2 Research Activities

In the first year of this endeavor, we started to investigate multithreaded programming and approach parallization challenges by choosing two different, contempory hardware platforms. The first one is the widely available x86-based general purpose multi-core processor system, which is currently the default for all portable, desktop, and server processors. The second one we chose to investigate is the emerging platform now commonly used for scientific computing applications in addition to its original purpose for graphics rendering — general-purpose graphics processor unit (GPGPU) which by nature supports a large number of hardware threads through the capability of their internal streaming processors. Note that there are many other potential systems such as IBM Cell processors (Playstation 3 or Cell blade servers, both we at Georgia Tech have access to). But given this research is aimed at improving education and learning experiences for entry-level undergraduate students, it will be easier to design problems and have students program on these two popular desktop platforms for preparing their PBL experiences. In reality, the demand for parallelizing these more popular platforms is also growing. In the second (half) year of the project, we continued to design the lab components (or lablets) for parallel programming using these two platforms and their respective commercial or open source programming languages and tools. By gaining hands-on experiences with respect to how to perform parallel programming on these two types of systems, the fundamental parallel programming principles required will be covered for most of the common parallel platforms.

2.1 General-Purpose Multi-Core Parallel Programming

There are a large number of parallel languages, interface, API, and libraries for parallel programming on x86-based general purposed multi-core processors. New effort toward unifying and standardizing these tools (e.g., OpenCL) is under way. For learning how to program these machines in parallel manner, we chose to use OpenMP as the programming interface, and use Intel's C/C++ compiler on a Redhat Enterprise Linux machine running on the dual-socket Quad-core Intel Xeon processor platform. The choice of these tools were based the following rationale: (1) the accessibility of the toolkit, (2) the supporting commercially available tools from Intel such as Thread Checker, Thread Building Block (TBB), Parallel Studio, and (3) the familiarity of languages (C/C++) to the students, therefore, students do not need to learn a brand new language from scratch and will be able to put more concentration on how to break up and parallelize an algorithm.

For the application, as in our original proposal, we started with a simple program *wc*, a Unix utility program that counts the number of words for given input text files. The algorithm implementation of this application is by default sequential. We consider this is a perfect yet simple enough example for students to analyze the algorithm and find out strategies with respect to how to break it apart for parallelization. Most likely, this type of applications will be used as the starting point for students' first parallel programming lab in PBL.

One obvious way to parallelize *wc* is to delegate each input text file to an OpenMP thread when there are more than one file given to the *wc* utility, in other words, implementing task-level parallelization. This type of parallelization strategy is easy to grasp for students who had no prior parallel programming experiences. However, the more interesting case we would like students to learn is the case when there is only one input file. Under this circumstance, students have to learn how to divide up the sequential execution model without running into any dependency hazard causing incorrect results. One can consider this is a classical **MapReduce** problem, which has been applied in many Google applications including their reverse indexing scheme. From this thought process, students will learn the basic concept of parallel programming, i.e., divide-and-conquer and then merge/combine the individual results into one final result. The principle of MapReduce basically is about how to **map** a sequential model into independent sub-tasks, the main part of parallelization, and then **reduce** these individually produced results into one final answer. Toward this for *wc* utility under the instructor's guidance, students should find that the input buffer can be sub-divided into almost equal-sized chunks, but to keep the integrity of words, the subdivision must be performed in a way that all the divvied-up chunks must end upon a word boundary, a correctness requirement. Therefore, the parallelized version will not count the same word twice. Through this example, students will learn this basic MapReduce concept for parallel programming and use OpenMP to parallelize the code.

In addition, we would also have students develop parallel programs for more complex algorithms using OpenMP after they become more efficient in handling parallelization as well as the API itself. In this context, a 3D medical image reconstruction algorithm based on computing tomography (CT) was investigated and studied to create the learning experiences. We studied the algorithm proposed by Alex Katsevich, the first theoretically exact cone beam image reconstruction algorithm for a helical scanning path in CT. We studied the parallelization strategy and had it parallelize at a very fine-grained to be able to take advantages of both single-instruction multiple-data (SIMD) SSE instructions provided by x86 instruction set as well as to execute these SIMD streams on multiple processor cores in a general purpose multicore processor. This exercise involves the dissection of the algorithm to isolate independent computation at fine-grained level, understanding the shared data structure, and the implementation of both SSE at instruction level and OpenMP at the task level. Through analyzing this problem, students will be able to follow the same methodology for parallelizing a complex real-life computing problem. More details will be given in the next section.

2.2 Parallel Programming on GPGPU

We chose GPGPU as another pedagogical tool for parallel programming given its popularity and high computation efficiency and cost-effectiveness. The objective of this exercise is for students to learn how to parallelize a program on a massively parallel machine. Such platforms used to exist only in large-scale domain back in the 90s (at a prohibitively high cost), but now we have such platform accessible to everyone on their desktop or even mobile computers. To begin with, the applications running on such platform must be parallel-friendly, in other words, their algorithms must be inherently data-parallel. We expect that through this effort, students will learn the drastic differences of parallelizing applications on a general purpose Intel processor versus on a GPGPU that supports data-level parallelism. They will also learn the differences in their programming productivity. We also examined the OpenCL, a standard for making heterogenenous computing easier, but did not have enough time and budget to pursue it further at the time of the project closing. However, we anticipate to provide OpenCL based programming exercises (lablets) in the future.

The platform we chose for students to use is Nvidia's G80 and Tesla C870. Students will use CUDA (Compute Unified Device Architecture), a parallel intrinsic-like interface developed by Nvidia to write their parallel programs on these platforms. Architecturally speaking, Nvidia's GPGPU is a *many-core* processor. However, each core (also known as a shader unit) was designed to excel in performing concurrent graphics operations or parallel SIMD-type of computation very efficiently rather than dealing with sequential tasks.

The application we chose for parallelizing is the same 3D medical image reconstruction algorithm designed by Alex Katsevich algorithm used in CT scanner as mentioned in the previous section. This application is data parallel inherently. But to run it efficiently on GPGPU, programmers require some delicate considerations. In addition, once students become proficient in parallelizing simple code on IA-based multicore system, they will be asked to try to parallelize the Katsevich algorithm on an IA-based multi-core platform and compare their productivity and performance results against their parallelized CUDA version running solely on a GPGPU.

To parallelize this algorithm on Nvidia's G80 and Tesla C870, we design the lablets to let students accomplish the following tasks through their thought process in PBL: (1) to recognize the independent portions that can be safely parallelized without compromising the correctness of computation and (2) to recognize the steps that cooperatively update the same piece of data. To gain insight for the above points, students have to analyze the structure of the Katsevich algorithm and set up their parallelization strategies. Some interesting aspect of parallel programming we want students to learn through this process is — in lieu of waiting for dependent yet simple, repeated data to be communicated, it is sometimes faster to just re-calculate the data on each individual core by exploiting its computing power. This highlights the common performance and scalability bottleneck of a parallel system: the cost of synchronization and communication. Once students grasp this concept in mind, certain part of an algorithm could become more parallelizable. The principle here is to trade communication off with additional computation. Since computation is cheaper (in terms of performance) than communication in a today's massively parallel system, eliminating communication and replacing them with additional computation is an acceptable strategy to gain performance.

One reason we chose to let students work on a GPGPU is to have them learn the architecture of massively parallel machines as well. People used to say *"Parallelizing code is easy if you don't care about performance."* That implies, it won't be easy to have programmers learn how to optimize applications without knowing the architecture of the underlying machine. In particular, for GPGPU, students have to deal with limited shared memory issue.

2.3 Material Integration into Georgia Tech's Computer Engineering Program

The final goal of this research effort is to integrate what was investigated and designed (such as programming exercises and labs) into our curriculum. Fortunately, Georgia Tech is undergoing a major curriculum revision for our computer engineering undergraduate program since 1999. No timing is more perfect than now to have our PBL research results to be integrated into our brand new curriculum. Moreover, PI Lee was appointed to chair the curriculum subcommittee for computer engineering program, therefore, his role gives a lot of leverage to turn their research concept into practice in curriculum. In essence, the design flow of PBL multithreaded programming including parallelization stratgy, parallelization tools, and parallelization methodology will be used as a gudie in the new courses that discuss concurrency. More information will be discussed in Section 3.2.