Visualizing Parallelism in CS 2

Sean Massung  
Computer Science Dept, College of Engineering  
University of Illinois at Urbana-Champaign  
massung1@illinois.edu

Cinda Heeren  
Computer Science Dept, College of Engineering  
University of Illinois at Urbana-Champaign  
c-heeren@illinois.edu

Abstract—This paper describes the incorporation of the IEEE-TCPP Curriculum Initiative into CS 2 at the University of Illinois at Urbana-Champaign. With control over only one course that requires a semi-rigid curriculum, we detail a sequence of three lessons that explore the basics of parallelism in a visual manner. We draw a contrast between standard teaching methods for parallelism and assert that our approach is more engaging and more accessible, particularly to spatial learners. We then present examples of our image-centric course material and discuss its deployment. Lastly, we reflect on the effectiveness of this technique over the past two semesters and consider its direction in the future.

I. INTRODUCTION

CS 225 (CS 2) at the University of Illinois is called Data Structures and Programming Principles. Its goal is to introduce students to classic abstraction data types, their implementations in C++, and elementary algorithm analysis. Initially learning how to use dynamic memory and create objects, students progress to simple data types such as linked lists, stacks, queues, and trees. Having mastered these essentials, more advanced topics such as AVL trees, B-trees, KD-trees, hash tables, priority queues, disjoint sets, and graphs are covered in detail—along with associated algorithms like mergesort, quicksort, heapsort, nearest-neighbor search, cycle detection, Kruskal’s, Prim’s, and Dijkstra’s.

This course is a cornerstone in any computer science curriculum and serves as a prerequisite for nearly every upper-level class. In our university, it is not only a mandatory course for CS majors, it is also a requirement for Computer Engineering students and a suggested elective for Electrical Engineers and Industrial Engineers. Serving the two largest departments in the College of Engineering, CS 225 is offered year-round and is taken by around 400 students each semester, and growing. The current semester induced a waiting list of 180 students, and has an enrollment of 450.

The course employs image data as a mechanism to explore course topics. Almost every weekly lab exercise and bi-weekly machine problem (MP, or programming assignment) processes an image in some way. For example, we flip, rotate, invert colors, adjust brightness, remove specific colors, draw collages, create photomosaics, compress images in memory with quadtrees, interlace and deinterlace, design mazes, as well as implement various animated fills and effects. Continuing this convention, our instruction in parallelism replicates some of our existing image manipulations and adds others, taking advantage of the opportunity to visualize the computation performed by multiple threads.

Tethering images to understanding parallelism enhances student interest and provides an alternative route to mastery of the material. Pixelated areas in a PNG give immediate and visual indication of a race condition especially when compared with the traditional technique of matching up output values in an array; it even lends itself to a deeper exploration of the problem: why is only the bottom of the image wrong? Parallelizing an operation across image partitions is much more intriguing than divvying up rows and columns in a matrix. Students easily engage when results are tangible.

II. STANDARD APPROACHES

The parallel skills desired are clearly articulated at schools well-known for their parallel curriculum [1] [2] as well as within the NSF/IEEE-TCPP Curriculum Initiative itself [3], but the specific manner in which those skills should be attained are often vague and instructor-dependent. In CS 2 it is standard to focus on parallelism rather than concurrency and to stick to the programmer’s point of view (i.e., using threads but not implementing them). However, the particular exercises that students should complete so as to achieve these goals is often left as an exercise to the reader.

Although schools such as the University of Washington and Carnegie Mellon University have enjoyed great results with their parallel programs [4] as referenced above, the successful completion of a parallel task usually means observing processor usage go up and execution time go down in some system monitor. It is difficult for students to visualize the effect of that computation on the data itself, especially while debugging. Students must create a mental model of parallel computation which is distinct from their sequential understanding. Images make this mental model very tangible. For some students (most notably the “visual learners”), this representation is critical.

We attempt to address this issue in our curriculum by almost exclusively applying data parallelism to images and building upon the strong foundation other reputable schools have created.

III. PARALLEL TOOLS

The instructional lab currently allotted to CS 225 consists of around forty Linux machines capable of running four threads. Students also have remote access to four eight-core servers. Our course infrastructure decisions were based on
these resources. Most relevant here are the parallel library and a diagnostic tool set for dissecting parallel programs. For the former, we chose OpenMP, and for the latter, the Intel Profiling Tools.

A. OpenMP (Open Multiprocessing)

We use OpenMP as a cross-platform parallel library because we do not want the student experience to depend on a particular lab or machine configuration. OpenMP’s widespread use ensures that wherever students go after this class, they will have access to the same parallel library.

OpenMP significantly reduces boilerplate code and lets students focus on the actual parallelism. In our program, most students’ next CS course will introduce them to pthreads, revealing the inner workings (and gaining an even greater appreciation for) OpenMP. At this point in their academic career, we consider knowing how parallelism works is more important than why it works.

In particular, OpenMP provides very intuitive ways to parallelize for loops. Since iterating through all pixels in an image with a double for loop is a common task in image manipulation, writing this code concisely is important.

B. Intel Parallel Tools

We use Intel’s suite of parallel evaluation tools via an academic license to profile and debug parallel code. They were chosen because of their general availability and usability. Detecting race conditions is done with the Inspector XE [5], and profiling, including a visual illustration of the difference between static and dynamic scheduling through processor usage charts, is given by VTune Amplifier XE [6].

Although these tools do not directly handle images themselves, they provide an intuitive and graphical look at threads that students would otherwise lack, and that we consider to be consistent with our goal of providing the students a visual understanding of the computation.

IV. CURRICULA

CS 225 is not a course on parallelism; it is a course on data structures and elementary algorithm analysis. In Spring, 2012, we chose to include parallelism as an important topic because of its far-spread and increasing relevance in computer science. We believe that to properly equip our students for future classes and industry, they should be comfortable in a parallel programming environment and they should be familiar with a parallel programming paradigm.

Although we do not have complete control over what content is taught in our course, we do have some leeway to intersperse parallel thinking within our course content. Using OpenMP, we focus on data parallelism. Pedagogically, materials emphasize algorithmic design and exercises are created to expose common pitfalls.

We focus on three main parallel programming concepts, each delivered during a two hour discussion/lab section. Material from these lab sections is graded, and some key topics are evaluated on exams in the form of multiple choice questions.

A. Intro to Parallelism

The first lab is an introduction to OpenMP and the parallel programming framework in general. Since students have not seen threads before, let alone heard of parallelism, this is as much a motivational lecture as an instructional one. The main learning objectives of the lab include the topics of run-time profiling, speedup, and parallel computation without data races (a.k.a algorithms that are “embarrassingly parallel”).

Students initially use the Intel tools to profile the execution of serial code. They select the two most time-expensive functions to parallelize, and speculate on expected speedup under parallel computation a la Amdahl’s Law. The profiler’s processor usage charts are particularly informative over simple metrics like overall CPU consumption. Since these graphs show thread usage over time, the students can see which portions of their programs run efficiently in parallel. Asking them to sketch their perceived usage graphs and comparing them to the real graphs gives them insight into what’s actually going on.

This is a great advantage over a raw percent; having a percentage under one hundred does not always mean that there was no parallelism. If /usr/bin/time indicates 90% CPU usage, that could mean that part of the program ran in parallel, but bottlenecks on disk I/O. Conversely, 130% CPU usage clearly shows that the program was running in parallel—however, perhaps using the Intel tools show that the program had 360% utilization for a fraction of run time and 80% for the remaining portion. Without this more detailed graph, it would be less obvious the portion of code running at 80% should be analyzed and potentially sped up.

In the coding portion of the lab, we expose students to computation across threads. For one task, students simply write a function that removes the green color component from an image:

Fig. 1. Screenshots of portions of the parallel tools. Top shows execution time broken down by function. Bottom shows thread usage over time. Courtesy of Intel.
To illustrate the progress of the parallelized code we give the students augmented code that stops execution midway through the operation. Students see that there are four threads operating on Figure 2 removing the green component. The threads are partitioning the image along the width as shown in the code snippet. Each thread operates on one fourth of the image; the darker half is processed and the lighter half is not. The non-determinism of the threads is illustrated in the slightly different widths of the completed sections.

Moving the #pragma omp parallel for to the second for loop produces a different image when stopped midway, and profiling reveals that many more threads are being created. Students learn that parallelizing the outer for loop is more efficient than parallelizing the inner one.

B. Race Conditions

Race conditions are the main topic in the second lab section. Students learn that correctly parallelizing programs does not just consist of blindly pasting a #pragma on an outer for loop.

The given code snippet above—part of a function whose purpose is to flip the image—shows the simplest case of a race condition. As their first exercise, the students are asked to diagnose the problem with the supplied code and correct it.

The incorrectly parallelized flip function produces the image in Figure 3. The fact that the image is incorrect, while its serial version is not, indicates that there is a problem with the parallelism.

Examine the incorrect image and function itself shed some light on the issue. The top half of the image seems correctly flipped, but the bottom is all pixelated. Inside the for loop, we have a temp variable that holds the pixel originally on top as we directly assign the bottom pixel to the top. This direct assignment statement works, but involving temp creates an error.

Moving the declaration RGBAPixel temp inside the for loop fixes the problem, creating a local temp variable for each thread (that does not get clobbered).

A similar thought process is required to finish the remaining exercises in the lab. We introduce the construct #pragma omp critical, which allows only one thread to operate on the critical section of code at one time. We explain how critical regions are a possible solution to some race conditions, though they must be used intelligently—it’s easy to “fix” a buggy parallel program by putting most of the work inside a critical region. To see whether their use of critical sections is appropriate, students use the tools introduced in the first parallel lab, making sure that a misplaced critical section does not serialize their execution.

C. Reductions

The last lab section introduces a paradigm for solving complex data dependency issues, namely reductions. We present reductions as a general algorithmic technique, so as to provide a stepping stone to understanding the MapReduce programming model.

In one portion of this lab section, we ask students to create a PNG color histogram. To do so we simply record the number of pixels of each color. This is trivial in serial, but requires a slightly different approach when applied across many threads, since the sub-problems on each thread must be combined into the whole.
This combination step—the reduction—is often tricky for students to grasp. Some may wonder how looping through to combine all the local frequency data results in any speedup at all. First, we note that this reduction step may take place while other threads are still traversing the image. Second, since the local maps only contain one entry for each unique pixel color, their length is greatly diminished by the dimensions of the original image.

Below is a simplified solution to this exercise.

```c++
map<RGBAPixel, int> ret_freq;
#pragma omp parallel
{ map<RGBAPixel, int> local_freq;
#pragma omp for
  for(int i = 0; i < width; ++i)
  { for(int j = 0; j < height; ++j)
    { ++local_freq[*image(i,j)];
    } }
#pragma omp critical
{ map<RGBAPixel, int>::iterator curr;
  curr = local_freq.begin();
  for( ; curr != local_freq.end(); ++curr)
  { int count = curr->second;
    ret_freq[ curr->first ] += count;
  }
}
return ret_freq;
```

Here we clearly see the map step (count colors locally) and reduce step (combine counts from each thread). Provided code creates a histogram of the colors from the image Figure 4, broken down by which thread counted them (labeled 0 to 3). Each thread’s contribution is roughly related to the rectangular portions it operated upon. Not all threads see every frequent color, but their combination is accurate. Actually seeing where the areas in the columns come from makes the results more believable, and gives a visual way of sanity checking the results.

This same divide-and-conquer thought process is reiterated throughout the rest of the lab as students complete more image manipulation functions. This is an especially valuable lesson for the many non-CS and non-Computer Engineering students. Usually this sort of algorithmic design pattern is reserved for a full-fledged algorithms course, but the topic is introduced in an exciting way early on to students that may not have the opportunity to see it again in the future.

V. EVALUATION AND ASSESSMENT

Student performance on lab exercises was exemplary with average scores of 95% on all three assignments. Besides grading lab work, questions regarding parallelism were included on exams. Below are examples of the questions we have assessed on various tests over the past two semesters.

The first, and most interesting, asks students to diagnose data races in small sections of parallel code. Unfortunately, student performance on this type of problem is weak—typically only 40 to 50% answer correctly. We attribute the weakness to a lack of practice, though that assertion merits further scrutiny.

Use the following 3 code examples to answer the question below. Please assume that all arrays and images have been properly initialized to hold valid data.

(i) // shift all colors to the right
#pragma omp parallel for
for (int i = 1; i < 100; i++)
  colorArray[i] = colorArray[i-1];

(ii) #pragma omp parallel for
for (int i = 0; i < width; i++)
{ for (int j = 0; j < height/2; j++)
{ RGBAPixel temp = *image(i, j);
  *image(i, j)
  = *image(i, height-1-j);
  *image(i, height-1-j) = temp;
} }

(iii) #pragma omp parallel for
for(int i = 0; i < 10; i++)
for(int j = 0; j < 10; j++)
table[i][j] = (i+1)*(j+1);

Which of the code examples above is/are NOT correctly parallelized?

1) Only item (i) is incorrect.
2) Only item (ii) is incorrect.
3) Only item (iii) is incorrect.
4) Two of the above examples are incorrect.
5) All statements (i), (ii), and (iii) are correct.

In (i), students must realize that any index apart from index $i$ is not guaranteed to be in the current thread. In (ii), they realize that since temp is declared locally in the current thread, there is no race condition. In (iii), it may appear that there is a race condition, but the $i+1$ and $j+1$ are not memory accesses. Rather, they are calculations on index values, so there are no cross-thread data dependencies. Thus only (i) is wrong.
Another typical question simply tests their knowledge of the term *reduction*, and its use in a parallel context. Again, student performance is not nearly perfect—typically only 50 to 60% of students answer correctly.

Which of the following is the primary purpose of *reductions* in parallel algorithms?

1) A reduction performs the same instructions on data across multiple threads.
2) A reduction occurs when private data on individual threads is assembled into a general solution.
3) Reduction is a technique wherein parallelism is applied to the portion of a program that requires the most computation.
4) Reduction is just another term for *speedup*.
5) None of these is the correct choice.

While 1) is generally true for almost all data parallel programs and 3) may be true in some cases, only 2) gives a rigid definition. Student performance on this simple question is closer to the average over all multiple choice questions on our exams—typically 60 to 75% of students answer correctly.

The last excerpt question requires students to recall discussions of Amdahl’s law from the first lab.

Suppose an algorithm takes 7 seconds to run serially, and 2 seconds to run in parallel. Then the *speedup* for the parallelized code is:

1) \( \frac{7}{2} \)
2) \( \frac{2}{7} \)
3) \( \frac{7-2}{7} \)
4) The speedup cannot be determined because the number of processors is not known.
5) None of these answers is correct.

This question verifies that students can calculate the effects of their parallel programming. Choice 4) may be a tricky option for some students, as one can imagine speedup to be \( t_{\text{serial}} \times n \), where \( n \) is the number of cores. In fact, this is the maximum possible (an usually unachievable) speedup on \( n \) cores. Since we explicitly give \( t_{\text{serial}} \) and \( t_{\text{parallel}} \), we know the ratio of parallel to serial (i.e., speedup), is \( \frac{t_{\text{parallel}}}{t_{\text{serial}}} \), or choice 2). The execution would be \( \frac{2}{7} \approx 3.5 \) times faster.

Including the parallel material on exams reinforces its importance and motivates students to take it seriously, in addition to allowing us to see how much they have retained. We are marginally discouraged with the results of the assessments, and we believe they are an indication that we should spend more class time on the material.

Each semester, the staff uses course evaluations to address specific areas of concern. Since there is a separate evaluation for each lab section, students will have a great opportunity to talk about how they felt about parallelism in this context. In addition to course evaluations and their personal interactions with the staff, students are active users of the course newsgroup, where many discussions of class material occur. Conversation on parallelism is easy to observe, encourage, and stimulate. In this semester we will be collecting these ancillary comments and discussions for use in evaluating the efficacy of our materials.

**VI. Future Work**

In the future, CS 225 will be able to assume much more prerequisite knowledge due to general curriculum reform. This frees a few weeks in the beginning of the semester originally dedicated to learning C++ and a few elementary data structures.

This available time can be used to deepen the parallel concentration of the course in addition to examining existing topics in more depth. A fourth lab section is under development that deals with OpenMP tasks and their use in parallel sorting algorithms. It will be added to the course syllabus after the curriculum change, expected in Fall 2014. Lectures specifically covering parallelism will also further reinforce its importance and understanding. With these changes, we anticipate better performance on exam questions such as those given in this paper.

Overall, our future work will be to deepen the connection to parallelism in CS 225.

**VII. Conclusion**

Foundations for including parallelism in introductory and second-level courses exist, but should be presented in a more approachable way. Applying parallelism to images gives a useful purpose to students’ work and immediate, visual feedback. Watching threads work on images increases student understanding and allows them to connect with the assignment on a tangible level.

**Acknowledgments**

We would like to thank the National Science Foundation, Intel, and the IEEE Technical Committee on Parallel Processing for granting Early Adopter Status to CS 225 for the Spring 2012 semester. We would also like to thank Intel for their generous academic licensing and supporting of their parallel tools at the University of Illinois. We thank Chase Geigle and Professor Maria Garzaran for their help in developing course curriculum and software.

**References**


