Contents

Preface . . . xv

CHAPTER 1 Why parallel computing . . . 1

1.1 Why we need ever-increasing performance . . . 2
1.2 Why we're building parallel systems . . . 3
1.3 Why we need to write parallel programs . . . 3
1.4 How do we write parallel programs? . . . 6
1.5 What we'll be doing . . . 8
1.6 Concurrent, parallel, distributed . . . 10
1.7 The rest of the book . . . 11
1.8 A word of warning . . . 11
1.9 Typographical conventions . . . 12
1.10 Summary . . . 12
1.11 Exercises . . . 14

CHAPTER 2 Parallel hardware and parallel software . . . 17

2.1 Some background . . . 17
2.1.1 The von Neumann architecture . . . 17
2.1.2 Processes, multitasking, and threads . . . 19
2.2 Modifications to the von Neumann model . . . 20
2.2.1 The basics of caching . . . 21
2.2.2 Cache mappings . . . 23
2.2.3 Caches and programs: an example . . . 24
2.2.4 Virtual memory . . . 25
2.2.5 Instruction-level parallelism . . . 27
2.2.6 Hardware multithreading . . . 30
2.3 Parallel hardware . . . 31
2.3.1 Classifications of parallel computers . . . 31
2.3.2 SIMD systems . . . 31
2.3.3 MIMD systems . . . 34
2.3.4 Interconnection networks . . . 37
2.3.5 Cache coherence . . . 45
2.3.6 Shared-memory vs. distributed-memory . . . 49
2.4 Parallel software . . . 49
2.4.1 Caveats . . . 50
2.4.2 Coordinating the processes/threads . . . 50
2.4.3 Shared-memory . . . 51
2.4.4 Distributed-memory . . . 55
2.4.5 GPU programming . . . 58
2.4.6 Programming hybrid systems . . . 60

2.5 Input and output . . . 60
2.5.1 MIMD systems . . . 60
2.5.2 GPUs . . . 61
2.6 Performance . . . 61
2.6.1 Speedup and efficiency in MIMD systems . . . 61
2.6.2 Amdahl's law . . . 65
2.6.3 Scalability in MIMD systems . . . 66
2.6.4 Taking timings of MIMD programs . . . 67
2.6.5 GPU performance . . . 70
2.7 Parallel program design . . . 71
2.7.1 An example . . . 71
2.8 Writing and running parallel programs . . . 75
2.9 Assumptions . . . 77
2.10 Summary . . . 78
2.10.1 Serial systems . . . 78
2.10.2 Parallel hardware . . . 79
2.10.3 Parallel software . . . 81
2.10.4 Input and output . . . 82
2.10.5 Performance . . . 82
2.10.6 Parallel program design . . . 83
2.10.7 Assumptions . . . 84
2.11 Exercises . . . 84
CHAPTER 3 Distributed memory programming with MPI . . . 89
3.1 Getting started . . . 90
3.1.1 Compilation and execution . . . 91
3.1.2 MPI programs . . . 92
3.1.3 `MPI_Init` and `MPI_Finalize` . . . 93
3.1.4 Communicators, `MPI_Comm_size`, and `MPI_Comm_rank` . . . 94
3.1.5 SPMD programs . . . 94
3.1.6 Communication . . . 95
3.1.7 `MPI_Send` . . . 95
3.1.8 `MPI_Recv` . . . 97
3.1.9 Message matching . . . 97
3.1.10 The `status_p` argument . . . 98
3.1.11 Semantics of `MPI_Send` and `MPI_Recv` . . . 99
3.1.12 Some potential pitfalls . . . 100
3.2 The trapezoidal rule in MPI . . . 100
3.2.1 The trapezoidal rule . . . 101
3.2.2 Parallelizing the trapezoidal rule . . . 102
3.3 Dealing with I/O . . . 104
3.3.1 Output . . . 105
3.3.2 Input . . . 107

3.4 Collective communication . . . 108
3.4.1 Tree-structured communication . . . 108
3.4.2 `MPI_Reduce` . . . 110
3.4.3 Collective vs. point-to-point communications . . . 112
3.4.4 `MPI_Allreduce` . . . 113
3.4.5 Broadcast . . . 113
3.4.6 Data distributions . . . 116
3.4.7 Scatter . . . 117
3.4.8 Gather . . . 119
3.4.9 Allgather . . . 121
3.5 MPI-derived datatypes . . . 123
3.6 Performance evaluation of MPI programs . . . 127
3.6.1 Taking timings . . . 127
3.6.2 Results . . . 130
3.6.3 Speedup and efficiency . . . 133
3.6.4 Scalability . . . 134
3.7 A parallel sorting algorithm . . . 135
3.7.1 Some simple serial sorting algorithms . . . 135
3.7.2 Parallel odd-even transposition sort . . . 137
3.7.3 Safety in MPI programs . . . 140
3.7.4 Final details of parallel odd-even sort . . . 143
3.8 Summary . . . 144
3.9 Exercises . . . 148
3.10 Programming assignments . . . 155

CHAPTER 4 Shared-memory programming with Pthreads . . . 159
4.1 Processes, threads, and Pthreads . . . 160
4.2 Hello, world . . . 161
4.2.1 Execution . . . 161
4.2.2 Preliminaries . . . 163
4.2.3 Starting the threads . . . 164
4.2.4 Running the threads . . . 166
4.2.5 Stopping the threads . . . 167
4.2.6 Error checking . . . 168
4.2.7 Other approaches to thread startup . . . 168
4.3 Matrix-vector multiplication . . . 169
4.4 Critical sections . . . 171
4.5 Busy-waiting . . . 175
4.6 Mutexes . . . 178
4.7 Producer–consumer synchronization and semaphores . . . 182
4.8 Barriers and condition variables . . . 187
4.8.1 Busy-waiting and a mutex . . . 188
4.8.2 Semaphores . . . 189
4.8.3 Condition variables . . . 190
4.8.4 Pthreads barriers . . . 193

4.9 Read-write locks 193
- 4.9.1 Sorted linked list functions 193
- 4.9.2 A multithreaded linked list 194
- 4.9.3 Pthreads read-write locks 198
- 4.9.4 Performance of the various implementations 200
- 4.9.5 Implementing read-write locks 201

4.10 Caches, cache-coherence, and false sharing 202
4.11 Thread-safety 207
- 4.11.1 Incorrect programs can produce correct output 210

4.12 Summary 210
4.13 Exercises 212
4.14 Programming assignments 219

CHAPTER 5 Shared-memory programming with OpenMP 221

5.1 Getting started 222
- 5.1.1 Compiling and running OpenMP programs 223
- 5.1.2 The program 224
- 5.1.3 Error checking 227

5.2 The trapezoidal rule 228
- 5.2.1 A first OpenMP version 229

5.3 Scope of variables 233
5.4 The reduction clause 234
5.5 The `parallel for` directive 237
- 5.5.1 Caveats 238
- 5.5.2 Data dependences 239
- 5.5.3 Finding loop-carried dependences 241
- 5.5.4 Estimating π 241
- 5.5.5 More on scope 244

5.6 More about loops in OpenMP: sorting 245
- 5.6.1 Bubble sort 245
- 5.6.2 Odd-even transposition sort 246

5.7 Scheduling loops 249
- 5.7.1 The `schedule` clause 250
- 5.7.2 The `static` schedule type 252
- 5.7.3 The `dynamic` and `guided` schedule types 252
- 5.7.4 The `runtime` schedule type 253
- 5.7.5 Which schedule? 255

5.8 Producers and consumers 256
- 5.8.1 Queues 256
- 5.8.2 Message-passing 256
- 5.8.3 Sending messages 257
- 5.8.4 Receiving messages 257
- 5.8.5 Termination detection 258
- 5.8.6 Startup 259
- 5.8.7 The `atomic` directive 259

5.8.8 Critical sections and locks 260
5.8.9 Using locks in the message-passing program 262
5.8.10 `Critical` directives, `atomic` directives, or locks? .. 263
5.8.11 Some caveats 264
5.9 Caches, cache coherence, and false sharing 265
5.10 Tasking 270
5.11 Thread-safety 274
5.11.1 Incorrect programs can produce correct output ... 276
5.12 Summary 277
5.13 Exercises 281
5.14 Programming assignments 286

CHAPTER 6 GPU programming with CUDA **291**
6.1 GPUs and GPGPU 291
6.2 GPU architectures 292
6.3 Heterogeneous computing 293
6.4 CUDA hello 295
6.4.1 The source code 296
6.4.2 Compiling and running the program 296
6.5 A closer look 298
6.6 Threads, blocks, and grids 299
6.7 Nvidia compute capabilities and device architectures 301
6.8 Vector addition 302
6.8.1 The kernel 303
6.8.2 `Get_args` 305
6.8.3 `Allocate_vectors` and managed memory 305
6.8.4 Other functions called from `main` 307
6.8.5 Explicit memory transfers 309
6.9 Returning results from CUDA kernels 312
6.10 CUDA trapezoidal rule I 314
6.10.1 The trapezoidal rule 314
6.10.2 A CUDA implementation 315
6.10.3 Initialization, return value, and final update 316
6.10.4 Using the correct threads 318
6.10.5 Updating the return value and `atomicAdd` 318
6.10.6 Performance of the CUDA trapezoidal rule 319
6.11 CUDA trapezoidal rule II: improving performance 320
6.11.1 Tree-structured communication 320
6.11.2 Local variables, registers, shared and global memory 322
6.11.3 Warps and warp shuffles 324
6.11.4 Implementing tree-structured global sum with a warp shuffle 325
6.11.5 Shared memory and an alternative to the warp shuffle 326

6.12 Implementation of trapezoidal rule with `warpSize` thread blocks 328
6.12.1 Host code 329
6.12.2 Kernel with warp shuffle 329
6.12.3 Kernel with shared memory 329
6.12.4 Performance 330
6.13 CUDA trapezoidal rule III: blocks with more than one warp 331
6.13.1 `__syncthreads` 331
6.13.2 More shared memory 333
6.13.3 Shared memory warp sums 333
6.13.4 Shared memory banks 334
6.13.5 Finishing up 336
6.13.6 Performance 336
6.14 Bitonic sort 338
6.14.1 Serial bitonic sort 338
6.14.2 Butterflies and binary representations 342
6.14.3 Parallel bitonic sort I 345
6.14.4 Parallel bitonic sort II 347
6.14.5 Performance of CUDA bitonic sort 348
6.15 Summary 349
6.16 Exercises 355
6.17 Programming assignments 358

CHAPTER 7 Parallel program development 361
7.1 Two n-body solvers 361
7.1.1 The problem 361
7.1.2 Two serial programs 363
7.1.3 Parallelizing the n-body solvers 368
7.1.4 A word about I/O 371
7.1.5 Parallelizing the basic solver using OpenMP 371
7.1.6 Parallelizing the reduced solver using OpenMP . . . 375
7.1.7 Evaluating the OpenMP codes 379
7.1.8 Parallelizing the solvers using Pthreads 380
7.1.9 Parallelizing the basic solver using MPI 381
7.1.10 Parallelizing the reduced solver using MPI 383
7.1.11 Performance of the MPI solvers 389
7.1.12 Parallelizing the basic solver using CUDA 390
7.1.13 A note on cooperative groups in CUDA 393
7.1.14 Performance of the basic CUDA n-body solver . . . 393
7.1.15 Improving the performance of the CUDA n-body solver 394
7.1.16 Using shared memory in the n-body solver 395
7.2 Sample sort 398
7.2.1 Sample sort and bucket sort 398
7.2.2 Choosing the sample 400

7.2.3 A simple implementation of the `Map` function 401
7.2.4 An alternative implementation of `Map` 402
7.2.5 Parallelizing sample sort 407
7.2.6 Implementing sample sort with OpenMP 410
7.2.7 Implementing sample sort with Pthreads 415
7.2.8 Implementing sample sort with MPI 418
7.2.9 Implementing sample sort with CUDA 429
7.3 A word of caution 439
7.4 Which API? 439
7.5 Summary 440
7.5.1 MPI 442
7.6 Exercises 444
7.7 Programming assignments 451

CHAPTER 8 Where to go from here **455**

Bibliography 459
Index 463