IEEE distributed systems Expert-authored articles and resources ONLINE
August 2008 (vol. 9, no. 8), art. no. 0808-mds2008080001
1541-4922 © 2008 IEEE
Published by the IEEE Computer Society

Education

Think Parallel: Teaching Parallel Programming Today

Ami Marowka • Shenkar College of Engineering and Design

Parallel programming represents the next turning point in how software engineers write software.[1] The transition from sequential computing to parallel computing accelerates every day. Today, low-cost, general purpose desktop parallel machines and parallel programming environments are widely available for students, so parallel computing courses, theory, and practice must be taught as early as possible.

The concurrency revolution calls for modifying almost any course in the computer science curriculum. The prefix "parallel" can be added to any topic: architectures, operating systems, algorithms, languages, data structures, databases, and so on. Some scientists call for integrating parallel and distributed computing topics in computer science curricula.[2] On one hand, changing the entire computer science curriculum at once is a radical step and isn't recommended. On the other, lecturers need to prepare their students for the parallel era. Therefore, I recommend starting with a broad introductory parallel computing course of representative topics, theoretical and practical, that teach what parallel computing is all about.

Here, I describe the framework of an introductory course that I developed over the last two years. This is a mandatory course for third-year B.Sc. students in the software engineering track at the Shenkar College of Engineering and Design, Israel.

Course overview

I designed the course framework with flexibility in mind for theoretical lessons and lab training. Lecturers can choose the topics they believe are more important and decide whether to emphasize lab work or theoretical lectures. The course description is organized by lessons and lab sessions. However, each lesson and lab session can be spread over more than one meeting. In describing the course, I focus on three points: how to present to students that parallel computing is everywhere; how I propose to train the students to think in parallel; and which parallel programming model is used for parallel programming practice.

There are many parallel programming environments on the market today. However, no one textbook covers every topic in parallel computing. I mention here the textbooks and programming environment that I use in my course and explain why I chose them, together with references to other recommended textbooks, Web resources, and programming tools.

The course lessons

Lesson 1: Overview

Students attending the introductory parallel computing course already have knowledge of computer architectures, computer languages, computer networks, operating systems, and databases. However, they aren't aware that most of the topics that they've just studied will be obsolete in a few years. My first goal is to bring to their attention the fact that, behind the scenes, the concurrency revolution is completely reshaping computing technology. To accomplish this goal, I start the lesson with a ten-

question quiz in which I ask the students to answer the questions based both on what they studied in previous courses and on personal knowledge. After discussing their answers, I present mine. At the end of the quiz, they understand that parallel computing is everywhere. Here are the questions and answers:

Q: What does "parallel" mean?
A: According to Webster, parallel is "an arrangement or state that permits several operations or tasks to be performed simultaneously rather than consecutively."

Q: What is parallel computing?
A: Parallel computing is a programming technique that involves using multiple processors operating together on a single problem.

Q: What is a parallel computer?
A: A parallel computer is "a large collection of processing elements that can communicate and cooperate to solve large problems fast."

Q: Is Intel Pentium 1 a parallel processor?
A: Yes.

Q: Is Intel Core 2 Duo a parallel processor?
A: Yes.

Q: Is C a parallel language? Java?
A: C, no; Java, yes.

Q: Is Oracle DBMS a parallel database? DB2? Informix?
A: All of them are.

Q: Is Linux a parallel operating system? NT?
A: Both of them are.

Q: What is the largest computer in the world?
A: IBM Roadrunner: 1 Petaflops, 122,400 cores.

Q: How many processors does the Google engine have? 100? 1,000? 10,000? 100,000?
A: More than 450,000 and counting.

The rest of the lesson is devoted to introducing

- the field of parallel computing;
- the past, present, and future of high-performance computing;
- why parallel computing, and why now;
- the transition to commodity processors;
- Moore's law;
- the power era and multicore processors;
- basic terminology such as latency, bandwidth, von Neumann bottleneck, scalability, speedup, and granularity;
- the top 500 supercomputers; and
- the course topics.

Lesson 2: Think parallel

This lesson is the most important one of the course. Parallelism calls for thinking that isn't intuitive for the human brain. The best way to teach how to think in parallel is to use examples to introduce the data-dependency problem. The examples represent program constructs that can be parallelized efficiently along with others that cannot be parallelized. For each program construct, I ask the students to explain whether they can parallelize it. At the end of this lesson, the students understand that compilers aren't smart enough to parallelize every sequential code and that programmers must understand issues such as true-dependency, anti-dependency, output dependency, loop-carried dependency, signal-wait, and fork-join synchronizations.

Assume that the following code examples run on top of a multiprocessor machine where the underlying physical processors are executed independently and there's no control over the order of execution between processors. For each example, answer the question, "Can the statements executed in parallel?"

• a = 2; b = 3;
A: These statements can be executed in parallel.

• a = 3; b = a;
A: These statements cannot be executed in parallel.

• b = a; a = 4;
A: These statements cannot be executed in parallel.

• a = 2; a = 3;
A: These statements cannot be executed in parallel.

• for (i=0; i<100; i++) b[i] = i;
A: There are no dependences, so iterations can be executed in parallel.

• for(i=0; i<100; i++) b[i] = f(b[i-1]);
A: There's a dependence between b[i] and b[i-1]. Loop iterations are not parallelizable.

• for(i=0; i<100; i++)
. for(j=0; j<100; j++) b[i][j] = f(b[i][j-1]);
A: There's a loop-independent dependence on i and a loop-carried dependence on j. The outer loop can be parallelized; the inner loop can't.

• a = f(x); b = g(x);
A: These statements could have a hidden dependence if f and g update the same variable.

• for(i=0; i<100; i++) b[i] = f(b[indexa[i]]);
A: Cannot tell for sure. Parallelization depends on user knowledge of values in indexa[i] so the user can tell, but the compiler can't.

Lesson 3: Parallel architectures

This lesson focuses on the three popular parallel architectures—shared memory (uniform memory access and symmetric multiprocessing), distributed memory (clusters and network of workstations), and shared distributed (non-uniform memory access)—and various interconnection network topologies that distinguish among them. The presentation includes

- Flynn's taxonomy (multiple instruction multiple data, single instruction multiple data, single instruction single data, and multiple instruction single data),

- vector vs. cache-based parallel machines,
- static interconnection networks (such as Hypercube, Mesh, Torus, and Ring) and their properties,
- dynamic interconnection networks (such as crossbar switch and multistage networks) and their properties,
- the cache coherence problem and its solutions,
- parallel machines-on-a-chip (multicore processors and Polaris test case), and
- examples of real machines from the top 500 list (www.top500.org).

Lesson 4: Design of parallel applications

This lesson follows Ian Foster's four-step design method (partitioning, communication, agglomeration, and mapping).[3] I demonstrate this method with examples for shared-memory and distributed-memory paradigms followed by in-depth analysis and discussion. Examples and very good explanations of this approach can be found in *Parallel Programming in C with MPI and OpenMP* by Michael Quinn.[4] These examples also demonstrate standard patterns of parallel programming such as data parallelism (regular, irregular) and task parallelism (pipelines, task queue).

Lesson 5: Performance and metrics

In this lesson, the students study how to measure and evaluate the performance of parallel applications using metrics and laws including speed-up, Amdahl's Law, Gustafson's Law, and efficiency. In addition, students become familiar with standard de-facto benchmarks such as the NAS parallel benchmarks, Standard Performance Evaluation Corporation benchmarks, and the "Packs" (LinPack, Lapack, and ScaLapack).

Lesson 6: Parallel algorithms

Parallel algorithms could fill an entire course. However, this lesson aims to show students how basic algorithms are parallelized, how to analyze their efficiency and computational complexity, and how to show the impact of the parallel architecture on the logic design and the structure of the algorithms. Recommended algorithms to start with are

- rank sort for shared memory architecture vs. odd-even transposition sort for distributed shared memory architectures,
- quick sort,
- matrix multiplication, and
- Floyd's algorithm.

I recommend scientific algorithms such as iterative solvers, fast Fourier transform and search algorithms for the advanced course. In terms of books, there are several good resources for parallel algorithms.[3–6]

Lesson 7: Programming paradigms

Prior to the first lab meeting, I introduce the student to parallel programming models and paradigms. Students study the shared-memory and message-passing parallel programming models, with emphasis on the single process multiple data programming paradigm. I briefly cover the message-passing model, devoting most of the lesson to the shared-memory model. I focus specifically on the OpenMP programming model, which is also the programming environment students use in the lab assignments. I chose OpenMP because it lets beginners move gradually from serial programming styles to parallel programming. It also helps them create multithreaded applications more easily while retaining the look and feel of serial programming. Two of the books I use have good overview chapters of MPI and OpenMP,[4,6] while others discuss MPI and OpenMP in detail.[7,8] The OpenMP (www.openmp.org) and MPI (www-unix.mcs.anl.gov/mpi) Web sites suggest many tutorials and include links to several other resources.

Lesson 8: Mapping and scheduling

The aim of this lesson is to discuss the changes that must be applied to operating systems to support parallelism. The main topic is scheduling. I explain that parallel OSs must support scheduling in two dimensions (time-sharing and space-sharing) rather than the one-dimensional (time-sharing) scheduling of sequential OSs. I demonstrate the implications of these changes on the OS's performance and efficiency using examples of scheduling algorithms. Other topics include static and dynamic mapping strategies, load balancing, and job scheduling strategies (such as gang scheduling and backfilling). Chapter 4 of *Parallel Programming: Techniques and Applications using Networked Workstations and Parallel Computers* by Barry Wilkinson and Michael Allen is an excellent source for the topics this lesson covers.[5]

Lesson 9: Parallel computation model

This lesson aims to present the efforts of the scientific research community in the last three decades to find a parallel computation model. The lecture starts with the theoretical Parallel Random Access Machine model and continues with a brief overview of more realistic models, such as LogP machine and Parallel Memory Hierarchy. Finally, I concentrate on the Bulk Synchronous Parallel (BSP) model.[9] *Parallel Scientific Computation: A Structured Approach using BSP and MPI* by Rob H. Bisseling presents the model step by step using examples of scientific algorithms. It also contains a software package with all the examples from the book, which students can run and test in the lab.[9]

Lesson 10: Parallel databases

The last lesson is devoted to parallel databases. Parallel databases were the first applications to adopt and integrate parallelism because parallelism is natural to database management systems processing. First, I introduce and compare the databases' architectures (shared-memory, shared-disk, and shared-nothing); then we review different types of DBMS parallelism (intra-operator parallelism, inter-operator parallelism, and inter-query parallelism). I follow this up with an explanation of automatic data partitioning techniques, parallel scans queries, parallel sort, parallel aggregates, parallel join, parallel hash join, parallel query plans, and parallel query optimization.

The lab sessions

Teaching parallel programming without practical work is useless. The lab sessions are devoted to gaining experience in writing parallel code.

Lab I

In the first lab meeting, I describe the lab's hardware and software environments and demonstrate the process of building a simple parallel program that includes configuration, compilation, running, debugging, and evaluation of performance and results correctness.

The lab uses three desktop parallel machines: dual-core machines based on Intel Pentium D and Core 2 Duo processors; and quad-core machine based on Intel Core 2 Quad processors. The students learn about the architectural differences among these machines and the implications of this architecture diversity on the design of the parallel programs for performance portability. On the software side, all the machines run the Windows XP operating system. The students use the OpenMP for C/C++ parallel programming model and two compilers: MS Visual Studio C++ 2005 and Intel C/C++ compiler version 9.1. The debugging tools are Intel VTune, and the threading tools are Intel Thread Profiler and Intel Thread Checker.

Why do I use two compilers? On the one hand, MS Visual Studio C++ 2005 is used more in classrooms, and it's installed in many of the students' home computers. On the other hand, this programming environment doesn't have threading tools for debugging and profiling parallel programs. Such tools are available from Intel, but they're expensive. By using two compilers, students can start lab assignments at home and finish them at the lab with the state-of-the-art Intel debugging and threading tools.

Lab II

In this meeting, I teach how to use debugging and profiling tools. The meeting starts with a short talk about the role of debugging and profiling tools in parallel programming. I emphasize that it's impossible to develop optimized parallel code without good profiling and debugging tools. Our motto is, "A parallel programming developer without debugging and profiling tools is like an astronomer without a telescope or a biologist without a microscope." Once the students are armed with this understanding, I begin to teach them the Intel Thread Profiler tool using the user guide and the examples enclosed in the tool package. I start with the Thread Profiler because it's relatively simple to understand, especially when used for profiling OpenMP programs compiled by the Intel compiler.

The Thread Profiler is a visualization tool that helps programmers

- determine the best sections of code to optimize for parallel performance,
- evaluate scalability across different numbers of processors,
- determine if load imbalance is hurting parallel efficiency,
- locate synchronization constructs that impact execution time, and
- identify and compare the performance impact of different synchronization methods or different algorithms.

With respect to time constraints, I decide whether to continue teaching the VTune performance analyzer and the Intel Thread Checker.

Lab III

For the remainder of the course, all the lab meetings are dedicated to self-instruction using the lab assignments. In these assignments, students parallelize serial programs as they studied in the parallel algorithms lesson. They report on the measured results and the computed metrics such as the algorithms' speedup and efficiency.

OpenMP programming model

Multithreaded programming breaks up an application into subtasks called *threads* that run concurrently and independently. To take advantage of multicore processors, applications must be redesigned so the processor can run them as multiple threads.

OpenMP simplifies the complex task of code parallelization, letting beginners move gradually from serial programming styles to parallel programming. It extends a serial code by using compiler directives. A programmer familiar with a language such as C/C++ needs to learn only a small set of directives. Adding these directives doesn't change the serial code's logical behavior. It only tells the compiler which piece of code to parallelize and how; the compiler handles the entire multithreaded task.

OpenMP uses the fork-join model of parallel execution (see Figure 1). An OpenMP program begins with a single thread of execution called the *master thread*. The master thread spawns teams of threads in response to OpenMP directives, which perform work in parallel. Parallelism is thus added incrementally: the serial program evolves into a parallel one. OpenMP directives in the form of comments in Fortran or C/C++ are inserted at key locations in the source code. The compiler interprets the directives and creates the necessary code to parallelize the indicated regions. The parallel region is the basic construct that creates a team of threads and initiates parallel execution.

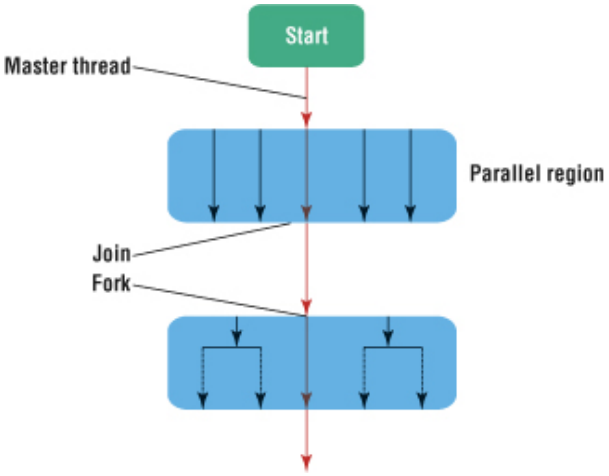

Figure 1. Fork-join programming model of OpenMP.

Most OpenMP directives apply to structured blocks, which are blocks of code with one entry point at the top and one exit point at the bottom. The number of threads created when entering parallel regions is controlled by an environment variable or by a function call from within the program. It's possible to vary the number of threads created in subsequent parallel regions. Each thread executes the block of code enclosed by the parallel region. Moreover, within a parallel region, there can be nested parallel regions, in which each thread of the original parallel region becomes the master of its own thread team (dotted arrows in Figure 1).

Pi program example

The following parallel program computes an approximation to π using numerical integration to calculate the area under the curve $4 / (1 + x^2)$ between 0 and 1 (see Figure 2). The interval [0,1] is divided into `num_subintervals` subintervals of width 1/`num_subintervals`. For each of these subintervals, the algorithm computes the area of a rectangle with height such that the curve $4 / (1 + x^2)$ intersects the top of the rectangle at its midpoint. The sum of the areas of the `num_subintervals` rectangles approximates the area under the curve. Increasing `num_subintervals` reduces the difference between the sum of the rectangle's area and the area under the curve.

```
1. #include < omp.h >
2. float num_subintervals = 10000; float subinterval;
3. #define NUM_THREADS 2
4. void main ()
5. {int i; float x, pi, area = 0.0;
6. subinterval = 1.0 / num_subintervals;
7. omp_set_num_threads (NUM_THREADS)
8. #pragma omp parallel for reduction(+ : area) private(x)
9. for (i=1; i<= num_subintervals; i++){
10.*x = (i - 0.5)  subinterval;
11.area = area + 4.0/(1.0 + x * x);
12.}
13.pi = subinterval * area;
14.}
```

Figure 2. OpenMP program to compute π.

Figure 2 presents an OpenMP program to compute π. The compiler directive inserted into the serial program in line 8 contains all the information needed for the compiler to parallelize the program. `#pragma omp` is the directive's sentinel. The parallel keyword defines a parallel region (lines 9-12) that `NUM_THREADS` threads are to execute in parallel. `NUM_THREADS` is defined in line 3, and the `omp_set_num_threads` function sets the number of threads to use for subsequent parallel regions in line 7. There's an implied barrier at the end of a parallel region. Only the master thread of the team continues execution at the end of a parallel region. The `for` keyword identifies an iterative work-sharing construct that specifies the iterations of the associated loop that will be executed in parallel. The iterations of the for loop are distributed across the threads in a round-robin fashion in order of thread number. The `private(x)` clause declares the variable `x` to be private to each thread in the team. A new object with automatic storage duration is allocated for the construct. This allocation occurs once for each thread in the team.

The `reduction(+ : area)` clause performs a reduction on the scalar variable area with the operator +. A private copy of each variable area is created, one for each thread, as if the private clause had been used. At the end of the region for which the reduction clause was specified, the original object is updated to reflect the result of combining its original value with the final value of each of the private copies using the + operator. The reduction operator + is associative, and the compiler can freely reassociate the computation of the final value. The value of the original object becomes indeterminate when the first thread reaches the containing clause and remains so until the reduction computation is complete. The computation is complete at the end of the work-sharing construct.

The concurrency revolution calls for updating almost each course in the computer science curriculum. This updating will take a few years to be accomplished. Meanwhile, I suggest starting with a broad overview course that encompasses the most significant topics of parallel computing. I am now working on an advanced parallel programming course that includes studying advanced scientific parallel algorithms and new parallel programming models for multicore systems.

References

1. A. Marowka, "Parallel Computing on Any Desktop," *Comm. ACM*, vol. 50, no. 9, 2007, pp. 74–78.
2. M. Paprzycki, "Integrating Parallel and Distributed Computing in Computer Science Curricula," *IEEE Distributed Systems Online*, vol. 7, no. 2, 2006, http://doi.ieeecomputersociety.org/10.1109/MDSO.2006.9.
3. I. Foster, *Designing and Building Parallel Programs*, Addison-Wesley, 1995, www-unix.mcs.anl.gov/dbpp/text/book.html.
4. M.J. Quinn, *Parallel Programming in C with MPI and OpenMP*, McGraw-Hill, 2004.
5. B. Wilkinson and M. Allen, *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers*, 2nd ed. Prentice Hall, 2005.
6. A. Grama et al., *An Introduction to Parallel Computing, Design and Analysis of Algorithms*, 2nd ed., Addison-Wesley, 2003.
7. W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message Passing Interface*, MIT Press, 1999.
8. B. Chapman, G. Jost, and R. van der Pas, *Using OpenMP Portable Shared Memory Parallel Programming*, MIT Press, 2007.
9. R.H. Bisseling, Parallel Scientific Computation: A Structured Approach using BSP and MPI, Oxford Univ. Press, 2004.

Ami Marowka is an assistant professor in the Software Engineering Department of the Shenkar College of Engineering and Design. Contact him at amimar2@yahoo.com.

Cite this article:
Ami Marowka, "Think Parallel: Teaching Parallel Programming Today," *IEEE Distributed Systems Online*, vol. 9, no. 8, 2008, art. no. 0808-o8002.