Abstract—This paper describes the incorporation of the IEEE-TCPP Curriculum Initiative into CS 2 at the University of Illinois at Urbana-Champaign. We detail a sequence of three lessons that explore the basics of parallelism in a visual manner. We then present examples of our image-centric course material and discuss its deployment. Next, we briefly cover how parallelism was emphasized in the honors component of the course. Lastly, we reflect on the effectiveness of this technique over the past two semesters.

I. INTRODUCTION

CS 2 at the University of Illinois is called Data Structures and Programming Principles. Its goal is to introduce students to classic abstract data types, their implementations in C++, and elementary algorithm analysis.

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. 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.

II. CURRICULA: THE MAIN CLASS

CS 2 is not a course on parallelism; it is a course on data structures and elementary algorithm analysis. In Spring 2012, we were able to include parallelism as a topic due to our status as Early Adopters of the NSF/TCPP Curriculum. 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.

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.

Students initially use parallel profiling tools to examine 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.

In one task, students remove the green color component of an image:

```c
#pragma omp parallel for
for(int i = 0; i < width; ++i) {
    for(int j = 0; j < height; ++j) {
        *output(i, j) = *source(i, j);
        output(i, j)->green = 0;
    }
}
```

Augmented code stops execution midway through the operation, showing four threads operating on the image to remove the green component. Each thread operates on one fourth of the image; the darker half is processed and the lighter half is not (Figure 1).

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 below--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.

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

    for(int j = height / 2; j < height; ++j) {
        temp = *image(i, j);
        *image(i, j) = *image(i, height-1-j);
        *image(i, height-1-j) = temp;
    }
}
```

Examining the incorrect image (Figure 2) and function itself shed some light on the issue. Moving the declaration
RGBAPixel temp inside the for loop fixes the problem, creating a local temp variable for each thread. A similar thought process is required to finish the remaining exercises in the lab.

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.

Provided code creates a histogram of the colors from the image, broken down by which thread counted them. Each thread’s contribution is roughly related to the rectangular portions it operated upon. Seeing where the areas in the columns come from makes the results seem 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 mere image manipulation functions. 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.

III. CURRICULA: THE HONORS COMPONENT

The honors section has the opportunity to explore parallelism in more depth than the normal class. We chose to focus on parallel tasks and their support in OpenMP. At their first scheduled meeting, students get an introductory lecture covering tasks and their uses.

Honors students are then entered into a competition where they attempt to implement the fastest sorting algorithm over a variety of test cases. The contest results are updated every fifteen minutes, so students’ submissions are judged in almost real time. In order to do well in the contest, they need to find a good sorting algorithm, implement various heuristics based on the data they’re sorting, and use parallel tasks.

Student feedback towards the competition was overwhelmingly positive. Students enjoyed competing against their peers, and were pushed to rethink their implementations as they saw the standings change.

At the end of the competition, the top three students presented their designs to the rest of the class. Doing these presentations allowed other students to see different perspectives on how various sorting algorithms could be parallelized with tasks.

IV. 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. The first excerpt 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.

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.

(i) #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 = *img(i, j);
        *img(i, j) = *img(i, height-1-j);
        *img(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);

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

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

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 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. We will be collecting these ancillary comments and discussions for use in evaluating the efficacy of our materials.

V. 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.