Student Outcomes in Parallelizing Recursive Matrix Multiply

Chris Fietkiewicz
Electrical Engineering and Computer Science Department
Case Western Reserve University
Cleveland, OH 44106 USA
001-216-368-8829
chris.fietkiewicz@case.edu

ABSTRACT
Students in a course on high performance computing were assigned the task of parallelizing an algorithm for recursive matrix multiplication. The objectives of the assignment were to: (1) design a basic approach for incorporating parallel programming into a recursive algorithm, and (2) optimize the speedup. Pseudocode was provided for recursive matrix multiplication, and students were required to first implement a serial version before implementing a parallel version. The parallel version had the following requirements: (1) use OpenMP to perform multithreading, and (2) use exactly 4 threads, where each thread computes one quadrant of the array product. Using a class size of 23 students, including undergraduate and graduate, approximately 70% of the students designed valid parallel solutions, and 13% achieved the optimal speedup of 4×. Common errors included recursively creating excessive threads, failing to parallelize all possible mathematical operations, and poor use of compiler directives for OpenMP.

Categories and Subject Descriptors
K.3.2 [**Computers and Education**]: Computer and Information Science Education - Computer Science Education, Curriculum

General Terms
Algorithms, Performance, Design.

Keywords
Parallel programming, OpenMP, recursion, matrix multiply.

1. INTRODUCTION
Matrix multiplication is a common application in high performance computing and is often used for benchmarking in distributed systems. The standard iterative, loop-based algorithm is easily parallelized, and we have previously used this as an elementary application of multithreaded programming in a course on high performance computing. Alternatively, a recursive approach serves as a basis for algorithms that are faster than the $O(N^3)$ time of the standard iterative algorithm [1]. We introduced recursive matrix multiply as an exercise for designing a parallel program and to have students use an application that contrasts with a previous experience using the standard iterative, loop-based algorithm. Certain constraints were given regarding the parallelization such that the solution would be different from common, public sources.

DOI: https://doi.org/10.22369/issn.2153-4136/10/1/4

2. METHODS
The class was taught during the Spring semester of 2018 at Case Western Reserve University in Cleveland, Ohio. The total enrollment was 23 students, including undergraduate, graduate, and non-degree students. Table 1 shows the distribution of students by level, including subcategories for undergraduate and graduate students. Graduate students include Ph.D. and Master's. Undergraduates include juniors (3rd year) and seniors (4th year). Results also include one student who was of non-degree status. Though the course had been offered twice before, this was the first time that the recursive algorithm was included in the content. Survey data was collected to determine whether students had prior experience with C programming and multithreading, and this data was considered with regard to student outcomes.

Table 1. Distribution of students by level.

Level	*Number*	*Portion*
Ph.D.	4	17%
Master's	6	26%
Senior	10	44%
Junior	2	9%
Non-degree	1	4%
Total	23	100%

Prior to the assignment used in the present study, students had already completed lectures and assignments on parallelizing the standard iterative matrix multiply using the C language. Coverage of the standard algorithm included techniques for cache optimization, multiprocess programming using *fork*(), and multithreading using OpenMP. To prepare students for programming the recursive algorithm, lecture coverage included three components: (1) a mathematical definition of the algorithm, (2) a pseudocode implementation, and (3) a primer on coding and debugging the primary recursive function.

Recursive matrix multiply is defined [1] as

$$\boldsymbol{C} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \boldsymbol{A} \times \boldsymbol{B} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$\boldsymbol{C} = \begin{bmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{bmatrix}$$

where C_{ij} is a subarray of $\boldsymbol{C}$ with quadrant indexes *i* and *j*. As part of the lecture, students were asked to perform a recursive multiplication by hand using arbitrary arrays of size 4×4. In a separate lecture, a pseudocode implementation was given for a function *multiply*(*A, B, C, size*), where *A, B,* and *C* are square arrays and *size* is the number of rows (and columns) of the arrays. The pseudocode for a serial algorithm was given as shown below:

1. Base case: if *size* = 1, then $C_{11} = A_{11}B_{11}$.
2. Otherwise: use a temporary array *T* and compute the following:
   - *multiply*(A_{11}, B_{11}, C_{11}, *size* / 2)
   - *multiply*(A_{11}, B_{12}, C_{12}, *size* / 2)
   - *multiply*(A_{21}, B_{11}, C_{21}, *size* / 2)
   - *multiply*(A_{21}, B_{12}, C_{22}, *size* / 2)
   - *multiply*(A_{12}, B_{21}, T_{11}, *size* / 2)
   - *multiply*(A_{12}, B_{22}, T_{12}, *size* / 2)
   - *multiply*(A_{22}, B_{21}, T_{21}, *size* / 2)
   - *multiply*(A_{22}, B_{22}, T_{22}, *size* / 2)
   - $C_{11} = C_{11} + T_{11}$
   - $C_{12} = C_{12} + T_{12}$
   - $C_{21} = C_{21} + T_{21}$
   - $C_{22} = C_{22} + T_{22}$

Due to the complexity of the algorithm, another lecture was given as a primer on coding and debugging the primary recursive function. Students were advised to use a reduced algorithm in order to test proper indexing and use of the temporary array *T*. As a test, it was recommended that students first compute only one of the four quadrants. An example was given in class for computing only C_{21} as follows:

$$\boldsymbol{C} = \begin{bmatrix} 0 & 0 \\ A_{21}B_{11} + A_{22}B_{21} & 0 \end{bmatrix}$$

An example of using this reduced algorithm was calculated by hand in class using the same 4×4 array that was previously demonstrated. Following the hand calculation, a discussion was provided regarding how to properly associate the quadrant indexes (*i,j* = [1, 2]) with array indexes in the range 0 to *size* – 1.

For the assignment, students were required to use the C language to implement separate serial and parallel versions. The parallel version had the following requirements: (1) use OpenMP to perform multithreading, and (2) use exactly 4 threads, where each thread computes one quadrant of the final array product. The first requirement was given because students had already completed a previous assignment in which the standard iterative algorithm was parallelized using OpenMP. The second requirement differs from other common approaches in which each call to *multiply*() is performed on a separate thread. We required exactly 4 threads, with one for each quadrant, for two reasons. First, this unusual approach was intended to discourage the use of publicly available source code. Second, it has the advantage of being appropriate for development on commonly available 4-core CPUs.

For the parallelized implementation, students were required to report the speedup which was defined as the ratio of the serial run time to parallel run time. A solution was considered valid if it achieved a speedup greater than 1×, up to 4×, as compared to the serial version. Students were not told in advance what speedup ratio could be expected.

3. RESULTS

All students achieved a successful serial implementation, suggesting that adequate coverage was provided in class regarding the algorithm and coding suggestions. Therefore, the outcomes were evaluated according to five categories of reported speedup values: (1) approximately 4×, (2) greater than 1× but less than 4×, (3) no speedup or approximately 1×, (4) decrease in speed (less than 1×), and (5) not applicable (N/A). The N/A category represents students who reported unreasonable speedup values above 4× that were due to errors. The results given in Table 2 show that 70% of the students successfully achieved a speedup in categories (1) or (2).

Table 2. Student outcomes categorized by speedup values.

Speedup	*Number*	*Portion*
4×	3	13%
Less than 4×	13	57%
1×	1	4%
Less than 1×	3	13%
N/A	3	13%
Total	23	100%

In analyzing the student outcomes, we considered the background experience of the students. In addition to the academic levels listed in Table 1, survey data showed that 57% of the class (*n* = 13) had prior experience with C and multithreaded programming prior to taking the course. Prior to the present assignment, however, all students had completed five previous programming assignments. All previous assignments required C programming, and two of the previous assignments required the use of multithreaded programming with OpenMP.

We determined that experience prior to the course was not a determining factor for success in the present assignment. With regard to academic experience, valid solutions were obtained by all students in the two lowest experience levels: undergraduate juniors and non-degree. Invalid solutions occurred for students in the three highest levels: Ph.D., Master's, and undergraduate seniors.

We also concluded that outcomes did not depend on prior experience with C and multithreaded programming. For students with valid solutions (*n* = 16), only 50% of those had prior experience (*n* = 8). For students who did not have valid solutions (*n* = 7), 71% of those had prior experience using C programming (*n* = 5). Furthermore, of the students that lacked prior experience (*n* = 10), 80% of them had valid solutions (*n* = 8), including one student with an optimal speedup of 4×. This high success rate among students without previous experience suggests that lectures and previous class assignments were appropriate in preparing students.

Among students with valid solutions, two particular factors affected the speedup value. One significant factor was the specific implementation for the temporary array *T* that was used for intermediate calculations. We observed three basic approaches to implementing the use of *T*: (1) allocating it privately within the *multiply*() function, (2) allocating it as a single, full-sized array that was shared among all threads, and (3) eliminating *T* altogether by performing addition in the recursion base case. Approach (1) was the most common and generally resulted in speedup values from

2× to 3×. All students who achieved optimal speedup of approximately 4× used approach (3).

A second factor affecting the speedup in valid solutions was whether the student actually parallelized the addition step $C = C + T$. Several students neglected to include this operation in the separate threads. While they still obtained a speedup, this error significantly reduced the speedup from what was otherwise possible. Interestingly, one student implemented a recursive version of the addition step, as opposed to the ubiquitous use of for-loops for this purpose. Unfortunately, this appears to have significantly limited the speedup value.

Among the 30% of students that did not have valid solutions (n = 7), there were many different types of errors. The expected solution, as explained in the assignment instructions, required the creation of 4 threads at the beginning of the program, with each thread making an initial call to *multiply*(). In some cases, students mistakenly created the 4 threads in the recursive function itself, resulting in an excessive number of threads. For large array sizes, the effect was a significant increase in run time. Another error that was observed was poor use of OpenMP compiler directives. In previous assignments using OpenMP, students were required to compare the clauses "parallel" and "parallel for". In the present assignment, some students attempted to use the "parallel for" clause with a for-loop to create the 4 threads. In all of these cases, the students introduced some type of error with regard to the number of threads that were created or the manner in which *multiply*() was called.

Some students (n = 3) reported unreasonably high speedup values greater than 4× and were categorized as N/A in Table 2. Each of these cases involved unique logic errors in the parallel implementation. We do not report the specific errors here because they were unique to each student. However, they all can be described as parallelization errors in which less than 100% of the required calculations were actually performed. Additionally, another common factor in these cases was that students apparently did not test their programs using appropriate array sizes. Their programs actually produced correct results for sizes up to N = 4, which happens to immediately lead to the recursion base case after the recursive subdivision into separate quadrants. However, their programs failed for N = 8 and higher.

In requiring students to first implement a serial version, we intentionally gave them complete freedom in how to associate the quadrant indexes (i,j = [1, 2]) with array indexes in the range 0 to $size$ – 1. An interesting observation is that several students included mechanisms to preserve the quadrant indexing from the original mathematical formulation. It is noteworthy that all of these attempts were successful in the serial version, but some students' designs involved inefficiencies that limited the parallel version for large arrays.

4. DISCUSSION

The assignment used in the present study was our first attempt at requiring students to parallelize recursive matrix multiply. Our intent was to provide experience in designing a parallel program and to use an application that contrasts with a previous experience using an iterative, loop-based algorithm. We consider the requirements for parallelization to have been straight forward. We conclude that the complexity of the algorithm posed the most significant challenge in understanding how to apply parallel programming techniques. It is important to consider the students' backgrounds, and our survey data suggests that appropriate preparation was given to the students prior to the assignment. This preparation included previous lectures and programming assignments on the topics of iterative matrix multiplication, C programming, and multithreaded programming with OpenMP. Additionally, the lecture on debugging may have also been important. Overall, we consider the assignment to have been successful in challenging students to work with an unusual application using familiar techniques.

In the future, we hope to expand the assignment to require a more detailed efficiency analysis. In the present study, we focused exclusively on the students' ability to implement valid parallel solutions with at least some amount of speedup. However, we did not require students to analyze and report on the efficiency of smaller components of their implementation. Many students casually accepted less than optimal speedup values, suggesting that it was due to unavoidable issues, such as unknown aspects to using recursion. In general, students did not analyze the efficiency of separate components of the algorithm. For example, the utilization of the T array involved a large amount of variability in students' results. Additionally, several students did not parallelize the summation of the C and T arrays. In future versions of the assignment, we will consider adding a requirement to use multiple approaches for comparison.

5. ACKNOWLEDGMENTS

This work made use of the High Performance Computing Resource in the Core Facility for Advanced Research Computing at Case Western Reserve University.

6. REFERENCES

[1] Weiss, M. 2012. *Data Structures and Algorithm Analysis in Java (3rd Edition)*. Addison-Wesley, Boston, MA.