Thinking in Parallel: Multicore Parallel Programming for STEM Education

[1,*] A. Asaduzzaman, [2] R. Asmatulu, and [1] R. Pendse

[1]Department of Electrical Engineering and Computer Science and [2]Department of Mechanical Engineering, Wichita State University, 1845 Fairmount St, Wichita, KS 67260
*E-mail: Abu.Asaduzzaman@wichita.edu; Tel: +1-316-978-5261

Abstract

Academic research and engineering challenges both have increasing demands for high performance computing (HPC), which can be achieved through multicore parallel programming. The existing curricula of most universities do not properly address the major transition from single-core to multicore systems and sequential to parallel programming. They focus on applying application program interface (API) libraries and techniques like open multiprocessing (OpenMP), message passing interface (MPI), and compute unified device architecture (CUDA)/graphics processing unit (GPU). This approach misses the goal of developing more long-term abilities to solve real-life problems. In this article, we propose a novel approach to teach parallel programming that will prepare science, technology, engineering, and mathematics (STEM) students for present and future computation challenges. Proposed approach requires some C/C++ programming knowledge. As a preliminary attempt, we introduce multithreaded parallel programming to our science and engineering students. Based on the Student Outcomes Assessment Reports and feedbacks from information technology (IT) professionals, proposed approach has potential to provide adequate knowledge so that students can analyze complex problems and develop parallel solutions. According to the Steady State Heat Equation experiment, CUDA/GPU parallel programming may achieve up to 241x speedup while simulating heat transfer on a 5000x5000 thin surface.

Keywords: Concurrent processing, CUDA/GPU technology, multicore/manycore systems, parallel programming, STEM education.

1. Introduction

According to job market trends, there are increasing demands for parallel programmers in the industries. Based on an insidePHC report, from November 2009 to July 2011, OpenMP jobs increased 85%, MPI jobs increased 33%, and CUDA jobs increased 22% [1]. Going forward, all processors (except some small embedded processors for specialized devices) will have multiple cores in their central processing units (CPUs). Moreover, attached GPU cards with large numbers of cores have become very attractive for high performance computing and can provide orders of magnitude speedup over using the CPU alone [2, 3]. To address this space, Intel has rolled out its Many Integrated Core (MIC) architecture [4]. Systems with a small number of cores such as present multicore processors can use a shared memory model whereas as the number of cores increase, hierarchical user-managed memory and distributed memory models can be expected. Undergraduate programming has yet to address this major transition from single core processors to multicore and many-core processors properly. Training students in this technology is critical to the future of exploiting new computer systems [5]. Today, with all the

advances in hardware technology, we as educators find ourselves with multicore computers as servers, desktops, personal computers, and even handheld devices in our laboratories (Labs) while still teaching undergraduate students how to design system software, algorithms and programming languages for sequential environment [6]. The current practice is to introduce parallel programming at graduate-level (only at some high-ranked universities), starting with parallel libraries – OpenMP and thread APIs for shared memory systems [7], MPI for message-passing distributed memory systems [8], and CUDA/GPU for CPU/GPU programming [2]. Usually, such a course begins with learning a library, typically MPI applied to a simple parallel applications such as matrix multiplication or sorting, then move onto thread-based tools such as OpenMP, and finally onto programming GPUs with multithreaded CUDA [9-13]. The focus is on learning programming libraries applied to a few simple parallel applications. This approach does not fulfill the goal of developing more enduring skills to rationalize about parallel solutions and solve larger problems for multiprocessor systems. Therefore, STEM education is badly in need of an approach to teach parallel programming that focuses on higher-level programming strategies for computational problems and especially on ease of programmability.

The rest of the paper is organized as follows: proposed approach to develop/update pedagogy for teaching parallel programming is presented in Section 2. Learning materials are discussed in Section 3. Section 4 summarizes the course overview. In Section 5, the Steady State Heat Equation is studied as an example of CUDA/GPU assisted multithreaded parallel programming model. Finally, this work is concluded in Section 6.

2. Proposed Approach

We propose an approach for teaching undergraduate/graduate students how to design and develop parallel algorithms for multicore architectures. The key success and challenge of this project will be helping students to "think in parallel". Major steps and goals of the proposed approach are discussed below.

2.1 Major Steps

Major steps to develop a new pedagogy or update an existing pedagogy are shown in Figure 1. Our proposal includes right-to-the-industry-needs activities to prepare students for present and future computational (science and engineering) challenges.

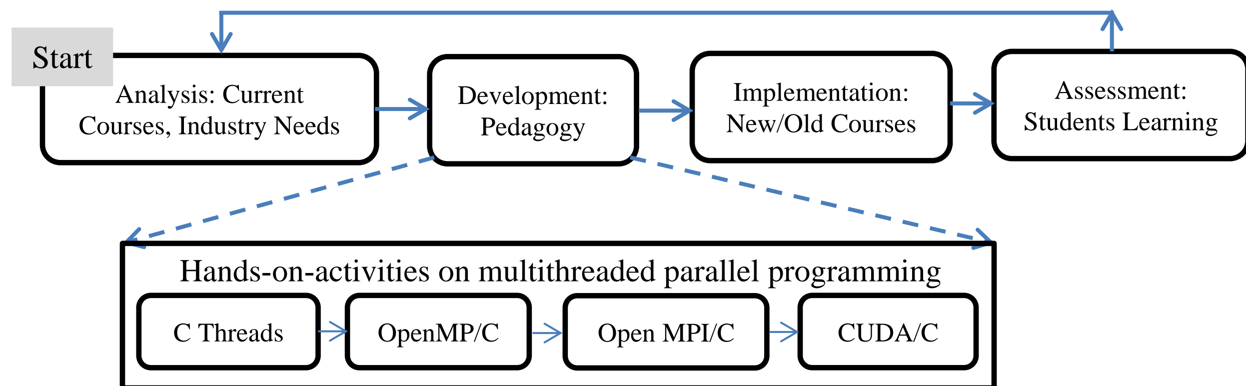

Figure 1. Top line: pedagogy development and integration with existing course(s). Bottom line: hands-on-activities based on real-world IT needs using multicore parallel programming in C.

Our proposed approach has four major steps: analysis, development, implementation, and assessment. First, industry needs and current courses are probed to determine if a new course is needed or existing course(s) should be updated. Then, pedagogy is developed and implemented (accordingly). Finally, student outcomes are assessed. Professional feedbacks and student outcomes are considered to improve the pedagogy. We envision that in order to fulfill the growing IT industry needs, multicore parallel programming will be made available to all undergraduate/graduate engineering students by updating and restructuring existing courses (rather than introducing new courses) [8, 14]. However, this paper provides a complete documentation to prepare a new course or update existing course(s).

2.2 Major Goals

Most current teaching follows a traditional path of teaching low-level APIs although the parallel programming problems now faced by programmers are complex. Our proposal is intended to move away from this approach and make the student programmers begin with decomposing the problem into widely known patterns and structures. Our hope is that our approach will make parallel programming easier and more productive. Major goals of our proposal include:

- Introduce multicore concept and parallel programming techniques in a Lab oriented teaching setting for undergraduate/graduate students.
- Produce educational materials using the pattern-based programming approach (as much as possible) for faculty and professional development.
- Evaluate the materials with diverse groups of students and related IT professionals.
- Develop and implement a solid cyclic strategy for revision and re-testing.

2.3 Involving High School and College Educators

We often realize that some university students have fear about science, technology, engineering, and mathematics courses. One reason may be that the STEM education, especially the new technology, is not effectively transformed to the students during their high-school and/or lower-level college/university years. This causes serious problems for educators to teach and students to learn at upper-level undergraduate and graduate-level courses. The teaching style to provide pre-STEM and/or STEM education to the high school and college students are very important. For example, in addition to sequential operations, parallel operation concepts can be given at some high school level courses like Math, Physics, Chemistry, and Science. Similarly, parallel operation concepts can be given at some college level programming courses like C/C++. That will help students understand multicore computer systems and parallel computing at the universities. Therefore, we think that it is important to involve high school, community college, technical training institution, and university teachers to discuss and address how to improve STEM education and students' learning. A combined effort of STEM educators should help review the current progress and determine the future adjustments.

3. Learning Materials

The importance of developing a successful strategy to teach parallel computing and programming has been raised many times in the past. We think the question is not whether we should incorporate the subject in the undergraduate/graduate curriculum, but what would be the

right approach. Peter Pacheco designed and offered a sophomore-level undergraduate/graduate parallel computing course in the department of computer science and mathematics at the University of San Francisco for the first time in 2004. One of the major goals of the course was to provide the students with hands-on experience and encourage them to start to think in parallel. His conclusions, after offering the course five times, were consistent with the observations that students find the concepts of synchronization, race conditions, and parallelism challenging. His final recommendation was "don't expect them to discover how to write parallel programs: give them a lot of guidance." Given that the goal of the course is to help students to "think in parallel", we should provide the environment within which students solve problems with parallelism as default. Given such a framework and starting with a problem statement that needs to be solved in this framework, will encourage the students to solve the problem in parallel.

We will focus on programming multicore computers with shared memory programming languages as well as on message passing programming environments for this course. We believe the foundations for thinking in parallel can be better built within the scope of shared memory which is the MIMD model for the multicore computers. The MPI programming and design developed for message passing distributed MIMD platforms adds an additional level of complexity and challenge to problem, data, and program partitioning that can be further explored as an advanced level. This experience was shared by Adams, Nevison, and Schaller who designed three different parallel computing courses at three different colleges, Calvin, Colgate, and RIT [15]. Their experience was that depending on the computing environment they had to design different courses and provide a different set of learning experiences for their students.

In many cases, the best parallel solution will perform poorly on a sequential machine. The parallel solution performs better only when it is executed concurrently in parallel on a parallel computer with enough number of processors. Learning about (i) the trade-offs between parallelism and memory usage, (ii) inherently sequential access data structures versus data structures that allow for parallel access, (iii) allowing more operations to be performed in a parallel version compared to the sequential version of the same problem, etc. can be done most effectively when students observe these factors in a hands-on laboratory environment.

Beyond this point is also the philosophy of rethinking the computer science curriculum for teaching students to start with parallel programs. Some of the questions to keep in mind as we introduce the concepts of parallelism and parallel programming to the students include:

- How can one analyze an application to determine what operations can be done in parallel?
- What aspects of a particular algorithm influence what can be done in parallel?
- What aspects of a particular algorithm influence what cannot be done in parallel?
- Is there a way to express a computation without introducing any artificial sequentiality?
- How dependences among operations can be used to structure special software organizations to carry out the computation efficiently on a parallel computer?

We intend to use the lessons we learn and the outcomes of students' experiences throughout the course to develop further strategies for developing ideas for introductory parallel problem solving computer science courses.

4. Course Overview

The best way to equip students with a rich set of experiences in parallel programming is to give them a chance to work with many types of parallel applications. We plan to combine the parallel computing concepts with the skills of parallelizing real world problems. Throughout the course we will use applications in linear algebra, scientific/engineering problems, and nanocomposites as examples in the lectures to relay the fundamental concepts, as frameworks for homework assignments, and as case studies for programming Lab assignments and team projects. Students will be engaged both in classroom and out-of-classroom actively by participating in the classroom discussion and laboratory exercises designed to engrain the concepts being taught through the lectures.

4.1 Structure of the Lab Assignments

We will design two types of Lab experiments: Individual Lab assignments and one final team-project. All assignments will have a classroom discussion followed by students writing a summary of their understandings, observations, and conclusions.

I. Individual Lab assignments: In laboratory room; about 10 Labs; 10% of the final grade. The progression of each in-Lab assignment will be first introducing the intended concept through a dedicated lecture and presenting the students with a complete demonstration of a working program which students can simply copy and run, observing the input/output behavior from the compiler and executing program, and checking execution times. The student is then asked to make a simple modification to the demonstration problem. This forces the students to use the editor, and to solve problems in compilation and execution. The Lab will be followed by a classroom discussion.

II. Final team-project: 20% of the final grade. Students will form teams based on their interest and will work under supervision of either the instructor or one graduate teaching assistant. For the final team-project, students will be presented with a number of case study alternatives to choose from. Students may also propose their project of interest for approval. Each team will produce a written report and will demonstrate their final projects to the class.

4.2 Structure of the Tests

We plan to have three types of tests: Homework (HW), quiz, and exam.

I. HW: Take-home tests; about 10 HW; 10% of the final grade. Each HW will be assigned based on the materials covered most recently.

II. Quiz: Classroom tests; about 4, 30 minutes per quiz; 20% of the final grade. Each quiz will cover most recently taught materials.

III. Exam: Classroom tests; 2 exams, 70 minutes per exam; 40% of the final grade. Exam-1 will be right before the Mid-Term point and Exam-2 will be right before the study day. Materials taught after Exam-1 will be covered in Exam-2.

4.3 Course Outline

Proposed course should be a 4 credit hours course with one credit hour laboratory activities. A high-level course outline of the proposed semester long senior-level course is presented next.

Prerequisites: Students should have adequate knowledge on the topics covered in Introduction to Computer Architecture and Problem Solving and Programming in C/C++ courses.

Module 1: Background and Motivation. This module will introduce the parallel computing by means of evolution of parallelism, concurrency, and multicore/manycore computer architectures with specific examples to demonstrate each concept.

Homework: Students will be asked to explain their understanding of parallelism in sequential machines and contrast it with the type of parallelism in the new multicore computers.

Module 2: Observing Parallelism. This module introduces the data dependence relationships and their impact on the ability to perform parallel operations using dataflow graphs. It will present several sequential and parallel computation examples using data dependent graphs to demonstrate the representation of parallel operations in a given algorithm. Performance analysis for these computations will be presented using size, depth, speedup and efficiency of algorithms. Graph theory will be introduced and applied to several example computations.

Homework: Design a parallel computation using data dependent graph; analyze the computation using size and depth; measure speedup and efficiency of the resulting algorithm. This homework problem may be regarding amount of storage needed for archiving many data structures which is a good application of parallel prefix computation covered in lectures.

Module 3: Getting Started. To express algorithms, a set of pseudo code conventions for expressing sequential and basic parallel operations such as process creation and termination (fork/join) and storage classes (shared/private variables) will be presented. An example parallel pseudo code such as matrix multiplication will illustrate the parallel operations. Pthread in C/C++ will be used to illustrate multithreaded programming.

Homework: Write a parallel code for a parallel prefix computation.

Lab Assignment: Warm-up with Multiple Threads. Students will learn how to use the computer systems in the Lab (activities include: login to the Linux machines, write programs, compile/debug codes, execute programs, prepare Lab reports, and submit reports for grading). They will be given a program framework that creates a number of processes to print a private message using Pthreads.

Module 4: Programming Shared Memory Multicore Computers. This module introduces OpenMP for parallel programming. OpenMP is presented within the context of solving some numerical algorithms. The key to this section is to start programming and present the examples in the context of global parallelism (think in parallel) [16]. Using the OpenMP "Parallel Region" construct, a parallel framework is designed where processes are created at the start of the

program and joined at the end of the program. The parallel applications will be implemented within this framework where parallelism is the default mode of operation. Lectures will cover the basic operations on dense matrices such as matrix multiplication, Laplace's equation, and Gaussian elimination to introduce storage layout and various parallel loops. Work distribution constructs, parallel loop scheduling techniques, the appropriate use of the correct schedule (static vs. dynamic) for specific applications will be presented and motivated using the numerical applications. Synchronization techniques (locks, critical sections, barriers, produce/consume) will be described and the use of the corresponding OpenMP synchronization constructs will be signified in example codes. With focus on data-parallelism various schemes of partitioning matrices for parallel computation including block, cyclic, and block-cyclic partitioning will be discussed.

Homework: Design a producer/consumer program consisting of many producer and consumer processes and a shared buffer area.

Lab Assignment: Synchronization in Parallel Programming. This Lab is to demonstrate the need to use synchronization for correct operation of parallel programs. Students will observe what happens if they are not used correctly. They will be asked to correct the parallel code by inserting appropriate synchronization constructs in the code. They will be asked to use alternative constructs and discuss their observations.

Programming Assignment: Gaussian Elimination. A detailed description of the Gaussian elimination methodology will be provided to the students. The basic components and hints for this assignment will also be included. Students will experiment with and learn the strategies of data partitioning and scheduling; they will observe the impact on performance. Then, they measure and analyze the efficiency of their parallel programs.

Module 5: Trivial Parallelism. In this module, we will introduce a set of interesting but easy to parallelize problems known as "embarrassingly" parallel. These problems require the simplest parallel solution whose computation can obviously be divided into a number of completely independent parts, each of which can be executed by a separate processor/core. Some problems in this category include low level image processing, Mandelbrot set [17], Monte Carlo calculations [18], rendering of computer graphics, brute-force searches in cryptography, BLAST searches in bioinformatics [19], etc. Several case studies and computing benchmarks will be introduced. With this module, students are able to recognize some real-application problems that are highly suitable for parallel execution as well as understand the benefits of parallelism in speeding up time consuming computation applications. This module introduces Open MPI for distributed memory parallel programming.

Lab Assignments: Parallel Matrix Multiplication. Students will be presented with two implementations of OpenMP parallel matrix multiplication programs using two data partition schemes block-partitioning and cyclic-partitioning to run and analyze their observations. They will be asked to revise the code using block-cyclic partitioning, measure performance for comparison and discuss their analysis. This assignment will include the instructions of using profiling tool (e.g. AMD CodeAnalyst [20]) for performance measurement.

Programming Assignment: Sequential to Parallel Program. Students are given a correctly running sequential program, for example a simple Monte Carlo simulation [18]. They are then required to parallelize this program using Pthreads and Open MPI. The assignment includes step-by-step instructions so that students can write their own multithreaded program, compile and benchmark it for performance and accuracy. At the same time, they will become familiar with the Monte Carlo simulations.

Module 6: Massive Parallelism. Graph algorithms will be solved using CUDA/GPU based parallel programming technique. Graph theory plays an important role in computer science and engineering because it provides an easy and systematic way to model problems. Many massive and/or complex problems can be expressed in terms of graphs, and can be solved using standard graph algorithms. This module will present the parallel formulations of fundamental graph algorithms. Students will learn how to decompose a graph into sub-graphs with the goal of optimizing load balance and minimizing synchronization overhead. The parallel implementations of graph theory including Prim's minimum spanning tree algorithm and Dijkstra's algorithm will be covered [21]. GPU-shared-memory programming will also be covered in this module.

Lab Assignment: Minimum Spanning Tree Algorithms. Students will be provided the source code of sequential minimum spanning tree algorithms. They will be asked to select one of three available solutions for parallelizing this program. Upon this selection, they will write the parallel program and present the performance result in class.

Programming Assignment: Number of Fixed Length Paths. Students will design and implement a parallel program to find the paths in a 1024-node graph with a fixed length using CUDA/GPU. They will implement CUDA programs with and without GPU's shared memory.

Module 7: (Optional; if time allows.) Sorting Algorithms. Sorting is one of the most common and important techniques in computer science and engineering and widely used in many applications such as database operations, image processing, statistical methodology. A number of different types of parallel sorting schemes have been developed for a variety of parallel computer architectures [22-24]. The lectures of this topic present several parallel sorting algorithms such as merge sort, quicksort, bucket sort and bitonic sort. By solving several parallel sorting algorithms, we address important techniques like work pool, recursive decomposition, data partitioning, synchronization, load balancing, and divide and conquer.

Lab Assignment: Parallel Quick Sort Algorithm. In this assignment, students will work with a demonstration of parallel quick sort algorithm. They will be asked to manipulate the code to satisfy certain requirements. For example, they may be asked to use parallel tasks instead of parallel sections or limit the depth of the recursion to certain level. Students will compare their performance results to the sequential implementation of quicksort and discuss their observations, improvements, and conclusions.

Programming Assignment: Parallel Sorting Algorithm. Students will design an application to sort 10 million numbers using any one of the parallel sorting algorithm presented in the lectures. They will be asked to explain the parallel techniques they used for their projects: why were they chosen and what is their significance?

Team Project Ideas: Any problem that can be solved by writing traditional/sequential computer programs and/or developing traditional/sequential computer simulations but takes significant amount of time may be a nice project topic for this course. We select several typical applications from different domains that can be parallelized for possible team projects. Students are welcome to propose their own project ideas for approval. Each team-project packet includes documents for background knowledge, literature references, project requirements and instructions for implementation, and outcome evaluation. Some team-project examples are:

Lightning Strike Protection on Nanocomposites: The lack of lightning strike protection (LSP) for the nanocomposite materials limits their use in many applications including aircrafts. As a result, there is a continuous interest in understanding the heterogeneous thermoelectric behavior of mixtures with carbon fibers/nanotubes of these materials. Currently available methodologies, including computer simulation, to assess the thermoelectric behavior of composite materials are extremely time consuming, expensive, and ineffectual. A fast and effective simulation model can be developed using CUDA/GPU technology to analyze LSP on nanocomposite aircrafts.

Processing Large Images: Processing large images is computing intensive and time consuming. Moreover, sequentially processing an image through the CPU does not take advantage of all the available processing resources such as general purpose GPU (GPGPU) cards. Applying various image filters through the GPGPU parallel programming should improve the overall performance while processing larger images without compromising the existing resources. Other parallel programming techniques like OpenMP and Open MPI can also be used.

Deterministic Primality Test: Prime numbers play an important role in maintaining the secret spy codes. Computer hackers try to steal information or break into private transactions. Computer security authorities use extremely large prime numbers when they devise cryptographs for protecting vital information that is transmitted between computers [25]. The primality test on GPGPU is expected to be faster than on CPU for large numbers, such as those used in public key cryptography. Using parallel solutions (like OpenMP, Open MPI, and CUDA/GPU) not only should save time, but also should reduce power consumption.

Improve Decryption in a Partially Homomorphic Encryption Schemes: In cryptography, public key algorithms are widely known to be slower than symmetric key alternatives for their basis in modular arithmetic. The modular arithmetic in RSA (for R. Rivest, A. Shamir and L. Adleman; 1977) [25] and Diffie Hellman is computationally heavy when compared to symmetric algorithms relying on simple operations like shifting of bits and XOR. Parallel techniques (like OpenMP, Open MPI, and CUDA/GPU) can be used to make a more efficient and faster implementation of public key algorithms.

Parallel Processing of String Matching Algorithms: It is expected to present the parallel implementations of the Quick-Search, Naive, Knuth-Morris-Pratt, and Boyer-Moore-Horspool and on-line exact string matching algorithms using the CUDA toolkit. Both the serial and the parallel implementations should be compared in terms of running time for different reference sequences, pattern sizes, and number of threads. It is expected that the parallel implementation of the algorithms is faster than the serial implementation, especially when larger text and smaller pattern sizes are used.

5. Preliminary Work: CUDA/GPU Assisted Parallel Programming Model

In this section, we present a CUDA/GPU accelerated parallel programming technique to solve steady state heat equation [26, 27]. Let's consider the heat flow in a one-dimensional uniform bar. If two nearby points on the rod, separately by a small distance *d* are at temperatures *t1* on the left and *t2* on the right, then the heat flow from left to right between these points is proportional to the temperature difference and inversely proportional to the distance as shown in Equation 1.

Amount of heat per unit time = k(t1 – t2)/d .... (1)

Where, the constant of proportionality *k* is the thermal conductivity and it depends only on the materials that make up the rod. Now, we explain the discrete approach of heat conduction on a 2D surface. Consider a physical region (width *w* * height *h*) and the boundary conditions as shown in Figure 2.

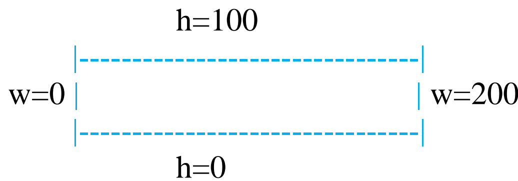

Figure 2. A physical region with width w=200 and height h=100.

Say, the region is covered with a grid of *m* * *n* nodes (see Figure 3). An *m* * *n* array *A* is used to record the temperature of each node. The correspondence between array indices and locations in the region is suggested by giving the indices of the four corners:

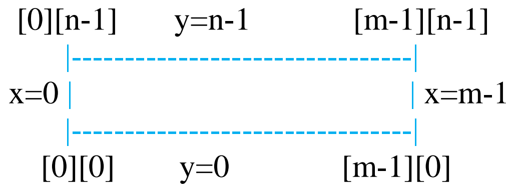

Figure 3. A physical region with boundary conditions.

The steady state solution to the discrete heat equation satisfies the following condition (see Equation 2) at an interior grid point:

A [x, y] = (1/4) * (A [x-1, y] + A [x+1, y] + A [x, y+1] + A [x, y-1]) .... (2)

Where, [x, y] is the index of the grid point, [x-1, y] is the index of its immediate neighbor to the "left/west", and so on. Given an approximate solution of the steady state heat equation, a "better" solution is given by replacing each interior point by the average of its 4 neighbors – i.e., by using the condition as an assignment statement (see Equation 3):

A [x, y] <= (1/4) * (a [x-1, y] + A [x+1, y] + A [x, y+1] + A [x, y-1]) .... (3)

If this process is repeated often enough, the difference between successive estimates of the solution will go to zero (or close to zero). In the main loop, after calculating each new A[x, y] value, it is checked if the value is in the acceptable range or not. We use this approach in our scheme. However, using parallel programming like CUDA/C, we convert the sequential loop into parallel equivalent threads and run them concurrently on the GPU cores.

We run the simulation programs on a CUDA server (a CPU/GPU system) in our research laboratory. Important parameters of the CPU and the GPU are: the CPU is an Intel Xeon E5506 processor with 8 cores the GPU is a NVIDIA Tesla C2075 card with 448 cores. Linux (Debian) is the operating system (OS).

To implement the steady state heat equation (as shown in Equation 3) on a 2D surface, we consider that an $n * n$ very thin metal surface has $N * N$ nodes; where N = 100, 500, 1000, 2500, or 5000. Initially, all the boundary nodes (where x=0, y=0, x=n-1, or y=n-1) are given a value of 0.00 (these values do not change). Also initially, any one node (x, y), where $1 \leq x \leq n-1$ and $1 \leq y \leq n-1$, is assumed to have a very high value (1000000 in our experiment). Then the new values for all nodes are calculated. These iterations are repeated until the new value of a node becomes less than a predefined small value, often called 'error tolerance' (0.0001 in our experiment). Experimental results (CPU time and GPU time) are shown in Table 1. First thing to notice is that both the CPU time and the GPU time increase significantly as the problem size increases. At the beginning, for smaller problem size, CPU time is actually less than the GPU time. However, as the problem size keeps getting bigger, the GPU time keeps getter better (i.e., smaller). It is also observed that the shared memory CUDA implementation outperforms the regular CUDA implementation.

Table 1. CPU and GPU Time due to the steady state heat equation with error tolerance = 0.0001

Problem Size NxN	**CPU Time (Sec)**	**GPU Time (Sec)**	
		Without Shared Memory	**With Shared Memory**
100 x 100	2.47	3.86	3.88
500 x 500	421.35	7.60	6.08
1000 x 1000	1572.87	19.17	11.63
2500 x 2500	6592.91	66.72	34.36
5000 x 5000	12071.26	116.71	50.02

We calculate the speedup due to CUDA/GPU implementation over CPU implementation as shown in Figure 4. For small problems (100 x 100 in our experiment), the speedup is less than 1.0. However, the speedup increases as the problem size increases. It should be noted that the speedup of CUDA without shared memory is always smaller than that of CUDA with shared memory. It should also be noted that after the problem size exceeds a limit, although the speedup of CUDA without shared memory is negligible, the speedup of CUDA with shared memory is significant. For problem size 5000x5000 in our experiment, CUDA with shared memory implementation helps reduce processing time from 12071 seconds to 117 seconds (i.e., a speed up factor of 241).

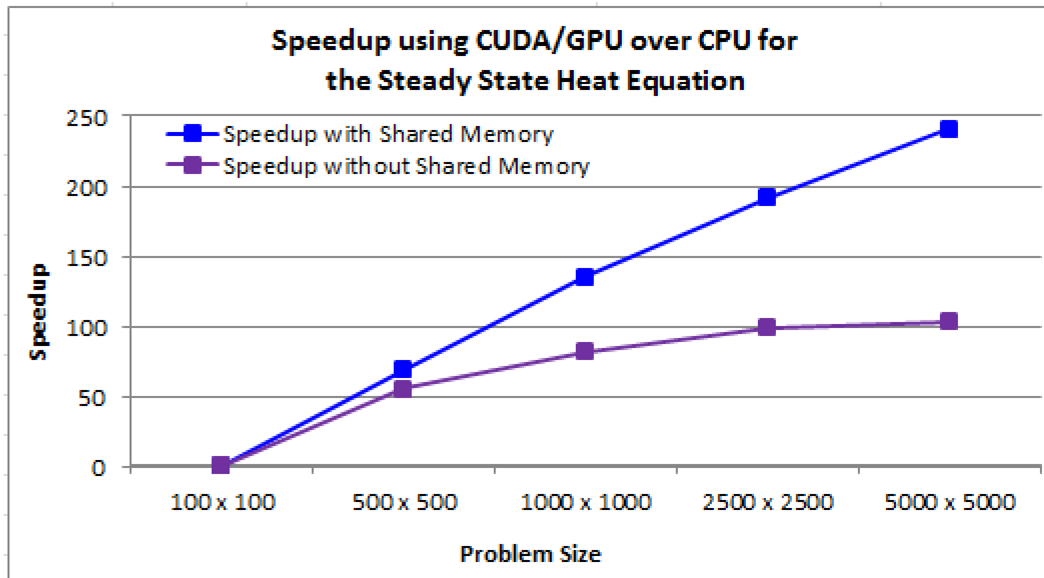

Figure 4. Speedup of discrete heat equation using CUDA/GPU based parallel programming.

6. Conclusions

Multicore/manycore computer and parallel computing are today's actuality. Concurrent processing, an advanced level parallel computing where multiple computations progress concurrently at the same time on multiple processing cores, has the potential to speed up the execution of very complex and large problems. The growing demands for high performance computing can be fulfilled by developing effective parallel programs suitable for multicore/manycore systems. Recent reports show that the demands in parallel programming jobs are growing significantly. Therefore, universities are expected to prepare the new graduates with proper knowledge and skills in parallel thinking. Present science and engineering curricula more or less teach the parallel programming APIs like OpenMP, MPI, and CUDA, but do not develop 'think in parallel' aptitudes by addressing the transition from single-core to multicore architecture and sequential to parallel programming. To address this issue, National Science Foundation (NSF) is currently supporting various projects to educate STEM educators about HPC techniques.

This paper introduces an effective approach to train the students with fundamental knowledge and analytic skills to understand large complex problems and develop parallel computing solutions to meet the current and future requirements. As an experiment, we introduce multithreaded parallel programming to undergraduate/graduate level science and engineering students by updating an existing course. We cover multicore architecture and multithreaded programming; teach how to dissect a problem and develop multithreaded parallel programming for multicore CPU and manycore GPU systems using CUDA/C. We greatly appreciate the feedbacks and advices from the director of Wichita State University (WSU) high performance computing center (HiPeCC) and the CEO of M2SYS Technology. We review the Student Outcomes Assessment Reports for this course. According to the experimental results, the proposed approach shows potential to provide adequate knowledge and training so that students should be able to develop parallel programs for any complex problems.

In our laboratory, we develop a CUDA/GPU assisted parallel program to solve the Steady State Heat Equation (see Equation 3) for different 2D thin surfaces. Experimental results show that up to 241x speedup can be achieved for an error tolerance of 0.0001 (see Figure 4). Results also suggest that the parallel solution has potential to save energy consumption by significantly reducing the execution time.

We feel the need to offer a complete/new course in Multicore Parallel Programming as presented in this paper for our undergraduate/graduate level STEM students very soon.

Acknowledgements

We sincerely acknowledge Mizan Rahman, Founder and CEO of M2SYS Technology, Atlanta, GA, for his encouragement to this work and effort to review the earlier drafts of this paper. We also acknowledge John Matrow, Director of WSU High Performance Computing Center (HiPeCC), Wichita, KS, for his effort to review students' projects and provide valuable advices. Finally, we acknowledge our students for their effort to provide constructive feedbacks.

References

1. "Trends Show Huge Growth in Parallel Programming Job Market," insidePHC, http://insidehpc.com/2011/07/16/trends-show-huge-growth-in-parallel-programming-job-market/ - accessed on June 21, 2013.
2. "Nvidia: CUDA," NVIDIA, http://www.nvidia.com/object/cuda_home_new.html - accessed on June 21, 2013.
3. "Introduction to Parallel Programming," Udacity, https://www.udacity.com/course/cs344 - accessed on June 21, 2013.
4. "Intel Many Integrated Core Architecture (Intel MIC Architecture)," Intel Developer Zone, http://software.intel.com/en-us/forums/intel-many-integrated-core - accessed on June 21, 2013.
5. Marowka, A. "Think Parallel: Teaching Parallel Programming Today," in IEEE Distributed Systems Online, Vol. 9, No. 8, 2008.
6. Mellor-Crummey, J., Gropp, W., and Herlihy, M. "Teaching parallel programming: a roundtable discussion," in XRDS: Crossroads, The ACM Magazine for Students - The Changing Face of Programming, Vol. 17, No. 1, pp. 28-30, 2010
7. "The OpenMP API specification for parallel programming," OpenMP, http://openmp.org/wp/ - accessed on June 21, 2013.
8. "Open MPI: High Performance Computing," Open MPI, http://www.open-mpi.org/ - accessed on June 21, 2013.
9. "Multicore Programming Education 2009," Workshop on Directions in Multicore Programming Education, Washington DC, 2009.
10. "Multicore LA 2011: Open Source Software, Multicore and Parallel Computing Miniconference," http://multicorelca.wordpress.com - accessed on June 21, 2013.
11. Zhu, Y. "Supercomputing Undergraduate Program in Maine (SuperMe)," NSF RUE Award 0754951, 2008.
12. Zhang, W. "Collaborative Proposal: Problem-Based Learning of Multithreaded Programming," NSF CCLI Award1063644, 2011.

13. Brown, R. “A strategy for injecting parallel computing education throughout the computer science curriculum,” NSF CCLI Award 0942190, 2010.
14. Ernst, D.J., et al, “Concurrent CS: Preparing Students for a Multicore World,” ITiCSE’08, 2008.
15. Adams, J., Nevison, C. and Schaller, N.C. “Parallel computing to start the millennium,” in Proceedings of the thirty-first SIGCSE technical symposium on Computer science education, ACM publication, Vol. 32 Issue 1, pp. 65-69, March 2000.
16. Alaghband, G. and Jordan, H.F. “Overview of the force scientific parallel language”, in the Journal Scientific Programming, Vol. 3, No. 1, Spring 1994.
17. Ashlock, D. “Evolutionary Exploration of the Mandelbrot Set,” in IEEE Congress on Evolutionary Computation, 2006.
18. Coates, R.F.W., Janacek, G.J., and Lever, K.V. “Monte Carlo Simulation and Random Number Generation,” in IEEE Journal on areas in communications, Vol. 6, No. 1, 1988.
19. "BLAST: Basic Local Alignment Search Tool," http://blast.ncbi.nlm.nih.gov/Blast.cgi - accessed on June 21, 2013.
20. "AMD CodeAnalyst Performance Analyzer," AMD, http://developer.amd.com/cpu/codeanalyst - accessed on June 21, 2013.
21. Cormen, T.H., Leiserson, C.E., Rivest, R.L., and Stein, C. “Introduction to Algorithms,” the MIT Press Cambridge (3rd Ed.), Massachusetts, 2009.
22. Amato, N.M., Iyer, R., Sundaresan, S., and Wu, Y. "A Comparison of Parallel Sorting Algorithms on Different Architectures,” in Technical Report 98-029, Department of Computer Science, Texas A&M University, 1996.
23. Blelloch, G.E., Leiserson, C.E., Maggs, B.M., Plaxton, C.G., Smith, S.J., and Zagha, M. “A comparison of sorting algorithms for the Connection Machine CM-2,” in Annual ACM symposium on parallel algorithms and architectures, pp. 3–16, 1991.
24. Li, H. and Sevcik, K.C. “Parallel sorting by over partitioning,” in the proceedings of the sixth annual ACM symposium on parallel algorithms and architectures (SPAA'94), pp. 46–56, 1994.
25. Rivest, R.L., Shamir, A., and Adleman, L.M. “RSA algorithm,” U.S. Patent 4,405,829, 1977.
26. “The 2D/3D heat equation,” www.maths.bris.ac.uk/~marp/apde2/week3notes.pdf - accessed on June 21, 2013.
27. Asaduzzaman, A., Yip, C.M., Kumar, S., and Asmatulu, R. “Fast, Effective, and Adaptable Computer Modelling and Simulation of Lightning Strike Protection on Composite Materials,” in IEEE SoutheastCon Conference 2013, Jacksonville, Florida, April 4-7, 2013.

Bibliographical Information

Abu Asaduzzaman

Abu Asaduzzaman received the Ph.D. and M.S. degrees, both in computer engineering, from Florida Atlantic University (FAU), USA. Currently, he is working as an assistant professor in the department of electrical engineering and computer science at Wichita State University (WSU) in Wichita, Kansas. He is the director of WSU’s CAPPLab, which is the CUDA teaching center at WSU. His research interests include computer architecture, parallel programming, and computer

simulation. To date, his scholarly activities have been cited more than 100 times, according to Google Scholar. Dr. Asaduzzaman is a recipient of Kansas NSF EPSCoR First Award 2013–2014. He is a member of the IEEE, ASEE, and the honor society of PKP, TBP, UPE, Golden Key, and Who's Who. He served as reviewer of NSF TUES (2011) and GRFP (2012), and EPSCoR RSV Panel-2 (2012) programs. He has served as Session Chair at various prestigious conferences. He is currently serving as a TPC member of IEEE IPCCC 2013 and as an IPC member of IEEE ICCIT 2013 conferences.

Ramazan Asmatulu

Ramazan Asmatulu received his Ph.D. degree in 2001 from the Department of Materials Science and Engineering at Virginia Tech. After having the postdoc experiences, he joined the department of mechanical engineering at Wichita State University (WSU) in August 2006 as an assistant professor, and received his tenure and promotion to be associate processor in July, 2012. Dr. Asmatulu is currently working with 14 M.S. and 7 Ph.D. students in the same department. Throughout his studies, he has published 57 journal papers and 132 conference proceedings, edited two books, authored 21 book chapters and 4 laboratory manuals, received 28 funded proposals, six patents and 26 honors/awards, presented 61 presentations, chaired many international conferences and reviewed several manuscripts in international journals and conference proceedings. To date, his scholarly activities have also been cited more than 500 times, according to the web of science.

Ravi Pendse

Ravi Pendse serves as the Vice President for Information Technology and Chief Information Officer at Wichita State University. He is also a Professor in the department of electrical engineering and computer science, Executive Director of advanced networking research institute, Wichita State Cisco Fellow, and Senior NIAR (national institution of aviation research) Fellow. Dr. Pendse serves on the advisory committee of Kan-Ed and on the KANREN board. He also serves on the Microsoft Higher Education Advisory Group (HEAG). He is a senior member of IEEE and an Independent Director on the Board of High Touch Technologies.

Both multicore and parallel systems processing units refer to the way and the amount of computer chips operate in a computational system. Click for more.Â Parallel systems are designed to decrease execution time of programs by portioning them into various fragments and processing these fragments simultaneously, these systems can also be known as tightly coupled systems. A parallel system can deal with multiple processors, machines, computers, or CPUs etc. by forming a parallel processing bundle or a combination of both entities. Parallel System Architecture. As it would be expected, these parallel systems, are more difficult to program than single processors because the architecture of they are comprised off, which includes many CPUs, as opposed Tutorial on Multicore OCaml parallel programming with domainslib. ISC License. 155 stars.Â Domainslib is a parallel programming library for Multicore OCaml. It provides the following APIs which enable easy ways to parallelise OCaml code with few modifications to sequential code: Task: Work stealing task pool with async/await parallelism and parallel_{for, scan}. Understand and use the parallel package multicore functions. Understand and use the foreach package functions. Introduction.Â Alternatively, think of remote sensing data. Processing airborne hyperspectral data can involve processing each of hundreds of bands of data for each image in a flight path that is repeated many times over months and years. NEON Data Cube. Why parallelism? ZPL is a language whose parallelism stems from operations applied to its arraysâ€™ elements. ZPL derives from the description of Orca C in Calvin Linâ€™s dissertation of 1992 [Lin92]. Since that time, Orca C has evolved to the point that it is hardly recognizable, although the foun-dational ideas have remained intact.Â Another challenge in parallel programming is the distribution of a problemâ€™s data. Most conventional parallel computers have a notion of data locality.Â Techniques for programming parallel computers can be divided into three rough categories: parallelizing compilers, parallel programming languages, and parallel libraries. This sec-tion considers each approach in turn. 1.3.1 Parallelizing Compilers. The concept of a parallelizing compiler is an attractive one. Parallel programming and the design of efcient parallel programs have been well established in high-performance, scientic computing for many years. The simu-lation of scientic problems is an important area in natural and engineering sci-ences of growing importance. More precise simulations or the simulations of larger problems need greater and greater computing power and memory space.Â Thus, using multicore processors makes each desktop computer a small parallel system. The technological development toward multicore processors was forced by physical reasons, since the clock speed of chips with more and more transistors cannot be increased at the previous rate without overheating.