Review of *Elements of Parallel Computing*
Author of Book: *Eric Aubanel*
Published by Chapman & Hall/CRC
238 pages, Hardcover, £160.00

Review by
Michele Amoretti (michele.amoretti@unipr.it)
Department of Engineering and Architecture
University of Parma

1 Introduction

As the title clearly states, this book is about parallel computing. Modern computers are no more characterized by a single, fully sequential CPU. Instead, they have one or more multicore/manycore processors. The purpose of such parallel architectures is to enable the simultaneous execution of instructions, in order to achieve faster computations. In high performance computing, clusters of parallel processors are used to achieve PFLOPS performance, which is necessary for scientific and Big Data applications.

Mastering parallel computing means having deep knowledge of parallel architectures, parallel programming models, parallel algorithms, parallel design patterns, and performance analysis and optimization techniques. The design of parallel programs requires a lot of creativity, because there is no universal recipe that allows one to achieve the best possible efficiency for any problem.

The book presents the fundamental concepts of parallel computing from the point of view of the algorithmic and implementation patterns. The idea is that, while the hardware keeps changing, the same principles of parallel computing are reused. The book surveys some key algorithmic structures and programming models, together with an abstract representation of the underlying hardware. Parallel programming patterns are purposely not presented using the formal design patterns approach, to keep an informal and friendly presentation that is suited to novices.

2 Summary of Contents

The book has two parts. In the first part (Chapters 1–5), the core concepts are presented. In the second part (Chapters 6–8), three case studies are presented with the purpose to reinforce the concepts of the earlier chapters. Every chapter ends with a section about further reading and a section of exercises.

1. Chapter 1: Overview of Parallel Computing. This chapter starts with an introduction to parallel computing, motivating the need for a new approach based on a suitable level of abstraction and the awareness that a limited number of solutions keep being re-used. Then, terminology is made clear, with reference to Flynn's taxonomy. The evolution of parallel computers is shortly presented, with particular emphasis on the cluster-grid-cloud progression and on the emergence of multicore and manycore processors. Then, parallel programming models are divided in three categories (implicit, semi-implicit and explicit) that are shortly presented. The relevance of *thinking in parallel* is outlined. The chapter includes a discussion about the parallel implementation of the high level patterns defined in the OPL (Our Pattern

Language) hierarchy. The top two layers, namely Structural Patterns and Computational Patterns, are presented. The chapter ends with the presentation of the algorithm that solves the word count problem as an instance of the MapReduce pattern, with a discussion on distributed and shared memory implementations.

2. Chapter 2: Parallel Machine and Execution Models. The first part of this chapter discusses three machine models, i.e., SIMD (Single Instruction Multiple Data), shared memory and distributed memory. SIMD ALUs are illustrated by means of register-level code examples. The difference between conventional SIMD and SIMT (Single Instruction Multiple Threads), characterizing NVIDIA GPUs, is also illustrated. Regarding MIMD (Multiple Instruction Multiple Data) parallel computers, the difference between multicore processors, multiprocessors and multicomputers is discussed. Shared memory execution and distributed memory execution are illustrated by means of simple examples. Regarding the former model, the concepts of data race, sequential consistency and cache coherence are presented. The second part of the chapter presents a task graph execution model defined as a directed acyclic graph, where each vertex represents the execution of a task, and an edge between two vertices represents a real data dependence between the corresponding tasks. Some simple examples of task graphs are explored, to get a feeling for how this execution model represents dependencies between tasks. The particular and highly relevant case of *reduction* is also presented, with reference to the word count example proposed in Chapter 1. The chapter ends with a discussion on mapping a task graph to a given machine.

3. Chapter 3: Parallel Algorithmic Structures. This chapter is about common patterns in parallel algorithm design, whose general guidelines are presented as a premise. The *embarrassingly parallel* pattern is considered, which applies to cases where the decomposition into independent tasks is self-evident. Two examples are proposed: the Monte Carlo estimation of π and the generation of fractal images. The second pattern being considered is *reduction*, which is illustrated by means of the sum reduction example. The corresponding task graph and data structures are given, considering both distributed and shared memory implementations. As a practical application of the pattern, k-means clustering is presented. Then, the *scan* pattern is illustrated, with particular reference to the implementation of the prefix sum case. The fourth pattern is *divide-and-conquer*, which is discussed and compared to reduction, and used to implement a clever parallel version of the merge sort algorithm. The penultimate pattern is *pipeline*, whose presentation is supported by three examples of increasing difficulty (the last one being pipelined merge sort, which is compared to the divide-and-conquer one). Last but not least, the *data decomposition* pattern is illustrated, which is most suitable when the same operation is performed on elements of a data structure. Four applications are presented: Conway's Game of Life, matrix-vector multiplication, bottom-up dynamic programming and pointer jumping.

4. Chapter 4: Parallel Program Structures. Coming back to the three machine models introduced in Chapter 2 (SIMD, shared memory and distributed memory), this chapter examines how they influence parallel algorithm design and implementation. The concept of *load balance* is introduced and used throughout the chapter. SIMD *strict data parallel* programming, where a single stream of instructions executes in lockstep on elements of arrays, is illustrated by means of row-wise matrix-vector multiplication and other algorithms. The section ends with guidelines to consider when relying on strict data parallel execution on SIMD platforms.

The *fork-join* pattern, which is well-suited for implementing divide-and-conquer algorithms in shared memory systems, is presented by means of two examples, namely the parallel fork-join estimation of π and the parallel fork-join merge sort. Also for this implementation pattern, useful guidelines are provided. Another important pattern for the shared memory model, *parallel loop* is illustrated by means of two examples, i.e., fractals and subset sum. Particular attention is devoted to variable scoping, thread synchronization and thread safety. Next section is about parallel language models that enable specification of *task dependencies*, which allows runtime software to schedule tasks more efficiently than a programmer can. The topic is illustrated with the help of an example, namely parallel subset sum with task dependencies. Then, the *single program multiple data (SPMD)* model is presented, which applies to both distributed and shared memory programming. In particular, it is the basis for parallel programming on GPUs. The presentation of this important model is supported by three examples, i.e., round-robin SPMD parallel fractal generation, SPMD merge sort and reduction on GPU. The penultimate section is devoted to the *master-worker* pattern, which is a common pattern for dynamic load balancing of independent tasks, frequently used with SPMD programming. It is very easy to describe and to use, but does not scale well as the number of workers increases. This aspect is discussed at the end of the section. Finally, the last section focuses on important aspects of distributed memory programming, such as distributed arrays, message passing, local vs. global communication, and the MapReduce pattern.

5. Chapter 5: Performance Analysis and Optimization. The first section discusses the *work-depth* analysis of the task graph. Definitions are given for depth, work and parallelism. These concepts are illustrated by means of some of the examples presented in Chapters 3 and 4. Then, the second section illustrates algorithm performance once tasks are mapped to a computing platform. The most common performance metrics are introduced, i.e., *speedup*, *cost*, *efficiency*, as well as the FLOPS and CUPS throughput metrics. The examples used to illustrate the scan pattern, in Chapter 3, are analyzed using these metrics. A specific subsection is devoted to communication analysis, which is presented by means of the following use cases: reduction, grid-based parallel applications like Conway's Game of Life, distributed arrays. The third section is devoted to barriers to performance, at both the algorithmic and implementation levels: load imbalance, communication overhead and false sharing are discussed. As a viable approach for reducing communication, it is proposed to create hierarchical algorithms to match hierarchical computing platforms. Then, the *Amdahl's law* is presented, which allows one to compute the speedup as a function of the fraction of the execution time of the operations that cannot be done in parallel. The last section is devoted to the use of performance profiling tools.

6. Chapter 6: Single Source Shortest Path. Problems on graphs arise in several different contexts. Solving them efficiently is very important. This chapter focuses on the problem denoted as *single source shortest path* (SSSP), consisting in finding the shortest path between one vertex and all others. The considered algorithms are suitable for all graphs with non-negative real-valued edge weights. In the first section, three sequential algorithms are presented, namely Bellman-Ford, Dijkstra's and delta-stepping. The second section is devoted to parallel design exploration. Parallel versions of the aforementioned algorithms based on task decomposition are illustrated and compared in terms of work, depth and parallelism. Then, it is outlined that distributed memory implementations require the decomposition of

the problem graph, and it is suggested to partition the graph into subgraphs of roughly the same size, maximizing the number of boundary vertices and minimizing the maximum path length from boundary vertices and the interfaces between subgraphs. Finally, three parallel algorithms are detailed, namely shared memory delta-stepping, SIMD Bellman-Ford for GPU and a message passing algorithm.

7. Chapter 7: The Eikonal Equation. Wave propagation is everywhere in nature and inspired many problem solving tools. Level set methods to track moving fronts have been the most successful. This chapter refers to the case where the problem is stationary. In the first section, a numerical framework is introduced for solving the *eikonal equation* and obtaining the plot of the stationary front. Then, two sequential algorithms are presented and discussed, namely the Fast Sweeping Method (FSM) and the Fast Marching Method (FMM). In the second section, parallel design is explored. Two approaches for modifying the sequential FSM algorithm to expose independent tasks are presented and analyzed in terms of performance: relaxing dependencies and reordering sweeps. Two parallel designs of the FMM algorithm are presented, namely the Fast Iterative Method (FIM) and the domain decomposed FMM. In the third section, it is discussed how the aforementioned designs can be developed into algorithms for the three machine models that have been used throughout the book, i.e., SIMD, shared memory and distributed memory.

8. Chapter 8: Planar Convex Hull. This last chapter is devoted to parallel algorithms for computing the *convex hull* of a set of points S, i.e., the intersection of all convex sets containing S. In particular, the planar case is considered. Two sequential algorithms are presented, namely QuickHull and MergeHull. Parallel design of these algorithms is discussed in the second section, starting from the most obvious task decomposition, which is divide-and-conquer. A long subsection is devoted to further improving the parallel MergeHull, resulting in an algorithm with optimal parallelism. The third section is devoted to the presentation and discussion of SIMD QuickHull, shared memory SMPD MergeHull and distributed memory MergeHull.

3 Opinion

The declared goal of *Elements of Parallel Computing* is to guide the reader toward being able to think in parallel, leveraging recent work on patterns in parallel algorithm design to identify the solutions that keep being re-used. The book presents the fundamental concepts of parallel computing not from the point of view of hardware, but from a more abstract view. It takes a language-neutral approach using pseudocode, which can be easily adapted for implementation in relevant languages such as C and C++. The chapters have references to real parallel language models like MPI, OpenMP and CUDA.

I think that students in their final year of bachelor's studies or in their first year of master in computer science or computer engineering would benefit from this book. Personally, in the last two years I have recommended it to the students that attend my course in High Performance Computing (MS in computer engineering). The book provides many examples of how to move from the problem statement to the design of an effective task decomposition and to different algorithms, depending on the machine model (SIMD, shared memory or distributed memory). Importantly, there are homework assignments at the ends of the chapters that may help the students to exercise the knowledge they acquire. Moreover, the book encourages further reading on advanced topics, providing high-quality references.

Performance analysis of parallel algorithms is adequately presented, including the most common performance metrics and some nontrivial examples. If I really have to find a defect in this book is the lack of a discussion on the NC complexity class and on cost optimal NC algorithms, especially those that achieve sublogarithmic time. Maybe in a future edition this gap will be filled.

In conclusion, the book is technically correct and well written, with a fair balance between conciseness and completeness. I think that it may be a constructive and enjoyable experience for the readers that are interested in learning how to design and implement highly efficient parallel programs.