A Multi-Semester, Co-Teaching Approach to Parallel Programming

Mike Morris and Karl Frinkle

Southeastern Oklahoma State University, Durant, Oklahoma, U.S.A.
`mmorris@se.edu, kfrinkle@se.edu`

Abstract

This paper describes a successful addition of high performance computing (HPC) into a traditional computer science curriculum. The approach incorporated a three-semester sequence of courses emphasizing parallel programming techniques, with the final course focusing on a research-level mathematical project that was actually executed on a TOP500 supercomputer. A group of students with very varied programming backgrounds participated in the program.

Contents

1 Introduction

Many baccalaureate computer science degree programs require very little or no course work in parallel programming, usually because the accreditation standards have been in place for years and are slow to change. This was indeed the case at Southeastern Oklahoma State University which is a four-year, regional liberal arts university located in Durant, Oklahoma, USA.

Mike Morris, a computer science faculty member, teamed with Karl Frinkle of the mathematics faculty, and developed a method of offering HPC/parallel computing components to the program. A three-course sequence was created that centered on parallel programming and these courses were offered as electives in the form of special seminars. This required no cumbersome administrative program changes and fit well with the existing curricula.

A decision was made early to require only a fundamental amount of programming experience so the courses could be offered to more students, particularly in non-CS disciplines such as mathematics. A first course in C or C++ was sufficient for enrollment.

Because the material was new to the program, interest was high and a course demographic naturally assimilated into three groups which were (1) upper-level CS majors, (2) CS majors with only the first programming course in their portfolio, and (3) mathematics majors with at least one semester of programming.

Based upon the success of these efforts, the investigators are planning to include students in other programs, such as physics, biology, finance and other data-intensive fields. This paper details the specifics of the courses and directions for enhancement of the parallel programming initiative.

2 Requirements and Support

The minimal hardware for a project of this scope is some sort of cluster[9]. This is a vague statement due to the fact that literally hundreds of thousands of clusters of varied sizes and power have been built over the years.

We began by visiting our university's collection of discarded and out-of-date equipment that was waiting for disposal. The equipment manager graciously gave us six retired desktops that were Pentium II powered and we were also given a complement of NICs and peripherals to connect them all together. It should be noted that actual supercomputing power was not needed for our courses. A proof of the concept and means of demonstrating the actual operation of a supercomputer was all that was needed to teach HPC parallel programming concepts.

In 2010 we were awarded a LittleFE machine[5]. The LittleFE machine gave our program credibility by having a professional custom-designed computer that was accepted as a bonafide example of a small supercomputer. It was sleeker and more powerful than the device we cobbled together earlier, but in actuality was very similar to our first effort.

In summary, for a meaningful sequence of HPC courses, a cluster of almost any sort will suffice. As for software, we originally used a version of BCCD (Bootable Cluster CD)[2] and in fact, a customized version of BCCD is shipped with the LittleFE. The LittleFE BCCD is a modified version of Debian Linux.

HPC course instructors have a plethora of support available to enhance their curricula. Supercomputing centers worldwide are sprouting up at major universities and many of them will share expertise and run-rime on their machines.

We sought out the services of OSCER (Oklahoma Supercomputing Center for Education and Research)[7]. "Boomer" is the name of OSCER's latest supercomputer. When it came online, it was the 221st most powerful computer in the globally acclaimed TOP500 list[8] founded by Erich Strohmaier. The ranking is based upon the Linpack benchmark[4]. Other professors introducing parallel programming concepts most likely can easily find a supercomputer at a university nearby that will offer similar assistance.

When one of our better programs has been debugged on our little cluster and seemed to be working as intended, we next ran it on Boomer. Students were amazed when something as seemingly simple as matrix multiplication could rack up several CPU-years of runtime in a few hours. The alliance we have formed with OSCER has given our project a considerable amount of respect and makes the students aware that they are doing work at the forefront of the current technology.

3 The Courses

We introduced a three semester program in parallel programming starting in the Spring of 2012 with a course titled Parallel Programming, followed by CUDA Parallel Programming in the Fall of 2012 and concluding with High Performance Computing in the Spring of 2013. This sequence was intended to immerse the student in many important aspects of high performance computing, such as, but not limited to, workload distribution, communication between nodes, network topology, GPU versus CPU computations, and research level programming.

3.1 Course One - Parallel Programming

The first course in the sequence, Parallel Programming, required as prerequisite only a single semester of C programming, or equivalent. Any further C programming skills required for the semester were taught as needed. The main theme of this semester was matrix multiplication. We considered the scenario where two matrices were large enough that they would not fit into the memory of a standard PC, with the further operating restriction that at most one row or column of each matrix could be stored in memory at any given time. The matrices were stored in binary files to minimize space requirements, so students had to learn about binary file structures reading, writing and seeking. One of the important consequences of the binary file structure, and the attempt at minimizing file sizes, was that the matrices were expressed as one-dimensional arrays. In terms of indexing, this required a lesson on expressing matrix multiplication based on one-dimensional array representations of the matrices. Refer to Appendix A for a sample of the discussion on matrix multiplication given to students in this course to help them go from two-dimensional matrix multiplication to the corresponding one-dimensional array multiplication.

The first program the students were required to write was a single process (non-MPI) program which performed matrix multiplication, loading only one row of the first matrix, and one column of the second matrix, and storing the result in a new binary file. To verify that the program worked, matrices of small dimension were generated, and a small matrix display program was created to display the contents of a binary matrix file. After the program was verified to work on small matrices, timed runs on successively larger dimension matrices were recorded to study the behavior of a standard matrix multiplication method in terms of the dimensions of the matrices.

By the time a single process matrix multiplication program was running smoothly and the results were analyzed, over a quarter of the semester had passed. During this time, an introduction to MPI was covered. Emphasis was placed on communication protocols. Complicated versions of 'Hello World' were made, unbeknownst to the students that the communication framework in these programs would be useful in several variants of MPI based matrix multiplication programs.

The first MPI based matrix multiplication program consisted of each node picking an entry in the resulting matrix and then locating and storing the corresponding row and column needed in local memory, performing the correct algebraic steps, and then storing the result in the correct entry of the product matrix. Since the program was written on a LittleFE[5] unit, only two processes had direct access to the hard drive, the other processes had to go through the ethernet to get access to the binary matrix files. After analysis of run times versus dimensions, and determining the workload per node, it became apparent that the head node, which has the hard drive on its motherboard, did 90% of the work, and the MPI read/write commands, along with the other MPI file sharing commands, were very slow on non-head nodes. This was an excellent example that enforced the students' awareness of the importance the role of network topology plays in performance.

After students became aware of the network topology restrictions, emphasis was placed on only the head node, which hosted two processes, performing operations requiring reading and writing to disk. This made communication and data transfer between the head node and slave nodes more important in the coding process, and many different approaches were taken by the students. Some ideas for communication structures were worked out in the classroom, with students playing the roles of head and slave nodes to see if any bottlenecks or other problems could be found. This was a very hands-on and beneficial exercise. On several occasions, problems were found in communication strategies.

We, as instructors, had a reasonably streamlined program which we benchmarked with increasing larger matrices. The times we recorded were used as a goal for our students to try to improve upon. Students worked in groups, wrote up plans on how to improve performance on their existing attempts by minimizing communication between nodes while maximizing computations per node per communication cycle. By the end of the semester, several groups' attempts came close to reaching our benchmarks and the students learned a great deal about the intricacies of parallel programming.

3.2 Course Two - CUDA Parallel Programming

For the second semester of parallel programming, CUDA Parallel Programming, our emphasis was on choosing projects for which CUDA would be an asset. The majority of students in this class were in the first semester course, and those who were not were paired with students which were. This semester focused more on groups working on semester long projects. The class was broken up into three groups, decided by the students, based on three projects. The projects were (1) multiplying very large numbers, (2) determining primality of a large number, and (3) simple password cracking. Each of these three projects involved repeated arithmetic operations that were perfect for GPU algorithms. For the multiplication of large numbers, the second group need to perform repeated multiplications of one number with large groupings of other numbers. The primality group worked with a modified sieve of Eratosthenes which required numerous multiplications as well. For the third group, password cracking involved creating passwords, generating hashtags, and verifying these hashtags against that of an unknown password. Many arithmetic operations were required for each generated password, which the GPU could easily speed up.

Each group was required to come up with a working C/MPI version first before writing any code in CUDA. These initial programs were used to benchmark results before GPU specific code was added. For two of the projects, this beginning goal was accomplished within a few weeks, but the password cracking project, due to its inherent complexity, took significantly longer to complete, and was actually more complicated to code without CUDA than with.

During the initial program phase, an introduction to CUDA was covered, including a Hello CUDA World program. By the time the beginning programs were running, the students in each group were familiar enough with CUDA to begin the next phase of the project. Due to the usually straightforward computations done on the GPUs, all of the groups were able to convert their initial programs to include CUDA and greatly speed up computations. Both the password cracking group and the primality group saw speed increases above and beyond expectation. As for the large number multiplication group, their programs runtime was greatly reduced as well, but since they were reading in these numbers from files, and writing out the result to file, the speedup was not quite as pronounced as with the other two groups.

3.3 Course Three - High Performance Computing

After a two semester long introduction to parallel programming, students had a reasonable feel for programming in parallel. At this point, we decided to offer a third course, High Performance Computing. This course was implemented with one goal, one project in mind. A colleague in the Mathematics department at Southeasten, Dr. Charles Matthews, had a computational topology problem that required a significant amount of computation time to yield a desired result[3][6]. On some occasions, running on a single process personal computer, the computation would take nine days[3]. Our goal was to guide students along the right path to creating a working program which would perform the desired computations much faster. To give students some time to think about this project, Dr. Matthews was asked to speak to the CUDA Parallel Programming class before the end of the semester. If students were interested after that, they were to enroll in the High Performance Computing class for Spring 2013.

At the beginning of Spring 2013, Dr. Matthews was once again invited to talk to the class about his research, focusing more on the computational aspects of the research. Clearly it was not our intention for the students to understand the fundamentals of Teichmüller spaces, but only what the program was meant to do. The basics of the code, as was presented to the class, was the transversing of a ternary tree to a depth determined by certain cut-off conditions which depended very much upon the location on the tree. In Appendix B, we include some documentation created to help students understand the tree structures we were to create.

Students not familiar with structures and pointers received a crash course in these topics and the main focus of the first third of the semester was on creating the ternary tree. Certain specifications were required, and Dr. Matthews had worked out examples of what these trees should look like. Once the ternary tree could be generated to an arbitrary non-uniform depth, certain calculations had to be done at each point on the tree. Möbius transformations had to be evaluated, derivatives taken and integrals computed. Derivatives of Möbius transforms were discussed, as well as integrals expressed as limits of Riemann sums. The integral portion of the problem lent itself well to MPI or CUDA code. Error bounds, with explicit formulas, associated with cut-off conditions down the branch of the generated tree, as well as error bounds on the integrals being approximated by the Riemann sums, were also computed during the tree transversal.

Once the program appeared to be working, we ran numerous simulations with parameters and initial conditions for which we could check our results against those already known by Dr. Matthews's work. After a few final tweaks, the class had a running program that would duplicate the results of Dr. Matthews's program. All that was left at this point was to attempt to run the program on a supercomputer that would bring the time down to a matter of hours. At this point, we took advantage of the cooperative agreement which the regional universities have with the University of Oklahoma's supercomputing facilities, and used their supercomputing cluster Boomer. Running the code with 64 processes cut the runtime down to approximately 4.5 hours. With access to processes numbering in the thousands, many more runs were completed and data was compiled that would have taken years on a single PC.

4 Results

4.1 Course Acceptance

Our university is a regional state university that offers mostly baccalaureate degrees. The CS department has traditionally followed ACM guidelines for accreditation, which only list parallel programming as an elective[1]. Many CS degree programs have no specific courses for this, which was the case for us.

We wanted to get students in the classroom immediately, and realized that adding new courses and modifying our required course list would be time consuming and difficult, so we took advantage of a course that many universities utilize to cover new topics that are not in the regular curriculum. We used a 'Special Seminar' listing that is similar to a directed reading class, except regular classroom attendance is required.

By using the special seminar route, we were able to offer three distinct courses with three distinct names that appear on the students transcript. The students were very appreciative of this arrangement which we continue to use.

4.2 Job Preparation

The HPC industry suffers a particularly frustrating conundrum. Supercomputers are springing up all over the world at an amazing rate so there is a great need for qualified people to manage and operate them. However, the world's universities are not turning out enough graduates with parallel programming knowledge to fill the void. We see ads on an almost-daily basis that are seeking employees to fill HPC vacancies.

While we have no illusion that students making it through our 3-course parallel sequence are experts ready to fill a high-level position in a supercomputing center, we do believe that a basic knowledge of parallel programming along with a traditional computer science degree will make them attractive to entry-level positions. Supercomputing centers realize that a good plan to get good people is to take graduates that have basic HPC knowledge and develop them from within. Our program is providing such a potential analyst.

5 Acknowledgments

- Mike Morris would like to thank...
- Karl Frinkle would like to thank...
- Other thanks go here...

References

[1] ACM. Computer science curricula 2008. `http://www.acm.org/education/curricula/ComputerScience2008.pdf`. Accessed: 2013.12.05.

[2] BCCD. Bootable Cluster CD. `http://bccd.net/`. Accessed: 2013.12.06.

[3] Yohei Komori and Charles A. Matthews. An Explicit Counterexample to the Equivariant $K = 2$ Conjecture. *Conform. Geom. Dyn.*, 10:184–196, 2006.

[4] Linpack. HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers. `http://www.netlib.org/benchmark/hpl/`. Accessed: 2013.12.07.

[5] LittleFE. `https://littlefe.net/`. Accessed: 2013.12.05.

[6] Charles A. Matthews. Approximation of a Map Between One-Dimensional Teichmüller Spaces. *Experiment. Math.*, 10(2):247–265, 2001.

[7] University of Oklahoma. OSCER – OU Supercomputing Center for Education & Research. `http://oscer.ou.edu/`. Accessed: 2013.12.05.

[8] Erich Strohmaier. Top 500 Supercomputer Sites – Boomer. `http://www.top500.org/system/177821`. Accessed: 2013.12.07.

[9] Ed Tittel. *Clusters for Dummies.* John Wiley & Sons Ltd., 2nd, IBM Platform Computing edition, 2012.

A Appendix A - Matrix Multiplication

As an example, if we consider two 4×4 matrices A and B, each having 16 elements, then the resulting product will be 4×4 as well. If we break this down in terms of elements of the matrices:

$$\begin{bmatrix} A_0 & A_1 & A_2 & A_3 \\ A_4 & A_5 & A_6 & A_7 \\ A_8 & A_9 & A_{10} & A_{11} \\ A_{12} & A_{13} & A_{14} & A_{15} \end{bmatrix} \cdot \begin{bmatrix} B_0 & B_1 & B_2 & B_3 \\ B_4 & B_5 & B_6 & B_7 \\ B_8 & B_9 & B_{10} & B_{11} \\ B_{12} & B_{13} & B_{14} & B_{15} \end{bmatrix} = \begin{bmatrix} C_0 & C_1 & C_2 & C_3 \\ C_4 & C_5 & C_6 & C_7 \\ C_8 & C_9 & C_{10} & C_{11} \\ C_{12} & C_{13} & C_{14} & C_{15} \end{bmatrix} \tag{1}$$

We know the standard formula for matrix multiplication is given by

$$C_{i,j} = \sum_{k=1}^{n} A_{i,k}\, B_{k,j} \tag{2}$$

where the i and j represent the row and column, respectively, of the matrices in question. The goal here, is to combine equations (1) and (2) so that C code can be written to perform matrix multiplication without the use of two-dimensional arrays. Thus, the indices i and j will be rewritten in terms of only one index, let us call it k. The k index will range from 0 to $n-1$, where n is the number of elements in each array. For the 4×4 example, we have $n = 16$ and thus $k \in \{0, 1, \ldots, 15\}$.

So remember, to get the entry in row i column j of the matrix C, we multiply entry-wise, row i of A by column j of B. So we need a formula that relates the row and column of a matrix to the entry k in the array. First, we will define $N = \sqrt{n}$, which gets used quite a bit in future work. Note that there are N rows, where the first row has entries $0, 1, \ldots, N-1$, the second row has entries $N, N+1, \ldots, 2N-1$. Thus, the row number r is given by

$$r = \left\lfloor \frac{k}{N} \right\rfloor + 1 = \text{floor}\left(\frac{k}{N}\right) + 1, \tag{3}$$

where the floor function rounds to the next largest integer. As examples

$$\left\lfloor \frac{7}{4} \right\rfloor + 1 = \lfloor 1.75 \rfloor + 1 = 2, \quad \left\lfloor \frac{8}{4} \right\rfloor + 1 = \lfloor 2 \rfloor + 1 = 3, \quad \left\lfloor \frac{9}{4} \right\rfloor + 1 = \lfloor 2.25 \rfloor + 1 = 3$$

Notice that if you look at entries 7, 8 and 9 in each matrix of (1), we do get the correct row number per entry.

We now move on to the columns, which means we must deal with remainders after division. For example, if we look at entries 7, 8 and 9 again, we have

$$7 = 4 \cdot 1 \text{ r } 3, \; 8 = 4 \cdot 2 \text{ r } 0, \; 9 = 4 \cdot 2 \text{ r } 1$$

notice that the remainder is always one less than the column in question. Thus, we can define the column number c for each entry as

$$c = (k\%N) + 1 = \text{fmod}(k, N) + 1 \tag{4}$$

So now we have the row r, and column c, corresponding to entry k in the matrix C. So we next need to multiply the entries in row r of A by the entries in column c of B. This begs the next question — What are the indices of the entries in row r of A and column c of B?

$$\text{row } r: \ \{(r-1)N, (r-1)N+1, \ldots, rN-1\} = \{0, 1, \ldots, N-1\} + (r-1)\cdot N \tag{5}$$

Similarly, for the columns of B, we have

$$\begin{aligned}\text{column } c: \{(c-1), N+(c-1), 2N+(c-1), \ldots, (N-1)N+(c-1)\} \\ = \{0, N, 2N, \ldots, (N-1)N\} + (c-1)\end{aligned} \tag{6}$$

Note that in the definition of the entries in row r and column c everything can be summed in terms of N. Thus, we have

$$C_k = \sum_{s=0}^{N-1} A[s+(r-1)N]\cdot B[sN+(c-1)] \tag{7}$$

Note that N, r and c are already predetermined and are functions of k, and thus this sum is completely determined to be a function of the entry value k.

As an example, we will attempt to find a formula for C_{11} in (1). In this case,

$$N = \sqrt{16} = 4, \ r = \left\lfloor \frac{11}{4} \right\rfloor + 1 = 2+1 = 3, \ c = (11\%4)+1 = 3+1 = 4$$

and (7) becomes

$$C_{11} = \sum_{s=0}^{3} A[s+8]\cdot B[4s+3] = A[8]\,B[3] + A[9]\,B[7] + A[10]\,B[11] + A[11]\,B[15]$$

If we look at the original matrix equation once, entry 11 of C corresponds to the product of row 3 of A and column 4 of B. The product is indeed given in the formula just presented.

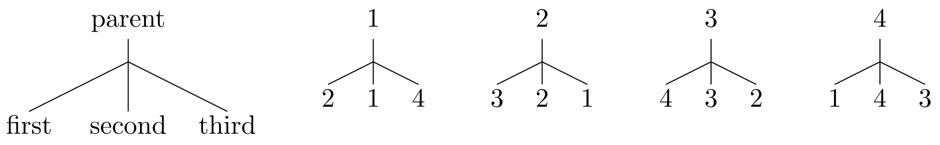
B Appendix B - Ternary Trees

We consider compositions of the four functions T, S^{-1}, T^{-1} and S, or equivalently, words made up of the four letters T, S^{-1}, T^{-1} and S. To make our discussion a little easier to understand, we will give each of these functions a corresponding number and direction:

function	number	direction
T	1	east
S^{-1}	2	south
T^{-1}	3	west
S	4	north

From the above table, we note that T and S are the positive x and y axes on a grid, (hence S^{-1} is in the $-y$ direction, and T^{-1} in the $-x$ direction). Thus the path $1 \to 2 \to 3 \to 4 \to 1$ is a full clockwise rotation about the origin (which we denote by I, the identity map), yielding the word $T\,S^{-1}\,T^{-1}\,S$.

We need to find a method for going through many permutations of the letters in a very methodical fashion. The rule of thumb is to always turn right, then straight, then left. For instance, if we are at a point on the grid, after having applied the function T, (which we call 1), then our next move will be to the right (S^{-1}, or 2). However if it has been determined that we have gone far enough down the current path, instead of turning right (applying S^{-1}), we would instead go straight, which in this case means applying T again (or 1). As a rule, here are how the steps work:



We were given that $T(z) = \mu + \frac{1}{z}$ and $S(z) = z + 2$, and can apply our newfound knowledge of matrix representation of Möbius tranformations to our program. First thing to note is that any Möbius transformation can be expressed in the following form:

$$M(z) = \frac{az + b}{cz + d} \tag{8}$$

where z is a complex valued variable, and the scalars a, b, c and d are complex constants such that $ad - bc \neq 0$. It should be noted that if we wish to compose two Möbius tranformations, M_1 and M_2, the result is another Möbius tranformation. We can represent both $T(z)$ and $S(z)$ in standard Möbius tranformation form as

$$T_\mu(z) = \frac{-i\,\mu \cdot z - i}{-i\ \cdot z + 0}, \quad S(z) = \frac{1 \cdot z + 2}{0 \cdot z + 1}, \tag{9}$$

which in turn gives

$$T_\mu = \begin{bmatrix} -i\,\mu & -i \\ -i & 0 \end{bmatrix}, \quad S = \begin{bmatrix} 1 & 2 \\ 0 & 1 \end{bmatrix} \tag{10}$$

We can now compute T^{-1} and S^{-1} using the matrix form of a transformation:

$$T_\mu^{-1} = \begin{bmatrix} 0 & i \\ i & -i\,\mu \end{bmatrix}, \quad S^{-1} = \begin{bmatrix} 1 & -2 \\ 0 & 1 \end{bmatrix} \tag{11}$$

It is easy to show that $T \cdot T^{-1} = T^{-1} \cdot T = I_2 = S \cdot S^{-1} = S^{-1} \cdot S$. Remember, once again, that we cannot assume that matrix multiplication is commutative, so if we compute $T \cdot S \cdot T^{-1}$, this should not be equal to S or $T^{-1} \cdot S \cdot T$:

$$T_\mu \cdot S \cdot T_\mu^{-1} = \begin{bmatrix} 1 + 2\mu & -2\mu^2 \\ 2 & 1 - 2\mu \end{bmatrix}, \quad T_\mu^{-1} \cdot S \cdot T_\mu = \begin{bmatrix} 1 & 0 \\ 2 & 1 \end{bmatrix}$$

Converting the above matrices back to function notation gives the two Möbius tranformations:

$$\left(T_\mu \circ S \circ T_\mu^{-1}\right)(z) = \frac{(1 + 2\mu)z - 2\mu^2}{2z + (1 - 2\mu)}, \quad \left(T_\mu^{-1} \circ S \circ T_\mu\right)(z) = \frac{z}{2z + 1}$$

We can also compute the derivative of $M(z)$ given in (8), which after some simplification, gives

$$M'(z) = \frac{ad - bc}{(cz + d)^2} \tag{12}$$

Since our functions in matrix form have determinant equal to 1, we have that $ad - bc = 1$. One might raise the question: What about the composition of these functions and their inverses?

The answer lies in the following two statements: (1) The determinant of a product is the product of determinants, and (2) the determinant of an inverse is the reciprocal of the determinant of the original matrix. These two facts imply that $ad - bc = 1$ for any composition and means that we can now simplify our derivative of $M(z)$ to

$$M'(z) = \frac{1}{(cz + d)^2} \tag{13}$$

This is quite a simplification. Now, remember that c and d are possibly complex constants, while z is a complex variable. So if we want to find the real part of $(M'(z))^2$, which is what we will need to for a series approximation later, we will have to do some complex-number simplification.