Scientific Programming 19 (2011) 67–70
DOI 10.3233/SPR-2011-0318
IOS Press

Book Review

Introduction to Concurrency in Programming Languages, by Matthew J. Sottile, Timothy G. Mattson and Craig E. Rasmussen, CRC Press, Boca Raton, London, New York, 2009, ISBN: 978-1-420-07213-6, US$79.95.

Parallel programming is mainstream scientific programming. A desktop workstation can easily have hundreds of pipelines. An undergraduate course in, say, computational physics should introduce the student to parallel solutions. What issues arise in parallel programming that don't arise in sequential programming? How are the issues treated? And what text to use to explain the differences? For bonus questions, how did we learn about parallel programming? Why are the solutions organized the way they are?

Introduction to Concurrency in Programming Languages has 13 Chapters, three Appendices, and totals 330 pages, including the references and the index. The book explores how concurrent programming can be assisted by language features, and what language features there are to help with concurrent programming. It also discusses historical evolution of hardware and the corresponding evolution of programming languages in support of concurrency in applications. The book starts with hardware details and takes us as far as design patterns for parallel programming. The appendices briefly describe OpenMP (in C), Erlang and Cilk. This book is part of the Chapman & Hall/CRC Computational Science Series, Horst Simon, LBNL, Series Editor.

Each chapter starts with a list of Objectives, and concludes with a set of Exercises. The book's web site, www.parlang.com, is a bare-bones web site (as of this writing) with links to the Publisher, Amazon.com, example codes, errata, compilers and languages (a good list, if missing Ada, APL (or J, its ASCII character derivative) and ZPL), course materials (PowerPoint sets for most chapters, more coming), press (reviews and such), and (how 21st century) "follow us" on Twitter. There is an email address for suggestions and reporting errata. I reported a couple of typos, and was promptly credited on the web site.

There is no Preface, so let's march straight into Chapter 1, Introduction. Our objectives are to motivate a discussion of concurrency and to demonstrate it with some examples. First, we must distinguish *concurrency* from *parallelism*. We make the usual distinction, that concurrency means "may be in progress simultaneously" while parallelism means "may be in execution simultaneously" so parallelism is a strict subset. Thus, single processor systems have, through timesharing, given us concurrency for years, but multi-core chips give us parallelism in the desktop system.

We learn that parallel applications have historically been the domain of scientific programmers, but that operating systems and databases have served to bring concurrency issues to a wider range of programmers. From early assemblers, programming languages have evolved to a higher level, but have not yet brought parallelism to the same degree of abstraction. Languages for writing full-blown parallel applications have not yet left the academy (where they still provide material for PhD dissertations). Where does concurrent programming now appear? The authors tell us it's in operating systems, distributed systems, user interfaces, databases and, of course, scientific applications. Missing is any discussion of the possibility of concurrency appearing in interpreters (for example, Python), or in workbench programs (for example, Octave, R or Scilab), we leave that as fodder for more of those PhD dissertations.

The rest of the book starts by examining some of the issues raised by concurrent programming, and so motivated, examines some of the tools a programming language might have for addressing them. This takes us to the parallel design patterns that form the last several chapters. We also learn that the book is aimed towards undergraduates with some programming experience. I imagine that indicates upper division undergraduates. The book might be helpful as supplemental reading for a first-year graduate course if the students have uneven backgrounds, or if the instructor wants to impart some of the historical background or cognitive issues discussed here.

So motivated and oriented, we press ahead to Chapter 2, Concepts in Concurrency. We want to be able to code our concurrent algorithms in a form readable by mortal humans, using standard features of programming languages, that avoid the correctness and performance issues concurrency gives us. Parallel program-

ming terminology is messy, but to proceed we need some. To keep the terms thread and process with their perhaps most popular meanings, the unit of concurrent execution is called a task. We identify parallelism as a subset of concurrency. Next, it's off to clarify data dependencies. After a spot of history, we find the concept of atomicity. The fact of the fetch, operate, store sequence tells us that what we may consider to be atomic in software may not be in hardware, the basis of our program's execution. So we need mutual exclusion and critical sections. Generalizing to consistency, we find the complications introduced by caches. Generalizing again, we find thread safety.

And so we advance to Chapter 3, Concurrency Control. The goal this time is to see the specific problems that concurrency brings, and examine in detail the causes in order to see what sort of correctives must be applied. The correctness issues we find are race conditions, deadlock and livelock. The livelock discussion leads to consideration of liveness, starvation and fairness. The techniques to address these problems include the synchronization tools of semaphores, mutual exclusion and rendezvous. The reader may already be familiar with locks. We learn enough Dutch, when discussing semaphores, to understand Dijkstra's P (prolagen: to try and lower) and V (verhogen: to raise). With monitor variables we attack the producer/consumer pattern. We check how databases treat transactions.

This leads us to Chapter 4, The State of the Art. We motivate language features by examining currently available libraries, message passing and explicit threading, to provide examples we can generalize into higher level abstractions, and we see how some languages have incorporated these ideas. The state of the art, unfortunately, is message passing libraries. Explicit threading is also a low-level library-based practice. Libraries are hobbled by not having, or having only in clumsy ways, access to language level concepts, such as types or arrays. Some higher-level ideas include transactional memory, event-driven programming and the actor model. Within languages, even within the higher-level languages, execution-time aliasing is the enemy of analysis, automatic or human.

Our next step is to learn what higher-level languages provide us, in Chapter 5, High-Level Language Constructs. We want to learn what sequential languages have, within their feature sets, to help with concurrent programming. Side effects come to the fore. Thus, there is a cognitive effect upon the programmer of using sequential languages for concurrent programming. We distinguish imperative languages and declarative languages. We see the high-level code, and peek at some assembler, seeking possible weak spots. We want to gain the goals of Readability, Writability and Reliability. The discussion of the cognitive aspects of concurrent programming is one of the most interesting parts of the book for me. While one certainly wants to use a high level language (especially after being re-acquainted with assembler!), one must beware of building an abstraction barrier to understanding the issues concurrency brings. Some of these issues are apparent only when the actual code to be executed is carefully examined using actual data (for example, aliasing analysis). Interpreted languages get a brief mention, due to the ability of the interpreter to query the state of the program.

It's time to take a step back and get some history, which is what we find in Chapter 6, Historical Context and Evolution of Languages. Here, we examine historic hardware evolution, and the corresponding programming language development, that are relevant to our study of concurrency. Nodding towards Moore's Law, we see how increasing sophistication of hardware allowed interrupt-driven input/output, which makes concurrent execution of programs attractive. Then we must consider the effects of caches. Our story takes us to the Solomon project at Westinghouse, leading to the ILLIAC IV at the University of Illinois. Then we turn our attention to the CDC line, leading, of course, to the Cray PVPs (parallel vector processors). We see the von Neumann machine, with its memory bottleneck before turning to the massively parallel computers of the 1980s. We get a reference to the Top 500 list. We next head towards VAXClusters, before finding ourselves at the currently popular Linux cluster of commodity boxes. We stop at Flynn's Taxonomy before turning to languages. This discussion starts with Fortran (in the beginning, spelled FORTRAN, which my spelling checker still prefers even though all caps has been incorrect for decades now). ALGOL leads to Pascal and Modula. A little further and we find Ada. We distinguish declarative and functional languages, move to dataflow languages and logic languages, before finally landing at parallel languages. Here, too, we begin with High Performance Fortran and examine data layout before heading to ZPL, with its expressive regions. A brief word of the limits to autoparallelization completes the chapter.

All of which brings us to Chapter 7, Modern Languages and Concurrency Constructs. We want to see how language arrays, message passing and control flow relate to concurrency. We also discuss functional lan-

guages from a concurrency vantage point. We need arrays as first class objects to proceed, alas, lacking in the C family of languages. We meet the term syntactic sugar, and its less-familiar cousin, syntactic salt. Next we visit array notation, which leads to the Connection Machine languages. Now we must face message passing. There are one-sided and two-sided varieties of message passing. Some languages have message passing as a feature of the language, so we meet the Erlang language with its actor model and channels. Now we're ready for coarrays (misspelled with a hyphen). Unfortunately, it's not the standard coarrays, but an earlier syntax from Cray's original definition. This puts us at the gates of the PGAS (partitioned global address space) languages, but there's a name-only mention of UPC (Unified Parallel C) and Titanium (with no mention of Split/C). The control flow discussion leads to parallel loops before we venture towards functional languages and functional operators.

It's back to reality in Chapter 8, Performance Considerations and Modern Systems. Our objectives are to examine processor performance versus memory performance, Amdahl's Law, the effects of locks and, more generally, the performance overhead of concurrency constructs. The memory discussion centers on the effects of caches. That leads to the issue of row-major arrays versus column-major arrays. The example given is passing MATLAB arrays to a C function. After reviewing Amdahl's Law (Amdahl's Second Law, of course), we discover that parallel overhead brings a new term into the equation. And excessive use of locks can serialize a whole program. Other sources of overhead include thread creation, so one might consider forming thread pools. Lastly we find the issue of the work performed per synchronization.

In the home stretch, we're ready for Chapter 9, Introduction to Parallel Algorithms. We try to see how to identify concurrency in our algorithms, and what to seek in the source code of a sequential program. This is a short chapter, and simply prepares us for the parallel design patterns that form the topics of the final four chapters. The patterns chapters are where the examples are, so without further ado, let's advance.

We start the patterns with Chapter 10, Pattern: Task Parallelism. We discuss task parallelism and its underlying algorithmic structures, use genetic algorithms and the tried-and-true Mandelbrot set as examples, and see examples in Erlang, Cilk and OpenMP (using C). The algorithms supporting the task parallelism pattern include the master-worker, SPMD (single program, multiple data), loop-level parallelism and fork-join. We see some snippets written in Cilk and Erlang for steps along the way. A view of a genetic algorithm in Erlang is shown. With some discussion of task granularity, a Cilk implementation of the Mandelbrot calculation follows.

Our next pattern is in Chapter 11, Pattern: Data Parallelism. This time, we learn about data parallelism with a matrix multiply as our example. Next we discuss its limitations, and how it shades into task parallelism along what is really a spectrum. In short, a geometric decomposition leads to the SPMD design pattern. There's the matrix multiply, followed by a cellular automaton. The cellular automaton example is array-based, so its coded in Fortran to take advantage of the array intrinsics and array control statements (with some sacrifice of efficiency due to copying the working arrays). Further examples in MATLAB and OpenMP (in C) complete the presentation.

Another pattern is discussed in Chapter 12, Pattern: Recursive Algorithms. We review recursion as a sequential pattern, and view some examples of parallel implementations. So out trots the Fibonacci function, and its sidekick, the factorial function. This leads to a discussion of function side effects, before plunging into the divide-and-conquer pattern and a sorting example. Then we see a divide-and-conquer Sudoku solver. (I may have just gotten a better idea of how to solve Sudoku puzzles.)

Our final pattern is in Chapter 13, Pattern: Pipelined Algorithms. We introduce the pipeline paradigm, demonstrate it in Erlang, and discuss the application to the visual cortex in the brain. We find a description, and an illustration of a pipeline in Figure 13.1, which has a bug, but the fix is on the web site. The software discussion leads us to some code snippets in Erlang.

Finally, there's the appendices. The appendices are brief discussions of the basic ideas of OpenMP (in C), Erlang and Cilk. There are enough references to lead the student to a fuller discussion of each.

So where have we been? At one level, we have an undergraduate computer science text. The stated goal is to give a discussion of concurrency and the issues it brings forward into the undergraduate computer science classroom. At this level, the book succeeds. The chapters are well ordered and structured, the exercises are appropriate for the material, the coverage is fairly broad. The references section is thorough so students can pursue further reading as desired. At another level, it's a Renaissance Reader of the history of concurrent (and parallel) programming. I also appreciate the discussions of cognitive issues. Too often we ignore these

important considerations. An instructor might want to read this book, just for examples and the broad coverage of the material. Also, this book will compliment well the main text, for example, in an undergraduate computational physics course.

What's missing? I saw nothing on debugging, or on when to use a particular pattern (matrix multiplication is an intrinsic procedure, or easily available from a library – the student likely won't code one other than as an exercise). I saw nothing regarding the effects different orders of operations can have on floating point results. The criticism that modern standardized languages don't support parallelism is fading fast. Ada has had parallelism for over 25 years now. Java is not an international standard language, but is stable, supports parallelism, and is herein mentioned, but little more. Today's computational science undergraduate is more likely, I believe, to encounter Java code than Erlang code or Cilk code, during a career in computational science. The current Fortran standard has coarrays, and coarrays are supported by several current compilers. The C++ draft standard describes thread classes, and supporting classes and templates. The upcoming C standard is reported to describe threads in a library, and pthreads are almost universally available to standard C programs anyway. And I like the PGAS languages. I think the PGAS model works well with many scientific problems (and the SPMD pattern is repeatedly mentioned in this book), so I would have appreciated more emphasis on UPC and Fortran's coarrays. But this is an undergraduate text, aimed towards general computer science students. For that, it's a good choice. Programmers today must be able to program on commodity hardware, and that means parallel hardware. Even if one imagines writing only sequential programs, concurrency is a mainstream issue. For undergraduates in computational science, perhaps the emphasis should be shifted somewhat, but this is still a good choice for a course text, or a supplemental text.

I like the historical reviews. They help complete my war-story-based version of how we got here. (Someday, I promise, I'll find time to read Sammet's *Programming Languages: History and Fundamentals* and the HOPL (History of Programming Languages) series, but no time today.) How hardware and software evolved together also helps motivate the solution to the issues raised by concurrency. In computational science, one must write code to get the right answer reliably, of course, but also write so colleagues can read it for understanding, as scholarly criticism is part of the scientific process. So the cognitive issues are, perhaps, more important in computational science than in computer science. All of that makes for a solid introduction.

Dan Nagle
E-mail: danlnagle@me.com