL, NULL) );
57
58    // Create the kernel function from the compiled OpenCL code
59    cl_kernel kernel = clCreateKernel(program, "diff", &err);
60    CheckErrorValue(err);
```

```

61
62 // Allocate memory for the input arguments, data will be copied from
63 // the inputs to the GPU at the same time.
64 double dx = 2*M_PI/N;
65 double* u;
66 u = (double*)malloc(N*sizeof(double));
67 int i;
68 for (i=0; i<N; ++i)
69     u[i] = sin(i*dx);
70
71 double* du = (double*)malloc(N*sizeof(double));
72 cl_mem dev_u = clCreateBuffer(context, CL_MEM_READ_ONLY,
73                                 N*sizeof(double), NULL, &err);
74 CheckErrorValue(err);
75 CheckError( clEnqueueWriteBuffer(queue, dev_u, 1, 0, N*sizeof(double),
76                                  u, 0, NULL, NULL) );
77
78 // Set up the memory on the GPU where the answer will be stored
79 cl_mem dev_du = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
80                                 N*sizeof(double), NULL, &err);
81 CheckErrorValue(err);
82
83 size_t maxwgs;
84 CheckError( clGetDeviceInfo(device_id, CL_DEVICE_MAX_WORK_GROUP_SIZE,
85                             sizeof(size_t), &maxwgs, NULL) );
86
87 // Define all the arguments for the kernel function
88 CheckError( clSetKernelArg(kernel, 0, sizeof(cl_mem), &dev_u) );
89 CheckError( clSetKernelArg(kernel, 1, sizeof(int), &N) );
90 CheckError( clSetKernelArg(kernel, 2, sizeof(double), &dx) );
91 CheckError( clSetKernelArg(kernel, 3, (maxwgs+2)*sizeof(double),
92                         NULL) );
93 CheckError( clSetKernelArg(kernel, 4, maxwgs*sizeof(double), NULL) );
94 CheckError( clSetKernelArg(kernel, 5, sizeof(cl_mem), &dev_du) );
95
96 size_t N1 = N;
97 CheckError( clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &N1,
98                                     &maxwgs, 0, NULL, NULL) );
99 CheckError( clFinish(queue) );
100
101 CheckError( clEnqueueReadBuffer(queue, dev_du, 1, 0, N*sizeof(double),
102                                du, 0, NULL, NULL) );
103
104 // Free the memory allocated on the GPU
105 CheckError( clReleaseMemObject(dev_du) );
106 CheckError( clReleaseMemObject(dev_u) );
107 CheckError( clReleaseKernel(kernel) );
108 CheckError( clReleaseProgram(program) );
109 CheckError( clReleaseCommandQueue(queue) );
110 CheckError( clReleaseContext(context) );
111
112 // Free the memory on the CPU
113 free(u);
114 free(du);
115 free(srccode);
116
117 return 0;
118 }

```

The local memory is declared in the kernel on Lines 18, 19 of `diff.cl`, and the amount of memory allocated is specified in `main.c` on Lines 91–93.

global u, du

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
---	---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----	----	----	----	----

local u

work-group id = 0

23	0	1	2	3	4
----	---	---	---	---	---

work-group id = 1

3	4	5	6	7	8
---	---	---	---	---	---

work-group id = 2

7	8	9	10	11	12
---	---	---	----	----	----

work-group id = 3

11	12	13	14	15	16
----	----	----	----	----	----

work-group id = 4

15	16	17	18	19	20
----	----	----	----	----	----

work-group id = 5

19	20	21	22	23	0
----	----	----	----	----	---

local du

work-group id = 0

0	1	2	3
---	---	---	---

work-group id = 1

4	5	6	7
---	---	---	---

work-group id = 2

8	9	10	11
---	---	----	----

work-group id = 3

12	13	14	15
----	----	----	----

work-group id = 4

16	17	18	19
----	----	----	----

work-group id = 5

20	21	22	23
----	----	----	----

**Figure 30.1.** Illustration of the memory layout for local memory in Example 30.1.

Suppose in this example there are six work-groups, each with four work-items; then the local and global memory layout for a data array of length 24 is illustrated in Figure 30.1. The logic in this kernel can be broken down into three stages: (1) transfer a piece of global memory to local memory, (2) do the local computation for this work-item, (3) move the result from local memory to global memory. In Example 30.1, the local and global indices into the data array are obtained on Lines 22 and 23 of `diff.c1`. Stage 1 of the kernel, where the global memory is copied to the local memory, is on Lines 26–30. Here, each work-item copies its corresponding global memory location to the local memory address on Line 26. However, because this is a finite difference approximation with a three-point stencil, the two data points on either side are needed. Therefore, the first and last work-items in the work-group additionally copy the neighboring value from the global data so that the finite difference calculation within any work-item is confined to the local memory and doesn't have to retrieve global memory for any reason. For example, using Figure 30.1, work-item 0 of work-group 1 will copy both global data points  $u[4]$  and  $u[3]$  and put them in local data with indices 1 and 0, respectively.

For stage 2, the finite difference calculation uses entirely local memory on Line 31.

Finally, stage 3 is on Line 32, where the locally stored solution is copied back to global memory so it can be accessed by the host. In this direction, the overlap points are not computed, so the extra copies set up in stage 1 are not needed.

Unfortunately, the kernel in Example 30.1 may not work correctly. Suppose work-item 1 starts to do its finite difference computation before work-item 0 finishes doing its copy from global to local memory; then the finite difference computed in work-item 1 may be using data from the local memory that hasn't been initialized yet and hence have random data in it. In order to guarantee that the local memory is ready before computing the finite difference, all the work-items must certify the copy from global to local memory is completed. This is accomplished by syncing the work-items using

the `barrier(CLK_LOCAL_MEM_FENCE)` function. This command is similar in spirit to the `MPI_Wait` command from MPI or the `__syncthreads` function in CUDA, where in this case, all work-items of the given work-group will stop and wait at the barrier until all work-items reach that point. When using `barrier(CLK_LOCAL_MEM_FENCE)`, be sure that all threads can reach it. For example, if the `barrier` is inside an `if` statement where only some of the work-items reach it, then the program will block waiting for the work-items that can never reach the barrier. The corrected kernel is shown in Example 30.2.

### Example 30.2.

```

1 #pragma OPENCL EXTENSION cl_khr_fp64 : enable
2
3 kernel void diff(global double* u,
4                  int N,
5                  double dx,
6                  local double* localu ,
7                  local double* localdu ,
8                  global double* du)
9 {
10    size_t g_i = get_global_id(0);
11    size_t l_i = get_local_id(0)+1;
12    int g_ip = (g_i+1)%N;
13    int g_im = (g_i+N-1)%N;
14    localu[l_i] = u[g_i];
15    if (l_i == 1)
16        localu[0] = u[g_im];
17    if (l_i == get_local_size(0))
18        localu[l_i+1] = u[g_ip];
19
20    barrier(CLK_LOCAL_MEM_FENCE);
21
22    localdu[l_i-1] = (localu[l_i+1] - localu[l_i-1])/dx/2.;
23    du[g_i] = localdu[l_i-1];
24 }
```

#### 30.1.1 • Accessing Data

When transferring global memory to local memory, it is far more efficient to access contiguous memory as opposed to noncontiguous. In fact this is true more generally when accessing global memory. To illustrate how this can affect performance, consider a large  $N \times N$  array, stored in row-major order. The following kernel would add one to each entry in a specified row of the matrix:

```

#pragma OPENCL EXTENSION cl_khr_fp64 enable

kernel void addone(global double* u, int row)
{
    int i = get_global_size(0)*row + get_global_id(0);
    u[i] = u[i] + 1.;
```

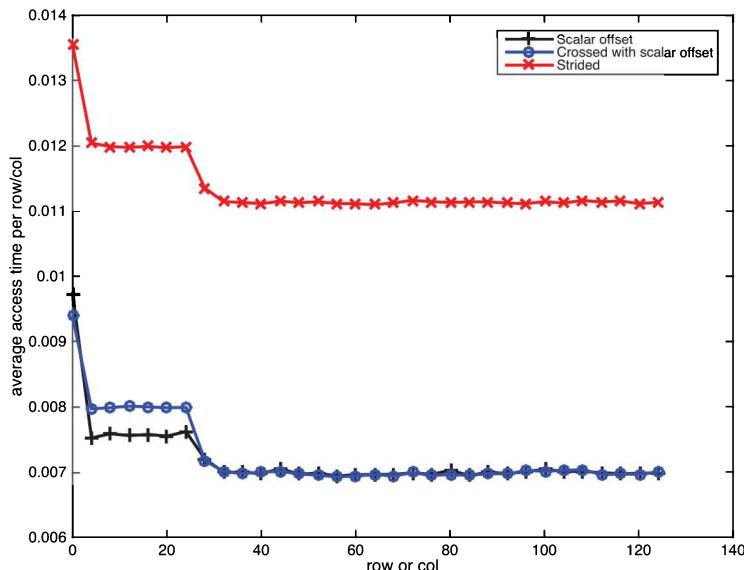
Because `get_global_id` is not multiplied by anything, consecutive work-items would generate consecutive values of `i` and hence would access consecutive locations of memory in the array `u`. On the other hand, to access the rows for a fixed column, the data is strided, i.e., the index `j` would be given by

<code>int j = get_global_size(0)*get_global_id(0) + column;</code>
--

In this case, consecutive work-items will access memory  $N$  elements apart. What is important is how large the stride is, so for the fixed row, the stride is one, while for the fixed column, the stride is  $N$ . To show that it's only memory address proximity, consider a third case where the index  $i$  exchanges even columns with odd:

```
int i = get_global_size(0)*row + get_global_id(0)
      + (threadIdx.x%2 ? -1 : 1);
```

The results of simply loading data using these indices are illustrated in Figure 30.2. Thus, if using a global work size that is  $N \times N$  to access a two-dimensional matrix, then it's best to use an  $N \times 1$  work-group size so that the work-items in the same work-group are accessing contiguous memory whenever possible to improve performance.



**Figure 30.2.** Comparison of efficiency of memory access with different strides. An entire row or column of a  $2048 \times 2048$  matrix is accessed. Scalar offset corresponds to accessing a fixed row with a stride of one. Crossed with scalar offset has stride one but where odd indices access even addresses and vice versa. Strided accesses a fixed column so that `global_work_id(0)` is multiplied by  $N$ . When stride is one, memory access is coalesced resulting in 50% faster memory access.

## 30.2 • Constant Memory

Constant memory is another form of fast-access, low-latency memory available on the GPU. For OpenCL certification, GPUs are required to have at least 64Kb of constant memory. This memory space is useful for values that are common within a work-group. Dynamically allocated constant memory still goes through the argument list, but it is designated with the `constant` label, moving it into constant memory space.

Of course, constants can be defined statically at compile time within the kernel as well, and those also count against the constant memory allocation. Example 30.3 shows how the constant memory is set up and used within the kernel function. In this case, the

input data  $u$  is assumed to be small enough to fit in the constant memory space and the parameters  $N$  and  $dx$  are the same for all work-items.

### Example 30.3.

#### File: diff.cl

```

1 #pragma OPENCL EXTENSION cl_khr_fp64 : enable
2
3 /*
4  * kernel diff
5  * Finite difference operator on a 1D array
6
7  * Inputs:
8  *   double* u: grid data
9  *   int N: Size of grid
10 *   double dx: grid space step size
11
12 * Outputs:
13 *   double* du: finite difference of u
14 */
15
16 kernel void diff(constant double* u,
17                   constant int* N,
18                   constant double* dx,
19                   global double* du)
20 {
21     size_t g_i = get_global_id(0);
22     size_t l_i = get_local_id(0);
23     int g_ip = (g_i+1)% *N;
24     int g_im = (g_i+*N-1)% *N;
25     du[g_i] = (u[g_ip] - u[g_im])/dx/2.;
26 }
```

#### File: main.c

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4 #include <CL/opencl.h>
5 #include "getkernel.h"
6
7 #ifndef M_PI
8 #define M_PI 3.1415926535897932384626433832795
9 #endif
10
11 /*
12  * Demonstrate a simple example for implementing a
13  * parallel finite difference operator
14
15  * Inputs: argc should be 2
16  * argv[1]: Length of the domain
17
18  * Outputs: the initial data and its derivative.
19 */
20
21 int main(int argc, const char * argv[]) {
22 }
```

```

23 // Read the input values
24 int N = atoi(argv[1]); // Get the length of the vector from input
25
26 // Request the platform ID
27 cl_platform_id platform;
28 clGetPlatformIDs(1, &platform, NULL);
29
30 // Request a GPU device
31 cl_device_id device_id;
32 int err = clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device_id,
33 NULL);
34
35 // Set up the context for the GPU
36 cl_context context = clCreateContext(0, 1, &device_id, NULL, NULL,
37 &err);
38
39 // Before accessing the GPU, we must create a dispatch queue, which is
40 // the sequence of commands that will be sent to the GPU.
41 cl_command_queue queue = clCreateCommandQueueWithProperties(context,
42 device_id, NULL, &err);
43
44 // Before going any further, make sure that we won't exceed the
45 // constant memory buffer
46 cl_ulong constmax_mem;
47 clGetDeviceInfo(device_id, CL_DEVICE_MAX_CONSTANT_BUFFER_SIZE,
48 sizeof(cl_ulong), &constmax_mem, NULL);
49 if ((N+2)*sizeof(double) > constmax_mem) {
50     printf("Oops, array is too large for constant memory buffer.\n");
51     exit(1);
52 }
53
54 // Prep the kernel source
55 char* srccode = GetKernelSource("diff.cl");
56 cl_program program = clCreateProgramWithSource(context, 1,
57 (const char**)&srccode, NULL, &err);
58
59 // Compile the kernel code
60 err = clBuildProgram(program, 0, NULL, NULL, NULL);
61
62 // Create the kernel function from the compiled OpenCL code
63 cl_kernel kernel = clCreateKernel(program, "diff", &err);
64
65 // Allocate memory for the input arguments, data will be copied from
66 // the inputs to the GPU at the same time.
67 double dx = 2*M_PI/N;
68 double* u;
69 u = (double*)malloc(N*sizeof(double));
70 int i;
71 for (i=0; i<N; ++i)
72     u[i] = sin(i*dx);
73
74 double* du = (double*)malloc(N*sizeof(double));
75 cl_mem dev_u = clCreateBuffer(context,
76 CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
77 N*sizeof(double), u, NULL);
78
79 cl_mem dev_N = clCreateBuffer(context,
80 CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
81 sizeof(int), &N, NULL);
82
83 cl_mem dev_dx = clCreateBuffer(context,
84 CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
85 sizeof(double), &dx, NULL);

```

```

86
87
88 // Set up the memory on the GPU where the answer will be stored
89 cl_mem dev_du = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
90                                N*sizeof(double), NULL, NULL);
91
92 size_t maxwgs;
93 clGetDeviceInfo(device_id, CL_DEVICE_MAX_WORK_GROUP_SIZE,
94                  sizeof(size_t), &maxwgs, NULL);
95
96 // Define all the arguments for the kernel function
97 err = clSetKernelArg(kernel, 0, sizeof(cl_mem), &dev_u);
98 err = clSetKernelArg(kernel, 1, sizeof(cl_mem), &dev_N);
99 err = clSetKernelArg(kernel, 2, sizeof(cl_mem), &dev_dx);
100 err = clSetKernelArg(kernel, 3, sizeof(cl_mem), &dev_du);
101
102 // Run the kernel
103 size_t N1 = N;
104 err = clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &N1, &maxwgs, 0,
105                               NULL, NULL);
106 clFinish(queue);
107
108 // Retrieve the results
109 err = clEnqueueReadBuffer(queue, dev_du, 1, 0, N*sizeof(double), du,
110                           0, NULL, NULL);
111
112 // Free the memory allocated on the GPU
113 clReleaseMemObject(dev_du);
114 clReleaseMemObject(dev_u);
115 clReleaseKernel(kernel);
116 clReleaseProgram(program);
117 clReleaseCommandQueue(queue);
118 clReleaseContext(context);
119
120 // Free the memory on the CPU
121 free(u);
122 free(du);
123 free(srccode);
124
125 return 0;
126 }

```

In Example 30.3, the variable `u` is defined as `constant` on Line 16 of `diff.cl`. The full word `constant` is used as compared to the C language `const` attribute because they mean different things. The latter tells the compiler that the value is fixed for compilation purposes, while the former tells the compiler to store the variable in higher speed read-only constant memory space on the GPU. Since the values of both `N` and `dx` will be common among all the work-groups, it makes sense to make them constant as well.

Constant memory space is limited and the data size is determined at run time, so it is wise to double check that it won't overrun the constant memory buffer. Thus, on Line 47, the size of the constant memory buffer for the current device is requested and then checked to see if it is within the limit of available constant memory.

The different techniques for managing memory and parallelism can give dramatically different results, so paying careful attention to these details is well worth the effort. To compare, the time to complete the task of computing the one-dimensional central finite difference operator on a double precision grid of varying lengths is shown in Figure 31.1.

## Exercises

- 30.1. Change Example 29.4 so that it uses local memory and verify the solution is correct.
- 30.2. Try running the kernel `diff` from Example 30.1, which does not have the `barrier(CLK_LOCAL_MEM_FENCE)`, and check the results. Insert the `barrier` and verify that it solves the problem.
- 30.3. Write a kernel to compute the sum of an array of double-precision numbers. See the discussion in Section 33.1.1 for how the computation should be organized. Write a corresponding loop without using the GPU; verify they get the same answer and compare the time required to compute the solution.
- 30.4. Write a kernel to compute the maximum absolute value of an array of double-precision numbers. The organization for the kernel is similar to compute the sum of a list of numbers replacing addition with a comparison. See the discussion in Section 33.1.1 for how the computation should be organized. Write a corresponding loop without using the GPU; verify they get the same answer and compare the time required to compute the solution.



# Chapter 31

# Command Queues

The examples so far have been limited to using a single command queue. The queue receives multiple kernels and they are sequentially executed within that queue. A single GPU device can handle multiple command queues; the maximum number can be obtained using `clGetDeviceInfo` with the `CL_DEVICE_MAX_ON_DEVICE_QUEUES` query. When a device has multiple command queues, the scheduler decides which command queue to use for the next task based on load balance.

Some GPU devices are able to overlap compute kernels with memory transfers so that data transfers using `clEnqueueWriteBuffer` or `clEnqueueReadBuffer` can be done simultaneously with compute kernels. In order for that to work the command queue must have an additional option inserted that will allow kernels to be executed out of order. With that option, the dependency of kernels upon each other must be managed manually, and this will require learning about events, which are covered in Section 31.1. Once events are mastered, it becomes possible to use events to measure the performance of kernels, which is discussed in Section 31.2, and also to implement concurrency of data transfers with compute kernels, which is in Section 31.3.

OpenCL is also able to utilize multiple heterogeneous devices so that work can be done simultaneously on the CPU and multiple GPUs and other devices. To accomplish this, each device will require its own queue. Section 31.4 shows how to employ multiple command queues to take advantage of multiple GPUs and multiple CPUs in order to complete a computational task.

## 31.1 • Events

Functions with names that begin with “`clEnqueue`” are functions that submit tasks to a command queue. In this text there are three that have come up: the host/device memory transfer functions `clEnqueueReadBuffer` and `clEnqueueWriteBuffer`, and the kernel submission function `clEnqueueNDRangeKernel`. By default, the tasks are executed in the order in which they are received by the command queue. Each of these functions can also handle events, which can be used to exert finer control on the order in which tasks are executed on the command queue, and to report execution times for the tasks to which they are assigned, among other things.

Recall the arguments to the function `clEnqueueWriteBuffer` discussed earlier, paying closer attention to the last three arguments:

```
cl_int clEnqueueWriteBuffer( cl_command_queue queue,
                            cl_mem dev_dest,
                            cl_bool is_blocking,
                            size_t offset,
                            size_t size,
                            void* src,
                            cl_uint wait_list_num,
                            cl_event* wait_list,
                            cl_event* event      );
```

The last argument `event` of type `cl_event*` is how an event is attached to a task. In this case, the task is to copy host memory into a memory buffer on the device.

Suppose a subsequent task is to read the buffer back into host memory. Obviously, this task should not begin until the previous task is complete. There are multiple avenues to enforce that. The first avenue, which has already been discussed, is to use the function `clFinish` to force the command queue to complete all tasks before the `clEnqueueReadBuffer` task is submitted to the command queue. This is the crudest form of control as it prevents any kind of concurrency from happening. Another option is to set the `is_blocking` value to `CL_TRUE` in `clEnqueueWriteBuffer` so that the queue is blocked while the write task is executed.

An alternative way to accomplish the same task using events is to use the function

```
cl_int clWaitForEvents( cl_uint wait_list_num,
                       cl_event* wait_list   );
```

to force the host program to wait until the specified list of events is completed. Thus, to copy data from the host to the device and then wait for the task to be completed, the code would be

```
cl_event write_event;
err = clEnqueueWriteBuffer(queue, dev_u, 0, 0, N*sizeof(double), u, 0,
                           NULL, &write_event);
err = clWaitForEvents(1, &write_event);
err = clEnqueueReadBuffer(queue, dev_u, 0, 0, N*sizeof(double), v, 0,
                           NULL, NULL);
```

The `cl_event` called `write_event` is tied to the task when it is passed by reference in the last argument of `clEnqueueWriteBuffer`. The program then blocks at the function `clWaitForEvents` until the `write_event` is completed.

Even finer control can be utilized by using the wait event arguments. Note the second to last two arguments of `clEnqueueReadBuffer` where it accepts a list of events for which this task should wait before executing. The `write_event` can be placed there, bypassing the need for `clWaitForEvents`.

```
cl_event write_event;
err = clEnqueueWriteBuffer(queue, dev_u, 0, 0, N*sizeof(double), u, 0,
                           NULL, &write_event);
// wait for 1 event: write_event
err = clEnqueueReadBuffer(queue, dev_u, 0, 0, N*sizeof(double), v, 1,
                           &write_event, NULL);
```

A task can be forced to wait for multiple events, e.g., suppose there are multiple buffers to be written before a compute kernel is executed:

```

cl_event write_event [2];
err = clEnqueueWriteBuffer (queue , dev_u , 0 , 0 , N*sizeof(double) , u , 0 ,
                           NULL , &write_event [0]);
err = clEnqueueWriteBuffer (queue , dev_v , 0 , 0 , N*sizeof(double) , v , 0 ,
                           NULL , &write_event [1]);
// wait for 2 events
err = clEnqueueNDRangeKernel(queue , kernel , 1 , NULL , &N , &wgs , 2 ,
                             write_event , NULL);

```

This time `write_event` is an array of two events, which are both passed to the function `clEnqueueNDRangeKernel`.

Under default conditions, these commands will still execute in sequential order. In order to get them to possibly overlap, the command queue needs a slight modification as well. When the command queue is created, the `cl_queue_properties` argument needs to be used. To this point, it was assumed that the default command queue was fine, and the third argument of the `clCreateCommandQueueWithProperties` function was specified to be zero. In fact, that argument is a zero-terminated list of properties to be set. Thus, to allow tasks to run out of sequence, the `out-of-order` property must be enabled by putting it into a zero-terminated array of `cl_queue_properties`:

```

cl_queue_properties qprop [] = {CL_QUEUE_PROPERTIES,
                               CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE , 0};
cl_command_queue queue = clCreateCommandQueueWithProperties (context ,
                                                             device_id , qprop , &err);

```

With this property turned on, events are needed to play the role of traffic cops to ensure that tasks are not executed until the tasks upon which they depend are completed. Example 31.1 is a modification of Example 30.1 where the `out-of-order` property is turned on and events are used to control the sequence of tasks.

### Example 31.1.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4 #include <CL/opencl.h>
5 #include "getkernel.h"
6 #include "cl_handle_error.h"
7
8 #ifndef M_PI
9 #define M_PI 3.1415926535897932384626433832795
10#endif
11
12/*
13   Demonstrate a simple example for implementing a
14   parallel finite difference operator
15
16   Inputs: argc should be 2
17   argv[1]: Length of the domain
18
19   Outputs: the initial data and its derivative.
20 */
21
22int main(int argc , const char * argv []) {
23

```

```

24 // Read the input values
25 int N = atoi(argv[1]); // Get the length of the vector from input
26 int err;
27
28 // Request the platform info
29 cl_platform_id platform;
30 err = clGetPlatformIDs(1, &platform, NULL);
31
32 // Request a GPU device
33 cl_device_id device_id;
34 err = clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device_id,
35 NULL);
36
37 // Set up the context for the GPU
38 cl_context context = clCreateContext(0, 1, &device_id, NULL, NULL,
39 &err);
40
41 // Enable the out-of-order property of the command queue
42 cl_queue_properties qprop[] = {CL_QUEUE_PROPERTIES,
43 CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE, 0};
44 cl_command_queue queue = clCreateCommandQueueWithProperties(context,
45 device_id, qprop, &err);
46
47 // In OpenCL, the kernel is compiled at runtime,
48 // try to do this only once and then save it.
49 char* srccode = GetKernelSource("diff.cl");
50 cl_program program = clCreateProgramWithSource(context, 1,
51 (const char**)&srccode, NULL, &err);
52
53 // Compile the kernel code
54 err = clBuildProgram(program, 0, NULL, NULL, NULL);
55
56 // Create the kernel function from the compiled OpenCL code
57 cl_kernel kernel = clCreateKernel(program, "diff", &err);
58
59 // Allocate memory for the input arguments, data will be copied from
60 // the inputs to the GPU at the same time.
61 double dx = 2*M_PI/N;
62 double* u;
63 u = (double*)malloc(N*sizeof(double));
64 int i;
65 for (i=0; i<N; ++i)
66     u[i] = sin(i*dx);
67
68 double* du = (double*)malloc(N*sizeof(double));
69
70 cl_mem dev_u = clCreateBuffer(context, CL_MEM_READ_ONLY,
71 N*sizeof(double), NULL, &err);
72
73 // Copy the input data into device memory
74 cl_event write_event;
75 err = clEnqueueWriteBuffer(queue, dev_u, CL_FALSE, 0,
76 N*sizeof(double), u, 0, NULL, &write_event);
77
78 // Set up the memory on the GPU where the answer will be stored
79 cl_mem dev_du = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
80 N*sizeof(double), NULL, &err);
81
82 size_t maxwgs;
83 err = clGetDeviceInfo(device_id, CL_DEVICE_MAX_WORK_GROUP_SIZE,
84 sizeof(size_t), &maxwgs, NULL);
85 if (N < maxwgs)
86     maxwgs = N;

```

```
87 // Define all the arguments for the kernel function
88 err = clSetKernelArg(kernel, 0, sizeof(cl_mem), &dev_u);
89 err = clSetKernelArg(kernel, 1, sizeof(int), &N);
90 err = clSetKernelArg(kernel, 2, sizeof(double), &dx);
91 err = clSetKernelArg(kernel, 3, sizeof(cl_mem), &dev_du);
92 err = clSetKernelArg(kernel, 4, (maxwgs+2)*sizeof(double), NULL);
93
94 size_t N1 = N;
95 cl_event run_event;
96 err = clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &N1, &maxwgs, 1,
97                               &write_event, &run_event);
98
99
100 cl_event read_event;
101 err = clEnqueueReadBuffer(queue, dev_du, CL_FALSE, 0,
102                           N*sizeof(double), du, 1, &run_event, &read_event);
103
104 err = clWaitForEvents(1, &read_event);
105
106 // Free the memory allocated on the GPU
107 err = clReleaseMemObject(dev_du);
108 err = clReleaseMemObject(dev_u);
109 err = clReleaseKernel(kernel);
110 err = clReleaseProgram(program);
111 err = clReleaseCommandQueue(queue);
112 err = clReleaseContext(context);
113
114 // Free the memory on the CPU
115 free(u);
116 free(du);
117 free(srccode);
118
119 return 0;
120 }
```

To begin, the command queue is created with the out-of-order property set on Line 44. While normally it makes more sense to use the CL\_MEM\_COPY\_HOST\_PTR option in `clCreateBuffer`, it is not used here so that an additional example of event usage is possible. Thus, a write event is created when the input data is copied to the device on Line 75. Note that the `is_blocking` argument is set to `CL_FALSE` so that control returns immediately to the host program.

The next task is to run the kernel on Line 97. In this case, it is told to wait for `write_event`, and `run_event` is tied to the kernel. The subsequent task to read the output on Line 101 is similarly set to nonblocking yet will wait for the `run_event` to be completed and the `read_event` is tied to the read task. Finally, before the data read from the device can be used, the function `clWaitForEvents` forces the program to wait for the `read_event` to be completed.

## 31.2 • Measuring Performance

Events themselves can carry information such as the type of command and the run status, but the most useful information is as a measure of the time required for a kernel to run. To get this information, the command queue requires another optional property `CL_QUEUE_PROFILING_ENABLE` to be turned on. This alerts the command queue that profiling information for events should be kept. Note that if both the profiling and the

out-of-order properties are desired, then the queue properties are combined using the bitwise “|” operator:

```
1 cl_queue_properties qprop [] = {CL_QUEUE_PROPERTIES,
2                                CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE
3                                | CL_QUEUE_PROFILING_ENABLE, 0};
```

After a task submitted to the command queue has been completed, the profiling information for the event can be requested using the function

```
cl_int clGetEventProfilingInfo( cl_event event,
                               cl_profiling_info request,
                               size_t info_size,
                               void* info,
                               size_t* ret_size );
```

The `request` argument is the information requested about the event; the options are listed in Table 31.1. All options return the type `cl_ulong`, so the `info_size` is always `sizeof(cl_ulong)`. Since all the return types are the same size, the last argument isn't really needed, so use `NULL` for the last argument. Thus, to get the start and end times for an event, the request would be

```
1 cl_ulong start_time , end_time ;
2 clGetEventProfilingInfo( event , CL_PROFILING_COMMAND_START,
3                         sizeof(cl_ulong) , &start_time , NULL );
4 clGetEventProfilingInfo( event , CL_PROFILING_COMMAND_END ,
5                         sizeof(cl_ulong) , &end_time , NULL );
```

**Table 31.1.** *Some of the available options for requesting profiling information about an event. All options return the specified time point in nanoseconds.*

Information Request	Meaning
<code>CL_PROFILING_COMMAND_QUEUED</code>	Time the event was put in the command queue on the host.
<code>CL_PROFILING_COMMAND_SUBMIT</code>	Time the host submitted the event to the device.
<code>CL_PROFILING_COMMAND_START</code>	Time the event started execution.
<code>CL_PROFILING_COMMAND_END</code>	Time the event finished execution.

For a careful measurement of the elapsed time of an event, it's best to first clear the command queue using `clFinish` before submitting the task to the command queue. Also, be sure to wait for the event to be finished using `clFinish` or `clWaitForEvents` before getting the profiling info or the data may be unreliable. Thus, to carefully measure the time cost for a kernel called `diff`, use

### Example 31.2.

```
1 cl_queue_properties qprop [] = {CL_QUEUE_PROPERTIES,
2                                CL_QUEUE_PROFILING_ENABLE, 0};
3 cl_command_queue queue
4     = clCreateCommandQueueWithProperties( context , device_id ,
5                                         qprop , &err );
6
7 // Other code for setting up the kernel such as preparing buffers ,
8 // compiling the kernel , etc .
```

```

9   // Clear the command queue before submitting the kernel
10  err = clFinish(queue);
11  cl_event event;
12  err = clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &N, NULL, 0,
13                               NULL, &event);
14
15
16 // Wait for the kernel to finish before taking measurements
17 err = clWaitForEvents(1, &event);
18
19 // Retrieve timing information
20 cl_ulong start_time, end_time;
21 err = clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_START,
22                               sizeof(start_time), &start_time, NULL);
23 err = clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_END,
24                               sizeof(end_time), &end_time, NULL);
25 // elapsed time in nanoseconds
26 cl_ulong elapsedtime = end_time - start_time;

```

There have been several methods described for implementing the finite difference approximation through a device kernel function. The methods can now be compared to see which is faster. Figure 31.1 shows the following methods discussed thus far applied to computing the central difference operator on a vector of length  $N$ :

1. Using a CPU with no OpenCL, just simply executing a loop in C.
2. Using a GPU with one work-item per work-group.
3. Using a GPU with the maximum number of work-items per work-group.
4. Using a GPU with the maximum number of work-items per work-group and using local memory.
5. Using two GPUs with local memory.<sup>11</sup>

For all these cases, timing measurements were made with the number of nodes being various powers of 2. Figure 31.2 shows the same set of tests, but this time for a two-dimensional grid and computing the Laplacian of the function.

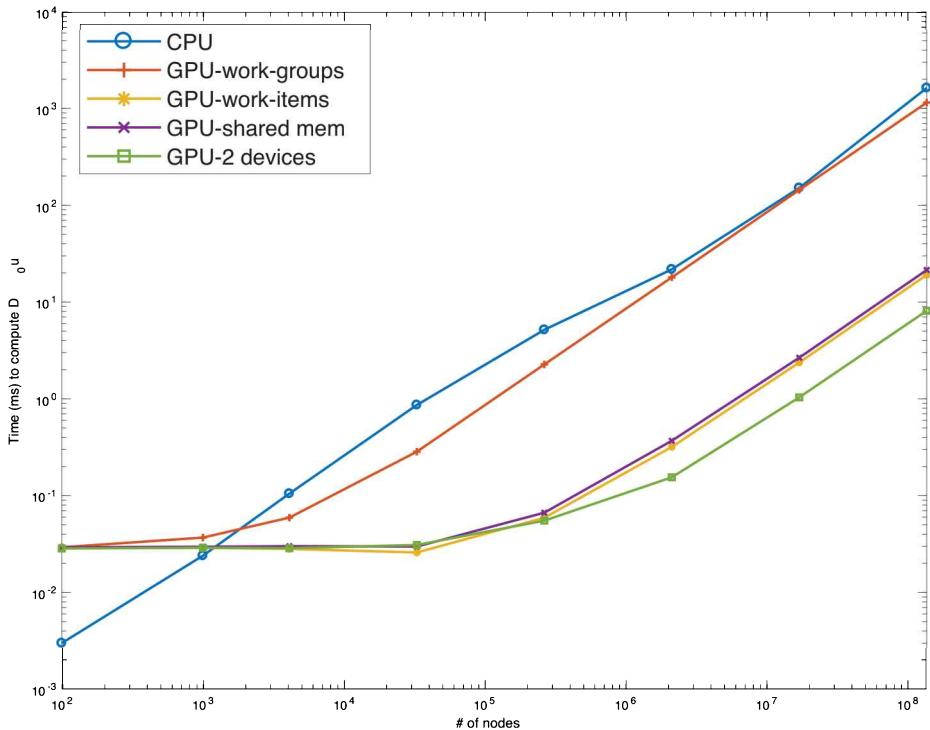
Note that the use of shared memory, constant memory, or global memory does not seem to make much of a difference in the speed of the kernel. This is likely to be due to the fact that the compiler will attempt to take advantage of cache memory as much as possible, and for this simple kernel, the memory required for this is likely cached automatically. For more complicated kernels, where more local storage is required to complete the task, more careful attention to which variables are stored globally and which are stored locally will be important.

### 31.3 • Concurrency

Devices can usually take more than one queue. The purpose of using more than one queue is to make it possible for tasks to be done concurrently. In particular, many devices are capable of managing host/device memory transfers simultaneously while doing compute kernels, thus speeding up the resulting calculation. To illustrate how this is done, consider Example 31.3, where a device is set up to handle multiple queues according to user input.

---

<sup>11</sup>This method will be discussed in Section 31.4.



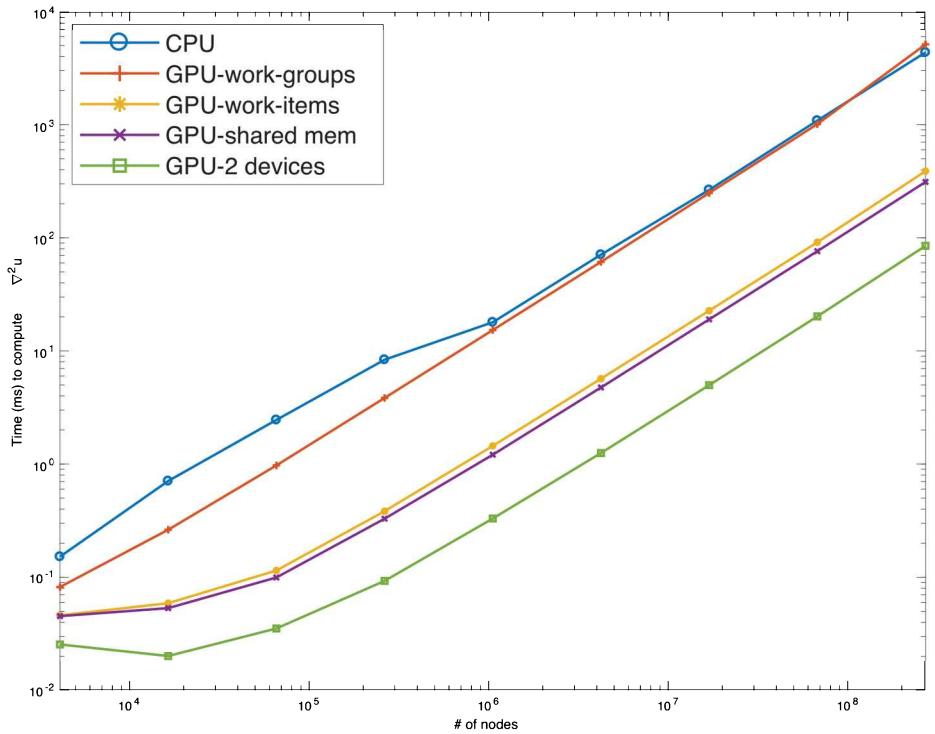
**Figure 31.1.** Scaling properties of one-dimensional finite central difference using various versions of parallelism using OpenCL compared to a serial CPU implementation. The horizontal axis shows the number of nodes in the grid, and the vertical axis shows the time to complete the finite difference kernel function in milliseconds. The legend refers to (CPU) Serial CPU code, no OpenCL code; (GPU-work-groups) Results from using Example 29.1; (GPU-work-items) Results from explicitly using the maximum number of work-items per work-group as in Example 29.2; (GPU-shared mem) results from using local memory as in Example 30.2; (GPU-2 devices) Results from choosing two GPU devices for one calculation using the modified version of Example 31.3 with the modifications described in Section 31.4.

### Example 31.3.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4 #include <CL/opencl.h>
5 #include "getkernel.h"
6
7 #ifndef M_PI
8 #define M_PI 3.1415926535897932384626433832795
9 #endif
10
11 /*
12 Demonstrate a simple example for implementing a
13 parallel finite difference operator
14
15 Inputs: argc should be 2
16 argv[1]: Length of the domain
17 argv[2]: Number of queues to use
18

```



**Figure 31.2.** Scaling properties of a two-dimensional finite central difference using various versions of parallelism using OpenCL compared to a serial CPU implementation. The horizontal axis shows the total number of nodes in the grid, and the vertical axis shows the time to complete the finite difference kernel function in milliseconds. The legend refers to (CPU) Serial CPU code, no OpenCL code; (GPU-work-groups) Results from using work-group size {1, 1}; (GPU-work-items) Results from explicitly using the maximum work-group size  $32 \times 32$ ; (GPU-shared mem) results from also loading global data into local memory; (GPU-2 devices) Results from choosing two GPU devices for one calculation using local memory.

```

19     Outputs: the initial data and its derivative .
20 */
21
22 int main(int argc , const char * argv []) {
23
24     // Read the input values
25     int N = atoi(argv[1]); // Get the length of the vector from input
26     int nqueues = atoi(argv[2]); // Get the number of queues
27     int err , q;
28
29     // Request the platform info
30     cl_platform_id platform;
31     err = clGetPlatformIDs(1, &platform , NULL);
32
33     // Request a GPU device
34     cl_device_id device_id ;
35     err = clGetDeviceIDs(platform , CL_DEVICE_TYPE_GPU, 1, &device_id ,
36                                     NULL);
37
38     // Set up the context for the GPU
39     cl_context context = clCreateContext(0, 1, &device_id , NULL, NULL,

```

```

40                                         &err);
41
42 // Create dispatch queues that allow out of order and profiling
43 cl_queue_properties qprop [] = {CL_QUEUE_PROPERTIES,
44                                     CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE
45                                     | CL_QUEUE_PROFILING_ENABLE, 0};
46 cl_command_queue queue[nqueues];
47 for (q=0; q<nqueues; ++q)
48     queue[q] = clCreateCommandQueueWithProperties (context , device_id ,
49                                                 qprop , &err);
50
51 // Load the kernel file
52 char* srccode = GetKernelSource("diff2.cl");
53 cl_program program = clCreateProgramWithSource(context , 1,
54                                                 (const char**)&srccode , NULL, &err);
55
56 // Compile the kernel code
57 err = clBuildProgram(program , 0, NULL, NULL, NULL);
58
59 // Create kernel functions from the compiled OpenCL code
60 cl_kernel kernel[nqueues];
61 for (q=0; q<nqueues; ++q)
62     kernel[q] = clCreateKernel(program , "diff" , &err);
63
64 // Allocate memory for the input arguments, data will be copied from
65 // the inputs to the GPU at the same time.
66 double dx = 2*M_PI/N;
67 double* u;
68 u = (double*)malloc ((N+2)*sizeof(double));
69 int i;
70 for (i=1; i<=N; ++i)
71     u[i] = sin(i*dx);
72 u[0] = u[N];
73 u[N+1] = u[1];
74
75 double* du = (double*)malloc (N*sizeof(double));
76 // Set up the memory on the GPU where the answer will be stored
77 cl_mem dev_du[nqueues];
78 for (q=0; q<nqueues; ++q)
79     dev_du[q] = clCreateBuffer(context , CL_MEM_WRITE_ONLY,
80                               N/nqueues*sizeof(double) , NULL, &err);
81
82 cl_mem dev_N = clCreateBuffer(context , CL_MEM_READ_ONLY
83                               | CL_MEM_COPY_HOST_PTR, sizeof(int) , &N, &err);
84
85 cl_mem dev_dx = clCreateBuffer(context , CL_MEM_READ_ONLY
86                               | CL_MEM_COPY_HOST_PTR, sizeof(double) , &dx , &err);
87
88 // Create an input buffer for each queue
89 cl_mem dev_u[nqueues];
90 for (q=0; q<nqueues; ++q) {
91     dev_u[q] = clCreateBuffer(context , CL_MEM_READ_ONLY,
92                               (N/nqueues+2)*sizeof(double) , NULL, &err);
93 }
94 size_t maxwgs;
95 err = clGetDeviceInfo(device_id , CL_DEVICE_MAX_WORK_GROUP_SIZE,
96                      sizeof(size_t) , &maxwgs , NULL);
97 if (N/nqueues < maxwgs)
98     maxwgs = N/nqueues;
99 size_t N1 = N/nqueues;
100
101 // Create multiple events for run, write, and read
102 cl_event run_event[nqueues];

```

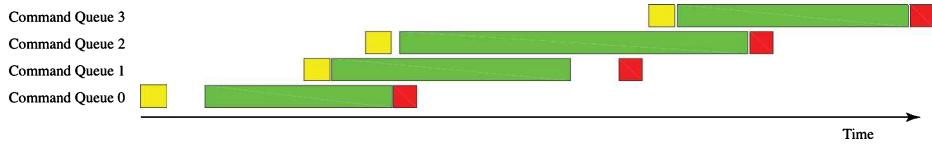
```

103    cl_event write_event[nqueues];
104    cl_event read_event[nqueues];
105
106    // Set the arguments for each of the kernels
107    for (q=0; q<nqueues; ++q) {
108        // Define all the arguments for the kernel function
109        err = clSetKernelArg(kernel[q], 0, sizeof(cl_mem), &dev_u[q]);
110        err = clSetKernelArg(kernel[q], 1, sizeof(cl_mem), &dev_N);
111        err = clSetKernelArg(kernel[q], 2, sizeof(cl_mem), &dev_dx);
112        err = clSetKernelArg(kernel[q], 3, sizeof(cl_mem), &dev_du[q]);
113        err = clSetKernelArg(kernel[q], 4, (maxwgs+2)*sizeof(double), NULL);
114    }
115
116    // Load the data, execute the kernel, retrieve the results
117    for (q=0; q<nqueues; ++q) {
118        err = clEnqueueWriteBuffer(queue[q], dev_u[q], 0, 0,
119                                   (N/nqueues+2)*sizeof(double), &u[q*N/nqueues],
120                                   0, NULL, &write_event[q]);
121        err = clEnqueueNDRangeKernel(queue[q], kernel[q], 1, NULL, &N1,
122                                      &maxwgs, 1, &write_event[q], &run_event[q]);
123        err = clEnqueueReadBuffer(queue[q], dev_du[q], 0, 0,
124                                   N/nqueues*sizeof(double), du,
125                                   1, &run_event[q], &read_event[q]);
126    }
127
128    // Wait for all the read events to finish
129    err = clWaitForEvents(nqueues, read_event);
130
131    // Free the memory allocated on the GPU
132    for (q=0; q<nqueues; ++q) {
133        err = clReleaseMemObject(dev_du[q]);
134        err = clReleaseMemObject(dev_u[q]);
135        err = clReleaseCommandQueue(queue[q]);
136        err = clReleaseKernel(kernel[q]);
137    }
138    err = clReleaseProgram(program);
139    err = clReleaseContext(context);
140
141    // Free the memory on the CPU
142    free(u);
143    free(du);
144    free(srccode);
145
146    return 0;
147}

```

This version with multiple queues is not much different from Example 31.1 except that now there are multiple queues, kernels, and buffers. The work load has been subdivided so that each kernel will compute a fraction of the full data depending on the number of queues requested by the user. For robust code, the program should check that the number of queues requested by the user are not more than the device can handle. To check the maximum number of queues a given device can take, use the `CL_DEVICE_MAX_ON_DEVICE_QUEUES` option in the `clGetDeviceInfo` function.

Note how the write, run, and read events are all set up so that the kernel in command queue `q` waits for the input data for command queue `q` to be transferred, and similarly the read events wait for their corresponding run events to be completed. It only requires one call to `clWaitForEvents` on Line 129 because it can wait for an array of events and will block until all the events in the array are completed.



**Figure 31.3.** Example of using overlapping of host/device data transfers and compute kernels. This data was generated with a modified version of the one-dimensional finite difference kernel computed 40 times per kernel so that the compute time is large enough to be visible in the plot. Each row corresponds to a separate queue linked to the same GPU device. The width of the box shows the time elapsed from event start to end. Yellow represents the `c1EnqueueWriteBuffer` events, green represents the `c1EnqueueNDRangeKernel` events, and red represents the `c1EnqueueReadBuffer` events.

To illustrate the results of this modification, it's best to view it graphically where the time intervals when each of the tasks took place are placed side by side as in Figure 31.3. In the figure, the host-to-device data transfers are in yellow, the kernel executions are in green, and the device-to-host data transfers are in red. The first thing to notice is that for a given command queue, the events do not overlap, which is good because waiting for events is meant to prevent such overlaps. Next, notice that while command queue 0 is executing its kernel, command queue 1 followed by command queue 2 are able to start transferring data to the device simultaneously. The device used for this experiment has multiple multiprocessors, so it is also able to handle multiple compute kernels simultaneously as well. By overlapping the data transfers with the computation, the total overall cost of the calculation is reduced.

## 31.4 • Using Multiple Devices

If a platform contains multiple devices, which can include both GPUs and CPUs, then they can all be put to use for a calculation. Example 31.3 is easily modified to accommodate multiple devices. The first change is to convert the variable `device_id` into an array and to retrieve the array of devices from the platform. The context must also be told that multiple devices will be used. Thus, Lines 33–40 become

```

33 // Request GPU device list
34 cl_device_id device_id[nqueues];
35 err = clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, nqueues, device_id,
36 NULL);
37
38 // Set up the context for the GPU
39 cl_context context = clCreateContext(0, nqueues, device_id, NULL,
40 NULL, &err);

```

where the number of requested devices is specified by `nqueues`. As discussed in Section 28.2, a robust code would check that the number of devices available is not exceeded by this value by checking how many devices are actually available. Also, to include CPU devices, the second argument would be changed to `CL_DEVICE_TYPE_ALL` or the combination `CL_DEVICE_TYPE_GPU | CL_DEVICE_TYPE_CPU`.

The other change is to assign the multiple command queues to different devices, one for each in this case. This requires a minor change to Lines 47–49:

```

47 for (q=0; q<nqueues; ++q)
48     queue[q] = clCreateCommandQueueWithProperties(context, device_id,
49                                         qprop, &err);

```

These are all the changes necessary to utilize multiple devices.

To see that this strategy actually works, check Figure 31.2, where using two identical devices results in a speed-up by a factor of about two. If mixing heterogeneous devices for a computing task, then load balancing becomes an issue to ensure optimal use of the available devices. For example, if a slow device is keeping fast devices idle due to a dependency in the algorithm, then the results may be worse than if the slow device were excluded entirely. There are also overhead costs for including multiple devices. However, by using the profiling information provided by events described in Section 31.2, load balancing can be achieved to obtain optimal results.

---

## Exercises

- 31.1. Modify Example 29.4 to subdivide the domain in such a way that a maximal work-group size is used to cover the given dimensions  $M$ ,  $N$  and so that it uses local work-group memory. Use event timing to measure the time required to execute the kernel for the original version and the modified version.
- 31.2. Write a program that uses multiple command queues to load a three-dimensional array from host memory onto the device, compute the Laplacian on the array, and then transfer the results back to host memory. Use event timing to estimate how much time is consumed doing memory transfers versus computing the finite difference.
- 31.3. If you have two different devices in your platform, modify Example 31.3 to utilize the two devices. Do load balancing between the two devices by measuring the time required for each device to compute the solution for different values of  $N$  and then divide the array accordingly so each spends approximately the same amount of time to complete the task assigned.



## Chapter 32

# OpenCL Libraries

The OpenCL package itself does not come with any ready-made libraries, but there are some available. Some of the libraries are less developed than their CUDA relatives, but they do provide the basic functionality that is needed to complete the projects contained in Part VI. The analogue of LAPACK for the OpenCL platform is clMAGMA, but it is available only for the AMD graphics card platform. The device maker AMD also put their OpenCL development tools into open source and one of the products of that effort is the clFFT library that can be used for doing Fourier transforms. Finally, there is no OpenCL specific random number generation library, but the Random123 library can be effectively used in the OpenCL framework to enable individual work-items to generate random numbers in kernel functions.

### 32.1 • Random123

The OpenCL package does not contain a native random number generator, and the generators described in Chapter 7 are not available within a kernel. What is needed is a lightweight kernel that can be used in parallel on a GPU. Fortunately, the Random123 library

<http://www.thesalmons.org/john/random123/releases/latest/docs/index.html>

fits the bill by providing a host of random number generators that can be used for this purpose. In Example 32.1, a list of double precision uniformly distributed random numbers is generated in the range  $[0, 1]$  using the ThreeFry4x32 algorithm provided in the library. The generators in this library produce random integers, so they must be converted to double precision.

As has been discussed in other parallel implementations of random number generators in this text, care must be taken to make sure that when random numbers are generated in parallel, the produced sequence of numbers being generated by each processor is not repeated. The Random123 library handles this well by taking advantage of the fact that the seed can be modified by the global index to make it unique for each process.

**Example 32.1.****File: get\_rand.cl**

```

1 // Include the source from the Random123 library
2 #include <Random123/threefry.h>
3
4 #pragma OPENCL EXTENSION cl_khr_fp64: enable
5
6 /*
7  kernel get_rand
8   Generate random numbers in range [0,1]
9
10 Inputs:
11   uint n: size of array to be generated
12
13 Outputs:
14   double* vals: array of random values
15 */
16 kernel void get_rand(uint N, global double *vals) {
17
18     size_t id = get_global_id(0);
19
20     // Set up and seed the RNG incorporating the process ID
21     threefry4x32_key_t k = {{id, 0xdecafbad, 0xfacebead, 0x12345678}};
22     threefry4x32_ctr_t c = {{0, 0xf00dcafe, 0xdeadbeef, 0xbeeff00d}};
23
24     if (id < N) {
25         // The threefry algorithm produces 32bit ints
26         union {
27             threefry4x32_ctr_t c;
28             int4 i;
29         } uval;
30         uval.c = threefry4x32(c, k);
31
32         // Convert two 32bit ints into a single 64bit long int
33         long int j = ((long)uval.i.x << 32) | ((long)uval.i.y & 0xFFFFFFFFL);
34
35         // Calculate the scaling factor to get into the [0,1) range
36         double factor = 1. / (LONG_MAX + 1.);
37         vals[id] = 0.5 + j * factor / 2.;
38     }
39 }
```

**File: main.c**

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <CL/opencl.h>
4 #include "getkernel.h"
5
6 /*
7  Demonstrate a simple example for implementing a
8  parallel finite difference operator
9
10 Inputs: argc should be 2
11      argv[1]: Length of the domain
12
13 Outputs: the initial data and its derivative.
```

```
14 */
15
16 int main(int argc, char* argv[])
17 {
18     // Get the length of the vector from input
19     size_t N = atoi(argv[1]);
20
21     // Request a platform
22     cl_platform_id platform;
23     int err = clGetPlatformIDs(1, &platform, NULL);
24
25     // Get the device ID
26     cl_device_id device_id;
27     err = clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device_id,
28                         NULL);
29
30     // Set up the context for the GPU
31     cl_context context = clCreateContext(0, 1, &device_id, NULL, NULL,
32                                         &err);
33
34     // Create a command queue
35     cl_command_queue queue;
36     queue = clCreateCommandQueueWithProperties(context, device_id, NULL,
37                                                &err);
37
38
39     // Load the kernel source code
40     char* srccode = GetKernelSource("get_rand.cl");
41     cl_program program = clCreateProgramWithSource(context, 1,
42                                                    (const char**)&srccode, NULL, &err);
43
44     // Compile the kernel code
45     err = clBuildProgram(program, 0, NULL, "-I/usr/local/include", NULL,
46                           NULL);
47
48     // Create the kernel function from the compiled OpenCL code
49     cl_kernel kernel = clCreateKernel(program, "get_rand", &err);
50
51     // Allocate memory to store the results
52     double *u = (double*)malloc(N*sizeof(double));
53
54     // Allocate memory on the device
55     cl_mem dev_u = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
56                                    N*sizeof(double), NULL, NULL);
57
58     // Define the arguments for the kernel function
59     err = clSetKernelArg(kernel, 0, sizeof(cl_uint), &N);
60     err = clSetKernelArg(kernel, 1, sizeof(cl_mem), &dev_u);
61
62     // Execute the kernel
63     err = clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &N, NULL, 0,
64                                  NULL, NULL);
65
66     // Wait for the queue to finish
67     clFinish(queue);
68
69     // Retrieve the results
70     err = clEnqueueReadBuffer(queue, dev_u, CL_TRUE, 0, N*sizeof(double),
71                               u, 0, NULL, NULL);
72
73     // Print the results
74     int i;
75     for (i=0; i<N; ++i)
76         printf("%f\n", u[i]);
```

```

77
78 // Free the resources
79 clReleaseMemObject(dev_u);
80 clReleaseKernel(kernel);
81 clReleaseProgram(program);
82 clReleaseCommandQueue(queue);
83 clReleaseContext(context);
84 free(u);
85 free(srccode);
86
87 return 0;
88 }
```

The components of the main program here have all been encountered before, so the breakdown of this example focuses on the kernel function defined in the file `get_rand.cl`. The Random123 functions are all in header files, and there is no compiled code, which is why this will work when using OpenCL. When using the ThreeFry4x32 generator, the appropriate header file must be included as on Line 2. Note that the presence of this `#include` in the kernel function may mean that an include path must be specified when the kernel is built on Line 45. In this case, the Random123 directory of header files is located in the directory `/usr/local/include`, so that path is specified in the fourth argument of `clBuildProgram` as shown on Line 45. Other flags can also be included in that argument such as other include paths or defined constants. For example, to define the constant `M_PI` within the kernel from the main program, the fourth argument of `clBuildProgram` would become

```
"-I/usr/local/include -DM_PI=3.1415926535897932384626433832795"
```

Next, the ThreeFry4x32 generator requires both a key and a counter, which are set up on Lines 21 and 22. The first entry of the key is important to make sure that every work-item has a unique seed to prevent the same set of random values being generated by each work-item. Thus, the work-item ID is used here as the first argument of the key. There is nothing special about the other constant values in the key `k` and the counter `c`; any hexadecimal value up to eight digits can be used. Every subsequent call to `threefry4x32(c, k)`, as is done on Line 30, produces a new group of four random integers, which can be accessed by using the `int4` type defined on Line 28. Two of those random integers are combined to make a single long integer on Line 33. That value is then converted into the double precision range  $[0, 1]$  on Line 37.

In Example 32.1, each work-item generates one random value to be stored in an array. The method works because the initial value for the `threefry4x32_key` on Line 21 depends on the global work-item ID. However, if each work-item is meant to generate a list of random values, for example, each of  $M$  work-items will generate  $N$  values to fill out an  $M \times N$  array, then the counter variable `c` defined on Line 22 needs to be updated with an iteration-dependent value. Example 32.2 shows how the kernel would be modified to update the counter on Line 33. For a two-dimensional array, the second index `c.v[1]` could also be updated with a loop-dependent variable. If Line 33 were omitted, then each work-item would generate a unique random value but the same random value for all  $n$  entries in the array. While not shown here, to use this kernel, the main program would also have to be modified so that  $M$  is the global work size and both the device and host memory arrays are of dimension  $M \times N$ .

**Example 32.2.**

```

1 // Include the source from the Random123 library
2 #include <Random123/threefry.h>
3
4 #pragma OPENCL EXTENSION cl_khr_fp64: enable
5
6 /*
7  kernel get_rand
8   Generate random numbers in range [0,1]
9
10 Inputs:
11   uint N: size of second dimension of generated array
12
13 Outputs:
14   double* vals: array of random values
15 */
16 kernel void get_rand(uint N, global double *vals) {
17
18     size_t M = get_global_size(0);
19     size_t id = get_global_id(0);
20
21     // Set up and seed the RNG incorporating the process ID
22     threefry4x32_key_t k = {{id, 0xdecafbad, 0xfacebead, 0x12345678}};
23     threefry4x32_ctr_t c = {{0, 0xf00dcafe, 0xdeadbeef, 0xbeeff00d}};
24
25     if (id < M) {
26         // The threefry algorithm produces 32 bit ints
27         union {
28             threefry4x32_ctr_t c;
29             int4 i;
30             } uval;
31             int i;
32             for (i=0; i<N; ++i) {
33                 c.v[0] = i;
34                 uval.c = threefry4x32(c, k);
35
36                 // Convert two 32bit ints into a single 64bit long int
37                 long int j = ((long)uval.i.x << 32)
38                                         | ((long)uval.i.y & 0xFFFFFFFFL);
39
40                 // Calculate the scaling factor to get into the [0,1) range
41                 double factor = 1. / (LONG_MAX + 1. );
42                 vals[id*N + i] = 0.5 + j * factor / 2. ;
43             }
44         }
45 }
```

In terms of how to use random numbers in a parallel context, one may either generate a string of random values for use within a single work-item or have many work-items generate a fraction of the total samples required. Obviously, this particular example uses the latter approach. The former approach simply means there is a loop to generate more samples within the kernel function. For initializing some of the random grid data requested in some of the projects, another strategy would be to have each work-item generate one random value per grid point. In that case, for two dimensions, for example, the generator key and counter could be initialized using the global ID like this, where `seed` is a long integer passed to the generator:

```

1 size_t i = get_global_id(0);
2 size_t j = get_global_id(1);
3
4 // Set up and seed the RNG incorporating the process ID
5 threefry4x32_key_t k = {{seed, 0xdecafbad, 0xfacebead, 0x12345678}};
6 threefry4x32_ctr_t c = {{i, j, 0xdeadbeef, 0xbeeff00d}};

```

This way, the counter is again different for each call of the generator. Fortunately, the Random123 library is robust under all these conditions, which is important for high-fidelity computations.

## 32.2 • clMAGMA

clMAGMA is an OpenCL implementation of the MAGMA library. It is specifically designed for use on AMD graphics cards. The clMAGMA library documentation can be found at

<http://icl.cs.utk.edu/magma/software/view.html?id=207>

where descriptions of the functions are available. The package is meant to be a version of LAPACK for the OpenCL architecture, and many of the arguments and calling conventions used for those packages are also used for clMAGMA so that the transition is simple.

### 32.2.1 • Simple Example of a Linear Solve

As a beginning test case for the clMAGMA package, Example 32.3 solves a linear system of equations using the function `magma_dgesv`, which is the MAGMA equivalent of the LAPACK function `dgesv_`. Note that the matrix `A` is stored in column-major format in the same way as is done for LAPACK.

#### Example 32.3.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <math.h>
5 #include "magma.h"
6 #include "magma_lapack.h"
7
8 /*
9 int main(int argc, char* argv[])
10
11 A matrix A is created in column-major format
12 and a right-hand side is created. The system of equations
13 is then solved storing the solution the vector b upon return.
14
15 Inputs: none
16
17 Outputs: none
18 */
19 int main(int argc, char* argv[])
20 {
21     // Initialize the clMAGMA system
22     magma_init();
23 }

```

```

24 // Get the device info for setting up a queue
25 magma_int_t num;
26 magma_device_t devices[2];
27 magma_getdevices( devices , 2, &num );
28
29 // Set up the work queue for device 0
30 magma_queue_t queue;
31 magma_queue_create( devices[0], &queue );
32
33 // temporary data required by dgesv similar to LAPACK
34 magma_int_t *piv , info ;
35
36 // the matrix A is m x m
37 magma_int_t m = 9;
38
39 // the right-hand side has dimension m x n
40 magma_int_t n = 1;
41
42 // error code for MAGMA functions
43 magma_int_t err;
44
45 // The matrix A and right hand side b
46 double* A;
47 double* b;
48
49 // Allocate matrices on the host
50 err = magma_dmalloc_cpu(&A, m*m);
51 err = magma_dmalloc_cpu(&b, m);
52 // Note: should check that err==0, meaning call was successful
53
54 // Create temporary storage for the pivots.
55 piv = (magma_int_t*) malloc(m*sizeof(magma_int_t));
56
57 // Generate matrix A
58 double row[9] = {3., 21., 4., 1., 13., 1., 7., 13., 1.};
59 int i, j, index = 0;
60 for (j=0; j<9; ++j)
61   for (i=0; i<9; ++i)
62     A[index++] = (i > j ? -1 : 1) * row[j];
63
64 // Generate b
65 b[0] = 17;
66 for (i=0; i<4; ++i) {
67   b[i+1] = 11;
68   b[i+5] = -15;
69 }
70
71 // MAGMA solve
72 magma_dgesv(m, n, A, m, piv, b, m, &queue, &info );
73 // Solution is stored in b
74
75 free(A);
76 free(b);
77 free(piv);
78 magma_finalize();
79 return 0;
80 }
```

The clMAGMA system is set up on Line 22. This function sets up the system context that will be used by clMAGMA. The clMAGMA functions that are computed on the GPU will also require specification of the command queue where the task will

be assigned. To do this, clMAGMA needs to know about the devices being used for this calculation, hence a list of devices is requested on Line 27. In this simple example, just one queue, of type `magma_queue_t`, is created and assigned to the first device in the device list on Line 31. Space for the matrix `A` and the right-hand-side vector `b` is created on Lines 50 and 51. The temporary pivot data can be allocated using the usual `malloc` function on Line 55.

The function `magma_dgesv` on Line 72 takes almost the same arguments as the LAPACK `dgesv_` function and in the same order. The only difference is that a pointer to the work queue to which the task will be sent must be added as the eighth argument. Upon return, the solution is stored in the vector `b` and any error information is stored in the variable `info`.

Finally, `magma_finalize` is called to clean up after using clMAGMA on Line 78.

For a matrix `A` with dimensions  $16,384 \times 16,384$ , a test comparison between the clMAGMA package and LAPACK produced a solution approximately four times faster. This may not seem like much, but one caveat to consider is that the LAPACK used in this test was from the Mac OS X Accelerate library, which optimizes the LAPACK routines to utilize the vector processing capabilities of the specific hardware. More substantial differences would be expected had the LAPACK package been a standard serial implementation. Regardless, utilizing the compute capability of the GPU does result in significant savings.

### 32.2.2 • Compiling and Linking clMAGMA Code

In order to compile and link the program listed in Example 32.3, a number of libraries must be included. For the compile line, use the following command:

```
g++ -m64 -O3 -DADD_ -DHAVE_clBLAS -DHAVE_CBLAS  
-I/usr/local/clmagma/include -I/usr/local/include -c program.c
```

The first thing to notice is the `gcc` compiler has been replaced by the `g++` compiler. This is because some of the clMAGMA header files include unguarded lines containing `extern "C"`, which is a C++ line for telling C++ that a given function should be treated as a plain C function. Unfortunately, the regular C compiler doesn't understand that construct. Rather than modifying the header files to make the whole system C compiler friendly, using the C++ compiler instead is a quick workaround even if it does produce some warnings. It is possible this change is not required on your system.

The `-D` flag is equivalent to putting `#define` at the top of the file. These flags are the correct settings for the current version of clMAGMA, so be sure to include these flags.

The `-I` flag instructs the compiler where to look for header files in addition to the defaults. In this case, the headers files from both the clBLAS package and from the clMAGMA package are not in the default list, so those directories are added using the `-I` flag. The location of your library may differ, so those paths may need adjustment.

The link line for this program on a Mac OS X system is

```
g++ -o program program.o -L/usr/local/magma/lib -lclmagma  
-lblas_fix -framework Accelerate -L/usr/local/lib -lclBLAS  
-framework OpenCL
```

where the cIMAGMA and cIBLAS libraries are explicitly linked along with links to the Accelerate and OpenCL frameworks provided by Apple. If using a non-Apple system, then the Accelerate framework would be replaced by the location of LAPACK from Section 7.1. Check with your system administrator to find out the locations of these libraries for your particular system.

### 32.2.3 • CPU Interface Versus GPU Interface

In Example 32.3, the results of the computation were left on the host. That may or may not be what is desired. More likely, the result should stay on the device rather than shifting it back and forth. For that reason, cIMAGMA also has a GPU interface where the inputs and the results are kept on the GPU. However, the input data must still be put on the device and the results read back off of it. The GPU interface version of Example 32.3 is shown below in Example 32.4.

#### Example 32.4.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4 #include <string.h>
5 #include "magma.h"
6 #include "magma_lapack.h"
7 /*
8 int main(int argc, char* argv[])
10
11 A matrix A is created in column-major format
12 and a right-hand side is created. The system of equations
13 is then solved storing the solution the vector b upon return.
14
15 Inputs: none
16
17 Outputs: none
18 */
19 int main(int argc, char* argv[])
20 {
21     // Initialize the MAGMA system
22     magma_init();
23
24     // Get device info, using default device 0
25     magma_int_t num;
26     magma_device_t devices[2];
27     magma_getdevices( devices, 2, &num);
28
29     // temporary data required by dgesv similar to LAPACK
30     magma_int_t *piv, info;
31
32     // the matrix A is m x m
33     magma_int_t m = 9;
34
35     // the right-hand side has dimension m x n
36     magma_int_t n = 1;
37
38     // error code for MAGMA functions
39     magma_int_t err;
40
41     // The matrix A and right-hand-side b
42     double* A;
43     double* b;
```

```

44    magmaDouble_ptr dev_A;
45    magmaDouble_ptr dev_b;
46
47    // Allocate matrices on the host
48    err = magma_dmalloc_cpu(&A, m*m);
49    err = magma_dmalloc_cpu(&b, m);
50
51    // Allocate matrices on the device
52    err = magma_dmalloc(&dev_A, m*m);
53    err = magma_dmalloc(&dev_b, m);
54    // Note: should check that err == 0, meaning call was successful
55
56    // Create temporary storage for the pivots.
57    piv = (magma_int_t*)malloc(m*sizeof(magma_int_t));
58
59    // Generate matrix A
60    double row[9] = {3., 21., 4., 1., 13., 1., 7., 13., 1.};
61    int i, j, index = 0;
62    for (j=0; j<9; ++j)
63        for (i=0; i<9; ++i)
64            A[index++] = (i > j ? -1 : 1) * row[j];
65
66    // Generate b
67    b[0] = 17;
68    for (i=0; i<4; ++i) {
69        b[i+1] = 11;
70        b[i+5] = -15;
71    }
72
73    // set up the queue that will do the work
74    magma_queue_t queue;
75    magma_queue_create(devices[0], &queue);
76
77    // copy matrix from host to device
78    magma_dsetmatrix(m, m, A, m, dev_A, 0, m, queue);
79    magma_dsetmatrix(m, n, b, m, dev_b, 0, m, queue);
80
81    // MAGMA solve
82 #ifdef USE_DGETRF
83    magma_dgetrf_gpu(m, m, dev_A, 0, m, piv, queue, &info);
84    magma_dgetrs_gpu(MagmaNoTrans, m, n, dev_A, 0, m, piv, dev_b, 0, m,
85                      queue, &info);
86 #else
87    magma_dgesv_gpu(m, n, dev_A, 0, m, piv, dev_b, 0, m, queue, &info);
88 #endif
89
90    // copy solution in dev_b back onto host
91    magma_dgetmatrix(m, n, dev_b, 0, m, b, m, queue);
92
93    free(A);
94    free(b);
95    magma_free(dev_A);
96    magma_free(dev_b);
97    free(piv);
98    magma_finalize();
99    return 0;
100}

```

In order to take advantage of the GPU interface, the matrix **A** and the right-hand-side **b** must be put onto the device before calling the solver. The data for the matrices is first set on the CPU in the same way as in Example 32.3. However, space for the

matrices must be allocated on the device as well, and for that the clMAGMA data type `magmaDouble_ptr` is used on Line 44. The space is allocated and assigned to the corresponding pointers on Lines 52 and 53. Once the data is set up on the host side, the data is transferred into the device memory using the function `magma_dsetmatrix` on Lines 78 and 79. Note that for most functions in the clMAGMA library, where the operation is performed on device memory, the functions will request both the memory address *and* an offset. In this example, the data is aligned with the allocated memory, no offset is needed, and there is a zero in the sixth argument of `magma_dsetmatrix` on Lines 78 and 79.

The linear solver can now be called, but here there are two different solvers shown. The use of conditional compiling has come up multiple times in this text, most recently in Section 29.3, so it should suffice to say that to use Lines 83–85 the macro `USE_DGETRF` must be defined; otherwise Line 87 is used.

If the `USE_DGETRF` macro is undefined, then the primary difference between Examples 26.1 and 26.2 is the use of the GPU interface solver `magma_dgesv_gpu`. Both the matrix `dev_A` and right-hand-side matrix `dev_b` are expected to be in device memory for this solver. Again, because device memory is specified, the function also requests offsets, hence the zeros in the fourth and eighth arguments of `magma_dgesv_gpu` as shown on Line 87. Otherwise, the arguments are the same as in the previous example.

If the `USE_DGETRF` macro is defined, then the functions `magma_dgetrf_gpu` and `magma_dgetrs_gpu` are used to solve the system. The primary purpose for separating the solution into two steps is because it is often the case that the matrix to be inverted remains constant, while the right-hand side may vary across multiple solutions. In that case, `magma_dgetrf_gpu` is called *once* to factor the matrix in `dev_A`, and the resulting factorization can be reused for multiple calls to `magma_dgetrs_gpu` leading to reduced computational cost. Another difference between the methods is that `magma_dgetrs_gpu` also takes an additional first argument that is either `MagmaNoTrans` or `MagmaTrans`. For `MagmaNoTrans`, the linear system  $\mathbf{A}\mathbf{x} = \mathbf{b}$  is solved, while for `MagmaTrans`,  $\mathbf{A}^T\mathbf{x} = \mathbf{b}$  is solved. That means that if the matrix  $\mathbf{A}$  is stored in row-major order, then using `MagmaTrans` would mean solving the linear system without having to transpose the matrix first. If `dev_b` is a matrix with more than one column, then it still must be stored in column-major order either way. Note that for complex matrices, there is the additional `MagmaConjTrans` corresponding to solving  $\mathbf{A}^H\mathbf{x} = \mathbf{b}$ .

For either method, upon completion, the result is stored in the `dev_b` vector on the device. To retrieve the results from the device, use `magma_dgetmatrix` as shown on Line 91.

The last difference between Example 32.4 and Example 32.3 is that the device memory allocated uses the `magma_free` function on Line 95 to release the memory on the device.

## 32.3 • clFFT

The clMath package is an open-source math library that began as a project by AMD. The library is available on Github at

<https://github.com/clMathLibraries>

It includes an FFT library called clFFT that is similar to the FFTW library discussed in other parts of this book. One key difference is that it assumes that the input and output will be on the device; it doesn't provide a CPU interface like the clMAGMA

package. Example 32.5 shows a simple code to compute a one-dimensional FFT in double precision:

**Example 32.5.**

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4 #include <clFFT.h>
5
6 #ifndef M_PI
7 #define M_PI 3.1415926535897932384626433832795
8 #endif
9
10 typedef struct{ double r, c; } dcomplex;
11
12 /*
13 Demonstrate a simple example of performing a forward
14 and backward Fourier transform
15
16 Inputs: argc should be 2
17 argv[1]: wave number in initial data
18
19 Outputs: the initial data and its derivative.
20 */
21
22 int main( int argc, char* argv[] )
23 {
24     // Get wave number from input
25     int K = atoi(argv[1]);
26
27     // Using a length 16 data set
28     size_t N = 16;
29
30     // Request a platform
31     cl_platform_id platform;
32     cl_int err = clGetPlatformIDs( 1, &platform, NULL );
33
34     // Next get the device ID
35     cl_device_id device_id;
36     err = clGetDeviceIDs( platform, CL_DEVICE_TYPE_GPU, 1, &device_id,
37                           NULL );
38
39     // Set up the context for the GPU
40     cl_context context;
41     context = clCreateContext(0, 1, &device_id, NULL, NULL, &err);
42
43     // Create a command queue for the device
44     cl_command_queue queue;
45     queue = clCreateCommandQueueWithProperties( context, device_id, NULL,
46                                               &err );
47
48     // Set up the initial data for a single wave number K
49     double dx = 2.*M_PI/N;
50     dcomplex* x = (dcomplex*)malloc(N*sizeof(dcomplex));
51     int i;
52     for (i=0; i<N; ++i) {
53         x[i].r = cos(K*dx*i);
54         x[i].c = sin(K*dx*i);
55     }
56
57     // Set up memory on the device for the calculation
58     cl_mem dev_x = clCreateBuffer( context, CL_MEM_READ_WRITE,

```

```

59                                     N*sizeof(dcomplex), NULL, &err );
60
61 // Copy the input data into device memory
62 err = clEnqueueWriteBuffer( queue, dev_x, CL_TRUE, 0,
63                             N*sizeof(dcomplex), x, 0, NULL, NULL );
64
65 // Set up clFFT
66 clfftSetupData fftSetup;
67 err = clfftInitSetupData(&fftSetup);
68 err = clfftSetup(&fftSetup);
69
70 // Create a default plan for a complex ID FFT and set options
71 clfftPlanHandle planHandle;
72 err = clfftCreateDefaultPlan(&planHandle, context, CLFFT_1D, &N);
73 // Will use double precision
74 err = clfftSetPlanPrecision(planHandle, CLFFT_DOUBLE);
75 // Will use interleaved values for complex arrays
76 err = clfftSetLayout(planHandle, CLFFT_COMPLEX_INTERLEAVED,
77                      CLFFT_COMPLEX_INTERLEAVED);
78 // Results will be put in place of the input buffer
79 err = clfftSetResultLocation(planHandle, CLFFT_INPLACE);
80
81 // Construct the plan before executing
82 err = clfftBakePlan(planHandle, 1, &queue, NULL, NULL);
83
84 // Compute the forward Fourier transform
85 err = clfftEnqueueTransform(planHandle, CLFFT_FORWARD, 1, &queue, 0,
86                            NULL, NULL, &dev_x, NULL, NULL);
87
88 // Wait for the computation to finish before extracting the results
89 err = clFinish(queue);
90
91 // Copy the results from the device back to the host
92 err = clEnqueueReadBuffer( queue, dev_x, CL_TRUE, 0,
93                           N*sizeof(dcomplex), x, 0, NULL, NULL );
94
95 // Print the results
96 for (i=0; i<N; ++i)
97     printf("(%.f, %.f) ", x[i].r, x[i].c);
98 printf("\n");
99
100 // Compute the backward Fourier transform
101 err = clfftEnqueueTransform(planHandle, CLFFT_BACKWARD, 1, &queue, 0,
102                            NULL, NULL, &dev_x, NULL, NULL);
103
104 // Wait for the computation to finish before extracting the results
105 err = clFinish(queue);
106
107 // Copy the results from the device back to the host
108 err = clEnqueueReadBuffer( queue, dev_x, CL_TRUE, 0,
109                           N*sizeof(dcomplex), x, 0, NULL, NULL );
110
111 // Print the results
112 for (i=0; i<N; ++i)
113     printf("(%.f, %.f) ", x[i].r, x[i].c);
114 printf("\n");
115
116 // Free device memory
117 clReleaseMemObject( dev_x );
118
119 free(x);
120
121 // Destroy the plan

```

```

122 err = clfftDestroyPlan( &planHandle );
123
124 // Free internal clFFT memory created in setup
125 clfftTeardown( );
126
127 // Clean up remaining OpenCL objects
128 clReleaseCommandQueue( queue );
129 clReleaseContext( context );
130
131 return 0;
132 }
```

The clFFT package doesn't supply its own complex variable type because it isn't strictly necessary, but for this example, a `typedef` command to create a new complex number type `dcomplex` on Line 10 is used. The `r` and `c` components of the `dcomplex` type contain the real and imaginary parts of the data. By doing this, an array created with this data type, as on Line 50, will be stored in an interleaved format, i.e., the array will have the form

x[0].r
x[0].c
x[1].r
x[1].c
:
x[N-1].r
x[N-1].c

It is not necessary to use the `dcomplex` struct—a simple double precision array could be used to store the values—but using the `dcomplex` struct makes it easier for indexing purposes.

The data may also be arranged in planar format, where the real part and the imaginary part of the data are stored contiguously. Mapping the interleaved data into the planar format would result in the data arranged this way:

x[0].r
x[1].r
:
x[N-1].r
x[0].c
x[1].c
:
x[N-1].c

Example 32.5 uses the interleaved format as indicated on Line 76. The planar format would be specified with `CLFFT_COMPLEX_PLANAR`. The FFT will only operate on data on the device, so a memory buffer is created on the device and the data is transferred on Lines 58 and 62.

Before actually doing any transforms, the clFFT package needs to be initialized in a two-step process on Lines 67 and 68. Next, a plan is created for a one-dimensional,

complex-valued transform for a data vector of length  $N$  on Line 72. Two-dimensional and three-dimensional transforms are specified by CLFFT\_2D, CLFFT\_3D, respectively. The length  $N$  must have value of the form  $2^m 3^n 5^p 7^q$  for nonnegative integers  $m, n, p, q$ . The most efficient transforms are when  $N = 2^m$ . On Line 74, the plan is set so it will use double precision as opposed to the default single precision. Finally, the results will be placed on top of the input data by using the label CLFFT\_INPLACE. If CLFFT\_OUTOFPLACE is used, then a second buffer on the device will be required where the output of the transform will be stored. Once the options are set, the kernel functions are built internally when completing the plan by “baking” it on Line 82. The plan is now ready to compute transforms.

The forward transform is computed on Line 85 and the backward transform is computed on Line 101. For this package, the forward transform results in the data being organized so that the spectral coefficients  $a_k$  appear in the following order multiplied by an additional factor  $N$ :

$Na_0$
$Na_1$
$\vdots$
$Na_{N/2}$
$Na_{-N/2+1}$
$\vdots$
$Na_{-2}$
$Na_{-1}$

The reverse transform divides out the factor of  $N$  again so that a forward transform followed by a backward transform results in the original data.

Finally, there is a bit of clean-up at the end that must be done for clFFT in addition to the clean-up for OpenCL. The plan is released by calling `clfftDestroyPlan` on Line 122. The clFFT framework that was set up on Line 67 is dismantled by a call to `clfftTeardown` on Line 125.

### 32.3.1 • Two-Dimensional Transforms

Transforming data in two and higher dimensions is accomplished in very much the same way as described for one dimension above. It is important to remember that unlike the FFTW libraries described elsewhere in this text, the clFFT library data is stored in C-friendly row-major order. This will be particularly relevant when computing spectral derivatives in the pseudospectral method applications in Chapter 37. A two-dimensional plan `p` to compute the forward transform on an  $M \times N$  grid is constructed like this:

```

1  size_t MbyN[2] = {M, N};
2  clfftPlanHandle plan;
3  clfftCreateDefaultPlan(&plan, context, CLFFT_2D, MbyN);
4  clfftSetPlanPrecision(plan, CLFFT_DOUBLE);
5  clfftSetLayout(plan, CLFFT_COMPLEX_INTERLEAVED,
6                           CLFFT_COMPLEX_INTERLEAVED);
7  clfftSetResultLocation(plan, CLFFT_OUTOFPLACE);
8  clfftBakePlan(plan, 1, &queue, NULL, NULL);

```

The dimensions of the transforms are specified in `clfftCreateDefaultPlan` on Line 3. The third argument, CLFFT\_2D, means it's a two-dimensional data set, and the dimen-

sions of the data are specified in the last argument. For this plan, the location of the result is set to CLFFT\_OUTOFPLACE on Line 7. Thus, when the plan is executed both an input and an output array must be provided such as

```
clfftEnqueueTransform( planHandle , CLFFT_FORWARD, 1, &queue , 1,
                      wait_events , &this_event , &dev_input , &dev_output , NULL);
```

Here the input data is in `dev_input` and the output data is in `dev_output`. For illustrative purposes, note that two events have been added. The argument `wait_events` is an array of `c1_event` events that the transform should wait for before executing the transform. The number preceding the `wait_events` list is the number of entries in the list. The event attached to this transform task is the argument `this_event`. See Section 31.1 for a discussion on the use of events to control tasks in a command queue.

The order of the coefficients will have the same structure in each dimension as for the one-dimensional results so that the array is organized as below:

$a_{0,0}$	$a_{0,1}$	$\cdots$	$a_{0,\frac{N}{2}}$	$a_{0,1-\frac{N}{2}}$	$a_{0,2-\frac{N}{2}}$	$\cdots$	$a_{0,-1}$
$a_{1,0}$	$a_{1,1}$	$\cdots$	$a_{1,\frac{N}{2}}$	$a_{1,1-\frac{N}{2}}$	$a_{1,2-\frac{N}{2}}$	$\cdots$	$a_{1,-1}$
$\vdots$	$\vdots$		$\vdots$	$\vdots$	$\vdots$		$\vdots$
$a_{\frac{M}{2},0}$	$a_{\frac{M}{2},1}$	$\cdots$	$a_{\frac{M}{2},\frac{N}{2}}$	$a_{\frac{M}{2},1-\frac{N}{2}}$	$a_{\frac{M}{2},2-\frac{N}{2}}$	$\cdots$	$a_{\frac{M}{2},-1}$
$a_{1-\frac{M}{2},0}$	$a_{1-\frac{M}{2},1}$	$\cdots$	$a_{1-\frac{M}{2},\frac{N}{2}}$	$a_{1-\frac{M}{2},1-\frac{N}{2}}$	$a_{1-\frac{M}{2},2-\frac{N}{2}}$	$\cdots$	$a_{1-\frac{M}{2},-1}$
$a_{2-\frac{M}{2},0}$	$a_{2-\frac{M}{2},1}$	$\cdots$	$a_{2-\frac{M}{2},\frac{N}{2}}$	$a_{2-\frac{M}{2},1-\frac{N}{2}}$	$a_{2-\frac{M}{2},2-\frac{N}{2}}$	$\cdots$	$a_{2-\frac{M}{2},-1}$
$\vdots$	$\vdots$		$\vdots$	$\vdots$	$\vdots$		$\vdots$
$a_{-1,0}$	$a_{-1,1}$	$\cdots$	$a_{-1,\frac{N}{2}}$	$a_{-1,1-\frac{N}{2}}$	$a_{-1,2-\frac{N}{2}}$	$\cdots$	$a_{-1,-1}$

### 32.3.2 • Transforming Real-Valued Data

When transforming real-valued data, some savings can be gained by using the plan functions that transform between real- and complex-valued data. In this case, the forward and backward plans in two dimensions would be created like this:

```
size_t MbyN[2] = {M, N};
// Build the forward plan
clfftPlanHandle fplan;
clfftCreateDefaultPlan(&fplan , context , CLFFT_2D, MbyN);
clfftSetPlanPrecision(fplan , CLFFT_DOUBLE);
clfftSetLayout(fplan , CLFFT_REAL , CLFFT_HERMITIAN_INTERLEAVED);
clfftSetResultLocation(fplan , CLFFT_OUTOFPLACE);
clfftBakePlan(fplan , 1, &queue , NULL, NULL);

// Build the backward plan
clfftPlanHandle bplan;
clfftCreateDefaultPlan(&bplan , context , CLFFT_2D, MbyN);
clfftSetPlanPrecision(bplan , CLFFT_DOUBLE);
clfftSetLayout(bplan , CLFFT_HERMITIAN_INTERLEAVED , CLFFT_REAL);
clfftSetResultLocation(bplan , CLFFT_OUTOFPLACE);
clfftBakePlan(bplan , 1, &queue , NULL, NULL);
```

This time, two separate plans are needed because the input and output data types are different for each. The first plan, `fplan`, is for a forward transform, and the second plan, `bplan`, is for a backward transform. For the one-dimensional transform, only the complex coefficients  $a_0, \dots, a_{N/2}$  are computed because the remaining coefficients are

the complex conjugates of these. In higher dimensions, only the last dimension is cut in half, so that the coefficients in two dimensions are arrayed as shown below:

$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$\cdots$	$a_{0,\frac{N}{2}}$
$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$\cdots$	$a_{1,\frac{N}{2}}$
$\vdots$	$\vdots$	$\vdots$		$\vdots$
$a_{\frac{M}{2},0}$	$a_{\frac{M}{2},1}$	$a_{\frac{M}{2},2}$	$\cdots$	$a_{\frac{M}{2},\frac{N}{2}}$
$a_{1-\frac{M}{2},0}$	$a_{1-\frac{M}{2},1}$	$a_{1-\frac{M}{2},2}$	$\cdots$	$a_{1-\frac{M}{2},\frac{N}{2}}$
$a_{2-\frac{M}{2},0}$	$a_{2-\frac{M}{2},1}$	$a_{2-\frac{M}{2},2}$	$\cdots$	$a_{2-\frac{M}{2},\frac{N}{2}}$
$\vdots$	$\vdots$	$\vdots$		$\vdots$
$a_{-1,0}$	$a_{-1,1}$	$a_{-1,2}$	$\cdots$	$a_{-1,\frac{N}{2}}$

### 32.3.3 • Linking with the clFFT Library

When building code to use the clFFT library, the additional library is included by adding `-lclFFT` to the linking line.

## Exercises

- 32.1. Modify Example 32.4 to solve a linear system of equations using the GPU interface for multiple right-hand-side vectors.
- 32.2. Write a program to generate  $N$  random uniformly distributed values on the device, and then write a kernel that sums all the values. See the discussion in Section 33.1.1.
- 32.3. Write a kernel to generate an  $N \times N$  array of random values using a uniform distribution in the range  $[0, 1]$ . Verify that no two values are repeated, and if the same seed is used twice, then the full array is exactly the same.
- 32.4. Write a kernel to compute the first derivative of the data in an array of length  $N$  following the discussion in Section 37.2.
- 32.5. Modify Example 32.5 to do a two-dimensional FFT transform. Take as inputs to your program the integer wave modes  $k_1$  and  $k_2$  so that the complex input data are

```
in[i][j][0] = cos((k1*i+k2*j)*dx);
in[i][j][1] = sin((k1*i+k2*j)*dx);
```

Print out the values of the spectral coefficients after the forward transform. Perform the reverse transform and verify that the initial data is recovered. Try different values of  $k_1$  and  $k_2$  and note how the spectral coefficients depend on the values.

- 32.6. Modify Example 32.5 to do a two-dimensional FFT transform of real-valued data.



## Chapter 33

# Projects for OpenCL Programming

The background for the projects below are in Part VI of this book. You should be able to build each of these projects based on the information provided in this text and utilize OpenCL to implement a GPU version or a combined GPU/CPU version of these projects. Where appropriate, there are some additional comments specific to OpenCL that will help to understand how to develop your algorithms.

### 33.1 • Random Processes

#### 33.1.1 • Monte Carlo Integration

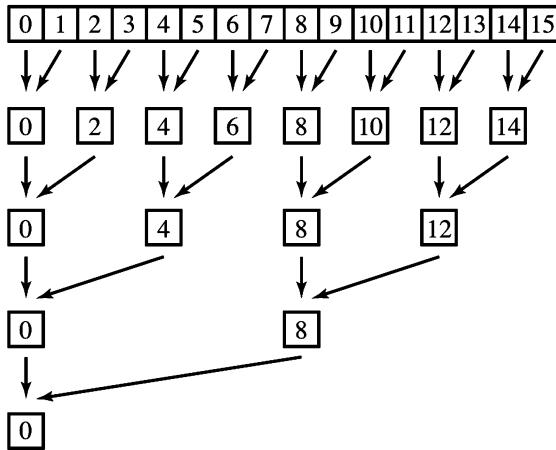
Because random number generation using the library Random123 relies on individual work-items to generate the random values, it makes sense to use a maximum number  $M$  of work-items to each compute a random value, apply  $f(x)$ , and then keep a running total for the sum for  $N/M$  samples. The resulting sum should be stored in device memory when completed.

A second kernel should be used to do a reduction operation to sum the  $M$  subtotals and produce the final mean. If  $M$  is large, then it makes sense to use a kernel that recursively sums two values in an array and places the result back in the array. The kernel simply sums two numbers in an array for specified indices and stores the result in the lower of the two indices. The kernel is then applied to the entire array resulting in  $M/2$  entries that are the sum of the two neighboring values. The kernel is reapplied to the  $M/2$  entries to get  $M/4$  entries, and so forth until the total is reduced. Figure 33.1 illustrates the process. This makes the reduction a  $\log_2 M$  operation.

Program the assignment in Section 34.3.1 using  $N$  samples, where  $N$  is an input parameter. Measure the time required to complete the calculation by using the `gettimeofday` function at the beginning and end of your program and print the elapsed time in seconds. Use the plot to estimate the cost to compute a solution with error  $10^{-5}$  with 99.5% confidence.

#### 33.1.2 • Duffing–Van der Pol Oscillator

When solving many stochastic differential equations simultaneously, the various realizations of the solution are all completely independent, so it's simple enough to have each work-item solve one SDE from beginning to end and then utilize as many work-items as possible in each work-group. Since the output only involves tallying the hit



**Figure 33.1.** Illustration of how a single kernel is used to do pairwise sums and storing the results in place in order to sum the total of the array. The whole array is summed in  $\log_2 N$  operations. The numbers indicate which array entries contain the current subtotals for the boxes to their right.

counts along the way, everything should be done in local memory except for incrementing the hits in global memory so it can be retrieved later.

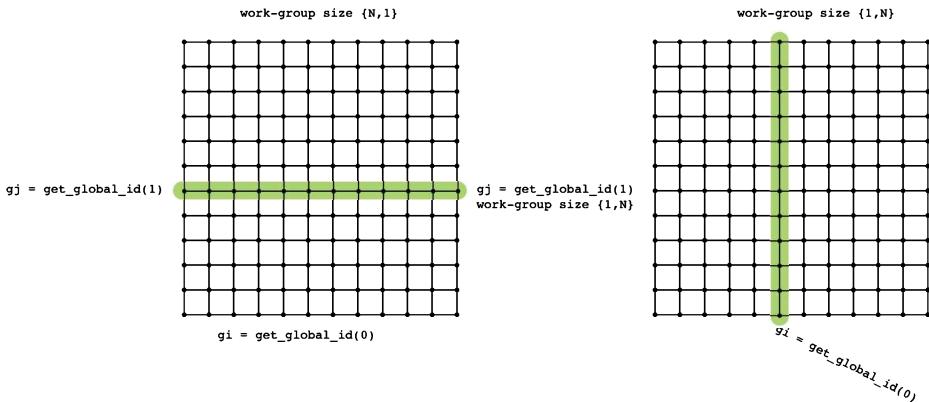
Program the assignment in Section 34.3.2 using  $N$  samples and using  $M$  time steps, i.e., take  $\Delta t = T/M$  in the Euler–Maruyama method. The values of  $N$ ,  $M$ , and  $\sigma$  should be given on the command line. Use  $\alpha = 1$ , which should be defined in the program. Measure the time required to complete the calculation by using the `gettimeofday` function at the beginning and end of the program and print the elapsed time in seconds. Plot an estimate for  $p(t)$  for  $0 \leq t \leq T = 10$ .

## 33.2 • Finite Difference Methods

For codes that involve grids, there are many ways in which the work-groups and work-items can be mapped onto the grid, and the choice depends heavily on the dimensions of the grid being used and the exact calculation being performed. For purposes of discussion, assume the maximum number of work-items per work-group is  $1024 = 32 \times 32$ .

For the ADI method being used for these projects, the explicit update steps are entirely one-dimensional, i.e., there is no interaction between parallel rows or columns of the grid. It makes sense to organize the work-groups so that each work-group handles one row and uses the number of available work-items. Thus, for an  $N \times N$  grid, where  $N \leq 1024$ , the one-dimensional explicit update kernel could be called with  $N$  work-groups, each with  $N$  work-items. The work-group size depends on the direction of the update as illustrated in Figure 33.2. If  $N > 1024$ , or greater than the maximum number of work-items per work-group, then the number of work-groups in the long dimension would have to be bigger.

If the finite difference stencil is two-dimensional as it is for the explicit with Runge–Kutta method, then the one-dimensional handling of work-groups is not so favorable, and it is better to go with a square arrangement of work-items. For the parameters assumed here, that would be a  $32 \times 32$  arrangement of work-items. The grid is then



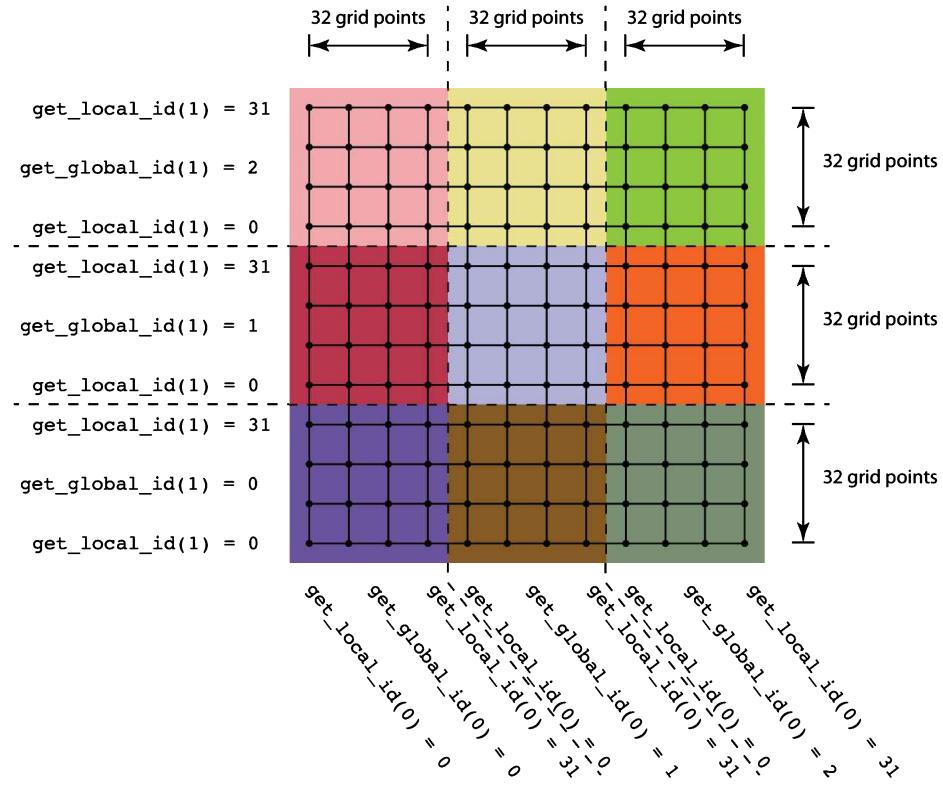
**Figure 33.2.** Illustration of having each work-group compute an operation on a single row or column in the grid. On the left, the work-group contains a row of the grid and the work-group size is  $\{N, 1\}$ . On the right, the work-group contains a column of the grid and the work-group size is  $\{1, N\}$ .

tiled in a manner as illustrated in Figure 33.3. For the sake of argument, assume there is a standard five-point stencil for doing a finite difference operation; in other words, to update  $u_{i,j}$ , the values  $u_{i,j}, u_{i\pm 1,j}, u_{i,j\pm 1}$  are required. Within this context, there are two choices for how to manage the edges of the local memory space: (1) use a local memory array that is  $34 \times 34$  and use the work-items at the edges to load the extra edge data, or (2) use a local memory array that is  $32 \times 32$  and have only the inner work-items perform the finite difference calculation. The two options are illustrated in Figure 33.4.

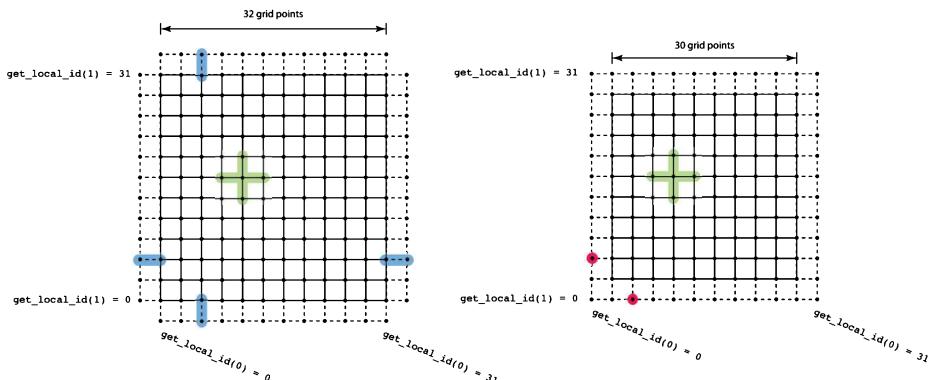
For the local grid on the left in Figure 33.4, there is an extra cost for having some work-items load two points (or three points for the case of the corner work-items) instead of one. The interior work-items must wait until the extra loads into shared memory are completed. By comparison, in the figure on the right, all work-items load only one value into local memory. However, there is a trade-off, because the grid on the left can update  $32^2$  points, while the one on the right can update only  $30^2$  because the edge work-items remain idle. There is no perfect solution, but for the way that OpenCL specifies the organization of work-groups and work-items, the bookkeeping for the case on the right is significantly more complicated than the one on the left. That is the method used for generating the data in Figure 31.2.

Once the numerical method and grid strategy are selected, multiple kernels will be needed to implement the algorithm. For the explicit with Runge–Kutta method, just one kernel is needed to handle one stage of the Runge–Kutta method. That kernel should take the stage number or the fraction of the time step that is different for each stage as an argument.

For the ADI method, two kernels are needed for the explicit steps, one for the  $x$ -direction and one for the  $y$ -direction, because the indexing in the data is different. Otherwise, these should be fairly simple adaptations of the `diff` kernel in the examples. The implicit steps will use the general matrix solver functions `magma_dgetrf_gpu` and `magma_dgetrs_gpu` from the clMAGMA library. Remember that the matrix that has to be inverted does not change with time, so that means `magma_dgetrf_gpu` should be used *once* to factor the matrix, and thereafter use `magma_dgetrs_gpu` to get the solutions, always leaving the data on the device. If the domain is square and the boundary conditions the same for both the  $x$ - and  $y$ -directions, then the factorization is done only



**Figure 33.3.** Illustration of tiling a complete grid with a  $3 \times 3$  array of work-groups that are each  $32 \times 32$  grid points.



**Figure 33.4.** Illustration of the two options for using a local square grid for handling a two-dimensional finite difference stencil. The green highlight illustrates the shape of the stencil for a representative grid point. On the left, the local grid is  $34 \times 34$ , and each work-item loads one grid value from global into local memory. Work-items on the edge, e.g., `work-item(0, j)` or `work-item(31, j)`, must load two values as illustrated by the blue highlight. All work-items participate in computing the finite difference. On the right, the local grid is  $32 \times 32$ , and each work-item loads only one value from global to local memory. The work-items on the edge, as illustrated by the red highlight, do not participate in the finite difference calculation.

once, but if the domain is not square or if the boundary conditions vary between the directions, then the factorization is done twice, once for each direction. For the solution, `magma_dgetrs_gpu` only needs to be called once each direction per iteration because it can be applied to the entire data set, i.e., it can solve for multiple rows or columns of the array in one call.

No matter which linear solver used, the right-hand side must still be ordered in the right direction so that the right-hand-side data is contiguous. To facilitate that, a simple kernel for transposing the grid data would be helpful.

### 33.2.1 • Brusselator Reaction

Program the assignment in Section 35.4.1 using an  $N \times N$  grid. In addition to the kernels discussed above, a kernel to initialize the domain with random values will also be needed. Use the methods described in Section 32.1 to populate the initial values for both  $U$  and  $V$ . Measure the time required to complete the calculation by using the `gettimeofday` function at the beginning and end of your program and print the elapsed time in seconds.

### 33.2.2 • Linearized Euler Equations

Program the assignment in Section 35.4.2 using an  $N \times N$  grid. In addition to the kernels discussed above, a kernel to initialize the grid will also be needed. Measure the time required to complete the calculation by using the `gettimeofday` function at the beginning and end of your program and print the elapsed time in seconds.

## 33.3 • Elliptic Equations and SOR

For maximal efficiency, write two kernels, one for the red points and one for the black, using the red/black ordering scheme. Store the residual in a separate array so that it can be tested for convergence. An additional kernel will be needed for a reduction operation of the maximum absolute value in the residual. The reduction operation should use the same scheme as described in Section 33.1.1.

### 33.3.1 • Diffusion with Sinks and Sources

Program the assignment in Section 36.1.1 using a  $(2N - 1) \times N$  grid. Experiment to find the optimal  $\omega$  that requires the fewest iterations to reach a tolerance of  $10^{-9}$ . Compute the time required to complete the calculation using the `gettimeofday` function, then divide by the number of iterations to give the average time used for a single pass through the grid. Repeat these steps for a grid of dimensions  $(4N - 1) \times 2N$ . Does the optimal value of  $\omega$  remain the same? Verify that the time cost for a single pass through the grid is proportional to the number of grid points.

### 33.3.2 • Stokes Flow 2D

Considering that the boundary conditions are slightly different for the three different variables, and that the grid dimensions are different, specialized kernels for each would be a good choice.

Program the two-dimensional assignment in Section 36.1.2 using a MAC grid that is approximately  $N \times N$ , i.e., the grid for  $u$  should be  $N \times N - 1$ , the grid for  $v$  should be

$N - 1 \times N$ , and the grid for  $p$  should be  $N - 1 \times N - 1$ . Experiment to find the optimal  $\omega$  that requires the fewest iterations to reach a tolerance of  $10^{-9}$ . Compute the time required to complete the calculation using the `gettimeofday` function, then divide by the number of iterations to give the average time used for a single pass through the grid. Verify that you get a parabolic profile for the velocity field in the  $x$ -direction.

### 33.3.3 • Stokes Flow 3D

Program the three-dimensional assignment in Section 36.1.2 using a MAC grid that is approximately  $N \times N \times N$ , i.e., the grid for  $u$  should be  $N \times N - 1 \times N - 1$ , the grid for  $v$  should be  $N - 1 \times N \times N - 1$ , the grid for  $w$  should be  $N - 1 \times N - 1 \times N$ , and the grid for  $p$  should be  $N - 1 \times N - 1 \times N - 1$ . Experiment to find the optimal  $\omega$  that requires the fewest iterations to reach a tolerance of  $10^{-9}$ . Compute the time required to complete a single pass through the grid. Verify that you get a roughly parabolic profile for the velocity field in the  $x$ -direction.

## 33.4 • Pseudospectral Methods

The Fourier transforms will be handled using the clFFT library keeping data on the device. One or more separate kernels will be needed to perform the pseudospectral derivative operations. In addition, kernels to initialize the grid with random values and to compute a stage of the Runge–Kutta method will be needed.

### 33.4.1 • Complex Ginsburg–Landau Equation

Because this is a complex-valued equation, use a storage layout for the clFFT plan that uses complex values, that is, use `CLFFT_HERMITIAN_INTERLEAVED` for both the input and output layouts.

Program the assignment in Section 37.5.1 using an  $N \times N$  grid. Measure the time required to complete the calculation by using the `gettimeofday` function at the beginning and end of your program and print the elapsed time in seconds. Plot the results for the phase and identify where the spiral waves have emerged through pattern formation.

### 33.4.2 • Allen–Cahn Equation

Because this is a real-valued equation, some gain in efficiency can be made by taking advantage of the reduced storage and lower computational cost of using the real to complex transforms described in Section 32.3.2. There are a couple advantages to doing this. First, there is less data to be read and fewer computations required because it is known that the imaginary part of a real value is zero. Second, numerical errors in the full complex transform will inevitably lead to small but nonzero values in the imaginary parts of the solution even though the equation is real-valued. These small errors could potentially pollute the solution later, or at the very least make it require extra effort to plot the results when done.

Program the assignment in Section 37.5.2 in two dimensions using an  $N \times N$  grid. Measure the time required to complete the calculation by using the `gettimeofday` function at the beginning and end of your program and print the elapsed time in seconds. Plot the results and verify that phase separation is occurring.

# **Part VI**

# **Applications**



This part describes some basic representative computational problems to be solved using the techniques presented in this text. These examples run the gamut from embarrassingly parallelizable methods to more subtle methods. The algorithms here are presented in a general context; implementation-specific differences are contained within the problem sets at the end of Parts I–V.

It is well beyond the scope of this book to present a comprehensive background on the mathematics of each representative problem. References for these topics are provided for the interested reader. Just the bare essentials necessary to understand the computational challenge is included.

### General Comments for all Applications

Each of the projects gives a list of input parameters to be used in that particular application. Those parameters are assumed to be given on the command line and accessed by using the `argv` variable in the `main` function. Rather than specifically numbering each argument, one quick way to make your program easier to edit later is to use a counter through the arguments. For example, one might use a variable `argi` to track which argument in the list is to be used, like this:

```

1 int main(int argc, char* argv[]) {
2
3     // Read arguments
4     int argi = 0;
5     int N = atoi(argv[+argi]);           printf("N = %d\n", N);
6     float T = atof(argv[+argi]);         printf("T = %f\n", T);
7     long int bigN = atol(argv[+argi]);   printf("bigN = %ld\n", bigN);
8     long int seed;
9     if (argi < argc-1) {
10         seed = atol(argv[+argi]);
11     } else {
12         // get random seed from another source
13     }
14     printf("seed = %ld\n", seed);
15
16     // argi now contains number of arguments read
17     // ... rest of program ...

```

By using this strategy, changes in the parameter list can be made without having to renumber all the argument indices. To verify that the arguments are read in correctly, the program should also print the values to the screen as illustrated above. For those applications that take an optional value for a random number generator seed, the value of `argc` must be checked to see whether an additional argument was given. Using the above framework, this is easily done by checking `argi < argc-1`. If this is true, then it must be that there is at least one more additional unread argument.

Almost all of the projects, Monte Carlo being the exception, expect the results to be stored in binary data files as discussed in Section 4.2 and illustrated in Example 4.2.

This means writing the data using the `fwrite` function and *not* the function `fprintf`. The `fwrite` function is better because it is more space efficient and loses no accuracy compared with printing text with `fprintf`.

For the time-dependent applications, the output is requested at specific intervals that includes both the initial conditions and the final values. In order to do that, the code for saving the data must appear in two places. One way to accomplish this is to save the initial data *before* the main loop, and then the rest of the data is saved inside, but at the end of the main loop like this where the value of the function  $u$  is saved at  $k = 0, 1, \dots, 10$ :

```

1 // assume u is initialized here
2 FILE* fileid = fopen("output.dat", "w");
3 fwrite(u, sizeof(double), N*N, fileid);
4
5 // other preloop stuff
6
7 for (int step=0; step < M; ++step) {
8
9     // do temporal update
10
11    if ((step+1)%M/10 == 0) {
12        fwrite(u, sizeof(double), N*N, fileid);
13    }
14}
15 fclose(fileid);
16
17 // other clean up

```

Alternatively, the `fwrite` can be put inside the beginning of the loop, and again outside the end of the loop:

```

1 // assume u is initialized here
2 FILE* fileid = fopen("output.dat", "w");
3
4 // other preloop stuff
5
6 for (int step=0; step < M; ++step) {
7
8    if (step%M/10 == 0) {
9        fwrite(u, sizeof(double), N*N, fileid);
10    }
11
12    // do temporal update
13
14}
15
16 fwrite(u, sizeof(double), N*N, fileid);
17 fclose(fileid);
18
19 // other clean up

```

Either method is acceptable.

## Chapter 34

# Stochastic Differential Equations

One of the easier types of equations to parallelize is the solution of stochastic ordinary differential equations. Numerical solutions of these equations require a large number of sample solutions in order to generate approximate distributions of the solution or to compute various statistical moments. Each individual simulation is comparably very inexpensive, but many simulations are required to generate the statistical data. Thus, it makes sense to have each simulation run by a single core, and then employ many cores to complete the computation.

### 34.1 • Mathematical Description

A stochastic differential equation is an equation of the form

$$dX_t = a(X_t, t) dt + b(X_t, t) dW_t, \quad X_0 = x_0, \quad (34.1)$$

where  $a(X_t, t)$  is the drift,  $b(X_t, t)$  is the diffusion, and  $W_t$  is a Wiener process, which is the stochastic or noise term in the equation. For those not familiar with stochastic differential equations, it may be helpful to consider the case where  $b(X_t, t) \equiv 0$  so that what remains is the *deterministic* equation

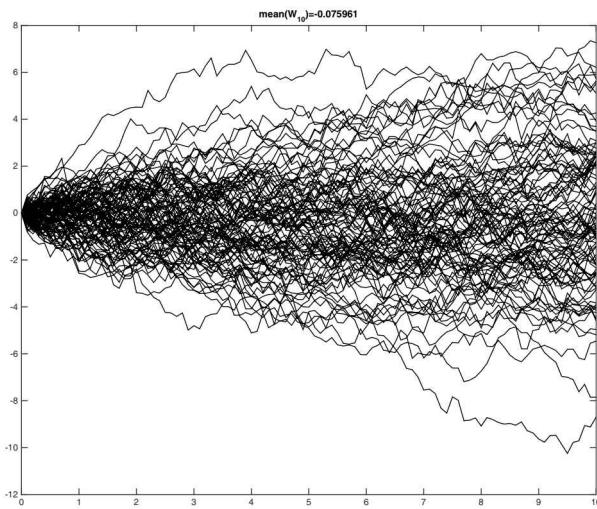
$$dX_t = a(X_t, t) dt, \quad X_0 = x_0, \quad (34.2)$$

which is more commonly written as the ordinary differential equation

$$X'(t) = a(X(t), t), \quad X(0) = x_0. \quad (34.3)$$

When  $b(X_t, t) \not\equiv 0$ , then stochastic noise is introduced. Writing  $dW_t/dt$  doesn't make sense because the Wiener process is not differentiable anywhere, hence the differential form as shown in Equation (34.1) is used. In this context,  $X_t$  is called a continuous time stochastic process, where  $X_t$  is a random variable associated with each time point  $t \in [0, \infty)$ .

The standard Wiener process is an example of a continuous time stochastic process with very specific properties. It starts at  $W_0 = 0$  and has mean or expected value of



**Figure 34.1.** Samples paths for 100 realizations of a Wiener process, i.e., solutions for Equation (34.5) on the interval  $0 \leq t \leq 10$ . By definition of a Wiener process, the expected value of  $X_{10}$  is  $E(X_{10}) = 0$ . In this example, where  $X_t^k$  is the  $k$ th sample path, the computed mean is  $\mu(X_{10}^k) \approx 0.076$ .

$E(W_t) = 0$ . It also has the property that each increment is independent with a Gaussian distribution. This means that the quantity  $\Delta W = W_t - W_s$  is itself a random variable with a Gaussian distribution that has no dependencies, i.e., it doesn't depend on  $s$ ,  $t$ , or the values  $W_s$ ,  $W_t$ , only on the difference  $t - s$  and has variance

$$\text{var}\{W_t - W_s\} = t - s \quad \text{for all } 0 \leq s < t. \quad (34.4)$$

Figure 34.1 illustrates 100 sample paths of the solution of the equation

$$dX_t = dW_t, \quad X_0 = 0, \quad (34.5)$$

in other words, samples paths for the standard Wiener process.

When solving stochastic differential equations, there are typically two different possible questions in mind. One question concerns the details of the sample paths, in which case it is important to get the paths correct. The other question is simpler in that it is about the statistical properties of the solution at a given time  $t$ . Which question is of interest can dictate the particular numerical method chosen to solve the problem. For the first question, one should use a strongly convergent method, while for the second question a weakly convergent method is sufficient. Strongly convergent methods are also weakly convergent, but not vice versa. Strongly convergent methods can be more expensive to compute than weakly convergent. Knowing up front which type of method you need is important to ensure maximal computational efficiency.

## 34.2 • Numerical Methods

### 34.2.1 • Euler–Maruyama Method

A simple numerical method that is 0.5-order strongly convergent, and first-order weakly convergent, is the Euler–Maruyama method given by

$$X_{t_{n+1}} = X_{t_n} + a(X_{t_n}, t_n) \Delta t + b(X_{t_n}, t_n) \Delta W_{t_n}, \quad (34.6)$$

where  $\Delta t$  is the time step size,  $t_n = n\Delta t$ , and  $\Delta W_{t_n}$  is a Gaussian distributed random variable with mean zero and variance  $\Delta t$ .

Each time step, the random variable  $\Delta W_{t_n}$  must be sampled from a standard Gaussian normal distribution with mean zero and variance  $\Delta t$ . Since many random number packages provide a normal Gaussian distribution generator, it is sufficient to sample such a distribution, which will have variance one, and then multiply by  $\sqrt{\Delta t}$  to get a resulting variance of  $\Delta t$ . Thus, if  $N(0; 1)$  is a normally distributed random variable with mean zero and variance one, then

$$\Delta W_{t_n} = \sqrt{\Delta t} N(0; 1).$$

The rest of the time step to advance from time  $t_n$  to  $t_{n+1}$  is straightforward using the formula in Equation (34.6).

For more information about numerical methods for stochastic differential equations, see the excellent text [24].

### 34.2.2 • Box–Muller and Polar Marsaglia Methods

Not all random number generator libraries provide a Gaussian distribution generator, but they do provide a uniform random number generator. Two methods for converting a uniform random number generator on the interval  $[0, 1]$  into a normally distributed Gaussian distribution with mean zero and variance one are presented here.

The first method is the Box–Muller method [25]. Let  $U_1, U_2$  be two independent random variables drawn from a uniform distribution on the  $[0, 1]$  interval. These two variables can be converted to two normally distributed random variables  $N_1, N_2$  by the equations

$$\begin{aligned} N_1 &= \sqrt{-2 \ln U_1} \cos(2\pi U_2), \\ N_2 &= \sqrt{-2 \ln U_1} \sin(2\pi U_2). \end{aligned}$$

An alternative method that avoids the cost of computing a trig function at the expense of having to discard a fraction of the numbers generated is called the polar Marsaglia method [26]. In this method, given two uniformly distributed random variables  $U_1, U_2$ , first compute

$$\begin{aligned} V_1 &= 2U_1 - 1, \\ V_2 &= 2U_2 - 1, \\ W &= V_1^2 + V_2^2. \end{aligned}$$

If  $W > 1$ , then start over by drawing two new values for  $U_1, U_2$ . Once there is a  $W \leq 1$ , then two normally distributed random variables are obtained by the formulae

$$\begin{aligned} N_1 &= \sqrt{\frac{-2 \ln W}{W}} V_1, \\ N_2 &= \sqrt{\frac{-2 \ln W}{W}} V_2. \end{aligned}$$

When comparing the two methods, consider the comparable costs. To generate two variables in the Box–Muller method, one log, one square root, and two trig functions are needed. To generate polar Marsaglia, once an acceptable  $W$  is chosen, two variables also require one log and one square root, but no trig functions. So the comparison is the cost of two trig functions versus the approximately 21% rejection rate for the  $W$  variables. With that in mind, check the output shown in Example 1.2 for the cost of a trig function. When considering this factor, the type of parallelism being used is relevant. For distributed architectures, where the various processes can operate largely independently, the rejections may not be a problem, but for GPU architectures, where `if` statements can delay the other threads, the more direct Box–Muller method may be a better choice.

## 34.3 • Problems to Solve

### 34.3.1 • Monte Carlo Integration

Probably the simplest application of parallel code is to numerically integrate a complex integral. Monte Carlo integration is the method of throwing darts to estimate the area under a curve. More specifically, suppose that  $p(x)$  is a probability distribution function, and we wish to calculate the integral

$$\int_{-\infty}^{\infty} p(x)f(x) dx. \quad (34.7)$$

To estimate the integral in (34.7), take  $N$  random samples  $x_1, \dots, x_N$  from the distribution given by  $p(x)$ ; then an estimate for the integral is given by

$$\int_{-\infty}^{\infty} p(x)f(x) dx = \langle f(x) \rangle \approx \frac{1}{N} \sum_{k=1}^N f(x_k). \quad (34.8)$$

Before using this method, it's important to understand the accuracy. The Chebyshev inequality tells us that

$$P \left\{ \left( \frac{1}{N} \sum_{k=1}^N p(x_k) - \langle p(x) \rangle \right)^2 \geq \frac{\delta^{-1}}{N} \text{var}\{p(x)\} \right\} \leq \delta, \quad (34.9)$$

where  $P$  is the probability of the contents inside the braces is true. Note that the quantity

$$\left( \frac{1}{N} \sum_{k=1}^N p(x_k) - \langle p(x) \rangle \right) \quad (34.10)$$

is the error in the approximation. So to be 99% confident that the estimate meets a certain tolerance, set  $\delta = 0.01$  and the Chebyshev inequality becomes

$$P \left\{ (\text{error})^2 \geq \frac{100}{N} \text{var}\{p(x)\} \right\} \leq 0.01. \quad (34.11)$$

Thus, to meet a specified tolerance with 99% confidence, the number of samples required to meet the tolerance is computed by solving for  $N$ :

$$N \geq \frac{100 \text{var}\{p(x)\}}{(\text{tolerance})^2}. \quad (34.12)$$

**Example 34.1.** Suppose  $p(x)$  is a uniform distribution for the interval  $[0, 1]$ , which has variance  $\text{var}\{p(x)\} = \frac{1}{12}$ , and  $x_k$  are samples from that generator; then

$$\int_{-\infty}^{\infty} p(x)f(x) dx = \int_0^1 f(x) dx \approx \frac{1}{N} \sum_{k=1}^N f(x_k). \quad (34.13)$$

So for an error tolerance of  $10^{-2}$  and a 99% confidence of meeting that tolerance, the number of samples required will be

$$N \geq \frac{100 \text{ var}\{p(x)\}}{(\text{tolerance})^2} = \frac{\frac{100}{12}}{(10^{-2})^2} \approx 83,333 \text{ samples}. \quad (34.14)$$

One thing to consider is what happens when the number of samples,  $N$ , gets very large. For example, suppose  $N = 10^{12}$  and machine epsilon is approximately  $10^{-16}$ . The sum of  $N$  numbers in double precision that are approximately  $O(1)$  is approximately  $O(10^{12})$ , so the division of the sum by  $N$  in order to compute the mean will only be accurate out to  $10^{12}/10^{16} = 10^{-4}$ . Therefore, taking  $N$  bigger to improve accuracy will not help, and if taken sufficiently large will actually make the solution worse! This is an instance where a `long double` for the sum is warranted and will provide an additional three digits of accuracy.

**Assignment.** Use Monte Carlo integration to estimate the integral

$$\int_0^{\infty} e^{-\lambda x} \cos x dx \quad (34.15)$$

for  $\lambda > 0$  by using an exponential distribution,  $p(x) = \lambda e^{-\lambda x}$  for  $x \geq 0$ , which has variance  $\text{var}\{p(x)\} = \frac{1}{\lambda^2}$ . Note that when using the  $p(x)$  distribution, the integrand has to be rewritten as

$$\int_0^{\infty} \lambda e^{-\lambda x} \frac{\cos x}{\lambda} dx = \int_0^{\infty} p(x) \frac{1}{\lambda} \cos x dx. \quad (34.16)$$

Thus, in Equation (34.13),  $f(x) = \frac{1}{\lambda} \cos x$ .

To generate random values with distribution  $p(x)$ , let  $X$  be a random value obtained from a uniform distribution on the interval  $[0, 1]$ ; then a random variable  $Y$  drawn from  $p(x)$  can be obtained via the transformation

$$Y = -\frac{1}{\lambda} \ln X. \quad (34.17)$$

Your program should take as input on the command line the number of samples  $N$ , the value of the parameter  $\lambda$ , and an optional initial seed value  $s$  so that your program named `mc` can be executed like this:

```
$ mc 1000000000 1
```

where  $N = 100000000$  is the number of samples, and  $\lambda = 1$ . You will need to check the value of `argc` to determine if  $s$  was given. Your program should print the value of the arguments given including the seed used for the computation. Verify that your seed argument is implemented correctly by taking the value printed by your program from the above test run and adding it to the argument list. For example, if your program reports using a seed of 12345, then execute your program again like this:

---

\$ mc 100000000 1 12345

and confirm you get exactly the same answer. Your program should print the approximate value of the integral and the error compared to the exact solution. Your program should also print the time required to complete the calculation.

Determine the number of samples required to achieve an error tolerance of  $10^{-4}$  with 99% confidence. Verify your approximation is within tolerance by comparing to the exact solution

$$\int_0^\infty e^{-\lambda x} \cos(x) dx = \frac{\lambda}{1 + \lambda^2}.$$

Calculate the time required to compute the solution for  $N = 10^p$  for  $p = 3, 4, \dots, 10$  and plot the time versus  $N$ . Use the plot to estimate the time required to meet the error tolerance of  $10^{-5}$  with 99.5% confidence.

### 34.3.2 • Duffing–Van der Pol Oscillator

The Duffing–Van der Pol oscillator describes the motion of a spring-mass system with a nonlinear spring coefficient. Adding a multiplicative noise forcing term, the equation becomes

$$x'' + x' - (\alpha^2 - x^2)x = \sigma x\xi$$

where  $\xi$  is white noise. Converting this to a first-order system of stochastic differential equations gives

$$dX_t = Y_t dt, \quad (34.18)$$

$$dY_t = ((\alpha^2 - X_t^2)X_t - Y_t) dt + \sigma X_t dW_t. \quad (34.19)$$

Figure 34.2 illustrates two sample paths, one for which  $\sigma = 0$ , i.e., with no noise, and one for  $\sigma = 0.5$ . Note that without noise, the path will converge asymptotically to  $(\pm\alpha, 0)$  depending on the initial condition, but with noise, the solution may bounce back and forth between the two equilibria.

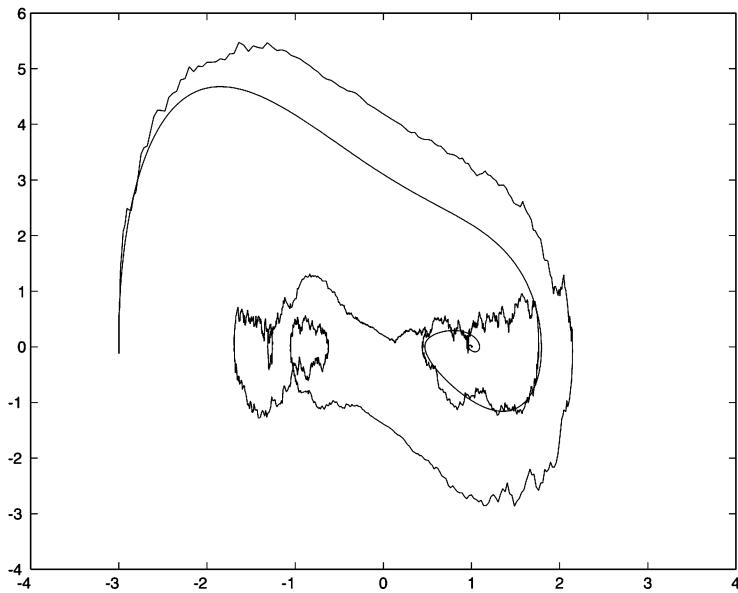
When solving such an equation, it is not done just once but done many, many times in order to get statistics about, for example, the value of  $X_T$  at some terminal time  $T$ . Figure 34.3 illustrates the histogram for the  $(X, Y)$  phase plane for the terminal time  $T = 10$ , where  $(X_0, Y_0) = (0.1 N(0; 1), 0)$  and with  $\alpha = 1$ ,  $\sigma = 0.5$ . The value  $N(0; 1)$  is a normally distributed random value with mean zero and variance one, i.e., a standard Gaussian distribution. The two stable equilibria at  $(\pm 1, 0)$  are clearly evident as expected. Simulating any single evolution may be cheap, but repeating the calculation to the point where the distribution of  $X_T$  is resolved may require a large amount of sampling.

This is a great example of an embarrassingly parallelizable algorithm. Suppose this simulation must be run  $N$  times, where  $N$  is large. The simplest strategy in this instance is to ask each of  $P$  cores to do  $N/P$  simulations and to keep their own statistics.

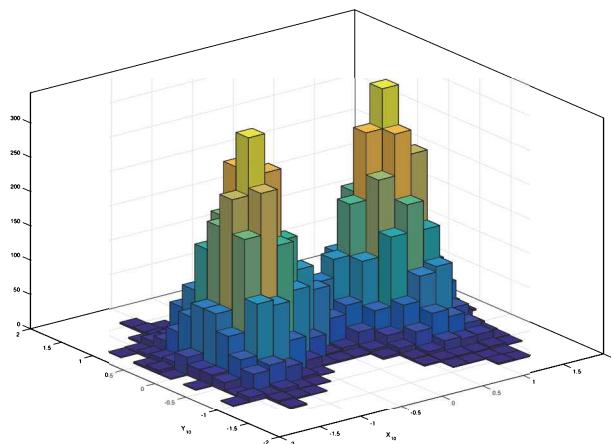
**Assignment.** Use the Euler–Maruyama method to solve the Duffing–Van der Pol oscillator equation.

Your program should take four input arguments plus an optional argument for the seed so that the program named vdp can be run with the command

\$ vdp 1 0.5 1000 100000



**Figure 34.2.** Sample phase plane plots for the cases of  $\sigma = 0$  and  $\sigma = 0.5$  for the Duffing–Van der Pol oscillator with multiplicative noise forcing, Equations (34.18), (34.19), and using  $\alpha = 1$ .



**Figure 34.3.** Example of a histogram for the Duffing–Van der Pol oscillator at terminal time  $T = 10$  with initial condition  $(X_0, Y_0) = (0.1 N(0; 1), 0)$ , and with parameter values  $\alpha = 1$ ,  $\sigma = 0.5$ . The histogram is generated by 10,000 sample runs solving Equations (34.18) and (34.19).

where  $\alpha = 1$  is the half distance between the two stable attractors,  $\sigma = 0.5$  is the strength of the noise,  $M = 1000$  is the number of time steps to use, and  $N = 100,000$  is the number of trials. The terminal time for each trial will be  $T = 10$  so that the time step size will be  $\Delta t = T/M$ . Use initial condition  $(X_0, Y_0) = (0.1 N(0; 1), 0)$  where  $N(0; 1)$  is a normally distributed variable with mean zero and variance one. Your

program should print the value of the arguments given including the initial seed used so that running the program a second time with the optional seed will give the same result. For example, if the first seed printed is 12345, then running the program a second time using

```
$ vdp 1 0.5 1000 100000 12345
```

will give the exact same results.

Define the function  $p(t)$  to be the probability of being close to an equilibrium point by

$$p(t) = P \left\{ \| (X_t, Y_t) - (-\alpha, 0) \| \leq \frac{\alpha}{2} \right\} + P \left\{ \| (X_t, Y_t) - (\alpha, 0) \| \leq \frac{\alpha}{2} \right\}.$$

To do this, for each time value  $t = k/10$ , count how many trials resulted in the point  $(X_t, Y_t)$  satisfying one of the two above inequalities. Create a double precision array  $P[101]$  where  $P[k] \approx p(k/10)$  for  $0 \leq k \leq 100$ . Your program should save the  $P$  array to a file called “Prob.out” in binary format using the `fwrite` function as described in Section 4.2.

Your program must also print to the screen the total time required to complete the calculation. Use varying values of  $M$  and  $N$  to generate a plot that illustrates how the time to compute the solutions varies with these values.

# Chapter 35

# Finite Difference Methods

There are a handful of techniques, broadly speaking, for solving PDEs. One very common strategy is the method of finite differences. It is well beyond the scope of this book to go into depth on the significant theory behind the method; the interested reader is encouraged to explore the text [27], among many others. Here a bare-bones discussion is presented that is enough to introduce the subject and program a parallel solver.

In this chapter the focus is exclusively on solving time-dependent PDEs, i.e., parabolic and hyperbolic PDEs. While some of the discussion in this chapter will be relevant to solving elliptic equations, the solution methods for elliptic equations are distinct enough to be in the following chapter.

## 35.1 • Approximating Spatial Derivatives

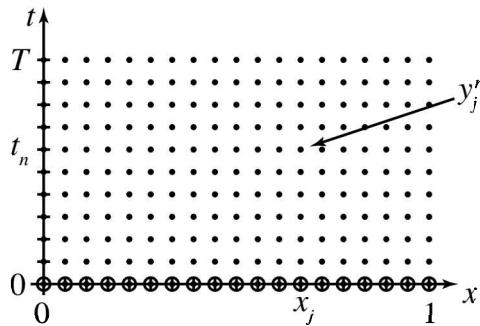
At the heart of finite difference methods is the finite difference approximation for derivatives. Suppose we have a function  $y = f(x)$  on the domain  $0 \leq x \leq 1$ , which is approximated by a set of discrete values  $y_j = f(x_j)$ , where the  $x_j$  are equally spaced with  $x_0 = 0$ ,  $x_J = 1$ , and  $x_{j+1} - x_j = \Delta x$  for all  $j = 0, \dots, J - 1$ . A quick calculation reveals that in this case,  $\Delta x = 1/J$ . The derivative  $f'(x_j)$  can be approximated from the discrete values in several ways, the simplest of which are

$$\frac{y_{j+1} - y_j}{\Delta x} = \frac{f(x_{j+1}) - f(x_j)}{x_{j+1} - x_j} = f'(x_j) + \frac{\Delta x}{2} f''(x_j) + O(\Delta x^2), \quad (35.1)$$

$$\frac{y_{j+1} - y_{j-1}}{2\Delta x} = \frac{f(x_{j+1}) - f(x_{j-1})}{x_{j+1} - x_{j-1}} = f'(x_j) + \frac{\Delta x^2}{3} f'''(x_j) + O(\Delta x^3), \quad (35.2)$$

$$\frac{y_j - y_{j-1}}{\Delta x} = \frac{f(x_j) - f(x_{j-1})}{x_j - x_{j-1}} = f'(x_j) + \frac{\Delta x}{2} f''(x_j) + O(\Delta x^2), \quad (35.3)$$

where the original function  $f(x)$  is assumed to be smooth enough to have multiple derivatives. These are called the forward difference, central difference, and backward difference formulas, respectively. Each formula approximates the first derivative of  $f(x)$  at the specified grid point,  $x_j$ , along with an estimate for the error given by the leading-order error term. The forward and backward difference formulas are called first-order accurate because their leading-order error term is first order in  $\Delta x$ , while the central difference formula is second order. It is not always the case that second order is



**Figure 35.1.** Illustration of the grid on which a PDE is solved in the domain  $0 \leq x \leq 1$ ,  $0 \leq t \leq T$ . The initial data for the differential equation means the data  $y_j^0$  in the circled grid points are given by  $y_j^0 = g(x_j)$ , where  $f(x, 0) = g(x)$  is the prescribed initial condition. The goal is to compute the remaining  $y_j^n$  for  $n > 0$ .

better due to stability considerations, but the theory behind why is beyond the scope of this book.

Higher-order derivatives can be obtained by multiple applications of the finite differences above. For example, a forward difference followed by a backward difference gives the standard second-order derivative approximation

$$\frac{y_{j+1} - 2y_j + y_{j-1}}{\Delta x^2} = f''(x_j) + \frac{\Delta x^2}{12} f'''(x_j) + O(\Delta x^4). \quad (35.4)$$

## 35.2 • Finite Difference Grid

When solving a time-dependent PDE, the equation must be solved in the domain space + time. Thus, to solve the heat equation

$$\frac{\partial f}{\partial t} = \alpha \frac{\partial^2 f}{\partial x^2}, \quad 0 \leq x \leq 1, \quad 0 \leq t \leq T, \quad (35.5)$$

the time domain must also be discretized. Therefore, the objective is to solve for the values  $y_j^n \approx f(x_j, t_n)$ , where  $x_j$  is defined as above and  $t_n = n\Delta t$ . Here,  $t_0 = 0$ ,  $t_N = T$ , and  $\Delta t = t_{n+1} - t_n$  for all  $n = 0, \dots, N - 1$ . A quick calculation shows that  $\Delta t = T/N$ . The grid on which the equation is solved numerically is illustrated in Figure 35.1. A well-posed problem will require an initial value for time  $t = 0$  such as  $f(x, 0) = g(x)$  for some given function  $g(x)$ . Thus, the points that are circled in the domain are determined by the initial value  $y_j^0 = g(x_j)$ .

### 35.2.1 • Explicit Method

In order to solve Equation (35.5) numerically, the value of  $y_j^n$  for all  $0 \leq j \leq J$ ,  $0 < n \leq N$  must be computed. We will do this by using the finite difference formulae to approximate the derivatives. The time derivative can be approximated by a forward finite difference, Equation (35.1), while the spatial derivatives can be approximated using Equation (35.4), resulting in the following finite difference equation:

$$\frac{y_j^{n+1} - y_j^n}{\Delta t} = \alpha \left( \frac{y_{j+1}^n - 2y_j^n + y_{j-1}^n}{\Delta x^2} \right). \quad (35.6)$$

Solving for  $y_j^{n+1}$  gives

$$y_j^{n+1} = y_j^n + \frac{\alpha \Delta t}{\Delta x^2} (y_{j+1}^n - 2y_j^n + y_{j-1}^n). \quad (35.7)$$

Given the data  $y_j^n$  for  $j = 0, \dots, J$ , Equation (35.7) can be used to compute  $y_j^{n+1}$  for  $j = 1, \dots, J-1$ . However, on the ends, something different must be done because, for example, when  $j = 0$ , Equation (35.7) requires the value of  $y_{-1}^n$ , which does not exist. From the theory of PDEs, for the mathematical problem to be well posed, boundary conditions at  $x = 0, 1$  are also required, which have not been specified yet.

Suppose the boundary conditions are

$$f(0, t) = h_0(t), \quad (35.8)$$

$$\frac{\partial f}{\partial x}(1, t) = h_1(t). \quad (35.9)$$

Equation (35.8) is an example of a Dirichlet-type condition, where the value of the solution  $f(0, t)$  is specified. Equation (35.9) is an example of a Neumann-type condition, where the flux, or slope, of the solution  $\frac{\partial f}{\partial x}(1, t)$  is specified. Numerically, the Dirichlet condition is easy to handle by simply setting  $y_0^n = h_0(t_n)$  for all  $n$  instead of using Equation (35.7) when  $j = 0$ . The Neumann condition at the other end requires a little more thought. There are multiple ways to handle it, but one way is to use a ghost point. Suppose  $y_{J+1}^n$  existed; then using the central difference formula, the boundary condition could be approximated by

$$\frac{y_{J+1}^n - y_{J-1}^n}{2\Delta x} = h_1(t_n). \quad (35.10)$$

Solving for  $y_{J+1}^n$  and substituting into Equation (35.7) results in a modified update equation for  $y_J^{n+1}$ :

$$\begin{aligned} y_J^{n+1} &= y_J^n + \frac{\alpha \Delta t}{\Delta x^2} (y_{J+1}^n - 2y_J^n + y_{J-1}^n) \\ &= y_J^n + \frac{\alpha \Delta t}{\Delta x^2} ((y_{J-1}^n + 2\Delta x h_1(t_n)) - 2y_J^n + y_{J-1}^n). \end{aligned} \quad (35.11)$$

The value of  $y_j^{n+1}$  can now be computed for  $j = 0, \dots, J$ . Repeating this process for  $n = 1, \dots, N$  means that  $y_j^n$  is computed throughout the domain.

The two-dimensional analogue of Equation (35.5) is

$$\frac{\partial f}{\partial t} = \alpha \nabla^2 f = \alpha \left( \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} \right) \quad (35.12)$$

and the finite difference equivalent to Equation (35.7) is

$$y_{j,k}^{n+1} = y_{j,k}^n + \alpha \Delta t \left( \frac{y_{j+1,k}^n - 2y_{j,k}^n + y_{j-1,k}^n}{\Delta x^2} + \frac{y_{j,k+1}^n - 2y_{j,k}^n + y_{j,k-1}^n}{\Delta y^2} \right). \quad (35.13)$$

### 35.2.2 • Implementation Notes

In practice, the entire grid illustrated in Figure 35.1 is not kept in memory. Instead, only enough storage to contain  $y_j^n$  and  $y_j^{n+1}$  for  $0 \leq j \leq J$  is kept. Once  $y_j^{n+1}$  is computed, the values of  $y_j^n$  are no longer needed to advance to the next time step. Only those time steps desired for viewing the results are stored in a file rather than keeping all the time steps in active memory. Otherwise, the amount of memory required could rapidly go beyond the capacity of the computer, particularly for higher-dimensional problems.

### 35.3 • Approximating Temporal Derivatives

The temporal update described above is the first-order accurate Euler's method. An example of a higher-order method is the Runge–Kutta method. Let  $\mathcal{L}$  be a spatial derivative operator on the function  $f(x, t)$  and suppose the PDE to be solved is

$$\frac{\partial f}{\partial t} = \mathcal{L}\{f(x, t)\}. \quad (35.14)$$

The fourth-order Runge–Kutta method to advance one time step is given by the following sequence of four stages. Let  $\mathbf{F}_n$  be the vector of data  $[f(x_0, t_n), \dots, f(x_{N-1}, t_n)]$ , and let  $\mathcal{L}\{\mathbf{F}_n\}$  be a numerical approximation of  $\mathcal{L}\{f(x, t_n)\}$ . Then the low-storage version of the fourth-order Runge–Kutta method is

$$\mathbf{F}_n^1 = \mathbf{F}_n + \frac{\Delta t}{4} \mathcal{L}\{\mathbf{F}_n\}, \quad (35.15)$$

$$\mathbf{F}_n^2 = \mathbf{F}_n + \frac{\Delta t}{3} \mathcal{L}\{\mathbf{F}_n^1\}, \quad (35.16)$$

$$\mathbf{F}_n^3 = \mathbf{F}_n + \frac{\Delta t}{2} \mathcal{L}\{\mathbf{F}_n^2\}, \quad (35.17)$$

$$\mathbf{F}_{n+1} = \mathbf{F}_n + \Delta t \mathcal{L}\{\mathbf{F}_n^3\}. \quad (35.18)$$

An example of how to apply this to the explicit two-dimensional heat equation method in Equation (35.13) is given by

$$y_{j,k}^{n,1} = y_{j,k}^n + \frac{\alpha \Delta t}{4} \left( \frac{y_{j+1,k}^n - 2y_{j,k}^n + y_{j-1,k}^n}{\Delta x^2} + \frac{y_{j,k+1}^n - 2y_{j,k}^n + y_{j,k-1}^n}{\Delta x^2} \right), \quad (35.19)$$

$$y_{j,k}^{n,2} = y_{j,k}^n + \frac{\alpha \Delta t}{3} \left( \frac{y_{j+1,k}^{n,1} - 2y_{j,k}^{n,1} + y_{j-1,k}^{n,1}}{\Delta x^2} + \frac{y_{j,k+1}^{n,1} - 2y_{j,k}^{n,1} + y_{j,k-1}^{n,1}}{\Delta y^2} \right), \quad (35.20)$$

$$y_{j,k}^{n,3} = y_{j,k}^n + \frac{\alpha \Delta t}{2} \left( \frac{y_{j+1,k}^{n,2} - 2y_{j,k}^{n,2} + y_{j-1,k}^{n,2}}{\Delta x^2} + \frac{y_{j,k+1}^{n,2} - 2y_{j,k}^{n,2} + y_{j,k-1}^{n,2}}{\Delta y^2} \right), \quad (35.21)$$

$$y_{j,k}^{n+1} = y_{j,k}^n + \alpha \Delta t \left( \frac{y_{j+1,k}^{n,3} - 2y_{j,k}^{n,3} + y_{j-1,k}^{n,3}}{\Delta x^2} + \frac{y_{j,k+1}^{n,3} - 2y_{j,k}^{n,3} + y_{j,k-1}^{n,3}}{\Delta y^2} \right). \quad (35.22)$$

#### 35.3.1 • Implicit Methods

The method described above to solve the heat equation is called an *explicit method* because the value of  $y_j^{n+1}$  is computed explicitly from prior known values. That method may not be the best choice, depending on the value of  $\alpha$  and the length of time  $T$  to be computed due to issues related to stability of the numerical method. One way to get around that issue is to use an implicit method.

Recall the forward finite difference approximation of the time derivative in Equation (35.7). Suppose the backward finite difference approximation is used instead; then the equation would be

$$y_j^n = y_j^{n-1} + \frac{\alpha \Delta t}{\Delta x^2} (y_{j+1}^n - 2y_j^n + y_{j-1}^n). \quad (35.23)$$

The difficulty here is that solving for  $y_j^n$  by itself is complicated because  $y_{j+1}^n$ ,  $y_{j-1}^n$  also appear in this equation. Bringing all the  $n$ th terms to the left side results in the

equation

$$-\frac{\alpha \Delta t}{\Delta x^2} y_{j+1}^n + \left(1 + 2 \frac{\alpha \Delta t}{\Delta x^2}\right) y_j^n - \frac{\alpha \Delta t}{\Delta x^2} y_{j-1}^n = y_j^{n-1}. \quad (35.24)$$

Thus, advancing from time  $t_{n-1}$  to  $t_n$  requires the solution of a system of linear equations:

$$\begin{aligned} y_0^n &= h_0(t_n), \\ -\frac{\alpha \Delta t}{\Delta x^2} y_2^n + \left(1 + 2 \frac{\alpha \Delta t}{\Delta x^2}\right) y_1^n - \frac{\alpha \Delta t}{\Delta x^2} y_0^n &= y_1^{n-1}, \\ &\vdots \end{aligned} \quad (35.25)$$

$$-\frac{\alpha \Delta t}{\Delta x^2} y_{j+1}^n + \left(1 + 2 \frac{\alpha \Delta t}{\Delta x^2}\right) y_j^n - \frac{\alpha \Delta t}{\Delta x^2} y_{j-1}^n = y_j^{n-1}, \quad (35.26)$$

$$\begin{aligned} &\vdots \\ -\frac{\alpha \Delta t}{\Delta x^2} y_J^n + \left(1 + 2 \frac{\alpha \Delta t}{\Delta x^2}\right) y_{J-1}^n - \frac{\alpha \Delta t}{\Delta x^2} y_{J-2}^n &= y_{J-1}^{n-1}, \\ \left(1 + \frac{2\alpha \Delta t}{\Delta x^2}\right) y_J^n - \frac{2\alpha \Delta t}{\Delta x^2} y_{J-1}^n &= y_J^{n-1} - \frac{2\alpha \Delta t}{\Delta x^2} h_1(t_n), \end{aligned} \quad (35.27)$$

where the boundary conditions described earlier are incorporated. This results in a tridiagonal system of equations that looks like

$$\begin{bmatrix} 1 & 0 & & & & \\ -\lambda & 1 + 2\lambda & -\lambda & 0 & & 0 \\ 0 & -\lambda & 1 + 2\lambda & -\lambda & & \\ & \ddots & \ddots & \ddots & & \\ & & & & & \\ 0 & -\lambda & 1 + 2\lambda & -\lambda & & \\ & 0 & -2\lambda & 1 + 2\lambda & & \end{bmatrix} \begin{bmatrix} y_0^n \\ y_1^n \\ y_2^n \\ \vdots \\ y_{J-1}^n \\ y_J^n \end{bmatrix} = \begin{bmatrix} h_0(t_n) \\ y_1^{n-1} \\ y_2^{n-1} \\ \vdots \\ y_{J-1}^{n-1} \\ y_J^{n-1} - 2\lambda h_1(t_n) \end{bmatrix}, \quad (35.28)$$

where  $\lambda = \alpha \Delta t / \Delta x^2$ . This tridiagonal system is easily solved using a linear algebra library such as LAPACK.

To simplify the discussion for the next method, note that the explicit and implicit methods can be written as simpler matrix equations of the form

$$\begin{aligned} \mathbf{Y}^{n+1} &= (\mathbf{I} + \Delta t \mathbf{D}) \mathbf{Y}^n, \\ (\mathbf{I} - \Delta t \mathbf{D}) \mathbf{Y}^{n+1} &= \mathbf{Y}^n, \end{aligned}$$

where  $\mathbf{Y}^n = [y_0^n, \dots, y_J^n]^T$  and where

$$\mathbf{D} = \begin{bmatrix} * & * & & & & \\ \frac{\alpha}{\Delta x^2} & -\frac{2\alpha}{\Delta x^2} & \frac{\alpha}{\Delta x^2} & 0 & & 0 \\ 0 & \frac{\alpha}{\Delta x^2} & -\frac{2\alpha}{\Delta x^2} & \frac{\alpha}{\Delta x^2} & & \\ & \ddots & \ddots & \ddots & & \\ & & & & & \\ 0 & \frac{\alpha}{\Delta x^2} & -\frac{2\alpha}{\Delta x^2} & \frac{\alpha}{\Delta x^2} & * & * \end{bmatrix}. \quad (35.29)$$

The \* in the matrix indicates that the boundary conditions must be accounted for when constructing the first and last rows of the matrix.

### 35.3.2 • Alternating Directions Implicit Method

One popular method for solving the heat equation using an implicit method in higher spatial dimensions is the alternating directions implicit (ADI) method [28]. In two dimensions, the heat equation with a reaction term is given by

$$\frac{\partial f}{\partial t} = \alpha \nabla^2 f + \rho = \alpha \left( \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} \right) + \rho, \quad (35.30)$$

where  $\rho$  is a possibly nonlinear reaction term that can depend on  $x$ ,  $y$ , and  $f(x, y)$ .

To solve this equation numerically, remember that the numerical solution will now be solved on a two-dimensional grid. This means the method will solve for the grid points  $u_{j,k}^n$  which are an approximation to the solution of Equation (35.30):

$$u_{j,k}^n \approx f(x_j, y_k, t_n).$$

An explicit method as discussed in Section 35.2.1 could be used, but it would be slow and subject to time step restrictions. A better choice is to use the ADI method. The method is easily described in terms of the one-dimensional operators given above.

For this algorithm, define

$$\mathbf{U}_{*,k}^n = \begin{bmatrix} u_{0,k}^n \\ u_{1,k}^n \\ u_{2,k}^n \\ \vdots \\ u_{J-1,k}^n \\ u_{J,k}^n \end{bmatrix}, \quad \mathbf{U}_{j,*}^n = \begin{bmatrix} u_{j,0}^n \\ u_{j,1}^n \\ u_{j,2}^n \\ \vdots \\ u_{j,K-1}^n \\ u_{j,K}^n \end{bmatrix}. \quad (35.31)$$

The ADI method is then as follows:

1. Apply the one-dimensional explicit operator in the  $x$  direction for a time step of  $\Delta t/2$  and for each row  $0 \leq k \leq K$ :

$$\mathbf{U}_{*,k}^{n+1/4} = \left( \mathbf{I} + \frac{\Delta t}{2} \mathbf{D}_x \right) \mathbf{U}_{*,k}^n + \frac{\Delta t}{2} \rho_{*,k}^n. \quad (35.32)$$

2. Apply the one-dimensional implicit operator in the  $y$  direction for a time step of  $\Delta t/2$  and for each column  $0 \leq j \leq J$ :

$$\left( \mathbf{I} - \frac{\Delta t}{2} \mathbf{D}_y \right) \mathbf{U}_{j,*}^{n+1/2} = \mathbf{U}_{j,*}^{n+1/4}. \quad (35.33)$$

3. Apply the one-dimensional explicit operator in the  $y$  direction for a time step of  $\Delta t/2$  and for each column  $0 \leq j \leq J$ :

$$\mathbf{U}_{j,*}^{n+3/4} = \left( \mathbf{I} + \frac{\Delta t}{2} \mathbf{D}_y \right) \mathbf{U}_{j,*}^{n+1/2} + \frac{\Delta t}{2} \rho_{j,*}^{n+1/2}. \quad (35.34)$$

4. Apply the one-dimensional implicit operator in the  $x$  direction for a time step of  $\Delta t/2$  and for each row  $0 \leq k \leq K$ :

$$\left( \mathbf{I} - \frac{\Delta t}{2} \mathbf{D}_x \right) \mathbf{U}_{*,k}^{n+1} = \mathbf{U}_{*,k}^{n+3/4}. \quad (35.35)$$

Here,  $\rho_{*,k}^n$  is the reaction term evaluated using the data from  $\mathbf{U}_{*,k}^n$  and similarly for  $\rho_{j,*}^{n+1/2}$ . The boundary conditions are also incorporated in each step as described for each of the methods.

This method is particularly useful for parallel implementation because each step can be done in parallel. For example, in Step 1, the explicit method is applied to each of the  $K$  rows of the grid. There is no dependency of one row upon another, so the  $K$  one-dimensional updates can be done simultaneously. More efficiency is gained by the fact that each of the matrices for the implicit steps are tridiagonal, which can be efficiently solved.

In three dimensions, the algorithm is similar but a bit more complicated in order to remain consistent with the original differential equation. Here it's condensed into three steps:

$$\left( \mathbf{I} - \frac{\Delta t}{2} \mathbf{D}_x \right) \mathbf{U}^{n+1/3} = \left( \mathbf{I} + \frac{\Delta t}{2} \mathbf{D}_x + \Delta t \mathbf{D}_y + \Delta t \mathbf{D}_z \right) \mathbf{U}^n + \rho^n, \quad (35.36)$$

$$\left( \mathbf{I} - \frac{\Delta t}{2} \mathbf{D}_y \right) \mathbf{U}^{n+2/3} = \mathbf{U}^{n+1/3} - \frac{\Delta t}{2} \mathbf{D}_y \mathbf{U}^n + \rho^{n+1/3}, \quad (35.37)$$

$$\left( \mathbf{I} - \frac{\Delta t}{2} \mathbf{D}_z \right) \mathbf{U}^{n+1} = \mathbf{U}^{n+2/3} - \frac{\Delta t}{2} \mathbf{D}_z \mathbf{U}^n + \rho^{n+2/3}. \quad (35.38)$$

Note that the spatial indexing is discarded as was used in the previous description. The derivative operator  $\mathbf{D}_x$  is applied in the one-dimensional  $x$  direction to  $\mathbf{U}_{*,j,k}$  for each  $0 \leq j \leq J$ ,  $0 \leq k \leq K$ , and likewise for the other derivative operators. In this case, to parallelize it, one would compute each of the operations on the right-hand side of Equation (35.36) separately and then combine them before solving the collection of  $JK$  tridiagonal solves to invert the matrix on the left. Likewise for Equations (35.37), (35.38).

## 35.4 • Problems to Solve

### 35.4.1 • Brusselator Model

The Brusselator is a model for an autocatalytic reaction, which can be written as a reaction-diffusion equation for two chemical species,  $U$  and  $V$ , which have different diffusion rates. The system of equations is

$$\frac{\partial U}{\partial t} = D_U \nabla^2 U + A + U^2 V - (B + 1)U, \quad (35.39)$$

$$\frac{\partial V}{\partial t} = D_V \nabla^2 V + BU - U^2 V, \quad (35.40)$$

$$U(\mathbf{x}, 0) = A + \sigma, \quad (35.41)$$

$$V(\mathbf{x}, 0) = B + \sigma, \quad (35.42)$$

$$\left. \frac{\partial U}{\partial t} \right|_{\Gamma} = 0, \quad (35.43)$$

$$\left. \frac{\partial V}{\partial t} \right|_{\Gamma} = 0, \quad (35.44)$$

where  $A$  and  $B$  are constants that determine the fixed point of the reaction ( $U = A$ ,  $V = B/A$ ), and  $D_U$ ,  $D_V$  are the diffusion rates of the two species  $U$ ,  $V$ , respectively. The term  $\sigma$  in the initial conditions for  $U$  and  $V$  should be a random value at each grid point drawn from a uniform distribution on the interval  $-1 \leq \sigma < 1$ . The boundary conditions for  $U$ ,  $V$  are such that the values should remain fixed at their initial random value for all time.

**Assignment.** (Explicit version) Use the Runge–Kutta method with full two-dimensional finite differences by modifying Equations (35.19)–(35.22). (Implicit version) Use the ADI method by modifying Equations (35.32)–(35.35).

Solve the Brusselator model on a unit box  $[0, 1] \times [0, 1]$  with parameter values  $A = 1$ ,  $B = 3$ ,  $D_U = 5 \times 10^{-5}$ ,  $D_V = 5 \times 10^{-6}$ . Solve until the terminal time  $T = 1000$ .

Your program should take six arguments with a seventh optional seed value so that your program named `bruss` can be executed like this:

```
$ bruss 128 5.0e-5 5.0e-6 1 3 10000
```

where arguments in order are the number of grid points in each dimension,  $N = 128$ , the double precision diffusion coefficients,  $D_u = 5 \times 10^{-5}$ ,  $D_v = 5 \times 10^{-6}$ , the double precision coefficients,  $A = 1$ ,  $B = 3$ , and the number of time steps,  $M = 10,000$ . If a seventh argument is specified, then it is a long integer seed value,  $s$ , for generating the initial random values. You will need to check the value of `argc` to determine if  $s$  was given. Your program must print the arguments given including the value of the seed, and you should verify that your seed argument is implemented correctly by taking the value printed by your program from the above test run and adding it to the argument list. For example, if your program reports using a seed of 12345, then execute your program again like this:

```
$ bruss 128 5.0e-5 5.0e-6 1 3 10000 12345
```

and verify that you get the exact same results.

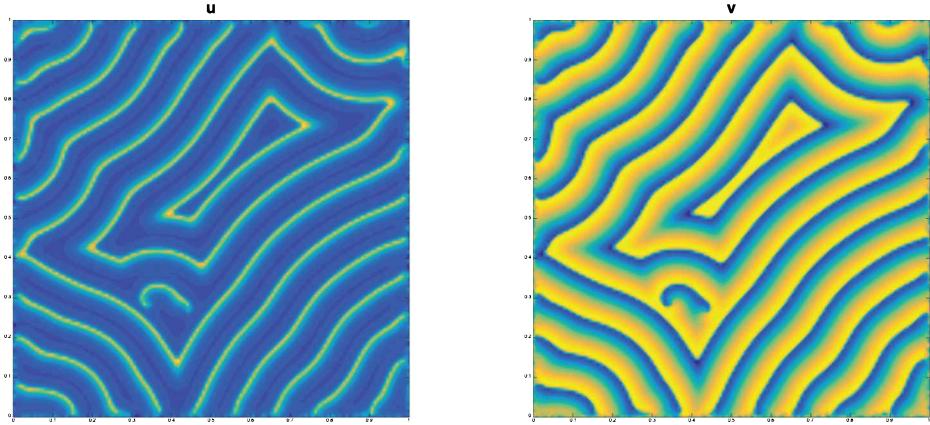
Your grid will be  $N \times N$  so that  $x_0 = 0$  and  $x_{N-1} = 1$ . Your program should output the grid values for  $U$  and  $V$  into two files called “BrussU.out” and “BrussV.out” that contain the data at the time points  $t = 100k$  for  $k = 0, 1, \dots, 10$ . The files should contain *only* the values of  $U$  and  $V$  on the grid and nothing else.

Your program must also print to the screen the total time required to complete the calculation. Vary the values of  $N$  to generate a plot that illustrates the computational cost as a function of  $N$ .

Plot the isocontours of  $U$ ,  $V$  and over time and a pattern of reaction waves should emerge that will be roughly equidistant such as shown in Figure 35.2.

### 35.4.2 • Linearized Euler Equations

The Euler equations are a simplification of the Navier–Stokes equations for inviscid adiabatic fluid flow. The linearized version of the equations, which can be used to study



**Figure 35.2.** Sample contour plot for the Brusselator model. The plot of  $U$  is on the left and the plot of  $V$  is on the right at the terminal time  $T = 1000$ . The pattern of contours is determined by the values for  $U$  and  $V$  on the domain boundary.

small disturbances in air, for example, are given by

$$\frac{\partial \rho}{\partial t} = -\mathbf{u}_0 \cdot \nabla \rho - \rho_0 \nabla \cdot \mathbf{u}, \quad (35.45)$$

$$\frac{\partial \mathbf{u}}{\partial t} = -\mathbf{u}_0 \cdot \mathbf{u} - \frac{1}{\rho_0} \nabla p, \quad (35.46)$$

$$\frac{\partial p}{\partial t} = -\mathbf{u}_0 \cdot \nabla p + \gamma p_0 \nabla \cdot \mathbf{u}, \quad (35.47)$$

where  $\rho$  is the fluid density,  $\mathbf{u}$  is the fluid velocity, and  $p$  is the fluid pressure. The constants  $\rho_0$ ,  $\mathbf{u}_0$ , and  $p_0$  are the equilibrium values around which the system is linearized. The constant  $\gamma$  is the gas constant. In the case of a small perturbation from a flow at rest, take  $\rho_0 = 1$ ,  $\mathbf{u}_0 = \mathbf{0}$ , and  $p_0 = 1$  so that the equations in (35.45)–(35.47) become

$$\frac{\partial \rho}{\partial t} = -\nabla \cdot \mathbf{u}, \quad (35.48)$$

$$\frac{\partial \mathbf{u}}{\partial t} = -\nabla p, \quad (35.49)$$

$$\frac{\partial p}{\partial t} = -\gamma \nabla \cdot \mathbf{u}. \quad (35.50)$$

For simplicity, assume this problem is periodic on a domain that has dimensions  $[-1, 1]$  in all directions.

The explicit implementation of this problem using the Runge–Kutta method and finite difference spatial derivatives is reasonably straightforward. The primary difference is that for the linearized Euler equations, the derivatives are all first order, so the Laplacian finite difference approximation for  $\nabla^2 f$  approximated in Equations (35.19)–(35.22) is replaced with the  $x$  and  $y$  central finite difference approximations

$$\frac{\partial f}{\partial x} \approx \frac{f_{i+1,j} - f_{i-1,j}}{2\Delta x}, \quad \frac{\partial f}{\partial y} \approx \frac{f_{i,j+1} - f_{i,j-1}}{2\Delta y}. \quad (35.51)$$

Thus, the method becomes

### Stage 1

$$\rho_{j,k}^{n,1} = \rho_{j,k}^n - \frac{\Delta t}{4} \left( \frac{u_{j+1,k}^n - u_{j-1,k}^n}{2\Delta x} - \frac{v_{j,k+1}^n - v_{j,k-1}^n}{2\Delta y} \right), \quad (35.52)$$

$$u_{j,k}^{n,1} = u_{j,k}^n - \frac{\Delta t}{4} \frac{p_{j+1,k}^n - p_{j-1,k}^n}{2\Delta x}, \quad (35.53)$$

$$v_{j,k}^{n,1} = v_{j,k}^n - \frac{\Delta t}{4} \frac{p_{j,k+1}^n - p_{j,k-1}^n}{2\Delta y}, \quad (35.54)$$

$$p_{j,k}^{n,1} = p_{j,k}^n - \frac{\gamma \Delta t}{4} \left( \frac{u_{j+1,k}^n - u_{j-1,k}^n}{2\Delta x} - \frac{v_{j,k+1}^n - v_{j,k-1}^n}{2\Delta y} \right), \quad (35.55)$$

### Stage 2

$$\rho_{j,k}^{n,2} = \rho_{j,k}^n - \frac{\Delta t}{4} \left( \frac{u_{j+1,k}^{n,1} - u_{j-1,k}^{n,1}}{2\Delta x} - \frac{v_{j,k+1}^{n,1} - v_{j,k-1}^{n,1}}{2\Delta y} \right), \quad (35.56)$$

$$u_{j,k}^{n,2} = u_{j,k}^n - \frac{\Delta t}{4} \frac{p_{j+1,k}^{n,1} - p_{j-1,k}^{n,1}}{2\Delta x}, \quad (35.57)$$

$$v_{j,k}^{n,2} = v_{j,k}^n - \frac{\Delta t}{4} \frac{p_{j,k+1}^{n,1} - p_{j,k-1}^{n,1}}{2\Delta y}, \quad (35.58)$$

$$p_{j,k}^{n,2} = p_{j,k}^n - \frac{\gamma \Delta t}{4} \left( \frac{u_{j+1,k}^{n,1} - u_{j-1,k}^{n,1}}{2\Delta x} - \frac{v_{j,k+1}^{n,1} - v_{j,k-1}^{n,1}}{2\Delta y} \right), \quad (35.59)$$

### Stage 3

$$\rho_{j,k}^{n,3} = \rho_{j,k}^n - \frac{\Delta t}{4} \left( \frac{u_{j+1,k}^{n,2} - u_{j-1,k}^{n,2}}{2\Delta x} - \frac{v_{j,k+1}^{n,2} - v_{j,k-1}^{n,2}}{2\Delta y} \right), \quad (35.60)$$

$$u_{j,k}^{n,3} = u_{j,k}^n - \frac{\Delta t}{4} \frac{p_{j+1,k}^{n,2} - p_{j-1,k}^{n,2}}{2\Delta x}, \quad (35.61)$$

$$v_{j,k}^{n,3} = v_{j,k}^n - \frac{\Delta t}{4} \frac{p_{j,k+1}^{n,2} - p_{j,k-1}^{n,2}}{2\Delta y}, \quad (35.62)$$

$$p_{j,k}^{n,3} = p_{j,k}^n - \frac{\gamma \Delta t}{4} \left( \frac{u_{j+1,k}^{n,2} - u_{j-1,k}^{n,2}}{2\Delta x} - \frac{v_{j,k+1}^{n,2} - v_{j,k-1}^{n,2}}{2\Delta y} \right), \quad (35.63)$$

### Stage 4

$$\rho_{j,k}^{n+1} = \rho_{j,k}^n - \frac{\Delta t}{4} \left( \frac{u_{j+1,k}^{n,3} - u_{j-1,k}^{n,3}}{2\Delta x} - \frac{v_{j,k+1}^{n,3} - v_{j,k-1}^{n,3}}{2\Delta y} \right), \quad (35.64)$$

$$u_{j,k}^{n+1} = u_{j,k}^n - \frac{\Delta t}{4} \frac{p_{j+1,k}^{n,3} - p_{j-1,k}^{n,3}}{2\Delta x}, \quad (35.65)$$

$$v_{j,k}^{n+1} = v_{j,k}^n - \frac{\Delta t}{4} \frac{p_{j,k+1}^{n,3} - p_{j,k-1}^{n,3}}{2\Delta y}, \quad (35.66)$$

$$p_{j,k}^{n+1} = p_{j,k}^n - \frac{\gamma \Delta t}{4} \left( \frac{u_{j+1,k}^{n,3} - u_{j-1,k}^{n,3}}{2\Delta x} - \frac{v_{j,k+1}^{n,3} - v_{j,k-1}^{n,3}}{2\Delta y} \right). \quad (35.67)$$

This problem can also be solved using the ADI method where the spatial derivative operator is replaced with a central difference operator. Let  $\mathbf{I}$  be the identity matrix and

$\mathbf{0}$  be a matrix of all zeros, and define the matrix

$$\mathbf{D} = \begin{bmatrix} 0 & \frac{\Delta t}{4\Delta x} & 0 & \cdots & \cdots & 0 & -\frac{\Delta t}{4\Delta x} \\ -\frac{\Delta t}{4\Delta x} & 0 & \frac{\Delta t}{4\Delta x} & 0 & \cdots & \cdots & 0 \\ 0 & -\frac{\Delta t}{4\Delta x} & 0 & \frac{\Delta t}{4\Delta x} & 0 & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & -\frac{\Delta t}{4\Delta x} & 0 & \frac{\Delta t}{4\Delta x} & 0 \\ 0 & \cdots & \cdots & 0 & -\frac{\Delta t}{4\Delta x} & 0 & \frac{\Delta t}{4\Delta x} \\ \frac{\Delta t}{4\Delta x} & 0 & \cdots & \cdots & 0 & -\frac{\Delta t}{4\Delta x} & 0 \end{bmatrix}. \quad (35.68)$$

In two dimensions, rewrite the system as the scalar differential equations in matrix form, where  $\mathbf{u} = (u, v)$ , to get

$$\begin{bmatrix} \rho \\ u \\ v \\ p \end{bmatrix}_t = - \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & \gamma & 0 & 0 \end{bmatrix} \begin{bmatrix} \rho \\ u \\ v \\ p \end{bmatrix}_x - \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & \gamma & 0 \end{bmatrix} \begin{bmatrix} \rho \\ u \\ v \\ p \end{bmatrix}_y. \quad (35.69)$$

The ADI method can then be implemented in the following four steps:

1. Explicit in  $x$ :

$$\begin{bmatrix} \rho_{*,k}^{n+1/4} \\ \mathbf{u}_{*,k}^{n+1/4} \\ \mathbf{v}_{*,k}^{n+1/4} \\ \mathbf{p}_{*,k}^{n+1/4} \end{bmatrix} = \begin{bmatrix} \mathbf{I} & -\mathbf{D} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{I} & \mathbf{0} & -\mathbf{D} \\ \mathbf{0} & \mathbf{0} & \mathbf{I} & \mathbf{0} \\ \mathbf{0} & -\gamma\mathbf{D} & \mathbf{0} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \rho_{*,k}^n \\ \mathbf{u}_{*,k}^n \\ \mathbf{v}_{*,k}^n \\ \mathbf{p}_{*,k}^n \end{bmatrix}. \quad (35.70)$$

2. Implicit in  $y$ :

$$\begin{bmatrix} \mathbf{I} & \mathbf{0} & \mathbf{D} & \mathbf{0} \\ \mathbf{0} & \mathbf{I} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{I} & \mathbf{D} \\ \mathbf{0} & \mathbf{0} & \gamma\mathbf{D} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \rho_{j,*}^{n+1/2} \\ \mathbf{u}_{j,*}^{n+1/2} \\ \mathbf{v}_{j,*}^{n+1/2} \\ \mathbf{p}_{j,*}^{n+1/2} \end{bmatrix} = \begin{bmatrix} \rho_{j,*}^{n+1/4} \\ \mathbf{u}_{j,*}^{n+1/4} \\ \mathbf{v}_{j,*}^{n+1/4} \\ \mathbf{p}_{j,*}^{n+1/4} \end{bmatrix}. \quad (35.71)$$

3. Explicit in  $y$ :

$$\begin{bmatrix} \rho_{j,*}^{n+3/4} \\ \mathbf{u}_{j,*}^{n+3/4} \\ \mathbf{v}_{j,*}^{n+3/4} \\ \mathbf{p}_{j,*}^{n+3/4} \end{bmatrix} = \begin{bmatrix} \mathbf{I} & \mathbf{0} & -\mathbf{D} & \mathbf{0} \\ \mathbf{0} & \mathbf{I} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{I} & -\mathbf{D} \\ \mathbf{0} & \mathbf{0} & -\gamma\mathbf{D} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \rho_{j,*}^{n+1/2} \\ \mathbf{u}_{j,*}^{n+1/2} \\ \mathbf{v}_{j,*}^{n+1/2} \\ \mathbf{p}_{j,*}^{n+1/2} \end{bmatrix}. \quad (35.72)$$

4. Implicit in  $x$ :

$$\begin{bmatrix} \mathbf{I} & \mathbf{D} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{I} & \mathbf{0} & \mathbf{D} \\ \mathbf{0} & \mathbf{0} & \mathbf{I} & \mathbf{0} \\ \mathbf{0} & \gamma\mathbf{D} & \mathbf{0} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \rho_{*,k}^{n+1} \\ \mathbf{u}_{*,k}^{n+1} \\ \mathbf{v}_{*,k}^{n+1} \\ \mathbf{p}_{*,k}^{n+1} \end{bmatrix} = \begin{bmatrix} \rho_{*,k}^{n+3/4} \\ \mathbf{u}_{*,k}^{n+3/4} \\ \mathbf{v}_{*,k}^{n+3/4} \\ \mathbf{p}_{*,k}^{n+3/4} \end{bmatrix}. \quad (35.73)$$

Note that in the above steps, the vectors are written as a single column vector for purposes of the computation. Thus, the vector  $[\rho_{*,k}^n, \mathbf{u}_{*,k}^n, \mathbf{v}_{*,k}^n, \mathbf{p}_{*,k}^n]^T$  is actually the vector:

$$[\rho_{0,k}^n, \rho_{1,k}^n, \dots, \rho_{N-1,k}^n, u_{0,k}^n, u_{1,k}^n, \dots, u_{N-1,k}^n, v_{0,k}^n, v_{1,k}^n, \dots, v_{N-1,k}^n, \\ p_{0,k}^n, p_{1,k}^n, \dots, p_{N-1,k}^n]^T$$

Note that the explicit steps in Equations (35.70) and (35.72) should *not* be implemented as a matrix vector multiplication. The algorithm is presented in this way here for compactness, but the actual implementation should be done with a simple `for` loop. For example, the update for  $\rho_{j,k}^{n+1/4}$  in Equation (35.70) should simply implement the equation

$$\rho_{j,k}^{n+1/4} = \rho_{j,k}^n - \frac{\Delta t}{4\Delta x} (u_{j+1,k}^n - u_{j-1,k}^n)$$

for  $j = 0, \dots, N - 1$  and noting that because of the assumed periodic boundary conditions, for purposes of this formula  $u_{-1,k}^n \equiv u_{N-1,k}^n$  and  $u_{N,k}^n \equiv u_{0,k}^n$ . The updates for  $u_{j,k}^{n+1/4}$ ,  $v_{j,k}^{n+1/4}$ , and  $p_{j,k}^{n+1/4}$  are similar.

For the implicit steps in Equations (35.71) and (35.73), note that the arrangement of the data requires some careful attention. Ideally, the right-hand side is arranged to be a  $4N \times N$  matrix so that each column corresponds to a given row or column of data depending on the equation being solved. Thus, for Equation (35.71), the right-hand-side matrix would have the form

$$\begin{bmatrix} \rho_{0,0} & \cdots & \rho_{N-1,0} \\ \vdots & & \vdots \\ \rho_{0,N-1} & \cdots & \rho_{N-1,N-1} \\ u_{0,0} & \cdots & u_{N-1,0} \\ \vdots & & \vdots \\ u_{0,N-1} & \cdots & u_{N-1,N-1} \\ v_{0,0} & \cdots & v_{N-1,0} \\ \vdots & & \vdots \\ v_{0,N-1} & \cdots & v_{N-1,N-1} \\ p_{0,0} & \cdots & p_{N-1,0} \\ \vdots & & \vdots \\ p_{0,N-1} & \cdots & p_{N-1,N-1} \end{bmatrix}$$

so that each column has a fixed value for the  $j$  index, whereas for Equation (35.73) the arrangement would have each block in the matrix above transposed so that each column

has a fixed value for the  $k$  index:

$$\begin{bmatrix} \rho_{0,0} & \cdots & \rho_{0,N-1} \\ \vdots & & \vdots \\ \rho_{N-1,0} & \cdots & \rho_{N-1,N-1} \\ u_{0,0} & \cdots & u_{0,N-1} \\ \vdots & & \vdots \\ u_{N-1,0} & \cdots & u_{N-1,N-1} \\ v_{0,0} & \cdots & v_{0,N-1} \\ \vdots & & \vdots \\ v_{N-1,0} & \cdots & v_{N-1,N-1} \\ p_{0,0} & \cdots & p_{0,N-1} \\ \vdots & & \vdots \\ p_{N-1,0} & \cdots & p_{N-1,N-1} \end{bmatrix}.$$

By arranging the data in this way, the calls to the linear solver, e.g., `dgetrs_`, can be done in one call with  $N$  vectors on the right hand side.

**Assignment.** (Explicit version) Use the Runge–Kutta method with full two-dimensional finite differences by modifying Equations (35.19)–(35.22). (Implicit version) Use the ADI method by modifying Equations (35.32)–(35.35).

Solve the linearized Euler equations for the case of an initial perturbation in pressure where the domain is the square  $[-1, 1] \times [-1, 1]$  and with initial conditions

$$\rho(x, y, 0) = \frac{2}{\gamma} e^{-100(x^2+y^2)}, \quad (35.74)$$

$$\mathbf{u}(x, y, 0) = 0, \quad (35.75)$$

$$p(x, y, 0) = 2e^{-100(x^2+y^2)}, \quad (35.76)$$

where  $\gamma = 1.4$ . Solve until the terminal time  $T = 2$ .

Your program should take two arguments so that your program named `euler` can be executed like this:

```
$ euler 128 2000
```

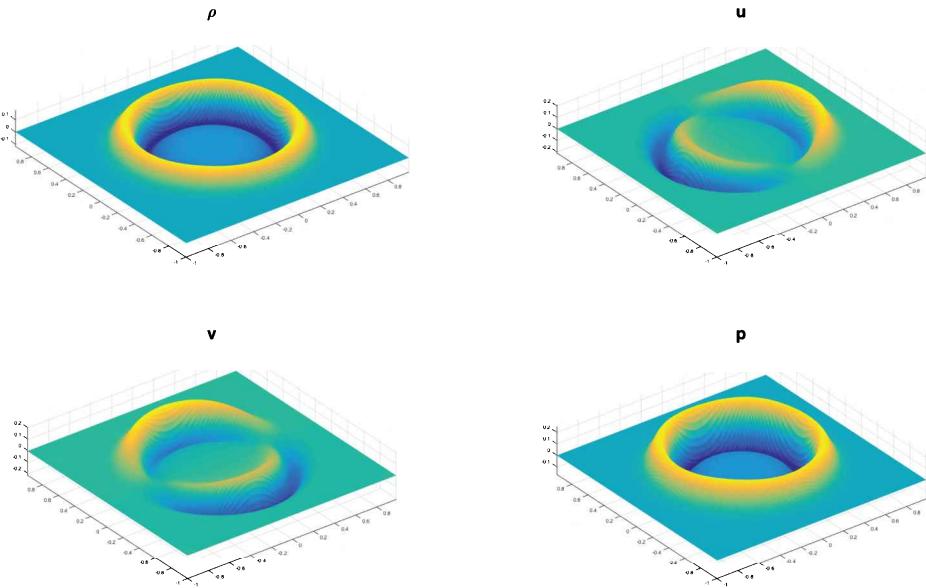
where the arguments are the dimensions of the grid,  $N = 128$ , and the number of time steps,  $M = 2,000$ . Your program should print the values of the arguments to verify that they are correct.

Because the grid is periodic, your grid spacing should be  $\Delta x = \frac{2}{N}$  so that  $x_0 = -1$ , and  $x_{N-1} = 1 - \Delta x$ .

Your program should output the grid values for  $\rho$ ,  $u$ ,  $v$ , and  $p$  into four files called “EulerR.out”, “EulerU.out”, “EulerV.out”, and “EulerP.out”, respectively, that contain the data at the time points  $t = 0.2k$  for  $k = 0, 1, \dots, 10$ . The files should *only* contain the values of  $\rho$ ,  $u$ ,  $v$ , and  $p$  on the grid and nothing else.

Your program must also print to the screen the total time required to complete the calculation. Use varying values of  $N$  to generate a plot that illustrates the scaling of the computational cost versus  $N$ .

Figure 35.3 shows how the solution should look at time  $t = 0.5$ .



**Figure 35.3.** Illustration of the solution for the linearized Euler equations at the time  $t = 0.5$  with  $\rho$  in the top left,  $u$  in the top right,  $v$  in the bottom left, and  $p$  in the bottom right. The initial central disturbance expands outward as a ring.

## Chapter 36

# Iterative Solution of Elliptic Equations

In the previous chapter, finite difference grids were introduced for solving time-dependent PDEs. In this chapter the description is extended to look at a method for solving elliptic equations. It should be noted that there are many different methods for solving elliptic equations that are very effective, for example, direct matrix inversion and multigrid methods, to name two. However, in this chapter a method is presented that is simpler and more straightforward to implement in parallel.

A second-order elliptic equation in two dimensions is an equation of the form

$$\frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} = g(\mathbf{x}), \quad (36.1)$$

where  $g(\mathbf{x})$  is some given function. Elliptic equations of this form require a boundary condition on the entire boundary.

Suppose a rectangular domain of dimensions  $0 \leq x \leq 1$ ,  $0 \leq y \leq 1$ , is discretized with spatial step sizes  $\Delta x$  and  $\Delta y$ , respectively. Following the discretization techniques from the previous chapter, Equation (36.1) can be approximated by the equation

$$\frac{u_{j+1,k} - 2u_{j,k} + u_{j-1,k}}{\Delta x^2} + \frac{u_{j,k+1} - 2u_{j,k} + u_{j,k-1}}{\Delta y^2} = g(x_j, y_k) \quad (36.2)$$

for  $1 \leq j \leq J - 1$ ,  $1 \leq k \leq K - 1$ . Again, the equations for the conditions along the boundaries of the domain will replace these equations by incorporating the given boundary conditions of the problem in the same manner as discussed in the previous chapter.

The collection of equations can be assembled into a large sparse matrix equation. One strategy for solving this type of problem is simply to use a sparse matrix solver. This is perfectly fine when the problem is not large, i.e., the number of grid points in the system is manageable, but that can change quickly. It should be kept in mind how large these systems can be. For example, in the simple two-dimensional problem described here, there will be  $(J + 1)(K + 1)$  variables to be solved given by  $u_{j,k}$  for  $0 \leq j \leq J$ ,  $0 \leq k \leq K$ . This means the dimensions of the matrix that must be inverted is  $(J + 1)(K + 1) \times (J + 1)(K + 1)$ . It is true that the matrix is sparse, in that it is mostly zeros, but it is no longer tridiagonal, so the solver is now more expensive and requires significantly more storage. It only gets worse going to three dimensions. The alternative presented here is to use an iterative method.

Returning to Equation (36.2), solving for  $u_{j,k}$  results in the equation

$$\begin{aligned} u_{j,k} = & \frac{1}{2} \frac{\Delta y^2}{\Delta x^2 + \Delta y^2} (u_{j+1,k} + u_{j-1,k}) \\ & + \frac{1}{2} \frac{\Delta x^2}{\Delta x^2 + \Delta y^2} (u_{j,k+1} + u_{j,k-1}) - \frac{1}{2} \frac{\Delta x^2 \Delta y^2}{\Delta x^2 + \Delta y^2} g(x_j, y_k) \end{aligned} \quad (36.3)$$

The equation is simpler still if we assume that  $\Delta x = \Delta y$ , resulting in

$$u_{j,k} = \frac{1}{4} (u_{j+1,k} + u_{j-1,k}) + \frac{1}{4} (u_{j,k+1} + u_{j,k-1}) - \frac{\Delta x^2}{4} g(x_j, y_k). \quad (36.4)$$

From here on, to simplify the discussion, we will assume  $\Delta x = \Delta y$ .

Adding and subtracting  $u_{j,k}$  to the right-hand side, the equation becomes

$$u_{j,k} = u_{j,k} + \frac{1}{4} (u_{j+1,k} - 2u_{j,k} + u_{j-1,k}) + \frac{1}{4} (u_{j,k+1} - 2u_{j,k} + u_{j,k-1}) - \frac{\Delta x^2}{4} g(x_j, y_k). \quad (36.5)$$

An initial iterative method can now be constructed from this equation. Let  $u_{j,k}^n$  be the  $n$ th iterate, then set

$$u_{j,k}^{n+1} = u_{j,k}^n + \frac{1}{4} (u_{j+1,k}^n - 2u_{j,k}^n + u_{j-1,k}^n) + \frac{1}{4} (u_{j,k+1}^n - 2u_{j,k}^n + u_{j,k-1}^n) - \frac{\Delta x^2}{4} g(x_j, y_k). \quad (36.6)$$

The iteration has converged when the residual, given by

$$R_{j,k}^n = \frac{1}{4} (u_{j+1,k}^n - 2u_{j,k}^n + u_{j-1,k}^n) + \frac{1}{4} (u_{j,k+1}^n - 2u_{j,k}^n + u_{j,k-1}^n) - \frac{\Delta x^2}{4} g(x_j, y_k), \quad (36.7)$$

is within a specified tolerance of zero for every  $j, k$ . This is called the Jacobi iterative method, which is not an ideal scheme.

The method can be improved by adding a relaxation factor  $\omega$ :

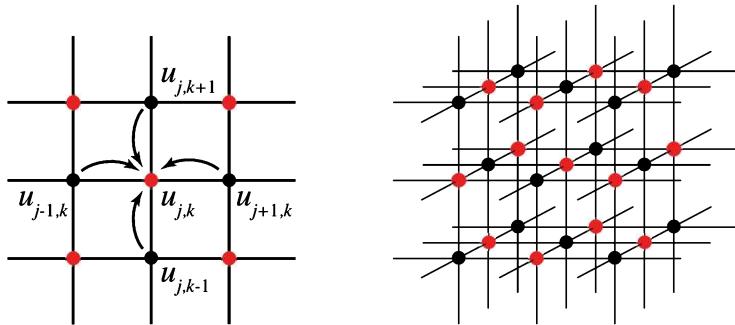
$$\begin{aligned} u_{j,k}^{n+1} = & u_{j,k}^n + \omega R_{j,k}^n, \\ = & (1 - \omega) u_{j,k}^n + \frac{\omega}{4} (u_{j+1,k}^n + u_{j-1,k}^n + u_{j,k+1}^n + u_{j,k-1}^n) \\ & - \frac{\omega \Delta x^2}{4} g(x_j, y_k), \end{aligned} \quad (36.8)$$

where  $0 < \omega < 2$ , but it still requires two grids. The successive overrelaxation (SOR) method is different in that it solves the solution on one grid by sweeping over the domain sequentially. If the points are swept in the order of increasing values of  $j$  and  $k$ , then it means that the equation becomes

$$u_{j,k}^{n+1} = (1 - \omega) u_{j,k}^n + \frac{\omega}{4} (u_{j+1,k}^n + u_{j-1,k}^{n+1} + u_{j,k+1}^n + u_{j,k-1}^{n+1}) - \frac{\omega \Delta x^2}{4} g(x_j, y_k). \quad (36.9)$$

In practice, what this means is that the method sweeps through the domain sequentially. At each grid point the residual is computed using whatever data is currently stored in the neighboring grid points. Once the residual is calculated, the value of  $u_{j,k}$  is updated and stored on top of the old value.

The relaxation factor  $\omega$  is generally in the range  $0 < \omega < 2$ , where for  $\omega > 1$  it is considered overrelaxation and for  $\omega < 1$  underrelaxation. The choice of  $\omega$  can



**Figure 36.1.** Examples of a checkerboard red/black ordering for two dimensions (left) and three dimensions (right). The grid points with the same color can be updated in parallel without conflict.

significantly improve the convergence rate of the method, so it is worthwhile to carefully choose  $\omega$  either through various estimation techniques or by experimentation when developing a high-speed solver.

It should also be noted that for the SOR method, the order in which the points are updated can be changed. The sequential algorithm described above is difficult to parallelize, but a reorganization using a checkerboard pattern can be useful for the case of solving Equation (36.8). Figure 36.1 illustrates the pattern used for parallelizing the SOR method in two and three dimensions using red/black ordering. In this framework, the grid points of the same color can be updated in parallel without conflicting with each other because of interdependencies. Thus, the SOR method updates all the red grid points in one step, and then updates all the black grid points in the second step.

## 36.1 • Problems to Solve

### 36.1.1 • Diffusion with Sources/Sinks

Consider the elliptic equation

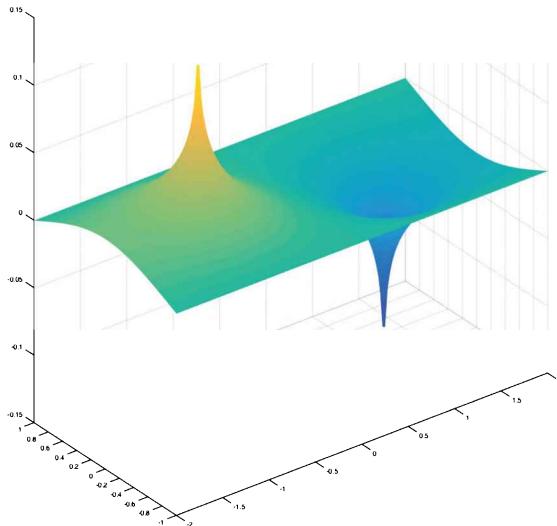
$$\nabla^2 u = \frac{10\lambda}{\sqrt{\pi}} e^{-\lambda^2((x-1)^2+y^2)} - \frac{10\lambda}{\sqrt{\pi}} e^{-\lambda^2((x+1)^2+y^2)}, \quad (36.10)$$

$$\begin{aligned} \frac{\partial u}{\partial x}(\pm 2, y) &= 0, \\ u(x, \pm 1) &= 0, \end{aligned}$$

in the domain  $-2 \leq x \leq 2$ ,  $-1 \leq y \leq 1$  for the parameter value  $\lambda = 100$ . Here, the two terms represent approximations of a Dirac delta point source of strength 10 at  $(-1, 0)$  and a sink at  $(1, 0)$ . This is a model of a plate with two sides held at a fixed temperature, and the other two sides insulated, and a source and sink of heat. The value of  $\lambda$  controls how narrow or wide the approximate Dirac delta is spread, with larger values being more narrow, and approaching the true Dirac delta as  $\lambda \rightarrow \infty$ .

**Assignment.** Solve Equation (36.10) using the SOR method with an error tolerance of  $10^{-9}$ . Your program should take four arguments so that your program named `diffusion` can be executed like this:

```
$ diffusion 128 1.8 1e-9 10000
```



**Figure 36.2.** Solution for Equation (36.10) when  $\lambda = 100$ . Note that the vertical scale is exaggerated compared to the horizontal scale.

where the number of grid points in the  $y$ -direction is  $N = 128$ , the double precision parameter is  $\omega = 1.8$ , the convergence tolerance is  $\tau = 10^{-9}$ , and a fail-safe stopping criterion for the maximum number of iterations is  $K = 10000$ . The domain is rectangular, so for the input value of  $N$  in the  $y$ -direction, this will lead to  $\Delta y = \frac{2}{N-1}$ . To make  $\Delta x = \Delta y$ , take the number of grid points in the  $x$ -direction to be  $M = 2N - 1$ . The fail-safe stopping criterion should be set up so that if the number of iterations exceeds  $K$ , then the calculation terminates with a warning message that it failed to converge and what the current maximum error is.

Write the final result for  $\mathbf{u}$  to the file named “Sources.out”, which must be the  $2N - 1 \times N$  solution.

Experiment with various values of the relaxation factor  $\omega$  to determine the value that requires the fewest iterations to meet the error tolerance.

Your program should also print to the screen the total time required to complete the calculation divided by the number of iterations used. Vary the values of  $N$  and generate a plot that compares the time to complete one iteration versus  $N$ .

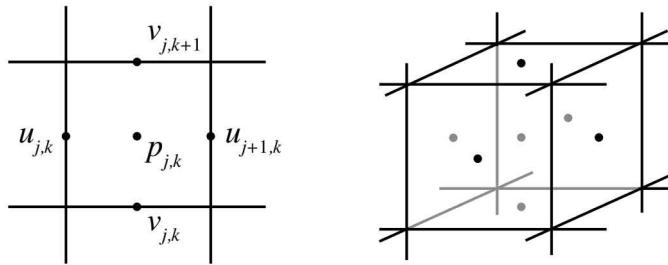
The solution for the case of  $\lambda = 100$  is shown in Figure 36.2.

### 36.1.2 • Stokes Flow

Stokes flow is an approximation for incompressible fluid flow when the flow is highly viscous or slow moving. It is a linearization of the more general Navier–Stokes equations for modeling fluid flow. The equations for Stokes flow are

$$\mu \nabla^2 \mathbf{u} = \nabla p - \mathbf{f}, \quad (36.11)$$

$$\nabla \cdot \mathbf{u} = 0, \quad (36.12)$$



**Figure 36.3.** Illustration of the MAC grid. The  $u$ -velocity grid lies on the vertical grid lines at the midpoints between horizontal grid lines. The  $v$ -velocity grid lies on the horizontal grid lines at the midpoints between vertical grid lines. The pressure  $p$  grid is in the center of the box (left). The MAC grid is similar for three dimensions, where the  $u$ -velocity grid has points on the centers of the facets with fixed  $x$  value, and likewise for the  $y$ - and  $z$ -velocity grids. The pressure  $p$  is again in the center of the box (right).

where  $\mathbf{u}$  is the fluid velocity,  $p$  is the fluid pressure,  $\mathbf{f} = (f, g)$  is the body force on the fluid, and  $\mu$  is the fluid viscosity.

Solving this system where the velocity and the pressure are on the same grid does not work very well, so instead a grid called the marker-and-cell (MAC) method is used. Figure 36.3 illustrates how the MAC grid is arranged for the unknowns  $\mathbf{u} = (u, v)$  and  $p$ . The grids are all staggered to improve the accuracy of the resulting discretization. Using this grid in two dimensions produces the following discrete versions of Equations (36.11) and (36.12):

$$\begin{aligned} \frac{\mu}{\Delta x^2}(u_{j-1,k} - 2u_{j,k} + u_{j+1,k}) + \frac{\mu}{\Delta y^2}(u_{j,k+1} - 2u_{j,k} + u_{j,k-1}) \\ = \frac{1}{\Delta x}(p_{j,k} - p_{j-1,k}) - f(x_j, y_k + \Delta y/2), \end{aligned} \quad (36.13)$$

$$\begin{aligned} \frac{\mu}{\Delta x^2}(v_{j-1,k} - 2v_{j,k} + v_{j+1,k}) + \frac{\mu}{\Delta y^2}(v_{j,k+1} - 2v_{j,k} + v_{j,k-1}) \\ = \frac{1}{\Delta y}(p_{j,k} - p_{j,k-1}) - f(x_j + \Delta x/2, y_k), \end{aligned} \quad (36.14)$$

$$0 = \frac{1}{\Delta x}(u_{j,k} - u_{j-1,k}) + \frac{1}{\Delta y}(v_{j-1,k+1} - v_{j-1,k}). \quad (36.15)$$

To convert these to the form needed for the SOR method, the equations are rescaled and then the residuals computed for Equations (36.13), (36.14), and (36.15) as

$$\begin{aligned} r_{j,k}^u = \frac{\mu \Delta y}{\Delta x}(u_{j-1,k} - 2u_{j,k} + u_{j+1,k}) + \frac{\mu \Delta x}{\Delta y}(u_{j,k+1} - 2u_{j,k} + u_{j,k-1}) \\ - \Delta y(p_{j,k} - p_{j-1,k}) + \Delta x \Delta y f(x_j, y_k + \Delta y/2), \end{aligned} \quad (36.16)$$

$$\begin{aligned} r_{j,k}^v = \frac{\mu \Delta y}{\Delta x}(v_{j-1,k} - 2v_{j,k} + v_{j+1,k}) + \frac{\mu \Delta x}{\Delta y}(v_{j,k+1} - 2v_{j,k} + v_{j,k-1}) \\ - \Delta x(p_{j,k} - p_{j,k-1}) + \Delta x \Delta y f(x_j + \Delta x/2, y_k), \end{aligned} \quad (36.17)$$

$$r_{j,k}^p = -(u_{j,k} - u_{j-1,k}) - \frac{\Delta x}{\Delta y}(v_{j-1,k+1} - v_{j-1,k}). \quad (36.18)$$

Once the residuals are computed, the SOR method becomes

$$u_{j,k}^{n+1} = u_{j,k}^n + \omega r_{j,k}^u, \quad (36.19)$$

$$v_{j,k}^{n+1} = v_{j,k}^n + \omega r_{j,k}^v, \quad (36.20)$$

$$p_{j,k}^{n+1} = p_{j,k}^n + \omega r_{j,k}^p. \quad (36.21)$$

**Assignment.** Use the SOR method to find the steady Stokes flow velocity field for a channel of width one and length one and with a pressure drop of magnitude  $P$  between the inlet and the outlet. The system of equations to solve is

$$\mu \left( \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) = \frac{\partial p}{\partial x}, \quad (36.22)$$

$$\mu \left( \frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} \right) = \frac{\partial p}{\partial y}, \quad (36.23)$$

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0, \quad (36.24)$$

$$u(x, 0) = u(x, 1) = v(x, 0) = v(x, 1) = 0, \quad (36.25)$$

$$p(0, y) = P, \quad p(1, y) = 0. \quad (36.26)$$

To solve the equations, use a MAC grid as discussed above. Figure 36.4 illustrates the location of the grid points for  $u$ ,  $v$ , and  $p$  in the domain. If the base grid is nominally  $N \times N$ , then the grid for  $u$  will be  $N \times N - 1$ , the grid for  $v$  will be  $N - 1 \times N$ , and the grid for  $p$  will be  $N - 1 \times N - 1$ . For a square domain, this will ensure that the space step will be  $\Delta x = \Delta y = \frac{1}{N-1}$ .

Next, modify the stencils at the boundary to account for the boundary conditions. For the top and bottom walls, where  $u = v = 0$ , clearly  $v_{j,0} = 0$ ,  $v_{j,N-1} = 0$  because the grid points for  $v$  lie exactly on the wall. On the other hand, the grid points for  $u$  are not on the wall, so use interpolation to enforce  $u(x, 0) = u(x, 1) = 0$ . Thus, it must be that if there were a grid point at  $u_{j,-1}$ , then it must be that  $u_{j,-1} = -u_{j,0}$  so that linear interpolation would produce  $u = 0$  at the wall. Similarly,  $u_{j,N-1} = -u_{j,N-2}$ . Rather than adding ghost points for  $u_{j,-1}$ ,  $u_{j,N}$ , just alter the stencil for the boundary cases. For example, after making the substitution  $u_{j,-1} = -u_{j,0}$ , Equation (36.13) for  $u_{j,0}$  becomes

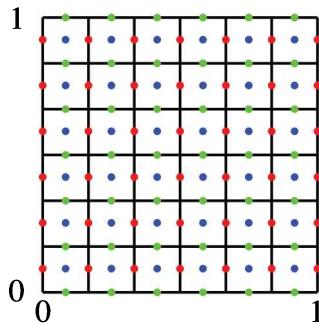
$$\frac{\mu}{\Delta x^2} (u_{j-1,0} - 2u_{j,0} + u_{j+1,0}) + \frac{\mu}{\Delta y^2} (u_{j,1} - 2u_{j,0} - u_{j,-1}) = \frac{1}{\Delta x} (p_{j+1,0} - p_{j,0}). \quad (36.27)$$

At the inlet and the outlet, assume that  $\frac{\partial u}{\partial x} = \frac{\partial v}{\partial x} = 0$  for purposes of the stencil. This means that  $u_{-1,k} = u_{0,k}$ ,  $v_{-1,k} = v_{0,k}$ ,  $u_{N+1,k} = u_{N,k}$ , and  $v_{N,k} = v_{N-1,k}$ . These substitutions are made in the appropriate stencils as above. To enforce the pressure boundary conditions, Equation (36.26) must be enforced. Since the pressure straddles the location of the boundary condition, enforce the pressure condition numerically by the equations

$$\frac{1}{2} (p_{-1,k} + p_{0,k}) = P, \quad (36.28)$$

$$\frac{1}{2} (p_{N-1,k} + p_{N-2,k}) = 0. \quad (36.29)$$

Solving Equation (36.28) for  $p_{-1,k}$  and substituting into Equation (36.16), where  $u_{-1,k} = u_{0,k}$  from the boundary conditions for  $u$  described above is also used, and where



**Figure 36.4.** Illustration of the MAC grid for the two-dimensional Stokes assignment. The  $u_{j,k}$  values are in red, the  $v_{j,k}$  values are in green, and the  $p_{j,k}$  values are in blue. If a standard grid (black lines) would be an  $N \times N$  grid, then the  $u_{j,k}$  grid would be an  $N \times N - 1$  grid, the  $v_{j,k}$  grid would be  $N - 1 \times N$ , and the  $p_{j,k}$  grid would be  $N - 1 \times N - 1$ . Care must be taken to ensure that the indexing for the three different grids is correct.

$f = 0$ , a new formula for the residual at the inlet results:

$$\begin{aligned} r_{0,k}^u &= \frac{\mu \Delta y}{\Delta x} (u_{0,k} - 2u_{0,k} + u_{1,k}) + \frac{\mu \Delta x}{\Delta y} (u_{0,k+1} - 2u_{0,k} + u_{0,k-1}) \\ &\quad - \Delta y (p_{0,k} - (2P - p_{0,k})), \\ &= \frac{\mu \Delta y}{\Delta x} (-u_{0,k} + u_{1,k}) + \frac{\mu \Delta x}{\Delta y} (u_{0,k+1} - 2u_{0,k} + u_{0,k-1}) - 2\Delta y (p_{0,k} - P). \end{aligned}$$

Follow the same procedures for the outlet boundary conditions.

With these changes made for the boundary conditions, a complete SOR solver for this system can be constructed. Compute the flow field for a suitable tolerance. For some values of  $\omega$  in the range  $0 < \omega < 2$ , the method will not converge, and for some values it will. Find the value of  $\omega$  that makes the method converge fastest.

Your program should take six arguments so that your program named `stokes` can be executed like this:

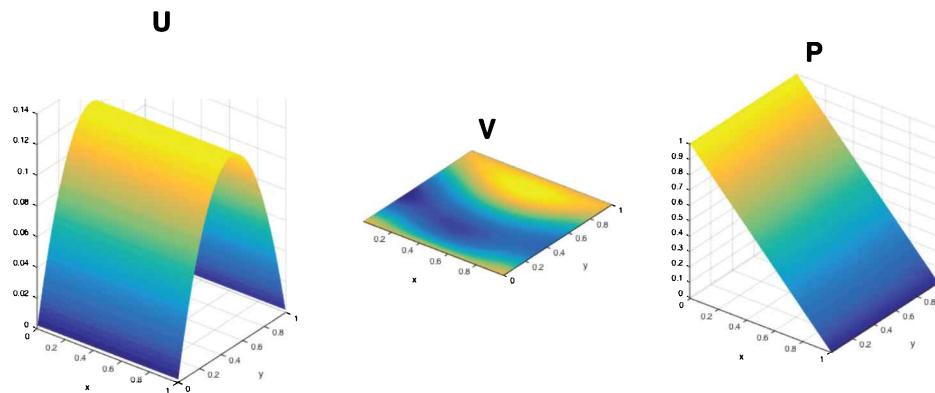
```
$ stokes 128 1 1 0.4 1e-9 100000
```

where  $N = 128$  is the grid size, the viscosity is  $\mu = 1$ , the pressure drop is  $P = 1$ , the relaxation parameter is  $\omega = 0.4$ , the error tolerance is  $\tau = 10^{-9}$ , and the maximum number of iterations is  $K = 100,000$ . Use initial data of  $u = v = p = 0$ . Experiment with different values of  $\omega$  to find an optimal relaxation parameter value. Save your final results into three files called “`StokesU.out`”, “`StokesV.out`”, and “`StokesP.out`”, respectively.

Your program must also print to the screen the total time required to complete the calculation divided by the number of iterations used. Vary the values of  $N$  and generate a plot that compares the time to complete one iteration versus  $N$ .

For the case of  $\mu = P = 1$ , the solution is shown in Figure 36.5.

**Bonus Assignment.** Solve the three-dimensional analogue of the above problem of an applied pressure differential across a duct with square cross section.



**Figure 36.5.** Illustration of the solutions for the Stokes flow problem with pressure drop  $P = 1$  and viscosity  $\mu = 1$ . The exact solution is  $u = \frac{1}{2}y(1-y)$ ,  $v = 0$ , and  $p = 1 - x$ .

## Chapter 37

# Pseudospectral Methods

For PDEs with periodic boundary conditions, sometimes it is advantageous to use spectral methods for the solution. There are a few ways to utilize spectral methods; however in this chapter only one method will be presented, namely, the pseudospectral method. For this method, the basic strategy is similar to the finite difference approach, but the means by which the spatial derivatives are computed is different. For the spectral method, the Fourier transform is used to transform a function into spectral space, and the spatial derivative is calculated in spectral space, where it is a simple operation. The result is then transformed back into real space.

### 37.1 • Fourier Transform

Before diving into the numerics, let's begin with a brief introduction to the Fourier transform. Given a function  $f(x)$  that is periodic on the interval  $[-L, L]$ , define  $g(x) = f(\pi x/L)$ ; then  $g(x)$  is a function that is periodic on the interval  $[-\pi, \pi]$ . Thus, it is safe to assume that the functions and equations being solved are periodic on the interval  $[-\pi, \pi]$ . This will simplify the discussion, so from here on assume that functions are periodic on  $[-\pi, \pi]$ .

The Fourier transform tells us that a function  $f(x)$  can be decomposed into a series of discrete wave forms via the equation

$$f(x) = \sum_{k=-\infty}^{\infty} a_k e^{ikx} = \sum_{k=-\infty}^{\infty} a_k (\cos kx + i \sin kx), \quad (37.1)$$

$$a_k = \frac{1}{2\pi} \int_{-\pi}^{\pi} f(x) e^{-ikx} dx, \quad (37.2)$$

where  $k$  is the wave number and the  $a_k$  are the complex-valued spectral coefficients. The series converges if and only if the function  $f(x)$  is continuous. The function  $f(x)$  may be complex valued, but if  $f(x)$  is in fact real-valued, then it is necessarily the case that  $a_{-k} = \bar{a}_k$  in order for the imaginary part of the series to cancel out. Furthermore, the magnitude of the coefficients  $|a_k| \rightarrow 0$  as  $|k| \rightarrow \infty$  and the rate of decay of the  $|a_k|$  increases with the number of continuous derivatives  $f(x)$  possesses. In fact, if  $f(x)$  has  $r$  continuous derivatives, then  $k^{2r} |a_k|^2 \rightarrow 0$  as  $|k| \rightarrow \infty$ .

Of course, an infinite sum is not very useful in a discrete computation, so an approximation is necessary. For spectral methods, it is made discrete by taking an orthogonal projection of the full space down to the space of functions that can be represented by a finite sum. In other words, the function  $f(x)$  is approximated by truncating the Fourier series:

$$f(x) \approx \sum_{k=-N}^N a_k e^{ikx}. \quad (37.3)$$

In practice, the  $a_k$  cannot be computed exactly as defined in Equation (37.2), but instead the computation will rely on an approximation based upon the trapezoidal rule resulting in the discrete formula

$$a_k \approx \frac{1}{2N} \sum_{j=-N}^{N-1} f(x_j) e^{-ikx_j}, \quad (37.4)$$

where the domain is discretized into  $2N$  points  $x_j = \frac{\pi j}{N}$  for  $j = -N, \dots, N-1$  that are often called collocation points. The keen-eyed student may notice that this means  $2N$  collocation points are being used to generate  $2N+1$  Fourier coefficients  $a_{-N}, \dots, a_N$  and hence there must be some linear dependency. In fact, there is, because when  $k = N$ ,  $e^{-iNx_j} = (-1)^j = e^{iNx_j}$ , and therefore

$$a_{-N} = \frac{1}{2N} \sum_{j=-N}^{N-1} f(x_j) e^{-iNx_j} = \frac{1}{2N} \sum_{j=-N}^{N-1} f(x_j) e^{-i(-N)x_j} = a_N.$$

Thus, when computing  $a_N$  using this approximation, it is assumed that the  $N$ th coefficient is divided equally between  $a_N$  and  $a_{-N}$  so that

$$a_N = a_{-N} = \frac{1}{2} \frac{1}{2N} \sum_{j=-N}^{N-1} f(x_j) e^{-iNx_j}. \quad (37.5)$$

Finally, it is important to note that this spectral approximation of the function  $f(x)$  is actually interpolating. That means that if the  $a_k$  are computed using the approximation in Equation (37.4) with  $a_N, a_{-N}$  defined as in Equation (37.5), and if  $g(x)$  is defined to be

$$g(x) = \sum_{k=-N}^N a_k e^{ikx}, \quad (37.6)$$

then  $g(x_j) = f(x_j)$  for all  $j = 0, \dots, 2N-1$ . Thus,

$$f(x_j) = \sum_{k=-N}^N a_k e^{ikx_j}. \quad (37.7)$$

As it stands thus far, the conversion of the  $2N$  values of  $f(x_j)$  into the  $2N+1$  Fourier coefficients appears to be an operation of order  $O(N^2)$  because a sum of length  $2N$  must be computed for each of the  $2N$  Fourier coefficients (noting from above that only one of  $a_N$  and  $a_{-N}$  must be computed to get both values). Thanks to the work of Cooley and Tukey [29], when computing the full range of Fourier coefficients there are efficiencies that can be exploited to reduce the number of operations resulting in an operation of order  $O(N \log N)$  provided  $N$  is ideally a power of 2, but more generally  $N$  should be the product of powers of small prime numbers. This is the fast Fourier transform, or FFT, and has become a core staple for spectral numerical methods. Different

implementations of the FFT produce the same coefficients up to a constant multiple, which may vary. They often return the coefficients in a different order, so it's important to understand how the package you are using operates. The sections in this book where an FFT library is used describe the nuances of those implementations.

Comparing the sums in Equations (37.4) and (37.7), it appears that both are sums of a known set of coefficients multiplied by  $e^{\pm ikx_j}$  with a possible adjustment for a constant factor out front and having to combine or split the values of  $a_{\pm N}$ . In fact it is the case, and hence the FFT can be used to go in both directions: forward from real space to spectral space, and backward from spectral space to real space. Thus, given the function values  $f(x_j)$ , applying the forward FFT will produce the coefficients  $a_{-N}, \dots, a_{N-1}$ , and then following Equation (37.5), the value returned in  $a_{-N}$  is divided by 2 and assigned to  $a_N$  and  $a_{-N}$ .

To go from spectral space to real space, the coefficients  $a_N$  and  $a_{-N}$  are summed together and assigned to  $a_{-N}$ , and then the backward FFT is applied to the coefficients  $a_{-N}, \dots, a_{N-1}$  to produce the values  $f(x_j)$  for  $j = -N, \dots, N-1$ .

## 37.2 • Spectral Differentiation

One of the key aspects of spectral methods is that differentiation of functions in this representation is very simple and accurate because it can be computed exactly on the spectral approximation. Let

$$g(x) = \sum_{k=-N}^N a_k e^{ikx}; \quad (37.8)$$

then differentiation by  $x$  gives

$$g'(x) = \sum_{k=-N}^N b_k e^{ikx} = \sum_{k=-N}^N ika_k e^{ikx}, \quad (37.9)$$

where the  $b_k$  are the Fourier coefficients of  $g'(x)$ . Therefore, spatial derivatives in real space translate into a simple transformation of the Fourier coefficients,

$$b_k = ika_k. \quad (37.10)$$

When  $k = \pm N$ , recall that when constructing  $a_{\pm N}$ , we set them so that  $a_N = a_{-N}$ . That means when taking the derivative,  $b_N = iNa_N$  and  $b_{-N} = -iNa_{-N} = -iNa_N$ . When taking the backward FFT to get back to real space, we must add  $b_N + b_{-N} = iNa_N - iNa_N = 0$ . In other words, when computing the first derivative in spectral space, the  $N$ th terms will cancel to produce zero. This will not happen for even-ordered derivatives.

Higher-order derivatives can be computed as easily as the first derivative by using the equation

$$g^{(n)}(x) = \sum_{k=-N}^N (ik)^n a_k e^{ikx}. \quad (37.11)$$

In other words, higher-order derivatives still only require one FFT into spectral space and one FFT back into real space.

Putting it all together, given function data defined on collocation points  $f(x_j)$  for  $x_j = \frac{\pi}{N}j$ ,  $j = -N, \dots, N-1$ , computing the  $n$ th derivative  $f^{(n)}(x_j)$  is done by the following steps:

1. Compute the forward FFT on the array  $[f(x_{-N}), \dots, f(x_{N-1})]$  to produce the spectral coefficients  $a_{-N}, \dots, a_{N-1}$ .
2. Set  $b_k = (ik)^n a_k$  for  $k = -(N-1), \dots, N-1$ , and set  $b_{-N} = (ik)^n a_{-N}$  if  $n$  is even and  $b_{-N} = 0$  if  $n$  is odd.
3. Compute the backward FFT on the array  $[b_{-N}, \dots, b_{N-1}]$  to produce the result  $[f'(x_{-N}), \dots, f'(x_{N-1})]$ .

### 37.3 • Pseudospectral Method

The pseudospectral method is essentially the coupling of spatial derivatives computed using the spectral approximation described above with a suitable method for time integration. One natural choice for the time integration is the Runge–Kutta method described in Section 35.3. Since the spectral derivative has the potential to have high orders of accuracy for smooth solutions, it makes sense to couple it with a high-order time integration scheme. The pseudospectral method is then a method where the spatial derivative operator  $\mathcal{L}$  is evaluated using the spectral derivative approximation combined with a suitable temporal integration such as Runge–Kutta.

### 37.4 • Higher Dimensions

To solve PDEs that are in two and higher dimensions, the only modification to the discussion above is that to transform an  $n$ -dimensional space, the Fourier transform is applied one dimension at a time to get the coefficients. The FFT libraries presented in this book have both two-dimensional and three-dimensional transforms built into the library, so no additional effort is required on the part of the user other than the corrections that must be done for the ends of the discrete spectra. Here the modifications for two dimensions are presented; higher dimensions are analogous.

The analogue of Equations (37.3) and (37.4) in two dimensions are

$$f(x, y) \approx \sum_{j=-M}^M \sum_{k=-N}^N a_{j,k} e^{i(jx+ky)}, \quad (37.12)$$

$$a_{j,k} \approx \frac{1}{2M} \frac{1}{2N} \sum_{\ell=-M}^{M-1} \sum_{m=-N}^{N-1} f(x_\ell, y_m) e^{-i(jx_\ell + ky_m)}. \quad (37.13)$$

When using the FFT to do the transforms, the coefficients  $a_{-M,k}$  and  $a_{\ell,-N}$  must be divided in two and split as before. Thus, if  $a_{j,k}$  is the output of the FFT, and  $\tilde{a}_{j,k}$  are the corrected coefficients, then

$$\begin{aligned} \tilde{a}_{-M,-N} &= \tilde{a}_{-M,N} = \tilde{a}_{M,-N} = \tilde{a}_{M,N} = \frac{1}{4} a_{-M,-N}, \\ \tilde{a}_{-M,k} &= \tilde{a}_{M,k} = \frac{1}{2} a_{-M,k}, \text{ for } k = -N+1, \dots, N-1, \\ \tilde{a}_{j,-N} &= \tilde{a}_{j,N} = \frac{1}{2} a_{j,-N}, \text{ for } j = -M+1, \dots, M-1. \end{aligned}$$

## 37.5 • Problems to Solve

### 37.5.1 • The Complex Ginsburg–Landau Equation

The complex Ginsburg–Landau equation can exhibit pattern formation for certain values of the coefficients [30]. The equation is given by

$$\frac{\partial A}{\partial t} = A + (1 + ic_1)\nabla^2 A - (1 - ic_3)|A|^2 A, \quad (37.14)$$

where  $A(\mathbf{x}, t)$  is a complex valued field.

**Assignment.** Solve the complex Ginsburg–Landau equation in two dimensions (or three dimensions) on a domain that is  $L = 128\pi$  on each side. Converting to a length of  $2\pi$  on each side means that Equation (37.14) is rescaled to be

$$\frac{\partial A}{\partial t} = A + \left(\frac{2\pi}{L}\right)^2 (1 + ic_1)\nabla^2 A - (1 - ic_3)|A|^2 A. \quad (37.15)$$

Use the pseudospectral method with the fourth-order Runge–Kutta method using random initial data in the range  $[-1.5, 1.5] + i[-1.5, 1.5]$  and run until terminal time  $T = 10^4$ . To see any resulting patterns, use a contour plot on the amplitude  $|A|$  and also a contour plot on the phase using `atan2(A.imag, A.real)`. For the parameters  $c_1 = 1.5$ ,  $0 < c_3 \leq 0.75$ , spiral waves should emerge.

Your program should take four arguments with an optional fifth argument for the seed value so that your program named `cgl` can be executed like this:

```
$ cgl 128 1.5 0.25 100000
```

where  $N = 128$  is the size of the grid in both  $x$ - and  $y$ -dimensions,  $c_1 = 1.5$ ,  $c_3 = 0.25$  are the real-valued coefficients in Equation (37.15), and  $M = 100,000$  is the number of time steps. If a fifth argument is specified, then it is a long integer seed value,  $s$ , for generating the initial random values. You will need to check the value of `argc` to determine if  $s$  was given. Your program must print the arguments given including the value of the seed, and you should verify that your seed argument is implemented correctly by taking the value printed by your program from the above test run and adding it to the argument list. For example, if your program reports using a seed of 12345, then execute your program again like

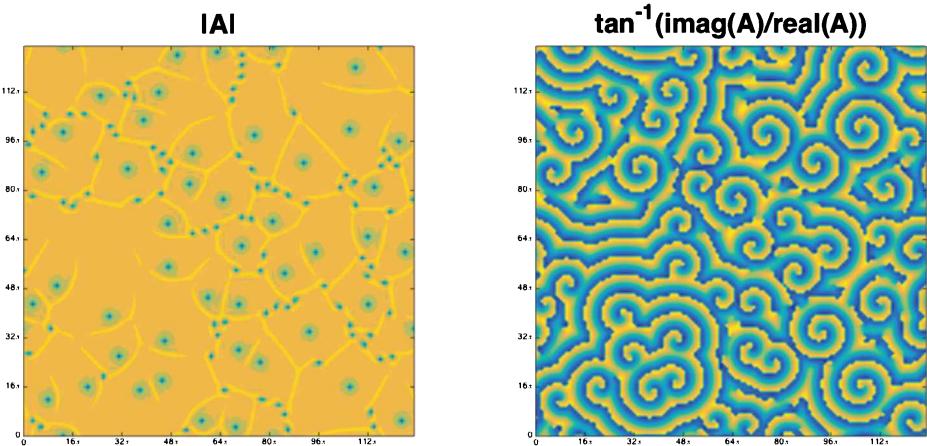
```
$ cgl 128 1.5 0.25 100000 12345
```

and verify that you get the exact same results.

Note that because this is on a periodic domain, the space step size will be  $\Delta x = \Delta y = \frac{2\pi}{N}$  with  $x_0 = -\pi$  and  $x_{N-1} = \pi - \Delta x$ . Your program should output the grid values for  $A$  in a file called “`CGL.out`” that contains *only* the data at the time points  $t = 1000k$  for  $k = 0, 1, \dots, 10$ .

Your program must also print to the screen the total time required to complete the calculation. Use varying values of  $N$  to generate a plot that illustrates the scaling of the computational cost versus  $N$ .

Figure 37.1 shows an example of the solution at the terminal time  $T = 10,000$  for the case of  $L = 128\pi$ ,  $c_1 = 1.5$ , and  $c_3 = 0.25$ .



**Figure 37.1.** Illustration of a sample solution at the terminal time  $T = 10,000$  for the case of  $c_1 = 1.5$ ,  $c_3 = 0.25$ , and  $L = 128\pi$  in Equation (37.15). On the left is a contour plot of the magnitude  $|A|$  and on the right is a contour plot of the phase,  $\arg(A) = \tan^{-1}(\text{Im}(A)/\text{Re}(A))$ . The spiral patterns that emerge can be seen in the right-hand plot. Results will vary according to the random initial data.

### 37.5.2 • Allen–Cahn Equation

The Allen–Cahn equation [31] is an equation that can be used to study phase separation in multicomponent alloys, among other things. It is a balance between a diffusion operator that smooths the transition area between the phases and the free energy density that drives the material to separate into one phase where  $\phi = -1$  or the other phase where  $\phi = 1$ . It is a general equation, but for this example it will take the form

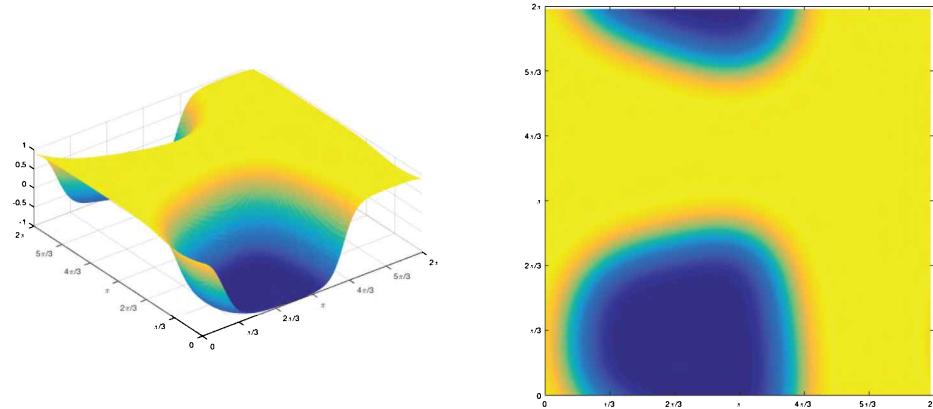
$$\frac{\partial \phi}{\partial t} = -\mathbf{v} \cdot \nabla \phi + b \left( \nabla^2 \phi + \frac{\phi(1 - \phi^2)}{W^2} \right). \quad (37.16)$$

Use the pseudospectral method with the fourth-order Runge–Kutta method using random initial data in the range  $-1 \leq \phi \leq 1$  and run until the terminal time  $T = 5$ . Use  $b = 0.25$ ,  $W = 0.25$ , and  $\mathbf{v} = [10, -5]$  in two dimensions or  $\mathbf{v} = [-10, -5, 0]$  in three dimensions.

Your program should take six arguments with an optional seventh argument for the seed value so that your program named `allen` can be executed like this:

```
$ allen 128 10 -5 0.25 0.25 50000
```

where  $N = 128$  is the size of the grid in both  $x$  and  $y$  dimensions,  $\mathbf{v} = (10, -5)$  are the two components of the velocity vector,  $b = 0.25$  and  $W = 0.25$  are the values of the coefficients, and  $M = 50,000$  is the number of time steps to use. If a seventh argument is specified, then it is a long integer seed value,  $s$ , for generating the initial random values. You will need to check the value of `argc` to determine if  $s$  was given. Your program must print the arguments given including the value of the seed, and you should verify that your seed argument is implemented correctly by taking the value printed by your program from the above test run and adding it to the argument list. For example,



**Figure 37.2.** Illustration of a solution for the Allen–Cahn equation (37.16) with  $b = W = 0.25$  at  $t = 2.5$ . Here, the phases have mostly separated into two phases represented by the order parameter  $\phi = \pm 1$ . Results will vary depending on the initial random data.

if your program reports using a seed of 12345, then execute your program again like

```
$ allen 128 10 -5 0.25 0.25 50000 12345
```

and verify that you get the exact same results.

Because this is a periodic domain, the space step size for the grid will be  $\Delta x = \Delta y = \frac{2\pi}{N}$  with  $x_0 = -\pi$  and  $x_{N-1} = \pi - \Delta x$ . Your program should output the grid values for  $\phi$  into a file called “Allen.out” that contains *only* the data at the time points  $t = 0.5k$  for  $k = 0, \dots, 10$ . Note that the variable  $\phi$  can be declared as a double or as a double complex when implementing the pseudospectral method, but don’t forget that the original differential equation is real-valued, so the data saved must also be real-valued.

Your program must also print to the screen the total time required to complete the calculation. Use varying values of  $N$  to generate a plot that illustrates the scaling of the computational cost versus  $N$ .

Figure 37.2 illustrates a solution at time  $t = 2.5$  for the case of  $b = W = 0.25$ . As the phases separate, there are regions that tend toward  $\pm 1$ , with transition regions in between. The width of the transition regions is dictated by the value of  $W$ .



# Bibliography

- [1] B. W. KERNIGHAN AND D. M. RITCHIE, *The C Programming Language*, Second ed., Prentice-Hall, Englewood Cliffs, NJ, 1988. (Cited on pp. xi, 15)
- [2] T. WIJMAN <https://newzoo.com/insights/articles/global-games-market-reaches-137-9-billion-in-2018-mobile-games-take-half/>, April 30, 2018. (Cited on p. xii)
- [3] K. KARIMI, N. G. DICKSON, AND F. HAMZE, *A Performance Comparison of CUDA and OpenCL*, <https://arxiv.org/pdf/1005.2581.pdf>. (Cited on p. xiii)
- [4] G. KHANNA AND J. MCKENNON, *Numerical Modeling of Gravitational Wave Sources Accelerated by OpenCL*, Computer Physics Communications, 181(9) (2010), pp. 1605–1611. (Cited on p. xiii)
- [5] <https://developer.apple.com/metal/>. (Cited on p. xiii)
- [6] <https://developer.microsoft.com/en-us/windows>. (Cited on p. 1)
- [7] <https://developer.apple.com/xcode>. (Cited on p. 1)
- [8] <https://www.eclipse.org>. (Cited on p. 1)
- [9] <https://developer.nvidia.com/nsight-eclipse-edition>. (Cited on p. 1)
- [10] <https://ninja-build.org>. (Cited on p. 6)
- [11] <https://cmake.org>. (Cited on p. 6)
- [12] J. LOELIGER AND M. MCCULLOUGH, *Version Control with Git*, Second ed., O'Reilly Media, Inc., Sebastopol, CA, 2012. (Cited on p. 10)
- [13] E. ANDERSON, Z. BAI, C. BISCHOF, S. BLACKFORD, J. DEMMEL, J. DONGARRA, J. DU CROZ, A. GREENBAUM, S. HAMMARLING, A. MCKENNEY, AND D. SORENSEN, *LAPACK Users' Guide*, SIAM, Philadelphia, PA, 1999. (Cited on p. 74)
- [14] <http://www.netlib.org>. (Cited on p. 77)
- [15] B. CHAPMAN, G. JOST AND R. VAN DER PAS, *Using OpenMP: Portable Shared Memory Parallel Programming*, MIT Press, Cambridge, MA, 2008. (Cited on p. 99)
- [16] G. KARNIADAKIS AND R. KIRBY, *Parallel Scientific Computing in C++ and MPI: A Seamless Approach to Parallel Algorithms and Implementation*, Cambridge University Press, New York, NY, 2003. (Cited on p. 156)
- [17] R. COURANT, K. FRIEDRICHHS, AND H. LEWY, *Partial Difference Equations of Mathematical Physics*, Mathematische Annalen, 100 (1928), pp. 32–74. (Cited on p. 172)
- [18] J. SANDERS AND E. KANDROT, *CUDA By Example*, Pearson Education, Inc., Boston, MA 2011. (Cited on p. 230)

- [19] J. CHENG, M. GROSSMAN, AND T. MCKERCHER, *Professional CUDA C Programming*, John Wiley & Sons, Inc., Indianapolis, IN 2014. (Cited on p. 230)
- [20] <https://docs.nvidia.com/cuda/profiler-users-guide/index.html> (Cited on p. 236)
- [21] A. CHRĘSZCZYK AND J. ANDERS *Matrix Computations on the GPU: CUBLAS, CUSOLVER, and MAGMA by Example*, <https://developer.nvidia.com/sites/default/files/akamai/cuda/files/Misc/mygpu.pdf>, 2017. (Cited on p. 287)
- [22] <https://www.khronos.org/opencl/> (Cited on p. 323)
- [23] B. GASTER, L. HOWES, AND D. R. KAEKI, *Heterogeneous Computing with OpenCL*, Second ed., Elsevier Science, Saint Louis, MO, 2012. (Cited on p. 323)
- [24] P. E. KLOEDEN AND E. PLATEN, *Numerical Solution of Stochastic Differential Equations*, Springer, Berlin, 1992. (Cited on p. 413)
- [25] G. E. P. BOX AND M. E. MULLER, *A Note on the Generation of Random Normal Deviates*, The Annals of Mathematical Statistics, 29(2) (1958), pp. 610–611. (Cited on p. 413)
- [26] G. MARSAGLIA AND T. A. BRAY, *A Convenient Method for Generating Normal Variables*, SIAM Review, 6 (1964), pp. 260–264. (Cited on p. 413)
- [27] R. J. LEVEQUE, *Finite Difference Methods for Ordinary and Partial Differential Equations: Steady-State and Time-Dependent Problems*, SIAM, Philadelphia, PA, 2007. (Cited on p. 419)
- [28] J. DOUGLAS, *Alternating Direction Methods for Three Space Variables*, Numerische Mathematik, 4(1) (1962), pp. 41–63. (Cited on p. 424)
- [29] J. W. COOLEY AND J. W. TUKEY, *An Algorithm for the Machine Calculation of Complex Fourier Series*, Mathematics of Computation, 19 (1965), pp. 297–301. (Cited on p. 442)
- [30] Z. NICOLAOU, H. RIECKE, AND A. MOTTER, *Chimera States in Continuous Media: Existence and Distinctness*, Physical Review Letters, 119(24) (2017), 244101-1–6. (Cited on p. 445)
- [31] S. M. ALLEN AND J. W. CAHN, *A Microscopic Theory for Antiphase Boundary Motion and Its Application to Antiphase Domain Coarsening*, Acta Metallurgica, 17(6) (1979), pp. 1085–1095. (Cited on p. 446)

# Index

- !, 28
- !=, 28
- \*, 25
- \*=, 26
- +, 25
- ++, 26
- +=, 26
- , 25
- , 26
- =, 26
- >, 37
- ., 37
- /, 25
- /=, 26
- /dev/urandom, 294
- <, 28
- <=, 28
- ==, 28
- >, 28
- >=, 28
- ?:, 26
- %, 25, 41
- &, 25
- &&, 28
- &&=, 26
- `__FILE__`, 247
- `__LINE__`, 247
- `__shared__`, 249
- `__shfl_down_sync`, 261
- `__shfl_sync`, 261
- `__shfl_up_sync`, 261
- `__shfl_xor_sync`, 261
- `__syncthreads`, 252, 255, 273, 278
- ^, 25
- \, 328
- `private`, 105, 106
- |, 25, 340
- ||, 28
- ||=, 26
- ADI, *see* alternating directions implicit
- Allen–Cahn equation, 446
- alternating directions implicit, 424, 428
- `argc`, 18
- argument, 18
- `argv`, 18
- arrays, 30
- assignment statement, 24
- `atof`, 38
- `atoi`, 36, 38
- `atol`, 38
- atomic, 135
- barrier, 120, 121, 127
- barrier, 362
- bash, 160
- BLAS, 74, 139, 287
- block, 234, 238, 240, 242, 244–246, 249–251, 254, 273, 283, 284, 313–315, 318
- `blockDim`, 245, 250, 252, 255
- `blockIdx`, 239, 242, 245, 250, 252, 255, 314, 315
- blocking, 166
- `bool`, 22, 23
- boolean algebra operations, 27
- Box–Muller method, 413
- Brusselator, 425
- C math library, 29
- `callgrind`, 8
- central finite difference, 427
- `checkCudaErrors`, 245
- `CheckError`, 180, 247, 350
- `CheckErrorValue`, 350
- checkpointing, 195, 196
- `CL_DEVICE_TYPE_ALL`, 380
- `CL_DEVICE_TYPE_CPU`, 380
- `CL_DEVICE_TYPE_GPU`, 380
- `cl_event*`, 370
- `cl_khr_fp64`, 342
- `cl_mem`, 340
- `CL_MEM_COPY_HOST_PTR`, 340, 341
- `CL_MEM_READ_ONLY`, 340
- `CL_MEM_READ_WRITE`, 340
- `CL_MEM_WRITE_ONLY`, 340
- `CL_PROFILING_COMMAND_END`, 374
- `CL_PROFILING_COMMAND_START`, 374
- `CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE`, 372
- `CL_QUEUE_PROFILING_ENABLE`, 328, 373
- `clBuildProgram`, 326, 343, 386
- `clCreateBuffer`, 326, 340
- `clCreateCommandQueueWithProperties`, 326, 339
- `clCreateContext`, 326, 333
- `clCreateKernel`, 326, 344
- `clCreateProgramWithSource`, 326, 343
- `clEnqueueNDRangeKernel`, 326, 341, 353
- `clEnqueueReadBuffer`, 327, 342, 370
- `clEnqueueWriteBuffer`, 340, 370
- `cfft`, 393
- `CLFFT_2D`, 397
- `CLFFT_3D`, 397
- `CLFFT_COMPLEX_INTERLEAVED`, 395
- `CLFFT_INPLACE`, 397
- `CLFFT_OUTOFPLACE`, 397
- `clfftBakePlan`, 395

**clfftCreateDefaultPlan**, 395, 397  
**clfftDestroyPlan**, 396, 397  
**clfftEnqueueTransform**, 395, 398  
**clfftInitSetupData**, 395  
**clfftSetLayout**, 395  
**clfftSetPlanPrecision**, 395  
**clfftSetResultLocation**, 395  
**clfftSetup**, 395  
**clfftTeardown**, 396, 397  
**clFinish**, 327, 328, 342, 370  
**clGetDeviceIDs**, 326, 333  
**clGetDeviceInfo**, 332, 335, 342, 354, 369  
**clGetEventProfilingInfo**, 374  
**clGetPlatformIDs**, 326, 332  
**clGetPlatformInfo**, 332  
**clGetProgramBuildInfo**, 344  
**CLK\_LOCAL\_MEM\_FENCE**, 362  
**clock**, 66, 69  
**clock\_gettime**, 66, 70  
**CLOCKS\_PER\_SEC**, 68  
**clReleaseCommandQueue**, 327  
**clReleaseContext**, 327  
**clReleaseKernel**, 327  
**clReleaseMemObject**, 327  
**clReleaseProgram**, 327  
**clSetKernelArg**, 326, 341, 357  
**clWaitForEvents**, 370  
coalesced memory access, 252  
column-major, 31, 75, 82, 85, 141, 212, 287, 292, 300, 353, 393  
command, 4  
command queue, 328, 337, 339–341, 369, 389  
compiler directive, 17  
complex numbers, 82  
compressed sparse column matrix form, 304  
compressed sparse row matrix form, 302, 303, 307  
compute nodes, 159  
conditional assignment, 26  
constant, 366  
constant memory, 249, 254, 363  
context, 329, 333  
continuation character, 328  
**copyprivate**, 128  
cores, xii  
**Courant–Friedrichs–Levy condition**, 172  
**critical**, 135  
**cuComplex**, 299, 311  
**CUDA Libraries**, 287  
**cuda-gdb**, 235  
**cuda-memcheck**, 237, 238  
**cudaBindTexture**, 256  
**cudaChooseDevice**, 234  
**cudaDeviceGetAttribute**, 234  
**cudaDeviceProp**, 234  
**cudaDeviceSynchronize**, 300, 303, 307  
**cudaEvent\_t**, 282  
**cudaEventCreate**, 282  
**cudaEventDestroy**, 273  
**cudaEventElapsedTime**, 283  
**cudaEventRecord**, 282  
**cudaEventSynchronize**, 282  
**cudaFree**, 237, 241  
**cudaFreeHost**, 270  
**cudaGetDeviceCount**, 281  
**cudaGetDeviceProperties**, 234, 285, 297  
**cudaGetErrorString**, 245  
**cudaHostAlloc**, 270  
**cudaLaunchKernel**, 237  
**cudaMalloc**, 237, 240, 241, 256  
**cudaMemcpy**, 237, 241  
**cudaMemcpyAsync**, 270, 282  
**cudaMemcpyToSymbol**, 256  
**cudaSetDevice**, 233, 281  
**cudaStream\_t**, 269  
**cudaStreamCreate**, 269  
**cudaStreamDestroy**, 269  
**cudaStreamSynchronize**, 271  
**cudaSuccess**, 245  
**cudaUnbindTexture**, 256  
**cuDoubleComplex**, 299, 311  
**cuFFT**, 297  
**cufftComplex**, 299  
**cufftDestroy**, 300  
**cufftDoubleComplex**, 299  
**cufftExecD2Z**, 301  
**cufftExecZ2D**, 301  
**cufftExecZ2Z**, 300  
**cufftPlan2d**, 300, 312  
**cuRAND**, 292, 313  
**curandGenerateUniform**, 295  
**curandState\_t**, 297  
**cuPARSE**, 302  
**cusparseDestroy**, 310  
**cusparseDgtsv**, 311  
**cusparseHandle\_t**, 310  
**cusparseMatDescr\_t**, 307  
**cusparseStatus\_t**, 310  
**device**, 231  
**dgesv\_**, 197, 201  
**dgetrf\_**, 89  
**dgetrs\_**, 222  
**dgtrf\_**, 81  
**dgtrs\_**, 222  
**dim3**, 242, 244  
Dirichlet boundary condition, 421  
distributed system, 155  
**do-while loop**, 54  
**drand48**, 87, 89, 93, 141, 292  
**drand48\_r**, 144  
Duffing–Van der Pol oscillator, 416  
dynamic libraries, 74  
dynamic memory allocation, 30, 32, 33, 36, 37  
**emacs**, 1  
escape code, 41  
Euler equations, 426  
Euler’s method, 422  
Euler–Maruyama method, 413  
**extern**, 208  
**false**, 22  
fast Fourier transform, 73, 82, 442  
**fclose**, 46  
FFT, *see* fast Fourier transform  
FFTW, 82, 139, 210, 226, 297  
**fftw\_alloc\_complex**, 83, 215  
**fftw\_alloc\_real**, 215  
**fftw\_complex**, 83, 86  
**fftw\_destroy\_plan**, 84, 140, 214  
**fftw\_execute**, 84, 86, 140, 213  
**fftw\_free**, 84, 140, 214  
**fftw\_malloc**, 83, 84, 86, 140  
**fftw\_mpi\_local\_size\_2d**, 215  
**fftw\_mpi\_plan\_dft\_2d**, 214  
**fftw\_mpi\_plan\_dft\_c2r\_2d**, 214  
**fftw\_mpi\_plan\_dft\_r2c\_2d**, 214  
**fftw\_plan**, 85, 86, 90, 140, 145, 213

- `fftw_plan_dft_1d`, 84  
`fftw_plan_dft_2d`, 90, 145  
`fftw_plan_dft_c2r_1d`, 86  
`fftw_plan_dft_r2c_1d`, 86  
file dependencies, 4  
`firstprivate`, 106, 112, 128, 133  
`fopen`, 46, 55  
Fourier transform, 441  
`fprintf`, 46  
`fread`, 49  
`free`, 33, 36, 64  
`fscanf`, 46, 47  
`fseek`, 347  
`ftell`, 347  
function declaration, 59  
function definition, 18, 60  
`fwrite`, 49  
  
`g++`, 3  
Gaussian distribution, 412, 413, 416  
`gcc`, 2, 3, 5, 6, 73, 75, 78, 87  
`gdb`, 6  
generator counter, 387  
generator key, 387  
`get_global_id`, 343, 353  
`get_global_size`, 353  
`get_group_id`, 355  
`get_local_id`, 355  
`get_local_size`, 355  
`get_num_groups`, 355  
`GetKernelSource`, 347  
`gettimeofday`, 4, 66, 70  
ghost point, 421, 438  
Ginsburg–Landau equation, 445  
`git`, 9  
global, 329  
global memory, 18, 233, 238, 249–251, 256, 261, 262, 274, 282, 314, 315, 317, 362, 402, 404  
global scope, 256  
`global_work_offset`, 342  
`global_work_size`, 342  
`gprof`, 7  
GPU memory, 249, 357  
grid communicator, 189, 191, 192, 219  
`gridDim`, 242, 245  
  
head node, 159  
header file, 17, 18, 24, 36, 59, 61, 62, 69  
heat equation, 420, 422, 424  
`helper_cuda.h`, 246  
host, 231  
`ifdef`, 291, 392  
`initcheck`, 237  
`initstate_r`, 144  
integer arithmetic, 25, 26, 54  
interleaved format, 396  
  
job, 159  
job script, 155  
  
kernel, 232, 239, 242, 244, 249–251, 254, 256, 258, 269–271, 273–275, 281, 282, 284, 327, 342  
kernel file, 343  
Kernel Queue, 274  
key code, 41, 43, 44  
  
lanes, 261  
`LAPACK`, 74, 75, 77, 78, 93, 94, 139, 148, 197, 200, 201, 208, 222, 287, 300, 302, 307, 308, 423  
`LAPACK_COL_MAJOR`, 75  
`LAPACK_ROW_MAJOR`, 75  
`LAPACKE`, 75  
`LAPACKE_dgesv`, 75, 78, 81, 89, 90  
`LAPACKE_dgetrf`, 81, 89, 90, 94  
`LAPACKE_dgetrs`, 81, 89, 90, 94  
`LAPACKE_dgttrf`, 81, 90, 93  
`LAPACKE_dgttrs`, 81, 90, 93  
`lastprivate`, 112, 133  
linker, 73, 74, 78  
linking, 3  
Linux, 1, 2, 6, 7, 103  
load balancing, 109  
local, 357  
local memory, 250, 314, 315, 357, 362, 402, 403  
local variable, 63, 133  
log-normal distribution, 295  
  
`M_PI`, 45  
MAC grid, 437  
machine epsilon, 23  
macro, 18, 22, 23, 247, 291, 350  
`MAGMA`, 287  
`magma_dgesv`, 287, 289  
`magma_dgesv_gpu`, 291, 292, 393  
  
`magma_dgetmatrix`, 291, 292, 393  
`magma_dgetrf_gpu`, 291, 292, 316, 392, 393, 403  
`magma_dgetrs_gpu`, 291, 292, 316, 392, 393, 403  
`magma_dmalloc`, 291  
`magma_dmalloc_cpu`, 291  
`magma_dmalloc_pinned`, 288  
`magma_dsetmatrix`, 291, 393  
`magma_finalize`, 289  
`magma_free`, 292, 393  
`magma_free_pinned`, 289  
`magma_init`, 288, 290  
`MagmaConjTrans`, 292, 393  
`magmaDouble_ptr`, 393  
`MagmaNoTrans`, 292, 393  
`MagmaTrans`, 292, 393  
`main`, 15  
`make`, 4  
`makefile`, 4  
`malloc`, 33, 36, 64, 83  
marker-and-cell method, 437  
`math.h` header file, 29  
Memory Copy Queue, 274  
memory leak, 7, 33, 36, 64  
`memset`, 50, 234  
Mersenne Twister, 87, 294  
Moab, 155, 160  
Monte Carlo integration, 414  
MPI libraries, 197  
`MPI_Allgather`, 177, 210, 222, 224  
`MPI_Allreduce`, 179, 225  
`MPI_Bcast`, 186, 188, 192, 217  
`MPI_Cart_coords`, 191  
`MPI_Cart_create`, 191, 192  
`MPI_Cart_shift`, 191  
`MPI_Cart_sub`, 192  
`MPI_Comm`, 183, 186, 188  
`MPI_COMM_NULL`, 186  
`MPI_Comm_rank`, 158, 186  
`MPI_Comm_set_errhandler`, 180  
`MPI_Comm_size`, 158, 186  
`MPI_Comm_split`, 186, 192  
`MPI_COMM_WORLD`, 164, 183, 186, 191, 214  
`MPI_Finalize`, 202, 209  
`MPI_Gather`, 176–178, 182, 196, 214, 217, 219–221, 226  
`MPI_Gatherv`, 177, 182, 225, 226

- MPI\_Group**, 183, 186  
**MPI\_Group\_free**, 185  
**MPI\_Group\_incl**, 184, 185  
**MPI\_Init**, 158, 172, 198  
**MPI\_Irecv**, 173, 175  
**MPI\_Isend**, 173, 175  
**MPI\_Recv**, 164, 165, 171, 175, 182  
**MPI\_Reduce**, 179, 182, 217  
**MPI\_Request**, 173  
**MPI\_Scatter**, 182, 211, 219, 223  
**MPI\_Scatterv**, 178, 182  
**MPI\_Send**, 164, 182  
**MPI\_Sendrecv**, 174, 175, 225  
**MPI\_Status**, 164, 173, 174  
**MPI\_STATUS\_IGNORE**, 164, 165  
**MPI\_Test**, 173  
**MPI\_Wait**, 173  
**MPI\_Wtick**, 194  
**MPI\_Wtime**, 180, 194, 217  
**mpicc**, 158, 159  
**mpirun**, 159, 161  
multithread region, 101, 125
- Neumann boundary condition, 421  
normal distribution, 295  
**nowait**, 119, 120, 128, 134  
Nsight, 1  
**nvcc**, 229–231, 287, 289  
**nvprof**, 235, 238  
**nvvp**, 235
- object file, 60  
**omp\_atomic**, 136–138  
**omp\_barrier**, 127, 128  
**omp\_critical**, 137, 138  
**omp\_ordered**, 114  
**omp\_section**, 133  
**omp\_sections**, 131, 133, 134  
**omp\_single**, 125, 126, 128, 130  
**omp\_get\_max\_threads**, 103, 141  
**omp\_get\_num\_threads**, 102  
**omp\_get\_thread\_num**, 102  
**omp\_get\_wtime**, 115  
OpenCL error codes, 348  
ordinary differential equation, 411  
parallel programming, 99
- parallel region, 103–106, 112, 120, 125, 127, 128, 130, 133, 134, 143, 148  
partial differential equation, 419–422, 433, 441, 444  
pass by reference, *see* reference, pass by  
pass by value, *see* value, pass by  
**pdgesv\_**, 197, 201  
periodic boundary condition, 441  
pinned memory, 289, 291  
pointer, 30, 32, 33, 36, 37, 46, 48, 55, 62–64  
Poisson distribution, 295  
polar Marsaglia method, 413  
**pow**, 25  
**pragma**, 102  
preprocessor, 247, 350  
**printf**, 18, 19, 34, 41–44, 46  
**private**, 99, 106, 111  
private variable, 104–107, 109, 111–113, 125, 128, 133, 136, 142  
process grid, 198, 200–202, 208  
pseudorandom numbers, 292  
quasi-random numbers, 292  
queuing system, 159  
race condition, 136  
**racecheck**, 237  
**rand**, 87  
**random**, 87, 141, 144, 295  
random number generator, 87, 89, 93  
**Random123 library**, 383  
**random\_r**, 144  
rank, 158  
red/black ordering, 94, 149, 192, 224, 225, 318, 435  
reduction, 179, 262, 313, 318  
**reduction**, 99, 106, 107, 113, 127, 133, 136, 147, 149  
reference, pass by, 62  
repository, 9  
row-major, 31, 75, 82, 219, 222, 292, 353, 393, 397  
Runge–Kutta method, 318, 406, 422, 427, 444–446  
**scanf**, 41, 42, 44, 47, 49  
**schedule**, 114, 116, 118  
scheduler, 159  
scope of a function, 60  
seed, 87, 107, 142, 144, 145, 217, 292–295, 297, 409, 415, 416, 418, 426, 445, 446  
segmentation fault, 6, 233  
serial programming, 99  
**shared**, 105, 106  
shared memory, 234, 249–252, 261, 274, 282–285, 315, 403  
shared memory architectures, 99  
shared variable, 100, 104, 105, 107, 109, 110, 125–127, 133–136  
shell script, 160  
single region, 128, 129  
single thread region, 126  
**sizeof**, 33, 48  
**sleep**, 116  
Sobol quasi-random generator, 294  
**SOR**, *see* successive overrelaxation method  
spectral approximation, 442  
spectral coefficient, 441  
spectral derivative, 85, 300, 397, 444  
spectral method, 443  
spectral space, 443  
**srand48**, 87, 93  
**srand48\_r**, 144  
**random\_r**, 144  
static libraries, 73  
static memory allocation, 30, 31, 33  
**stdbool.h**, 22  
Stokes flow, 436  
stream, 270  
Streams, 269  
string, 18, 34  
strong scaling, 120, 194  
strongly convergent method, 412  
**struct**, 36  
successive overrelaxation method, 434, 435, 437–439  
Sunway TaihuLight, xii, 155, 193, 196  
**switch-case**, 54  
**synccheck**, 237
- target, 4, 5, 10  
**tex1Dfetch**, 259  
texture memory, 249, 256

thread, 101, 102, 104–107, 234, 245, 246, 250, 251, 261–263, 273, 283  
thread-safe, 139  
**threadIdx**, 245, 261  
ThreeFry4x32, 383  
**time**, 66, 69, 294  
**true**, 22  
type cast, 25  
**typedef**, 24, 37, 396  
**unsigned**, 42  
**urandom**, 50, 87  
**valgrind**, 7, 8, 24, 31, 237  
value, pass by, 62  
**vi**, 1  
**vim**, 1  
warp, 238, 252, 261, 262, 265, 283, 314, 315  
warp shuffle, 261, 283, 314, 315  
weak scaling, 120, 194  
weakly convergent method, 412  
**while loop**, 54  
Wiener process, 411, 412  
work-group, 353, 354, 402, 403  
work-item, 342, 353, 354, 402  
**work\_dim**, 342  
**xemacs**, 1  
zero-based, 31

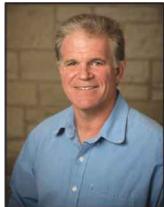
Based on a course developed by the author, *Introduction to High Performance Scientific Computing* introduces methods for adding parallelism to numerical methods for solving differential equations. It contains exercises and programming projects that facilitate learning as well as examples and discussions based on the C programming language, with additional comments for those already familiar with C++.

The text provides an overview of concepts and algorithmic techniques for modern scientific computing and is divided into six self-contained parts that can be assembled in any order to create an introductory course using available computer hardware.

- Part I introduces the C programming language for those not already familiar with programming in a compiled language.
- Part II describes parallelism on shared memory architectures using OpenMP.
- Part III details parallelism on computer clusters using MPI for coordinating a computation.
- Part IV demonstrates the use of graphical programming units (GPUs) to solve problems using the CUDA language for NVIDIA graphics cards.
- Part V addresses programming GPUs for non-NVIDIA graphics cards using the OpenCL framework.
- Finally, Part VI contains a brief discussion of numerical methods and applications, giving the reader an opportunity to test the methods on typical computing problems.

*Introduction to High Performance Scientific Computing* is intended for advanced undergraduate or beginning graduate students who have limited exposure to programming or parallel programming concepts. Extensive knowledge of numerical methods is not assumed. The material can be adapted to the available computational hardware, from OpenMP on simple laptops or desktops to MPI on computer clusters or CUDA and OpenCL for computers containing NVIDIA or other graphics cards. Experienced programmers unfamiliar with parallel programming will benefit from comparing the various methods to determine the type of parallel programming best suited for their application.

The book can be used for courses on parallel scientific computing, high performance computing, and numerical methods for parallel computing.



**David L. Chopp** is a professor in the Northwestern University Engineering Sciences and Applied Mathematics Department, where he has been teaching since 1996 and has been chair since 2013. He was named a Charles Deering McCormick Professor of Teaching Excellence in 2008. Chopp has developed multiple courses and is the author of nearly 50 refereed publications, including fundamental contributions to the development of the popular level set method for computing moving interfaces. His research interests include numerical methods and mathematical modeling in applications such as microbiology, materials science, fracture mechanics, and neurobiology.

---

For more information about SIAM books, journals, conferences, memberships, or activities, contact:

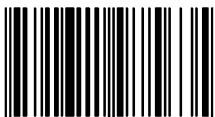


Society for Industrial and Applied Mathematics  
3600 Market Street, 6th Floor  
Philadelphia, PA 19104-2688 USA  
+1-215-382-9800 • Fax +1-215-386-7999  
[siam@siam.org](mailto:siam@siam.org) • [www.siam.org](http://www.siam.org)

SE30



ISBN 978-1-611975-63-5



9781611975635