Using the N-Body Problem to Engage Undergraduates in Parallel Programming

Ben White, Miriam Robinson, Chris Mitchell, Jens Mache
Department of Mathematical Sciences, Lewis & Clark College, Portland, Oregon, USA
{benwhite, miriamr, chrism, jmache}@lclark.edu

Abstract— *With the rise of multicore hardware, it is increasingly apparent that parallel programming is overdue for integration into undergraduate curricula. While some institutions are trying this, their assignments are often not interesting or not engaging, and the difficulty of low-level parallel languages might be overlooked. We have created a fun and compelling problem for students to parallelize: an N-body simulation. Although our example is under development, we suggest that more undergraduate institutions implement similar examples in the classroom.*

Keywords: parallel computing, n-body simulation, computer science education

1. Introduction

Parallel programming is often given cursory treatment in current undergraduate curricula, even though it is now necessary for any computer scientist who wishes to write programs that properly utilize multicore machines. The subject of parallel programming has been traditionally reserved for students pursuing graduate-level studies, but the rising importance of parallelism brings the need to introduce parallel concepts earlier on in computer science education. We aim to do this by creating exercises and materials that are targeted for undergraduate students.

Background research on recent efforts for bringing parallelism to the undergraduate classroom revealed two trends. First, assigned programming exercises were not very compelling. Often, these assignments included overused problems like matrix multiplication which are unlikely to catch student's interests and do not encourage experimentation. A student is likely to just "try to get the assignment done" if they do not care about it. Second, we noticed that many introductions to parallel programming involved lower-level abstractions like POSIX threads (Pthreads) [1] or Message Passing Interface (MPI) [2], which may obfuscate the higher-level parallelism concepts with a swath of low-level details. Students' fundamental understanding of parallelism may suffer as a result of being overwhelmed by these details.

This paper describes an exercise in which students use OpenMP to parallelize a graphical implementation of the N-body problem written in C. The N-body problem is an $O(N^2)$ algorithm for computing the motion of N bodies under the influence of physical forces. In addition to being visually interesting program, the naive algorithm for computing the forces is easy to understand and computationally expensive, making it a good fit for teaching parallel concepts to undergraduates. OpenMP [3] is an industry standard API for shared-memory parallel programming that is high-level and easy to grasp. Adding OpenMP directives to one's code causes the compiler to emit programs that will make use of threads to complete their work.

2. Related Work

One college or university that has already introduced parallel programming to undergraduates is the University of San Francisco, which has a lower-division elective course in the subject [4]. Students are taught the C programming language during the first month of the course, and then are taught parallel programming using MPI, Pthreads, and OpenMP. The course also uses an N-body simulation, but devotes a large amount of time to replacing the naive $O(N^2)$ implementation with a faster $O(N \log N)$ implementation that uses the Barnes-Hut algorithm [5]. Using Barnes-Hut is a valuable lesson in algorithm optimization, but not particularly useful for teaching parallel concepts to students.

At the University of Washington, a three-week introduction to parallelism and concurrency is taught in a sophomore level data structures course [6]. The students in this course learn about parallelism with Java's built-in Thread library. Since these threads have a large amount of overhead, they are ill-suited for small computations and do not offer much speedup for the types of programs that students are likely to write. Even though C and C++ offer fast threading that may provide students with a more satisfying speedup, at the University of Washington, students are already familiar with Java, so teaching the course in Java enables more time to be spent teaching parallelism concepts instead of a programming language.

A breadth-first course in parallel programming for undergraduates was introduced at Sonoma State University using three high-level languages: OpenMP, Intel Threading Building Blocks (TBB) [7], and CUDA [8], [9]. The labs and assignments used in this course were standard problems such as finding a maximum value in an array or incrementing all the elements of an array. While these are simple and easy to understand problems, they may be better suited for during a lecture instead of as a homework assignment.

At Lewis & Clark College, Mitchell *et al.* created a lab exercise for introducing the CUDA parallel programming language [10]. The exercise asks students to use CUDA to parallelize an animated implementation of John Conway's Game of Life. The problem was chosen for its strength in providing visual feedback and for its more recreational feel when compared to typical numerical problems. Because CUDA requires an NVIDIA graphics processing unit (GPU) to run, it may be impractical to teach this type of parallelism in an undergraduate environment that is not equipped with NVIDIA GPUs.

This paper builds on these works by combining two qualities that seem to be useful for teaching this subject: interesting and engaging examples, and languages that have high-level and easy to learn abstractions for parallelism.

3. The N-body Problem

The N-body problem simulates the movement of a collection of N bodies under the influence of physical forces. In our case, we simulate gravity in two dimensions. Each body in the system moves according to the net force exerted on it by the other $N-1$ bodies, yielding a computational complexity of $O(N^2)$. Our serial approach to the n-body problem uses numerical integration to estimate the positions of the bodies at regular time steps throughout the simulation. To turn the n-body problem into an assignment, after learning some OpenMP, students may be given our serial implementation and challenged to parallelize it with OpenMP directives to make it run as fast as possible. At a very high level, our serial implementation works like this:

```
 1 For each step:
 2    draw the bodies
 3    move the bodies one time step
```

Updating the position of the bodies for each time step involves using the positions and masses of the bodies to calculate the gravitational forces they exert on each other, and then using these net forces and body positions and body velocities to update the bodies with new positions and velocities. Pseudocode for progressing the simulation one time step looks like:

```
 4 For each body q:
 5    Initialize net force on q to zero
 6    For every body k besides q:
 7        Calculate the force k exerts
              on q
 8        Add this force to the net
              force on q
 9 For each body q:
10    Update q's position and velocity
```

Note the doubly-nested loop that calculates the net forces in the system and gives the algorithm an $O(N^2)$ runtime. The gravitational force exerted on a body q by another body k is given by:

$$F_{qk} = \frac{G m_q m_k}{r_{qk}} \tag{1}$$

where G is the gravitational constant, m_q and m_k are the masses of bodies q and k, respectively, and r_{qk} is the distance between the two bodies.

4. N-Body as an Assignment

We propose providing finished serial code and challenging the students to add OpenMP directives to gain the best speedup they can. Providing a serial implementation allows students to focus on the parallel task at hand and not having to worry about implementing the physics. Instructors could also challenge students to produce the fastest version in the class.

There are a number of ways a student might approach parallelizing the code.

The most obvious section to parallelize is the nested loop which computes the forces on each particle. This is done by adding a compiler directive for OpenMP in front of the loop on line 4 (missing proper private/shared clauses):

```
# pragma omp parallel for
    num_threads(n)
```

Here `n` is an integer indicating the number of threads in the thread pool. Using the `num_threads` clause allows the user to vary the number of threads to test the performance of the program.

The exercise affords a little bit of experimentation because there are a few ways to do this parallelism incorrectly and also correctly. For a complete serial implementation and for more information on parallelizing this code, please refer to our website [11].

5. Discussion

5.1 Potential Difficulties

Despite the apparent simplicity of inserting OpenMP directives to add parallelism, a careless approach will cause the student to make correctness errors. These mistakes will encourage them to reason about race conditions and variable scope (private/shared) to understand why their program has failed and to determine what works and what doesn't.

The first potential problem is that of shared and private variables. For example, a naive approach to the exercise is to simply insert a `parallel for` directive before the outer loop. However, since OpenMP automatically treats any variables declared outside of an OpenMP directive as shared, those variables will be visible and accessible by all threads. This can lead to race conditions between iterations of the loop and cause incorrect answers. A simple fix is to declare all critical variables as private. For example, if we have a variable x used inside the for loop, but it was declared outside the loop, we can make it private by adding the private clause:

```
# pragma omp parallel for private(x)
```

Another way to avoid this type of race condition is to implicitly set variables as private for OpenMP by placing their declaration inside the for loop.

A second problem students may come across is the issue of handling output. Since I/O is not inherently thread-safe, different threads attempting to print simultaneously can cause unexpected output. For example, two threads participating in a parallel loop with each thread trying to print `Answer from Thread N\n` might produce this jumble:

```
> Answer fromAnswer from Thread 2
>  Thread 1
>
```

In order to avoid this, students should ensure the print commands are serialized by placing them in code blocks with the `# pragma omp critical` directive. The `critical` directive ensures only one thread is executing the code in the block at any time, preventing simultaneous printing to the screen.

5.2 Expected Performance

We timed the N-Body program on an 8-core Sun server (with 64 hardware threads) with varying numbers of threads, 1000 bodies, and with termination occurring after 100 frames. The data we gathered should serve as a guide for the kind of performance increase students can expect to see. Table 1 shows the speedup of the parallelized force calculation using the OpenMP `parallel for` pragma. Speedup was defined as

$$S = \frac{T_1}{T_p}, \tag{2}$$

where T_1 is the execution time of the sequential program (one thread) and T_p is the execution time of the parallel program. The number of threads was controlled by varying the `num_threads` clause, and the runtimes were gathered by using the UNIX `time` command. The loops equivalent to those in the pseudocode on lines 4 and 9 were parallelized with a `parallel for` pragma.

6. Conclusion

While we have not classroom-tested this assignment yet, we anticipate that this example will be engaging and useful for students not interested in typical problems such as matrix multiplication. In addition to being a fun example, it is also useful in situations where students have diverse backgrounds and interests that may not include the usual computational mathematics problems. The usefulness of this paper should not be limited to implementing just the N-body problem either; we hope that other instructors will create their own interactive and interesting examples for their students.

Table 1: Speedup of the parallelized force calculation using the OpenMP `parallel for` pragma.

Number of Threads	Speedup
2	2.00
4	3.99
8	7.93
16	14.24
32	21.80
64	26.72

7. Acknowledgements

Partial support for this work was provided by the National Science Foundation's "Transforming Undergraduate Education in Science, Technology, Engineering and Mathematics (TUES)" program under Award No. 1044932, by the John S. Rogers Science Research Program and by the James F. and Marion L. Miller Foundation. We would also like to thank Quentin Froemke and Jingke Li.

References

[1] B. Barney. (2012) POSIX Threads Programming. [Online]. Available: https://computing.llnl.gov/tutorials/pthreads/
[2] (2012) The Message Passing Interface (MPI) standard. [Online]. Available: http://www.mcs.anl.gov/research/projects/mpi/
[3] L. Dagum and R. Menon, "OpenMP: an industry standard API for shared-memory programming," *Computational Science Engineering, IEEE*, vol. 5, no. 1, pp. 46 –55, 1998.
[4] P. Pacheco. (2012) Computer Science 202, Introduction to Parallel Computing. [Online]. Available: http://cs.usfca.edu/∼peter/cs220/
[5] J. Barnes and P. Hut, "A heirarchical O(NlogN) force-calculation algorithm," *Nature*, vol. 324, no. 6096, pp. 446–449, 1986.
[6] D. Grossman. (2012) Sophomoric Parallelism and Concurrency. [Online]. Available: http://www.cs.washington.edu/homes/djg/teachingMaterials/spac/
[7] (2012) Intel Threading Building Blocks (TBB). [Online]. Available: http://threadingbuildingblocks.org
[8] (2012) CUDA Zone. [Online]. Available: http://developer.NVIDIA.com/category/zone/cuda-zone
[9] S. Rivoire, "A breadth-first course in multicore and manycore programming," in *Proceedings of the 41st ACM technical symposium on Computer science education*, ser. SIGCSE '10. New York, NY, USA: ACM, 2010, pp. 214–218.
[10] C. T. Mitchell, J. Mache, and K. L. Karavanic, "Learning CUDA: lab exercises and experiences, part 2," in *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, ser. SPLASH '11. New York, NY, USA: ACM, 2011, pp. 201–202.
[11] J. Mache. (2012) Teaching parallel computing with higher-level languages and activity-based laboratories. [Online]. Available: http://lclark.edu/ jmache/parallel