Parallelization lets applications exploit the high throughput of new multicore processors, and the OpenMP parallel programming model helps developers create multithreaded applications.

PARALLEL COMPUTING ON ANY DESKTOP

By Ami Marowka

"The first law of massive parallelism is the foundation for massive marketing that supports massive budgets that supports the search for massive parallelism," Gordon Bell, 1992 [2].

For many years parallel computers have been used by an exclusive scientific niche. Only rich universities and research institutions backed by government budgets or by multibillion-dollar corporations could afford state-of-the-art parallel machines. Multiprocessor machines are very expensive and demand highly specialized expertise in systems administration and programming skills.

Commodity workstations first appeared in the 1990s. New computer networking technologies allowed the harnessing of tens, and later hundreds, of them together to form clusters of workstations. The "do-it-yourself" Beowulf clusters represented great progress (www.beowulf.org), and many more colleges and universities established parallel computing labs. Beowulf clusters are attractive because they are cost-effective, easy to construct, and scalable. They are built from relatively inexpensive, widely available components; most systems administrators have the skills necessary to install and support clusters. If the processing power requirement increases, the performance and size of a Beowulf cluster is easily scaled up by adding more computer nodes. Beowulf clusters represent the fastest-growing choice for building clusters for high-performance computing and networking. As of September 2006, 361 systems (72%) were categorized as clusters in the list of TOP 500 supercomputers (www.top500.org).

Unfortunately, this achievement did not

illustration by Robert Saunders

increase the adoption of parallel computing. Beowulf clusters and supercomputers assembled from off-the-shelf commodity processors are still expensive, complicated to manage, difficult to program, and require specialized knowledge and skills. The batch processing involved preserves mainframe methods rather than making them more interactive and user friendly.

The computing industry has been ready for the parallel computing era for more than a decade. Most small-to-mid-size organizations use multiprocessor servers; commercial databases (such as Oracle and Microsoft SQL servers) support parallelism; Linux and Microsoft Windows Server System operating systems are multithreaded; and programming languages (such as Java) support multithreaded programming. However, the massive breakthrough of parallel computing many have been waiting for has still not occurred. Two things were missing until 2005: low-cost parallel computers and simple, easy-to-use parallel programming environments. However, they are now available and will change the way developers design and build software applications.

Two complementary technologies bring parallel computing to the desktop. On the hardware side is the multicore processor for desktop computers; on the software side is the integration of the OpenMP parallel programming model into Microsoft Visual C++ 2005. These technologies promise massive exposure to parallel computing that nobody can ignore; a technology shift is unavoidable.

Multicore Processors

Dual-core processors first appeared on the market in 2001 [3]. Chip makers Sun Microsystems and IBM were first; Sun introduced the Microprocessor Architecture for Java Computing, and IBM introduced the POWER4 dual-core processor. Like their predecessors, these processors were expensive and optimized for special-purpose computing-intensive tasks running on high-end servers.

The greatest change in processor architecture came with the dual-core processors that AMD and Intel introduced in 2005. Both were designed for desktop computers and caused a dramatic drop in the price of desktop computers and laptops, as well as of multicore processors. A desktop computer with a dual-core processor can today be bought for less than $500, a price affordable by students and computer science departments alike. Dual-core processors are only the beginning of the multicore-processor era. Chip makers are working on the next generation of multicore processors that will contain four, eight, and 16 cores on a single die for both desktop computers and laptops. The primary consequence is that applications will increasingly need to be parallelized to fully exploit processor throughput gains now becoming available.

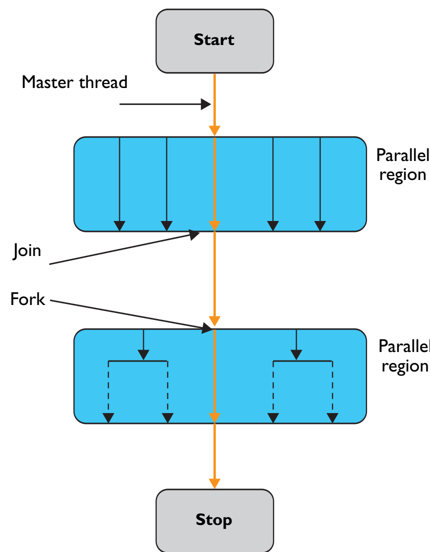

Figure 1. Fork-join programming model of OpenMP.

Unfortunately, writing parallel code is more complex than writing serial code [8]. This is where the OpenMP programming model enters the parallel computing picture. OpenMP helps developers create multithreaded applications more easily while retaining the look and feel of serial programming.

OpenMP Programming Model

Multithreaded programming breaks an application into subtasks, or "threads," that run concurrently and independently. To take advantage of multicore processors, applications must be redesigned for the processor to be able to run them as multiple threads.

OpenMP simplifies the complex task of code parallelization [6], allowing even beginners to move gradually from serial programming styles to parallel programming. OpenMP extends 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 them does not change the logical behavior of the serial code; it tells the compiler only which piece of code to parallelize and how to do it; 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, or "master thread," which spawns teams of threads in response to OpenMP directives that perform work in parallel. Parallelism is thus added incrementally, with the serial program evolving into a parallel program. OpenMP directives are inserted at key locations in the source code. These directives take the form of comments in Fortran or C/C++. The compiler interprets the directives and creates the necessary code to parallelize the indicated regions. The parallel

TO TAKE ADVANTAGE OF MULTICORE PROCESSORS, applications must be redesigned for the processor to be able to run them as multiple threads.

region is the basic construct that creates a team of threads and initiates parallel execution.

Most OpenMP directives apply to structured blocks, or 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 is 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, from within a parallel region, nested parallel regions can exist in which each thread of the original parallel region becomes the master of its own thread team (the broken arrows in Figure 1).

```
1.  #include <omp.h>
2.  float num_subintervals = 10000; float subinterval;
3.  #define NUM_THREADS 5
4.  void main ()
5.  {int i; float x, pi, area = 0.0;
6.    subinterval = 1.0 / num_subintervls;
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.

Pi Program

The parallel program described in the following paragraphs computes an approximation of π using numerical integration to calculate the area under the curve $4/(1+\chi 2)$ 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 $4/(1+\chi 2)$ 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.

Figure 2 is an OpenMP program for computing 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 is to be executed by NUM_THREADS threads 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 is 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 to be executed in parallel. The iterations of the *for* loop are distributed across the threads in a round-robin fashion according to the order of the 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 is 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 may freely re-associate the computation of the final value. The value of the original object becomes indeterminate when the first thread reaches the containing clause, remaining indeterminate until the reduction computation is complete. The computation is complete at the end of the work-sharing construct.

The pi program demonstrates only the basic constructs and principles of OpenMP, though OpenMP is a large and powerful technology for parallelizing applications.

Programmability and Performance

The example involving the pi program shows that with OpenMP, the programmer has little to do when parallelizing serial code. In this case, only two lines are added to the source code. However, the programmer has much to understand, including: the relationship between the logical threads and the underlying physical processors and cores; how

threads communicate and synchronize; how to measure performance in a parallel environment; and the sources of load unbalancing. The programmer must check for dependencies, deadlocks, conflicts, race conditions, and other issues related to parallel programming. Parallel programming is no doubt much more tedious and error-prone than serial programming [9].

The fork-join execution model for OpenMP is simple and useful for solving a variety of problems in large array-based applications. However, many classes of problems demand other types of programming models not currently supported by the OpenMP standard [7]. For example, OpenMP is a shared-data programming model, while many high-performance applications must be run on distributed-shared memory machines and Beowulf clusters. This requirement demands facilities for data placement among processors and threads to achieve data locality absent from the standard. Solutions involve using vendor-specific extensions or learning and implementing sophisticated techniques [4].

Likewise, in its current form, OpenMP does not adequately address task-graph parallelism computing [5]. A variety of applications induce task-graph parallelism with coarse-grain granularity. Task-graph parallelism occurs when independent program parts are executed on different cores based on precedence relationships among the threads. These applications must be exploited to achieve the best possible performance on multicore processors.

The extra development effort and code complexity of parallel programming begs the question: Is it worth the trouble? The good news is that for a variety of applications, it is, because parallelism allows full exploitation of the gains in multicore-processor throughput. Recent benchmark results for multicore-based platforms using OpenMP are very encouraging. For example, the Portland Group produced a version of LS-DYNA compiled for an AMD Opteron processor-based dual-core system using the Portland Group's OpenMP compiler (www.pgroup.com/). This version represents a speed improvement of over 30% compared to the previous best single-chip performance reported. LS-DYNA is an explicit, general-purpose multi-physics simulation software package used to model a range of complex real-world problems (such as crash analysis and metal forming).

More evidence of the scalability potential of dual-core processors has been demonstrated using SPEC OMPM2001 on IBM platforms (www.spec.org/omp/). SPEC OMPM2001 is a standard benchmark for OpenMP-based applications in both industry and research; it uses a set of shared-memory, parallel-processing applications to measure the performance of the computing system's processors, memory architecture, operating system, and compiler. A total of 11 different application benchmarks—covering everything from computational chemistry to finite-element crash simulation to shallow water modeling—are included in the benchmark suite. Comparing the performance of IBM eServer OpenPower 720 with two POWER5 dual processors running eight threads to a system with one POWER5 dual processor running four threads shows a performance gain of 95%.

Conclusion

Parallelism represents the next turning point in how software engineers write software. Today, as general-purpose desktop parallel machines are widely available for the first time, new opportunities are available for many more researchers to generate new parallel concepts and designs worldwide. Parallel computing can even be made available to students in high school and college, small software houses, and small-business start-ups. Many challenges must still be addressed (such as practical parallel-computation models, simpler parallel programming models, and efficient parallel algorithms) [1]. Meanwhile, we must wait and see whether the first law of massive parallelism is ever proved. C

References

1. Asanovic, K. et al. *The Landscape of Parallel Computing Research: A View from Berkeley.* Electrical Engineering and Computer Sciences, University of California, Berkeley. Technical Report No. UCB/EECS-2006-183, Dec. 18, 2006; www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html.
2. Bell, G. Massively parallel computers: Why not parallel computers for the masses? In *Proceedings of the Fourth Symposium on the Frontiers of Massively Parallel Computers* (McLean, VA, Oct. Oct. 19–21). IEEE Press, Los Alamitos, CA, 1992, 292–297.
3. Geer, D. Chip makers turn to multicore processors. *IEEE Computer 38,* 5 (May 2005), 11–13.
4. Marowka, A., Liu, Z., and Chapman B. OpenMP-oriented applications for distributed shared memory architectures. *Concurrency & Computation: Practice & Experience 16,* 4 (Apr. 2004), 371–384.
5. Marowka, A. Extending OpenMP for task parallelism. *Parallel Processing Letters 13,* 3 (Sept. 2003), 341–352.
6. OpenMP Architecture Review Board. OpenMP Application Program Interface, Version 2.5 (May 2005); www.openmp.org/.
7. Skillcorn, D. and Talia, D. Models and languages for parallel computation. *ACM Computing Surveys 30,* 2 (June 1998), 123–169.
8. Sutter, H. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal 30,* 3 (Mar. 2005), 292–210.
9. Sutter, H. and Larus, J. Software and the concurrency revolution. *ACM Queue 3,* 7 (Sept. 2005), 54–62.

Ami Marowka (amimar2@yahoo.com) is an assistant professor in the Department of Software Engineering of Shenkar College of Engineering and Design, Ramat-Gan, Israel.

© 2007 ACM 0001-0782/07/0900 $5.00