Scientific Programming 17 (2009) 343–345
DOI 10.3233/SPR-2009-0295
IOS Press

Book Review

Dan Nagle
George Mason University, Computational and Data Sciences, 4311-G Bob Ct., Fairfax, VA 22030, USA
E-mail: dnagle@gmu.edu

The Art of Concurrency, by Clay Breshears, O'Reilly, Sebastapol, CA, USA, 2009, ISBN 978-0596521530

The Art of Concurrency is subtitled *A Thread Monkey's Guide to Writing Parallel Applications*. It has a Preface, 11 Chapters, a Glossary and 285 pages, including the index. The author wants to help experienced professional applications programmers master the task of multi-threaded programming. The idea is to take simple problems, find a suitable algorithm, and apply a multi-threaded solution using one (or more) of OpenMP, pthreads, Intel Threading Building Blocks (hereinafter TBB), or Windows threads. Most of the examples require C, except that the TBB examples require C++. C is a good choice, for it is a *lingua franca* of professional programmers and it clearly displays all the steps involved.

The Preface explains the hardware reasons for multicore chips, and the response of software that is now to be written as multi-threaded applications. This book is intended for all programmers everywhere. The programmer should have some familiarity with multithreading programming methods, specifically on any scheme to be actually used. The focus here is algorithms rather than library details. Then an outline of the chapters is presented. Not that scientific programmers are not professionals, but where does this book fit in the scientific programming literature? Let's keep reading, and find the answer to that question.

The first chapter, *Want to Go Faster? Raise Your Hand If You Want to Go Faster!* tells us that the author wants to share his experience parallel programming. And he certainly has a great deal of experience to share. We next learn that "*Thread Monkey*" is good, just like "*Grease Monkey*" is good. On the other hand, "*Code Monkey*" is bad. This not only assuaged my ego, but it also helped to get my filters adjusted to the author's sense of humor. The author clearly defines and distinguishes parallelism and concurrency. Concurrent means may be *in progress* at the same time as something else, parallel means may be *executing* at the same time as something else. The author distinguishes the kind of parallelism he will discuss from the scalable and popular, if tedious, message passing. So why would a programmer need to understand multithreading? Because that's where the hardware is going. Isn't multi-threading hard? Not if you follow the rules, which aren't all that hard to learn. We'll assume that the programmer has a working serial program. What are the steps towards parallelism? We are told: first, Analysis: Identify Possible Concurrency; next, Design and Implementation: Threading the Algorithm; next, Test for Correctness: Detecting and Fixing Threading Errors; finally, Tune for Performance: Removing Performance Bottlenecks. And so we've already gotten some guidance. Why not start our parallel application from scratch? Because then you've got two sources of error: logic (ordinary bugs) and parallelism. Now, we can examine some idealized hardware (including Parallel Random Access Machine, PRAM) and we're off to examine algorithms for concurrency.

The next chapter, *Concurrent or Not Concurrent*, discusses design models, specifically, data decomposition and task decomposition. We'll start with task decomposition. So we seek tasks that are independent, that is, that do not have dependencies. How does one identify independent tasks? Experience, which to the novice means practice. There are only two data dependencies: Want new value and got old value; or want old value and got new value. One must guard against both. One may have first encountered them when vectorizing.

Try imagining two sections of code executing simultaneously. Do they interfere? Then one must ask how to map tasks to threads, and how much work will be in each task, that is, the granularity of each task. The greater the work per thread, the better. The author warns us away from using Thread Local Storage (TLS) as requiring high latency and use other language constructs to achieve the desired end. How to assign tasks to threads? With OpenMP and TBB, it's done automatically, although the programmer can exert some influ-

ence. With bare threads, the programmer must be explicit with every detail. An example of a numerical definite integral provides concreteness to the discussion.

We advance to data decomposition, and the decisions of how to divvy data into independent chunks. The issues of load balance and the boundary to interior ratio come to the fore. Thus, we meet ghost cells. The game of Life provides the example. Here, we meet the author's scorecard. Parallelization is to be judged on the basis of Efficiency, Simplicity, Portability and Scalability. We end this chapter with a discussion of algorithms that are not parallel, at least, not without some rearranging.

I won't repeat it for each algorithm that is parallelized, but one of my favorite characteristics of this book is the author's scorecard. Each parallelization effort is described in terms of the Efficiency, Simplicity, Portability and Scalability mentioned above. The Efficiency score is especially useful to scientific programmers, as the author pays close attention to cache efficiency as well as parallel efficiency. The cache aspect of programming is all too often overlooked. Yet it can have a powerful effect on overall performance of a code. As more cores populate each chip, the amount of off-chip memory access will become more critical.

The next chapter, *Proving Correctness and Measuring Performance*, gives us a technique for proving correctness. Consider the program to be a sequence of atomic statements, and then consider the state of the program when the atomic statements of multiple threads are interleaved with all possible sequential orderings. So armed, we'll attempt to write code to enforce mutual exclusion. We'll fail in four attempts, but it's an instructive effort. So we try Dekker's algorithm, which works, but is only defined for two threads. It is better to use the mutual exclusion objects supplied by our chosen parallelization package. Next, the issue of scaling and performance is discussed. This, of course, brings us to Amdahl's Law and the Gustafson–Barsis criticism of it. A short further discussion of hardware, and we're ready to advance to the next chapter.

The next chapter, *Eight Simple Rules for Designing Multithreaded Applications* is exactly what it says it is. I won't give the rules away (one should have some incentive to buy the book, after all). I will say that these rules are practical and useful, and they are referenced in the chapters discussing actual problems. They sometimes provide the decisive argument for choosing between two serial algorithms to be parallelized.

The next chapter, *Threading Libraries*, is slightly misnamed. It also describes OpenMP and TBB, which is a template library for use with C++. But explicit threading libraries are discussed, specifically, pthreads and Windows threads. The chapter contains a short list of other parallelization schemes, missing are Ada's parallelism and, for scientific programmers, glaringly, Unified Parallel C (UPC, upc.gwu.edu and links from there).

With the next chapter, *Parallel Sum and Prefix Scan*, we commence the main event, the discussion of converting serial codes to parallel codes. We'll start with a Parallel Random Access Machine algorithm (PRAMs were discussed in the hardware section of Chapter 1). This introduces us to the idea of a logarithmic reduction scheme. The question becomes one of how to synchronize the algorithm when using threads. This issue of the decreasing number of threads at work is also explored. An OpenMP or TBB reduction could be used, but a pthreads parallel solution is also shown, in all its gory details. We next investigate a prefix scan, again starting with a PRAM algorithm. This leads us to find a Windows threads parallel solution. These solutions lead to the investigation of the problem of selecting the kth largest element from an array. We start with a serial selection algorithm, which is then parallelized via TBB. There's a discussion of packing by use of the prefix scan.

The next chapter, *MapReduce*, shows us how to parallelize problems involving the pair of operations, map and reduce. First, we'll parallelize map, and then we'll parallelize reduce. Along the way, we'll find the need for a barrier. The author uses a RED/BLUE color barrier as the barrier implementation. I suspect it's the equivalent of the NYU barrier, but I haven't traced the execution of both to verify it. Anyway, barriers are harder to implement than they appear, as the author shows. So it's of value to scientific programmers to have a working barrier presented and explained. The chapter concludes by applying MapReduce to several problems.

The next chapter, *Sorting*, starts with a bubblesort. A simple parallelization of bubblesort gives data races all along the array, so we must apply critical sections liberally, using Windows threads. After some blocking, a careful analysis reveals that all is well, the apparent remaining data races are either benign, or, upon close inspection, nonexistent. Having seen all the locking in the bubble sort, we move to an Even/Odd Transposition sort. This may the considered to be akin to a red/black scheme, but in one dimension. This is implemented using OpenMP. Final polishing of this code leads us to try to place the parallelization as far up the

call tree as possible. We move to a shellsort, along with a review of insertion sorts. After seeing the serial code, and modifying it a bit, we see an OpenMP implementation of a parallel shellsort. We move further to Quicksort, with a serial version and a parallel version made using Windows threads. Lastly, we analyze a radix sort, producing a parallel version using pthreads. Yes, there are many ways to sort. But a point raised and answered here is that the best serial algorithm need not be the best algorithm to try to parallelize. The author clearly makes this important point.

The next chapter, *Searching*, leads us through sorted and unsorted datasets. For the unsorted dataset, the data decomposition is easy, and we are lead to n-ary searches. The issue becomes how to control the stop of the search when the target is found. For the sorted case, there's the binary search, which can be parallelized via OpenMP, taking advantage of the implicit barriers.

The last chapter describing techniques is *Graph Algorithms*. This chapter starts with a set of definitions from graph theory, to get us on a common ground. We'll explore depth-first algorithms. We need to consider how many locks are required for a correct search. Now, we can parallelize the depth first search via Windows threads. To switch to breadth-first algorithms, we merely exchange our thread-safe stack for a thread-safe queue. Thus prepared, we tackle an All Pairs Shortest Path problem and a Minimum Spanning Tree problem.

The very last chapter is a short one, *Threading Tools*. This chapter surveys some of the debuggers and analyzers available for multi-thread applications. The survey starts with the free yet thread-aware Gnu gdb and moves to various commercial products before describing Intel's thread product set.

This book is aimed towards the professional programmer, most likely employed by a commercial software house, and making home and office applications. For its intended audience, it's excellent. The author has great experience, and he has conveyed it in a clear and well-organized fashion. Kudos. Where does this book fit with scientific programmers? That is, where does it fit with academic and research-oriented programmers, perhaps having teaching duties as well as research duties?

This book describes parallelizing several algorithms of use to scientific programmers. In fact, I think I can say that most, if not all, the algorithms are useful to scientific programmers in some scientific application or other. The explanations are clear, and the pointers to the literature show where to go for further information.

One item I missed is a Bibliography. Several times, when reading the book, I read past a citation to the literature as the source of an idea or algorithm. After reading the parallel implementation, and deciding it was very good indeed, I had to skim backwards, seeking the italic font used in the citation. The author has great knowledge of parallel programming and a great library to support his knowledge. Placing the citations all in one place would be helpful. When teaching, I might use this book as a supplemental text. But it lacks exercises at the end of its chapters, so I would want something else as a primary text. Perhaps I would use it as a recommended reading book for a second term of an introductory scientific programming course, or second year parallel programming course. Or perhaps I would assign a larger, term-length project that could be built step-by-step using threads.

I hope the author will bring forward a second edition of this book after the C and C++ standardization committees publish their new standards (C $1\times$ and C++ $0\times$), which will have standard threading. The standardization of threading in the C/C++ languages will reduce the pthreads versus Windows threads tension. (A second edition would give a chance to include a bibliography!)

I think scientific programming is moving in the direction of the Partitioned Global Address Space languages (UPC, Titanium, Coarray Fortran are the most popular), rather than threading, as the parallelization paradigm of favor (at least at the moment). Thus, examples using UPC, or even standardized Fortran coarrays, would be of interest to scientific programmers. One can get solid strategy ideas from this book, together with clear explanations. It will be up to the programmer, however, to translate them into UPC or Coarray Fortran. Even if one wants to stick with threading, one will need a separate book on the specific thread scheme chosen. The same publisher has several good ones, and there are others.

So, if a researcher wants a gentle yet clear introduction to multi-threading, this is a great start. The serial algorithms are explained, as are the steps to parallelization, along with the pitfalls and gotchas, very clearly. The scorecards show how to evaluate what one's efforts have produced. The references show where to seek further information. Somehow, I think multi-threaded programming ought to be worth more than the title of "*Thread Monkey*" but this book certainly is a solid, and very helpful, introductory exposition of *The Art of Concurrency.* And that is exactly what is says it is.