

Introduction to High Performance Scientific Computing

D. L. Chopp

April 30, 2018

Contents

Preface	ix
1 Tools of the Trade	1
1.1 Text Editors	1
1.2 Compilers	2
1.3 Makefiles	3
1.4 Debugging	4
Exercises	5
I Elementary C Programming	7
2 Structure of a C Program	11
2.1 Comments	12
2.2 The Preamble	13
2.3 The Main Program	14
Exercises	15
3 Data Types and Structures	17
3.1 Basic Data Types	17
3.2 Mathematical Operations and Assigning Values	20
3.3 Boolean Algebra	22
3.4 Arrays and Pointers	22
3.5 Structures	28
Exercises	29
4 Input and Output	31
4.1 Terminal I/O	31
4.2 File I/O	34
Exercises	37
5 Flow Control	39
5.1 For Loops	39
5.2 While and Do-While Loops	41
5.3 If-Then-Else	42
5.4 Switch-Case Statements	44
Exercises	45
6 Functions	47

6.1	Declarations and Definitions	47
6.2	Function Arguments	50
6.3	Measuring Performance	52
	Exercises	54
7	Using Libraries	57
7.1	BLAS and LAPACK	58
7.2	FFTW	65
7.2.1	Complex numbers in C	66
7.2.2	1D serial FFT	67
7.2.3	Building code with FFTW	69
7.3	Random Number Generation	70
	Exercises	72
8	Projects for Serial Programming	75
8.1	Random Processes	75
8.1.1	Monte Carlo Integration	75
8.1.2	Duffing-Van der Pol Oscillator	75
8.2	Finite Difference Methods	75
8.2.1	Brusselator Reaction	75
8.2.2	Linearized Euler Equations	76
8.3	Elliptic Equations and SOR	76
8.3.1	Diffusion with Sinks and Sources	76
8.3.2	Stokes Flow 2D	76
8.3.3	Stokes Flow 3D	76
8.4	Pseudo-Spectral Methods	76
8.4.1	Complex Ginsburg-Landau Equation	76
8.4.2	Allen-Cahn Equation	76
II	Parallel Computing Using OpenMP	77
9	Intro to OpenMP	81
9.1	Creating Multiple Threads: #pragma omp parallel	81
9.2	The num_threads() clause	83
9.3	The shared() clause	84
9.4	The private() clause	85
9.5	The reduction() clause	87
	Exercises	88
10	Subdividing For-Loops	89
10.1	Subdividing Loops: #pragma omp for	89
10.2	The private() clause	91
10.3	The reduction() clause	92
10.4	The ordered clause	93
10.5	The schedule clause	94
10.6	The nowait clause	99
	Exercises	101
11	Serial Tasks Inside Parallel Regions	103
11.1	Serial subregions: #pragma omp single	103

11.2	The copyprivate clause	106
Exercises		108
12	Distinct Tasks in Parallel	109
12.1	Multiple Parallel Distinct Tasks: #pragma omp sections	109
12.2	The private() clause	111
12.3	The reduction() clause	111
12.4	The nowait clause	112
Exercises		112
13	Critical and Atomic Code	113
13.1	Atomic Statements	113
13.2	Critical Statements	115
Exercises		116
14	OpenMP Libraries	117
14.1	FFTW	117
14.2	Random Numbers	119
15	Projects for OpenMP Programming	125
15.1	Random Processes	125
15.1.1	Monte Carlo Integration	125
15.1.2	Duffing-Van der Pol Oscillator	125
15.2	Finite Difference Methods	126
15.2.1	Brusselator Reaction	126
15.2.2	Linearized Euler Equations	126
15.3	Elliptic Equations and SOR	126
15.3.1	Diffusion with Sinks and Sources	127
15.3.2	Stokes Flow 2D	128
15.3.3	Stokes Flow 3D	128
15.4	Pseudo-Spectral Methods	128
15.4.1	Complex Ginsburg-Landau Equation	128
15.4.2	Allen-Cahn Equation	128
III	Distributed Programming and MPI	129
16	Preliminaries	133
16.1	Submitting Jobs	133
16.2	Parallel Hello World!	136
16.3	Compiling and Running MPI Code	137
Exercises		138
17	Passing Messages	139
17.1	Blocking Send and Receive	139
17.2	Non-Blocking Send and Receive	142
17.3	Gather/Scatter Communications	150
17.4	Broadcast and Reduction	153
Exercises		154
18	Groups and Communicators	157

18.1	Subgroups and Communicators	157
18.2	Communicators Using Split	161
18.3	Grid Communicators	163
	Exercises	167
19	Measuring Efficiency and Checkpointing	169
19.1	Efficiency Measures	169
19.2	Checkpointing	171
20	MPI Libraries	173
20.1	ScaLAPACK	173
20.1.1	Setting up the process grid	174
20.1.2	Distributing the matrix	174
20.1.3	Calling the ScaLAPACK routine	177
20.1.4	Release the process grid	177
20.1.5	Description of Example 20.1	183
20.1.6	Compiling and linking with ScaLAPACK	186
20.2	FFTW	186
20.2.1	Using Serial FFTs in parallel	186
20.2.2	Distributed FFTW	187
21	Projects for Distributed Programming	191
21.1	Random Processes	191
21.1.1	Monte Carlo Integration	191
21.1.2	Duffing-Van der Pol Oscillator	191
21.2	Finite Difference Methods	192
21.2.1	Domain Decomposition and MPI	192
21.2.2	ADI and MPI	193
21.2.3	Brusselator Reaction	195
21.2.4	Linearized Euler Equations	195
21.3	Elliptic Equations and SOR	195
21.3.1	Diffusion with Sinks and Sources	196
21.3.2	Stokes Flow 2D	196
21.3.3	Stokes Flow 3D	197
21.4	Pseudo-Spectral Methods	197
21.4.1	Complex Ginsburg-Landau Equation	197
21.4.2	Allen-Cahn Equation	197
IV	GPU Programming and CUDA	199
22	Intro to CUDA	203
22.1	First CUDA Program	203
22.2	Selecting the Correct GPU	205
23	Parallel CUDA Using Blocks	207
23.1	Running Kernels in Parallel	207
23.2	Organization of Blocks	210
23.3	Threads	213
23.4	Error Handling	213

24 GPU Memory	217
24.1 Shared Memory	217
24.2 Constant Memory	220
24.3 Texture Memory	222
24.3.1 Texture memory and double precision	225
25 Streams	229
25.1 Stream creation and destruction	229
25.2 Asynchronous memory copies	230
25.3 Synchronizing streams	231
25.4 Single stream example	231
25.5 Multiple streams	234
25.6 General strategies	239
25.7 Measuring Performance	240
Exercises	242
26 CUDA Libraries	243
26.1 MAGMA	243
26.1.1 Simple example of a linear solve	243
26.1.2 Compiling and linking MAGMA code	245
26.1.3 CPU interface vs. GPU interface	246
26.2 cuRAND	248
26.2.1 Host interface	249
26.2.2 Device interface	251
26.3 CUDA FFT	253
26.3.1 Compiling and linking with cufft library	256
26.4 cuSPARSE	256
26.4.1 Sparse formats	257
26.4.2 Matrix Vector Multiplication	259
26.4.3 Tri-diagonal solver	262
26.4.4 Linking the library	266
Exercises	266
27 Projects for CUDA Programming	267
27.1 Random Processes	267
27.1.1 Monte Carlo Integration	267
27.1.2 Duffing-Van der Pol Oscillator	268
27.2 Finite Difference Methods	268
27.2.1 Brusselator Reaction	269
27.2.2 Linearized Euler Equations	269
27.3 Elliptic Equations and SOR	269
27.3.1 Diffusion with Sinks and Sources	269
27.3.2 Stokes Flow 2D	269
27.3.3 Stokes Flow 3D	270
27.4 Pseudo-Spectral Methods	270
27.4.1 Complex Ginsburg-Landau Equation	270
27.4.2 Allen-Cahn Equation	270

V GPU Programming and OpenCL	271
28 Intro to OpenCL	275
28.1 First OpenCL Program	275
28.2 Selecting the Correct GPU	278
29 Parallel OpenCL Using Workgroups	281
29.1 Parallel OpenCL Example	281
29.2 Compiling Kernel Functions	285
29.3 Organization of Workgroups	290
29.4 Work-Items	293
29.5 Error Handling	294
30 GPU Memory	297
30.1 Local memory	297
30.2 Constant Memory	302
31 Command Queues	307
31.1 Using Multiple Devices	307
31.2 Measuring Performance	311
Exercises	313
32 OpenCL Libraries	315
32.1 clMAGMA	315
32.1.1 Simple example of a linear solve	315
32.1.2 Compiling and linking clMAGMA code	317
32.1.3 CPU interface vs. GPU interface	318
32.2 clFFT	320
32.2.1 Compiling and linking with c1FFT library	324
32.3 Random123	324
Exercises	327
33 Projects for OpenCL Programming	329
33.1 Random Processes	329
33.1.1 Monte Carlo Integration	329
33.1.2 Duffing-Van der Pol Oscillator	329
33.2 Finite Difference Methods	330
33.2.1 Brusselator Reaction	331
33.2.2 Linearized Euler Equations	331
33.3 Elliptic Equations and SOR	331
33.3.1 Diffusion with Sinks and Sources	331
33.3.2 Stokes Flow 2D	331
33.3.3 Stokes Flow 3D	331
33.4 Pseudo-Spectral Methods	331
33.4.1 Complex Ginsburg-Landau Equation	332
33.4.2 Allen-Cahn Equation	332
VI Applications	333
34 Stochastic Differential Equations	337

34.1	Mathematical description	337
34.2	Numerical Methods	339
34.2.1	Euler-Maruyama Method	339
34.2.2	Box-Muller and Polar Marsaglia Methods	339
34.3	Problems to Solve	340
34.3.1	Monte Carlo Integration	340
34.3.2	Duffing-Van der Pol Oscillator	342
35	Finite Difference Methods	345
35.1	Approximating Spatial Derivatives	345
35.2	Finite Difference Grid	346
35.2.1	Explicit Method	346
35.2.2	Implementation Notes	347
35.2.3	Implicit Methods	348
35.2.4	Alternating Directions Implicit Method	349
35.3	Problems to Solve	351
35.3.1	Brusselator Model	351
35.3.2	Linearized Euler Equations	351
36	Iterative Solution of Elliptic Equations	355
36.1	Problems to Solve	357
36.1.1	Diffusion With Sources/Sinks	357
36.1.2	Stokes Flow	358
37	Pseudo-Spectral Methods	363
37.1	Fourier Transform	363
37.2	Spectral Differentiation	365
37.3	Pseudo-Spectral Method	366
37.4	Higher Dimensions	366
37.5	Problems to Solve	367
37.5.1	The Complex Ginsburg-Landau Equation	367
37.5.2	Allen-Cahn Equation	368
	Bibliography	371
	Index	373

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 ten week course put together by the author at Northwestern University. It is assumed that the reader understands some elementary programming concepts, such as writing scripts in Matlab. Each part of this book is intended to be largely independent of the others so that the parts can be chosen so as to tailor an introductory course based on the available computer hardware. For example, a ten 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, or substitute Part V on OpenCL in place of CUDA. Using OpenMP in place of MPI makes sense 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, but if that is not uniformly available, you could opt for OpenCL instead which will work on a more diverse set of graphics cards. I cover Parts I, III, and IV when I teach this in a ten week course. For a fifteen week course I would suggest Parts I-III plus either IV or V and adding a little more time devoted to the applications in Part VI.

Part I of this book is a crash-course on C programming to get everyone up to speed for what will come later. For some this is old news (though you may learn a few new things or pick up better habits), and for some it is entirely new. We will use C as the main language because it is commonly available, widely used, and yet simple enough in structure that it is easy to learn. This book provides a very barebones description of what you need to get started. 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.

Part II of this book introduces parallel programming for shared memory machines using OpenMP. OpenMP is no doubt 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. Not only that, but it requires very little modification to the algorithms you would normally write for a single core implementation, often only the addition of some compiler directives to take advantage of opportunities for parallelism. Since this requires no special hardware to implement, it's a natural first step to 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 computer has 679 nodes with a total of 16,028 cores. By comparison, as of this writing, the current fastest supercomputer is 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, we must write programs that can break the task into smaller pieces that each core can do simultaneously. These many subtasks have to be coordinated to get a final answer, and this is accomplished by enabling *message passing* between cores. MPI programming is one method of message passing between cores.

Part IV addresses growing interest in high performance computing using graphics cards, also knowns as GPUs. The video gaming industry is approaching \$20 billion dollar 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 Tesla K80 GPU has 4,992 cores. So where the CPUs individually have more capability, GPU cards make up for it by large numbers. For people doing scientific computing, it is often the case that we may be updating an array of data many times using the same basic formula over and over. By doing those tasks simultaneously, and quickly, 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 V introduces OpenCL, which is a companion to Part IV for people that do not have NVIDIA graphics cards, or perhaps do have NVIDIA cards but want to develop something that is 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 does not have quite the polish that CUDA has, nor quite as extensive of support libraries, it is still a very viable means of programming GPUs. And without an NVIDIA card, it's pretty much your only option. This part is in the book to provide support for those in that situation.

In my experience, there's no substitute for learning material than to build something with it. Part VI discusses some representative scientific computing tasks 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 so as 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 each part.

Note to instructors: Details about the development environment such as the choices for text editors, variation in compilers, debugging tools, makefile environments, code library locations, job submission procedures, etc. are not discussed in this book, but you as the instructor should be sure to know these things beforehand. When I teach this course, I discuss these because students who have not programmed significantly before need this kind of basic instruction. In this book, I'm assuming a basic Linux setup that uses `gcc` as a C compiler, `mpicc` for the MPI compiler, and `nvcc` for the CUDA compiler. Your local system may differ.

Chapter 1

Tools of the Trade

In order to code development, you need 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 way to organize 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.

Development environments such as Eclipse [6], the Windows Development Environment [7], and XCode [8] all have integrated edit/compile/debug tools that are nice to use. These are just a few of the most popular development environments available. Covering each of these environments is beyond the scope of this book. Check with the documentation for your system to learn how to interface with the editor, compilers, and debug tools that are supported. The tools presented in this chapter assume you don't have access to one of these specialized development environments.

1.1 • Text Editors

In order to write, edit, and run the codes, you will need a text editor. 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 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/>

1.2 ▪ Compilers

On Linux platforms, a commonly used open-source compiler is gcc. Compiling programs can be done in either one stage or two depending on how you're doing it. When compiling programs, the compiler assumes your program has the correct suffix: all the code you write should go in either header files, which end with the suffix ".h", or code files, which end with ".c". We will explain more about the distinction between these files later, but only the code files get compiled when using the compiler, the header files are included semi-automatically in the process.

Suppose your program consists of two code files, `main.c` and `func.c`. The program can be compiled into an executable by entering the following command at the prompt:

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

where here the \$ symbol is the 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 everything is OK, then this command will result in a new executable binary program called `a.out`. If you wish to name your program something a little more intuitive, e.g. `myprog`, then the output binary can be renamed by using the `-o` option:

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

Often, you will have many code files to be compiled and you may not want to compile all of them every time you wish to build your program. For large codes compiling can take a little while and redoing everything because you made a small change in one file just doesn't make sense. In that case, only the updated file has to be compiled and then joined to the other 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 multi-step process:

```
$ 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 `.o` files are called object files and they consist of compiled code corresponding to the code file of the same name. The object files are self-contained except for possibly some loose ends that connect to other object files when the program is fully assembled. The last step above is what connects all the loose ends together, and is called *linking*. It is at this stage that external libraries are also included when needed. Some external libraries are included by default. For example, the commands for doing I/O 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.

1.3 ■ 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 you make a change in some files, but not all, that requires rebuilding your program. You could follow the steps above, but that requires remembering which files you edited, typing in the commands to build those object files, and then linking them all together. This is a process that is prone to mistakes, so to make this 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. You may find that makefiles are very useful for many other tasks when working in Linux.

To use `make`, create a file called `makefile`. Alternatively, you can also name it `Makefile`, but any other name would require explicitly typing in the name of the makefile, so let's stick with `makefile`. Following our compiling example above, suppose there is also a header file `func.h` that corresponds to the code file `func.c`. 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.

To understand what this file does, we'll walk through the thought process that happens when the `make` command is issued. To begin, we decide which target we wish to make. The targets are the expressions that end with a colon and begin at the left, and are the products that we want to make. To build our program, we can either simply type `make` at the command prompt, which is equivalent to making the first target, or `make myprog` where we are specifying which target we want to build. Suppose we just want to compile `main.c`, then we could type `make main.o`. 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 you want to quickly remove stray data files that are generated during execution of your program. In that case, you may want to rebuild everything from scratch, and a way to force that would be to have this target in your `makefile` and then type `make clean`.

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 of dependency is checked to see if it is also a target. If a dependent file is a target, then a `make` command is generated for each 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. 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 are a target in the makefile, then 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 back to our original target `myprog`, the timestamp of `myprog` is compared to `func.o` and `main.o`. Again, `func.o` was recently updated, so 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? By carefully constructing a make file, a single command is all you have to do to bring your program up to date in the most efficient manner. You don't have to keep track of which files were modified and require recompiling because `make` will check the time stamps 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. All you have to do is type `make` at the command prompt, and all the compile commands are issued automatically.

The description of makefiles here is quite simple, and they are capable of much more sophisticated techniques. To learn more, check out this website:

<http://www.gnu.org/software/make/manual/make.html>

1.4 • Debugging

Assuming you are a human, it is unlikely every program you write will work on the first attempt. Once you get past the errors caught by the compiler, there are the errors that occur during runtime. You could, of course, carefully pore through your 100,000 lines of code to find where that accidental typo is located, but that is unlikely to be fruitful. Alternatively, you could use strategically placed print statements in your program to report when a certain line of code is passed or to print the values of variables at critical junctures that may help explain the error you are trying to understand. However, a more efficient means of squashing bugs is to turn to a debugging tool.

A debugger allows you to step through your code while it's running so that you can check the values of variables, follow the sequence of commands, etc. so that it is easier to see what is happening. If you hit a segmentation fault or other unexpected termination, a debugger can show you where in the code it happened and hopefully give you a clue as to why.

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

```
$ 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 your program with the debugger, type:

```
$ gdb myprog
```

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

```
(gdb) run myargs
```

where `myargs` are the arguments you would give to your program to run it on the command line. If you want it to stop at a particular line in the code, then before running the program type

```
(gdb) break 123
```

to stop the debugger at line 123 of your file. You can then step through your program one line at a time by using the `next` command or the `step` command to follow the progress through the program more carefully.

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 also be found at this website:

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

There is a nice graphical front end for `gdb` called `ddd`. If you are on a Linux system using X11 windows, then `ddd` is the way to go. 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/>

Exercises

- 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 being 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 we will focus here on C because it is widely available, 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 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 variable 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 multi-type structures. Much of scientific programming revolves around computing operations on arrays, so this is an important section to understand.

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

Operations on arrays require a robust understanding of loops for flow control, so in Chapter 5, the basic program flow control structures are introduced that are for creating loops and for branching according to a given condition.

Trying to fit a large calculation into a single file is simply not practical in a real setting. Thus, in Chapter 6, the fundamental building block of functions are introduced.

The whole purpose of this textbook is to teach how to speed up mathematical computations. To appreciate how effective these techniques are, we will need some means of measuring the performance. In other words, how long does it take to solve a problem? In Chapter ??, a brief discussion is given for how to measure the time required to solve a problem. This will be expanded in other parts of the book.

Of course you plan to do original research, but there's no reason you must start from scratch. Instead, 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 algebra solver or Fourier transform function when others have already done it, tested it, and optimized it. Therefore, in Chapter 7, some of the standard libraries of code available for solving linear algebra problems, taking Fourier transforms, and generating random numbers are discussed. 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, in Chapter 8, the instructions for doing some sample projects are given. The details about the implementation of the various projects are provided in Part VI. The projects are an opportunity to test your abilities in C programming by solving some representative scientific computing applications. Compare your results for these projects with implementations from other parts of this book to appreciate the im-

provement in performance that parallel computing provides.

Chapter 2

Structure of a C Program

If you are already familiar with a programming language, then you will no doubt find many parallels between C and whatever language you already know. Languages such as C++ and Java have very strong similarities, even including much of the syntax, but all languages need to know where the program starts and ends. Another common feature in programs is inserting comments. Commenting your program is absolutely essential. It is too easy to finish a program and set it aside only to return to it months or even years later and then have no clue what the program is supposed to be doing even though you wrote it. Even more challenging if someone else wrote it and you are trying to understand it. 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. Below is a simple example (with many comments) that accomplishes the simple task of printing “Hello World!”. It illustrates the main pieces required for C programs. We'll break it down line by line.

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 are 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 non-zero 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 }
```

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 your programs is essential. When you go back to reuse your code, or remind yourself how something worked, you will be eternally grateful for the information. So make them 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 multi-line comments. Lines 15–19 illustrate another form of comments. Any text following “`//` *on the same line* are also comments. These are best for short one or two line comments. In truth, either method of using comments is perfectly fine, as long as you **always comment your code!**

Example 2.1 illustrates three key places where you should routinely put comments in your code. First, at the top of each file you should include information about the contents of the file like those included on Lines 1–12. This may include descriptive information, authorship, and editing history. Second, at the beginning of each function, comments should be provided that describe the function, what inputs the function requires and what outputs are expected. Lines 23–45 illustrate the comments related to the `main` function that begins on Line 47. Third, each substantive step your program should receive a comment so that you or others can follow the logic of the program. This first example is too simple to provide a good example, but Line 54 shows one example. In the remainder of this book, the first type of comment will mostly be intentionally omitted in the code examples for the sake of brevity. This should not indicate 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, we don’t want to have to write the instructions for how to translate text from the program to the screen, or from the keyboard to the input. To include these libraries and pre-built functions, there are two tasks required: (1) the input and output parameters of the functions must be declared, and (2) the precompiled code stored in the library must be added to the final binary. The latter topic is a little more advanced, so we will discuss that later. It should suffice for now that many basic library functions are automatically linked to our final code, so no additional actions are required.

The declaration of functions must always be done. 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 declarations required for the associated library. All that information can be included into our 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 gives the function declarations relevant to text input/output commands for the terminal window. Thus, the function `printf` on Line 52 will not work without including this file. 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.

You may write header files for our programs as well. For example, suppose you write a function that multiplies two matrices together. You may want to reuse that function again in the future for other purposes, so you may create a file `matrixops.c` that contains functions for doing matrix operations. To make use of it in a program, you would create a file `matrixops.h` that contains the function declarations in your `matrixops.c` file, and in your main program, include the file `matrixops.h`. For header files that you write, the file name should be inside double quotes in the compiler directive, like this:

```
#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, type definitions, among other things, all are 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 will be the same for every C program you will ever write. 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 is an integer named `argc`, and the second is an array of strings named `argv`. You don't have to use these names, but it's generally the way main programs are set up so there's no advantage to changing them. We will discuss the typing of variables and construction of arrays 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 at the command line we issued the command:

```
$ hello there my friend <return>
```

then `argc` will have value 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"}.

Note that the function itself also has a type definition of `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 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, if we successfully print "Hello World!", then we will return the value of zero 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 are not part of the main function execution. The main function could contain the entire program, but typically it will involve calls to other functions you write to complete your program. In this simple example, Line 52 is the only line that will actually *do* something. In this case, it prints "Hello World!" in the terminal (without the double-quotes). The function `printf` is used to send formatted text to the terminal window. We will discuss the formatting statement passed to the `printf` function as an argument in Chapter 4.

Finally, note that each line of the program is terminated by a semi-colon. This is a requirement for each line. Any line not terminated by a semi-colon is assumed to be continued on the following line. 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 semi-colon.

Exercises

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

Chapter 3

Data Types and Structures

It's time to expand the range of things our programs can do. A critical component of scientific computing is data storage, retrieval, and organization. The better organized and designed your data is managed, the more efficient your program will be. And after all, the reason we are using high performance computing is because the equations we are trying to solve are *hard*, and require significant effort to compute. In this chapter we will explore the various data types and their properties and limitations. We will also look at elementary mathematical operations such as addition and multiplication, and logical operations like AND and OR. Finally, we'll look at how these data can be organized into arrays and structures.

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 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 you need for your computation, and not just lazily define everything to be a floating point value.

3.1 • Basic Data Types

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`, `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 only applied to decimal values and means that the value will only be non-negative. Assuming the value is non-negative 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 less often. In other words, `signed int` is equivalent to simply using `int`.

Tables 3.1, 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 3.1, 3.3 are usually defined in the system header file `limits.h` in this way:

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

Table 3.1. Table of basic and modified data types in C programming

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 × 10 ^{−38} , 3.402823 × 10 ³⁸]
double	[DBL_MIN, DBL_MAX]	[2.225074 × 10 ^{−308} , 1.797693 × 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 × 10 ^{−4932} , 1.189731 × 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 3.2. Table of basic and modified data types 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.

```
#define INT_MAX 2147483647
```

For some compilers, the macro M_PI is defined to be the number π , but for some it is not. If it is not, you may need to wish to add a macro to define it when you need it, e.g.

```
#define M_PI 3.1415926535897932384626433832795
```

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

Table 3.3. Table of machine epsilon values for different floating point types and their associated macros

To be really safe, you may want to guard against when it is defined:

```
#ifdef M_PI
#else
#define M_PI 3.1415926535897932384626433832795
#endif
```

This tells the compiler to check whether `M_PI` is already defined. If so, then use the system version, but if not, then define it so it can be used later. What happens when you compile is the compiler will watch 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 text with the macro definition, and then compiles the resulting code.

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 seven digits of accuracy, while `long double` offers nineteen digits of accuracy. Choosing the right precision is important when speed is a factor. Using `float` will be faster, but result in less precision, while using `long double` will be much slower, but offers much greater precision. This is a tradeoff that must be weighed in every numerical method code. [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`. 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 multi-threaded 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, so get in the habit of doing it. 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. `Index` is a variable that 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 will issue warnings about use of uninitialized variables when it occurs.

3.2 • Mathematical Operations and Assigning Values

Of course, we wouldn't be bothering with any of this coding if we couldn't actually do any math. We have already seen some assignment statements. The value 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 variable *y*:

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

The basic arithmetic operators are: +, -, *, /, (, and), with order of operations the same as for mathematical expressions. C does not have an exponential operator, so x^2 is not the square of *x*, but is an error. Exponentials must be done using the pow function, discussed later. 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:

```

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

Here, because *y* is an array, then 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.

In addition to the = assignment statement, C provides some shortcuts for other common assignments, namely +=, -=, *=, /=. 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);
```

There is also a conditional assignment that is written like this:

```
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 like this:

```
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 illustrates 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 critical operations are 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 we are checking for whether `n` 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>	false if the values of <code>x</code> and <code>y</code> are equal
<code>x < y</code>	true if the value of <code>x</code> is less than <code>y</code>
<code>x <= y</code>	true if the value of <code>x</code> is less than or equal to <code>y</code>
<code>x > y</code>	true if the value of <code>x</code> is greater than <code>y</code>
<code>x >= 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 > 0) && (x <= 10)</code>	true if $0 < x \leq 10$.
<code>(x <= 0) (x > 10)</code>	true if $x \leq 0$ or $x > 10$.
<code>!(x < 0)</code>	false if $x < 0$.

The newest version of the C language also includes the constant values `true` and `false`, which are equivalent to the values 1 and 0 respectively.

3.4 ▪ Arrays and Pointers

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

arrays in C is a bit involved, so we will discuss it in some detail. The memory for arrays must be allocated before it can be used, and that memory must be contiguous in order for it to work properly. Also, 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 locks the dimensions of the array by requiring the dimensions to be determined at compile time. Dynamically allocated memory requires the use of data types called pointers, so they require a little more work, but the dimensions of the array can be determined at run time, giving your program more flexibility. The latter form is preferred in general, but either method is acceptable depending on your goals.

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 3.3.

```

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 5 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. Take care with loops and other constructs that rely on indices until you are accustomed to this style of indexing.

The order the memory is laid out 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 3.1 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 different than that used for Fortran, for example, which uses a row first storage. The equivalent statement in Fortran would give the transpose of the matrix on the left in Figure 3.1. As we will see 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 the dimensions of the arrays have to be set at *compile time*. That means that the dimensions are written explicitly into your program. Doing convergence studies, for example, would be much more difficult if we always assume fixed dimensions for our grids. Therefore, we also want to understand how to create dynamically allocated arrays. These arrays are allocated during run time, 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

grid[0][0] = 1	grid[0][1] = 2	grid[0][2] = 3	grid[0][0] = 1	grid[0][1] = 2	grid[0][2] = 3
grid[1][0] = 4	grid[1][1] = 5	grid[1][2] = 6	grid[1][0] = 4	grid[1][1] = 5	grid[1][2] = 6
grid[2][0] = 7	grid[2][1] = 8	grid[2][2] = 9	grid[2][0] = 7	grid[2][1] = 8	grid[2][2] = 9

Figure 3.1. Layout of memory in an array. On left is how continuous memory corresponds to entries in a matrix. On right is how the data is laid out in memory space.

computational domain.

Before we can discuss how to construct dynamic arrays, we will have to take a small detour and discuss pointers. They are simple constructs, but they often cause beginning programmers fits because of their perceived abstraction. Every point in memory space has an address. When we talk about 32-bit or 64-bit systems (most modern computers are now 64-bit systems), we are referring to the number of bits used to identify a point in memory space, i.e. its address. A pointer is simply a single variable that 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, we are declaring `floatPointer` to be a pointer variable that points to the type `float`. On Line 14, we are declaring `anotherPointer` to be a variable that has type pointer to `float`. Both locations of `*` are acceptable.

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 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 can also be obtained by de-referencing it. Therefore, we may set `x = &y;` as on Line 28. The ampersand means to take the address of the variable `y`, not its contents. In that case, `dblPointer` points to the first entry in the memory laid out to store `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, we're using 0-based indexing!).

So far, we have only used pointers to access data created statically. To create an array dynamically, we must create an appropriate amount of memory, and then use a pointer to point to the beginning of that memory. In so doing, we have dynamically created an array. Suppose we wish to allocate the `data` array from the above example dynamically, then the equivalent of Line 4 would be:

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

Let's dissect this line. The function `malloc` is used to allocate memory in terms of the number of bytes. We want five floats. C provides a utility function `sizeof` that will tell you how many bytes a variable type will occupy. So we request 5 chunks of memory required to contain a float on this computer. In front of the `malloc` call we have `(float*)`. Putting a type inside parentheses like this is called a *type cast*. The function `malloc` returns a generic memory pointer of type `void*` that knows nothing about the data the memory will hold. By putting the type cast in there, we are converting generic memory to memory that holds an array of floats so that it matches the variable on the left side of the equation. In this case, type casting is not strictly required, but it is a good habit to put type casting in when you are mixing different types in an equation.

After any memory is allocated dynamically, you are also responsible for de-allocating it or it will result in a *memory leak*, which is where memory is requested and then not returned to the system. That memory is unusable until the program terminates. For small short programs, memory leaks are unlikely to be a problem, but as our programs start squeezing the available memory for every ounce of space, careful memory management is essential if you want 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, our example above where we created the array `data` dynamically should be matched with a corresponding call like this:

```
free(data);.
```

A special kind of array that is useful for communicating text is an array of charac-

ters, also known as a *string*. Strings can also be allocated either statically or dynamically, but there is one extra twist for string variables. Suppose we construct a string like this:

```
char name[5] = "Dave";.
```

As a result of this, the memory pointed to by name will contain ‘D’, ‘a’, ‘v’, ‘e’, ‘/0’. The last character is equivalent to zero. The zero character is essential because it is used by functions like `printf` to know when to stop reading characters. So if you are saving strings of text, make sure you add at least one more memory spot for the zero terminator. For example, Consider the following code snippet:

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

All characters after the ‘/0’ are ignored because they are after the zero terminator.

As you can see, allocation of single dimensional arrays is quite straightforward. Higher dimensional arrays are a bit more problematic. The example below illustrates how to construct a 2D and also a 3D array and two different ways in which the data can be accessed.

Example 3.5.

```
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 numberOfCols = 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 doubly 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 }
```

Let's take this program apart. This is the first example where we make use of the input arguments. If the resulting executable file is called `arrays`, then we would call this program by typing this 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 we can convert decimal numbers from strings 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 our arrays later on.

On Line 29 we allocate a two-dimensional array dynamically. Explaining the way the compiler interprets this line would take too much time, so we'll just focus on the nuts and bolts. For an $M \times N$ 2D array, the proper construction is

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

where `data` is the variable name for the array. It's not the most intuitive way to construct a 2D array, but it is how it was designed in the language. The 3D analog is located on Line 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 3D analog 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 we have the same number of calls to `free` as we have to `malloc`. This ensures that we don't generate a memory leak 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 we've seen before. We will cover those topics more thoroughly later.

3.5 • Structures

Data types can be combined into larger data structures. By combining multiple data types into a single variable it 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 you wish to create a grid, allocated dynamically, then you would want to keep information about the grid together in one place 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, we could create a `struct` for a 2D data set in this way:

Example 3.6.

```
1 struct myGridType {
2     int dim[2];
3     double del[2];
4     double x0, y0;
5     double *dataPtr;
6 } 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 multi-dimensional arrays inside of a `struct`, so we would have to use a simple pointer for keeping track of the memory if we make it part of the `struct`. Here, we've got the dimensions of the data kept in the `dim` variable, the grid spacing with `del`, and the location of the lower left corner of the domain with `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, we could 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, you add a period followed by the variable name inside the struct. Though it probably won't come up often, structs can be nested, and if so, then the dot would be used to traverse down through the hierarchy of structs.

Note that we named the struct with the name `myGridType`. This is optional but is sometimes helpful to keep different structs distinct. If you will be defining many such structs in a given code, you can also turn it into a type name that can be used later. Suppose we wish to make several variables of this struct type, then we can use the `typedef` command like this:

Example 3.7.

```
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 we can construct multiple grids using the same definition. The component entries for each of the two variables, `grid1` and `grid2` are separate and distinct.

Exercises

- 3.1. Suppose you a set of random numbers x_i 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 one 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 3.4. For each below, suppose the given code is also inserted on Line 25 in Example 3.4. Give the value in the variable `z` at the end of the program.

1. 25	<code>z = grid[1][2];</code>
-------	------------------------------

```

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. Create a struct for storing complex values in double precision. Modify Example 3.5 so that it allocates complex values and initializes the two-dimensional array with $\text{array}[i][j]$ the complex value

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

Initialize the three-dimensional array with

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

Note that you will have to separately set the real and complex values.

- 3.5. 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 3.7 and populate its entries where dim should contain the dimensions of the array, and del , x_0 , y_0 are also given on the command line. Use the function `atof(string)` to convert input strings into double precision floating point values.
- 3.6. 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.
- 3.7. 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. $x_i + x_{i+1}$

- 3.8. 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. $(n < 3.5 ? n : 10-n) \geq 3$
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 you can't communicate the results. 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 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, let's 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, if you have two double precision floating point variables `x`, `y` and you want to print them out as a matrix on their own line of text, then you might write a format string like this:

```
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 com-

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

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

mon. If the format string does not contain a “\n”, then the next `printf` will continue printing *on the same line*. This is handy for when you want to print a sequence of numbers on the 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 you’re 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.

The key codes can be further modified with number format modifiers. These are very helpful when trying to format text into columns or have other special considerations. An integer inserted between % 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.

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 provide a prompt for the user. For example, if you want the user to enter an integer and want to prompt for an answer, you **would 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`:

```
printf("What is your number? "); 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 ampersand 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
```

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*	printf: character string, prints until the first '\0' scanf: character string, reads until the first white space or new line

Table 4.2. Table of basic key codes for reading and writing variables.

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

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

```
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), care must be used with `scanf` because the amount of storage must 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.

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 is 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 %6.3f %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. We see that rhythm 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 we read the file in, we open it for reading on Line 41, we read the data on Line 46, and then close the file on Line 49. If you keep that rhythm in mind, it will help to keep all the file I/O contained and correct.

Note that the `fprintf` and `fscanf` functions appear to be 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 (variable of type `FILE*`). In fact, `printf` and `scanf` are just specializations where the file buffer is the pre-defined 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 why was π cut off at three decimal places? And what happened to the rest of the text phrase? To figure it out, let's look at the file that was saved. It looks like this:

```
17 3.142 Hello World!
```

It looks like when we saved the value of π , we used a format statement that allowed only three decimal places of accuracy, so we only got the first four digits of π saved to the file. When we read the data 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 it puts you at risk of losing accuracy.** We'll see how to remedy this shortly.

For the string, it appears that the whole string was saved, so we wrote it properly, but it didn't get read in as we expected. In this case, when `scanf` or `fscanf` reads into 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 we have seen above is really best suited for reading files of parameters written by a user. For example, you may have a parameter file the user provides that gives values for your program to run such as:

```
Parameters for run on April 30, 2018
x dimension: 50
y dimension: 100
```

Your program could read this data in, looking for the key parameters you define, extract the two values you're looking for, and then use that to control your program. The advantage of this approach is that (a) you can keep a record of the input arguments provided to your program in readable form to match with the output, (b) the input can be done offline. The latter advantage is rather important because when your program is run on a large cluster, it will have to be scheduled and won't be able to ask

for interactive user input. Therefore, your program is ultimately going to need to get its input values from a file. Finally, the more flexible your program is to read 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 that your program was unable to read or incorrectly read the input data file.

So how do we fix up our data storage problem? We don't want to lose 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. In this case, to write the data, we use `fwrite`. Take a look at Line 32 where we store the value of `num`. The first argument for the function is a pointer to the memory that should be written to the file, hence we give it `&num`. The next two arguments will be multiplied together to get the total amount of memory to be written. The `sizeof` call gives how big the data type being stored is, and the following argument says how many of those type will be stored. In this case, we'll use 4 bytes total because that's the amount of storage required for a single `int`. 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 chars we declared on Line 21. We didn't use all the storage in that string, but that's OK.

The data is read back in the same order as it was written using the complementary function `fread`, which has the same arguments as `write`. Those calls are on Lines 58, 61, and 64. This time, when the results are printed, we get what we would have wanted:

```
17 3.141593 Hello World!
```

Exercises

- 4.1. Write a program to read a 4 by 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 by 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 file name using binary format. The second program should take

as input a filename and then read the data from the file with that filename and print the results on the terminal.

- 4.3. Write a program that creates this struct:

```
1 struct inputarg {  
2     char name[20];  
3     char value[50];  
4 }
```

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:

```
1 numberOfRows      10  
2 numberOfCols      5  
3 outputFileName    foo.dat  
4 studentName       John Smith  
5 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 a 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.

Chapter 5

Flow Control

My thesis advisor once told me that “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 if you can do those two, then for sure you can do 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.

The structure of basic scientific computing applications is really a big for loop as we advance through time or through the arrays of data.

5.1 • For Loops

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

```
1 int array [100];
2 for (int n=0; n<100; ++n)
3     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 only be a valid variable *inside the loop*. If n were declared outside the loop, then it can 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 int main(int argc, char* argv[])
4 Program illustrates the concept of the scope of the incrementing
5     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 only defined 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, the last is acceptable. Again, it just depends on the purpose of the loop within your 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. We'll discuss boolean values in a bit more detail when we discuss **if** statements.

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.

As a final example, consider these loops:

```

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");

```

This illustrates how multiple variables can be initialized, separated by a comma in the initializer part, and multiple variables can be updated in the increment part. Compare the program and output in the above example to make sure you have a clear understanding of how the loop structure works.

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:

```

1 for ( ; test; ) {
2     ...
3 }
4
5 while (test) {
6     ...
7 }

```

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 New-

ton's method could be programmed for solving $f(x) = 0$, where df/dx 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 }
```

Note here that the `fabs()` function is the absolute value function.

The do-while loop is similar to the while loop except the contents of the loop is 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 when there is a choice between two alternate paths, and the latter handles when there are more than two alternate paths. We'll look at them both.

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:

```

1 if (test) {
2     // statements to execute if test is true
3 } else {
4     // statements to execute if test is false
5 }
```

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 you have a main program that expects one argument, then you would expect `argc` to be 2 (remember the name of the program itself counts as one of the arguments). You might put a check for that at the beginning of your program, and then print an error message or reminder of the proper way to call your function as is done in this main program:

```

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.;
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, one equal sign, with testing for equality, 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, you must use curly brackets to indicate your intent. 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 it 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. When we begin distributed computing, this structure will be very helpful because at some point the program will have to branch depending on which processor we are 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, and 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 in case none of the other cases match. Note the presence of the `break` statement 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, and 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

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

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 using a `while` loop to implement Newton's method to find the root of a quadratic polynomial. Let $f(x) = ax^2 + bx + c$ where the coefficients a , b , and c are read in from the command line as floating point values. Newton's method has the following update equation:

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

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.

- 5.6. The code in Exercise 5.5 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 .
- 5.7. Modify the code in Exercise 5.5 to check whether the coefficients given as input will admit a solution, i.e. check that the discriminant $b^2 - 4ac \geq 0$. It should terminate with a warning if the coefficients do not admit a solution.

Chapter 6

Functions

None of the programs you will write in practice 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. If you recode everything every time you want to solve a new problem, your productivity will plummet. Keeping your programs as modular as possible, and as generic as possible, will make your code reusable. One way in which to assure this is to compartmentalize your algorithm into functions. Functions are a way to put repeated tasks in one place, to simplify your algorithms into simpler building blocks, to make testing and validating your 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 the `clock()` function and how it can be used to measure the performance of your program, i.e. the time it takes for the program to complete its calculations.

6.1 • Declarations and Definitions

The concept of functions is not foreign, we've used them already in some of our discussion. For example, we've been using 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 only return a single value, although we will see 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`. If you try to use the `sin` function without including the `math.h` header file, the compiler will give you 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 you write 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 you make a call to it. So part of what must be considered is whether a function you plan to write is for use in many other code files, or whether it is a specialized function that is only relevant to the current code file. These two scenarios get to the point of the *scope of a function*. Again, we'll work through some common scenarios to get a feel of what the scope of a function is. Suppose you wish to create 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 the 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 int main(int argc, char* argv[])
29 {
30     int n = atoi(argv[1]); // Get the input number
31
32     printf("%d^2 = %u\n", n, squareOf(n));
33
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 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
6
7     printf("%d^2 = %u\n", n, squareOf(n));
8
9     return 0;
10}

```

In this example, the two files would be compiled separately, and 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.

You may have noticed that the `#include` line on Line 1 of `main.c` used 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 you want the compiler to look for your files, use the double quotes and it will first look in the directory where the file is being compiled. Additional local directories can be added to the search path using a `-I/path/to/header/files` option when compiling.

6.2 ■ Function Arguments

The way we communicate with functions is through their arguments and return value. Functions may receive any fixed number of arguments (variable length argument lists is possible, but too advanced for this course). 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. If we rewrite Example 6.2 to be passed by reference, it would look 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
14 // The definition here matches the declaration above
15 unsigned int squareOf(int* n)
16 {
17     return *n * *n;
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 ampersand 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, you're getting the original data. This means that the value of the argument can be changed within the function. 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 anything more than that in terms of storage requirements then passing by reference is preferred.

The return value of a function behaves in a very similar way, but in the opposite direction. But because function variables are only temporary, care must be used when returning variables by reference. 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 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:

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

```
1 int n = 5;
2 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 can have the return value of `void`, and when that is done the `return` statement is omitted. For example, the function above could be rewritten as below and the variable `n` will still be changed to 25.

```
1 void alsogoodfunction(int* n)
2 {
3     *n = *n * *n;
4 }
```

6.3 • Measuring Performance

It is assumed that the reason you are reading this text is to learn how to do scientific computing faster using modern computer architectures. That means there must be a means of evaluating that performance, and 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 so that when you are solving your own research problem, you will have a better idea of which type of hardware configuration is best suited for your task, and to have an idea

of the expected improvement to be gained by utilizing that hardware. This means we need a way to compute the elapsed time taken by a computer program.

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. If we want to evaluate your program alone, then those other extraneous tasks should not be included because they may very depending on the load on the computer at the time your 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 you want 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, and not break things down into smaller pieces. To do this, we need to dig inside the code and use the `clock()` function.

Example 6.6.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4
5 int main(int argc, char* argv[])
6 {
7     // start the timer
8     clock_t begin = clock();
9
10    // run some code where timing is desired
11    int N = atoi(argv[1]);
12    my_function(N);
13
14    // stop the timer and print the results
15    clock_t end = clock();
16    double elapsed_time = (double)(end - begin)/CLOCKS_PER_SEC;
17    printf("elapsed time = %e\n", elapsed_time);
18
19    return 0;
20 }
```

Example 6.6 shows how to use the `clock()` function to measure the elapsed user

time from within a program. Here, we wish to know how much time is being spent in the function `my_function`. Think of this as starting and stopping a stopwatch. On Line 8, the stopwatch is started running by measuring the precise time that this point in the program is reached. The stopwatch is stopped on Line 15. Actually, it's more like a lap timer, the clock continues to run so that you can sample the time many times and 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, the header file `time.h`, included on Line 3, provides the conversion factor `CLOCKS_PER_SEC`. An example of its use is on Line 16. This is the preferred method for measuring the performance of your code.

It is important to understand that 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 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. Nonetheless, it is important to use these measurements to demonstrate that the methods learned in this text results in actual time savings in the end.

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 following declaration:

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

that 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 deriva-

tive using periodic boundary conditions in the direction specified as either 'x' or 'y'. The results should be stored in the array `deriv`.

- 6.4. Experiment with some of your programs you've written so far to measure the elapsed time. In particular, look for a program that takes a variable length as a parameter. Make a plot comparing the elapsed time vs. the input parameter. Use several runs for each value of the input parameter to add error bars to your plot.

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 lots of options, so we will just give a few examples of calling representative functions that will be particularly useful for the projects in this book. You should learn to access the documentation for the libraries to take full advantage of their capabilities. In this chapter, we will learn how to incorporate library content into our 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 explore 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 we've learned already, when we compile code files, 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'. For example, the C math library can be found with the name `libm.a`. If we used a trigonometric function in our program, then we will need link this file into our program. When we link the program we use the link library command that begins with '`-l`'. So we may have something like

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

The extra '`-lm`' means resolve any loose function references by using the math library, `libm.a`.

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 your final binary program. If you call lots and lots of different functions, that 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 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 the upgrade changes the interface or builds the code with a different and incompatible version of the compiler, then you may be required to recompile your program again. Both strategies have their merits, but for our purposes we’ll primarily stick with static libraries.

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 that you are using. It includes simple operations like vector and matrix arithmetic, but does not include any matrix inversion or linear equation solvers. The LAPACK library mentioned above 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 dotproduct 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. As we've seen before, we have to have the function declaration, and 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, we've created two statically allocated arrays on Lines 18, 19 of length five. The second and fourth arguments are the two vectors, and the third and fifth arguments are the strides. You'll find that many of these functions require the stride. This is done because it allows easy traversal of matrices where the stride may be one for going along columns, while larger than one for traversing rows.

To build the program, we must link the BLAS library, which is in the directory `/usr/lib/libblas.a`. 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.orgblas/>

Another package that you should learn about is LAPACK. 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. That means that when you store matrices for use in LAPACK, you have to store them in Fortran order, not C order. Arrays in C are stored in *row-major order*, i.e. the rows, or the first index, increments slowest, the columns or second index increment fastest. Figure 3.1 illustrates row-major ordering. Fortran uses *column-major ordering*, where the row, or first index, increments fastest. Therefore, if a square matrix is stored in a given length of memory, then the Fortran interpretation is the transpose of the C interpretation. Another affect of a Fortran library is that when calling Fortran functions from C, the function name must have an underscore tacked onto the end. Finally, in Fortran, all variables to functions are passed by reference, so we will have to do the same.

To illustrate these points, consider Example 7.2 where the function `dgesv` solves a single linear system of equations.

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

First off, the C interface for LAPACK is included on Line 2. Note that not all systems have this header file installed by default. If that's the case for your system, you can download the file from Netlib and other web sites. Different versions of `lapacke.h` have different ways of calling the LAPACK functions. More recent versions make it so that the call to a standard LAPACK function, like `dgesv`, has C-type calls by value where appropriate, and there is a leading argument that allows for the choice of whether the data is row-major or column-major.

Other versions, or if you do not have this header file, require that the Fortran version of the arguments be used. Line 78 is how the Fortran version would be called. Note that when calling a Fortran subroutine, the function name must be followed by a trailing underscore character so that the compiler can bind it to the proper function in the library. Note also that Fortran passes all variables by reference, so all the single integer values must be passed to the function with the leading ampersand indicating that you are passing the address of the variable, not the value. The variables `A`, `ipiv`, and `b` are already pointers because they are arrays, and hence should not have the leading ampersand. 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 the 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, or tr for tridiagonal, etc. The rest of the name represents the actual operation. In our example, sv means solve. See the LAPACK manual for more information about the naming convention (<http://www.netlib.org/lapack>).

Finally, recall that the storage of an array in Fortran is column-major, while C uses row-major. For square matrices, this means that the matrix is stored as its transpose. This is why the indices appear reversed on Line 88.

To build this program the link line will look like

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

The BLAS library is also listed even though we didn't actually call any BLAS functions ourselves because the LAPACK library is built on top of the BLAS library, meaning it uses functions found in the BLAS library.

Note that the link to `-llapack` is before the link to `-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 `dgesv_`. It then goes to the first library and checks to see if it any of these functions can be resolved by this library. It finds `dgesv_` in the LAPACK library, but this function in turn calls some functions in the BLAS library. So in fact, while we've resolved one function, a collection of new unresolved functions are left behind. Fortunately, those functions are found in the BLAS library, and everything is resolved. If the two libraries 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 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 there are among your 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, for example. Because tridiagonal solvers are particularly useful in many numerical methods, we present another example of using LAPACK where we solve the same problem as in Example 7.2. It is also worth noting that it is often more efficient to factor the matrix A once, and then re-use 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 dgttrs.
70     dgttrf_(&n, ald, ad, aud, auud, ipiv, &info);
71
72     // To solve Ax=b, we set the transpose character to 'N'
73     char trb = 'N';
74
75

```

```

74 // To solve  $A'x=c$ , we set the transpose character to 'T'
75 char trc = 'T';
76
77 // This is where the backsubstitution is done.
78 // This function does not alter the factorization so it can be reused.
79 dgtrrs_(&trb, &n, &nrhs, ald, ad, aud, auud, ipiv, b, &n, &info);
80 dgtrrs_(&trc, &n, &nrhs, ald, ad, aud, auud, ipiv, c, &n, &info);
81
82 // Print out the original system  $Ax=b$  replacing  $x$  with the solution
83 printf("Check Ax=b:\n");
84 for (int i=0; i<5; ++i) {
85     printf("[ ");
86     for (int j=0; j<5; ++j) {
87         switch (j-i) {
88             case -1: printf("%9.3f", aorig[0][i-1]); break;
89             case 0: printf("%9.3f", aorig[1][i]); break;
90             case 1: printf("%9.3f", aorig[2][i]); break;
91             default: printf("%9.3f", 0.); break;
92         }
93     }
94     printf("] [%9.3f]", b[i]);
95     if (i==2)
96         printf(" = ");
97     else
98         printf("    ");
99     printf("[%5f]\n", borig[i]);
100 }
101
102 // Print out the original system  $A'x=c$  replacing  $x$  with the solution
103 printf("\nCheck A'x=c:\n");
104 for (int i=0; i<5; ++i) {
105     printf("[ ");
106     for (int j=0; j<5; ++j) {
107         switch (j-i) {
108             case -1: printf("%9.3f", aorig[2][i-1]); break;
109             case 0: printf("%9.3f", aorig[1][i]); break;
110             case 1: printf("%9.3f", aorig[0][i]); break;
111             default: printf("%9.3f", 0.); break;
112         }
113     }
114     printf("] [%9.3f]", c[i]);
115     if (i==2)
116         printf(" = ");
117     else
118         printf("    ");
119     printf("[%5f]\n", corig[i]);
120 }
121
122 return 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 69. Like for the function dgesv_ in the previous example, the function dgtrf_ also modifies the diagonals, so if the original matrix is needed, be sure to make a copy. However, the back substitution function, dgtrs_ does not modify the output of the dgtrf_ function. That means you need only factor a matrix once, and then you can re-use the factorization multiple times as illustrated on Lines 79–80.

The function dgtrs_ has another argument that didn't appear in the earlier example. The first argument allows you to modify the problem. If the first argument is the character 'N', then the function will solve the equation $Ax = b$. However, if the first argument is 'T', then the function solves $A^T x = b$ (the diagonals are still entered in the same order). These two cases are illustrated on Lines 79–80 respectively. In general, if many equations of the form $Ax = b$ have to be solved for a given matrix A , but for many different 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 b vectors. You may also want to take advantage of using multiple right hand sides to solve multiple systems simultaneously. In that case, the variable nrhs would be set to the number of right hand side vectors, and the vector b would be a column-major matrix where each column is a different right hand side vector. This is by far the most efficient way to solve multiple problems simultaneously.

As a final note, for general matrices, the functions corresponding to the pair dgtrf_, dgtrs_ are dgetrf_, dgetrs_ and the arguments follow a similar format:

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

where 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 $Ax = b$, and 'T' for solving $A^T x = b$. If your matrix A is stored in the column-major format instead of the expected Fortran-style row-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 row-major order either way.

7.2 • FFTW

The Fast Fourier Transform (FFT) have many uses for studying signals or for developing pseudo-spectral methods for solving partial differential equations. 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, we will focus 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

7.2.1 • Complex numbers in C

Before discussing the FFT, we should first discuss how to handle complex numbers in C. In standard C, there is no complex number type, so to use complex numbers, it was common to define a struct like this:

```
typedef struct {double real, imag;} complex;
```

and define functions like

```
complex cexp(complex z)
{
    complex w;
    w.real = exp(z.real)*cos(z.imag);
    w.imag = exp(z.real)*sin(z.imag);
    return w;
}
```

Fortunately, complex numbers have also recently been added to the C standard, and to get to those functions, you would include the system header file `complex.h`. 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` that is defined in the header file like this:

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

It also defines many complex functions such as

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

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. Alternatively, the FFTW package can provide 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, which is what we will use here. This means that arrays of complex numbers are organized as:

<code>real(z₀)</code>	<code>imag(z₀)</code>	<code>real(z₁)</code>	<code>imag(z₁)</code>	...	<code>real(z_{n-1})</code>	<code>imag(z_{n-1})</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 ■ 1D serial FFT

The basic strategy for all the FFTW routines for calculating an FFT is to: (1) allocate memory for the array of data, (2) create an execute plan, and (3) apply the FFT to the data by executing the plan. 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 int main(int argc, char* argv[])
6
7 Compute the Fast Fourier Transform for data from sin(x)
8
9 Inputs: none
10
11 Outputs: Prints the Fourier coefficients and the data after
12 both forward and backward transformation.
13 */
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 the 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 the 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 are set up. The FFT is actually executed on Line 34 in the forward direction, and 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 you would expect. 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 $i \cdot \text{dx}$ on Line 31 with $k \cdot i \cdot \text{dx}$. You should practice with this simple code until you understand how different values of k change the resulting output. Once you understand that, try linear combinations of the form $a \cdot \cos(k \cdot i \cdot \text{dx}) + b \cdot \sin(k \cdot i \cdot \text{dx})$ to see what happens.

Another consideration for this package is that the backward transform is not 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 you need the Fourier coefficients, 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 you divide by N , not N^2 on Line 41 to recover the original function.

If you are using this package to transform real-valued data, then some savings can be had by using the plan functions that transform between real and complex-valued data. For 1D, the functions are

```
fftw_plan fftw_plan_dft_r2c_1d(N, r_in, c_out, FFTW_ESTIMATE);
fftw_plan fftw_plan_dft_c2r_1d(N, c_in, r_out, FFTW_ESTIMATE);
```

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 1D transform, then 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	$a_{0,N/2}$
$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	\cdots	$a_{1,N/2}$
\vdots	\vdots	\vdots		\vdots
$a_{N,0}$	$a_{N,1}$	$a_{N,2}$	\cdots	$a_{N,N/2}$
$a_{1-N,0}$	$a_{1-N,1}$	$a_{1-N,2}$	\cdots	$a_{1-N,N/2}$
$a_{2-N,0}$	$a_{2-N,1}$	$a_{2-N,2}$	\cdots	$a_{2-N,N/2}$
\vdots	\vdots	\vdots		\vdots
$a_{-1,0}$	$a_{-1,1}$	$a_{-1,2}$	\cdots	$a_{-1,N/2}$

7.2.3 • Building code with FFTW

Linking the FFTW library to your code is straightforward. It requires linking both the FFTW library and also the math library because of the trigonometric function calls involved. Hence, to compile and link the above example, use a line like this:

```
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 purposes here, we will stick with the stock pseudo random number generator, `drand48()`.

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 your 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(long int seed)`. The more difficult part is how to choose the value of `seed`. One can rely on the user to provide a number, but better is 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, which is 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.

The second step is to call the generator to get random values. That part is straightforward, but the user should always be aware of the limitations of the generator to make sure that it is sufficient for the needs of the application. Every subsequent call to `drand48()` will produce a new random value in the range $[0, 1)$ using a linear congruential generator. An example generating a set of data is given in Example 7.5.

Example 7.5.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
```

```

4  /*
5   int main(int argc, char* argv[])
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 }
```

In Example 7.5, the random number generator is seeded either by user input as an additional argument, or by an initial source of randomness provided by `/dev/urandom`. It is assumed that if the program is run with only one argument, it will be the length of the array of random values. If an additional long integer is given as an argument, then it will be a given seed. The purpose for doing this is so that if for debugging purposes the same sequence of random values is desired, it can be recreated on the spot. To il-

lustrate this, we first run the program without specifying the seed value, and then on the subsequent run, the seed value is added resulting in the same sequence as illustrated below:

```
$ 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
```

When the initial seed is not specified, then it is taken from `/dev/urandom` beginning on Line 31. It 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. For this reason, it's not efficient to use it as a random generator all by itself, but is a good way to set an initial seed value. Treating `/dev/urandom` as a file, the file is then opened on Line 31 and enough bits to fill a long integer are read in on Line 34. Note that on Line 41, the seed generated is printed to the terminal so that it can be reused as described above.

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.

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 `dgesv_` with the pair of functions given below:

```
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 `dgetrf_`, the value 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 `dgesv_`. When `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 `dgetrs_`.

For `dgetrs_`, the first argument is a pointer to a character that has one of three values: '`N`', '`T`', and '`C`' indicating whether the program to be solved is $\mathbf{Ax} = \mathbf{b}$, $\mathbf{A}^T\mathbf{x} = \mathbf{b}$, or $\mathbf{A}^*\mathbf{x} = \mathbf{b}$. For this exercise, you will want to use '`N`'. The arrays `A` and `ipiv` will be the output of calling `dgetrf_`, while the remaining arguments are the same as for `dgesv_`.

Once you have this working, modify the program again to make `b` be a matrix the same size as `A` (this will require setting `nrhs` to be the same as `n`). Verify that all the solutions returned in the matrix `b` are correct.

- 7.2. Write a function that uses the functions `dgtrf_`, `dgttrs_` to solve the $N \times N$ tri-diagonal 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 δ and the $N \times N$ matrix `B` are inputs to the function returning the answer in `B`.

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

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

The value δ and the $N \times N$ matrix `B` are inputs to the function returning the answer in `B`.

- 7.4. Modify Example 7.4 so that you can specify one wave mode given as 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.
- 7.5. 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. Verify that after doing a forward and then backward transform that the results are equal.

Chapter 8

Projects for Serial 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. The other parts of this book will also draw upon the same projects so you can compare your parallel implementation 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 comment your 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. Measure the time required to complete the calculation by using the `clock()` function at the beginning and end of your program and print the elapsed time in seconds. Note that the function `log(double)` takes a double precision value and returns the natural logarithm.

8.1.2 ■ Duffing-Van der Pol Oscillator

Program the assignment in Section 34.3.2. Measure the time required to complete the calculation by using the `clock()` function at the beginning and end of your program and print the elapsed time in seconds.

8.2 ■ Finite Difference Methods

8.2.1 ■ Brusselator Reaction

Program the assignment in Section 35.3.1. Measure the time required to complete the calculation by using the `clock()` 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 35.3.2 using an $N \times N$ grid. Measure the time required to complete the calculation as a function of N .

8.3 • Elliptic Equations and SOR

8.3.1 • Diffusion with Sinks and Sources

Program the assignment in Section 36.1.1 using an $(2N - 1) \times N$ grid. For a serial implementation, it is not necessary to use red-black ordering, though you may wish to do it to prepare for implementations later in this book. Experiment to find the optimal ω that requires the fewest iterations to reach a tolerance of 10^{-9} . Compute the time required to complete a single pass through the grid. Repeat these steps for a grid of dimensions $(4N - 1) \times 2N$. Does the optimal value of ω remain the same? 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 MAC grid that is approximately $N \times N$. For a serial implementation, it is not necessary to use red-black ordering, though you may wish to do it to prepare for implementations later in this book. Experiment to find the optimal ω 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 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 $N \times N \times N$. For a serial implementation, it is not necessary to use red-black ordering, though you may wish to do it to prepare for implementations later in this book. Experiment to find the optimal ω 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.

8.4 • Pseudo-Spectral 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. Compute the time required to execute your code as a function of N .

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. Compute the time required to execute your code as a function of N .

Part II

Parallel Computing Using OpenMP

The type 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. When we do this, we are the ones that must decide how best 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 of the code.

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 2 or more cores within the single processor chip. Each core can do independent tasks, but they still must 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 which directives to which they can be applied, the reader is referred to more comprehensive texts, such as the presentation in [2].

In Chapter 9, we introduce the concept of multi-thread computing using OpenMP. 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. 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 multi-threading 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 multi-threaded region is discussed in Chapter 11.

When your algorithm has multiple serial tasks that are not interdependent, then you may want to designate different threads 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`, which 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 now 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 you should look for *multi-thread safe* versions. 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, in Chapter 15 we revisit the projects in Part VI to see how multi-threading can be applied to improve computational performance.

Chapter 9

Intro to OpenMP

For OpenMP, the various parallel “programs” that are created are called *threads*. Your 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 do they each get their 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 both at the point of entry and 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  * int main(int argc, char* argv[])
5  * The main program is the starting point for an OpenMP program.
6  * This one simply reports its own thread number.
7  *
8  * Inputs: none
9  *
10 * Outputs: Prints "Hello World #" where # is the thread number.
11 */
12
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 }
```

First note that we are going to be using a collection of new functions designed for creating and managing threads, so we need the function declarations in a header file. That header file 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

```
#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 that you can't assume the threads will execute in sequential order.

Each of the threads has a thread ID that allows you to alter the behavior of specific threads should you need to. The thread ID can be obtained by calling the function `omp_get_thread_num()`, and the total number of threads currently in use can be obtained by the function `omp_get_num_threads()`. These functions are on Line 28 of the example and is what allows 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. We emphasize the word *statement* 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     printf("Hello World! I am thread %d out of %d\n",
4            omp_get_thread_num(), omp_get_num_threads());
5     printf("All done with the parallel code.\n");
6 }
7 }
```

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

Finally, before we get into a more refined discussion about the OpenMP system, note that building the executable is also very easy. First, when compiling your code, you must add the language option `-fopenmp` like this:

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

This enables the extra compiler directives such as on Line 24 of Example 9.1. Second, the OpenMP library `-lgomp` must be added when linking to make the executable like this:

```
$ 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 this section, we look at the clauses used for the `#pragma omp parallel` directive.

9.2 • The num_threads() clause

Because algorithms often depend on knowing exactly how many threads are working simultaneously, the number of threads used in a particular parallel region can be controlled with the `num_threads()` clause. On our example computer, the default number of threads is eight, as evidenced by the output of Example 9.1. If we wanted only four threads, then we add the `num_threads` clause to Line 24 like this:

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

which results in the output

```
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(void)`. Alternatively, in the Linux environment, you can use the command `nproc --all`. The maximum number of threads is implementation dependent, but typically 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 <omp.h>
3
4 /*
5  int main(int argc, char* argv[])
6
7 computes the number of instances of a given index
8
9 Inputs: input array of integers
10
11 Outputs: Prints the count of integers that match the thread ID
12
13 */
14
15 int main(int argc, char* argv[])
16 {
17 // number of arguments is the length of the list except for argv[0]
18 int N = argc-1;
19 int a[N];
20 int i;
21 // convert the arguments into an array of ints
22 for (i=0; i<N; ++i)
23     a[i] = atoi(argv[i+1]);
24 int count = 0;
25
26 #pragma omp parallel num_threads(4) shared(N)
27 {
28 // each thread counts how many inputs match their thread id
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 times\n",
32         omp_get_thread_num(), count);
33 }
34
35 return 0;
36 }
```

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 times  
The number 3 appears in the list 2 times  
The number 0 appears in the list 1 times  
The number 2 appears in the list 2 times
```

Note that we made the variable `N` a shared variable by using the `shared(N)` clause. 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` that we didn't specify whether is shared or not. 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 we designate the variable `count` to be shared by changing Line 26 to be

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

This results in the following output:

```
The number 1 appears in the list 5 times  
The number 3 appears in the list 2 times  
The number 0 appears in the list 6 times  
The number 2 appears in the list 8 times
```

As you can see, we got completely wrong results. This is because by declaring the variable `count` to be shared, when `count` is incremented on Line 30, it is incremented in *all* the threads.

There are lessons to be drawn from this example. First, while it worked out well that we let OpenMP assume that `count` is not shared, it is strongly recommended that every variable used in the parallel region be specified either as shared or private to avoid unexpected or incorrect results. Second, variables that are only read and 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 we modify Line 26 to be

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

we might expect better results, but instead we get this:

```
The number 1 appears in the list 1 times  
The number 3 appears in the list 1 times  
The number 0 appears in the list 1 times  
The number 2 appears in the list 1 times
```

So what is going on? There is a simple fix, but this is also an opportunity to discuss 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. So in our case, in the parallel region, there are four separate variables called count, one for each thread. Now, 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 like this:

```

25 #pragma omp parallel num_threads(4) shared(N) private(count)
26 {
27     count = 0;
28     for (i=0; i<N; ++i)
29         count += a[i] == omp_get_thread_num() ? 1 : 0;
30     printf("The number %d appears in the list %d times\n",
31           omp_get_thread_num(), count);
32 }
```

For this simple example, this is probably the best solution, but in more complex situations, you may not want to go through a complicated calculation again just to set the initial value of a private variable. To account for this, instead of using the `private(count)` clause, we instead use the `firstprivate(count)` clause like this:

```

25 #pragma omp parallel num_threads(4) shared(N) firstprivate(count)
26 {
27     for (i=0; i<N; ++i)
28         count += a[i] == omp_get_thread_num() ? 1 : 0;
29     printf("The number %d appears in the list %d times\n",
30           omp_get_thread_num(), count);
31 }
```

By using the `firstprivate(count)` 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 24 of Example 9.2 and using the `firstprivate(count)` 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 we did not do so for two other variables in Example 9.2: `i`, `a`. The program seemed to run fine using the default state, so what is that default state and what *should* `i` and `a` be? 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.

Now that we understand how private variables work, a corresponding question about the exit of the parallel region should be asked: what happens to the various values of count at the end of the parallel region? To answer that, if we print the value of count after the parallel region, 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 24 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. There is no remedy for this here, but in the context of other OpenMP directives there is.

9.5 ▪ The reduction() clause

As noted above, 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 we can combine all the private instances of count by adding them up and storing them in the count variable in the main scope outside the parallel region. The modified code looks like this:

```

25 #pragma omp parallel num_threads(4) shared(N) reduction(+:count)
26 {
27     for (i=0; i<N; ++i)
28         count += a[i] == omp_get_thread_num() ? 1 : 0;
29         printf("The number %d appears in the list %d times\n",
30             omp_get_thread_num(), count);
31     }
32     printf("Total of all found numbers is %d\n", count);

```

Running the program with this input:

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

produces the output

```
The number 1 appears in the list 3 times
The number 3 appears in the list 2 times
The number 0 appears in the list 1 times
The number 2 appears in the list 2 times
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. 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 the table below:

Operator	Description
+	Sum
*	Product
-	Negative sum
&&	Logical and
	Logical or

Note that when using the - operator in the reduction clause, it's equivalent to using the + operator and multiplying by -1 at the end. The logical and operator returns true if the variable is true in *all* the threads, while the logical or operator returns true if the variable is true in at least one of the threads.

Exercises

- 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 with the each thread adding its thread number to the seed. Use the `reduction()` clause to accumulate the results. Verify that each thread generates a unique sequence of numbers, and that your 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. Where there are additional nuances, we will revisit the concept of private variables and how they interact with the code region outside the loop. 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 or not. 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  * int main(int argc, char* argv[])
6  *
7  * The main program illustrates how to divide a loop into smaller
8  * pieces each handled in parallel by different threads.
9  *
10 * Inputs: argc should be 2
11 *         argv[1] is the length of the array
12 *
13 * Outputs: Initializes an array in parallel
14 */
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 were created, and each thread initialized two of the sixteen 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 were 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. For the variable `N`, it is safe to be shared 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, you see that 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.

It should be noted that the `#pragma omp for` directive does not work for all loops. For example, one cannot parallelize a loop for which the number of iterations cannot be determined *a priori* 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

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

where the termination condition must be one of $<$, $>$, \leq , or \geq . 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 we know the number of iterations ahead of time and can use loops of this sort. The compile will fail if the conditions for parallelizing the loop are not met.

10.2 ■ The private() clause

Similar to the `#pragma omp parallel` directive, clauses can also be applied to other directives such as for `#pragma omp for`. One example is the use of the `private(var)` clause on Line 28 in Example 10.1. That part of the code 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 only be private 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 that a variable would have at the conclusion of a 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, omp_get_thread_num())
33     ;
34     k = i;
35   }
36   printf("Final k = %d\n", k);
37 
```

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

```

You should recall from the discussion about private variables in Section 9.4, that the value of `k` reverted to the value that was set before the loop was entered because the private `k` variables in each thread are separate from the one created outside the parallel region. However, if we change Line 30 to be

```
#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 we chose $N = 2$). Note that the final value is according to the last value of `k` as if the loop had 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 we will review it again here to avoid some confusion in the usage. Consider the following example usage:

Example 10.2.

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

```

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

To understand what is happening, 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, which is a partial sum of the array, 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

You may have noticed 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, we want the `printf` statements 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 we place on the computation, the potentially less efficient the resulting code may be. That said, making the above changes results in the output we would normally expect:

```
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 we have used as examples in this section, 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 and 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.

Example 10.3.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <omp.h>
4
5 /*
6  int main(int argc, char* argv[])
7 {
```

```

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

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.

One new function introduced here is the function `omp_get_wtime()`, which returns the current wall time from the system. The methods used to measure performance in a typical single-threaded program will not work as expected when creating a multi-threaded 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. 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

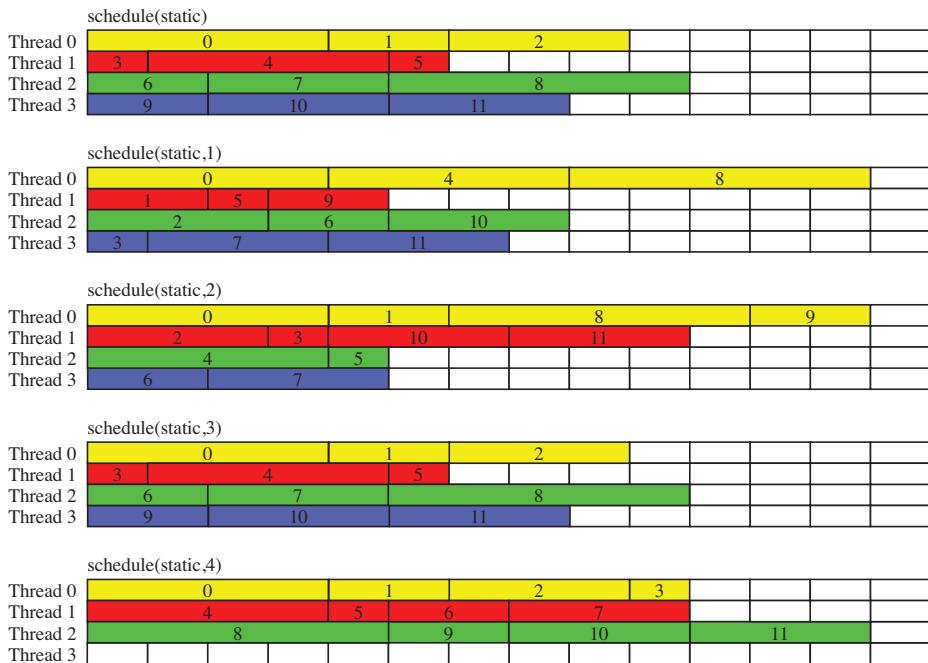


Figure 10.1. Chart of threads executing iterations in the loop in Example 10.3 for $N = 12$ iterations and four threads. Time required for each iteration was chosen at random, the same for all cases. Static scheduling with various block sizes are illustrated. Optimal case is for block size 3 in this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

clock time, to complete the code between the two calls.

Another new function introduced in this example is the `sleep(int sec)` function which pauses the program for the number of seconds given in the input. Thus, beginning with Line 36, the current time is sampled. Next, the loop is paused for a randomly chosen predetermined length of time. After the pause, the time is sampled again on Line 38. This example is to illustrate 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 33, 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 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 31). Similarly, thread 1 will start with iteration 1 and then continue with every fourth iteration, and so forth for the other two threads. Since we don't know anything *a priori* how long each iteration will take to run, using a static

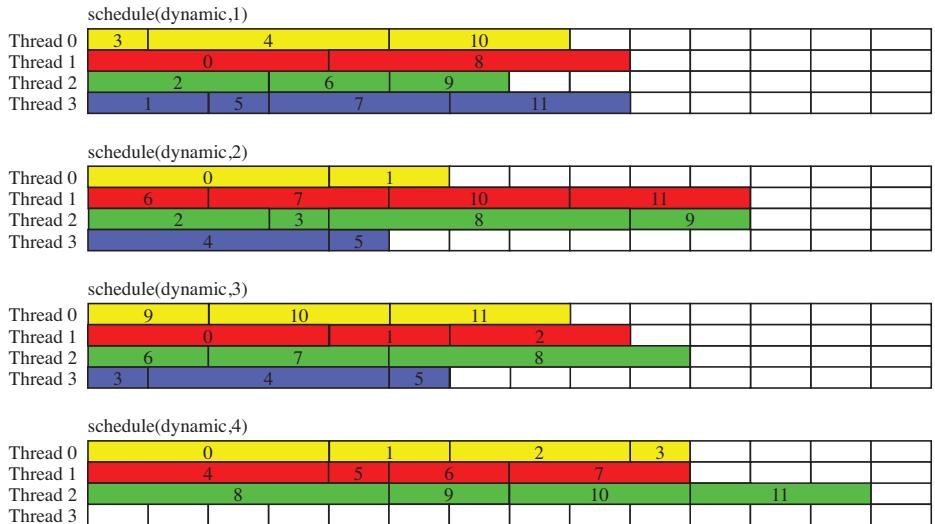


Figure 10.2. Chart of threads executing iterations in the loop in Example 10.3 for $N = 12$ iterations and four threads. Time required for each iteration was chosen at random, the same for all cases. Dynamic scheduling with various block sizes are illustrated. Optimal case is for block sizes 1 in this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

scheduling scheme is problematic if one particular thread happens to disproportionately get longer threads. 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, then 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 33 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 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

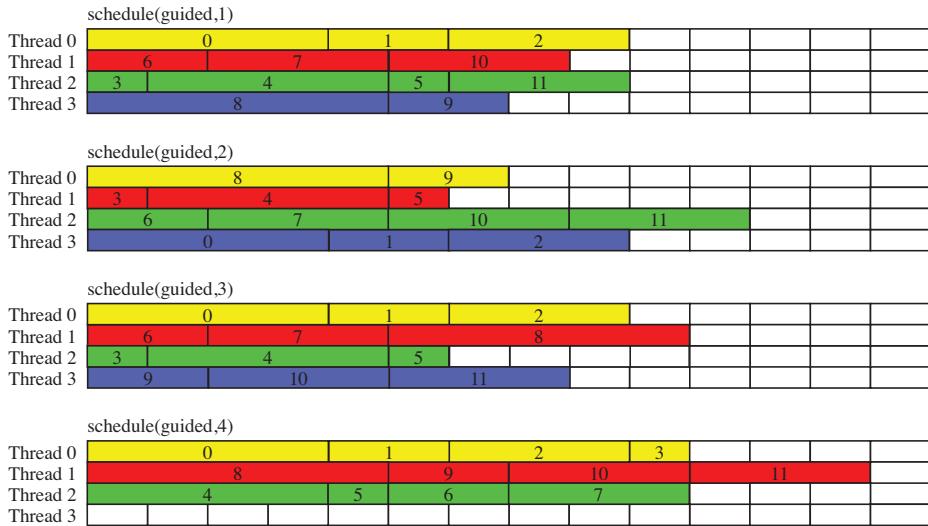


Figure 10.3. Chart of threads executing iterations in the loop in Example 10.3 for $N = 12$ iterations and four threads. Time required for each iteration was chosen at random, the same for all cases. Guided scheduling with various block sizes are illustrated. Optimal case is for block size 1 in this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

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 produced 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, and 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, 10.3. For dynamic scheduling with block size 2, iterations are doled in groups of size two regardless of how many iterations remain. Thus, we see that for the case `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 dwindle, 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 are 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, we need to add a second parallel for loop to Example 10.3:

Example 10.4.

```

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

```

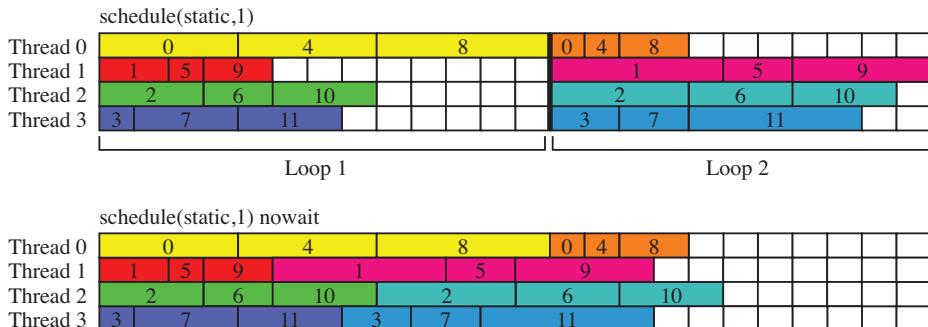


Figure 10.4. Chart of threads executing iterations in the loop in Example 10.4 for $N = 12$ iterations and four threads. Time required for each iteration was chosen at random, the same for all cases. Without the `nowait` clause, all threads wait at the barrier, indicated by the thick vertical line, until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

```

48     t2 = omp_get_wtime();
49     printf("Loop 2: Iteration %d: Thread %d, started %e, duration %e\n"
50             " ",
51             i, omp_get_thread_num(), (double)(t1-t0), (double)(t2-t1));
52 }
53
54 // stop the timer
55 t2 = omp_get_wtime();
56 printf("CPU time used: %.2f sec\n", t2-t0);
57 return 0;
58 }
```

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 41, 42 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 41 is removed by adding the `nowait` clause on Line 33. 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 first loop calculation, then 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.

Exercises

- 10.1. 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 with the each thread adding its thread number to the seed. Use the `reduction()` clause to accumulate the results. Verify that each thread generates a unique sequence of numbers, and that your final result is approximately 2^{N-1} . Explicitly declare each variable as either shared or private.
- 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, you may wish to handle it with a single thread within the parallel region. This is accomplished by using the `#pragma omp single` directive. In this chapter, we explore the use of single thread regions inside of a multithread region. 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 we will also explore those nuances where 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 #pragma omp parallel num_threads(4) shared(N,M,a,count) reduction(+:sum)
26 {
27     // initialize the shared variables, only one thread needed
28 #pragma omp single
29 {
30     printf("Thread %d is initializing.\n", omp_get_thread_num());
31     M = 4;
32     N = 16;
33 }
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
53 printf("The sum is %d\n", sum);
54 return 0;
55 }
```

On Lines 29 and 45, the code inside the braces will be run by a single thread, while the code in Lines 37–42 are 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?

Let's 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 0 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. Because 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, we will place the barrier 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, 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 // 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 printf("The sum is %d\n", sum);
56 return 0;
57 }
```

This example also shows how there can be subtle errors in your code that may not appear every time, but may randomly malfunction if not proper attention and care are 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 your 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 we haven't encountered yet, which is the `copyprivate()` clause. This clause will take the private variable value that is presumably computed inside the single region and copy the result to all the threads in the parallel region. For example, suppose in Example 11.2, we had wanted to make the variables `M`, `N` private and also wanted to initialize the private variable `sum` 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 is 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[])
6  * 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, 0,
20                  6, 9, 10, 7, 0, 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)
27         reduction(+:sum)
28     {
29         // initialize the shared variables, only one thread needed
30 #pragma omp single copyprivate(M,N,sum)
31     {
32         printf("Thread %d is initializing.\n", omp_get_thread_num());
33         M = 4;
34         N = 16;
35         sum = 0.;
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. On Line 29, the `copyprivate()` clause is used 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.

Exercises

- 11.1. Write a program that creates an array x_j of N values generated using `drand48()`. Within a single parallel region use a subdivded loop that computes

$$x_j \leftarrow \frac{1}{4}(x_{j+1} - 2 * x_j + x_{j-1}),$$

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, we have either had all threads perform the exact same task, or we have had one thread perform 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, we explore the directive `#pragma omp sections` that allows for 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, we wish to display the data, and compute the max, min, and mean of the data. 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   // This section will print the list of numbers
42 #pragma omp section
43 {
44   printf("Thread %d will print the contents of the array\n",
45         omp_get_thread_num());
46   for (i=0; i<N-1; ++i)
47     printf("%d, ", u[ i ]);
48   printf("%d\n", u[ N-1 ]);
49 }
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 Section 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, then the private variables behave in the same manner as we have seen 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, this means that we may safely use `lastprivate(minu)` because the value of `minu` is computed in the last section in the code beginning on Line 74. Had we tried using, for example, `lastprivate(maxu)`, the results will 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, if we use the clause `reduction(+:maxu)` and set `maxu` to be zero in each section except for the one where the max 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. When using the `nowait` clause, when a thread completes its assigned section, or was an extra thread that was not assigned a section, then 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.

Exercises

- 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

$$y_j^n = \sum_{k=0}^n x_j^k,$$

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 is desired, but need to make sure that the threads are not updating the shared variables simultaneously. This kind of data collision can cause unexpected results. In this chapter, we explore two different options 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. However, 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 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 printf ("%d/%d numbers are below %d\n", count, N, mid);
51 return 0;
52 }
```

Note that we have designated the variable `count` to be shared. Typically, we try to avoid having multiple threads storing to the same variable to avoid conflicts called data race conditions. We have already seen a different strategy for this type of situation where `count` is private and then combined at the end via the `reduction(+:count)` clause. This is an alternative strategy to illustrate the `#pragma omp atomic` directive.

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 here 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, in our simple example, thread 0 would complete steps 1–3 above before thread 1 is permitted to do begin. This way, thread 0 updates count to 1, and thread 1 updates it to 2 as we would want. If we run the example with and without the `#pragma omp atomic` directive, we do indeed see the effect. For the exact same data, we get the following results:

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 we would not have caught had we not checked.

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, suppose we also wish to sum up all the numbers that are below that threshold, then we would modify the example to get 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 }
```

```

30 // Generate the random values
31 #pragma omp parallel for private(i) shared(a,N)
32   for(i=0; i<N; ++i)
33     a[i] = rand();
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, we have replaced the `#pragma omp atomic` directive with the `#pragma omp critical` directive. The critical directive allows for multiple lines surrounded by braces, and allows for more complex statements.

One should use the `#pragma omp atomic` directive for very simple incrementing operations because of its improved efficiency over the more general `#pragma omp critical` directive, but the critical directive is there for the cases where the atomic directive is not sufficient.

Exercises

- 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 than 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 than how they would be used in serial code as was discussed in Chapter 7.1. Consequently, we won't repeat that discussion here. For other libraries there can be some differences, and so we address those differences 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, you will not need to actually use the OpenMP directives in order to take advantage of the parallel implementation of FFTW. For example, if one wants to build multiple FFT plans, one for each thread, then one would use OpenMP directives to complete that task, each plan being built by one thread. Alternatively, each plan can be built using multiple threads sequentially, which is the preferred strategy. Example 14.1 shows how a multi-threaded OpenMP implementation of Example 7.4 is done.

Example 14.1.

```
1 #include <stdio.h>
2 #include <math.h>
3 #include "fftw3.h"
4
5 /*
6  int main(int argc, char* argv[])
7
8  The main program illustrates how to use the multi-threaded
9  version of the FFTW library
10
11 Inputs: none
12
```

```

13  Outputs: Prints the original data and the result of a forward
14  and backward transform for comparison
15
16 */
17
18 int main(int argc, char* argv[])
19 {
20 #ifndef M_PI
21     const double M_PI = 4.0*atan(1.0);
22 #endif
23     int N = 16;
24     fftw_complex *in, *out, *out2;
25     fftw_plan p, pinv;
26
27 // initialize the threads to be used in the FFT execution
28 fftw_init_threads();
29
30 // allocate memory for the data
31 in = (fftw_complex *) fftw_malloc(sizeof(fftw_complex)*N);
32 out = (fftw_complex *) fftw_malloc(sizeof(fftw_complex)*N);
33 out2 = (fftw_complex *) fftw_malloc(sizeof(fftw_complex)*N);
34
35 // create an FFT plan that uses all the available threads
36 fftw_plan_with_nthreads(omp_get_max_threads());
37
38 // construct the plans for the forward and backward transforms
39 p = fftw_plan_dft_1d(N, in, out, FFTW_FORWARD, FFTW_ESTIMATE);
40 pinv = fftw_plan_dft_1d(N, out, out2, FFTW_BACKWARD, FFTW_ESTIMATE);
41
42 // construct the initial data
43 for (int i=0; i<N; ++i) {
44     double dx = 2.*M_PI/N;
45     in[i][0] = sin(i*dx);
46     in[i][1] = 0.;
47 }
48
49 // perform the forward transform, results are in variable out
50 fftw_execute(p);
51
52 // perform the backward transform, results are in out2
53 fftw_execute(pinv);
54
55 // print the original data and the results for comparison
56 for (int i=0; i<N; ++i)
57     printf("a[%d]: %f + %fi\n", (i<=N/2 ? i : i-N), out[i][0]/N,
58                                     out[i][1]/N);
59     printf("----\n");
60     for (int i=0; i<N; ++i)
61         printf("f[%d]: %f + %fi == %f + %fi\n", i, out2[i][0]/N,
62                                         out2[i][1]/N, in[i][0], in[i][1]);
63
64 // clean up the threads from OpenMP
65 fftw_cleanup_threads();
66
67 // free the remaining memory
68 fftw_destroy_plan(p);
69 fftw_destroy_plan(pinv);
70 fftw_free(in);
71 fftw_free(out);
72 fftw_free(out2);
73 }
```

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. This is done on Line 28. For multi-threading to work it's essential to put that at the beginning of the program. Next, on Line 36, we tell the multi-threaded plan construction how many threads it can expect to use for building the FFT plan. This function can be called multiple times if the number of threads to be used is different for different plans. In this instance, we will assume that we'll throw all available threads into 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 50, 53 are done in parallel using OpenMP. Finally, the extra memory used for the multi-threaded version must be cleaned up, hence there is an additional thread cleanup on Line 65.

Finally, when linking this program, the additional library `libfftw3_omp` must be added in addition to the ones used to build Example 7.4. Thus, to compile and link Example 14.1, would look like:

```
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()` that we used in Part I are not necessarily thread-safe. For example, consider Example 14.2 where we use `random()` 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 k 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 we should expect that this will result in the same list of numbers 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 we 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 non-negative integer values, but the last entry in the first column is negative. What is happening is that the functions `random()` and `srand()` are not thread-safe. Thus, the results we get 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 we are not able to make that 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 below.

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     srandom_r(seed, state);
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 our random number generator is thread-safe. Note that we are now using functions on Lines 36, 39, 43 which have a suffix of `_r`. These functions are designed to be thread-safe versions of the functions we started with. However, the changes are more substantial than just adding the suffix because we must keep the random number generator separate for each thread, which will require additional private storage to keep each thread independent.

The first change is that 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. We will allocate memory for a state variable for each thread within the parallel region on Line 32. As it happens, it is not sufficient to just have a local state variable, the state variable also relies on a separate variable length buffer,

which we have declared on Line 35. This also must be private to each thread. The size of the buffer should be a power of two, and can be anywhere from 8 to 256. The larger the buffer, the more complex the random number generation. Before we can use the state variable, we must initialize it properly 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, we can then set the initial seed for the random number generator using `srandom_r` as is done on Line 39. Here we see that compared to the non-thread-safe version, the state variable is also passed 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 we dynamically allocated memory for the state variable within each thread, we need to make sure that memory is released again before the thread is destroyed as is done on Line 48.

By making the above changes, we now get what we expected given the limitation that each thread was started with the same seed (for demonstration purposes) as shown below.

63	63	63	63	63	63	63	63
40	40	40	40	40	40	40	40
15	15	15	15	15	15	15	15

Of course, what we really wanted was to have each column of numbers be independent, and to do that, we need each thread 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. One difference is that for that generator, the extra state storage space is not required, it is maintained by the `struct drand48_data` structure and initialized when it is seeded using the function `strnd48_r(seed, state)`. 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("%d\t", nums[k][i]);
47     printf("\n");
48 }
49
50 return 0;
51 }
```

The first difference to note is that the random number generator state has a different type, as shown on Line 25, which again must be private for each thread. We allocate the space for the state 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, since we aren't using the mod function, we can simply retrieve the random values and store them directly into the data array on Line 38. Otherwise, the structure of this example is the same as before.

Chapter 15

Projects for OpenMP 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 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 help give hints how to best take advantage of the OpenMP system, while for others, it should be obvious, such as using `#pragma omp for` to parallelize just be subdividing 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 as a function of N . Use the plot 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 σ should be given on the command line. Use $\alpha = 1$, which should be defined in your program. Compute the time required to complete the calculation as a function of NM and plot the results. 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 sample paths, and the resulting binning of the data can be handled by using an `omp critical` or `omp atomic` to ensure that different threads do not conflict when updating the terminal location.

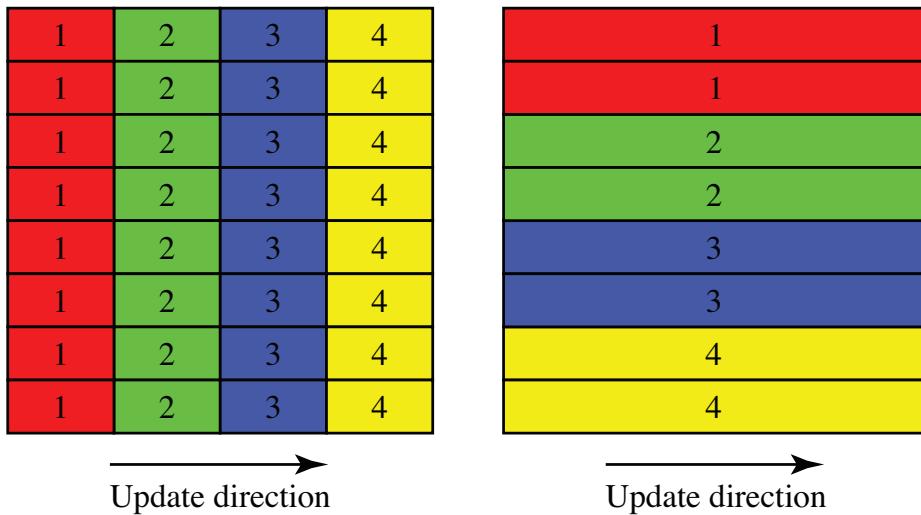


Figure 15.1. Organization of threads for doing updates in the x direction. On the left, each row is updated by subdividing the inner loop. On the right, each row is updated by a single thread, and the rows are subdivided among the threads. Either method will work for an explicit update, but only the right hand case can be used for an implicit update.

15.2 ▪ Finite Difference Methods

In the ADI method, there are both explicit and implicit temporal updates. Explicit updates can be handled by using the `omp_for` 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.

A tridiagonal solver is not so easily parallelizable because of the sequential dependency of the algorithm. Consequently, for the implicit update, each thread will call the tridiagonal solver in LAPACK to do the matrix inversion for a single row or column, hence only the right hand case in Figure 15.1 can be used.

15.2.1 ▪ Brusselator Reaction

Program the assignment in Section 35.3.1 using an $N \times N$ grid. Measure the time required to complete the calculation as a function of N . Compare with the time required to solve the method using a serial version of the code.

15.2.2 ▪ Linearized Euler Equations

Program the assignment in Section 35.3.2 using an $N \times N$ grid. Measure the time required to complete the calculation as a function of N .

15.3 ▪ Elliptic Equations and SOR

As discussed in Chapter 36, a parallel implementation of the 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 outer loop. The two choices are illustrated in Figure 15.2. If the inner loop is subdivided, then each thread takes a piece of

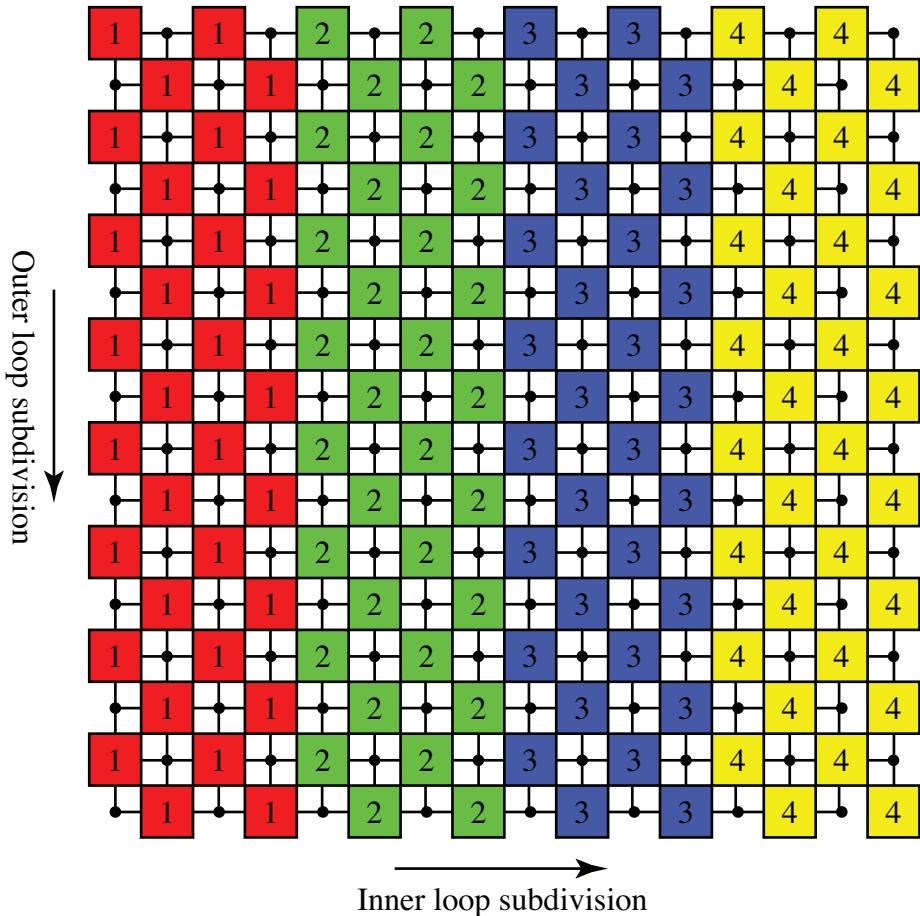


Figure 15.2. Organization of threads for doing updates for red/black ordering in SOR. This figure represents updating the red points. The other points are black points and would be updated afterward. If the inner loop is subdivided, then the data is updated row by row as illustrated. If the outer loop is updated, then the columns represent the organization, where each column is updated by a single thread.

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, then the black grid points are updated in the same manner.

15.3.1 • Diffusion with Sinks and Sources

Program the assignment in Section 36.1.1 using an $(2N - 1) \times N$ grid. Experiment to find the optimal ω that requires the fewest iterations to reach a tolerance of 10^{-9} . Compute the time required to complete a single pass through the grid. Repeat these steps for a grid of dimensions $(4N - 1) \times 2N$. Does the optimal value of ω 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 $N \times N$. For a serial implementation, it is not necessary to use red-black ordering, though you may wish to do it to prepare for implementations later in this book. Experiment to find the optimal ω 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 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 $N \times N \times N$. For a serial implementation, it is not necessary to use red-black ordering, though you may wish to do it to prepare for implementations later in this book. Experiment to find the optimal ω 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.

15.4 • Pseudo-Spectral Methods

The FFTW package for computing the Fourier transform is already multi-threaded, so it is best to perform the Fourier transforms using the package without attempting to use OpenMP directly. However, the computation of the spatial derivatives requires looping through the Fourier modes, and 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. 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. Compute the time required to execute your code as a function of N .

15.4.2 • 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. Compute the time required to execute your code as a function of N .

Part III

Distributed Programming and MPI

So far we have been focusing 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 we need to be able to write programs that can communicate across multiple nodes (i.e. multiple independent computers coupled together). 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 are where a core is unable to proceed and hence stalls because it is waiting for information to arrive upon which it depends. Communication costs are the cost of sending data from one node to another. Ideally, you want to send as little data as possible between different nodes, while keeping all processors as active as possible. 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 your laptop, and can also be used 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 what's called 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 the computer resources. If some small program is running on one node while someone else is trying to do a large computation that includes that same node, then the timing of the large computation can be disrupted so that all the cores in the computation end up waiting around from time to time leading to significantly less efficiency. Therefore, we will also discuss some basics of shell scripting and job submissions with the understanding that there may be significant differences between systems. Check with your systems administrator to get the job scheduling commands that are set up for your system.

A brief description of one 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 can 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. Additionally, each of these communication types come in blocking and non-blocking 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 amongst subgroups.

Chapter 19 covers two useful discussions about efficiency and not wasting the val-

able computing resource. On the efficiency side, there is a discussion about how efficiency is measured compared to serial computing. These measures are important not only 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 recommend 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 a 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.

Chapter 16

Preliminaries

MPI can be used on a single node, and for debugging purposes that is sometimes a good way to test algorithms on small data sets before launching onto a large cluster. In that case, MPI codes can be launched at the command line as was done in the previous parts of this book. However, 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 you request are started at the same time and that communication channels between the cores are established. In order to accomplish this, you will need to learn about submitting jobs through a queuing system, and how to write elementary shell scripts. In this chapter, we will begin with a parallel version of “Hello World!” and also learn how to request a cluster to perform the task on each of multiple cores. We will also discuss the new compiler name, `mpicc`, and how to run programs using `mpirun`.

16.1 • Submitting Jobs

You may be very accustomed to having your own computer and being able to run programs whenever you want, and get the answer as soon as it is completed. Computer clusters, on the other hand, are a shared resource so everyone must wait to take their turn. 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 *head node*, which is the computer that runs very much the same as your own computer, handling multiple tasks at the same time. However, the *compute nodes* are where the parallel algorithms are executed, and they are not shared.

When you want to run a program on the compute nodes, it is called a *job*, and to run it you submit it to a *scheduler*, also called a *queuing system*. The scheduler takes many factors into account about the jobs in the queue such as how much memory is required, how many cores are required, how much time will be required, and what your priority status is (i.e. did you pay money to run this job?). The latter data is maintained by the system and beyond your control, 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 in the marketplace, 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 gives the sequence of commands necessary to run your program. A shell script is yet another language for you to learn, but you won't need too much of it. For our purposes, we'll use a very simple shell script given here.

Example 16.1.

```

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 so you can adjust your code appropriately.

Line 1: This is simply an alert to the system that this is a bash 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 your job is set here, when you want to check on the status of a job, or when a job is finished and sends you a note, this name is what will be used as a reference.

Line 3: Your account number is put here.

Line 4: This line is when you want information about your job emailed to you. For this to work, you set the file `~/.forward` to contain an email address where you want notices to be sent. The letters `abe` refer to (a)abort, (b)egin, and (e)nd corresponding to the events for which you wish to receive messages. You may choose a subset of those letters and put them in any order.

Line 5: This is where you state how many resources your job will require. The number of processes you will need should be less than or equal to the number of nodes times the number of processes (cores) per node. Make sure you know how many cores are available in each node because if you request processes per node than are available your job will either be rejected or you may get significant performance degradation due to multiple processes sharing time on a node. Communication costs within a single node are *much* cheaper than between different nodes, so you want to keep `N` as small as possible and `p` as large as possible within the confines of your system parameters. In the end, when you run your MPI code with the option “`-np #`”, the number you specify on this line should be less than or equal to the number of nodes times the number of processes per node requested on this line.

Queue	Time Limit	Max Cores
short	4 hours	1024
normal	48 hours	1536
long	7 days	512

Table 16.1. Available queues on Quest. The listed amounts are maximums for that queue. The less resources you require, the sooner your job will be executed.

- Line 6: If you don't handle your communications calls correctly, you 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. When you're not running the code on your terminal, you can't easily hit control-C to stop it. This line specifies the length of time you expect your code will take to run. It's a useful backstop to keep your program from idling indefinitely. The amount of time you allocate is important because it will also determine which queue you will end up in when submitting your job. If you specify less time than required, your job will terminate prematurely. If you specify too much time, you could end up in a longer queue which will mean your program may wait longer before it is executed. Providing a fair estimate of what resources you will need is the best way to achieve a positive outcome in as little time as possible.
- Line 7: Depending on your configuration, this line may be optional or unnecessary. Different systems will 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 into the tray where there is continuous turnover of blocks. The scheduler sorts the blocks by size. 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 are 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 your program will be stored in that directory and not cluttering up your home directory.
- Line 9: On some systems, the collections of libraries you will be using may have to be specifically loaded onto the system before they can be used. Because this script will be run when you are not necessarily at the computer, the environment for running your jobs has to be set up. Since we are doing MPI, we will need the MPI tools in order to use `mpirun`. Therefore, the MPI module of libraries are loaded prior to running the program. If you are using LAPACK, or other libraries, you may need to load them here as well.
- Line 10: This is where your program is actually executed, and the number of processes should be no more than the number of nodes times the number of cores per node specified on Line 5.

Your shell script should be saved in a file with suffix “.sh” indicating that it is a shell script.

Once you’ve created your shell script, the next task is to submit the job to the scheduler. Suppose your shell script has the final name “myjob.sh,” then the job is submitted to the scheduler by issuing the command

```
$ msbatch myjob.sh
```

After the job is submitted, you can monitor the status of your job, or manage its status in the queue using additional commands. You can see the queue by using the `showq` command. 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 be different for your system, so check with your administrator for the corresponding tools.

16.2 ▪ Parallel Hello World!

The first thing to grasp when writing code using MPI is that the code you write will be used on multiple processes simultaneously. That means that if you took your original Hello World program from Part I and ran it in a parallel environment, say using 4 cores, then you would get 4 copies of “Hello World” spit back at you. The four different executions would not know about each other and will happily dump their output to your terminal. So how do you get different processes to do *different* things in parallel. That is where MPI comes into play. You want 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 16.2 shows a bare-bones MPI program that has each process report its status.

Example 16.2.

```

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
12 Outputs: Prints "Hello World #" where # is the rank of the process.
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 // Determine the number of processes
27 int size;
28 MPI_Comm_size(MPI_COMM_WORLD, &size);
29
30 printf("Hello World! I am process %d out of %d\n", rank, size);
31
32 // Must shut down the MPI system when you're done.
33 // If you don't, there can be lots of junk left behind.
34 MPI_Finalize();
35
36 return 0;
37 }
38 }
```

First note that we are going to be using a collection of new functions designed for interprocess communications, so we need the function declarations in a header file. That header file is included on Line 2.

When an MPI program is run, the number of processes to be used is defined at run time by using the argument `-np #` where `#` is the number of processes. This argument is not something your 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 our function 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 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 you can process them in your program if you use them. Unless you know what you are doing, **the first line of your 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. The rank is obtained by calling `MPI_Comm_rank`. The first argument to that function indicates that we are asking from among all the processes available to this job. It is possible to create subgroups of processes, in which case, you can have a rank within a certain subgroup. For now, we'll stick with using a single group. The total number of 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.

16.3 • Compiling and Running MPI Code

To compile the program, we could use our usual compiler, e.g. `gcc`, but that would require a bunch of include directives, library linkages, etc. that are specific to the machine on which you are working. To simplify this process, a wrapper program is created that handles all that. Therefore, to compile the program, you enter

```
$ mpicc -o hello main.c
```

Think of `mpicc` as the MPI version of `gcc`. Under the hood it is still `gcc` or whatever compiler you are using.

Similarly, there is a wrapper program for running MPI programs called `mpirun`. To run this program on four processes, we type

```
$ mpirun -np 4 hello
```

The default number of processes to be used is one, so to actually use more than one core, we must specify the number of processes, and that is done using the option `-np 4`. This argument gets passed through to the `MPI_Init` function at the start of our program allowing MPI to set up for the processes.

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 that the order in which the processes report their output may be different. This shows how the different processes are independent and are not synchronized automatically.

Exercises

- 16.1. Compile and run the code in Example 16.2. 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 simply 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. Now that we can create multiple processes, we need to get them to talk to each other by passing messages. In this chapter, we will cover 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. We will also discuss blocking versus non-blocking communications so that we can overlap computation with communications for less latency time.

17.1 • Blocking Send and Receive

Considering again Example 16.2, we can force the order in which the processes report by forcing them to talk to each other so that they don’t go too early. What we will do is 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, we can sketch out the strategy for each of the processes as shown in Table 17.1 where we have aligned the steps for rank 0 next 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.

The next question is how to send a message. The basic message functions come in pairs consisting of a Send and a Receive. The declaration for these functions are shown below:

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 $N-1$	

Table 17.1. Steps required for the ordered sequence of printing "Hello" by using communications

<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. in our case `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 you have 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 the 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 /*
5 int main(int argc, char* argv[])
6
7 The main program is the starting point for an MPI program.
8 This one simply prints each process rank in order. Sorting is done
9 by using message passing to coordinate.
10
11 Inputs: none
12
13 Outputs: Prints "Hello World #" where # is the rank of the process
14 in order.
15
16 */

```

```

17 int main(int argc, char* argv[])
18 {
19     // First thing is to initialize the MPI interface. Some arguments can
20     // be passed through to the MPI interface, so the argument list is
21     // sent by sending the argument list
22     MPI_Init(&argc, &argv);
23
24     // Determine the rank of the current process
25     int rank;
26     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
27
28     // Determine the number of processes
29     int size;
30     MPI_Comm_size(MPI_COMM_WORLD, &size);
31
32     int data;
33     // If not the first process, wait for permission to proceed
34     if (rank > 0) {
35         // Wait for a message from rank-1 process
36         MPI_Recv(&data, 1, MPI_INT, rank-1, 0, MPI_COMM_WORLD,
37                  MPI_STATUS_IGNORE);
38         printf("Rank %d has received message with data %d from rank %d\n",
39                rank, data, rank-1);
40     }
41
42     // All processes print hello
43     printf("Hello from rank %d out of %d\n", rank, size);
44
45     // All processes send the go ahead message except the last process
46     if (rank < size-1) {
47         data = rank*rank;
48         // Send the go ahead message to rank+1. Using rank^2 as fake data
49         MPI_Send(&data, 1, MPI_INT, rank+1, 0, MPI_COMM_WORLD);
50     }
51
52     // Must shut down the MPI system when you're done.
53     // If you don't, there can be lots of junk left behind.
54     MPI_Finalize();
55
56     return 0;
57 }

```

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 our 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, we see that the first task for all processes except the first one is to wait for the green light to proceed. This is done on Line 37, where we are receiving one integer (`MPI_INT`) from the process of one lower rank. The ID tag for the message is arbitrarily set to zero. We are also not going to make use of the status information, so we'll just ignore it using the pre-defined `MPI_STATUS_IGNORE`. On Line 39, we show when the message is received and what information was received.

On Line 44 we print our “Hello” phrase as we did before. All processes must do this task.

Finally, after printing the “Hello” phrase, we send the message to the next one in the chain. This is done on Line 50. Note how the function arguments match up with the receive function on Line 37.

At first, it may seem counter-intuitive that the receive should be written before the send, but why that sequence is appropriate is easily seen by comparing to Table 17.1. The proof is in the output, which is 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 as well. For example, suppose we had accidentally forgotten to exclude the rank 0 process from the initial receive, i.e. left out Line 35, then the program would get stuck, unable to proceed. It's equivalent to an infinite loop in a program, each process would wait indefinitely for the message that would 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 ■ Non-Blocking 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 non-blocking communications.

Non-blocking 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, let's look at the steps of this dance where process rank 0 wants to send a message to process rank 1:

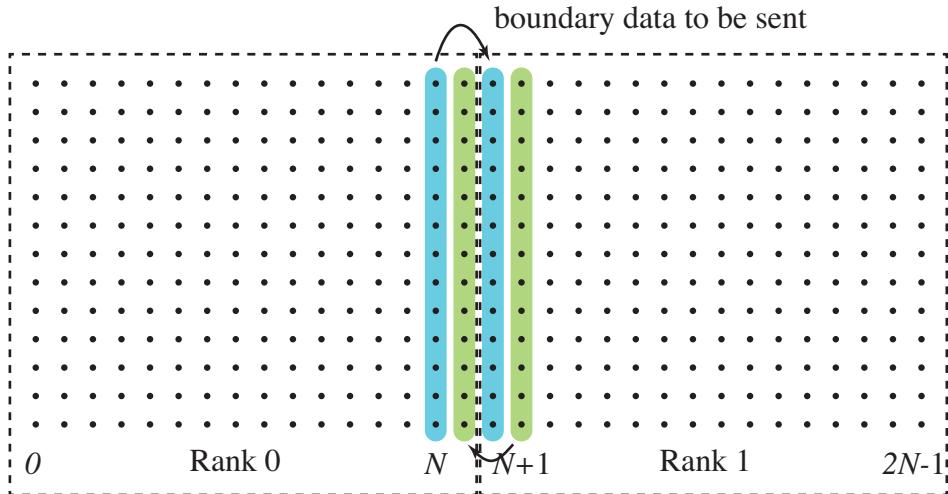


Figure 17.1. Simple layout for domain decomposition of a finite difference grid. Rank 0 will send its boundary data to Rank 1, and Rank 1 will do the same in the opposite direction following the arrows. The color-coded data will be the same after the communication transactions. This will require two sends, and two receives, one in each direction.

Rank 0	Rank 1
Identify data to be sent	Identify where data will be stored
Issue non-blocking send command	Issue non-blocking receive command
Do work that does not alter the data to be sent	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, let's explore a real application. Suppose we are solving the heat equation using a simple explicit finite difference time-stepping method. Rank 0 will work with the grid values $u_{i,j}^n$ for $i = 0, \dots, N$ at time step n , and Rank 1 will work with grid values $i = N + 1, \dots, 2N - 1$ as illustrated in Figure 17.1. Suppose that 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 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 $i = 0, \dots, N+1$	Allocate array $u_{i,j}^0$ for $i = N, \dots, 2N-1$
Initialize $u_{i,j}^0 = f(x_i, y_j)$ for $i = 0, \dots, N$	Initialize $u_{i,j}^0 = f(x_i, y_j)$ for $i = N+1, \dots, 2N-1$
Begin time step loop	Begin time step loop
Do non-blocking send of blue data, $u_{N,j}^n$, to Rank 1	Do non-blocking send of green data, $u_{N+1,j}^n$, to Rank 0
Begin non-blocking receive of green data, $u_{N+1,j}^n$, from Rank 1	Begin non-blocking receive of blue data, $u_{N,j}^n$, from Rank 0
Compute $u_{i,j}^{n+1}$ for $i = 0, \dots, N-1$	Compute $u_{i,j}^{n+1}$ for $i = N+2, \dots, 2N-1$
Wait for both non-blocking send and receive to be completed	Wait for both non-blocking send and receive to be completed
Compute $u_{N,j}^{n+1}$	Compute $u_{N+1,j}^{n+1}$
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, we could have initialized the data for Rank 0 for $i = 0, \dots, N+1$ and then skipped the first send/receive pair. The cost savings would be minimal since this would only be valid on the first time step, and at the same time, it would have made writing the algorithm a little more complicated, so for purposes of the notes 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 the Wait command is issued, it is very unlikely that any waiting will actually take place because by the time all the data in the interior of the two subdomains is 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 location for $u_{N,j}^n$ and $u_{N,j+1}^n$ must be adjacent. In this simple example, it's easy to do that, but if we're talking about a 3D domain subdivided into 4^3 cubic subdomains, it gets much more difficult. In that case, it may be necessary to set up a special buffer 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 receive must be completed before proceeding**. Obviously, Rank 0 doesn't want to update the $i = N$ data until the boundary data at $i = N+1$ has been received. At the same time, even if the data at $i = N+1$ is ready to go, the data at $i = N$ 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 $u_{N,j}^n$ gets updated to $u_{N,j}^{n+1}$ before the send is complete, Rank 1 will receive the $u_{N,j}^{n+1}$ 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 non-blocking communications. In practical terms, a separate array, call it v_j , just for storing $u_{N,j}^n$ could be created. Before doing the non-blocking send, the grid data would be copied into this buffer, i.e. $v_j = u_{N,j}^n$, and then the non-blocking send would send the v_j data rather than $u_{N,j}^n$. By doing this, the $u_{N,j}^{n+1}$ 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 non-blocking 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 you are ready to update the boundary data.

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 int main(int argc, char* argv[])
20 {
21     // First thing is to initialize the MPI interface.
22     // Some arguments can be passed through to the MPI interface,
23     // so the argument list is sent by sending the argument list
24     MPI_Init(&argc, &argv);
25
26     int N = atoi(argv[1]); // Get the number of grid points
27
28     // Determine the rank of the current process
29     int rank;
30     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
31
32     // Determine the number of processes
33     int size;
34     MPI_Comm_size(MPI_COMM_WORLD, &size);
35
36     // allocate memory. Each core should have roughly N/size + 2 grid
37     // points, but if N is not evenly divisible by size, give one extra
38     // grid point to the low ranks to make up the difference. Also, the
39     // rank=0 and rank=size-1 processes do not have to store boundary
40     // data for a neighbor.
41     int localN = N/size + (N%size > rank ? 1 : 0) + (rank > 0 ? 1 : 0)
42                                         + (rank < size-1 ? 1 : 0);
43
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.;
```

```

53  if (rank == 0) {
54      // Read the boundary conditions from the input list.
55      // sscanf is like scanf except it reads from a string
56      // rather than the keyboard.
57      // Get the left boundary condition.
58      sscanf(argv[2], "%lf", &(u[0][0]));
59      u[1][0] = u[0][0];
60  }
61
62  if (rank == size-1) {
63      // Get the right boundary condition
64      sscanf(argv[3], "%lf", &(u[0][localN-1]));
65      u[1][localN-1] = u[0][localN-1];
66  }
67
68  // CFL condition: dt < 0.5 dx^2
69  double dx = 1.0/(N-1);
70  double dx2 = dx*dx;
71  double dt = 0.25 * dx2;
72
73  // Storage for tracking communication info
74  MPI_Request sendLeftRequest;
75  MPI_Request sendRightRequest;
76  MPI_Request recvLeftRequest;
77  MPI_Request recvRightRequest;
78
79  // main loop: terminal time is T=1
80  int i, newi, oldi;
81  for (i=0; i*dt < 1.0; ++i) {
82      newi = (i+1)%2;
83      oldi = i%2;
84      // Exchange end data to the left,
85      // the tag will correspond to the step #.
86      if (rank > 0) {
87          MPI_Isend(&(u[oldi][1]), 1, MPI_DOUBLE, rank-1, i, MPI_COMM_WORLD,
88                      &sendLeftRequest);
89          MPI_Irecv(&(u[newi][0]), 1, MPI_DOUBLE, rank-1, i, MPI_COMM_WORLD,
90                      &recvLeftRequest);
91      }
92      // Exchange end data to the right,
93      // the tag will correspond to the step #
94      if (rank < size-1) {
95          MPI_Isend(&(u[oldi][localN-2]), 1, MPI_DOUBLE, rank+1, i,
96                      MPI_COMM_WORLD, &sendRightRequest);
97          MPI_Irecv(&(u[newi][localN-1]), 1, MPI_DOUBLE, rank+1, i,
98                      MPI_COMM_WORLD, &recvRightRequest);
99      }
100
101     // Now do update in the interior where it doesn't matter if we have
102     // the current data from neighboring processes
103     for (int j=2; j<localN-2; ++j)
104         u[newi][j] = u[oldi][j]+dt*(u[oldi][j+1] - 2*u[oldi][j]
105                                  + u[oldi][j-1])/dx2;
106
107     // Can update points next to boundary
108     if (rank == 0)
109         u[newi][1] = u[oldi][1]+dt*(u[oldi][2] - 2*u[oldi][1]
110                                  + u[oldi][0])/dx2;
111     if (rank == size-1)
112         u[newi][localN-2] = u[oldi][localN-2]+dt*(u[oldi][localN-1]
113                                     - 2*u[oldi][localN-2] + u[oldi][localN-3])/dx2;
114
115     // Now check to see boundary data is ready.

```

```

116    // Indicate whether data is ready {sent left , received left ,
117    // sent right, received right}
118    int ready[4] = {0, 0, 0, 0};
119    // Indicate that the update of the end point has been done after the
120    // data transfer.
121    bool done[2] = {false , false};
122
123    // There is no data transfer at the ends, so mark those as ready.
124    if (rank == 0) {
125        ready[0] = 1;
126        ready[1] = 1;
127    }
128    if (rank == size-1) {
129        ready[2] = 1;
130        ready[3] = 1;
131    }
132
133    // Keep checking until data is ready and end points updated.
134    while (!done[0] || !done[1]) {
135
136        // Check whether interchange of left data is complete
137        if (rank > 0 && !done[0]) {
138            if (!ready[0])
139                MPI_Test(&sendLeftRequest, &(ready[0]), MPI_STATUS_IGNORE);
140            if (!ready[1])
141                MPI_Test(&recvLeftRequest, &(ready[1]), MPI_STATUS_IGNORE);
142        }
143
144        // If the data is exchanged and endpoint hasn't been updated yet,
145        // then update it.
146        if (ready[0] && ready[1] && !done[0]) {
147            // data on the left has been sent and received, it's safe to
148            // update now
149            u[newi][1] = u[oldi][1] + dt*(u[oldi][2] - 2*u[oldi][1]
150                                         + u[oldi][0])/dx2;
151            done[0] = true;
152        }
153
154        // Check whether interchange of right data is complete
155        if (rank < size-1 && !done[1]) {
156            if (!ready[2])
157                MPI_Test(&sendRightRequest, &(ready[2]), MPI_STATUS_IGNORE);
158            if (!ready[3])
159                MPI_Test(&recvRightRequest, &(ready[3]), MPI_STATUS_IGNORE);
160        }
161
162        // If the data is exchanged and endpoint hasn't been updated yet,
163        // then update it
164        if (ready[2] && ready[3] && !done[1]) {
165            // data on the right has been sent and received, it's safe to
166            // update now
167            u[newi][localN-2] = u[oldi][localN-2] + dt*(u[oldi][localN-1]
168                                         - 2*u[oldi][localN-2] + u[oldi][localN-3])/dx2;
169            done[1] = true;
170        }
171    }
172
173    // We have to gather all the data from the various
174    // processes so it can be printed.
175    // We'll send all the data to rank==0.
176    // We'll see a better way to do this later.
177    if (rank == 0) {

```

```

179 // allocate space for the full array
180 double* finalu = (double*)malloc(N * sizeof(double));
181
182 // copy the local data to the full array
183 for (int j=0; j<localN-1; ++j)
184     finalu[j] = u[newi][j];
185
186 // Track where the next array data will be inserted in the
187 // full array
188 int nextj = localN-1;
189 int datalen;
190
191 // request the data from each of the other processes ,
192 // appending as we go.
193 for (int r=1; r<size; ++r) {
194     // amount of data in rank=r process excluding internal boundary
195     // data
196     datalen = N/size + (N%size > r ? 1 : 0);
197     MPI_Recv(&(finalu[nextj]), datalen, MPI_DOUBLE, r, 0,
198             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
199     nextj += datalen; // update the new insertion point
200 }
201
202 // Store the final array in matlab format to file finalu.m
203 FILE* fileid = fopen("finalu.dat", "w");
204 for (int j=0; j<N; ++j)
205     fprintf(fileid, "%f %le\n", j*dx, finalu[j]);
206 fclose(fileid);
207
208 // Release the finalu allocation of memory
209 free(finalu);
210
211 } else {
212     // All ranks other than 0 must send their interior data to rank 0
213     // The last rank doesn't have a boundary point,
214     // so it's one bigger than the others.
215     if (rank < size-1)
216         MPI_Send(&(u[oldi][1]), localN-2, MPI_DOUBLE, 0, 0,
217                   MPI_COMM_WORLD);
218     else
219         MPI_Send(&(u[oldi][1]), localN-1, MPI_DOUBLE, 0, 0,
220                   MPI_COMM_WORLD);
221 }
222
223 free(u[0]);
224 free(u[1]);
225
226 // Must shut down the MPI system when you're done.
227 // If you don't, there can be lots of junk left behind.
228 MPI_Finalize();
229
230 return 0;
231 }
```

The example 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. In this example we are using a simple forward Euler time step method, so we will need two copies of the data. The data is initialized in Lines 50–

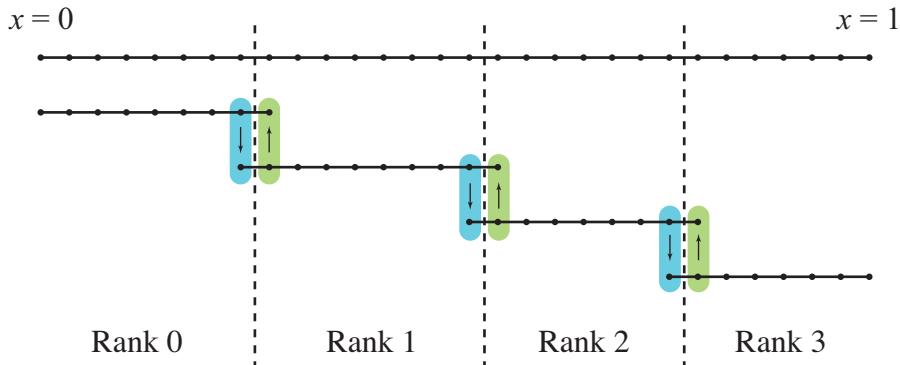


Figure 17.2. Diagram of local data allocation for the 1D heat equation. The colored points indicate points that must be synchronized across processes, and the arrow indicates the flow of the communication. The processes only update the heat equation for the nodes between the dotted lines.

66. 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 69–71 following the CFL condition.

Before looking at the main loop, let's look at the non-blocking communications call format. The two matching functions are `MPI_Isend` and `MPI_Irecv`, and their call statements are similar to the blocking send and receive we learned 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're just requests to start the process of communications. As discussed above, that data that is involved in a send or receive pair should not be used before the transaction is completed, so that means you have to check for completion, and that is what the `MPI_Request` object is for, so you can keep track of all your outstanding communication requests.

Like the send and receive, there are blocking and non-blocking 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. However, in our code, we have two sets of requests for each end of the grid, and the two ends are independent of each other. We don't want to get stuck waiting for the left end if the right end is ready to go and can be acted on. In that case, it is better to use a non-blocking check and continue to loop through open communications and acting

accordingly as they become available, and that is done by using MPI_Test:

```
MPI_Test(MPI_Request* request, int* ready, MPI_Status* status)
```

The value of `ready` will be non-zero if the communication is complete, and zero if it is incomplete. With this information in hand, we can then understand the rest of Example 17.2.

As we discussed in our algorithm outline, the first thing we do in the main loop is initiate the communications of boundary data, which is done on Lines 86–99. Depending on which process we are in, a request to send and a request to receive boundary data for both ends of the local domain is 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.

In case you haven't seen this trick before, the indexing in the arrays on Line 104 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 time-waster. It can be avoided by just going back and forth. On Line 104 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 we see that `u[0]` is on the left side and `u[1]` is on the right side.

Next we deal with completing the communications started at the top of the loop. We assume none of the transactions are ready and that the end point updates have not been done (except at $x = 0, 1$). We then begin looping through the communications, checking on them as we go. On Lines 137–142, we check whether the left end is ready and we only keep checking if it isn't ready. If the communications are ready and we haven't already updated the end point value, then we update it on Lines 149–151. Lines 155–170 are the same construction but for the right end. Once both ends are updated, the loop terminates and we're able to go to the next time step.

Finally, on Lines 178–221, the data is gathered in order to store to a file. Remember, no single process has a complete set of the data, so we have to communicate the data to the lead process, which we've designated as rank 0. The rank 0 process creates a single array to store the data on Line 180, and then copies its local data to it on Lines 183–184. 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 193–200, while the remaining processes must send their data, which is on Lines 215–221. Once the data is retrieved in the rank 0 process, it is written to a single file on Lines 203–206. We will see next that there are better ways to do group send and receives such as this.

17.3 ▪ Gather/Scatter Communications

In Example 17.2, we made things perhaps a little too general because we allowed different processes to have different sized subdomains. Suppose we had planned ahead a little better and required each subprocess to be responsible for exactly the same number of nodes, then the process of gathering all the data into a single array, as was done on Lines 178–221, could be simplified down to a single command, MPI_Gather. It would look like this:

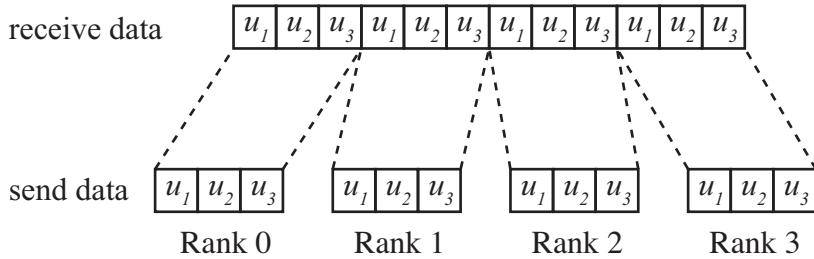


Figure 17.3. Diagram of the MPI_Gather operation. The different processes send their data to the destination process where it is assembled into a single concatenated array.

```

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 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 our example 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 size, while `MPI_Gather` requires them all to be the same size. Since it is quite common for there to be different sizes for each buffer, it makes sense to allow for this, and that is the case for the more general `MPI_Gatherv`:

```
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 we wish to read the initial data for our heat equation problem from a file. In that case, the data could be read by the rank 0 process, but it then has to 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 )
```

As for `MPI_Gather`, the send and receive counts should be the same, as well as the send and receive data types. 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. Again, all processes must be sure to allocate sufficient space for the data before this call.

17.4 • Broadcast and Reduction

As we discussed in Part I of this course, it is best to control your 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 your main process read the file, and then distribute the information to the rest of the processes. Of course, we could use the point to point type of communication of the send/receive pairs, but that would require many communications calls. Instead, the tool to use is a 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 )
```

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, to send the time step size `dt` to all the processes, you would do the following:

```
1 double dt;
2 if (rank == 0)
3     dt = 0.25/dx/dx;
4 MPI_Bcast(&dt, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
5 // dt is now set for all processes
```

As long as we're 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, you might want to know the maximum value of a variable across all processes, or the sum of all the values across the processes. This is where a reduction comes in handy. 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:

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

Table 17.2. Sample list of operations available for the MPI_Reduce function.

```
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 will contribute, hence `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*`, you should make sure that both are defined to be of the same type. The rest of the arguments are similar to what we have seen before for 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 are listed in Table 17.2, many more operators, and even user-defined operators, can also be used. The result of the operation will be available to the destination process only. To make the result available to all processes, use the function `MPI_Allreduce`, which has the same arguments minus the destination rank.

Exercises

- 17.1. Modify Example 17.1 so that the rank 3 process sends a message back to rank 0, and then 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 non-blocking communications.
- 17.3. Write a program where each core creates an array of length N and populates it with integer values $1000*rank+j$ 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.
- 17.4. Write a program to broadcast an integer N to all processes, and then use MPI_Reduce to sum the values. Verify this results in the value $P*N$ where P is the number of processes.

Chapter 18

Groups and Communicators

For some applications, it is useful, or sometimes necessary, to subdivide the processes into functional groups. For example, suppose you wish to break down a grid into a 2-dimensional array of subgrids, and you want to have communications shared within each row or column. This isn't possible with what we have done so far except for explicitly point-to-point communications because we have been exclusively using the communications group `MPI_COMM_WORLD` that includes all processes. 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, we will do three examples showing different ways in which a 2×3 grid of processes can be organized into rows and columns so that each column or each row can do shared communications. Before we embark on this, it's important to understand that 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, by default we have been using the `MPI_Group` corresponding to the entire list of available processes, but we will soon see how to create subgroups. On top of a given `MPI_Group` there can be different communication patterns, and that is determined by the type `MPI_Comm`. We've been using the same `MPI_Comm` every time so far, namely `MPI_COMM_WORLD`, but we can create new ones, which is what this section is about.

There are two components to setting up group communications for a subpopulation of the processes. First, the subpopulation must be identified, and then 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, you will see different ways of creating group communicators that will enable group communications among a subpopulation of the processes. In all cases, it is assumed that six processes will be organized into a 2×3 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 sub group, 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, non-members 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
77  // Create three column communicators and do a sample broadcast
78  int col_ranks[3][2] = {{0,3},{1,4},{2,5}};
79  MPI_Group col_group[3];
80  MPI_Comm col_comm[3];
81
82  int col;
83  for (col=0; col<3; ++col) {
84      value = world_rank;
85
86      // For each column, set the members of the column as a new group
87      // and then build the corresponding communicator
88      MPI_Group_incl(world_group, 2, col_ranks[col], &col_group[col]);
89      MPI_Comm_create(MPI_COMM_WORLD, col_group[col], &col_comm[col]);
90
91      // Only do the broadcast among processes that are group members
92      if (col_comm[col] != MPI_COMM_NULL) {
93          int col_rank, col_size;
94
95          // Get the local rank and size and broadcast the local rank 0
96          // value of the world_rank
97          MPI_Comm_rank(col_comm[col], &col_rank);
98          MPI_Comm_size(col_comm[col], &col_size);
99          MPI_Bcast(&value, 1, MPI_INT, 0, col_comm[col]);
100         printf("World rank: %d, col: %d, col_rank: %d/%d, value: %d\n",
101                world_rank, col, col_rank, col_size, value);
102     }
103 }
104
105 // Clean up all the created groups and communicators
106 for (row=0; row<2; ++row)
107     if (row_comm[row] != MPI_COMM_NULL) {
108         MPI_Comm_free(&row_comm[row]);
109         MPI_Group_free(&row_group[row]);
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 our new row group. The identification of the processes are sent as an array, which are enumerated on Line 44. Once the subgroup is created, a communicator designed to work on that group can then 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 we are done, since memory has been allocated in the creation of the MPI_Comm and MPI_Group objects, we must also free them as well as shown on Lines 108, 109. Again, because those groups and communicators do not apply to all processes, we must use the guard checking for MPI_COMM_NULL as on Line 107.

The rest of Example 18.1 shows how the columns have group communicators as well. Note that each process can be a member of one or more subgroups and that communicators can overlap in different ways. The key is to make sure that when you use a group communicator that you clearly understand to which group the communicator applies.

Think about what you think this program should output and 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 using 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 example, the 2×3 example we have been using, 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.

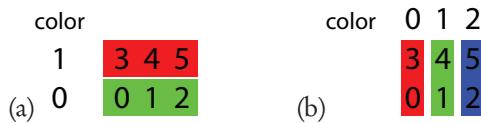


Figure 18.1. Illustration of the color scheme used for the communicators generated with the `MPI_Comm_Split` function in Example 18.2. (a) Color scheme used for the row communicator, (b) color scheme used for the column communicator.

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 sub group, 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 to notice 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 both rows. Similarly, only one MPI_Bcast is used for the column communicator. It makes sense that the type of operations you're doing 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 allow you to organize your processes into grids of any number of dimensions. Each dimension can be made into a loop by designating it as periodic so that when you request the rank of a neighboring process it will loop around rather than give a non-existent 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 sub group, 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], grid_coords[1], value);
73
74  // Create a communicator to link the columns
75  dims_together[0] = 1; dims_together[1] = 0;
76  MPI_Comm col_comm;
77  MPI_Cart_sub(grid_comm, dims_together, &col_comm);
78
79  // Test row communicator by doing a broadcast
80  value = world_rank;
81  MPI_Bcast(&value, 1, MPI_INT, 0, col_comm);
82  printf("Col test: Coordinate: (%d,%d), value = %d\n",
83                  grid_coords[0], grid_coords[1], value);
84
85  MPI_Comm_free(&grid_comm);
86  MPI_Comm_free(&row_comm);
87  MPI_Comm_free(&col_comm);
88
89  MPI_Finalize();
90  return 0;
91 }
```

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 our purposes that will be MPI_COMM_WORLD more often than not. Next the number of dimensions of the grid are 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 2×3 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 our layout means decrementing the column index, would give us 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 world 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 your code is being set up exclusively on the Cartesian grid, and you are doing this right at the beginning, then it makes perfect sense to permit the reordering, but if this is for a

sub-problem where the world rank is used elsewhere, then you should set this to false. The resulting grid communicator is returned through the last argument.

Once the Cartesian grid communicator is created, there are a number of operations that can be done that are useful. First, similar to obtaining the 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 does not have to have value 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 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, we want to lump together, for example, grid processes (0, 0), (0, 1), and (0, 2). In other words, we want to include all entries in the second coordinate. We indicate that 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 you can create groups of varying dimensions such as a 1-dimensional group by setting `dims_included = {1, 0, 0}`, or 2-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 processes of each row or column is broadcast to the corresponding rows and columns as expected and 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 4×4 grid of processes where one group is the red squares and the other group is the black squares of a red/black coloring 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 4×4 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, but just because it's impossible doesn't mean that we can't get as close as possible. But to get there, we need some sort of measure for how close our algorithm gets to the perfect efficiency of using N processes to solve our 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 when you also factor in the size of the cluster, such as 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 you are using a fraction of a cluster, maybe you are lucky and are using the cores not affected, or maybe you're not so lucky. In either case, you need to prepare for disruptions beyond your flawless computing skills, 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 our 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 we use twice as many cores, then we should expect the job to 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 than the others 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, we want to have some way of measuring the efficiency of our use of the parallel architecture. There are two fundamental measures that test algorithm efficiency: strong scaling and weak scaling. *Strong scaling* is measuring how much time is required to complete a fixed size problem compared to the number of processors used. A perfectly efficient algorithm is strongly scalable if a problem that takes time T for one processor requires time T/N if N processors are used. *Weak scaling* is a measure of how the time to completion changes when the size of the problem grows with the number of processors. In this case, suppose you have problem that has a 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 changes, and the number of processors changes by the same factor, then the time to compute the solution remains constant. For example, if you double the total number of grid points in a solver for a PDE, and you double the number of processes used to do the computation, does the time T remain the same?

To test for strong scaling, it's a fairly simple task. If you have a code that solves a problem and you run it once with, say 10 processes, and then you run it again with 20 processes, does the elapsed time get cut in half? To test this, you would collect data on the time elapsed for various size problems. Let T_N be the time to completion for a fixed problem using N processes. The strong scaling efficiency is then measured by

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

To test for weak scaling, it requires your problem to be able to change in size. For example, if you are solving a PDE problem, the measure of problem size could be the number of grid points in the problem. Let T_1 be the time to solve a problem of size M on a single process, and let T_N be the time to solve a problem of size NM , then the weak scaling efficiency is measured by

$$\text{weak efficiency} = \frac{T_1}{T_N}.$$

For example, suppose you are solving a PDE on a 3D grid of size M^3 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 you now have a grid with $8M^3$ points and solve that problem on 8 processors and let T_8 be the time to solution. The weak efficiency would then be given by T_1/T_8 .

In order to take these measurements, we need to accurately compute the time elapsed. MPI provides functions for this purpose: `MPI_Wtime` and `MPI_Wtick`. If we wanted to measure the efficiency of our heat equation solver in Example 17.2, then we would insert these lines at Line 36:

```
49     double precision = MPI_Wtick();  
50     double starttime = MPI_Wtime();
```

and then insert these lines at Line 225 of Example 17.2:

19.2 ■ Checkpointing

I'm sure this has happened to you at one point in your experience of writing code (if it hasn't, I promise it will); you run your program, it's going well, and then something breaks down. 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 the computer. On a small computer, running small jobs, you shrug, fix whatever caused the problem, and start the program over from scratch.

When doing high performance computing, the resources you are using are not cheap. A single compute-core-hour may cost approximately \$0.05. That may sound cheap, but suppose you are requesting 512 cores for 7 days, the bill would be just over \$4,300 if the job runs to termination only for you to discover that you made an error. And that is a still a comparatively small job compared to the large high-performance systems that are presently in use. **Translate that cost to the Sunway TaihuLight, and that same error cost almost \$90 million!**

The dollar figures are certainly impressive, and they should scare you enough to take greater care in conserving this resource than you would your personal laptop. It goes without saying that you should test your algorithms on smaller systems and shorter times before making a large request for resources. Typically, practitioners will write their code on small systems for thorough testing. These days even the least expensive laptops have at least 2-4 cores so you can test your algorithms quite easily on just about any machine. This much should be obvious. What you may not consider is the process known as checkpointing.

Checkpointing is where your code should periodically store enough information about its state that it could pick up where it left off by reading a checkpoint file. How you do this is completely up to you so long as it works. What this means is that it is not sufficient to simply dump your output to a file without also storing any other data that may be relevant to continuing. Furthermore, your code should have a built-in capability to read checkpoint data and then resume the computation from where you last stored the data. On some systems, jobs that are terminated due to some sort of system failure would be expected to be able to automatically restart from checkpoint data without user intervention by using the same job script. Checkpointing is an important part of high performance computing and you will be expected to include that in your codes.

As an example of checkpointing, let us consider the 1D heat equation example from earlier. It is customary to store intermediate time points as part of your output (but please, not every time step!). 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 , we would also store the time step size, the current time step, the spatial discretization, and the terminal time. The extra data can be stored in the same file as your u values, or it could be stored in a separate parameters file that only gets stored once. In addition, your input parameter file should be such that it can use a checkpoint file as a means of retrieving the necessary parameters to run your problem over again.

Checkpointing is often overlooked, but it is a vitally important task as the size of your jobs get larger and larger. And don't forget to *test* the checkpointing ability of your code as well!

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/slugs/>

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 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_`. Here, we present the same code, but for using the ScaLAPACK library version, namely `pdgesv_`.

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

We will go through each of these steps in order.

20.1.1 • Setting up the process grid

For ScaLAPACK, the process grid is a 2-dimensional array arrangement of the processes available. 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 2×3 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 we have seen previously (e.g. ranks 0–5). When setting up the process grid, you want to try and 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)
```

The first argument is an output argument for a single integer. Upon return, the `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 you have N cores to use, 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 context for different problems.

Next, the 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 similar role to the role of `rank` in more general MPI programs, while `pg_rows` and `pg_columns` corresponds to the `size` variable. We will use this information in order 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, we next must put the pieces of the matrix in block-cyclic form onto the process grid. Before we discuss the details of the distribution, we must first understand 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 3 processes. The initial vector is given here:

$$[1, 2, 3, 4, 5, 6, 7, 8, 9]$$

We 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 3 processes like dealing cards from a deck. The first process re-

ceives the first block, the second process receives the second block, the third process receives the third block, and then we begin again with the first process so that the first process receives the fourth block, and so on. After following this procedure, we now have the data divided into 3 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 we have

$$\{1, 2, 7, 8\}, \{3, 4, 9\}, \{5, 6\}.$$

To distribute a matrix, we follow the same steps, but independently for the rows and columns. For example, suppose we wish to distribute a 9×9 matrix onto a process grid of dimensions 2×3 with block size 2×2 . 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, we break the matrix into blocks of size 2×2 :

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 so that we have

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));
```

Verify these formulae are correct for the matrix example above.

Now, it is common that a given system to be solved may originally be on one process, but we want to use the parallel architecture to solve the linear system. Therefore, the matrix must be distributed to the processes within the process grid. To do this, we will use the point to point communications available on the grid. The standard blocking send and receive are given by:

<code>dgesd2d_(</code> <code> int* context,</code> <code> int* m,</code> <code> int* n,</code> <code> double* Asrc,</code> <code> int* nrowssrc,</code> <code> int* prowdest,</code> <code> int* pcoldest)</code>	<code>dgerv2d_(</code> <code> int* context,</code> <code> int* m,</code> <code> int* n,</code> <code> double* Adest,</code> <code> int nrowsdest,</code> <code> int* prowsrsrc,</code> <code> int* pcolsrsrc)</code>
--	---

The first argument is the context from when the process grid was created. 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 receiving side, the leading dimension is `nrowsdest`. The last two arguments are the coordinates to/from which the data is sent/received.

The communications protocol follows a similar naming scheme 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 handful 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

```
descinit_(int* desc, int* m, int* n, int* mb, int* nb,
           int* irsrc, int* icsrc, int* ictxt, 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 $\mathbf{Ax} = \mathbf{b}$, and in the Example 7.2, we used the one time driver function `dgesv_`. To keep in theme, this parallel implementation has a similar function named `pdgesv_`. In general, the attempt is to keep the naming scheme consistent with LAPACK except the additional “p” prefix.

In Example 20.1, is an example of using `pdgesv_`. The function is given by

```
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 \mathbf{A} , the number of columns in the right hand side vector \mathbf{b} , the local data for the matrix stored in block-cyclic order, the row and column of the top left corner of the matrix \mathbf{A} (typically 1,1), the matrix descriptor for \mathbf{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 \mathbf{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, you can release the process grid only by setting `notdone` to true (or 1). If `notdone` is false (or 0), the call to this function also calls `MPI_Finalize()` to shutdown the MPI system.

Also at this time, don't forget to release all the memory allocated for the various work arrays, matrices, and so forth.

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 // function to initialize the matrix when solving Ax=b
18 void matinit(int context, double** A, int** desca, double** b,
19               int** descb, double** work, int** pivot);
20 // function to compute the dimensions of the local matrix
21 int numroc(int dimen, int bsize, int nprowcol);
22 // function for gathering the block-cyclic scattered solution vector
23 // onto the 0,0 process.
24 void gathervec(int ictxt, double* bfull, double* blocal, int* descb);

25 /*
26  * int main(int argc, char* argv[])
27
28  Solve Ax=b on a distributed architecture using ScaLAPACK
29
30  Inputs: none
31
32  Outputs: Prints out the final values.
33
34 */
35
36 int main(int argc, char* argv[])
37 {
38     int ictxt = 0; // The handle to the system context
39     int nprow = 2; // Number of process rows in the grid
40     int npcol = 3; // Number of process columns in the grid
41     int zero = 0;
42     int one = 1;
43     int six = 6;
44     double done = 1.0;
45
46     int myrow;
47     int mycol;
48
49     int* desca = NULL;
50     int* descb = NULL;
51     int* ipiv = NULL;
52     double* work = NULL;
53
54

```

```

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, &nproc, &npcol);
62  blacs_gridinfo_(&ictxt, &nproc, &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], nproc,
73          *numroc(desca[3], desca[5], nproc));
74  double* b0 = (double*)malloc(sizeof(double)
75          *numroc(descb[2], descb[4], nproc));
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 = %le\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 {
194     // 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, &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 rhodata[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] = rhodata[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][ ncol ][ mxloc ][ 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)%ncol; // the process col for this col of A
267             // the local offset for this process col
268             int locj = j%blksize + blksize*(j/(blksize*ncol));
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<ncol; ++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*mxloc; ++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, &mxloc, &(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 /*
304 void gathervec(int ictxt, double* bfull, double* blocal, int* descb)
305
306     This function uses point to point communication to gather the solution

```

```

307    vector stored in block cyclic fashion, into a contiguous single vector
308    in process grid 0,0.
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][loci];
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**. We provide some comments about the code below:

- Lines 5–16: The standard ScaLAPACK package is written in Fortran, while the communications library (BLACS, or (B)asic (L)inear (A)lgebra (C)ommunication (S)ystem) 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 to be used in your code must be declared at the top of your files. The keyword `extern` means that the function argument lists are all empty. That is OK, the compiler will trust you to get the argument list correct. However, when it comes time to link your code against the ScaLAPACK library, the arguments will have to match what is listed in the library or the link will fail.
- 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 what is the current node in the process grid is retrieved on Line 61.
- Line 65: There may be more processes in the system than are required for this process grid. Those processes not involved for 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 `gathervec()` 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 you are finished.
- Line 121: The process grid is shut down here. Because zero is passed as the argument, indicating not not done (i.e. done), 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 your 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 $(\text{loc}_i, \text{loc}_j)$ 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, you will use the commands:

```
$ mpicc -c mympiprogram.c
$ mpicc -o mympiprogram mympiprogram.o -lscalapack -llapack -lblas
```

20.2 • FFTW

One can certainly make use of the serial version of FFTW as described in Section 7.2 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 1D FFT function to do higher dimensional FFTs in parallel using MPI. We will then see how the MPI implementation of FFTs provided by FFTW is much easier if it is available.

20.2.1 • Using Serial FFTs in parallel

Recall that a multi-dimensional FFT is actually a one-dimensional FFT applied first in one dimension, and applied again in a second dimension, and so forth until all dimensions have been computed. We can take advantage of this in a parallel architecture. For simplicity, we'll discuss only the two-dimensional case, but higher dimensions are analogous.

Consider a large two-dimensional array for which a two-dimensional FFT is desired. If the array isn't too large to fit a copy 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

Note here that the use of MPI_Scatter is to distribute the transposed data in a contiguous fashion to the other processes. The application of the one-dimensional FFT must be confined to a single process, but each of the applications of the FFT can be done in parallel because they are completely independent. However, when we must apply the FFT in the second dimension, the array is distributed incorrectly, hence the scatter operation is so that each process gets the corresponding submatrix from each of the other processes so that it can piece together its local array. The transpose is used so that the data is contiguous. Otherwise, many more Send/Receive pairs would be required to redistribute the matrix properly.

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 section is of use, but really is more an opportunity to think about how to handle computations more generally using domain decomposition. When using FFTW on a distributed architecture, it is much easier to use the MPI versions of the software. The general organization of using plans to execute transforms 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: none
14
15 Outputs: Prints out the final values.
16
17 */
18
19 int main(int argc, char* argv[])
20 {
21 // Initialize MPI
22 MPI_Init(&argc, &argv);
23
24 // Initialize FFTW for MPI
25 fftw_mpi_init();
26
27 // Get dimensions of the domain
28 const ptrdiff_t M = atoi(argv[1]);
29 const ptrdiff_t N = atoi(argv[2]);
30 ptrdiff_t localM, localN;
31

```

```

32 // Determine the amount of local memory required
33 ptrdiff_t alloc_local = fftw_mpi_local_size_2d(M, N, MPI_COMM_WORLD,
34 &localM, &local0);
35
36 // Allocate the local memory
37 fftw_complex* datain = fftw_alloc_complex(alloc_local);
38 fftw_complex* dataout = fftw_alloc_complex(alloc_local);
39
40 // Set up the transform plans
41 fftw_plan pf, pb;
42 pf = fftw_mpi_plan_dft_2d(M, N, datain, dataout, MPI_COMM_WORLD,
43 FFTW_FORWARD, FFTW_ESTIMATE);
44 pb = fftw_mpi_plan_dft_2d(M, N, dataout, dataout, MPI_COMM_WORLD,
45 FFTW_BACKWARD, FFTW_ESTIMATE);
46
47 #ifndef M_PI
48 const double M_PI = 4.0*atan(1.0);
49#endif
50
51 // Set up the initial data on the local allocation
52 double dx = 2*M_PI/M;
53 double dy = 2*M_PI/N;
54 for (int i=0; i<localM; ++i) {
55     double x = dx*(local0+i);
56     for (int j=0; j<N; ++j) {
57         double y = dy*j;
58         datain[i*N+j][0] = cos(2*x+y);
59         datain[i*N+j][1] = sin(2*x+y);
60     }
61 }
62
63 // Execute the forward transform
64 fftw_execute(pf);
65
66 // Set up rank 0 to receive the full array upon completion
67 double* fulldatain = NULL;
68 double* fulldataout = NULL;
69 int rank;
70 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
71 if (rank == 0) {
72     fulldatain = (double*)malloc(2*M*N*sizeof(double));
73     fulldataout = (double*)malloc(2*M*N*sizeof(double));
74 }
75
76 // Assemble the results into the full array on rank 0 process
77 MPI_Gather(dataout, 2*localM*N, MPI_DOUBLE, fulldataout, 2*localM*N,
78 MPI_DOUBLE, 0, MPI_COMM_WORLD);
79
80 // Print the results
81 if (rank == 0) {
82     for (int i=0; i<M; ++i) {
83         for (int j=0; j<N; ++j) {
84             double real = fulldataout[i*2*N+2*j];
85             double imag = fulldataout[i*2*N+2*j+1];
86             printf("%4.2f%4.2fi\n", real/M/N,
87                   imag >= 0 ? '+' : '-', fabs(imag/M/N));
88         }
89         printf("\n");
90     }
91 }
92
93 // Execute the backward transform
94 fftw_execute(pb);

```

```

95 MPI_Gather(dataout, 2*localM*N, MPI_DOUBLE, fulldataout, 2*localM*N,
96             MPI_DOUBLE, 0, MPI_COMM_WORLD);
97
98 fftw_destroy_plan(pf);
99 fftw_destroy_plan(pb);
100 fftw_free(datain);
101 fftw_free(dataout);
102 if (rank == 0) {
103     free(fulldatain);
104     free(fulldataout);
105 }
106 MPI_Finalize();
107 }
108 }
```

The first thing to observe about the MPI version of FFTW is that we now must initialize the library on Line 25, which wasn't necessary for the serial version.

For the distributed version, the data is organized so that the first subscript is broken into P subdomains subdivided equally 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 we are using a regular `MPI_Gather` on Lines 77, 96 we have implicitly assumed that all the subdomains are equal, i.e. M is a multiple of P . The size of the local domain is determined on Line 33. The last two arguments are where the size of the first dimension, `localM`, and the first index within the full array, `local0`, are set. The local memory is allocated on Lines 37, 38.

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. On Line 42 the forward plan is set up and on Line 44 the backward plan is set up. Execution of the plans is done on Lines 64, 94.

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, note 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 54*). The global index for the first array is the sum of the local index with `local0` that was obtained on Line 33. Line 55 illustrates how to use `local0` to determine the global coordinate.

Finally, as for the serial version of FFTW, the plans and the data allocated by the library must also be freed as on Lines 99–102.

Chapter 21

Projects for Distributed 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 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 help give hints how to develop your algorithms.

21.1 • Random Processes

Random processes are easily parallelizable because each individual calculation is independent of the others. In this case, care must be taken to ensure that the random number generators are initialized with different values in each process, but otherwise, the various processes can be computed independently. One way to ensure that is to use a single seed for all processes, but then each process would add its process rank to the seed so that each process has a unique starting point. The only communications required are to use `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 as a function of N . Use the plot to estimate the cost to compute a solution with error 10^{-5} with 99.5% confidence. Compute an estimate for both the weak and strong scaling of your algorithm.

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. The values of N , M , and σ should be given on the command line. Use $\alpha = 1$, which should be defined in your program. Compute the time required to complete the calculation as a function of NM and plot the results. Plot an estimate for $p(t)$ for $0 \leq t \leq T = 10$. Compute an estimate for both the weak and strong scaling of your algorithm.

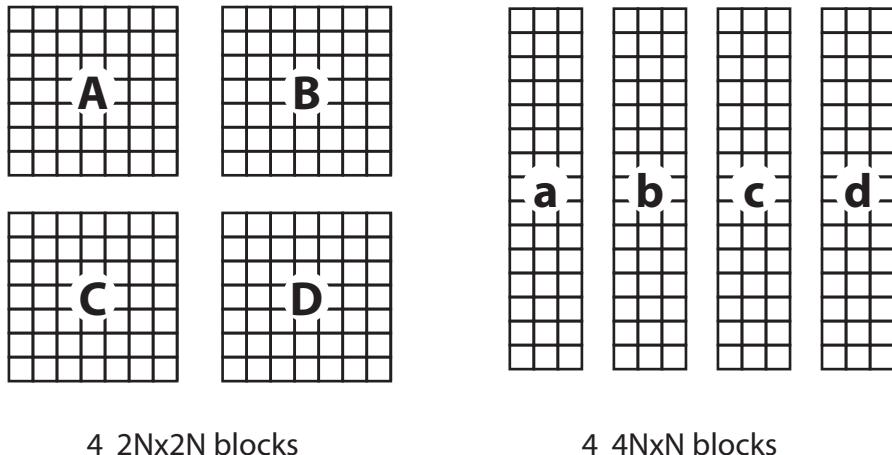


Figure 21.1. Different choices for a 2D spatial domain decomposition. Compare the number of grid points that will require communications and to how many different processes.

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. We have already seen an example of this concept when we discussed the 1D heat equation, where the domain was subdivided into equal segments, and then each segment was updated in parallel. This type of process requires communication of enough boundary data between domains to allow the update.

In one dimension, it is fairly straightforward to subdivide the domain, but let's consider a 2-dimensional domain that has dimensions $4N \times 4N$ grid points, and let's consider two different decompositions as depicted in Figure 21.1. In each case we break the domain into four subdomains, but of different equal 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 . Similarly for the other subdomains. 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.

The four rectangular subdomains each contain the same number of grid points to be updated as for the previous case. The difference lies in the communications. For this system, 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 system. If you experience significant latency due to the higher level of interconnectivity, then the right hand system may be a better choice. It's simpler in construction and has much fewer communications

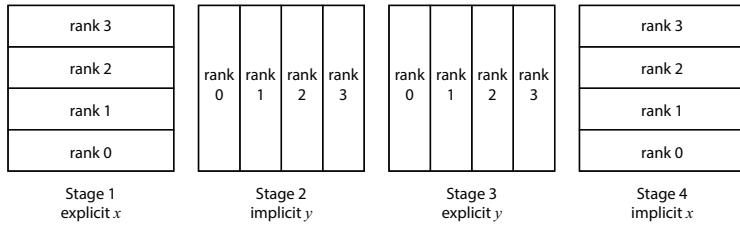


Figure 21.2. Illustration of the four stages of the ADI algorithm for four processes. Each process gets a strip of the domain aligned with the x -direction to update. Between Stages 1 and 2 data must be transferred and transposed between the processes so that the strips can be updated in the y -direction. The transfer transpose operation is repeated between Stages 3 and 4 to return to the x -direction.

requirements, but the amount of the data transferred will be higher. On the other hand, if it's the volume of data being transferred, then reducing the amount of data transferred, but using the left-hand case would be better.

Since we already discussed this strategy at length earlier, we won't repeat that here.

21.2.2 • ADI and MPI

As a rough sketch of the ADI algorithm for MPI, suppose we break the full domain, which is $4N \times 4N$ gridpoints in size and we have 4 processes that will share the work. Stage 1 will have each of the 4 processes doing 1/4 of the explicit update in the x -direction. The stages are illustrated in Figure 21.2. In Stage 1, each process will do an explicit update in the x -direction, which is not dependent on data in the y -direction, so 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 we will discuss 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 must be put back in horizontal strips where we started 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. We'll discuss two different approaches and some of the details that are easily missed when just looking at the rough sketch. 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 the tridiagonal 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. Now, suppose the data is arranged originally in horizontal strips, then recalling that C is stored as row-major, it means that if i is 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 we need the x -direction to be contiguous in memory. That means that when we switch to doing updates in the y -direction, we must also transpose the data so that the columns become rows and the data can again be contiguous. That means the data will be accessed using $u[i][j]$ in that configuration. This issue will also be relevant when we discuss gather/scatter operations because these operations work on linear arrays.

The first approach for doing ADI would be where every process contains a copy of the full grid. This is probably 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. That said, let's go through the algorithm stages and communications. Assume that we begin with each process containing the current version of the data for the full grid laid out so that the x -direction is contiguous.

Stage 1: The first step is to do the explicit update in the x -direction corresponding to the right hand side of Equation 35.21. 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 updated by the other processes using `MPI_Allgather`. The data will need to be in the y -direction next, so transpose the matrix in memory.

Stage 2: The implicit update in the y -direction is next, corresponding to the left hand side of Equation 35.22. We need to apply the LAPACK tridiagonal solver to contiguous data in the y -direction, so that's why the data was transposed in the communications step. Now each process applies the LAPACK routine to the data with the `nrhs` variable set to the number of rows the process must update in this subdomain.

Stage 3: The update on the right hand side of Equation 35.23 is next. The data is still in the right direction to do an explicit update in the y -direction, so execute that.

Communications: Do the same thing 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: Finally, an implicit step in the x -direction corresponding to the left hand side of Equation 35.24 is done. The data is now laid out the same as at the top of the loop, and the LAPACK tridiagonal solver can be applied to the entire data in each processes subdomain.

As you can see, this means that you are 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 memory storage and also in terms of communication costs.

The second approach is a little more difficult to implement, but has a much smaller memory footprint, and a smaller communications cost. In this case, we start by assuming that each process has a copy of only its local subdomain oriented so that the x -direction is contiguous, similar to the start of the previous algorithm. 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 35.21. 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. What we want to do is send a subset of the columns in the array to each of the other processes, not the rows, so the data must first be transposed. Now the transposed data can be 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 as before on the subdomain data.

Stage 3: Do the explicit y -direction update as before on the subdomain data.

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 we can now do the implicit x -direction update 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 illustration of this example and the two approaches serves 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 worth it.

21.2.3 • Brusselator Reaction

Program the assignment in Section 35.3.1 using an $N \times N$ grid. Measure the time required to complete the calculation as a function of N . Compute an estimate for both the weak and strong scaling of your algorithm.

21.2.4 • Linearized Euler Equations

Program the assignment in Section 35.3.2 using an $N \times N$ grid. Measure the time required to complete the calculation as a function of N . Compute an estimate for both the weak and strong scaling of your algorithm.

21.3 • Elliptic Equations and SOR

The red/black ordering is used in a distributed architecture as well. The grid is updated in two separate parallel passes with communications between each pass. Each process

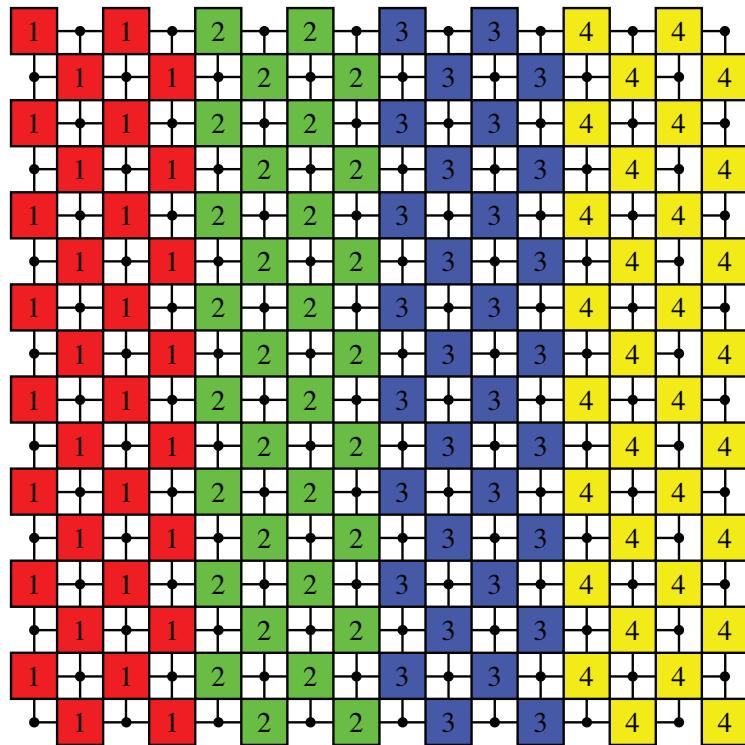


Figure 21.3. Illustration of updating red points in a red/black ordering for SOR. Each color represents the data that each of four cores would be responsible for computing. When all red points are updated, use MPI_Allgatherv to share the results with all neighbors.

will contain a complete grid. On the first pass, the red grid points are divided equally among all the processes and then updated using the SOR formula. After all the red grid points are updated, use MPI_Allgatherv to share the updated values on all the processes. On the second pass, the black grid points are distributed among the processes and updated, using MPI_Allgatherv to assemble the data back into a single updated grid. Use MPI_Allreduce to test for convergence. The subdivision is illustrated in Figure 21.3.

21.3.1 • Diffusion with Sinks and Sources

Program the assignment in Section 36.1.1 using an $(2N - 1) \times N$ grid. Experiment to find the optimal ω that requires the fewest iterations to reach a tolerance of 10^{-5} . Compute the time required to complete a single pass through the grid. Repeat these steps for a grid of dimensions $(4N - 1) \times 2N$. Does the optimal value of ω remain the same? Verify that the time cost for a single pass through the grid is proportional to the number of grid points.

21.3.2 • Stokes Flow 2D

Program the two-dimensional assignment in Section 36.1.2 using a MAC grid that is approximately $N \times N$. Experiment to find the optimal ω that requires the fewest

iterations to reach a tolerance of 10^{-5} . Compute the time required to complete a single pass through the grid. Verify that you get a parabolic profile for the velocity field in the x direction.

21.3.3 • Stokes Flow 3D

Program the three-dimensional assignment in Section 36.1.2 using a MAC grid that is approximately $N \times N \times N$. Experiment to find the optimal ω that requires the fewest iterations to reach a tolerance of 10^{-5} . 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.

21.4 • Pseudo-Spectral Methods

The FFTW package for computing the Fourier transform is already capable of using multiple processes, so it is best to perform the Fourier transforms using the package without using MPI directly. However, the computation of the spatial derivatives requires looping through the Fourier modes, and those can be done by doing domain decomposition in spectral space, i.e. subdividing the differentiation in spectral space and then using `MPI_AllGatherv` to collect the results. Since the derivative calculation has no dependency between Fourier modes, this can be safely parallelized without having to worry about conflicts.

21.4.1 • 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. Compute the time required to execute your code as a function of N . Compute an estimate for both the weak and strong scaling of your algorithm.

21.4.2 • 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. Compute the time required to execute your code as a function of N . Compute an estimate for both the weak and strong scaling of your algorithm.

Part IV

GPU Programming and CUDA

In Parts II, III, we saw parallelism for shared memory and distributed architectures where many processors, all of the same or very similar type are harnessed to solve a problem. Programming with GPUs is quite different, and much more resembles the old vector processors popularized many years ago. The idea was that you have many small 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. In the meantime, GPUs offer substantial speed-ups over serial programs, or even codes written for distributed architectures, especially for the types of problems we often solve 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 we've been using.

In fact, the CUDA compiler is actually a wrapper that couples two distinct compilers into one. When you write code using 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 you're actually invoking two distinct compilers to produce the final product. However, this will all be transparent to you.

Before we get started, it is worth pointing out that other graphics card makers, e.g. Intel, AMD, and others, 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 is covered in Part V. The openCL language is designed to be agnostic toward the graphics card manufacturers, and is being pushed as a standard for graphics card developers to follow. The advantage of openCL is that the same code can be used to run on different makes of GPU cards, or on CPUs. It is meant to harness all the computing resources available on a given machine. NVIDIA cards can also be programmed using openCL, and it has been observed that, with suitable tuning, code written with openCL can be comparable in speed to code written in CUDA, but that it is not as easy to attain peak performance [9].

Almost any computer with an NVIDIA graphics card can be used for programming in CUDA. Before you can compile using CUDA, you will need to install a special driver as well as the compiler and tools. To obtain the driver, go to the website <http://www.nvidia.com/cudand> click on the download drivers link to find the correct driver for your computer, operating system, and graphics card model. In addition to the device driver, you will also need the compiler and tools, which can be obtained by going to <http://developer.nvidia.com/object/gpucomputing.htmlnd> look for the CUDA toolkit. For both packages follow the installation instructions provided by NVIDIA.

In Chapter 22 we introduce the CUDA framework including how to connect with

and select the GPU device to be used, and to detect its relevant properties. Similar to the wrapper compiler `mpicc` introduced in Part III, the CUDA language also has a wrapper compiler called `nvc`, which is introduced in this chapter.

Of course, 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, their structure and speed, and then compares them for purposes of doing a simple finite difference calculation.

In Chapter 25, the blocks and threads are subdivided into parallel tasks using streams. 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 of 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, 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 hardware.

This part of the book concludes with solving the application problems from Part VI using the CUDA language and any specialized information related to GPU implementation.

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, we refer to operations performed in main memory or by the CPU as being 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 doing computing on the device. Since terminal text I/O is not a function for the device, it doesn't make sense to start with "Hello World!" so we instead will start with a simple addition of two numbers as a substitute. 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 we have used in earlier parts of this text. CUDA 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. Furthermore, we must distinguish that a file has CUDA code in it, and is not a plain C code file, so files with CUDA code embedded in them will have the suffix .cu instead of the typical .c for C code. 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 is 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.

Note that we have also allocated memory in a different way on Line 48. It is important to remember that we now will have two sets of memory addresses, conventional or CPU memory addresses, 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 being 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 you don't specify which one to use, then you can run into trouble or get less than optimal results. On my local machine, there are two dedicated compute GPUs, and one GPU that is on the CPU board. The compute GPUs are called the Tesla K20c, while the on-board GPU is the Quadro K600. Table 22.1 shows some relevant data about the characteristics of the two types of GPUs. To ensure that you get the device that you want, you must specify it by setting the proper device. At the beginning of your code, you specify the characteristics your device should have, and then ask the CUDA runtime environment to select the closest match. Note that the call to `cudaSetDevice` must occur before any direct access to the GPU is initiated, e.g. by allocating memory.

Device Name	Quadro K600	Tesla K20c
Compute version	3.0	3.5
Clock rate	876Mhz	706Mhz
Multiprocessors	1	13
Cores per multiprocessor	192	192
Total cores	192	2496
Shared memory per block	49,152	49,152
Registers per block	65,536	65,536
Global memory	1Gb	5Gb

Table 22.1. Comparison of two different GPU units on a representative system. There are two Tesla K20c units and one Quadro K600.

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 }
```

In Example 22.2, the process of choosing the GPU device is demonstrated. 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, which is an indicator that you have 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, we are effectively eliminating the Quadro K600 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, you may want to determine some key characteristics such as the shared memory size, or the maximum number of threads per block, or other quantities for that particular GPU. That information can all be retrieved using the same `cudaDeviceProp` variable by calling

```
cudaGetDeviceProperties(&prop, dev);
```

which was how Table 22.1 was assembled.

Note that for brevity the subsequent examples will skip past the device selection step because the default device is the one desired. For your code, you should routinely include the device selection process.

Chapter 23

Parallel CUDA Using Blocks

Adding two single numbers is not the best use of a GPU device, what we really need to do is 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

So far, the codes we have seen are essentially serial codes, we haven't done any parallelism yet. Here is our first example of a parallel code, where we implement a simple finite difference approximation for a 1D 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 Compute the central difference operator on periodic data
11
12 Inputs:
13     double* u: Function data, assumed periodic
14     int* N: pointer to the length of the data array
15     double* dx: pointer to the space step size
16
17 Outputs:
18     double* du: central difference of the u data
19 */
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 centraldiff
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, copied to the device memory where the finite difference is computed, and the result 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 indicate 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, we pass a reference to the device memory pointer as the first argument (note the ampersand 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:

```
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 your calculations on the device as much as possible.

The interesting bit for our purposes is the kernel function `diff`, which is defined on Lines 21–33, and called on Line 75. Note how we use only device memory inside the kernel function, the device operations do not have access to host memory directly, which is why data must be transferred back and forth outside the kernel. On the calling line, we have requested N blocks with one thread per block. These are the numbers in the angle brackets. We will discuss the relationship between blocks, threads, and warps in the next section. In the function itself, note that there's no loop. This is because each block will compute its one entry in the array. It's a similar idea to how this could be done in MPI, where each process would compute one value in the result vector, and then we would use `MPI_Allgather` to reassemble the output into a single array. In this case, the equivalent of the MPI rank variable is the `blockIdx` structure that is supplied by the CUDA compiler. Blocks can be multi-dimensional, but here we're using only the one dimension, and so `blockIdx.x` gives the equivalent `rank`. There may be more cores than the length of the array, so we must test 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 25.3.

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. However, we are not restricted to that organization. Blocks can be organized into 1, 2, and even 3-dimensional grids. This 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. A two-dimensional grid of blocks is created by replacing Line 75 with

```
dim3 blockdim(N1,N2);
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). Before your eyes get too big, 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! We are 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`).

To illustrate the use of multiple block dimensions, we could apply this method to computing the two-dimensional Laplacian as in Example 23.2.

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 Demonstrate a simple example for implementing a
15 parallel finite difference operator on a 2D array
16
17 Inputs: argc should be 3
18     argv[1]: Size of first dimension
19     argv[2]: Size of second dimension
20
21 Outputs: the initial data and its derivative.
22 */
23
24 int main(int argc, char* argv[]) {
25     int M = atol(argv[1]);
26     int N = atol(argv[2]);
27     // Allocate the memory on the host
28     double *u = (double*)malloc(M*N*sizeof(double));
29     double *du = (double*)malloc(M*N*sizeof(double));
30
31     double* dev_u;
32     double* dev_du;
33     double* dev_dx;
34     double* dev_dy;
35     int* dev_M;
36     int* dev_N;
37
38     // Allocate the memory on the device
39     cudaMalloc((void**)&dev_M, sizeof(int));
40     cudaMalloc((void**)&dev_N, sizeof(int));
41     cudaMalloc((void**)&dev_u, M*N*sizeof(double));
42     cudaMalloc((void**)&dev_du, M*N*sizeof(double));
43     cudaMalloc((void**)&dev_dx, sizeof(double));
44     cudaMalloc((void**)&dev_dy, sizeof(double));
45
46     // Initialize the data on the host.
47     double dx = 2*M_PI/M;
48     double dy = 2*M_PI/N;
49     for (int i=0; i<M; ++i)
50         for (int j=0; j<N; ++j)
51             u[i+j*M] = sin(i*dx)*cos(j*dy);
52
53     // Copy the parameters and data to the device
54     cudaMemcpy(dev_dx, &dx, sizeof(double), cudaMemcpyHostToDevice);
55     cudaMemcpy(dev_dy, &dy, sizeof(double), cudaMemcpyHostToDevice);
56     cudaMemcpy(dev_u, u, M*N*sizeof(double), cudaMemcpyHostToDevice);
57     cudaMemcpy(dev_M, &M, sizeof(int), cudaMemcpyHostToDevice);
58     cudaMemcpy(dev_N, &N, sizeof(int), cudaMemcpyHostToDevice);
59
60     // Construct a 2D grid of blocks
61     dim3 meshDim(M,N);
62

```

```

63 // Invoke the kernel using the 2D array of blocks
64 diff<<<meshDim,1>>>(dev_u, dev_M, dev_N, dev_dx, dev_dy, dev_du);
65
66 // Copy the results back to the host
67 cudaMemcpy(du, dev_du, M*N*sizeof(double), cudaMemcpyDeviceToHost);
68
69 // Release the memory allocated on the device
70 cudaFree(dev_u);
71 cudaFree(dev_du);
72 cudaFree(dev_dx);
73 cudaFree(dev_dy);
74 cudaFree(dev_M);
75 cudaFree(dev_N);
76
77 // Release the memory allocated on the host
78 free(u);
79 free(du);
80 return 0;
81 }
82
83 /*
84 void diff(double* u, int* M, int* N, double* dx, double* dy,
85           double* du)
86
87 Compute the central difference operator on periodic data
88
89 Inputs:
90     double* u: Function data, assumed periodic
91     int* M: pointer to the x length of the data array
92     int* N: pointer to the y length of the data array
93     double* dx: pointer to the x space step size
94     double* dy: pointer to the y space step size
95
96 Outputs:
97     double* du: Laplacian of the u data
98 */
99 __global__ void diff(double* u, int* M, int* N, double* dx, double* dy,
100                      double* du) {
101 if (blockIdx.x < *M && blockIdx.y < *N) {
102     int ij00 = blockIdx.x + blockIdx.y * gridDim.x;
103     int ijp0 = (blockIdx.x + 1)%gridDim.x + blockIdx.y * gridDim.x;
104     int ijm0 = (blockIdx.x + gridDim.x - 1)%gridDim.x
105                                         + blockIdx.y * gridDim.x;
106     int ij0p = blockIdx.x + ((blockIdx.y + 1)%gridDim.y) * gridDim.x;
107     int ij0m = blockIdx.x + ((blockIdx.y + gridDim.y - 1)%gridDim.y)
108                                         * gridDim.x;
109     du[ij00] = (u[ijp0]-2.*u[ij00]+u[ijm0])/ *dx/ *dx
110             + (u[ij0p]-2.*u[ij00]+u[ij0m])/ *dy/ *dy;
111 }
112 }
```

On Line 28, note that the two-dimensional array is being allocated in the manner of a one-dimensional array. Unfortunately, the CUDA compiler does not allow for the two-dimensional contiguous array allocation the way we did it before. To compensate for this, we have to adjust the two-dimensional indices to access a one-dimensional array. This is accomplished on Line 51 where the translation from (i, j) coordinates to linear coordinates is $i+j*M$ where M is the length of each row in the dataset.

On Line 61, 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 64.

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, note the use of both `blockIdx.x` and `blockIdx.y` 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. This is not a typical arrangement because it does not provide any efficiencies to offer an improvement over a CPU implementation. To get efficiency, we have to take advantage of 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 on one block, we would change Line 75 to

```
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, we would also change Line 24 to be

```
int tid = threadIdx.x;
```

Threads can also be arranged in two and three dimensions using the same CUDA type `dim3`, and on the Tesla K20c, the maximum thread dimensions are (1024, 1024, 64). The thread dimension information, the equivalent of the variable `gridDim` for block dimensions, is given by the built-in struct (`blockDim.x`, `blockDim.y`, `blockDim.z`).

Putting it together, one can imagine the picture for our computations to be organized as 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 that has been sacrificed is careful attention to error detection. For example, in MPI the output variables that would record errors in communications were routinely ignored for expedience. This is a double edged sword. 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 must be checked and dealt with appropriately. Because of the low-level nature of the coding required to utilize a GPU, it is more important here to pay closer 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.

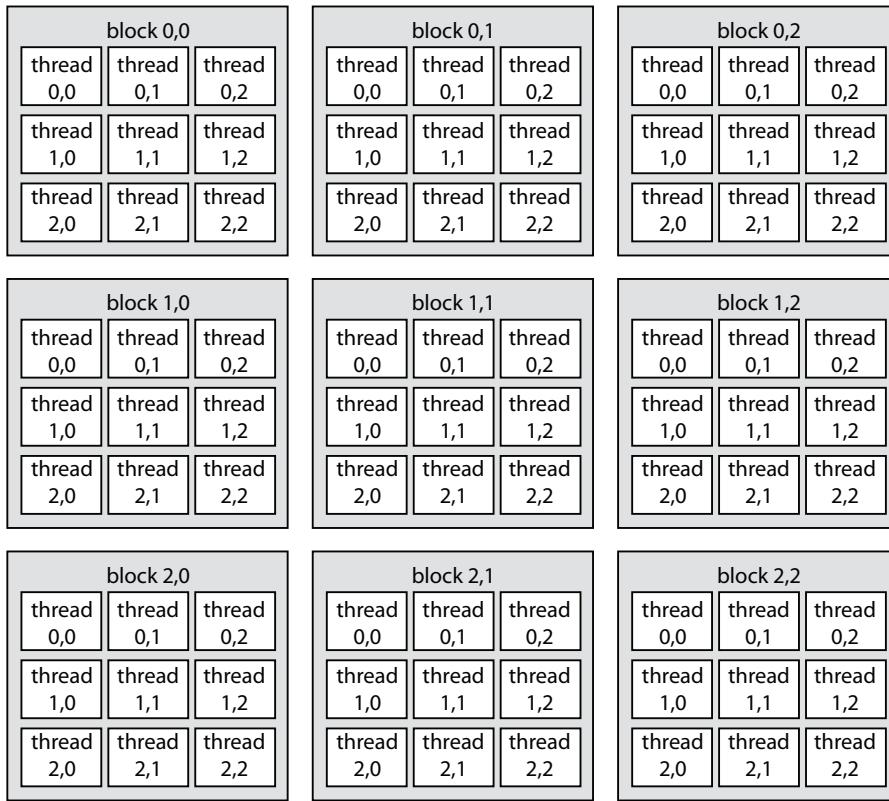


Figure 23.1. Diagram of the organization of threads within blocks with both in arrays of different dimensions. The dimensions of the threads arrays need not be the same as the blocks.

Following the technique presented in [4], rather than writing this all out for every single function call, there is a simple code that can be included in your programs that will help you pinpoint when an error occurs. The best solution is to create a header file for your CUDA-specific information, and include the following code into that header file:

```

1 static void HandleError( cudaError_t err, const char* file, int line) {
2     if (err != cudaSuccess) {
3         printf("%s in %s at line %d\n", cudaGetErrorString(err), file,
4                                         line);
5         exit(1);
6     }
7 }
8 #define HANDLE_ERROR( err ) (HandleError(err, __FILE__, __LINE__))

```

To use this code, whenever a CUDA function is called, simply wrap it inside a function call to HANDLE_ERROR like this:

```
HANDLE_ERROR( cudaMalloc((void**)&dev_u, N*sizeof(double)) );
```

Since we have not discussed the C preprocessor in any great detail, this usage will require some discussion. Preprocessor directives are identified by the leading # character. An example of such a directive we have used many times already is the directive

#include. So, the “function” HANDLE_ERROR is in fact not a function at all, but simply a preprocessor directive that can also take an argument. When the preprocessor sees the text “HANDLE_ERROR(X)”, it will replace it with “(HandleError(X, __FILE__, __LINE__))”. The strings “__FILE__” and “__LINE__” are also identified by the pre-processor, and it will substitute the name of the current file for __FILE__ and the line number within that file will be put in place of __LINE__. Therefore, when the preprocessor is done, it will replace

```
HANDLE_ERROR( cudaMalloc((void**)&dev_u, N*sizeof(double)) );
```

with

```
HandleError( cudaMalloc((void**)&dev_u, N*sizeof(double)),
             "filename.cu", 17 );
```

Now, HandleError *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 precisely what you want an error handler to do.

Chapter 24

GPU Memory

Up to this point we have been exclusively using global memory on the GPU to do the computations, 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 make it the most efficient. There is also constant memory, which is fast, but not as plentiful, and texture memory which 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 sufficient to produce any 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. The current organization is using global memory throughout. Because a significant latency arises due to all the threads or blocks making requests from global memory simultaneously, they are effectively forced to queue up sequentially resulting in no better performance than would be achieved by using the CPU in the first place.

To gain in speed, we need to take advantage of shared 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. So why don't we just use one block and only shared memory? Shared memory is much smaller than available global memory space. By comparison, the global memory space for the Tesla K20c is about 5Gb, while the shared memory for a single block is a little over 49Kb.

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 like this:

```
__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 like this:

```
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. To see how this is done, 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      __shared__ double localu[threadIdx.x+2];
3      __shared__ double localdu[threadIdx.x];
4
5      // Transfer global memory to shared memory
6      int g_i = (threadIdx.x + blockIdx.x * blockDim.x) % *N;
7      int l_i = threadIdx.x + 1;
8      int g_im = (g_i + *N - 1) % *N;
9      int g_ip = (g_i + 1) % *N;
10     localu[l_i] = u[g_i];
11     if (threadIdx.x == 0)
12         localu[0] = u[g_im];
13     if (threadIdx.x == threadsPerBlock-1)
14         localu[l_i+1] = u[g_ip];
15
16     // Compute the finite difference for this thread and store
17     // in shared memory
18     localdu[threadIdx.x] = (localu[threadIdx.x+2]-localu[threadIdx.x])
19                               /*dx) / 2. ;
20
21     // Transfer the results from shared memory to global memory
22     du[g_i] = localdu[threadIdx.x];
23 }
```

In this example, suppose we have 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 6. Stage 1 of the kernel, where the global memory is copied to the shared memory is done on Lines 10–14. Here, each thread copies its corresponding global memory location to the local shared memory address on Line 10. However, because we are doing a finite difference approximation that has a three-point stencil, we also need the two data points on either side. Therefore, the first and last thread in the block additionally copy the neighboring value from the global data so that the finite difference calculation within any thread is confined to the shared memory and doesn't have to retrieve global memory for any

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

blockIdx.x = 0
23 0 1 2 3 4

blockIdx.x = 1
3 4 5 6 7 8

blockIdx.x = 2
7 8 9 10 11 12

blockIdx.x = 3
11 12 13 14 15 16

blockIdx.x = 4
15 16 17 18 19 20

blockIdx.x = 5
19 20 21 22 23 0

local du

blockIdx.x = 0
0 1 2 3

blockIdx.x = 1
4 5 6 7

blockIdx.x = 2
8 9 10 11

blockIdx.x = 3
12 13 14 15

blockIdx.x = 4
16 17 18 19

blockIdx.x = 5
20 21 22 23

Figure 24.1. Illustration of the memory layout for shared memory in Example 24.1

reason. 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 18.

Finally, stage 3 is done on Line 22, 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 in this case, all threads of the given block will stop and wait at the synchronization point until all threads reach that point. When using `__syncthreads()`, care must be taken to make sure that it's not, for example, inside an if statement where only some of the threads reach the synchronization point. If that happens, then the program will lock up waiting for the threads that can never reach the synchronization point. The corrected kernel is shown in Example 24.2.

Example 24.2.

```

1 __global__ void diff(double* u, int* N, double* dx, double* du) {
2     __shared__ double localu[threadIdx.x + 2];
3     __shared__ double localdu[threadIdx.x];

```

```

4
5 // Transfer global memory to shared memory
6 int g_i = (threadIdx.x + blockIdx.x * blockDim.x) % *N;
7 int l_i = threadIdx.x + 1;
8 int g_im = (g_i + *N - 1) % *N;
9 int g_ip = (g_i + 1) % *N;
10 localu[l_i] = u[g_i];
11 if (threadIdx.x == 0)
12     localu[0] = u[g_im];
13 if (threadIdx.x == threadsPerBlock - 1)
14     localu[l_i + 1] = u[g_ip];
15
16 __syncthreads();
17
18 // Compute the finite difference for this thread and store
19 // in shared memory
20 localdu[threadIdx.x] = (localu[threadIdx.x + 2] - localu[threadIdx.x])
21                                         /*dx*/ / 2;
22
23 // Transfer the results from shared memory to global memory
24 du[g_i] = localdu[threadIdx.x];
25

```

A final note about shared memory is that it is retrieved synchronously in groups of requests called a half-warp, the warp size for the Tesla K20c is 32, so that means that memory access is done in groups of 16 at a time. For that reason, there is a preference for performance reasons to use a number of threads that is a multiple of a warp.

24.2 • Constant Memory

There is another form of fast-access, low-latency memory available on the GPU, and that is constant memory. The GPUs on kepler each have 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 in our current model, the number of nodes in the array is not determined until runtime. Example 24.3 shows how the constant memory is set up and used within the kernel function.

Example 24.3.

```

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 // 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 // declare the kernel function
16 __global__ void diff(double* u, double* du);
17 /*
18 */
19 /* Demonstrate a simple example for implementing a
20 parallel finite difference operator
21
22 Inputs: argc should be 2
23 argv[1]: Length of the vector of data
24
25 Outputs: the initial data and its derivative.
26 */
27 int main(int argc, char* argv[]) {
28
29 // read in the number of grid points
30 int N = atoi(argv[1]);
31
32 // determine how many blocks are needed for the whole grid
33 const int blocksPerGrid = N/threadsPerBlock
34 // + (N%threadsPerBlock > 0 ? 1 : 0);
35
36 // allocate host memory
37 double* u = (double*)malloc(N*sizeof(double));
38 double* du = (double*)malloc(N*sizeof(double));
39
40 double* dev_u;
41 double* dev_du;
42
43 // allocate device memory
44 cudaMalloc((void**)&dev_u, N*sizeof(double));
45 cudaMalloc((void**)&dev_du, N*sizeof(double));
46
47 // initialize the data on the host
48 double dx = 2*M_PI/N;
49 for (int i=0; i<N; ++i)
50     u[i] = sin(i*dx);
51
52 // cudaMemcpyToSymbol copies data into the constant memory space
53 // on the GPU
54 cudaMemcpyToSymbol(dev_N, &N, sizeof(int));
55 cudaMemcpyToSymbol(dev_dx, &dx, sizeof(double));
56 cudaMemcpy(dev_u, u, N*sizeof(double), cudaMemcpyHostToDevice);
57
58 // The kernel call no longer needs to send dev_N or dev_dx
59 // to the kernel
60 diff<<<blocksPerGrid, threadsPerBlock>>>(dev_u, dev_du);
61
62 // Copy the results back to the host
63 cudaMemcpy(du, dev_du, N*sizeof(double), cudaMemcpyDeviceToHost);
64
65 // Notice that we don't need to free the dev_N and dev_dx pointers
66 // anymore.
67 cudaFree(dev_u);
68 cudaFree(dev_du);
69 free(u);
70 free(du);
71 return 0;
72 }
73
74 // The constant variables are not passed through
75 // the kernel argument list.
76 __global__ void diff(double* u, double* du) {
```

```

78
79    __shared__ double localu[threadIdx.x+2];
80    __shared__ double localdu[threadIdx.x];
81
82    // copy data from global to local memory
83    // dev_N is now in constant memory
84    int g_i = (threadIdx.x + blockIdx.x * blockDim.x) % dev_N;
85    int l_i = threadIdx.x + 1;
86    int g_im = (g_i + dev_N - 1) % dev_N;
87    int g_ip = (g_i + 1) % dev_N;
88    localu[l_i] = u[g_i];
89    if (threadIdx.x == 0)
90        localu[0] = u[g_im];
91    if (threadIdx.x == threadsPerBlock - 1)
92        localu[l_i + 1] = u[g_ip];
93
94    __syncthreads();
95
96    // compute the finite difference
97    localdu[threadIdx.x] = (localu[threadIdx.x + 2] - localu[threadIdx.x])
98                                / dev_dx / 2.;
99
100   // save the results back into global memory
101   du[g_i] = localdu[threadIdx.x];
102 }
```

The first difference is that the variable `dev_N` and `dev_dx` are no longer declared inside the main program, but instead are declared on Lines 13–14. On those lines, the variables are identified as `__device__ static`, but they are not given constant values yet. When the value of the constant values are determined at runtime, the data for the number of nodes and the space step size are loaded onto the device using a different function, `cudaMemcpyToSymbol` on Lines 55–56. Instead of loading the data into global memory, it gets loaded into constant memory. Because it's in constant memory, it does not get loaded through the argument list, hence when the kernel is called, on Line 61, the `dev_N` and `dev_dx` variables are no longer in the argument list. Finally, the variables can be read within the kernel like normal such as on Line 84.

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 25.3.

24.3 ▪ Texture Memory

There is yet another way to store data on the GPU that can provide a boost to performance, and that is texture memory. Texture memory is different from the other types we have 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 as we shall see. Finally, texture memory is read-only, so we can only use it for the input data, not for the output.

To begin, suppose our data is single precision floating point rather than double precision so we can avoid some complications. There are multiple kinds of texture

data, but for simplicity we'll just do a single layer of floating point values. Texture data must be declared in a 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 our task is to do a one-dimensional finite difference operation, we'll use a simple one-dimensional arrangement for reading.

Inside the main program, the memory for the input data is allocated using `cudaMalloc` as before, 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 <time.h>
4 #include <cuda.h>
5 #include <cuda_runtime.h>
6
7 #ifndef M_PI
8 #define M_PI 3.1415926535897932384626433832795
9 #endif
10
11 const int threadsPerBlock = 256;
12
13 // declare constant memory on the device
14 __device__ static int dev_N;
15 __device__ static float dev_dx;
16 // Declare the texture memory in global scope like the constant memory
17 texture<float, cudaTextureType1D, cudaMemcpyKind> tex_u;
18
19 // Note that the kernel function now only has the output for an argument
```

```

20 --global__ void diff(float* du);
21
22 /*
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 printf("Time: %gms\n", elapsedTime);
80 return 0;
81 }
82

```

```

83 __global__ void diff(float* du) {
84
85     int g_i = (threadIdx.x + blockIdx.x * blockDim.x) % dev_N;
86     int g_im = (g_i + dev_N - 1) % dev_N;
87     int g_ip = (g_i + 1) % dev_N;
88
89     // Texture memory is accessed through specialized fetch functions
90     // such as tex1Dfetch()
91     du[g_i] = (tex1Dfetch(tex_u, g_ip) - tex1Dfetch(tex_u, g_im)) / dev_dx / 2.;
92 }
```

24.3.1 • Texture memory and double precision

Notice that this example is different than the previous ones in that it only uses floats. That's because texture memory is not designed to handle double precision data. However, double precision data can be stored on there, it just requires a little extra effort to make it work. Double precision values require 8 bytes, or twice what an int 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 17:

```
texture<int2, cudaTextureType1D, cudaMemcpyKind> 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 <time.h>
4 #include <cuda.h>
5 #include <cuda_runtime.h>
6
7 #ifndef M_PI
8 #define M_PI 3.1415926535897932384626433832795
9 #endif
10
11 const int threadsPerBlock = 256;
12
13 // declare constant memory on the device
14 __device__ static int dev_N;
15 __device__ static double dev_dx;
16 // Declare the texture memory using int2 type
17 texture<int2, cudaTextureType1D, cudaReadModeElementType> tex_u;
18
19 // Declare the kernel function
20 __global__ void diff(double* du);
21
22 /*
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 }
```


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 we hope to gain 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. In order to take advantage of this capability, we have to utilize streams. We have been implicitly using streams all along, 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. In this chapter we'll see 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 following code:

```
cudaStream_t stream;  
cudaStreamCreate( &stream );
```

Once the stream is created, tasks can begin to be assigned to it. For example, to assign a kernel call to a stream, the stream is inserted as the fourth argument inside the angle brackets like this:

```
diff<<< blocks, threads, shared_memory, stream>>>(dev_u, dev_du);
```

In order to take advantage of streams, there are some changes that will have to be made to the examples we've been using. As a starting point, we'll use the shared memory example, but now make it work with multiple threads. As you may recall, one of the steps in implementing a kernel is that data must be copied from the host to the device, and back again (if we want to know the results). The process can be depicted graphically as follows:

Memcpy host to device
Execute kernel
Memcpy device to host

The whole 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.

At the end of the program, in the clean up phase, the stream must also be disposed of using the function

```
cudaStreamDestroy(stream);
```

25.2 • Asynchronous memory copies

In our 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 programmers 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 to be handled asynchronously, we have to make sure it's where it says it will be at the time that the transfer takes place, and we can't guarantee when that will be when commands are issued asynchronously.

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. Any memory that is to be copied to/from the device asynchronously must not be allocated using the usual `malloc`, but instead using a different allocation function that behaves in a very similar way:

```
double* u;
cudaHostAlloc((void**)&u, N*sizeof(double), cudaHostAllocDefault);
```

This creates page-locked memory that will not move while it is active. Since the memory is locked, that means that other programs or tasks on the same computer now 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, not to worry, any subsequent task in this particular stream will wait until it's completed.

25.3 • Synchronizing streams

Suppose the last task in the stream queue is an asynchronous memory copy from device to host to store the results of the computation. Before printing or saving the output data, we have to ensure that the memory copy is completed. To do this, we need a comparable function to `_syncThreads()` but on the host side. That command is

```
cudaStreamSynchronize(stream);
```

The process will stop at this command until all the tasks assigned to `stream` are completed.

25.4 • 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 the shared memory example introduced earlier.

So that you can see an example of both the timing tasks and the error handling that was discussed, that code is also added to create Example 25.1. Lines that may 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 "handleerror.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 Demonstrate a simple example for implementing a
18 parallel finite difference operator
19
20 Inputs: argc should be 2
21     argv[1]: Length of the vector of data
22
23 Outputs: the initial data and its derivative.
```

```

24 */
25
26 int main(int argc, char* argv[]) {
27
28 // Get the length of the data from the argument list
29 int N = atol(argv[1]);
30 double dx = 2*M_PI/N;
31 const int blocksPerGrid = N/threadsPerBlock
32                                     + (N%threadsPerBlock > 0 ? 1 : 0);
33 double *u, *du;
34
35 double* dev_u;
36 double* dev_du;
37
38 // Set the constant values for N and dx on the device
39 HANDLE_ERROR( cudaMemcpyToSymbol(dev_dx, &dx, sizeof(double)) );
40 HANDLE_ERROR( cudaMemcpyToSymbol(dev_N, &N, sizeof(int)) );
41
42 // Replace regular malloc with cudaHostAlloc so that memory is pinned
43 HANDLE_ERROR( cudaHostAlloc( (void**)&u, (N+2)*sizeof(double),
44                                         cudaHostAllocDefault) );
45 HANDLE_ERROR( cudaHostAlloc( (void**)&du, N*sizeof(double),
46                                         cudaHostAllocDefault) );
47
48 // Allocate memory on the device
49 HANDLE_ERROR( cudaMalloc((void**)&dev_u, (N+2)*sizeof(double)) );
50 HANDLE_ERROR( cudaMalloc((void**)&dev_du, N*sizeof(double)) );
51
52 // Create a new stream
53 cudaStream_t stream;
54 HANDLE_ERROR( cudaStreamCreate( &stream ) );
55
56 // Initialize the data
57 for (int i=0; i<N; ++i)
58     u[i+1] = sin(i*dx);
59 u[0] = u[N];
60 u[N+1] = u[1];
61
62 // The memory copy is performed asynchronously,
63 // control returns immediately
64 HANDLE_ERROR( cudaMemcpyAsync(dev_u, u, (N+2)*sizeof(double),
65                                         cudaMemcpyHostToDevice, stream) );
66
67 // Create the CUDA events that will be used for timing
68 // the kernel function
69 cudaEvent_t start, stop;
70 HANDLE_ERROR( cudaEventCreate(&start) );
71 HANDLE_ERROR( cudaEventCreate(&stop) );
72
73 // Click, the timer has started running
74 HANDLE_ERROR( cudaEventRecord(start, 0) );
75
76 // The kernel call is also issued asynchronously because we've
77 // specified a stream which is the 4th argument
78 diff<<<blocksPerGrid, threadsPerBlock, 0, stream>>>(dev_u, dev_du);
79
80 // Click, the timer event stops when all threads synchronize.
81 HANDLE_ERROR( cudaEventRecord(stop, 0) );
82 HANDLE_ERROR( cudaEventSynchronize(stop) );
83
84 // The elapsed time is computed by taking the difference between
85 // start and stop
86 float elapsedTime;

```

```

87 HANDLE_ERROR( cudaEventElapsedTime(&elapsedTime , start , stop) );
88
89 // The solution data can also be copied asynchronously from the device
90 // back to the host
91 HANDLE_ERROR( cudaMemcpyAsync(du , dev_du , N*sizeof(double) ,
92                               cudaMemcpyDeviceToHost , stream ) );
93
94 // Because the data is copied asynchronously, we can't use it
95 // until we're sure the stream has finished
96 HANDLE_ERROR( cudaStreamSynchronize(stream) );
97
98 // The CUDA events have to be disposed of properly
99 HANDLE_ERROR( cudaEventDestroy (start) );
100 HANDLE_ERROR( cudaEventDestroy (stop) );
101
102 HANDLE_ERROR( cudaFree(dev_u) );
103 HANDLE_ERROR( cudaFree(dev_du) );
104
105 // The pinned memory must also be freed using a CUDA function
106 HANDLE_ERROR( cudaFreeHost(u) );
107 HANDLE_ERROR( cudaFreeHost(du) );
108
109 // The stream must be disposed of as well.
110 HANDLE_ERROR( cudaStreamDestroy(stream) );
111
112 printf("Time: %gms\n" , elapsedTime );
113 return 0;
114 }
115
116 __global__ void diff(double* u, double* du) {
117     __shared__ double localu[threadsPerBlock+2];
118     int g_i = threadIdx.x + blockIdx.x * blockDim.x + 1;
119
120     // The length of the data vector may be shorter than
121     // the number of threads
122     if (g_i <= dev_N) {
123         int l_i = threadIdx.x + 1;
124         localu[l_i] = u[g_i];
125
126         // The first thread must pick up the boundary data at the left
127         if (threadIdx.x == 0)
128             localu[0] = u[g_i-1];
129
130         // The last thread in the range needs to also double up for the
131         // boundary data on the right
132         if (g_i == dev_N
133             || (g_i < dev_N && threadIdx.x == threadsPerBlock-1))
134             localu[l_i+1] = u[g_i+1];
135
136         // The threads must complete the copy from global to shared memory
137         // before the calculation
138         __syncthreads();
139
140         du[g_i-1] = (localu[threadIdx.x+2]-localu[threadIdx.x])/dev_dx/2.;
141     } else {
142
143         // All threads must issue this command whether involved
144         // in the computation or not
145         __syncthreads();
146     }
147 }
```

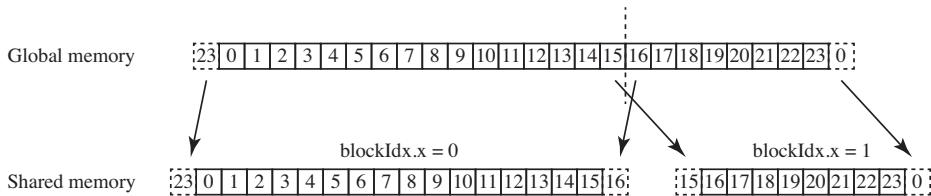


Figure 25.1. Illustration of the global and shared memory arrangement for Example 25.1 assuming 16 threads per block. The first block gets a complete set of shared memory data, but the second block is incomplete because the vector length is too short.

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 index too large are excluded on Line 122. Additionally, the first and last threads with a role in computing the finite difference are also used to copy the boundary data on Lines 127–134. 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 138, the rest of the threads also have to synchronize or the program will block here, hence the second synchronization point on Line 145.

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 two 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 you don't have threads doubling up, but then you have fewer threads performing the finite difference operator, hence requiring more blocks/threads to complete the task. These kinds of tradeoffs are important to consider when designing a fast code because they can have a large effect on the final performance.

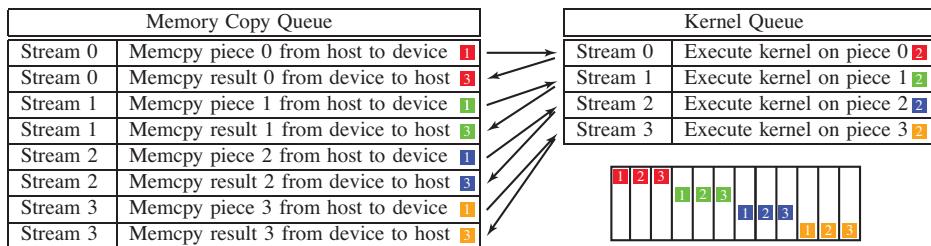
25.5 • Multiple streams

The power of streams don't really kick in until you start getting them to work in tandem to make sure the GPU is taking full advantage of its various computing components. For example, there is no reason to leave the memory transfer unit idle while a kernel is running if there is memory to be transferred. This makes the steps of the dance of the various streams and operations in the streams critical for optimal performance. Unfortunately, the sequential nature of how we think of programming is not quite in line with how the GPU works.

Suppose we plan to take our finite difference calculation and break it into 4 pieces, one piece for each of four streams. One way we could queue up the commands for the streams would be to post them with a loop through each stream like this:

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 here we have unrolled the loop to see the sequence of instructions issued. The GPU will now queue these instructions in the order received into two queues, memory transfers and kernel operations. The two queues now look like this after the sequence of commands:

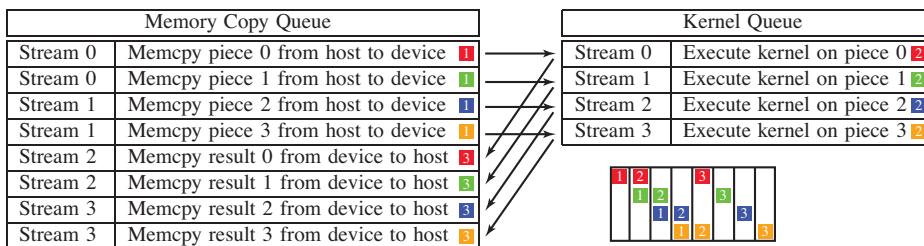


where the arrows show the dependencies according to the order of operations within the different streams. The Memory Copy Queue and the Kernel Queue can operate simultaneously in this model, so that means that items labelled 2 can be executed simultaneously with items labelled 1 or 3 in the figure above. 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 rather than simultaneously as illustrated in the sequence in the bottom right.

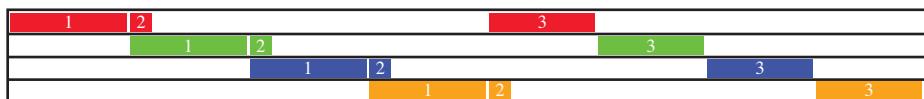
Compare this to the following sequence of issuing commands:

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

By ordering the operations in this way, we can get the Memory Copy Queue and the Kernel Queue to work simultaneously to complete operations as illustrated below:



As you can see, we've cut the steps down from 12 to 8 resulting in some savings in effort. This kind of stacking is very easy to implement, but for the best results, consideration as to 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 above should look more like the one below:



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 lions 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 for as long as possible without interacting between the host and the GPU.

The implementation of four streams in this manner is not much different than how the single stream was written, but the example code is presented below for your reference.

Example 25.2.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4 #include "handleerror.h"

```

```

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 * Demonstrate a simple example for implementing a
20 parallel finite difference operator
21 */
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     HANDLE_ERROR( cudaMemcpyToSymbol(dev_dx, &dx, sizeof(double)) );
51     HANDLE_ERROR( 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     HANDLE_ERROR( cudaHostAlloc( (void**)&u, (N+2)*sizeof(double),
56                                         cudaHostAllocDefault ) );
57     HANDLE_ERROR( cudaHostAlloc( (void**)&du, N*sizeof(double),
58                                         cudaHostAllocDefault ) );
59
60     // Create an array of streams , and allocate device memory
61     // for each of them.
62     cudaStream_t stream[numStreams];
63     for (int i=0; i<numStreams; ++i) {
64         HANDLE_ERROR( cudaMalloc((void**)&dev_u[i],
65                               (N/numStreams+2)*sizeof(double)) );
66         HANDLE_ERROR( cudaMalloc((void**)&dev_du[i],
67                               N/numStreams*sizeof(double)) );
68     }
69
70     // Set up the streams
71     for (int i=0; i<numStreams; ++i) {
72         stream[i] = stream[0];
73     }
74
75     // Copy the data from host to device
76     HANDLE_ERROR( cudaMemcpy( dev_u[0], u, N*sizeof(double), cudaMemcpyHostToDevice ) );
77
78     // Compute the derivatives
79     for (int i=0; i<numStreams; ++i) {
80         diff<<>>(
81             dev_u[i], dev_du[i]
82         );
83     }
84
85     // Copy the results back to host
86     HANDLE_ERROR( cudaMemcpy( u, dev_du[0], N*sizeof(double), cudaMemcpyDeviceToHost ) );
87
88     // Clean up
89     for (int i=0; i<numStreams; ++i) {
90         cudaStreamDestroy(stream[i]);
91     }
92
93     return 0;
94 }
```

```

68     HANDLE_ERROR( cudaStreamCreate( &(stream[ i ]) ) );
69 }
70
71 for (int i=0; i<N; ++i)
72     u[ i+1 ] = sin( i*dx );
73     u[ 0 ] = u[ N ];
74     u[ N+1 ] = u[ 1 ];
75
76 // First we queue the memcpy from host to device for all streams
77 for (int i=0; i<numStreams; ++i)
78     HANDLE_ERROR( cudaMemcpyAsync( dev_u[ i ], u+(i*N/numStreams) ,
79                                 (N/numStreams+2)*sizeof(double) , cudaMemcpyHostToDevice ,
80                                         stream[ i ] ) );
81
82 cudaEvent_t start , stop;
83 HANDLE_ERROR( cudaEventCreate(&start) );
84 HANDLE_ERROR( cudaEventCreate(&stop) );
85 HANDLE_ERROR( cudaEventRecord(start , 0) );
86
87 // Next queue the kernel functions for all streams
88 for (int i=0; i<numStreams; ++i)
89     diff<<<blocksPerGrid , threadsPerBlock , 0, stream[ i ]>>>(dev_u[ i ],
90                                         dev_du[ i ] );
91
92 HANDLE_ERROR( cudaEventRecord(stop , 0) );
93 HANDLE_ERROR( cudaEventSynchronize(stop) );
94 float elapsedTime;
95 HANDLE_ERROR( cudaEventElapsedTime(&elapsedTime , start , stop) );
96
97 // Finally queue the memcpy to get the results off the device and
98 // back in the host
99 for (int i=0; i<numStreams; ++i)
100    HANDLE_ERROR( cudaMemcpyAsync( du+i*N/numStreams , dev_du[ i ],
101                                N/numStreams*sizeof(double) , cudaMemcpyDeviceToHost , stream[ i ] ) );
102
103 // Each stream must be synchronized before using the results.
104 for (int i=0; i<numStreams; ++i)
105    HANDLE_ERROR( cudaStreamSynchronize(stream[ i ]) );
106
107 // Time to clean up
108 HANDLE_ERROR( cudaEventDestroy (start) );
109 HANDLE_ERROR( cudaEventDestroy (stop) );
110
111 for (int i=0; i<numStreams; ++i) {
112     HANDLE_ERROR( cudaFree(dev_u[ i ]) );
113     HANDLE_ERROR( cudaFree(dev_du[ i ]) );
114 }
115
116 HANDLE_ERROR( cudaFreeHost(u) );
117 HANDLE_ERROR( cudaFreeHost(du) );
118
119 for (int i=0; i<numStreams; ++i)
120     HANDLE_ERROR( cudaStreamDestroy(stream[ i ]) );
121
122 printf("Time: %gms\n" , elapsedTime );
123 return 0;
124 }
125
126 __global__ void diff(double* u, double* du) {
127     __shared__ double localu[threadsPerBlock+2];
128     int g_i = threadIdx.x + blockIdx.x * blockDim.x + 1;
129     if (g_i <= dev_N) {
130         int l_i = threadIdx.x + 1;

```

```

131 localu[l_i] = u[g_i];
132 if (threadIdx.x == 0)
133     localu[0] = u[g_i-1];
134 if (g_i == dev_N || (g_i < dev_N
135                                         && threadIdx.x == threadsPerBlock-1))
136     localu[l_i+1] = u[g_i+1];
137
138 __syncthreads();
139
140 du[g_i-1] = (localu[threadIdx.x+2]-localu[threadIdx.x])/dev_dx/2.;
141 } else {
142     __syncthreads();
143 }
144 }
```

To illustrate the improvement, in Figure 25.2 the timings for the cases of 1, 2, and 4 streams with shared memory and asynchronous memory copies are added to the previous timing plots. Note that using 2 or 4 streams with shared memory gives results comparable to the texture memory example discussed earlier (the plots overlap for high values of N).

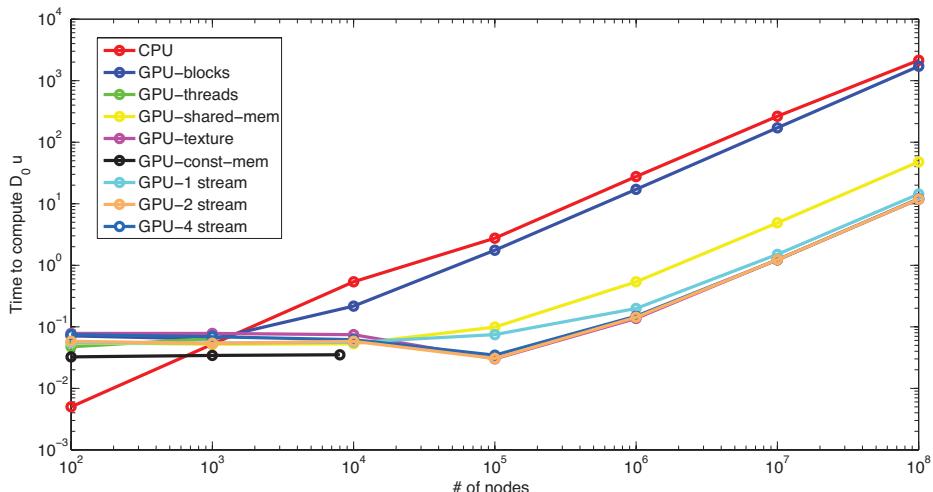


Figure 25.2. Timing plots for the kernel functions only for the version with 1, 2, and 4 streams as well as the previous timing data. Note that the plots for 2 and 4 streams coincide with the texture memory plot for larger values of N

One final comment about streams is that you may think of the CPU as being a completely independent stream as well. This means that the GPU can be doing a sequence of tasks at the same time as the CPU. For example, while a kernel is running, you might slip a function into the CPU stream to handle a different task. Taking advantage of the CPU where possible is another high-speed stream that can be utilized to create a fast solver.

25.6 • General strategies

As you can see, the largest expense is in the transfer of data between the host and the device, so for a fast code, you want to minimize those transfers. Consequently, you

want to keep the data on the GPU as much as possible and just issue sequences of kernel functions. For example, suppose you have 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, but maybe you want data every 100th step. Then you might set up a loop like this for one stream:

```

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 that, you're keeping the data on the GPU for faster access.

There are additional tricks one can play (and this is true for MPI applications as well). 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, there is a threshold where this makes things worse rather than better, but that is part of the tuning process.

25.7 ▪ Measuring Performance

The whole purpose of exploring the use of GPUs is to gain speed and so we need some means of evaluating the speed of kernels. This can be done using CUDA events.

Similar to previous timing methods, we mark the time at the beginning of a kernel execution, and again at the end and take the difference. To record the time at a particular location, create 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 with id 0.

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 we need

another synchronization point, but this time on the host side. This is accomplished with a `cudaEventSynchronize` function that takes as an argument the event record that requires synchronization. Therefore, to record the ending time of a kernel call, use

```
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
```

Upon completion of the synchronization, the time will again be recorded in a variable named `stop` that has type `cudaEvent_t`.

Finally, use the function `cudaEventElapsedTime` to get the elapsed time between two events. The function takes a pointer to float as the first argument where the time will be recorded, and the two CUDA events between which the time is measured. The time between events is stored in the first argument and is measured in milliseconds.

The example below shows how to record the time required to execute the kernel function `diff` on the GPU:

Example 25.3.

```
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);
```

Now that we know how to record the time elapsed for a kernel function, let's try it out. In this section there have been several methods described for implementing the finite difference approximation through a GPU kernel function, each in increasing levels of sophistication. We can now compare the different methods to see which is faster. Figure 25.3 shows the following different methods discussed thus far applied to computing the central difference operator on a vector of length N :

1. Using no GPU or CUDA, but simply execute a loop in C.
2. Using N blocks through the kernel call `diff<<<N,1>>>`
3. Using N threads and only one block. (Note, the maximum number of threads for a single block is 1,024)
4. Using $N/256$ blocks and 256 threads per block and using shared memory in each block.
5. Using $N/256$ blocks and 256 threads per block and using texture memory.

Figure 25.3 shows that simply using a kernel with many blocks does not achieve much improvement over a regular 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, simply using one block with many threads is also not

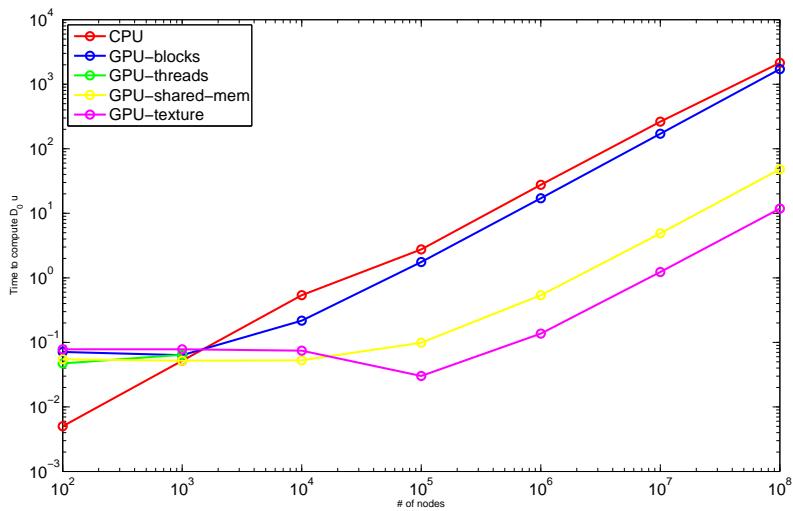


Figure 25.3. Scaling properties of 1D finite central difference using various versions of parallelism using CUDA compared to a serial CPU implementation. The horizontal axis shows the number of nodes in the grid, and the vertical axis shows the time to complete the finite difference kernel function. 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 one block and $N < 1024$ threads, (GPU-shared-mem) Results from using 256 threads per block and shared memory as in Example 24.2, (GPU-texture) Results from using texture memory instead of shared memory as in Example 24.5.

feasible. On the other hand, taking advantage of cached shared memory or texture memory can result in almost 200 times faster code. Texture memory is the fastest, but it is read-only, whereas shared memory is both read/writeable. Thus, it is well worth the effort to learn these more detailed implementations on the GPU, and failure to do so will result in almost no improvement in performance.

Exercises

- 25.1. Build and run Example 22.1 to make sure you are able to access CUDA and a suitable GPU.
- 25.2. Modify Example 23.2 to read the device properties using `cudaGetDeviceProperties`. Use the properties to subdivide the domain in such a way that a maximal number of threads are used per block to cover the given dimensions M, N . Use shared memory for the local array to improve efficiency.
- 25.3. 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 vs. computing the finite difference.

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. In this chapter, we'll go 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 here:

<http://icl.cs.utk.edu/magma/software/>

/noindent and there is a very helpful document [18] with examples here:

<https://developer.nvidia.com/sites/default/files/akamai/cuda/files/Misc/mygpu.pdf>

that details many examples of usage on the NVIDIA website. 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 simple.

Note that the CUDA language calls are all embedded into the MAGMA library, so if you wish to use MAGMA solely as a linear algebra engine and not use any CUDA code, you will save your program with the .c suffix and compile it 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 gives a simple code that 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 was done for LAPACK.

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, we want to use pinned memory for the input data. MAGMA takes care of the arguments necessary to create the pinned memory so that our 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 we won't repeat that information here. 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:

```
gcc -O3 -DADD_ -DHAVE_CUBLAS -c program.c
```

The flag `-O3` is an optimization flag, and asks for level 3 optimizations. We have not discussed compiler flagged optimizations yet, but they are worth learning and also being wary of. The number following the `-O` is the level of optimization, where the higher the number the more aggressive (max 3). Setting optimization level to zero means no optimizations will be done, and is what is the default. More aggressive optimization does lead to longer compile times, but can result in faster code. However, care should be taken when using high level optimization because it is possible for the compiler to optimize so much it breaks your code (and it will be impossible to find why). Always develop your code using `-O0` and then try out higher levels of optimization to see if you get improvement in performance and/or unexpected results.

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

```
gcc -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 multi-core multi-GPU system, so for CPU implementations it still requires linking the LAPACK library and the corresponding gfortran library.

26.1.3 • CPU interface vs. GPU interface

In Example 26.1, the results of the computation were left on the host. That may or may not be what is desired. In fact, you may want the result to stay on the device rather than shifting it back and forth. For that reason, MAGMA also has a GPU interface, where the inputs and the results are kept on the GPU. However, for us to check it, we still must get the input data to the device, and read the results back off of it. 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, we need to put the matrix **A** and the right hand side **b** onto 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. We also have to set up device memory to hold the input data and the results, and that 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 `magma_dsetmatrix` function 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. This is a command to the compiler preprocessor to make a decision to compile one set of code or another. 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, 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 allows to show how both the single solver driver is called compared to separating the factorization from the solver.

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 a few details for it to be done properly. For example, When you have 2,000 threads all trying to generate random numbers, how do you seed them uniquely? Failing to do so would cause your 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 you can call the random number generator from within a kernel function that you’ve written to produce a single random value. The library also has other distributions to choose from beside just the uniform distribution as is produced by functions like `drand48()`.

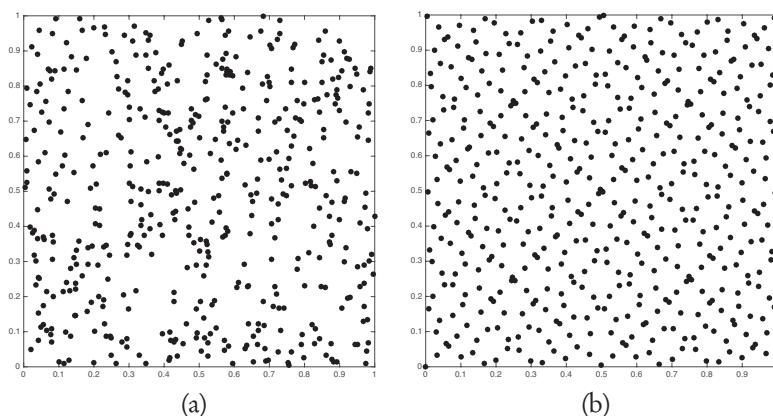


Figure 26.1. (a) Random points in a box generated with a pseudo-random number generator. (b) Random points in a box generated with a quasi-random number generator.

The package provides two different styles of random numbers: pseudo-random numbers and quasi-random numbers. Pseudo-random numbers are what you are normally accustomed to; the numbers are generated by a deterministic linear congruential generator algorithm, that is initiated 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, you have a uniform random number generated over the interval $[0, 1]$, the collection of random samples will be roughly uniformly spaced throughout the interval so that if you do a histogram of the results, it will be reasonably flat with not much variation. To see how the two cases look different, in Figure 26.1, two sets of random points in the box $[0, 1] \times [0, 1]$ are shown. These illustrate how the quasi-random is more uniformly spaced compared to the pseudo-random values. Within these two general classifications of generators, the library offers multiple random sequence algorithms. We will not go into depth on this topic here, if you want more details check the documentation in

<http://docs.nvidia.com/cuda/curand>

26.2.1 • Host interface

There are three basic steps to generating a list of random values using the host interface: (1) create the generator, which allocates dynamically allocated space, (2) seed the generator, and (3) use the generator to produce a list of random samples. The steps are illustrated in Example 26.3.

Example 26.3.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4 #include <cuda.h>
5 #include <curand.h>
6
7 /*
8  * Example generating random numbers
9  */
10
11 // Inputs: argc should be 2
12 // argv[1]: Number of samples to generate
13
14 // Outputs: the initial data and its derivative.
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));
```

```

31  double *dev_x;
32  cudaMalloc((void**)&dev_x, num_samples*sizeof(double));
33
34  // Set up random number generator, use default pseudo
35  // random number generator
36  curandGenerator_t gen;
37  curandCreateGenerator(&gen, CURAND_RNG_PSEUDO_DEFAULT);
38
39  // Seed the random number generator using the time
40  curandSetPseudoRandomGeneratorSeed(gen, time(NULL));
41
42  // Generate num_samples on the device using a uniform distribution
43  curandGenerateUniformDouble(gen, dev_x, num_samples);
44
45  // Copy results back to the host
46  cudaMemcpy(x, dev_x, num_samples*sizeof(double),
47             cudaMemcpyDeviceToHost);
48
49  // Store result
50  FILE* outfile = fopen("example19.3.out", "w");
51  fwrite(x, sizeof(double), num_samples, outfile);
52  fclose(outfile);
53
54  // Free the memory allocated including the generator
55  curandDestroyGenerator(gen);
56  cudaFree(dev_x);
57  free(x);
58  return 0;
59 }
```

The first step to generate random numbers is to set up the random number generator. On Line 37, a random number generator is created from the default pseudo-random number algorithm. There are a number of other algorithm choices, see the online documentation for more information. It is worth noting that the generator allocates some memory dynamically, so it will need to be freed later.

The second step is to seed the random number generator, which is one on Line 40. The first argument is the created generator, and the second argument is an unsigned long long int value. In this example, we are using the output of the `time()` function to create a new starting value. For debugging purposes, it is recommended to set this value to a fixed value so that the same sequence of random values is generated every time, so that errors are reproducible.

Finally, 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 43. The first argument is the generator that was set up in the previous two steps. The second argument is the address *in device memory* where the values will be stored, and finally, the number of samples requested is given in the third argument.

As you can see, if just the list of random values were desired, then the array must be copied to the host as is done on Line 46. However, if further operations are to be performed on the array, then it is best to leave the data on the device and then write kernels to manipulate the data as desired before returning the results to the host. For example, Line 43 might be followed by a transform that will change the distribution of the random values, or sums of values computed for purposes of computing the mean and variance are done on the device, and only the results transferred back to the host. As noted several times, the number of data transfers between device and host are to be minimized as much as possible.

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 `float` and `double` precisions. The single precision uniform distribution has the same name as in Example 26.3, but without “Double” in the name.

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 your own kernel functions. For example, you may want to have each thread solve a stochastic differential equation requiring many random values, but that each thread should have its own initial seed. Rather than repeatedly calling the host random generator as was done in Example 26.3, you would want to request a random generator from within your kernel.

To use the device interface, there is again a three 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, and finally, your kernel can then call the generator for the desired distribution. The sequence is illustrated in Example 26.4, where the kernel function generates a single log-normal distributed random variable. Ordinarily, you would do more than generate a single random value, for this interface, but it’s just for illustration purposes.

Example 26.4.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4 #include <cuda.h>
5 #include <curand.h>
6 #include <curand_kernel.h>
7
8 __global__ void setup_kernel(int N, long int seed, curandState_t *state)
9 {
10    int id = threadIdx.x + blockIdx.x * blockDim.x;
11    if (id < N)
12        curand_init(seed, id, 0, &state[id]);
13 }
14
15 __global__ void do_samples(int N, double mean, double stddev,
16                           double* samples, curandState_t* state)
17 {
18    int id = threadIdx.x + blockIdx.x * blockDim.x;
19    if (id < N) {
20        curandState_t localState = state[id];
21        samples[id] = curand_log_normal_double(&localState, mean, stddev);
22    }
23 }
24
25 /*
26 Example generating random numbers with exponential
27 distribution
28
29 Inputs: argc should be 3
30        argv[1]: Number of samples to generate
31        argv[2]: Mean of the distribution
32        argv[3]: Standard deviation of the distribution
33

```

```

34  Outputs: the initial data and its derivative.
35 */
36
37 int main(int argc, char* argv[]) {
38
39 // Choose the GPU card
40 cudaDeviceProp prop;
41 int dev;
42 memset(&prop, 0, sizeof(cudaDeviceProp));
43 prop.multiProcessorCount = 13;
44 cudaChooseDevice(&dev, &prop);
45 cudaSetDevice(dev);
46
47 // Get maximum thread count for the device
48 cudaGetDeviceProperties(&prop, dev);
49 int num_threads = prop.maxThreadsPerBlock;
50
51 // Get number of samples
52 int N = atoi(argv[1]);
53 int num_blocks = N/num_threads + (N%num_threads ? 1 : 0);
54
55 // Get parameters for the log normal distribution
56 double mean = atof(argv[2]);
57 double stddev = atof(argv[3]);
58
59 // Allocate space for the kernel functions
60 double *x = (double*)malloc(N*sizeof(double));
61 double *dev_x;
62 cudaMalloc((void**)&dev_x, N*sizeof(double));
63
64 // Set up a random number generator state for each thread
65 curandState_t* dev_state;
66 cudaMalloc((void**)&dev_state, N*sizeof(curandState));
67
68 // Seed the generators
69 setup_kernel<<<num_blocks, num_threads>>>(N, time(NULL), dev_state);
70
71 // Convert to exponential distribution
72 do_samples<<<num_blocks, num_threads>>>(N, mean, stddev, dev_x,
73                                         dev_state);
74
75 // Copy device data to host
76 cudaMemcpy(x, dev_x, N*sizeof(double), cudaMemcpyDeviceToHost);
77
78 // Store result
79 FILE* outfile = fopen(argv[4], "w");
80 fwrite(x, sizeof(double), N, outfile);
81 fclose(outfile);
82
83 // Clean up the allocated memory
84 cudaFree(dev_state);
85 cudaFree(dev_x);
86 free(x);
87 return 0;
88 }

```

First notice we are using the `cudaDeviceProp` information for the chosen device to determine the maximum number of threads available per block. We retrieve the information on Line 48 using the function `cudaGetDeviceProperties()` introduced in Section 22.2. If N values are to be generated in this example, with one value gener-

ated per thread, then we can calculate the number of blocks and threads necessary to compute N values efficiently as is done on Line 53.

The setup kernel begins on Line 8, where an array of random generator states, one for each entry in the array to be produced. 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 for each block/thread pair 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. In order to make this possible, a different generator state of type `curandState_t` is created for each index. For this reason, we need an array of generator states for each index. Those states are allocated on Line 66. The offset is there in case you wish to not start at the seeded value, but to first run off `offset` random values before using the generator to sample random values.

Once all the data is allocated, the setup kernel function is called on Line 69. Here, the initial seed is being set by calling the `time(NULL)` function to sample the clock to get a long integer. Obviously, if you want to debug, this value should be set to a fixed value.

Next, now that the generator states are initialized, we can call our kernel function that calls a generator from within. This is done on Line 72. The kernel function begins on Line 15. 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 resulting values will be such that $\log(\langle x \rangle) = \text{mean}$). The parameter values are passed through the argument list to the kernel function.

Finally, on Line 84, the allocations are released. Note that the array of all the random generators must be freed as well.

This example could be speeded up by using streams if N is large enough. In that case, you would generate as many random values as possible in one stream, then while asynchronously downloading those data, the next stream of random values can be generated, and so forth.

26.3 • CUDA FFT

The standard CUDA package also includes an FFT library that is based on the FFTW library discussed in other parts of this book. Some of the type names are different, and a few of the function names differ a bit, but otherwise it is the same thing. One key difference is that it assumes that the input and output will be on the device, and it doesn't provide a CPU interface like the MAGMA package. That is not much of a limitation, however. The differences are so small, that we go straight to the example code:

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/*
11int main(int argc, char* argv[])
12
13The main program takes no arguments and creates a data set to run
14through the FFT server and outputs the results from doing a forward
15and inverse transform.
16
17Inputs: none
18
19Output: Sample output of the FFT server
20*/
21
22int 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 its multiplied by
63    // N/2, correct it with a rescaling kernel
64    rescale <<<1>>>(dev_a, 2./N);
65
66    // Wait for the stream to be completed before retrieving the data

```

```

67     cudaThreadSynchronize();
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     cudaThreadSynchronize();
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`.

The strategy for building the FFT is similar to how FFTW worked before. The first step is to create a plan that will optimize the computation of the FFT for the given compute resources, in this case the GPU. The plan is of type `cufftHandle_t` and a 1D 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	cufftComplex
Z	cufftDoubleComplex

In this case, we want to use N collocation points with a cufftDoubleComplex input and 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 2 data sets of length N , then the data would be given as contiguous $2N$ data points and the last argument would be a 2. 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, 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.

As for the previous discussion about 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 spectral 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 the function `cudaThreadSynchron` should be called at the end to make sure the FFT is completed before using the data.

Finally, note that any plans created must also be disposed of using the `cufftDestroy` function as shown on Line 98.

26.3.1 ▪ 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 very 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

<t>	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 $AY = aX$

Table 26.1. Options for the naming convention for (a) data type, (b) matrix format, and (c) operation to be performed.

compressed sparse row matrix form. Example 26.6 illustrates the use of this function.

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 is comprised of a data vector and an index vector. The data vector contains only the non-zero entries in the vector, and the index vector gives the indices of those non-zero 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 non-zero 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 non-zero 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 non-zero entries. The column index vector gives the column index within the row where the non-zero entries are listed. For example:

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	3	5	8	10	13		---	---	---	---	----	----																													
Column Index vector:																---	---	---	---	---	---	---	---	---	---	---	---	---		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 for the transpose. 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	9	4	2	6	2	1	5	3	7	3	8	4		---	---	---	---	---	---	---	---	---	---	---	---	---															
Row Index vector:																---	---	---	---	---	---	---	---	---	---	---	---	---		0	3	1	4	2	0	3	1	4	2	0	2	4		---	---	---	---	---	---	---	---	---	---	---	---	---															
Column Index vector:													---	---	---	---	---	---	----	----	----	----		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);
88    cudaMemcpy(dev_Arowptr, Arowptr, (N+1)*sizeof(int),
89               cudaMemcpyHostToDevice);
90    cudaMemcpy(dev_Acolind, Acolind, 3*N*sizeof(int),
91               cudaMemcpyHostToDevice);
92    cudaMemcpy(dev_xval, xval, N*sizeof(double), cudaMemcpyHostToDevice);
93    cudaMemcpy(dev_xind, xind, N*sizeof(int), cudaMemcpyHostToDevice);
94
95    // Initialize the cuSPARSE library and get the environment context
96    cusparseHandle_t handle = NULL;
97    cusparseCreate(&handle);
98
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   cusparseDscsr(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   cusparseDcsrmm(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 }
```

You should look over 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. Our matrix has no special structure that we'll exploit 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, and that is done by creating the environment context, which is of type `cusparseHandle_t` and is created by the function `cusparseCreate`. This is done on Line 98.

The matrix vector multiply requires the vector to be in dense format, so we use the function `cusparseDsctr` to convert the sparse vector into a dense vector on Line 107. The matrix vector multiplication is now done with the function `cusparseDcsrsv` as shown on Line 113, and followed by `cudaDeviceSynchronize` to ensure that all operations are complete.

The output of this program, so that you can see how the data was constructed, is shown below:

```

[ -2.0  1.0  0.0  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 ■ Tri-diagonal 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 matrices, and also for tri-diagonal systems. Since tri-diagonal systems arise frequently, it would be informative to see how to do it.

Similar to the LAPACK tri-diagonal 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 the matrix:

$$\begin{array}{cccccc} d_m[0] & d_u[0] & 0 & \cdots & \cdots & 0 \\ d_\ell[0] & d_m[1] & d_u[1] & & & \vdots \\ 0 & d_\ell[1] & d_m[2] & d_u[2] & & \vdots \\ \vdots & & \ddots & \ddots & \ddots & 0 \\ \vdots & & & d_\ell[N-3] & d_m[N-2] & d_u[N-2] \\ 0 & \cdots & \cdots & 0 & d_\ell[N-2] & d_m[N-1] \end{array}$$

Matlab arrangement

$$d_\ell[0](=0) \quad \begin{array}{cccccc} d_m[0] & d_u[0] & 0 & \cdots & \cdots & 0 \\ d_\ell[1] & d_m[1] & d_u[1] & & & \vdots \\ 0 & d_\ell[2] & d_m[2] & d_u[2] & & \vdots \\ \vdots & & \ddots & \ddots & \ddots & 0 \\ \vdots & & & d_\ell[N-2] & d_m[N-2] & d_u[N-2] \\ 0 & \cdots & \cdots & 0 & d_\ell[N-1] & d_m[N-1] \end{array} \quad d_u[N-1](=0)$$

cuSPARSE arrangement

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.

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 factorizaton. The cuSPARSE solver requires it to redo the factorization every application, resulting in slower code. In my experience, the MAGMA dense matrix solver is about twice as fast as the cuSPARSE tridiagonal solver for the same problem. Example 26.7 shows how the tridiagonal solver can be 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 Example program for using the tri-diagonal solver in the
28 cuSPARSE library. This program solves the NxN problem Ax = b
29 where
30   [2 -1           ]      [ 1 ]
31   [-1 2 -1       ]      [ 2 ]
32 A = [ -1 2 -1       ]      [ 3 ]
33   [ ...          ]      [ ... ]
34   [           -1 2 -1]      [ N-1 ]
35   [                 -1 2 ]      [ N ]
36
37 Input: argc should be 2
38 argv[1]: Dimensions of A, b
39
40 Output
41   solution vector is printed
42 */
43 int main(int argc, char* argv[])
44 {
45 // Pick the GPU device
46 cudaDeviceProp prop;
47 int dev;
48 memset(&prop, 0, sizeof(cudaDeviceProp));
49 prop.multiProcessorCount = 13;
50 cudaChooseDevice(&dev, &prop);
51 cudaSetDevice(dev);
52 cudaGetDeviceProperties(&prop, dev);
53
54 int N = atoi(argv[1]);
55
56 // Initialize cuSPARSE library
57 cusparseStatus_t status;
58 cusparseHandle_t handle=0;
59 status= cusparseCreate(&handle);
60
61 // Allocate Memory on host(CPU) and device (GPU)
62 double *x = (double*) malloc(N*sizeof(double));
63 double *b = (double*) malloc(N*sizeof(double));
64 double *dev_dl, *dev_d, *dev_du, *dev_b;
65 cudaMalloc((void**)&dev_dl, N*sizeof(double));
66 cudaMalloc((void**)&dev_d, N*sizeof(double));
67 cudaMalloc((void**)&dev_du, N*sizeof(double));
68 cudaMalloc((void**)&dev_b, N*sizeof(double));
69
70 // initialize the three diagonals
71 if (N > 256) {
72     int num_blocks = N/256 + (N%256 > 0 ? 1 : 0);
73     init<<<num_blocks, 256>>>(N, dev_dl, dev_d, dev_du);
74 } else
75     init<<<1,N>>>(N, dev_dl, dev_d, dev_du);

```

```

76
77 // initialize the B vector and put it on the device
78 int i;
79 for (i=0; i<N; ++i)
80     b[i] = i+1;
81 cudaMemcpy(dev_b, b, N*sizeof(double), cudaMemcpyHostToDevice);
82
83 // Solve the tridiagonal system for a single right hand side
84 // Solution is placed in dev_b, diagonals are left unchanged
85 status = cusparseDgtsv(handle, N, 1, dev_dl, dev_d, dev_du, dev_b, N);
86 if (status != CUSPARSE_STATUS_SUCCESS) {
87     printf("CUSPARSE solver failed");
88 }
89
90 // Retrieve the solution from the device
91 cudaMemcpy(x, dev_b, N*sizeof(double), cudaMemcpyDeviceToHost);
92
93 // Print the solution
94 for (i=0; i<N; ++i)
95     printf("%f\n", x[i]);
96
97 /*Free the Memory*/
98 cusparseDestroy(handle);
99 cudaFree(dev_dl);
100 cudaFree(dev_d);
101 cudaFree(dev_du);
102 cudaFree(dev_b);
103 free(x);
104 free(b);
105 return 0;
106 }
```

In order to use the cuSPARSE tri-diagonal solver, the library requires some setup, which is done on Line 59 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 deallocated later using `cusparseDestroy(handle)` as is done on Line 98.

The tri-diagonal 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 81. The solver can now be called as is done on Line 85. 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 in N );
```

Here, the initialized sparse matrix system initialized earlier is stored in `handle`. The dimensions of the matrix to be inverted is $N \times N$. 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 are 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 86 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 91.

26.4.4 • Linking the library

Finally, the cuSPARSE library is linked into your code by adding `-lcusparse` to the command when building the executable.

Exercises

- 26.1. Modify Example 26.1 to solve a tridiagonal system of equations using the GPU interface.
- 26.2. 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 27.1.1.
- 26.3. Add a kernel to compute the first derivative of the data in an array of length N following the discussion in Section 37.2.

Chapter 27

Projects for CUDA 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 CUDA to implement GPU versions of these projects. Where appropriate, there are some additional comments specific to CUDA that will help 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 amongst 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 is not of much interest here. Let's look at the other two cases where we assume that the device interface for CURAND has generated 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 the same as Case (2). For simplicity of discussion, we will 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 it here.

CUDA does not provide a global reduction kernel function, 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. Sup-

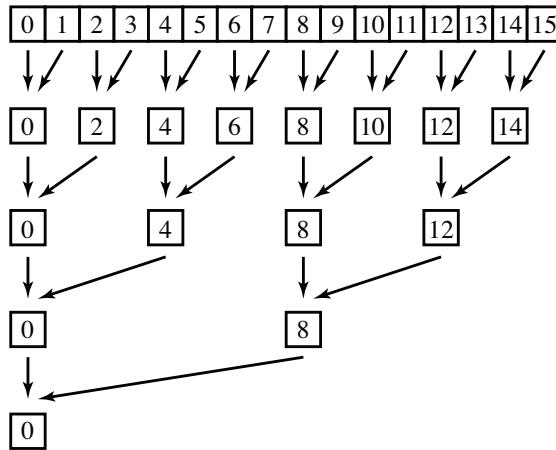


Figure 27.1. Illustration of how a single kernel is used to do pairwise sums and storing the results in place in order to sum the total of the array. The whole array is summed in $\log_2 N$ operations. The numbers indicate which array entries contain the current subtotals for the boxes to their right.

pose the kernel sums two numbers in an array for specified indices and stores the result in the lower of the two indices. The kernel could then be applied to the entire array resulting in $N/2$ entries that are the sum of the two neighboring values. The kernel is reapplied to the $N/2$ entries to get $N/4$ entries, and so forth until the total is reduced. Figure 33.1 illustrates the process. This makes the reduction a $\log_2 N$ 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 as a function of N . Use the plot to estimate the cost to compute a solution with error 10^{-5} with 99.5% confidence and then validate the claim. Compare the pseudo-random 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 have each thread solve one SDE from beginning to end and then utilize as many threads as possible in each block. Since you are only tallying the final value, everything should be done in local memory except for storing the final state 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 σ should be given on the command line. Use $\alpha = 1$, which should be defined in your program. Compute the time required to complete the calculation as a function of NM and plot the results. Plot an estimate for $p(t)$ for $0 \leq t \leq T = 10$.

27.2 ■ Finite Difference Methods

You will need multiple kernels to employ the ADI method, and the kernels will need to maintain all their computations on the device. You will need two kernels 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 should use the MAGMA library with the tridiagonal solver. One important point to remember is that the matrix that has to be inverted does not change with time, so that means you should use the `magma_dgetrf_gpu` once to factor the matrix, and thereafter use `magma_dgetrs_gpu` to get the solutions, all 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 only done once, but if the domain is not square or the boundary conditions vary between the directions, then you will need to do the factorization 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.

27.2.1 • Brusselator Reaction

Program the assignment in Section 35.3.1 using an $N \times N$ grid. Measure the time required to complete the calculation as a function of N .

27.2.2 • Linearized Euler Equations

Program the assignment in Section 35.3.2 using an $N \times N$ grid. Measure the time required to complete the calculation as a function of N .

27.3 • Elliptic Equations and SOR

For maximal efficiency, you should write two kernels, one for the red points and one for the black using the red/black ordering scheme. You should be storing 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 an $(2N - 1) \times N$ grid. Experiment to find the optimal ω that requires the fewest iterations to reach a tolerance of 10^{-9} . Compute the time required to complete a single pass through the grid. Repeat these steps for a grid of dimensions $(4N - 1) \times 2N$. Does the optimal value of ω 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 $N \times N$. Experiment to find the optimal ω 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 parabolic profile for the velocity field in the x direction.

27.3.3 • Stokes Flow 3D

Program the three-dimensional assignment in Section 36.1.2 using a MAC grid that is approximately $N \times N \times N$. Experiment to find the optimal ω 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.

27.4 • Pseudo-Spectral Methods

The Fourier transforms will be handled using the CUDA FFT library keeping data on the device. You will need to write a separate kernel to perform the pseudo-spectral derivative operations.

27.4.1 • Complex Ginsburg-Landau Equation

Because this is a complex valued equation, you will want to use the `cufftExecZ2Z` plan execution.

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. Compute the time required to execute your code as a function of N .

27.4.2 • Allen-Cahn Equation

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 you would use for the forward transform because you have real data, and the latter would be used for the inverse transform because it is known there should be no complex part in the solution. 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 transform will inevitably lead to small, but non-zero 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. Compute the time required to execute the solution to the indicated terminal time. Plot the results and verify that phase separation is occurring. Compute the time required to execute your code as a function of N .

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 are built with NVIDIA graphics cards, so what is available for the rest? An open-source cross-platform alternative to CUDA is called OpenCL [19]. Most of the high end graphic card makers have an OpenCL backend so that a generic OpenCL interface can be used to access lots of subsystems on a computer including not just the 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. We will focus just on using the GPUs here because that is the most useful for the type of programs required for scientific computing.

It should be noted that the OpenCL system is not as polished as CUDA, but is currently standard on all Apple OS X systems [20], so it is worth learning if you use those systems. For a reference to the OpenCL functions and more information, check out the website:

<https://www.khronos.org>

There are also helpful reference books [5].

In Chapter 28, the basic steps in creating 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 workgroups and work-items, which are parallels to the CUDA concepts of blocks and threads, is also presented.

For OpenCL, the memory space on the device is broken down into global, local, and constant. Because OpenCL is meant to be more general, some forms of specialized memory as described in Part IV is not so easily accessed. Chapter 30 describes how to manage the different types of memory within the OpenCL framework.

Streams in CUDA have a corresponding version in OpenCL called command queues. Chapter 31 explores the role of command queues and how to insert events into command queues to be able to measure the cost of computing a kernel function.

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 the access to the GPU through the use of kernel functions is all handled by an external 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 workgroups, work-items, and command queues respectively.

Without a doubt, the biggest difference between CUDA and OpenCL is how and when the kernel functions are compiled and invoked. Where CUDA has a nice streamlined specialized notation that hides the details from the user, OpenCL has it all on display. In this chapter, we do the equivalent of “Hello World!” by using the GPU to add two numbers together. We also go through the basic steps of using OpenCL including choosing a device, setting up the environment, and introduce the compiling process.

28.1 • First OpenCL Program

The basic structure of OpenCL and CUDA programs are similar in the sense that there is 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 are *compiled at run-time*. This can be expensive if the same kernel function is compiled repeatedly, so care must be taken to ensure that compilations are done only once.

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;
```

```

9 } ";
10 /*
11 * Example generating random numbers
12 Inputs: argc should be 3
13 argv[1]: first addend
14 argv[2]: second addend
15
16 Outputs: the sum
17 */
18
19 int main(int argc, const char * argv[]) {
20
21 // Read the input values
22 int a = atoi(argv[1]);
23 int b = atoi(argv[2]);
24
25 // c is the place for the result
26 int c;
27
28 // Request a GPU device
29 cl_uint numPlatforms;
30 int err = clGetPlatformIDs(0, NULL, &numPlatforms);
31 cl_platform_id platform;
32 err = clGetPlatformIDs(1, &platform, NULL);
33 cl_device_id device_id;
34 err = clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device_id,
35
36
37
38 // Setup the context for the GPU
39 cl_context context = clCreateContext(0, 1, &device_id, NULL, NULL,
40
41
42 // Before accessing the GPU, we must create a dispatch queue,
43 // which is the sequence of commands that will be sent to the GPU.
44 cl_command_queue queue = clCreateCommandQueue(context, device_id, 0,
45
46
47 // In OpenCL, the kernel is compiled at runtime
48 const char* srccode = addkernel;
49 cl_program program = clCreateProgramWithSource(context, 1, &srccode,
50
51
52 // Compile the kernel code
53 err = clBuildProgram(program, 0, NULL, NULL, NULL);
54
55 // Create the kernel function from the compiled OpenCL code
56 cl_kernel kernel = clCreateKernel(program, "add", &err);
57
58 // Set up the memory on the GPU where the answer will be stored
59 cl_mem dev_c = clCreateBuffer(context, CL_MEM_WRITE_ONLY, sizeof(int),
60
61
62 // Define all the arguments for the kernel function
63 err = clSetKernelArg(kernel, 0, sizeof(int), &a);
64 err |= clSetKernelArg(kernel, 1, sizeof(int), &b);
65 err |= clSetKernelArg(kernel, 2, sizeof(cl_mem), &dev_c);
66
67 const unsigned long one = 1;
68
69 // Execute the kernel
70 err = clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &one, &one, 0,
71

```

```

72         clFinish(queue);                                     NULL, NULL);
73
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 }
```

As you can see, the underlying construction of the program is similar to CUDA, 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 transported back to the host. And once again, we must remember to clean up when we're done. However, there are some significant differences in how each of these steps are implemented between OpenCL and CUDA. The most significant difference between them is in setting up the kernel in the first place. For CUDA, all the detail work of setting up the device for use is handled by the CUDA compiler and is hidden from sight. OpenCL doesn't have this luxury in part because there are more devices in the world than just CPUs and GPUs. This added flexibility comes at the price of having to explicitly prepare the system to use the GPU in the first place.

In OpenCL, before a kernel can be run, the device must be prepared, which entails the sequence of (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. In CUDA, except for the memory allocation, all this was encapsulated into the line `add<<<1,1>>>(a, b, dev_c)` and was handled by the CUDA compiler behind the scenes. For OpenCL, this is all done on Lines 32–66.

The process of compiling the kernel, since for OpenCL is done at run-time, means that the kernel must start as a string of text. Thus, on Line 5, the kernel program is stored in a string variable that will be given as an argument to the compiler. When covered C in Part I, we learned that strings cannot be broken across lines of code without special treatment. Here, the '\` character is a continuation marker to indicate that the string continues on the next line. Thus, the entire kernel function is stored in a single string variable.

The kernel is actually run by submitting the kernel to the device's command queue, which is done on Line 71. The program will continue after queuing the kernel and for a long kernel, may not be done before the host proceeds to attempt to read the output. Thus, the `clFinish(queue)` function acts like a blocking receive, where the host will stop at this line until the kernel has finished performing it's kernel.

After the kernel is executed the sequence of steps is very much the same between CUDA and OpenCL, the data is mapped back to the host on Line 76, and finally all the memory allocated is released.

28.2 ▪ Selecting the Correct GPU

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. If you don't specify the correct type, then you can run into trouble or make the code run slower rather than faster. The devices available on a MacBook Pro with Retina display are shown in Table 28.1. To ensure that you get the device that you want, you must specify it by setting the proper device for each command queue. Multiple command queues with multiple devices are possible. In example 28.2, the system is polled for the available devices and their characteristics. A GPU device is chosen for the command queue, which in this example turns out to be the Intel card, which is built into the i7 CPU chip.

Device Name	Intel i7	Intel HD Graphics 4000	NVIDIA GeForce GT 650M
Device Type	CPU	GPU	GPU
Clock rate	2600 Mhz	1250 Mhz	900 Mhz
Multiprocessors	8	16	2
Max. work group size	1024	512	1024
Available global mem.	8.5Gb bytes	1Gb bytes	1Gb bytes
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 28.1. Info for the three available devices on a MacBook Pro Retina Display

Device Name	Intel Xeon	AMD Radeon HD D300	AMD Radeon HD D300
Device Type	CPU	GPU	GPU
Clock rate	2600 Mhz	850 Mhz	150 Mhz
Multiprocessors	8	20	20
Max. work group size	1024	256	256
Available global mem.	12.8Gb bytes	2.1Gb bytes	2.1Gb bytes
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 28.2. Info for the three available devices on a Mac Pro

Device Name	NVIDIA Tesla K20c	NVIDIA Quadro K620	NVIDIA Tesla K20c
Device Type	GPU	GPU	GPU
Clock rate	705 Mhz	1124 Mhz	705 Mhz
Multiprocessors	13	3	13
Available global memory	5.0Gb bytes	2.1Gb bytes	5.0Gb bytes
Available local memory	49,152 bytes	49,152 bytes	49,152 bytes

Table 28.3. Info for the three available devices on a Dell workstation with dual Tesla K20c cards.

Example 28.2.

```
1 #include <stdio.h>
2
3 // Header file for OpenCL functions
4 #include <CL/opencl.h>
5
6 #define MAX_DEVICES 32
7
8 /*
9  * Get device info
10
11 Inputs: none
12
13 Outputs: useful info about each device
14 */
15
16 int main(int argc, const char * argv[]) {
17
18     cl_device_id devices[MAX_DEVICES];
19     cl_uint num_devices;
20     char name[128];
21
22     // Get the information about the version of OpenCL supported
23     cl_platform_id platform;
24     cl_int error = clGetPlatformIDs(1, &platform, NULL);
25
26     // Get a list of all the devices available to OpenCL
27     if (clGetDeviceIDs(platform, CL_DEVICE_TYPE_ALL, MAX_DEVICES, devices,
28                         &num_devices) == CL_SUCCESS) {
29         int i;
30         for (i=0; i<num_devices; ++i) {
31
32             // Get information about each device
33             cl_device_type device_type;
34             error = clGetDeviceInfo(devices[i], CL_DEVICE_TYPE,
35                                    sizeof(cl_device_type), &device_type, NULL);
36             if (error == CL_SUCCESS) {
37                 printf("%25s:\t", "Device type");
38                 if (device_type == CL_DEVICE_TYPE_CPU)
39                     printf("CPU ");
40                 if (device_type == CL_DEVICE_TYPE_GPU)
41                     printf("GPU ");
42                 if (device_type == CL_DEVICE_TYPE_ACCELERATOR)
43                     printf("Accelerator ");
44                 if (device_type == CL_DEVICE_TYPE_CUSTOM)
45                     printf("Custom ");
46                 if (device_type == CL_DEVICE_TYPE_DEFAULT)
47                     printf("Default ");
48                 printf("\n");
49             } else {
50                 switch(error) {
51                     case CL_INVALID_DEVICE: printf("Invalid device\n"); break;
52                     case CL_INVALID_VALUE: printf("Invalid value\n"); break;
53                     default: printf("Unknown error\n");
54                 }
55             }
56             clGetDeviceInfo(devices[i], CL_DEVICE_NAME, 128, name, NULL);
57             printf("%25s:\t%s\n", "Device name", name);
58
59             clGetDeviceInfo(devices[i], CL_DEVICE_VENDOR, 128, name, NULL);
59             printf("%25s:\t%s\n", "Device name", name);
60
61 }
```

```

62
63     int clock_rate;
64     clGetDeviceInfo(devices[i], CL_DEVICE_MAX_CLOCK_FREQUENCY,
65                         sizeof(int), &clock_rate, NULL);
66     printf("%25s:\t%dd Mhz\n", "Clock rate", clock_rate);
67
68     int num_cores;
69     clGetDeviceInfo(devices[i], CL_DEVICE_MAX_COMPUTE_UNITS,
70                         sizeof(int), &num_cores, NULL);
71     printf("%25s:\t%td\n", "Multiprocessors", num_cores);
72
73     cl_ulong max_global_mem;
74     clGetDeviceInfo(devices[i], CL_DEVICE_GLOBAL_MEM_SIZE,
75                         sizeof(cl_ulong), &max_global_mem, NULL);
76     printf("%25s:\t%ld bytes\n", "Available Global Memory",
77                         max_global_mem);
78
79     cl_ulong max_mem;
80     clGetDeviceInfo(devices[i], CL_DEVICE_LOCAL_MEM_SIZE,
81                         sizeof(cl_ulong), &max_mem, NULL);
82     printf("%25s:\t%ld bytes\n\n", "Available Local Memory", max_mem);
83 }
84
85 // Request a GPU device
86 cl_device_id gpu;
87 clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &gpu, NULL);
88
89 // Display which device got chosen for the dispatch queue
90 clGetDeviceInfo(gpu, CL_DEVICE_NAME, 128, name, NULL);
91 printf("%20s:\t%s\n", "Device chosen", name);
92 }
93
94 return 0;
95 }
```

Once a device is selected, you may want to determine some key characteristics such as the shared memory size, or the maximum number of threads per block, or other quantities for that particular device. That information can all be retrieved using the `clGetDeviceInfo` function:

```
clGetDeviceInfo(gpu, PROPERTY_DESIRED, size_of_data, &answer,
               NULL);
```

In the remainder of this section, there will be comments about the some of the parameters such as available memory made about the GPUs used on the course computer. Those values are all obtained using this command.

Chapter 29

Parallel OpenCL Using Workgroups

In this chapter the first parallel OpenCL example program is presented and dissected a little more carefully. The number of steps required to invoke a kernel is considerably more complex than was necessary in CUDA and some of the steps need closer examination to ease both code development and improve performance. One of the steps that will get extra scrutiny is how kernel functions are compiled, which was a much more transparent process when using CUDA. The methods for compiling shown in this text are called runtime compiling, meaning the compiling occurs at runtime. The other option is 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.

After the careful treatment of compiling the kernel, we will return to the finite difference kernel as a test case for doing parallel computations. The basic building blocks of parallel code using OpenCL are workgroups and work-items, which are functionally equivalent to blocks and threads in CUDA. We specify the number of workgroups and the number of work-items per workgroup when we put the kernel into the command queue. The organization of workgroups and work-items will also be covered in this chapter.

29.1 • Parallel OpenCL Example

Here is our first example of a parallel code using OpenCL, where we implement a simple finite difference approximation for a 1D data set. Note that not all GPUs are capable of handling double precision, and by default double precision is turned off.

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 / 2.;           \
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 Demonstrate a simple example for implementing a
26 parallel finite difference operator
27
28 Inputs: argc should be 2
29     argv[1]: Length of the vector of data
30
31 Outputs: the initial data and its derivative.
32 */
33
34 int main(int argc, const char * argv[]) {
35
36 // Read the input values
37 int N = atoi(argv[1]); // Get the length of the vector from input
38
39 // Get platform info
40 cl_platform_id platform;
41 int err = clGetPlatformIDs(1, &platform, NULL);
42
43 // Request a GPU device
44 cl_device_id device_id;
45 err = clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device_id,
46                     NULL);
47
48 // Setup the context for the GPU
49 cl_context context = clCreateContext(0, 1, &device_id, NULL, NULL,
50                                     &err);
51
52 // Before accessing the GPU, we must create a dispatch queue, which is
53 // the sequence of commands that will be sent to the GPU.
54 cl_command_queue queue = clCreateCommandQueue(context, device_id, 0,
55                                               &err);
56
57 // In OpenCL, the kernel is compiled at runtime.
58 const char* srccode = kernel;
59 cl_program program = clCreateProgramWithSource(context, 1, &srccode,
60                                                 NULL, &err);
61
62 // Compile the kernel code
63 err = clBuildProgram(program, 0, NULL, NULL, NULL);
64
65 // Create the kernel function from the compiled OpenCL code
66 cl_kernel kernel = clCreateKernel(program, "diff", &err);
67
68 // Allocate memory for the input arguments, data will be copied from
69 // the inputs to the GPU at the same time.
70 double dx = 2*M_PI/N;
71 double* u;
72 u = (double*)malloc(N*sizeof(double));
73 int i;

```

```

74    for (i=0; i<N; ++i)
75        u[ i ] = sin (i*dx);
76
77    double* du = (double*) malloc (N*sizeof(double));
78    cl_mem dev_u = clCreateBuffer(context , CL_MEM_READ_ONLY,
79                                     N*sizeof(double) , NULL, NULL);
80    err = clEnqueueWriteBuffer(queue , dev_u , 1, 0, N*sizeof(double) , u , 0,
81                               NULL, NULL);
82
83    // Set up the memory on the GPU where the answer will be stored
84    cl_mem dev_du = clCreateBuffer(context , CL_MEM_WRITE_ONLY,
85                                    N*sizeof(double) , NULL, NULL);
86
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
93    size_t maxwgs;
94    clGetDeviceInfo(device_id , CL_DEVICE_MAX_WORK_GROUP_SIZE,
95                    sizeof(size_t) , &maxwgs , NULL);
96
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 }
```

Example 29.1 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, copied to the device memory where the finite difference is computed, and the result copied back to host memory to be written to the terminal.

The GPU is set up on Lines 41–66. The setup process begins by requesting a platform ID. In Example 28.2, we saw how the full list of available devices along with their features can be obtained using the function `clGetDeviceIDs()` and the function `clGetDeviceInfo()`, but for the present example, we are simply choosing the default GPU device because we have requested only a single device ID. Next, the context for the device is set up. In this case, we are taking the default context, which is fine for most of our needs. The last step in setting up the GPU is to create a command

queue. The command queue is the sequence of commands, comparable to streams in CUDA, where commands are issued in order to the device, and it requires both the device ID and the context.

The next step is to actually create the binary code that the device will run, which requires a run-time compilation. Since this is a very new concept for us, and because it is obviously critical to the development process, we will spend some time carefully covering this step in Section 29.2.

Before the kernel is executed, we must first set up the memory allocation on the device. All device memory are objects of type `c1_mem`. In this example, the device memory is allocated in the global memory space because it has the `global` keyword in front as on Lines 4, 7. The input buffer is called `dev_u`, and is marked as *read only* because the GPU will not write to that buffer, only read the data from it. Before the GPU can read the buffer, though, it must be populated with the data from the host. To do this, we put our first command on the command queue, which is to copy data from the host to the device using the function `c1EnqueueWriteBuffer()`. This command can be blocking or non-blocking. In this case it is blocking as evidenced by the 1 in the third argument, which is equivalent to “true”. The host data stored in `u` will be copied to the device memory `dev_u` upon completion. Similarly, the buffer for storing the result is also created for the device memory `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 are hints to the compiler 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 `c1SetKernelArg` function. Note how allocated memory on the device are passed as `c1_mem` objects while parameter values such as `N` and `dx` are passed as simple pointers to host memory.

The next step is to insert the kernel into the command queue. In order to do this, there is some additional information required, namely the number of tasks to be completed and how many of the tasks can be done simultaneously. The function to call the kernel is

```
int c1EnqueueNDRangeKernel(queue, kernel, work_dim,
                           global_work_offset, global_work_size,
                           local_work_size, 0, NULL, NULL)
```

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 3 dimensions. The finite difference operation we’ve set here is naturally one-dimensional, so we’ve set the `work_dim` to 1. The `global_work_offset` is meant to allow for the indexing into the workgroup to be something other than all zeros. This is not always implemented in OpenCL libraries, so it’s safest to set the offset to `NULL`. The `global_work_size` describes the number of times the kernel is to be run. In our case, we have `N` grid points where we wish to compute the finite difference, so we will naturally set the global work size to be the array `{N, 0, 0}`. Since the `work_dim` is set to 1, then only the first entry in the global work size is evaluated, and hence we are able to get by with simply supplying a pointer to variable `N`. We couldn’t use `N` because it was an `int`, while 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 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()` on Line 94. It's important to remember that **the global work size must be greater than or equal to the work group size**, i.e. $N1 \geq \text{maxwgs}$ or problems can occur. Also, remember that the maximum work group size is total for all dimensions of the work group. Thus, if the work group size is three-dimensional with dimensions $\ell \times m \times n$, then the product ℓmn must be less than the maximum work group size. We'll see an example of this in the next section. For our purposes in the present example, we have assumed that N is bigger than `maxwgs`. So, suppose N is 1000, and the maximum work group size is 256 as in Table 28.2, then the kernel will operate on 256 grid points at one time, and this will be repeated until all 1000 finite differences are computed. The remainder of the arguments for this function are concerned with more advanced topics, so we will give the default values.

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, if we permitted that to happen, we may try to use the results of the finite difference operation before it's finished. Thus, similar to the blocking commands in MPI, we can block the program execution while the kernel is running by using the `clFinish(queue)` function. It basically stops the program until the queue is emptied of instructions. In this program, the block is there so that the finite difference operations are completed before the data is copied from the device back to the host using the function `clEnqueueReadBuffer()` function on Line 102.

Once the data is read back, we have to release all the memory allocated. The OpenCL objects, including all memory, kernel, program, queue, and context objects must be released. And finally, we also release the memory allocated on the host in the usual manner.

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 and to the equivalent CUDA version, see Figure 31.2.

29.2 • Compiling Kernel Functions

Let's return to Lines 57–66. To compile a kernel, we first require a string variable that contains the code for the kernel. This can be done in multiple ways. One way is to actually store a separate file as a kernel 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 how we handled kernel files for CUDA. The difference is that we would then use file I/O to read the kernel file as a character string. If you have several kernels that require building, or have a kernel that is more substantial than a handful of lines, then this is a more practical option. However, for the simple kernel in Example 29.1, we simply created the string as a static variable at the top of the file on Lines 2–13. 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 run-time, those continuation characters would not appear. Nonetheless, they are required in this context because string literal constants are not permitted to have line breaks.

On Line 2 of the kernel code it begins with `#pragma`, which is a way that a flag can be set temporarily for the compiler. This must be included here because the OpenCL

compiler by default does not handle variables of type `double`. Enabling the extension on Line 2, it tells the compiler to allow the use of double precision variables in the subsequent code. Whenever a compiler command, denoted by a leading '#' is put in code, it must be on a line of text by itself. Because of that, you should note the insertion of the "\n" on Line 2, which you know from the discussion on strings, represents a new line, hence the `#pragma` command is on a line of text by itself.

Whichever way the kernel code is stored into a string variable, the code must be assembled into a program object using

```
clCreateProgramWithSource(context, count, strings, lengths,
                           errcode)
```

Here, `context` is the context set up on Line 49. The kernel function may be broken into multiple strings, so `count` is the number of strings that define the complete kernel, `strings` is an array of string pointers, and `lengths` is an array of integers that give the lengths of each of the strings. In this example, we have only one string that defines the kernel. If that's the case, then `NULL` can be passed for the `lengths` variable. The program must then be compiled using the `clBuildProgram()` function. A number of options can be added to the build using the extra arguments, but for our purposes, we'll go with the default.

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 regular code compiles without errors. And then realize that the compilation will be happening at run-time when you can't go back and edit. With that in mind, it's good to check the error value and get feedback if there's a problem. Thus, Line 63 should be replaced with the following code:

```
52 // Compile the kernel code
53 err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
54 if (err != CL_SUCCESS)
55 {
56     size_t len;
57
58     clGetProgramBuildInfo(program, device_id, CL_PROGRAM_BUILD_LOG,
59                           NULL, NULL, &len);
60     char* buffer = (char*)malloc(len*sizeof(char));
61
62     printf("Error: Failed to build program executable!\n");
63     clGetProgramBuildInfo(program, device_id, CL_PROGRAM_BUILD_LOG,
64                           len, buffer, NULL);
65     printf("%s\n", buffer);
66     free(buffer);
67     exit(1); // kernel didn't compile, stop the program
68 }
```

Should the kernel not be successfully compiled, 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 you know how large the output is. The second call to the function 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 kernel code can then be turned into a binary instruction set

that can be delivered to the GPU using the function

```
clCreateKernel(program, "kernel_name", &err)
```

Note that the second argument on Line 66 matches the name of the kernel code on Line 2.

In Example 29.1, the kernel was written as a string within the same file where the kernel was built and used. Furthermore, when writing that code, it required the cumbersome use of string continuation characters, i.e. the '\' characters, at the end of every line making writing and editing the kernel code inconvenient. As kernel codes become more complex, this can be a hindrance, so an alternative strategy may be of interest.

One simple solution that is closer to how we normally program, is to put the kernel code in a separate file, e.g. `diff.ocl`, where the kernel is stored, the `.ocl` suffix to indicate that it is an OpenCL kernel function. Then we can add a generic function that will read the source of a file and store it as a string to load that kernel function from the file. To illustrate this, our example could be rewritten like this using two files, our `main.c` file and a `diff.ocl` file that contains the kernel as shown in Example 29.2.

Example 29.2.

File: diff.ocl

```
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 fclose(file);
30 return srccode;
31 }
32
33 /*
34 Demonstrate a simple example for implementing a
35 parallel finite difference operator
36
37 Inputs: argc should be 2
38 argv[1]: Length of the vector of data
39
40 Outputs: the initial data and its derivative.
41 */
42
43 int main(int argc, const char * argv[]) {
44
45 // Read the input values
46 int N = atoi(argv[1]); // Get the length of the vector from input
47
48 // Get platform info
49 cl_platform_id platform;
50 int err = clGetPlatformIDs(1, &platform, NULL);
51
52 // Request a GPU device
53 cl_device_id device_id;
54 err = clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device_id,
55 NULL);
56
57 // Setup the context for the GPU
58 cl_context context = clCreateContext(0, 1, &device_id, NULL, NULL,
59 &err);
60
61 // Before accessing the GPU, we must create a dispatch queue, which is
62 // the sequence of commands that will be sent to the GPU.
63 cl_command_queue queue = clCreateCommandQueue(context, device_id, 0,
64 &err);
65
66 // In OpenCL, the kernel is compiled at runtime.
67 char* srccode = GetKernelSource("diff.ocl");
68 cl_program program = clCreateProgramWithSource(context, 1,
69 (const char**)&srccode, NULL, &err);
70
71 // Compile the kernel code
72 err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
73
74 // Create the kernel function from the compiled OpenCL code
75 cl_kernel kernel = clCreateKernel(program, "diff", &err);
76
77 // Allocate memory for the input arguments, data will be copied from
78 // the inputs to the GPU at the same time.
79 double dx = 2*M_PI/N;

```

```

80    double* u;
81    u = (double*) malloc(N*sizeof(double));
82    int i;
83    for (i=0; i<N; ++i)
84        u[i] = sin(i*dx);
85
86    double* du = (double*) malloc(N*sizeof(double));
87
88    // Transfer the input data to the device
89    cl_mem dev_u = clCreateBuffer(context, CL_MEM_READ_ONLY,
90                                    N*sizeof(double), NULL, NULL);
91    err = clEnqueueWriteBuffer(queue, dev_u, 1, 0, N*sizeof(double), u, 0,
92                               NULL, NULL);
93
94    // Set up the memory on the GPU where the answer will be stored
95    cl_mem dev_du = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
96                                    N*sizeof(double), NULL, NULL);
97
98    // Define all the arguments for the kernel function
99    err = clSetKernelArg(kernel, 0, sizeof(cl_mem), &dev_u);
100   err |= clSetKernelArg(kernel, 1, sizeof(int), &N);
101   err |= clSetKernelArg(kernel, 2, sizeof(double), &dx);
102   err |= clSetKernelArg(kernel, 3, sizeof(cl_mem), &dev_du);
103
104   size_t maxwgs;
105   clGetDeviceInfo(device_id, CL_DEVICE_MAX_WORK_GROUP_SIZE,
106                   sizeof(size_t), &maxwgs, NULL);
107
108   size_t N1 = N;
109   err = clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &N1, &maxwgs, 0,
110                                NULL, NULL);
111   clFinish(queue);
112
113   err = clEnqueueReadBuffer(queue, dev_du, 1, 0, N*sizeof(double), du,
114                             0, NULL, NULL);
115
116    // Free the memory allocated on the GPU
117    clReleaseMemObject(dev_du);
118    clReleaseMemObject(dev_u);
119    clReleaseKernel(kernel);
120    clReleaseProgram(program);
121    clReleaseCommandQueue(queue);
122    clReleaseContext(context);
123
124    // Free the memory on the CPU
125    free(u);
126    free(du);
127    free(srccode);
128
129    return 0;
130}

```

The first observation is that we've moved the kernel `diff` into its own file named `diff.ocl`. The kernel then has the same formatting as any other C function that we may choose to write, which is far more intuitive than squeezing the program into a single character string. However, once that kernel function is on its own, we need a way to get it back into our main code.

To get the kernel back as a string, we've added a new function on Line 10 that has the sole purpose of reading a text file and storing it into a string. Because there are some additional functions in there that we did not cover in Section 4.2, let's take

a closer look at some new functions used on file pointers. In order to read the file, we will need to know how many characters are in the file containing the kernel so we need a means of measuring the length of the file. Now, when a file is opened, there is an implicit pointer to a location in the file, 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. In any case, on Line 16, that implicit pointer is forced to move to the end of the file using the `fseek` function. Once at the end, on Line 19, the `ftell` function gives the position of the implicit pointer relative to the beginning of the file, so in other words, if the pointer is at the end, then `ftell` will return the length of the file. We now know how much memory must be allocated to store the text of the file, which is done on Line 25. Note that one is added to the length because we need space for the terminating '\0' character at the end. On Line 22, we then move the implicit pointer back to the beginning so that when the file is subsequently read, we get the entire contents of the file.

The next modification is on Line 69 where the variable `srccode` is recast as a `const char*` to satisfy the argument of the function `clCreateProgramWithSource`.

The rest of the program is the same until we reach the end, where on Line 127, we must remember to free the memory for `srccode` that was allocated in the function `GetKernelSource`.

One final comment about this alternative way of loading the source code for kernels is that since it's something that will be done more than once, 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, we will use this new strategy and assume 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 `.ocl` suffix.

29.3 ■ Organization of Workgroups

In the preceding example, the data was organized into a linear fashion, namely a single linear list of N gridpoints. The following program is the equivalent only using a two-dimensional grid. Inside the kernel, the dimensions and the location within the grid can be determined using the `get_global_size` and `get_global_id` functions where the argument is the dimension of interest (0 indexed).

To illustrate the use of multiple dimensions, we could apply this method to computing the two-dimensional Laplacian as in Example 29.3

Example 29.3.

File: diff.ocl

```

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 /*
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 int main(int argc, const char * argv[]) {
23
24 // Read the input values
25 int M= atoi(argv[1]); // Get the length of the vector from input
26 int N= atoi(argv[2]);
27
28 // Request platform ID
29 cl_platform_id platform;
30 int 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 // Setup the context for the GPU
38 cl_context context = clCreateContext(0, 1, &device_id, NULL, NULL,
39 &err);
40
41 // Set up the command queue
42 cl_command_queue queue = clCreateCommandQueue(context, device_id, 0,
43 &err);
44
45 // Load the source code from a file
46 char* srccode = GetKernelSource("diff.ocl");
47 cl_program program = clCreateProgramWithSource(context, 1, &srccode,
48 NULL, &err);
49
50 // Compile the kernel code
```

```

51 err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
52
53 // Create the kernel function from the compiled OpenCL code
54 cl_kernel kernel = clCreateKernel(program, "diff", &err);
55
56 // Allocate memory for the input arguments and initialize
57 double dx = 2*M_PI/M;
58 double dy = 2*M_PI/N;
59 double* u = (double*)malloc(M*N*sizeof(double));
60 int i, j;
61 for (i=0; i<M; ++i)
62     for (j=0; j<N; ++j)
63         u[i+j*M] = sin(i*dx)*cos(j*dy);
64
65 double* du = (double*)malloc(M*N*sizeof(double));
66 cl_mem dev_u = clCreateBuffer(context, CL_MEM_READ_ONLY,
67                               M*N*sizeof(double), NULL, NULL);
68 err = clEnqueueWriteBuffer(queue, dev_u, 1, 0, M*N*sizeof(double), u,
69                           0, NULL, NULL);
70
71 // Set up the memory on the GPU where the answer will be stored
72 cl_mem dev_du = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
73                               M*N*sizeof(double), NULL, NULL);
74
75 // Define all the arguments for the kernel function
76 err = clSetKernelArg(kernel, 0, sizeof(cl_mem), &dev_u);
77 err |= clSetKernelArg(kernel, 1, sizeof(double), &dx);
78 err |= clSetKernelArg(kernel, 2, sizeof(double), &dy);
79 err |= clSetKernelArg(kernel, 3, sizeof(cl_mem), &dev_du);
80
81 // Execute the kernel
82 size_t dims[2] = {M, N};
83 err = clEnqueueNDRangeKernel(queue, kernel, 2, NULL, dims, NULL, 0,
84                             NULL, NULL);
85 clFinish(queue);
86
87 // Retrieve the results from the device
88 err = clEnqueueReadBuffer(queue, dev_du, 1, 0, M*N*sizeof(double), du,
89                           0, NULL, NULL);
90
91 // Free the memory allocated on the GPU
92 clReleaseMemObject(dev_du);
93 clReleaseMemObject(dev_u);
94 clReleaseKernel(kernel);
95 clReleaseProgram(program);
96 clReleaseCommandQueue(queue);
97 clReleaseContext(context);
98
99 // Free the memory on the CPU
100 free(u);
101 free(du);
102 free(srccode);
103
104 return 0;
105 }

```

On Line 66, note that the two-dimensional array is still being allocated 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 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.

On Line 82, the dimensions of the device are specified. In this particular example, the number of cores requested will conform to the number of gridpoints in the domain, namely $M \times N$. Ideally, we would want to choose a workgroup size that is maximally efficient for the given dimensions. For the machine on which this example was written, the maximum workgroup size is 256 ($= 16 \times 16$). It's reasonable to assume that both M and N are much bigger than that. What will happen is that the data will be subdivided into subarrays of size 16×16 , and then each subarray will be executed by the kernel. Obviously, for maximum efficiency, it is wise to choose M and N to be even multiples of the dimensions of the workgroup size. There is no guarantee which order the subarrays will be completed.

On Line 83, the workgroup size vector is specified as NULL. When this is done, the OpenCL package will choose the workgroup size for you, and it may or may not be ideal, which is why it's important to specify it when possible. For example, if $M = N = 256$, then the workgroup size chosen in my experiments was 64×4 . This is actually an interesting choice because it focuses the memory access for any given workgroup into a narrower range than trying 16×16 because the i variable corresponds to sequential entries in the dataset. On the other hand, if $M = N = 500$, the workgroup size was 25×1 , an apparently much less efficient choice. This illustrates the importance of understanding the hardware and the grid dimensions you are using and controlling them carefully.

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 time we've opted to match the data size with the dimensions in the requested range, we can get the dimensions of the data through the `get_global_size` functions where the argument is 0, 1, or 2 indicating which dimension is desired as seen on Line 8. Similarly, the index within the two-dimensional domain is obtained on Line 10 similar to how we did it in the previous example.

29.4 • Work-Items

Each workgroup on the device is further subdivided into work-items, where the work-items are able to share a small amount of memory per workgroup. The work-item dimension information is specified by the workgroup size variable when the kernel is placed on the command queue as was done on Line 83 of Example 29.3. To specify the number of work-items to be a 16×16 array per workgroup, we would change Lines 82–83 to

```

70 // Execute the kernel
71 size_t dims[2] = {M, N};
72 size_t localdims[2] = {16, 16};
73 err = clEnqueueNDRangeKernel(queue, kernel, 2, NULL, dims, localdims,
74 0, NULL, NULL);

```

where we have created a second set of dimensions `localdims` to be the number of work-items per workgroup, and they are listed as the argument following the dimensions of the workgroups. If the number of work-items are not specified, then the system will choose the arrangement for you. For most kernels, the actual arrangement of workgroups and work-items is important for algorithm development, so be sure to specify them rather than trusting the OpenCL driver. As noted at that time, the workgroup size limitations for the device selected can be obtained using the `clGetDeviceInfo()` function.

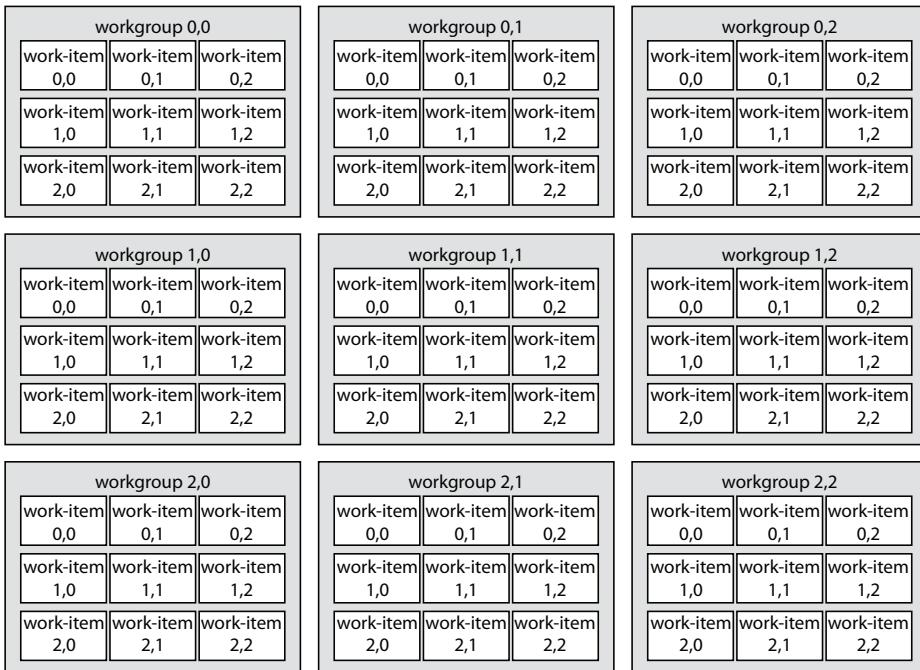


Figure 29.1. Diagram of the organization of work-items within workgroups with both in arrays of different dimensions. The dimensions of the work-items arrays need not be the same as the workgroups.

So far we have only used global indexing, but when using shared memory within a workgroup, we will also need to know the local indexing. To get the local information, the local workgroup size can be obtained by calling `get_local_size(dim)` to get the size of the workgroup, and `get_local_id(dim)` to get the local index within the workgroup. The number of workgroups can be obtained with `get_num_groups(dim)`, and the index of the workgroup is given by `get_group_id(dim)`.

Putting it together, one can imagine the picture for our computations to be organized as in Figure 29.1, where the task is broken down into groups of workgroups, and each workgroup further subdivided into work-items working in tandem.

29.5 ▪ Error Handling

To build solid code, the error codes created by the various OpenCL functions must 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 errors via an `err` 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. This is also done by comparing the error value to the macro `CL_SUCCESS`. When the error value is non-zero, it means a problem occurred, and the value of the error will give a clue as to what the problem is. Similar to CUDA, we can add a function that will handle error codes in the background, which will be helpful should a problem arise

when you are implementing your codes. A header file `cl_handle_error.h` containing the following code makes it easier to check for errors during the execution of your program.

cl_handle_error.h

```
1 static void HandleError( int err, const char* file, int line) {
2     if (err != CL_SUCCESS) {
3         printf("Error code %d in %s at line %d\n", err, file, line);
4         exit(1);
5     }
6 }
7 #define HANDLE_ERROR( err ) (HandleError(err, __FILE__, __LINE__))
```

In the examples we've been using, we've used the variable name `err` to collect the error messages, so after each OpenCL function, insert

```
HANDLE_ERROR(err);
```

on the line after to catch errors during run-time. 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, you may not want to terminate, but respond to the problem according to the situation. It's up to you to create your own error handler, or modify this one, to deal with this more sophisticated approach.

Chapter 30

GPU Memory

Up to this point we have been exclusively using 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. Unlike the specialized CUDA language that has a closer connection with the underlying hardware, OpenCL has a more generic approach and hence breaks down memory into global, local, and constant. Local and constant memory is the fastest, so it is important to take advantage of that memory where appropriate. In this chapter, we'll look at how to invoke local and constant memory.

30.1 • Local memory

Though it is highly device dependent, one bottleneck in the codes 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. Thus far, we have been using global memory throughout. Because a significant latency arises due to all the work-items or workgroups making requests from global memory simultaneously, they are effectively forced to queue up sequentially resulting in no better performance than would be achieved by using the CPU in the first place.

To gain in speed, we need to take advantage of local memory. Each block has an allocation of local memory assigned to it for which read/write times are significantly shorter than read/write to global memory. So why don't we just use one workgroup and only local memory? Local memory is much smaller than available global memory space. Compare the entries for global and local memory space in Tables 28.1–28.3.

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 local storage.

For example, a kernel with a dynamically allocated array of `double` would be declared as:

```
kernel void foo(local double* data)
```

and then the memory is allocated in the host code like this:

```
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 we have been using throughout this section:

Example 30.1.

File: diff.ocl

```

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
11 * Outputs:
12 *   double* du: finite difference of u
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
7 #ifndef M_PI
8 #define M_PI 3.1415926535897932384626433832795
9 #endif
10 /*
11  * Demonstrate a simple example for implementing a
12  * parallel finite difference operator
13
14  * Inputs: argc should be 2
15  *         argv[1]: Length of the domain
16
17  * Outputs: the initial data and its derivative.
18 */
19
20 int main(int argc, const char * argv[]) {
21
22     // Read the input values
23     int N = atoi(argv[1]); // Get the length of the vector from input
24
25     // Request the platform info
26     cl_platform_id platform;
27     int err = clGetPlatformIDs(1, &platform, NULL);
28
29     // Request a GPU device
30     cl_device_id device_id;
31     err = clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device_id,
32                         NULL);
33
34     // Setup the context for the GPU
35     cl_context context = clCreateContext(0, 1, &device_id, NULL, NULL,
36                                         &err);
37
38     // Before accessing the GPU, we must create a dispatch queue, which is
39     // the sequence of commands that will be sent to the GPU.
40     cl_command_queue queue = clCreateCommandQueue(context, device_id, 0,
41                                                   &err);
42
43     // In OpenCL, the kernel is compiled at runtime,
44     // try to do this only once and then save it.
45     char* srccode = GetKernelSource("diff.ocl");
46     cl_program program = clCreateProgramWithSource(context, 1, (const char
47     **) &srccode,
48                                         NULL, &err);
49
50     // Compile the kernel code
51     err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
52
53     // Create the kernel function from the compiled OpenCL code
54     cl_kernel kernel = clCreateKernel(program, "diff", &err);
55
56     // Allocate memory for the input arguments, data will be copied from
57     // the inputs to the GPU at the same time.
58     double dx = 2*M_PI/N;
59     double* u;
60     u = (double*) malloc(N*sizeof(double));
61     int i;
```

```

62  for (i=0; i<N; ++i)
63      u[i] = sin(i*dx);
64
65  double* du = (double*)malloc(N*sizeof(double));
66  cl_mem dev_u = clCreateBuffer(context, CL_MEM_READ_ONLY,
67                                N*sizeof(double), NULL, NULL);
68  err = clEnqueueWriteBuffer(queue, dev_u, 1, 0, N*sizeof(double), u, 0,
69                            NULL, NULL);
70
71 // Set up the memory on the GPU where the answer will be stored
72 cl_mem dev_du = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
73                                N*sizeof(double), NULL, NULL);
74
75 size_t maxwgs;
76 clGetDeviceInfo(device_id, CL_DEVICE_MAX_WORK_GROUP_SIZE,
77                  sizeof(size_t), &maxwgs, NULL);
78
79 // Define all the arguments for the kernel function
80 err = clSetKernelArg(kernel, 0, sizeof(cl_mem), &dev_u);
81 err |= clSetKernelArg(kernel, 1, sizeof(int), &N);
82 err |= clSetKernelArg(kernel, 2, sizeof(double), &dx);
83 err |= clSetKernelArg(kernel, 3, (maxwgs+2)*sizeof(double), NULL);
84 err |= clSetKernelArg(kernel, 4, maxwgs*sizeof(double), NULL);
85 err |= clSetKernelArg(kernel, 5, sizeof(cl_mem), &dev_du);
86
87 size_t N1 = N;
88 err = clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &N1, &maxwgs, 0,
89                               NULL, NULL);
90 clFinish(queue);
91
92 err = clEnqueueReadBuffer(queue, dev_du, 1, 0, N*sizeof(double), du,
93                           0, NULL, NULL);
94
95 // Free the memory allocated on the GPU
96 clReleaseMemObject(dev_du);
97 clReleaseMemObject(dev_u);
98 clReleaseKernel(kernel);
99 clReleaseProgram(program);
100 clReleaseCommandQueue(queue);
101 clReleaseContext(context);
102
103 // Free the memory on the CPU
104 free(u);
105 free(du);
106 free(srccode);
107
108 return 0;
109 }

```

In this example, suppose we have six workgroups 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 thread, (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, 23. Stage 1 of the kernel, where the global memory is copied to the local memory is done on Lines 26–30. Here, each work-item copies its corresponding global memory location to the local memory address on Line 26. However, because we are doing a finite difference approximation that has a three-point stencil, we also need the two data points on either side. Therefore, the first

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

group id = 0				group id = 1				group id = 2														
23	0	1	2	3	4	3	4	5	6	7	8	7	8	9	10	11	12					

group id = 3						group id = 4					group id = 5											
11	12	13	14	15	16	15	16	17	18	19	20	19	20	21	22	23	0					

local du

group id = 0				group id = 1				group id = 2													
0	1	2	3	4	5	6	7	8	9	10	11										

group id = 3				group id = 4				group id = 5													
12	13	14	15	16	17	18	19	20	21	22	23										

Figure 30.1. Illustration of the memory layout for local memory in Example 30.1

and last work-items in the workgroup 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 workgroup 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 in local memory on Line 31.

Finally, stage 3 is done 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 used here.

Unfortunately, the kernel in Example 30.1 will not necessarily 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, 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, the work-items have to certify that they've all completed their global to local memory copy. This is accomplished by syncing the threads 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 workgroup will stop and wait at the synchronization point until all work-items reach that point. When using `barrier(CLK_LOCAL_MEM_FENCE)`, care must be taken to make sure that it's not, for example, inside an `if` statement where only some of the work-items reach the synchronization point. If that happens, then the program will lock up waiting for the work-items that can never reach the synchronization point. 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 }
```

30.2 ▪ Constant Memory

There is another form of fast-access, low-latency memory available on the GPU, and that is constant memory. 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 block. Unlike CUDA, constant memory that must be dynamically allocated must still go through the argument list, but we can designate them with the constant label, moving them into the 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, we assume that the input data *u* is small enough to fit in the constant memory buffer.

Example 30.3.

File: *diff.ocl*

```

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                   int N,
18                   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 // Setup 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 = clCreateCommandQueue(context, device_id, 0,
42 // &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.ocl");
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, 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 // Set up the memory on the GPU where the answer will be stored
80 cl_mem dev_du = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
81                                 N*sizeof(double), NULL, NULL);
82
83 size_t maxwgs;
84 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 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
93 // Run the kernel
94 size_t N1 = N;
95 err = clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &N1, &maxwgs, 0,
96                             NULL, NULL);
97 clFinish(queue);
98
99 // Retrieve the results
100 err = clEnqueueReadBuffer(queue, dev_du, 1, 0, N*sizeof(double), du,
101                           0, NULL, NULL);
102
103 // Free the memory allocated on the GPU
104 clReleaseMemObject(dev_du);
105 clReleaseMemObject(dev_u);
106 clReleaseKernel(kernel);

```

```

107    clReleaseProgram(program);
108    clReleaseCommandQueue(queue);
109    clReleaseContext(context);
110
111    // Free the memory on the CPU
112    free(u);
113    free(du);
114    free(srccode);
115
116    return 0;
117 }
```

The first difference is that the variable `u` is defined as constant on Line 16. Note that we use the full word `constant` as compared to the C language `const` attribute, and this is because they mean different things. The latter simply tells the compiler that the value is fixed for compilation purposes, while the former tells the compiler to store the variable in higher speed constant memory space on the GPU.

Next, because constant memory space is limited, but we allowed the data size to be determined at run time, we need to double check that we won't overrun the constant memory buffer. Thus, on Line 47, we request the size of the constant memory buffer for the current device and then check if the constant memory we will request is within that limit.

The next change is one we could have done before, but introduce here. On Line 75, we combine the buffer creation and initialization at the same time. Thus, the following two code snippets are equivalent:

```

cl_mem dev_u = clCreateBuffer(context, CL_MEM_READ_ONLY,
                               N*sizeof(double), NULL, NULL);
err = clEnqueueWriteBuffer(queue, dev_u, true, 0,
                           N*sizeof(double), u, 0, NULL, NULL);
```

and

```

cl_mem dev_u = clCreateBuffer(context,
                               CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                               N*sizeof(double), u, NULL);
```

The second snippet combines the buffer creation and data initialization in one step. Note also that for the `u` array, we are designating it as constant memory, so that means we must state that it will be read only, which is what we were setting all along anyway.

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 31.2.

Chapter 31

Command Queues

In CUDA we were introduced to the concept of streams, which are a way of executing multiple distinct kernels at the same time. OpenCL has the same concept, named *command queues*. That means that a device can receive multiple kernels in the same command queue, they can be synchronized, made sequential, or not, depending on the algorithm, and load balancing is left to the device to handle the different tasks. If using OpenCL with an NVIDIA GPU card, then it is possible to overlap compute kernels with memory transfers as discussed in Chapter 25, but most GPU devices do not currently support simultaneous data transfer and computation. On the other hand, OpenCL does make it easier to utilize multiple devices so that work can be done simultaneously on the CPU and multiple GPUs, and each requiring their own queue. In this chapter, we'll look at how multiple command queues can be used to take advantage of multiple GPUs and multiple CPUs in order to complete a computational task. The chapter concludes with a discussion about events that allow us to put timing events into the command queue in order to measure the performance of an OpenCL kernel.

31.1 • Using Multiple Devices

An implementation of multiple devices, in this case two GPUs, is illustrated in Example 31.1 below where we have left out the kernel function `diff.ocl` which is unchanged from the previous example:

Example 31.1.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4 #include <string.h>
5 #include <OpenCL/opencl.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   // Get the length of the vector from input
24   unsigned long N = atoi(argv[1]);
25
26   // Request the number of GPU devices
27   cl_device_id* device_id;
28   cl_uint num_devices;
29   int err;
30   err = clGetDeviceIDs(NULL, CL_DEVICE_TYPE_GPU, 0, NULL, &num_devices);
31
32   // Set the device_id's for each GPU
33   device_id = (cl_device_id*)malloc(num_devices*sizeof(cl_device_id));
34   err = clGetDeviceIDs(NULL, CL_DEVICE_TYPE_GPU, num_devices, device_id,
35                       NULL);
36
37   // Setup the context for the GPUs
38   cl_context context = clCreateContext(0, num_devices, device_id, NULL,
39                                       NULL, &err);
40
41   // Create a command queue for each device
42   cl_command_queue* queue;
43   queue=(cl_command_queue*)malloc(num_devices*sizeof(cl_command_queue));
44   for (int i=0; i<num_devices; ++i)
45     queue[i] = clCreateCommandQueue(context, device_id[i], 0, &err);
46
47   // Build the program
48   char* srccode = GetKernelSource("diff.ocl");
49   cl_program program = clCreateProgramWithSource(context, 1,
50                                                 (const char**)&srccode, NULL, &err);
51
52   // Compile the kernel code
53   err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
54
55   // Create a kernel for each device
56   cl_kernel* kernel = (cl_kernel*)malloc(num_devices*sizeof(cl_kernel));
57   for (int i=0; i<num_devices; ++i)
58     kernel[i] = clCreateKernel(program, "diff", &err);
59
60   // Allocate memory for the input arguments
61   double dx = 2*M_PI/N;
62   double* u;
63   u = (double*)malloc((N+2)*sizeof(double));
64   for (int i=0; i<N+2; ++i)
65     u[i] = sin((i-1)*dx);
66
67   // Mark the start and stop locations of the memory
68   // that will be passed to the device
69   size_t* mem_start = (size_t*)malloc(num_devices*sizeof(size_t));
70   size_t* mem_len = (size_t*)malloc(num_devices*sizeof(size_t));
71   mem_start[0] = 1;
72   mem_len[0] = N/num_devices + (!num_devices > 0);
73   for (int i=1; i<num_devices; ++i) {
74     mem_start[i] = mem_start[i-1] + mem_len[i-1];
75     mem_len[i] = N/num_devices + (!num_devices > i);
76   }

```

```

77 // Allocate space for the results
78 double* du = (double*)malloc(N*sizeof(double));
79
80
81 // Create input and output buffers for each device
82 cl_mem* dev_u = (cl_mem*)malloc(num_devices*sizeof(cl_mem));
83 for (int i=0; i<num_devices; ++i)
84     dev_u[i] = clCreateBuffer(context,
85                             CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
86                             (mem_len[i]+2)*sizeof(double),
87                             &(u[mem_start[i]-1]), NULL);
88
89 // Set up the memory on the GPU where the answer will be stored
90 cl_mem* dev_du = (cl_mem*)malloc(num_devices*sizeof(cl_mem));
91 for (int i=0; i<num_devices; ++i)
92     dev_du[i] = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
93                             mem_len[i]*sizeof(double), NULL, NULL);
94
95 // Send the data to the devices
96 cl_event write_event[num_devices];
97 for (int i=0; i<num_devices; ++i)
98     err = clEnqueueWriteBuffer(queue[i], dev_u[i], false, 0,
99                             (mem_len[i]+2)*sizeof(double), &(u[mem_start[i]-1]), 0, NULL, &
100    write_event[i]);
101
102 // Set the arguments
103 for (int i=0; i<num_devices; ++i) {
104     // Define all the arguments for the kernel function
105     err = clSetKernelArg(kernel[i], 0, sizeof(cl_mem), &dev_u[i]);
106     err |= clSetKernelArg(kernel[i], 1, sizeof(int), &N);
107     err |= clSetKernelArg(kernel[i], 2, sizeof(double), &dx);
108     err |= clSetKernelArg(kernel[i], 3, sizeof(cl_mem), &dev_du[i]);
109 }
110
111 // Execute the kernel
112 cl_event exec_event[num_devices];
113 for (int i=0; i<num_devices; ++i)
114     err = clEnqueueNDRangeKernel(queue[i], kernel[i], 1, NULL,
115                             &(mem_len[i]), NULL, 1, &write_event[i], &exec_event[i]);
116
117 // Retrieve the data from the device
118 for (int i=0; i<num_devices; ++i)
119     err = clEnqueueReadBuffer(queue[i], dev_du[i], true, 0,
120                             mem_len[i]*sizeof(double), &(du[mem_start[i]-1]),
121                             1, &exec_event[i], NULL);
122
123 // Free the memory allocated on the GPU
124 for (int i=0; i<num_devices; ++i) {
125     clReleaseMemObject(dev_du[i]);
126     clReleaseMemObject(dev_u[i]);
127     clReleaseKernel(kernel[i]);
128     clReleaseCommandQueue(queue[i]);
129 }
130 clReleaseProgram(program);
131 clReleaseContext(context);
132
133 // Free the memory on the CPU
134 free(queue);
135 free(kernel);
136 free(dev_u);
137 free(dev_du);
138 free(mem_len);
139 free(mem_start);

```

```

139     free (device_id) ;
140     free (u) ;
141     free (du) ;
142
143     return 0;
144 }
```

If multiple devices will be used, the first task is to make sure multiple devices are available. On Line 30, the number of available GPU devices is requested by setting the third and fourth arguments to 0 and NULL respectively. The result is that `num_devices` will contain the number of available devices. This is done first so that when allocating the necessary storage for each of the devices, the number is known such as allocating for enough device ID's on Line 33. Only one context is created, but it is aware of the full list of devices that will be available for use.

On Line 45, a command queue is created for each device so that kernels can be sent to each. The third argument being set as `false` means that the memory transfer will be done as non-blocking.

On Lines 69–76, the bounds of the sub-arrays that will be sent to each of the devices are mapped out. Most of this should look straightforward except perhaps the inequality that appears in parentheses on Lines 72, 75. If the inequality is true, then the value in the parentheses will evaluate to the integer 1, and 0 if false. Thus, the expression in the parentheses will add one to the length where needed if N is not evenly divisible by `num_devices`. For example, suppose $N = 11$ and `num_devices = 3`. Then we should expect that the first two devices will have four points and the third device will have three for a total of 11. The integer arithmetic for $N/\text{num_devices}$ will evaluate to 3 because the fraction is discarded. The expression $N\% \text{num_devices}$ is the mod operator and $11 \bmod 3$ is 2, which is bigger than 0. Thus, $(N\% \text{num_devices} > 0)$ evaluates to `true`, which is equal to the integer 1. Therefore, `mem_1en[0]` will have value $3+1=4$. In this way, the partial overage is distributed as one extra grid point for a subset of the devices. The various pieces of the input and output data are mapped to the devices through the `c1CreateBuffer` function calls on Lines 84, 92, and then written to the devices on Line 98.

Note that there is no call to the function `c1Finish` as has been done in previous examples. Instead, events are being used to make sure tasks are completed before the next step is taken. On Line 98, a pointer to an event is given as the last argument in the `c1EnqueueWriteBuffer` function which marks the task with an event tag. To ensure that the buffer is completely transferred before the kernel function is applied on Line 113, the 7th and 8th arguments in the `c1EnqueueNDRangeKernel` function indicate that there is one event, namely `write_event[i]`, that must be completed before the kernel is executed. Similarly, the 9th argument in `c1EnqueueNDRangeKernel` assigns the event `exec_event[i]` to the kernel execution and subsequently on Line 118 the `c1EnqueueReadBuffer` is instructed to wait for that event before retrieving the data from the device.

To see that this strategy actually works, check out Figure 31.1, where you can see that using 2 devices results in a speed-up by a factor of about 2. In the figure, the multiple streams corresponds to using multiple command queues, but the performance improvements in that case only apply to the case of using an NVIDIA GPU that allows for stacking of memory transfers and computation at the same time as described in Chapter 25

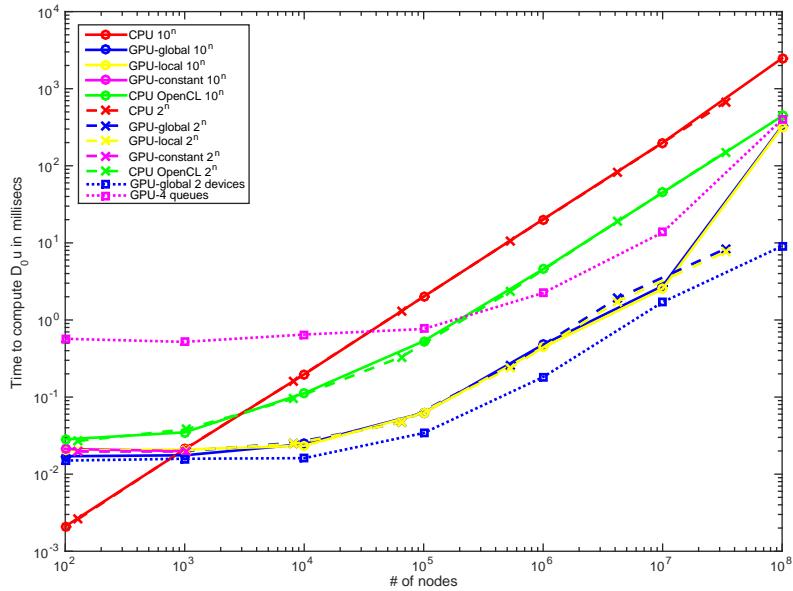


Figure 31.1. Timing plots for the kernel functions only for the version with 1, 2, and 4 streams as well as the previous timing data. Note that the plots for 2 and 4 streams coincide with the texture memory plot for larger values of N

31.2 • Measuring Performance

So far in this section, we've seen several different implementations of the same task, which is a common task required for numerical methods. Which method is the best? Obviously, the whole purpose of exploring the use of GPUs is to gain speed and so we need some means of evaluating the speed of kernels. This can be done using OpenCL events. To make this possible, the first thing we have to do is enable events for our command queue. This is done when the queue is created by replacing the third argument, that has thus far been set to zero, to be as follows:

```
cl_command_queue queue
= clCreateCommandQueue(context, device_id,
CL_QUEUE_PROFILING_ENABLE, &err);
```

This alerts the command queue that a profiling event will be requested at some point.

Next, a pointer to a `cl_event` structure is passed as the last argument when the kernel is run. For accurate timing, we also want to make sure the current queue is flushed of events beforehand. Thus, the command to run the kernel will look like:

```
clFinish(queue);
cl_event event;
err = clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &N, NULL,
0, NULL, &event);
```

Recall that when a job is submitted to the device, the kernel function is run asynchronously with the host code, so the time to finish the kernel must be after all the workgroups have completed, so we need another block in the code on the host side to wait until the device is finished. This is accomplished with the function `clWaitForEvents()` that takes as arguments the number of events to wait for, and an array of events that are to be completed. Therefore, after the kernel is submitted to run, it's followed by

```
clWaitForEvents(1, &event);
```

After this, we can then retrieve information about the execution time from the event data by using the function `clGetEventProfilingInfo()`. Both the start and end times are retrieved in the same manner, and it reports the data in nanoseconds. For example:

```
cl_ulong start_time, end_time;
clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_START,
                       sizeof(start_time), &start_time, NULL);
clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_END,
                       sizeof(start_time), &end_time, NULL);
```

Finally, to get the elapsed time, simply take the difference between the end and start times to get the number of nanoseconds that elapsed. The example below shows how to record the time required to execute the kernel function `diff` on the device:

Example 31.2.

```
1  cl_command_queue queue
2      = clCreateCommandQueue(context, device_id,
3                               CL_QUEUE_PROFILING_ENABLE, &err);
4
5 // Other code for setting up the kernel such as preparing buffers,
6 // compiling the kernel, etc.
7
8 clFinish(queue);
9 cl_event event;
10 err = clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &N, NULL, 0,
11                               NULL, &event);
12 clWaitForEvents(1, &event);
13 cl_ulong start_time, end_time;
14 clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_START,
15                         sizeof(start_time), &start_time, NULL);
16 clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_END,
17                         sizeof(start_time), &end_time, NULL);
18 // elapsed time in nanoseconds
19 cl_ulong elapsedtime = end_time - start_time;
```

Now that we know how to record the time elapsed for a kernel function, let's try it out. In this section there have been several methods described for implementing the finite difference approximation through a device kernel function. We can now compare the different methods to see which is faster. Figure 31.2 shows the following different 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 execute a loop in C.

2. Using a GPU with OpenCL and global memory
3. Using a GPU with OpenCL and local memory
4. Using a GPU with OpenCL and constant memory
5. Using multiple cores of a CPU with OpenCL

For all these cases, they were tested using nodes in powers of 10 and also in powers of 2 to see if there are any improvements to be had when the workgroup size can be optimized. The last case is worth taking special notice, where the only change in the code is requesting a CPU device rather than a GPU in the `clGetDeviceIDs()` function like so:

```
int err = clGetDeviceIDs(NULL, CL_DEVICE_TYPE_CPU, 1,  
                        &device_id, NULL);
```

Figure 31.2 shows that for sufficiently large problems, the GPU solution can be quite efficient, achieving up to a 40-fold improvement in speed. There is a clear degradation of performance at the very high end of the range which requires further investigation. The multicore CPU implementation shows an approximately 8 fold increase in speed, which is to be expected since it is an 8 multiprocessor computer. Finally, note that using power of 2 grids vs. powers of 10 does not make a significant difference, at least for this simple kernel.

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.

Exercises

- 31.1. Build and run Example 28.1 to make sure you are able to access the OpenCL library and a suitable GPU.
- 31.2. Modify Example 29.1 to read the device properties using `clGetDeviceInfo`. Use the properties to subdivide the domain in such a way that a maximal number of work-items are used per workgroup to cover the given dimensions M, N . Use shared memory for the local array to improve efficiency.
- 31.3. 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 vs. computing the finite difference.

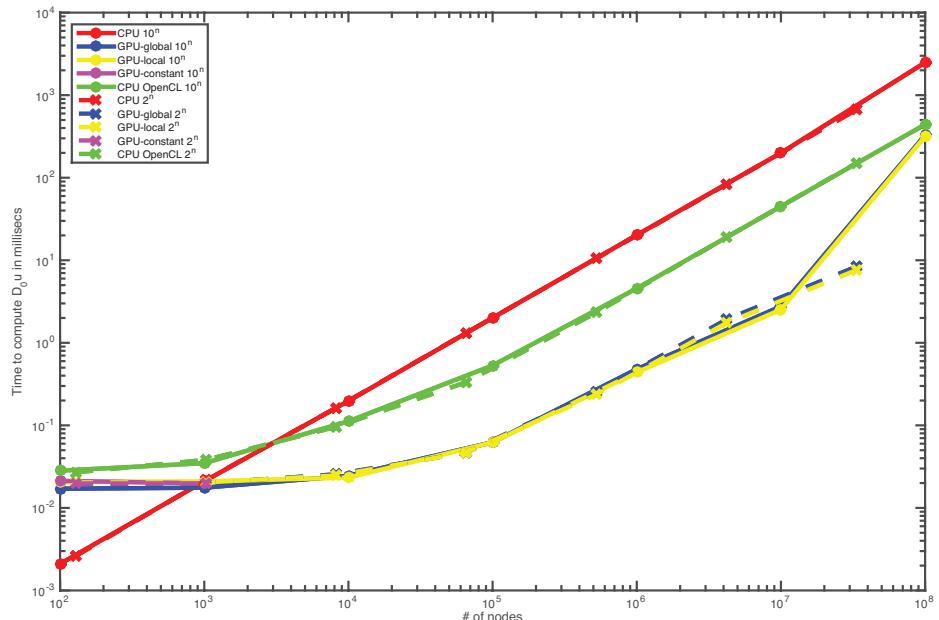


Figure 31.2. Scaling properties of 1D finite central difference using various versions of parallelism using OpenCL compared to a serial CPU implementation. The horizontal axis shows the number of nodes in the grid, and the vertical axis shows the time to complete the finite difference kernel function in milliseconds. The legend refers to: (CPU) Serial CPU code, no OpenCL code, (GPU-global) Results from using Example 29.1, (GPU-local) Results from explicitly using local memory as in Example 30.2, (GPU-constant) Results from using constant memory as in Example 30.3, (CPU-OpenCL) Results from choosing a CPU device instead of a GPU in Example 29.1.

Chapter 32

OpenCL Libraries

The OpenCL package itself does not come with any ready-made libraries, but there are some available elsewhere. Some of the libraries are less developed than their CUDA relatives, but they do provide most of the basic functionality that is needed to complete the projects contained in Part VI. The clMAGMA library is the OpenCL equivalent to the MAGMA library for CUDA and the more general LAPACK and produced by the same group. The device maker AMD has 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 specific random number generation library comparable to cuRAND for CUDA, 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 ▪ clMAGMA

clMAGMA is an OpenCL implementation of the MAGMA library. It is designed for use on AMD graphics cards, where CUDA is not available. 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.

32.1.1 ▪ Simple example of a linear solve

As a beginning test case for the clMAGMA package, Example 32.1 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 32.1.

```

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 dMAGMA 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, we need to first know about the devices being used for this calculation, hence we request the list of devices on Line 27. In this simple example, we'll just create one queue, of type `magma_queue_t` and assign it to the first device received as is done on Line 31. Space for the matrix `A` and the right hand side vector `b` is created on Lines 50, 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 we must also provide a pointer to the work queue to which the task will be sent 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, to clean up after using clMAGMA, we call the `magma_finalize()` function 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 4 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, we see that utilizing the compute capability of the GPU does result in significant savings.

32.1.2 ■ Compiling and linking clMAGMA code

In order to compile and link the program listed in Example 32.1 on a Max OS X system, 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 you may notice is that we've switched from the `gcc` compiler to 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. Without modifying the header files to make the whole system C compiler friendly, using the C++ compiler instead is a quick and dirty work-around even if it does produce some warnings.

The flag `-O3` is an optimization flag, and asks for level 3 optimizations. We have not discussed compiler flagged optimizations yet, but they are worth learning and being wary of. The number following the capital-O is the level of optimization, where the higher the number the more aggressive (max 3). Setting optimization level to zero means no optimizations will be done, and is what is the default. More aggressive optimization can result in faster code, however, care should be taken when using high level optimization because it is possible for the compiler to optimize so much it breaks your code (and it will be impossible to find why). Always develop your code using `-O0` and then try out higher levels of optimization to see if you get improvement in performance and/or unexpected results.

The `-D` flag is equivalent to putting `#define` at the top of your file. These flags are the correct settings for the current version of clMAGMA, so be sure to include both flags.

The `-I` flag instructs the compiler where to look for header files besides your directory and the default `/usr/include`. In this case, we want to include headers files from both the clBLAS package and from the clMAGMA package, so we add those include directories using the `-I` flag. Where your library is located may differ, so you may need to adjust those paths.

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 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 clMAGMA and clBLAS libraries are explicitly linked along with links to the Accelerate and OpenCL frameworks provided by Apple. Check with your system administrator to find out the locations of these libraries for your particular system.

32.1.3 • CPU interface vs. GPU interface

In Example 32.1, the results of the computation were left on the host. That may or may not be what is desired. In fact, you may want the result to stay on the device rather than shifting it back and forth. For that reason, clMAGMA also has a GPU interface where the inputs and the results are kept on the GPU. However, for us to check it, we still must get the input data to the device, and read the results back off of it. The GPU interface version of Example 32.1 is shown below in Example 32.2.

Example 32.2.

```
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[])
9   *
10  A matrix A is created in column-major format
11  and a right hand side is created. The system of equations
12  is then solved storing the solution the vector b upon return.
13
14  Inputs: none
15
16  Outputs: none
17 */
18 int main(int argc, char* argv[])
19 {
20     // Initialize the MAGMA system
21     magma_init();
22
23     // Get device info, using default device 0
24     magma_int_t num;
25     magma_device_t devices[2];
26     magma_get_devices( devices, 2, &num)
27
28     // temporary data required by dgesv similar to LAPACK
29     magma_int_t *piv, info;
30
31     // the matrix A is m x m.
32     magma_int_t m = 9;
33
34     // the right hand side has dimension m x n
35     magma_int_t n = 1;
36
37     // error code for MAGMA functions
38     magma_int_t err;
39
40     // The matrix A and right hand side b
41     double* A;
42     double* b;
43     magmaDouble_ptr dev_A;
44     magmaDouble_ptr dev_b;
45
46     // Allocate matrices on the host
47     err = magma_dmalloc_cpu(&A, m*m);
48     err = magma_dmalloc_cpu(&b, m);
49
50     // Allocate matrices on the device
51     err = magma_dmalloc(&dev_A, m*m);
52     err = magma_dmalloc(&dev_b, m);
53     // Note: should check that err == 0, meaning call was successful
54
55     // Create temporary storage for the pivots.
56     piv = (magma_int_t*) malloc(m*sizeof(magma_int_t));
57
58     // Generate matrix A
59     double row[9] = {3., 21., 4., 1., 13., 1., 7., 13., 1.};
60     int i, j, index = 0;
61     for (j=0; j<9; ++j)
62         for (i=0; i<9; ++i)
63             A[index++] = (i > j ? -1 : 1) * row[j];
64
65     // Generate b
66     b[0] = 17;
67     for (i=0; i<4; ++i) {
68         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(n, n, b, m, dev_b, 0, m, queue);
80
81 // MAGMA solve
82 magma_dgesv_gpu(m, n, dev_A, 0, m, piv, dev_b, 0, m, queue, &info);
83
84 // copy solution in dev_b back onto host
85 magma_dgetmatrix(m, n, dev_b, 0, m, b, m, queue);
86
87 free(A);
88 free(b);
89 magma_free(dev_A);
90 magma_free(dev_b);
91 free(piv);
92 magma_finalize();
93 return 0;
94 }
```

In order to take advantage of the GPU interface, we need to put the matrix **A** and the right hand side **b** onto the device before calling the solver. The data for the matrices are first set on the CPU in the same way that it was done in Example 32.1. However, we will also need to allocate space for the matrices on the device as well, and for that we use the clMAGMA data type `magmaDouble_ptr` as shown on Line 44. The space is allocated and assigned to the corresponding pointers on Lines 52, 53. Once the data is set up on the host side, the data is transferred into the device memory using the function `magma_dsetmatrix` as shown on Lines 78, 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. Because we want to store the data directly on the memory we allocated, i.e. without any offset, there is a zero in the sixth argument of `magma_dsetmatrix` on Lines 78, 79.

The linear solver can now be called, but this time the function is `magma_dgesv_gpu` as shown on Line 82, where the other change of note is that device memory is used as input rather than host memory. Again, because we are specifying device memory, the function also requests offsets, hence the zeros in the fourth and eighth arguments of `magma_dgesv_gpu`. 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 85.

The last difference between Example 32.2 and Example 32.1 is that the device memory allocated uses the `magma_free` function on Line 89 to release the memory on the device.

32.2 ■ 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 32.3 shows a simple code to compute a 1-dimensional FFT in double precision:

Example 32.3.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4
5 #include <clFFT.h>
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 = clCreateCommandQueue( context, device_id, 0, &err );
46
47     // Set up the initial data for a single wave number K
48     double dx = 2.*M_PI/N;
49     dcomplex* x = (dcomplex*) malloc(N*sizeof(dcomplex));
50     int i;
51     for (i=0; i<N; ++i) {

```

```

52     x[ i ].r = cos (K*dx*i);
53     x[ i ].c = sin (K*dx*i);
54 }
55
56 // Set up memory on the device for the calculation
57 cl_mem dev_x = clCreateBuffer( context , CL_MEM_READ_WRITE,
58                               N*sizeof(dcomplex) , NULL, &err );
59
60 // Copy the input data into device memory
61 err = clEnqueueWriteBuffer( queue , dev_x , CL_TRUE, 0,
62                           N*sizeof(dcomplex) , x , 0, NULL, NULL );
63
64 // Setup clFFT
65 clfftSetupData fftSetup;
66 err = clfftInitSetupData(&fftSetup);
67 err = clfftSetup(&fftSetup);
68
69 // Create a default plan for a complex 1D FFT and set options
70 clfftPlanHandle planHandle;
71 err = clfftCreateDefaultPlan(&planHandle , context , CLFFT_1D, &N);
72 // Will use double precision
73 err = clfftSetPlanPrecision(planHandle , CLFFT_DOUBLE);
74 // Will use interleaved values for complex arrays
75 err = clfftSetLayout(planHandle , CLFFT_COMPLEX_INTERLEAVED,
76                      CLFFT_COMPLEX_INTERLEAVED);
77 // Results will be put in place of the input buffer
78 err = clfftSetResultLocation(planHandle , CLFFT_INPLACE);
79
80 // Construct the plan before executing
81 err = clfftBakePlan(planHandle , 1, &queue , NULL, NULL);
82
83 // Compute the forward Fourier transform
84 err = clfftEnqueueTransform(planHandle , CLFFT_FORWARD, 1, &queue , 0,
85                            NULL, NULL, &dev_x , NULL, NULL);
86
87 // Wait for the computation to finish before extracting the results
88 err = clFinish(queue);
89
90 // Copy the results from the device back to the host
91 err = clEnqueueReadBuffer( queue , dev_x , CL_TRUE, 0,
92                           N*sizeof(dcomplex) , x , 0, NULL, NULL );
93
94 // Print the results
95 for (i=0; i<N; ++i)
96     printf("(%.f, %.f) ", x[ i ].r, x[ i ].c);
97     printf("\n");
98
99 // Compute the backward Fourier transform
100 err = clfftEnqueueTransform(planHandle , CLFFT_BACKWARD, 1, &queue , 0,
101                            NULL, NULL, &dev_x , NULL, NULL);
102
103 // Wait for the computation to finish before extracting the results
104 err = clFinish(queue);
105
106 // Copy the results from the device back to the host
107 err = clEnqueueReadBuffer( queue , dev_x , CL_TRUE, 0,
108                           N*sizeof(dcomplex) , x , 0, NULL, NULL );
109
110 // Print the results
111 for (i=0; i<N; ++i)
112     printf("(%.f, %.f) ", x[ i ].r, x[ i ].c);
113     printf("\n");
114

```

```

115 // Free device memory
116 clReleaseMemObject( dev_x );
117
118 free(x);
119
120 // Destroy the plan
121 err = clfftDestroyPlan( &planHandle );
122
123 // Free internal clFFT memory created in setup
124 clfftTeardown( );
125
126 // Clean up remaining OpenCL objects
127 clReleaseCommandQueue( queue );
128 clReleaseContext( context );
129
130 return 0;
131 }
```

Much of the beginning part of Example 32.3 is what we have encountered already such as getting device and context information and setting up a command queue. The clFFT package doesn't supply its own complex variable type because it isn't strictly necessary, but for this example, we use a `typedef` command to create a new complex number type `dcomplex` on Line 10. We will use the `r` and `c` components of the `dcomplex` type to contain the real and complex imaginary parts of our data. By doing this, if we create an array of this data type, as is done on Line 49, the data 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

We could just as easily created a simple double precision array and stored the values in this manner, but this can be easier to track for indexing purposes. The FFT will only operate on data on the device, so we next create a memory buffer on the device and copy the data over on Lines 57, 61.

Before actually doing any transforms, the clFFT package needs to be initialized, which is a two-step process on Lines 66, 67. The next step is to create a plan for doing the FFT. On Line 71, a plan is created for a one-dimensional, complex-valued transform for a data vector of length N . 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 non-negative integers m, n, p, q . The most efficient transforms are when $N = 2^m$. On Line 73, we specify that our plan will use double precision as opposed to the default single precision. On Line 75, we specify that the data will be organized in an interleaved format for both input and output as described above. Finally, we specify that 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 that is where the output of the transform will be stored. Once the non-default options are set, the kernel functions are built internally when completing

the plan by “baking” it on Line 81. The plan is now ready to compute transforms.

The forward transform is computed on Line 84 and the backward transform is computed on Line 100. We must wait for the transform to be completed on Line 88, before retrieving the results on Line 91. For this package, the forward transform results in the data being organized so that the wave numbers appear in the following order:

Na_0
Na_1
\vdots
$Na_{N/2}$
$Na_{-N/2+1}$
\vdots
Na_{-2}
Na_{-1}

where the a_k are the Fourier coefficients. 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 121. The clFFT framework that was set up on Line 66 is dismantled by a call to `clfftTeardown` on Line 124.

32.2.1 ▪ Compiling and linking with clFFT library

When building code to use the clFFT library, the additional library `-lclFFT` must be added.

32.3 ▪ Random123

The OpenCL package does not contain a native random number generator, so we must turn elsewhere to generate a sequence of random values. The trick is to write 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 the Example 32.4, a list of double precision uniformly distributed random numbers are generated in the range $[0, 1]$ using the ThreeFry4x32 algorithm provided in the library. The generators in this library produce random integers, so we will have to convert them to double precision.

As has been discussed in other parallel implementations of random number generators, care must be taken to make sure that when random numbers are generated in parallel, we don’t end up with the same sequence of numbers being generated by each processor. The Random123 library handles this well, and we can take advantage of the fact that the seed can be modified by the process ID to make it unique for each process.

Example 32.4 gives an example of generating a sequence of random values.

Example 32.4.**File: get_rand.ocl**

```

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 (tid < 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 us into the [0,1) range
36     double factor = 1.0/(LONG_MAX+1.0);
37     vals[tid] = 0.5+j*factor/2.0;
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     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 = clCreateCommandQueue(context, device_id, 0, &err);
37
38     // Load the kernel source code
39     char* srccode = GetKernelSource("get_rand.ocl");
40     cl_program program = clCreateProgramWithSource(context, 1,
41                                                 (const char**)&srccode, NULL, &err);
42
43     // Compile the kernel code
44     err = clBuildProgram(program, 0, NULL, "-I/usr/local/include", NULL,
45                          NULL);
46
47     // Create the kernel function from the compiled OpenCL code
48     cl_kernel kernel = clCreateKernel(program, "get_rand", &err);
49
50     // Allocate memory to store the results
51     double *u = (double*)malloc(N*sizeof(double));
52
53     // Allocate memory on the device
54     cl_mem dev_u = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
55                                    N*sizeof(double), NULL, NULL);
56
57     // Define the arguments for the kernel function
58     err = clSetKernelArg(kernel, 0, sizeof(cl_uint), &N);
59     err = clSetKernelArg(kernel, 1, sizeof(cl_mem), &dev_u);
60
61     // Execute the kernel
62     err = clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &N, NULL, 0, NULL
63                                 ,
64                                 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]);

```

```
76 // Free the resources
77 clReleaseMemObject (dev_u);
78 clReleaseKernel (kernel);
79 clReleaseProgram (program);
80 clReleaseCommandQueue (queue);
81 clReleaseContext (context);
82 free (u);
83 free (srcode);
84
85 return 0;
86 }
87 }
```

The components of the main program here have all been encountered before, so the break down of this example focuses on the kernel function defined in the file `get_rand.ocl`. The Random123 functions are all in header files, there is no compiled code, which is why this will work when using OpenCL. Since we will be using the ThreeFry4x32 generator, then we must include the appropriate header file as done 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 44. In this case, the Random123 directory of header files is located in the directory `/usr/local/include`, so we must specify that path in the fourth argument of `clBuildProgram`. Other flags can also be included in that argument such as other include paths or defined constants. For example, suppose we wanted to define the constant `M_PI` within the kernel from the main program, then the fourth argument of `clBuildProgram` will become:

```
"-I/usr/local/include -DM_PI=3.14159"
```

Next, the ThreeFry4x32 generator requires both a key and a counter, which are set up on Lines 21, 22. The first entry of the key is important in order to make sure that every work item has a unique seed so that we don't get the same set of random values from each work item. Thus, we have put the work item ID as the first argument of the key. 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 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. Fortunately, the Random123 library is robust under both conditions, which is important for high-fidelity computations.

Exercises

- 32.1. Modify Example 32.1 to solve a tridiagonal system of equations using the GPU interface.
- 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 compute the first derivative of the data in an array of length N following the discussion in Section 37.2.

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 the random number generation using the library Random123 relies on individual work-items to generate the random values, the strategy for doing Monte Carlo integration will be different for OpenCL than would be done with CUDA. In this case, 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 as a function of N . 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-

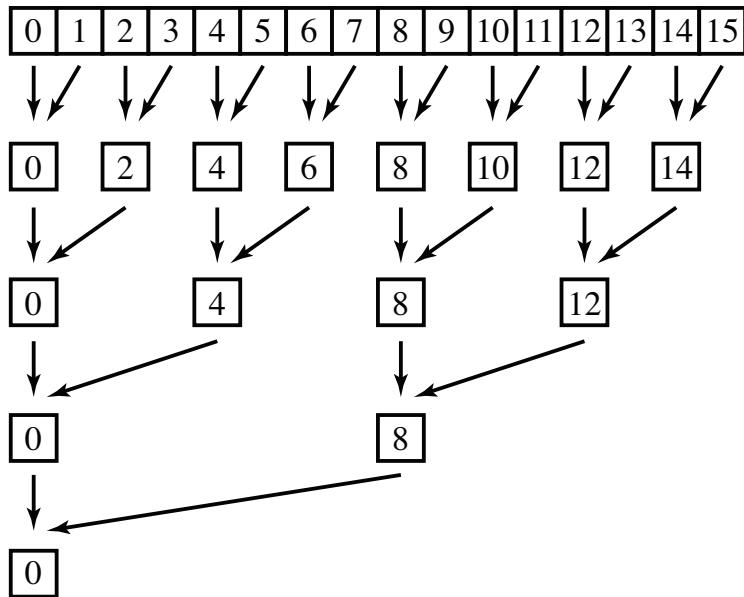


Figure 33.1. Illustration of how a single kernel is used to do pairwise sums and storing the results in place in order to sum the total of the array. The whole array is summed in $\log_2 N$ operations. The numbers indicate which array entries contain the current subtotals for the boxes to their right.

items as possible in each workgroup. Since you are only tallying the final value, everything should be done in local memory except for storing the final state 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 σ should be given on the command line. Use $\alpha = 1$, which should be defined in your program. Compute the time required to complete the calculation as a function of NM and plot the results. Plot an estimate for $p(t)$ for $0 \leq t \leq T = 10$.

33.2 ▪ Finite Difference Methods

You will need multiple kernels to employ the ADI method. You will need two kernels for the explicit steps, one for the x direction and one for the y direction because the indexing in the data is different. These should be fairly simple adaptations of the `diff` kernel in the examples. You will also need a kernel to do a matrix transpose. The implicit steps should use the `clMAGMA` library with the tridiagonal solver. One important point to remember is that the matrix that has to be inverted does not change with time, so that means you should use the `magma_dgetrf_gpu` once to factor the matrix, and thereafter use `magma_dgetrs_gpu` to get the solutions, all 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 only done once, but if the domain is not square or the boundary conditions vary between the directions, then you will need to do the factorization 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.

33.2.1 ■ Brusselator Reaction

Program the assignment in Section 35.3.1 using an $N \times N$ grid. Measure the time required to complete the calculation as a function of N .

33.2.2 ■ Linearized Euler Equations

Program the assignment in Section 35.3.2 using an $N \times N$ grid. Measure the time required to complete the calculation as a function of N .

33.3 ■ Elliptic Equations and SOR

For maximal efficiency, you should write two kernels, one for the red points and one for the black using the red/black ordering scheme. You should be storing 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 33.1.1.

33.3.1 ■ Diffusion with Sinks and Sources

Program the assignment in Section 36.1.1 using an $(2N - 1) \times N$ grid. Experiment to find the optimal ω that requires the fewest iterations to reach a tolerance of 10^{-9} . Compute the time required to complete a single pass through the grid. Repeat these steps for a grid of dimensions $(4N - 1) \times 2N$. Does the optimal value of ω 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, 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 $N \times N$. Experiment to find the optimal ω 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 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 $N \times N \times N$. Experiment to find the optimal ω 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 ■ Pseudo-Spectral Methods

The Fourier transforms will be handled using the clFFT library keeping data on the device. You will need to write a separate kernel to perform the pseudo-spectral derivative operations.

33.4.1 • Complex Ginsburg-Landau Equation

Because this is a complex valued equation, you will want to use a storage layout that uses complex values, hence use CLFFT_HERMITIAN_INTERLEAVED for both the input and output layout.

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. Compute the time required to execute your code as a function of N .

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. For the forward transform you will want to use CLFFT_REAL for the input layout and CLFFT_HERMITIAN_INTERLEAVED for the output layout. The roles are reversed when doing the backward transform. 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 transform will inevitably lead to small, but non-zero 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. Compute the time required to execute the solution to the indicated terminal time. Plot the results and verify that phase separation is occurring. Compute the time required to execute your code as a function of N .

Part VI

Applications

In this part, we provide some basic representative computational problems to be solved using the techniques presented in this text. Here we present the basic mathematical statements, specific implementation instructions are contained in the problem sets in the 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 within the book.

Each of the projects give a list of input parameters to be used in that particular application. You should assume those parameters are 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 param1 = atoi(argv[+ + argi]);
6     float param2 = atof(argv[+ + argi]);
7     long int param3 = atol(argv[+ + argi]);
8
9     // argi now contains number of arguments read
10    // ... rest of program ...

```

By using this strategy, if you decide that you no longer want to read `param2` from the parameter list, you can simply delete that line without having to renumber all your argument numbers.

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. We present here just the bare essentials necessary to understand the computational challenge.

Chapter 34

Stochastic Differential Equations

One of the easier types of equations to parallelize are the solutions 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. Because 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, but 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. If you are not familiar with stochastic differential equations, then it may be helpful to consider the case where $b(X_t, t) \equiv 0$ so that we are left with 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. We can't write dW_t/dt because the Wiener process is not differentiable anywhere, hence we use the differential form as shown in Equation 34.1. In this context, we refer to X_t as 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 $E(W_t) = 0$. 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 a random variable with

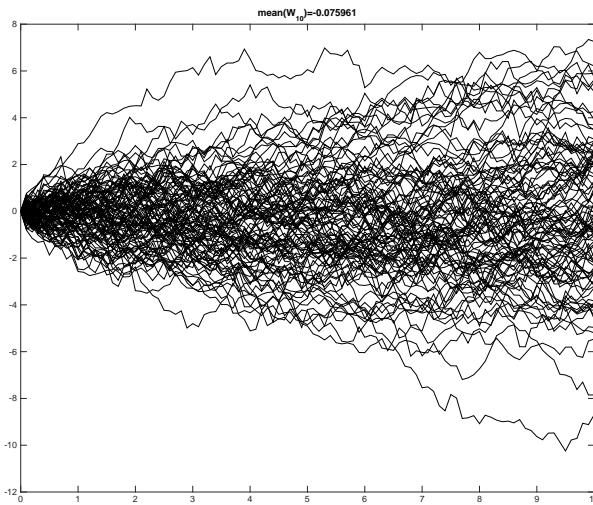


Figure 34.1. Samples paths for 100 realizations of a Wiener process, i.e. solutions for Equation (34.5) on the interval $0 \leq t \leq 10$. By definition of a Wiener process, the expected value of X_{10} is $E(X_{10}) = 0$. In this example, where X_t^k is the k^{th} sample path, the computed mean is $\mu(X_{10}^k) \approx 0.076$.

a Gaussian distribution that has no dependencies, i.e. it doesn't depend on s , t , or the values W_s , W_t , 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 you will choose 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 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 1st-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 Δt is the time step size, $t_n = n \Delta t$, and ΔW_{t_n} is a Gaussian distributed random variable with mean zero and variance Δt .

Each time step, the random variable ΔW_{t_n} must be sampled from a standard Gaussian normal distribution with mean zero and variance Δt . 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 Δt . 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 [14].

34.2.2 ■ Box-Muller and Polar Marsaglia Methods

Not all random number generator libraries provide a Gaussian distribution generator, but do provide a uniform random number generator. We present here 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.

The first method is the Box-Muller method [16]. 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

$$N_1 = \sqrt{-2 \ln U_1} \cos(2\pi U_2)$$

$$N_2 = \sqrt{-2 \ln U_1} \sin(2\pi U_2)$$

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 [13]. In this method, given two uniformly distributed random variables U_1, U_2 , first compute

$$V_1 = 2U_1 - 1$$

$$V_2 = 2U_2 - 1$$

$$W = V_1^2 + V_2^2.$$

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

$$N_1 = \sqrt{\frac{-2 \ln W}{W}} V_1$$

$$N_2 = \sqrt{\frac{-2 \ln W}{W}} V_2$$

When comparing the two methods, consider the comparable costs. To generate two variables in the Box-Muller method, we use one log, one square root, and two trig functions. 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. 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 where we use 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

$$P \left\{ \left(\frac{1}{N} \sum_{k=1}^N p(x_k) - \langle p(x) \rangle \right)^2 \geq \frac{\delta^{-1}}{N} \text{var}\{p(x)\} \right\} \leq \delta \quad (34.9)$$

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. Suppose we want to be 99% confident that our estimate meets a certain tolerance, then we 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, if we want our error to meet a specified tolerance with 99% confidence, we can now estimate the number of samples required 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 $[0, 1]$, and x_k are samples from that generator, then

$$\int_{-\infty}^{\infty} p(x)f(x)dx = \int_0^1 f(x)dx \approx \frac{1}{N} \sum_{k=1}^N f(x_k) \quad (34.13)$$

To determine the number of samples required, we need the variance $\text{var}\{p(x)\} = \frac{1}{12}$. So for an error tolerance of 10^{-2} and a 99% confidence of meeting that tolerance, we will require

$$N \geq \frac{100 \text{var}\{p(x)\}}{(\text{tolerance})^2} = \frac{\frac{100}{12}}{(10^{-2})^2} \approx 83,333 \text{ samples.} \quad (34.14)$$

One thing you should 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} . When you take the sum of N numbers in double precision that are approximately $O(1)$, then 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 may be useful, it will give you 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), we should take $f(x) = \frac{1}{\lambda} \cos x$.

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 λ , and an optional initial seed value s . You will need to check the value of `argc` to determine if s was given. Your program should print the value of the seed used for the computation. 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

$$\int_0^{\infty} e^{-\lambda x} \cos(x) dx = \frac{\lambda}{1 + \lambda^2}.$$

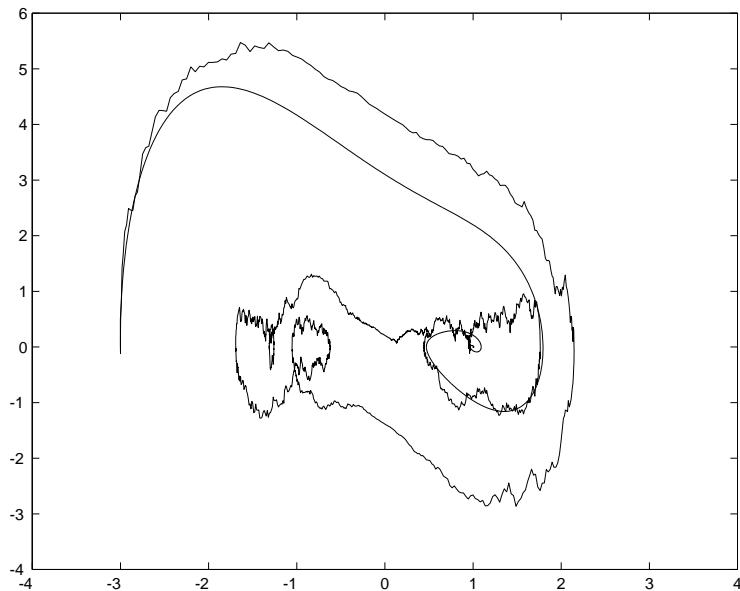


Figure 34.2. Sample phase plane plots for the cases of $\sigma = 0$ and $\sigma = 0.5$ for the Duffing-Van der Pol Oscillator with multiplicative noise forcing, Equations (34.18), (34.19), and using $\alpha = 1$.

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 non-linear spring coefficient. If we add a multiplicative noise forcing term, the equation becomes

$$x'' + x' - (\alpha - x^2)x = \sigma x\xi$$

where ξ is white noise. Converting this to a first order system of stochastic differential equations, we get

$$dX_t = Y_t dt \quad (34.18)$$

$$dY_t = ((\alpha^2 - X_t^2)X_t - Y_t)dt + \sigma X_t dW_t \quad (34.19)$$

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

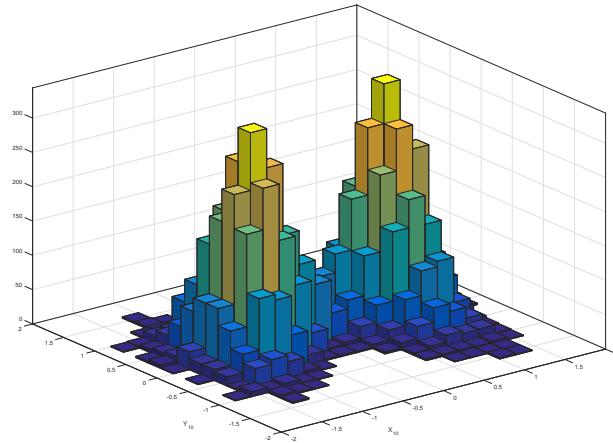


Figure 34.3. Example of a histogram for the Duffing-Van der Pol oscillator at terminal time $T = 10$ with initial condition $(X_0, Y_0) = (0.1 N(0; 1), 0)$, and with parameter values $\alpha = 1$, $\sigma = 0.5$. The histogram is generated by 10,000 sample runs solving Eqns (34.18), (34.19).

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. The first four arguments in order are the parameters α , σ , the number of time steps M , and the number of trials, N . 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.

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, you must count for each time value $t = k/10$, 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.

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 partial differential equations. One very common strategy is the method of finite differences. It is well beyond the scope of this book to go into depth in the significant theory behind the method, the interested reader is encouraged to explore the reference [15]. Here we present a bare-bones discussion that is enough to introduce the subject and program a parallel solver.

In this chapter we will focus exclusively on solving time-dependent partial differential equations, i.e. parabolic and hyperbolic partial differential equations. While some of the discussion in this chapter will be relevant to solving elliptic equations, the solution methods for elliptic equations are distinct enough for us to put that 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, $\Delta x = 1/J$. We can approximate the derivative $f'(x_j)$ from the discrete values in several ways, the simplest of which are:

$$\frac{y_{j+1} - y_j}{\Delta x} = \frac{f(x_{j+1}) - f(x_j)}{x_{j+1} - x_j} = f'(x_j) + \frac{\Delta x}{2} f''(x_j) + O(\Delta x^2) \quad (35.1)$$

$$\frac{y_{j+1} - y_{j-1}}{2\Delta x} = \frac{f(x_{j+1}) - f(x_{j-1})}{x_{j+1} - x_{j-1}} = f'(x_j) + \frac{\Delta x^2}{3} f'''(x_j) + O(\Delta x^2) \quad (35.2)$$

$$\frac{y_j - y_{j-1}}{\Delta x} = \frac{f(x_j) - f(x_{j-1})}{x_j - x_{j-1}} = f'(x_j) + \frac{\Delta x}{2} f''(x_j) + O(\Delta x^2) \quad (35.3)$$

where here we have assumed that the original function $f(x)$ is smooth enough to have multiple derivatives. These are called the forward difference, central difference, and backward difference respectively. We see that each formula approximates the first derivative of $f(x)$ at the specified grid point, x_j , along with an estimate for the error

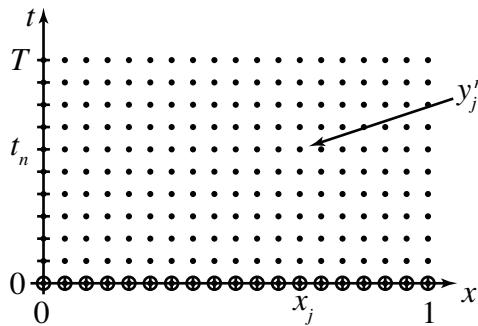


Figure 35.1. Illustration of the grid on which we solve a partial differential equation in the domain $0 \leq x \leq 1$, $0 \leq t \leq T$. The initial data for the differential equation means the data y_j^0 in the circled grid points are given by $y_j^0 = g(x_j)$, where $f(x, 0) = g(x)$ is the prescribed initial condition. The goal is to compute the remaining y_j^n for $n > 0$.

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 Δx , while the central difference formula is second-order. It is not always the case that second order is 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 partial differential equation, we need to solve the equation in the domain space + time. Thus, if we wish to solve the heat equation

$$\frac{\partial f}{\partial t} = \alpha \frac{\partial^2 f}{\partial x^2}, \quad 0 \leq x \leq 1, \quad 0 \leq t \leq T, \quad (35.5)$$

we will also need to discretize the time domain. Thus, 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 we must solve the equation numerically is illustrated in Figure 35.1. A well-posed problem will require an initial value for time $t = 0$, which we will state as $f(x, 0) = g(x)$ for some 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, we must now compute the value of y_j^n for all $0 \leq j \leq J$, $0 < n \leq N$. 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 partial differential equations, we know that for the problem to be well posed, we also need boundary conditions at $x = 0, 1$, which we have not specified.

Suppose we have the boundary conditions

$$f(0, t) = g(t), \quad (35.8)$$

$$\frac{\partial f}{\partial x}(1, t) = h(t). \quad (35.9)$$

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, we simply write $y_0^n = g(t_n)$ instead of Equation (35.7) when $j = 0$. The Neumann condition at the other end requires a little more thought. There are multiple ways to hand it, but one way is to use a *ghost point*. Suppose y_{J+1}^n existed, then we could approximate the boundary condition as

$$\frac{y_{J+1}^n - y_J^n}{\Delta x} = h(t_n) \quad (35.10)$$

Solving for y_{J+1}^n and substituting into Equation (35.7), we get a modified update equation for y_j^{n+1} :

$$\begin{aligned} y_J^{n+1} &= y_J^n + \frac{\alpha \Delta t}{\Delta x^2} (y_{J+1}^n - 2y_J^n + y_{J-1}^n) \\ &= y_J^n + \frac{\alpha \Delta t}{\Delta x^2} ((y_J^n + \Delta x h(t_n)) - 2y_J^n + y_{J-1}^n) \end{aligned} \quad (35.11)$$

We now can compute y_j^{n+1} for $j = 0, \dots, J$. Repeating this process for $n = 1, \dots, N$ means that we have solved for all y_j^n in the domain.

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 $0 \leq j \leq J$ are 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.

35.2.3 • 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 α 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 that we chose to use the forward finite difference to approximate the time derivative in Equation (35.7). Suppose we used the backward finite difference instead, then the equation would be

$$y_j^n = y_j^{n-1} + \frac{\alpha \Delta t}{\Delta x^2} (y_{j+1}^n - 2y_j^n + y_{j-1}^n). \quad (35.12)$$

The difficulty here is that we can't solve for y_j^n by itself because y_{j+1}^n , y_{j-1}^n also appear in this equation. Bringing all the n^{th} terms to the left side, we get the equation

$$-\frac{\alpha \Delta t}{\Delta x^2} y_{j+1}^n + \left(1 + 2\frac{\alpha \Delta t}{\Delta x^2}\right) y_j^n - \frac{\alpha \Delta t}{\Delta x^2} y_{j-1}^n = y_j^{n-1}. \quad (35.13)$$

Thus, in order to advance from time t_{n-1} to t_n , we must solve a system of linear equations:

$$\begin{aligned} y_0^n &= g(t_n), \\ -\frac{\alpha \Delta t}{\Delta x^2} y_2^n + \left(1 + 2\frac{\alpha \Delta t}{\Delta x^2}\right) y_1^n - \frac{\alpha \Delta t}{\Delta x^2} y_0^n &= y_1^{n-1}, \\ &\vdots \end{aligned} \quad (35.14)$$

$$-\frac{\alpha \Delta t}{\Delta x^2} y_{j+1}^n + \left(1 + 2\frac{\alpha \Delta t}{\Delta x^2}\right) y_j^n - \frac{\alpha \Delta t}{\Delta x^2} y_{j-1}^n = y_j^{n-1}, \quad (35.15)$$

$$\vdots \quad (35.16)$$

$$\begin{aligned} -\frac{\alpha \Delta t}{\Delta x^2} y_J^n + \left(1 + 2\frac{\alpha \Delta t}{\Delta x^2}\right) y_{J-1}^n - \frac{\alpha \Delta t}{\Delta x^2} y_{J-2}^n &= y_{J-1}^{n-1}, \\ \left(1 + \frac{\alpha \Delta t}{\Delta x^2}\right) y_J^n - \frac{\alpha \Delta t}{\Delta x^2} y_{J-1}^n &= y_J^{n-1} - \frac{\alpha \Delta t}{\Delta x^2} b(t_n), \end{aligned}$$

where we have incorporated the boundary conditions described earlier. This results in a tri-diagonal system of equations that looks like

$$\left[\begin{array}{cccccc} 1 & 0 & & & & \\ -\lambda & 1+2\lambda & -\lambda & 0 & & \\ 0 & -\lambda & 1+2\lambda & -\lambda & 0 & \\ & \ddots & \ddots & \ddots & & \\ & 0 & -\lambda & 1+2\lambda & -\lambda & \\ & & 0 & -\lambda & 1+\lambda & \end{array} \right] \begin{bmatrix} y_0^n \\ y_1^n \\ y_2^n \\ \vdots \\ y_{J-1}^n \\ y_J^n \end{bmatrix} = \begin{bmatrix} g(t_n) \\ y_1^{n-1} \\ y_2^{n-1} \\ \vdots \\ y_{J-1}^{n-1} \\ y_J^{n-1} - \lambda b(t_n) \end{bmatrix} \quad (35.17)$$

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 we can write the explicit and implicit methods as simpler matrix equations of the form

$$\mathbf{Y}^{n+1} = (\mathbf{I} + \Delta t \mathbf{D}) \mathbf{Y}^n$$

$$(\mathbf{I} - \Delta t \mathbf{D}) \mathbf{Y}^{n+1} = \mathbf{Y}^n,$$

where $\mathbf{Y}^n = [y_0^n, \dots, y_J^n]^T$ and where

$$\mathbf{D} = \begin{bmatrix} * & * & & & \\ \frac{\alpha}{\Delta x^2} & -\frac{2\alpha}{\Delta x^2} & \frac{\alpha}{\Delta x^2} & 0 & \\ 0 & \frac{\alpha}{\Delta x^2} & -\frac{2\alpha}{\Delta x^2} & \frac{\alpha}{\Delta x^2} & 0 \\ & & \ddots & \ddots & \ddots \\ & & 0 & \frac{\alpha}{\Delta x^2} & -\frac{2\alpha}{\Delta x^2} & \frac{\alpha}{\Delta x^2} \\ & & & & * & * \end{bmatrix} \quad (35.18)$$

The * in the matrix indicates that the boundary conditions must be accounted for when constructing the first and last rows of the matrix.

35.2.4 • 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 [12]. In two dimensions, the heat equation with a reaction term is given by

$$\frac{\partial f}{\partial t} = \alpha \nabla^2 f + \rho = \alpha \left(\frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} \right) + \rho \quad (35.19)$$

where ρ is a possibly non-linear reaction term that can depend on x , y , and $f(x, y)$.

To solve this equation numerically, we must first remember that the numerical solution will now be solved on a two-dimensional grid. This means that we must solve for the grid points $u_{j,k}^n$ which is an approximation to the solution of Equation (35.19):

$$u_{j,k}^n \approx f(x_j, y_k, t_n).$$

An explicit method as discussed in Section 35.2.1 could be used, but it would be slow. a better method is to use the ADI method. The method is easily described in terms of the one-dimensional operators given above.

For this algorithm, define

$$\mathbf{U}_{*,k}^n = \begin{bmatrix} u_{0,k}^n \\ u_{1,k}^n \\ u_{2,k}^n \\ \vdots \\ u_{J-1,k}^n \\ u_{J,k}^n \end{bmatrix}, \quad \mathbf{U}_{j,*}^n = \begin{bmatrix} u_{j,0}^n \\ u_{j,1}^n \\ u_{j,2}^n \\ \vdots \\ u_{j,K-1}^n \\ u_{j,K}^n \end{bmatrix} \quad (35.20)$$

The ADI method is then

1. Apply the 1D explicit operator in the x direction for a time step of $\Delta t/2$ and for each row $0 \leq k \leq K$:

$$\mathbf{U}_{*,k}^{n+1/4} = \left(\mathbf{I} + \frac{\Delta t}{2} \mathbf{D}_x \right) \mathbf{U}_{*,k}^n + \frac{\Delta t}{2} \rho_{*,k}^n. \quad (35.21)$$

2. Apply the 1D implicit operator in the y direction for a time step of $\Delta t/2$ and for each column $0 \leq j \leq J$:

$$\left(\mathbf{I} - \frac{\Delta t}{2} \mathbf{D}_y \right) \mathbf{U}_{j,*}^{n+1/2} = \mathbf{U}_{j,*}^{n+1/4}. \quad (35.22)$$

3. Apply the 1D explicit operator in the y direction for a time step of $\Delta t/2$ and for each column $0 \leq j \leq J$:

$$\mathbf{U}_{j,*}^{n+3/4} = \left(\mathbf{I} + \frac{\Delta t}{2} \mathbf{D}_y \right) \mathbf{U}_{j,*}^{n+1/2} + \frac{\Delta t}{2} \rho_{j,*}^{n+1/2}. \quad (35.23)$$

4. Apply the 1D implicit operator in the x direction for a time step of $\Delta t/2$ and for each row $0 \leq k \leq K$:

$$\left(\mathbf{I} - \frac{\Delta t}{2} \mathbf{D}_x \right) \mathbf{U}_{*,k}^{n+1} = \mathbf{U}_{*,k}^{n+3/4}. \quad (35.24)$$

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 tri-diagonal, 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.

$$\left(\mathbf{I} - \frac{\Delta t}{2} \mathbf{D}_x \right) \mathbf{U}^{n+1/3} = \left(\mathbf{I} + \frac{\Delta t}{2} \mathbf{D}_x + \Delta t \mathbf{D}_y + \Delta t \mathbf{D}_z \right) \mathbf{U}^n + \rho^n, \quad (35.25)$$

$$\left(\mathbf{I} - \frac{\Delta t}{2} \mathbf{D}_y \right) \mathbf{U}^{n+2/3} = \mathbf{U}^{n+1/3} - \frac{\Delta t}{2} \mathbf{D}_y \mathbf{U}^n + \rho^{n+1/3}, \quad (35.26)$$

$$\left(\mathbf{I} - \frac{\Delta t}{2} \mathbf{D}_z \right) \mathbf{U}^{n+1} = \mathbf{U}^{n+2/3} - \frac{\Delta t}{2} \mathbf{D}_z \mathbf{U}^n + \rho^{n+2/3}. \quad (35.27)$$

Note that we have dispensed with the spatial indexing 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 $0 \leq j \leq J$, $0 \leq k \leq K$. 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 (35.25) separately and then combine them before solving the collection of JK tridiagonal solves to invert the matrix on the left. Likewise for Equations (35.26), (35.27).

35.3 ■ Problems to Solve

35.3.1 ■ 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

$$\frac{\partial U}{\partial t} = D_U \nabla^2 U + A + U^2 V - (B + 1)U, \quad (35.28)$$

$$\frac{\partial V}{\partial t} = D_V \nabla^2 V + BU - U^2 V, \quad (35.29)$$

$$U(\mathbf{x}, 0) = A + \sigma, \quad (35.30)$$

$$V(\mathbf{x}, 0) = B + \sigma, \quad (35.31)$$

$$\left. \frac{\partial U}{\partial t} \right|_{\Gamma} = 0, \quad (35.32)$$

$$\left. \frac{\partial V}{\partial t} \right|_{\Gamma} = 0, \quad (35.33)$$

$$(35.34)$$

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 σ 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 to the initial value for all time.

Assignment: Use the ADI method to solve this system of equations 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. The arguments in order are the number of grid points in each dimension, N , the double precision diffusion coefficients, D_u , D_v , the double precision coefficients, A , B , and the number of time steps, M . If a seventh argument is specified, then it is a long integer seed value for generating the initial random values. 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 contains the data at the time points $t = 100k$, for $k = 0, \dots, 10$.

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 you should start to see a pattern of reaction waves that will be roughly equidistant such as is shown in Figure 35.2

35.3.2 ■ 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 small disturbances in air, for example, are given by

$$\frac{\partial \rho}{\partial t} = -\mathbf{u}_0 \cdot \nabla \rho - \rho_0 \nabla \cdot \mathbf{u}, \quad (35.35)$$



Figure 35.2. Sample contour plot for the Brusselator model.

$$\frac{\partial \mathbf{u}}{\partial t} = -\mathbf{u}_0 \cdot \mathbf{u} - \frac{1}{\rho_0} \nabla p, \quad (35.36)$$

$$\frac{\partial p}{\partial t} = -\mathbf{u}_0 \cdot \nabla p + \gamma p_0 \nabla \cdot \mathbf{u}, \quad (35.37)$$

where ρ is the fluid density, \mathbf{u} is the fluid velocity, and p is the fluid pressure. The constants ρ_0 , \mathbf{u}_0 , and p_0 are the equilibrium values around which the system is linearized. The constant γ is the gas constant. In the case of a small perturbation from a flow at rest, we can take $\rho_0 = 1$, $\mathbf{u}_0 = \mathbf{0}$, and $p_0 = 1$ so that the equations in (35.35)–(35.37) become

$$\frac{\partial \rho}{\partial t} = -\nabla \cdot \mathbf{u}, \quad (35.38)$$

$$\frac{\partial \mathbf{u}}{\partial t} = -\nabla p, \quad (35.39)$$

$$\frac{\partial p}{\partial t} = -\gamma \nabla \cdot \mathbf{u}. \quad (35.40)$$

For simplicity, assume this problem is periodic on a domain that has dimensions $[-1, 1]$ in all directions.

This problem can be effectively solved by using ADI where the spatial derivative operator is replaced with a central difference operator. Let \mathbf{I} be the identity matrix, $\mathbf{0}$

be a matrix of all zeros, and define the matrix

$$\mathbf{D} = \begin{bmatrix} 0 & \frac{\Delta t}{4\Delta x} & 0 & \cdots & \cdots & 0 & -\frac{\Delta t}{4\Delta x} \\ -\frac{\Delta t}{4\Delta x} & 0 & \frac{\Delta t}{4\Delta x} & 0 & \cdots & \cdots & 0 \\ 0 & -\frac{\Delta t}{4\Delta x} & 0 & \frac{\Delta t}{4\Delta x} & 0 & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & -\frac{\Delta t}{4\Delta x} & 0 & \frac{\Delta t}{4\Delta x} & 0 \\ 0 & \cdots & \cdots & 0 & -\frac{\Delta t}{4\Delta x} & 0 & \frac{\Delta t}{4\Delta x} \\ \frac{\Delta t}{4\Delta x} & 0 & \cdots & \cdots & 0 & -\frac{\Delta t}{4\Delta x} & 0 \end{bmatrix} \quad (35.41)$$

In two dimensions, rewrite the system as the scalar differential equations in matrix form, where $\mathbf{u} = (u, v)$, to get

$$\begin{bmatrix} \rho \\ u \\ v \\ p \end{bmatrix}_t = - \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & \gamma & 0 & 0 \end{bmatrix} \begin{bmatrix} \rho \\ u \\ v \\ p \end{bmatrix}_x - \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & \gamma & 0 \end{bmatrix} \begin{bmatrix} \rho \\ u \\ v \\ p \end{bmatrix}_y \quad (35.42)$$

the ADI method can then be implemented in the following four steps:

1. Explicit in x :

$$\begin{bmatrix} \rho_{*,k}^{n+1/4} \\ \mathbf{u}_{*,k}^{n+1/4} \\ \mathbf{v}_{*,k}^{n+1/4} \\ \mathbf{p}_{*,k}^{n+1/4} \end{bmatrix} = \begin{bmatrix} \mathbf{I} & -\mathbf{D} & 0 & 0 \\ 0 & \mathbf{I} & 0 & -\mathbf{D} \\ 0 & 0 & \mathbf{I} & 0 \\ 0 & -\gamma\mathbf{D} & 0 & \mathbf{I} \end{bmatrix} \begin{bmatrix} \rho_{*,k}^n \\ \mathbf{u}_{*,k}^n \\ \mathbf{v}_{*,k}^n \\ \mathbf{p}_{*,k}^n \end{bmatrix} \quad (35.43)$$

2. Implicit in y :

$$\begin{bmatrix} \mathbf{I} & 0 & \mathbf{D} & 0 \\ 0 & \mathbf{I} & 0 & 0 \\ 0 & 0 & \mathbf{I} & \mathbf{D} \\ 0 & 0 & \gamma\mathbf{D} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \rho_{j,*}^{n+1/2} \\ \mathbf{u}_{j,*}^{n+1/2} \\ \mathbf{v}_{j,*}^{n+1/2} \\ \mathbf{p}_{j,*}^{n+1/2} \end{bmatrix} = \begin{bmatrix} \rho_{j,*}^{n+1/4} \\ \mathbf{u}_{j,*}^{n+1/4} \\ \mathbf{v}_{j,*}^{n+1/4} \\ \mathbf{p}_{j,*}^{n+1/4} \end{bmatrix} \quad (35.44)$$

3. Explicit in y :

$$\begin{bmatrix} \rho_{j,*}^{n+3/4} \\ \mathbf{u}_{j,*}^{n+3/4} \\ \mathbf{v}_{j,*}^{n+3/4} \\ \mathbf{p}_{j,*}^{n+3/4} \end{bmatrix} = \begin{bmatrix} \mathbf{I} & 0 & -\mathbf{D} & 0 \\ 0 & \mathbf{I} & 0 & 0 \\ 0 & 0 & \mathbf{I} & -\mathbf{D} \\ 0 & 0 & -\gamma\mathbf{D} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \rho_{j,*}^{n+1/2} \\ \mathbf{u}_{j,*}^{n+1/2} \\ \mathbf{v}_{j,*}^{n+1/2} \\ \mathbf{p}_{j,*}^{n+1/2} \end{bmatrix} \quad (35.45)$$

4. Implicit in x :

$$\begin{bmatrix} \mathbf{I} & \mathbf{D} & 0 & 0 \\ 0 & \mathbf{I} & 0 & \mathbf{D} \\ 0 & 0 & \mathbf{I} & 0 \\ 0 & \gamma\mathbf{D} & 0 & \mathbf{I} \end{bmatrix} \begin{bmatrix} \rho_{*,k}^{n+1} \\ \mathbf{u}_{*,k}^{n+1} \\ \mathbf{v}_{*,k}^{n+1} \\ \mathbf{p}_{*,k}^{n+1} \end{bmatrix} = \begin{bmatrix} \rho_{*,k}^{n+3/4} \\ \mathbf{u}_{*,k}^{n+3/4} \\ \mathbf{v}_{*,k}^{n+3/4} \\ \mathbf{p}_{*,k}^{n+3/4} \end{bmatrix} \quad (35.46)$$

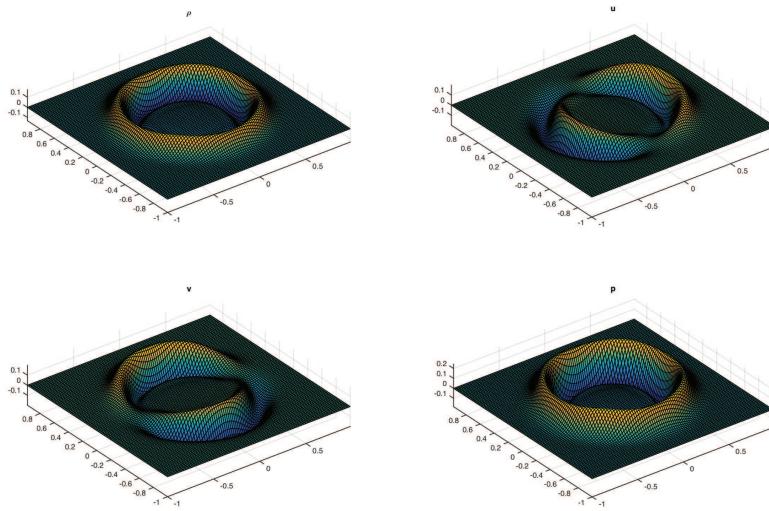


Figure 35.3. Illustration of the solution for the linearized Euler equations at the time $t = 0.5$. The initial central disturbance expands outward as a ring.

Note that in the above steps, the vectors are written as a single column vector for purposes of the computation. Thus, the vector $[\rho_{*,k}^n, \mathbf{u}_{*,k}^n, \mathbf{v}_{*,k}^n, p_{*,k}^n]^T$ is actually the vector:

$$[\rho_{0,k}^n, \rho_{1,k}^n, \dots, \rho_{N-1,k}^n, u_{0,k}^n, u_{1,k}^n, \dots, u_{N-1,k}^n, v_{0,k}^n, v_{1,k}^n, \dots, v_{N-1,k}^n, p_{0,k}^n, p_{1,k}^n, \dots, p_{N-1,k}^n]^T$$

Assignment: Use the ADI method to 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

$$\rho(x, y, 0) = \frac{2}{\gamma} e^{-100(x^2 + y^2)}, \quad (35.47)$$

$$\mathbf{u}(x, y, 0) = 0, \quad (35.48)$$

$$p(x, y, 0) = 2e^{-100(x^2 + y^2)}, \quad (35.49)$$

where $\gamma = 1.4$. Solve until the terminal time $T = 2$.

Your program should take two arguments, the dimensions of the grid, N , and the number of time steps, M . 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 ρ , 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 $k = 0, \dots, 10$.

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 35.3 shows how the solution should look when time $t = 0.5$.

Chapter 36

Iterative Solution of Elliptic Equations

In the previous chapter, we introduced finite difference grids for solving time dependent partial differential equations. In this chapter we extend that description 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 quite effective, for example direct matrix inversion and multigrid methods to name two. However, in this chapter we will look at a different method that has a more straightforward parallel implementation.

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 we have a rectangular domain of the form $0 \leq x \leq 1$, $0 \leq y \leq 1$, with spatial step sizes Δx and Δy 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 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 when we go 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 in the iteration, then we get

$$u_{j,k}^{n+1} = u_{j,k}^n + \frac{1}{4} (u_{j+1,k}^n - 2u_{j,k}^n + u_{j-1,k}^n) + \frac{1}{4} (u_{j,k+1}^n - 2u_{j,k}^n + u_{j,k-1}^n) - \frac{\Delta x^2}{4} g(x_j, y_k). \quad (36.6)$$

From this we see that 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 ω :

$$\begin{aligned} u_{j,k}^{n+1} &= u_{j,k}^n + \omega R_{j,k}^n, \\ &= (1 - \omega) u_{j,k}^n + \frac{\omega}{4} (u_{j+1,k}^n + u_{j-1,k}^n + u_{j,k+1}^n + u_{j,k-1}^n) - \frac{\omega \Delta x^2}{4} g(x_j, y_k) \end{aligned} \quad (36.8)$$

where $0 < \omega < 2$, but it still requires two grids. The Successive Over-Relaxation method (SOR) 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 ω is generally in the range $0 < \omega < 2$, where for $\omega > 1$ it is considered over-relaxation, and for $\omega < 1$ under-relaxation. The choice of ω can significantly improve the convergence rate of the method, so it is worthwhile to carefully

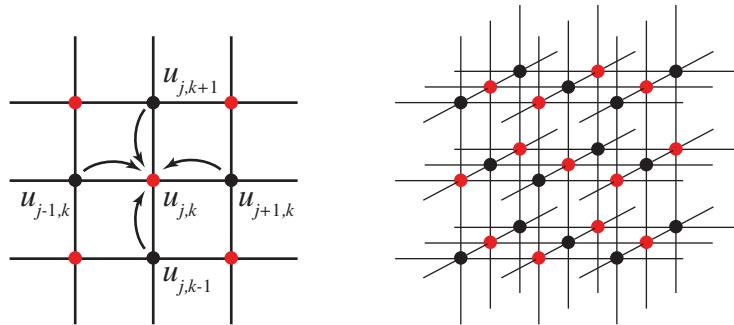


Figure 36.1. Examples of a checkerboard red/black ordering for two-dimensions (left) and three-dimensions (right). The grid points with the same color can be updated in parallel without conflict.

choose ω 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 the 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

$$\begin{aligned} \nabla^2 u &= \frac{10\lambda}{\sqrt{\pi}} e^{-\lambda^2((x-1)^2+y^2)} - \frac{10\lambda}{\sqrt{\pi}} e^{-\lambda^2((x+1)^2+y^2)}, \\ \frac{\partial u}{\partial x}(\pm 2, y) &= 0, \\ u(x, \pm 1) &= 0, \end{aligned} \quad (36.10)$$

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 λ 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 as arguments the number of grid points in the y -direction, N , the double precision parameter ω , and the convergence tolerance τ . 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

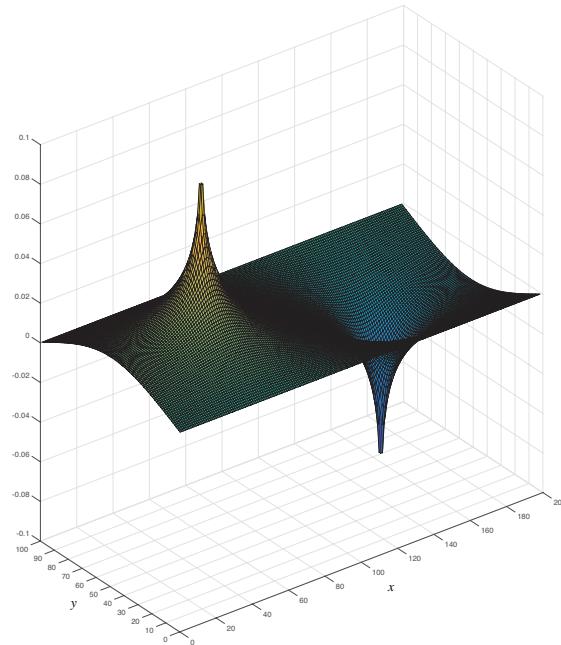


Figure 36.2. Solution for Equation (36.10) when $\lambda = 100$.

to be $M = 2N - 1$. You should also put in a failsafe of stopping the calculation should it exceed 1000 iterations. Note that the absolute value function for double precision numbers in C is the function `fabs(x)`, which is declared in the header file `math.h`.

Your program should also print to the screen the total time required to complete the calculation. Vary the values of N and ω and generate a plot that illustrates compares the time to completion divided by the number of iterations versus N .

Experiment with various values of the relaxation factor ω to determine the value that requires the fewest iterations to meet an error tolerance. Is ω still optimal if the spatial resolution is doubled?

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

$$\mu \nabla^2 \mathbf{u} = \nabla p - \mathbf{f}, \quad (36.11)$$

$$\nabla \cdot \mathbf{u} = 0, \quad (36.12)$$

where \mathbf{u} is the fluid velocity, p is the fluid pressure, $\mathbf{f} = (f, g)$ is the body force on the fluid, and μ 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 method (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, we get the following discrete versions

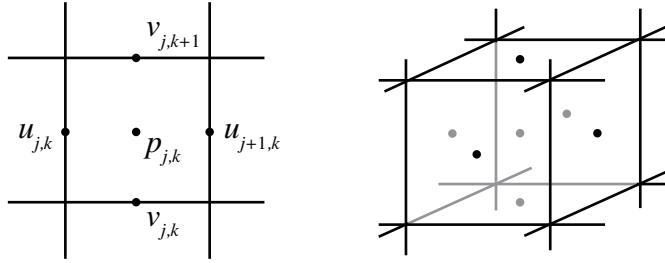


Figure 36.3. Illustration of the marker-and-cell (MAC) grid. The u -velocity grid lies on the vertical grid lines at the midpoints between horizontal grid lines. The v -velocity grid lies on the horizontal grid lines at the midpoints between vertical grid lines. The pressure p grid is in the center of 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).

of Equations (36.11), (36.12):

$$\begin{aligned} \frac{\mu}{\Delta x^2}(u_{j-1,k} - 2u_{j,k} + u_{j+1,k}) + \frac{\mu}{\Delta y^2}(u_{j,k+1} - 2u_{j,k} + u_{j,k-1}) \\ = \frac{1}{\Delta x}(p_{j+1,k} - p_{j,k}) - f(x_j, y_k + \Delta y/2), \end{aligned} \quad (36.13)$$

$$\begin{aligned} \frac{\mu}{\Delta x^2}(v_{j-1,k} - 2v_{j,k} + v_{j+1,k}) + \frac{\mu}{\Delta y^2}(v_{j,k+1} - 2v_{j,k} + v_{j,k-1}) \\ = \frac{1}{\Delta y}(p_{j+1,k} - p_{j+1,k-1}) - f(x_j + \Delta x/2, y_k) \end{aligned} \quad (36.14)$$

$$0 = \frac{1}{\Delta x}(u_{j,k} - u_{j-1,k}) + \frac{1}{\Delta y}(v_{j-1,k+1} - v_{j-1,k}) \quad (36.15)$$

To convert these to the form needed for the SOR method, we first rescale these equations and then compute the residuals for Equations (36.13), (36.14), and (36.15) as

$$\begin{aligned} r_{j,k}^u = \frac{\mu \Delta y}{\Delta x}(u_{j-1,k} - 2u_{j,k} + u_{j+1,k}) + \frac{\mu \Delta x}{\Delta y}(u_{j,k+1} - 2u_{j,k} + u_{j,k-1}) \\ - \Delta y(p_{j+1,k} - p_{j,k}) + \Delta x \Delta y f(x_j, y_k + \Delta y/2) \end{aligned} \quad (36.16)$$

$$\begin{aligned} r_{j,k}^v = \frac{\mu \Delta y}{\Delta x}(v_{j-1,k} - 2v_{j,k} + v_{j+1,k}) + \frac{\mu \Delta x}{\Delta y}(v_{j,k+1} - 2v_{j,k} + v_{j,k-1}) \\ - \Delta x(p_{j+1,k} - p_{j+1,k-1}) + \Delta x \Delta y f(x_j + \Delta x/2, y_k) \end{aligned} \quad (36.17)$$

$$r_{j,k}^p = -(u_{j,k} - u_{j-1,k}) - \frac{\Delta x}{\Delta y}(v_{j-1,k+1} - v_{j-1,k}) \quad (36.18)$$

Once the residuals are computed, we can now apply the SOR method as

$$u_{j,k}^{n+1} = u_{j,k}^n + \omega r_{j,k}^u \quad (36.19)$$

$$v_{j,k}^{n+1} = v_{j,k}^n + \omega r_{j,k}^v \quad (36.20)$$

$$p_{j,k}^{n+1} = p_{j,k}^n + \omega r_{j,k}^p \quad (36.21)$$

When solving this system using SOR, be sure to order the equations so that all the $u_{j,k}$

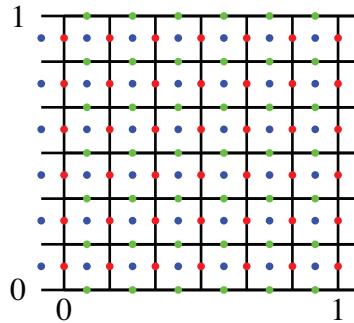


Figure 36.4. Illustration of the marker-and-cell (MAC) grid for the 2D Stokes assignment if using the ghost points for the pressure. The $u_{j,k}$ values are in red, the $v_{j,k}$ values are in green, and the $p_{j,k}$ values are in blue. If a standard grid (black lines) would be an $N \times N$ grid, then the $u_{j,k}$ grid would be a $N \times N - 1$ 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 are correct.

values are updated holding the $v_{j,k}$ and $p_{j,k}$ values fixed. Then do the same for the $v_{j,k}$ values, and then the $p_{j,k}$ values. Solving them interwoven can lead to instabilities.

Assignment: Use the SOR method to find the steady Stokes flow velocity field for a channel of width one, length one, and with a pressure drop of magnitude P between the inlet and the outlet. The system of equations to solve is

$$\mu \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) = \frac{\partial p}{\partial x}, \quad (36.22)$$

$$\mu \left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} \right) = \frac{\partial p}{\partial y}, \quad (36.23)$$

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0, \quad (36.24)$$

$$u(x, 0) = u(x, 1) = v(x, 0) = v(x, 1) = 0, \quad (36.25)$$

$$p(0, y) = P, \quad p(1, y) = 0. \quad (36.26)$$

Use a MAC grid as discussed above. Figure 36.4 illustrates the location of the grid points for u , v , and p . This means that 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 $N + 1 \times N - 1$, where the extra points on the left and right boundary are for the ghost points. For a square domain, this will ensure that the space step will be $\Delta x = \Delta y = \frac{1}{N-1}$.

Next, we should address the boundary conditions. For the top and bottom walls, where $u = v = 0$, clearly we simply set $v_{j,0} = 0$, $v_{j,N-1} = 0$. On the other hand, we 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 $u_{j,-1} = -u_{j,0}$. Similarly, $u_{j,N-1} = -u_{j,N-2}$. Rather than adding ghost points for $u_{j,-1}$, $u_{j,N}$, we can just alter the stencil for the boundary cases. For example, after making the substitution $u_{j,-1} = -u_{j,0}$, the equation for $u_{j,0}$ would be

$$\frac{\mu}{\Delta x^2} (u_{j-1,0} - 2u_{j,0} + u_{j+1,0}) + \frac{\mu}{\Delta y^2} (u_{j,1} - 2u_{j,0} - u_{j,-1}) = \frac{1}{\Delta x} (p_{j+1,0} - p_{j,0}). \quad (36.27)$$

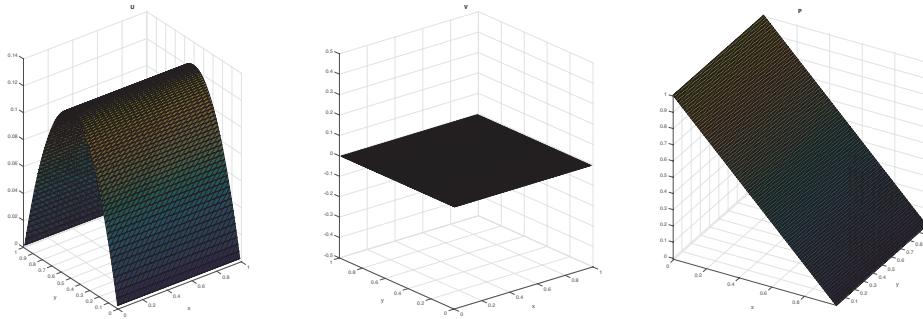


Figure 36.5. Illustration of the solutions for the Stokes flow problem with pressure drop $P = 1$ and viscosity $\mu = 1$. The exact solution is $u = \frac{1}{2}y(1-y)$, $v = 0$, and $p = 1 - x$.

At the inlet and the outlet, you may 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 should be made in the appropriate stencils as above. To enforce the pressure boundary conditions, we must enforce Equation (36.26). Since the pressure straddles the location of the boundary condition, then we numerically enforce the condition by the equations

$$\frac{1}{2}(p_{0,k} + p_{1,k}) = P, \quad (36.28)$$

$$\frac{1}{2}(p_{N,k} + p_{N-1,k}) = 0. \quad (36.29)$$

These equations replace Equation (36.15) at the inlet and outlet.

With these changes made for the boundary conditions, you should be able to assemble a complete SOR solver for this system and compute the flow field for a suitable tolerance. Aim for a relative error of 10^{-5} , i.e. so that the maximum value of the residual divided by the maximum values of u , v , and p is less than 10^{-5} . For some values of ω in the range $0 < \omega < 2$, the method will not converge, and for some values it will. Find the value of ω that makes the method converge fastest.

Your program should take as arguments the grid size N , the viscosity μ , the pressure drop P , the relaxation parameter ω , and the error tolerance τ . You should put in a failsafe number of iterations as 10,000. Use initial data of $u = v = p = 0$. Save your 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. Vary the values of N and ω and generate a plot that illustrates compares the time to completion divided by the number of iterations versus N .

For the case of $\mu = P = 1$, the solution is shown in Figure 36.5.

Version without ghost points

The description for the grid above assumes that ghost points are used for the pressure. An alternative strategy is to not use ghost points, but instead embed the pressure drop P into the stencils for the boundary conditions. In that case, the changes appear in both the equation for u and for p . Figure 36.6 shows the new configuration for the grid without ghost points.

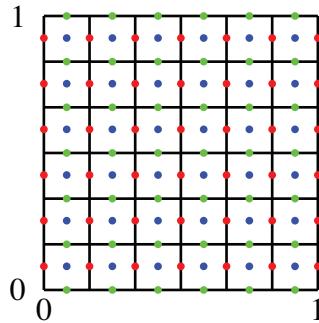


Figure 36.6. Illustration of the marker-and-cell (MAC) grid for the 2D Stokes assignment if using the ghost points for the pressure. The $u_{j,k}$ values are in red, the $v_{j,k}$ values are in green, and the $p_{j,k}$ values are in blue. If a standard grid (black lines) would be an $N \times N$ grid, then the $u_{j,k}$ grid would be a $N \times N - 1$ 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 are correct.

The indexing for the residuals will now change to account for the shift in the grid for $p_{j,k}$ so Equations (36.16)–(36.18) become

$$\begin{aligned} r_j^u &= \frac{\mu \Delta y}{\Delta x} (u_{j-1,k} - 2u_{j,k} + u_{j+1,k}) + \frac{\mu \Delta x}{\Delta y} (u_{j,k+1} - 2u_{j,k} + u_{j,k-1}) \\ &\quad - \Delta y (p_{j,k} - p_{j-1,k}) + \Delta x \Delta y f(x_j, y_k + \Delta y / 2) \end{aligned} \quad (36.30)$$

$$\begin{aligned} r_j^v &= \frac{\mu \Delta y}{\Delta x} (v_{j-1,k} - 2v_{j,k} + v_{j+1,k}) + \frac{\mu \Delta x}{\Delta y} (v_{j,k+1} - 2v_{j,k} + v_{j,k-1}) \\ &\quad - \Delta x (p_{j,k} - p_{j,k-1}) + \Delta x \Delta y f(x_j + \Delta x / 2, y_k) \end{aligned} \quad (36.31)$$

$$r_j^p = -(u_{j+1,k} - u_{j,k}) - \frac{\Delta x}{\Delta y} (v_{j,k+1} - v_{j,k}) \quad (36.32)$$

The boundary conditions for the u and v equations remain the same, but now the values of $p_{-1,k}$ and $p_{N-1,k}$ appear at the left and right ends of the domain in Equation (36.30). In that case, we use the adjusted form of Equations (36.28), (36.29):

$$\frac{1}{2} (p_{-1,k} + p_{0,k}) = P, \quad (36.33)$$

$$\frac{1}{2} (p_{N-1,k} + p_{N-2,k}) = 0. \quad (36.34)$$

The remainder of the discussion is otherwise the same.

Bonus Assignment: Solve the three-dimensional analogue of the above problem of an applied pressure differential across a duct with square cross-section.

Chapter 37

Pseudo-Spectral Methods

For partial differential equations 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 we will consider only one, namely the pseudo-spectral 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 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. Suppose you are 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 we are solving are periodic on the interval $[-\pi, \pi]$. This will simplify the discussion, so from here on we will 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 orthog-

onal 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, we won't be able to compute the a_k exactly as defined in Equation (37.2), but instead will need to 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 we are using $2N$ collocation points 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 $k = N$, $e^{-iN x_j} = (-1)^j = e^{iN x_j}$, and therefore

$$a_{-N} = \frac{1}{2N} \sum_{j=-N}^{N-1} f(x_j) e^{-iN x_j} = \frac{1}{2N} \sum_{j=-N}^{N-1} f(x_j) e^{-i(-N)x_j} = a_N.$$

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

$$a_N = a_{-N} = \frac{1}{2} \frac{1}{2N} \sum_{j=-N}^{N-1} f(x_j) e^{-iN x_j}. \quad (37.5)$$

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 [17], 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 try to keep N to be the product of powers of small prime numbers. This is called the Fast Fourier Transform, or FFT for short, 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, so it is assumed from here that you can convert data on the collocation points into the Fourier coefficients a_k .

Comparing the sums in Equations (37.4), (37.7), it appears that both are sums of a known set of coefficients multiplied by $e^{\pm ikx}$, 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} 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 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 ▪ Pseudo-Spectral Method

The pseudo-spectral method is essentially the coupling of spatial derivatives computed using the spectral approximation described above with a suitable time integration scheme similar to what was encountered in the discussion on finite differences. 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.

There are many choices, but here we'll summarize the popular fourth order Runge-Kutta method. Let \mathcal{L} be a spatial derivative operator on the function $f(x, t)$ and suppose we wish to solve the partial differential equation

$$\frac{\partial f}{\partial t} = \mathcal{L}\{f(x, t)\}. \quad (37.12)$$

The fourth order Runge-Kutta method to advance one time step is given by the following sequence of steps. Let \mathbf{F}_n be the vector of data $[f(x_{-N}, t_n), \dots, f(x_{N-1}, t_n)]$, and let $\mathcal{L}\{\mathbf{F}_n\}$ be the spectral approximation of $\mathcal{L}\{f(x, t_n)\}$. Then the low-storage version of the fourth order Runge-Kutta method is given by

$$\mathbf{F}_n^1 = \mathbf{F}_n + \frac{\Delta t}{4} \mathcal{L}\{\mathbf{F}_n\}, \quad (37.13)$$

$$\mathbf{F}_n^2 = \mathbf{F}_n + \frac{\Delta t}{3} \mathcal{L}\{\mathbf{F}_n^1\}, \quad (37.14)$$

$$\mathbf{F}_n^3 = \mathbf{F}_n + \frac{\Delta t}{2} \mathcal{L}\{\mathbf{F}_n^2\}, \quad (37.15)$$

$$\mathbf{F}_{n+1} = \mathbf{F}_n + \Delta t \mathcal{L}\{\mathbf{F}_n^3\}. \quad (37.16)$$

The Pseudo-spectral 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 partial differential equations 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 analog of Equations (37.3), (37.4) in two dimensions are

$$f(x, y) \approx \sum_{j=-M}^M \sum_{k=-N}^N a_{k,\ell} e^{i(jx+ky)}, \quad (37.17)$$

$$a_{j,k} \approx \frac{1}{2M} \frac{1}{2N} \sum_{\ell=-M}^{M-1} \sum_{m=-N}^{N-1} f(x_\ell, y_m) e^{-i(jx_\ell+ky_m)}. \quad (37.18)$$

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 [10]. The equation is given by

$$\frac{\partial A}{\partial t} = A + (1 + i c_1) \nabla^2 A - (1 - i c_3) |A|^2 A, \quad (37.19)$$

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 2π on each side means that Equation (37.19) is rescaled to be

$$\frac{\partial A}{\partial t} = A + \left(\frac{2\pi}{L}\right)^2 (1 + i c_1) \nabla^2 A - (1 - i c_3) |A|^2 A, \quad (37.20)$$

Use the Pseudo-spectral 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, the dimensions of the grid, N , the real valued coefficients c_1 , c_3 , and the number of time steps, M . It should also accept the optional fifth argument of a seed for the random number generator. 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 the data at the time points $t = 1000k$ for $k = 0, \dots, 10$.

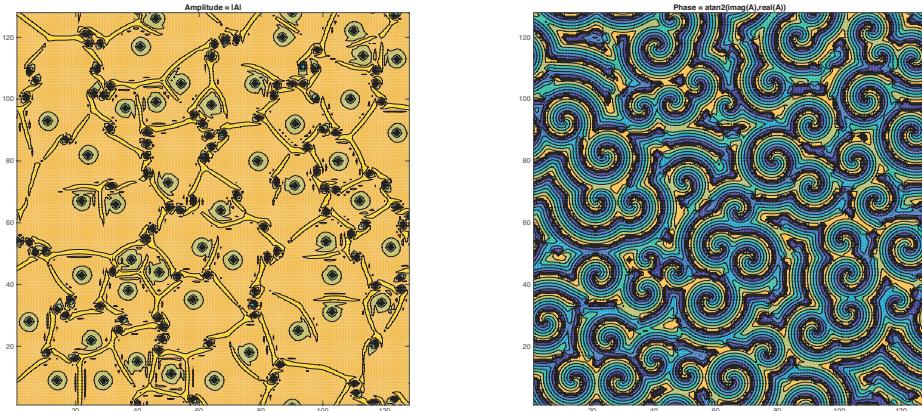


Figure 37.1. Illustration of a sample solution at the terminal time $T = 10,000$ for the case of $c_1 = 1.5$, $c_3 = 0.25$, and $L = 128\pi$ in Equation (37.20). On the left is a contour plot of the magnitude $|A|$ and on the right is a contour plot of the phase, $\arg(A) = \tan^{-1}(Im(A)/Re(A))$. The spiral patterns that emerge can be seen in the right hand plot. Results will vary according to the random initial data.

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 $c_3 = 0.25$.

37.5.2 • Allen-Cahn Equation

The Allen-Cahn equation [11] is an equation that can be used to study phase separation in multi-component 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

$$\frac{\partial \phi}{\partial t} = -\mathbf{v} \cdot \nabla \phi + b \left(\nabla^2 \phi + \frac{\phi(1-\phi^2)}{W^2} \right). \quad (37.21)$$

Use the Pseudo-spectral method with fourth order Runge Kutta using random initial data in the range $-1 \leq \phi \leq 1$ and run until there is a single phase or until the terminal time $T = 5$, whichever comes first. 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, the dimensions of the square grid, N , the two components of the velocity vector, \mathbf{v} , the values of the coefficients b and W , and the number of time steps to use. It should also take an optional seventh argument for the value of the random number seed. 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 ϕ into a file called “allen.out” that contains the data at the time points $t = 0.5k$ for $k = 0, \dots, 10$.

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 .

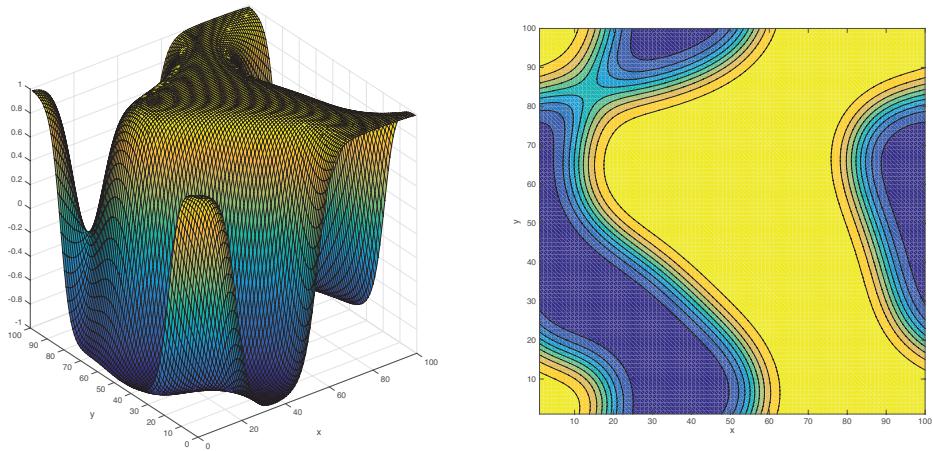


Figure 37.2. Illustration of a solution for the Allen-Cahn equation (37.21) with $b = W = 0.25$ at $t = 2.5$. Here, the phases have mostly separated into two phases represented by the order parameter $\phi = \pm 1$. Results will vary depending on the initial random data.

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 ± 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. ix, 11)
- [2] B. CHAPMAN, G. JOST AND R. VAN DER PAS, *Using OpenMP: Portable Shared Memory Parallel Programming*, MIT Press, Cambridge, MA, 2008. (Cited on p. 79)
- [3] 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. (Not cited)
- [4] J. SANDERS AND E. KANDROT, *CUDA By Example*, Pearson Education, Inc., Boston, MA 2011. (Cited on p. 214)
- [5] B. GASTER, L. HOWES, AND D. R. KAELI, *Heterogeneous Computing with OpenCL*, Second ed., Elsevier Science, Saint Louis, MO, 2012. (Cited on p. 273)
- [6] <https://www.eclipse.org>. (Cited on p. 1)
- [7] <https://developer.microsoft.com/en-us/windows>. (Cited on p. 1)
- [8] <https://developer.apple.com/xcode>. (Cited on p. 1)
- [9] <https://arxiv.org/pdf/1005.2581.pdf>. (Cited on p. 201)
- [10] Z. NICOLAOU, H. RIECKE, AND A. MOTTER, *Chimera States in Continuous Media: Existence and Distinctness*, Physical Review Letters, 119(24) (2017), pp. 244101-1-6. (Cited on p. 367)
- [11] 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. 368)
- [12] J. DOUGLAS, *Alternating Direction Methods for Three Space Variables*, Numerische Mathematik, 4(1) (1962), pp. 41–63. (Cited on p. 349)
- [13] G. MARSAGLIA AND T. A. BRAY, *A Convenient Method for Generating Normal Variables*, SIAM Review, 6 (1964), pp. 260–264. (Cited on p. 339)
- [14] P. E. KLOEDEN AND E. PLATEN, *Numerical Solution of Stochastic Differential Equations*, Springer, Berlin, 1992. (Cited on p. 339)
- [15] R. J. LEVEQUE, *Finite Difference Methods for Ordinary and Partial Differential Equations: Steady-State and Time-Dependent Problems*, SIAM, Philadelphia, 2007. (Cited on p. 345)
- [16] 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. 339)

- [17] 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. 364)
- [18] A. CHRZĘSZCZYK 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. 243)
- [19] <https://www.khronos.org/opencl/> (Cited on p. 273)
- [20] <https://developer.apple.com/opencl/> (Cited on p. 273)

Index

- argc, 9
- argv, 9
- Available Libraries, 171
- blocking, 139
- Checkpointing, 169
- column-major ordering, 57
- compile time, 15
- compiler directive, 9
- compute nodes, 128
- cores, iii
- Critical and Atomic Code, 107
- CUDA Libraries, 239
- Data Types and Structures, 11
 - declared, 45
 - defined, 46
 - device, 205
- Directing Individual Threads, 103
 - dynamic libraries, 55
 - dynamically allocated, 14
- Efficiency Measures, 167
- Elementary Mathematics, 33
- emacs, 1
- file dependencies, 2
- Finite Difference Methods, 345
- Flow Control, 37
- function definition, 9
- Functions, 45
- gcc, 1
- Groups and Communicators, 155
- head node, 128
- header file, 9
- host, 205
- Input and Output, 23
 - input arguments, 9
- Intro to CUDA, 205
- Intro to MPI, 133
- Intro to OpenCL, 273
- Intro to OpenMP, 77
- Iterative Solution of Elliptic Equations, 355
- job, 128
- kernel, 207
- linking, 2
- macros, 11
- main(), 7
- makefile, 2
- Measuring Performance, 53,
 - 111
- memory leak, 17
- message passing, iv
- OpenCL Libraries, 315
- parallel programming, 75, 125
 - passed by reference, 48
 - passed by value, 48
- Passing Messages, 137
- Preliminaries, 127, 203
- Projects for CUDA Programming, 265
- Projects for Distributed Programming, 191
- Projects for OpenCL Programming, 329
- Projects for OpenMP Programming, 119
- Projects for Serial Programming, 71
- Pseudo-Spectral Methods, 361
- queueing system, 128
- row-major order, 57
- scheduler, 128
- scope of a function, 46
- serial programming, 75, 125
- Serial Tasks Inside Parallel Regions, 97
- shared memory architectures, 75
- static libraries, 55
- statically allocated, 14
- Stochastic Differential Equations, 337
- string, 17
- Strong scaling, 167
- Structure of a C Program, 7
- Subdividing For-Loops, 85
- targets, 2
- threads, 77
- Tools of the Trade, 1
- type cast, 16
- Using Libraries, 55
- vi, 1
- vim, 1
- Weak scaling, 167
- xemacs, 1
- zero-based, 14