J. Parallel Distrib. Comput. 118 (2018) 233–246

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

TSGL: A tool for visualizing multithreaded behavior

Joel C. Adams *, Patrick A. Crain, Christopher P. Dilley, Christiaan D. Hazlett, Elizabeth R. Koning, Serita M. Nelesen, Javin B. Unger, Mark B. Vande Stel

Department of Computer Science, Calvin College, Grand Rapids, MI, USA

HIGHLIGHTS

- Why a thread safe graphics library (TSGL) is needed in the multicore era.
- Overview of the project goals, design, implementation issues and how they were resolved.
- Detailed presentation of TSGL visualization examples.
- Description of a CS 2 lab exercise in which TSGL has been used.
- Assessment evidence that use of a multithreaded visualization significantly improved student understanding of a parallel abstraction.
- Guidelines for creating multithreaded visualizations.
- How visualization can aid testing and debugging.

ARTICLE INFO

Article history:
Received 14 June 2017
Received in revised form 26 January 2018
Accepted 26 February 2018
Available online 9 March 2018

Keywords:
Graphics
Library
Multicore
Parallel
Threads
Visualization

ABSTRACT

Since multicore processors are now the architectural standard and parallel computing is in the core CS curriculum, CS educators must create pedagogical materials and tools to help their students master parallel abstractions and concepts. This paper describes the *thread safe graphics library* (TSGL), a tool by which an educator can add graphics calls to a working multithreaded program in order to make visible the underlying parallel behavior. Using TSGL, an instructor (or student) can create parallel visualizations that clearly show the parallel patterns or techniques a given program is using, allowing students to see the parallel behavior in near real-time as the program is running. TSGL includes many examples that illustrate its use; this paper presents a representative sample, that can be used either in a lecture or a self-paced lab format. We also present evidence that such visualizations improve student understanding of abstract parallel concepts.

1. Introduction

The vast majority of central processing units (CPUs) being manufactured today have multiple cores, and each of a CPU's cores can simultaneously execute different statements in real-time. Quad-core CPUs are common, and with sufficient budget, one can purchase CPUs with 8, 12, 16, 18, 20, 22, 24, or even more cores [10,9].

Traditional sequential programs will not run faster on such CPUs, as such programs have a single thread of execution. Indeed, as such programs are run on CPUs with more and more cores, sequential programs use the available hardware less and less efficiently, as illustrated in the formula given in Fig. 1.

For example, if *availableCores* is 16, then *sequentialProgramCoreUtilizationPct* is just 6.25%.

To take advantage of multicore hardware, programs must be designed and written as *parallel programs*, with multiple threads of execution. An effective multithreaded program not only uses the underlying hardware efficiently, it also runs faster on such hardware. Put differently, its performance *scales* with the number of available cores.

Since multicore CPUs are the hardware foundation on which virtually all of today's software will run, it follows that future software developers need to learn about parallel programming in general and multithreaded programming in particular. Accordingly, where parallel computing used to be an elective topic in the CS curriculum, it is now a core topic in both the *IEEE TCPP Curriculum Recommendations* [17] and the *ACM/IEEE CS 2013 Curriculum Recommendations* [18].

* Corresponding author.
E-mail address: adams@calvin.edu (J.C. Adams).

https://doi.org/10.1016/j.jpdc.2018.02.025
0743-7315/

$$sequentialProgramCoreUtilizationPct = 100\% \times \frac{1.0}{availableCores}$$

Fig. 1. Sequential Program Core Utilization Percentage.

Most future software developers are trained by computer science (CS) faculty members. It is thus the responsibility of CS faculty members to ensure that their students learn about parallelism. That is, CS faculty members must create and use the pedagogical tools and materials that will help their students understand parallel abstractions and concepts [6,14].

There is an old saying:

"*A picture is worth 1000 words.*"

This saying claims that one well-done visual presentation of information can communicate as effectively as a lengthy textual or verbal presentation. In keeping with this saying, many CS instructional materials use figures and diagrams to try to help students build mental models of concepts and abstractions. These can work for some topics, but their static nature makes them less effective at illustrating the behavior of complex algorithms and processes.

Recognizing this limitation, computer science education researchers have created dynamic visualizations of sequential algorithms and found that such visualizations help students understand those algorithms (e.g., [7,11,19]). Subsequent research found that those who explore a visualization *interactively* learn significantly more than those who view it passively [16]. Others are using block-based languages such as Scratch [5] and Snap! [8] to help programming novices visualize parallelism or concurrency, or creating such visualizations for pre-college students [20]. This work differs from these previous works by exploring how the interactive visualization of *parallel design patterns* [15] affects a student's understanding of those patterns.

We began our project by trying to create interactive, real-time visualizations of multithreaded behavior. To start on this project, we first searched for a graphics library that was *thread-safe*, meaning a library that would allow multiple threads to write to the screen without producing a race condition. Imagine our surprise when we were unable to find one anywhere! We examined nearly a dozen graphics libraries, and none of them would guarantee thread-safety.

The root problem is that graphics libraries store the graphical information being displayed on the screen in a data structure called a *frame buffer*. This frame buffer resides in memory, and if two threads try to write graphical information to it at the same time, a data race occurs, usually causing the multithreaded program to crash.

Unable to find a graphics library that was thread-safe, we decided to create one. We (descriptively but unimaginatively) named our creation the *thread-safe graphics library*, or TSGL.

In Section 2, we provide an overview of TSGL, and Section 3 presents several examples that illustrate how it can be used to let students see multithreaded algorithms operating in near real-time. Section 4 presents evidence that such visualizations improve student learning. Section 5 presents our recommendations for those wanting to use TSGL to build their own visualizations, and Section 6 finishes with our conclusions.

2. TSGL

In this section, we present our design goals for TSGL, our design, and some of the implementation details.

2.1. TSGL design goals

Our list of objectives for TSGL included:

- An easy-to-use *Canvas* class supporting 2D graphics, to which multiple threads can safely draw (or read) pixels, and a thread-safe *CartesianCanvas* class (a subclass of *Canvas*) to easily create Cartesian coordinate systems.
- A *Shape* class hierarchy for drawing basic shapes such as triangles, rectangles, circles, polygons, and so on.
- A *Function* class hierarchy for easily plotting functions.
- The ability to create and display multiple *Canvas* or *CartesianCanvas* objects, simultaneously or in sequence.
- Support for reading, writing, displaying, and processing PNG, JPEG, and BMP image files; plus safely getting and/or setting the individual pixels in such images.
- Interacting with a *Canvas* using a mouse or keyboard.
- Support for each thread to draw in a unique color, so that items drawn by different threads can be easily identified.
- Support for easily delaying a thread's execution, if slowing down a computation is desired.
- Platform independence and high performance.
- Operability with C++11, OpenMP, and POSIX threads.
- HTML-based API documentation like the Java API.

2.2. TSGL design

To achieve our design goals, we designed classes to provide the needed functionality and organized them into a class hierarchy, part of which is shown in Fig. 2.

For example, the *Timer* class in Fig. 2 provides the functionality needed to slow down a computation.

To achieve our goals of platform independence and high performance, we chose OpenGL as our graphical foundation.

To handle OpenGL extensions conveniently, we used the OpenGL Extensions Wrangler (GLEW) library (glew.sourceforge.net). To interact with a *Canvas* using a mouse or keyboard, we used the GLFW library (glfw.org).

To ensure operability with C++11, OpenMP, and POSIX threads, we wrote TSGL in C++11, and used features of the OpenMP and POSIX thread libraries.

To create HTML-based API documentation, we used the Doxygen system (www.doxygen.org).

2.3. TSGL implementation issues

TSGL was created over many person-months of effort, and many implementation issues arose during that time. In the rest of this session, we discuss a sample of some of the more interesting issues and how we resolved them.

One issue we encountered was the frame-buffer race condition described in Section 1. To address this problem, we used the *Shared Queue* parallel design pattern [15]. More precisely, each TSGL *Canvas* has its own:

- shared queue, capable of storing graphical items; and
- render-thread, responsible for rendering graphical items for that *Canvas*.

The *Canvas* class provides a variety of drawing methods, including drawPixel(), drawLine(), drawRectangle(), and so on, each with parameters appropriate for the object being drawn. Each method uses its parameters to define the graphical item being drawn, and then deposits that item in the *Canvas*'s shared queue, which is thread-safe. The render-thread retrieves the graphical items from the shared queue.

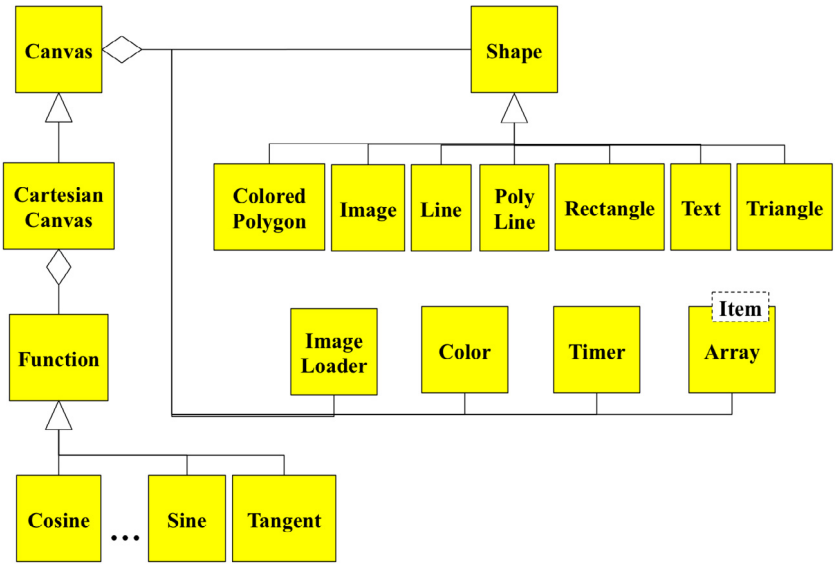

Fig. 2. A partial TSGL class structure diagram.

Initially, we had the render-thread draw each graphical item to OpenGL's framebuffer. This resolved the frame buffer race condition because the render-thread was the only thread interacting with the framebuffer.

However having the render-thread do all the drawing proved to be a bottleneck, so we revised the render-thread to have it convert the graphical items to textures in OpenGL's framebuffer, which the GPU then draws on the screen. The render-thread was still the only thread interacting with the framebuffer, but since a typical GPU has hundreds or thousands of cores, the OpenGL textures are processed in parallel, eliminating the bottleneck. With this refinement, we were able to stress-test TSGL with 1024 threads all drawing to the same *Canvas* and maintain a full 60 frames per second display rate.

3. Examples

In this section, we explore the use of TSGL to see the behavior of parallel algorithms. We have organized this section around *parallel design patterns* [13,12,15], which are elegant, reliable, and efficient solutions to problems that occur commonly in parallel programming. As an example, the *Shared Queue* data structure mentioned in Section 2.3 is a parallel design pattern — a self-synchronizing buffered channel through which parallel tasks can send or receive information. We emphasize parallel patterns for the following reasons:

1. Parallel patterns are the result of decades of experience by professional developers of parallel software. As such, they represent the industry's best practices in writing software that is both reliable and scalable.
2. Parallel software professionals think in terms of these patterns, so the more we can get our students to incorporate these patterns into their thinking, the more like professionals our students will be.
3. The low-level details of parallel hardware and software technologies change frequently, and keeping abreast of those changes may seem overwhelming. By contrast, the parallel design patterns are a relatively stable body of knowledge, making them a useful intellectual framework anyone writing parallel software [3].

In the rest of this section, we explore ways that TSGL can be used to visualize the behavior of two of these patterns.

3.1. The Parallel Loop pattern

In many programs, most of the time is spent in loop statements. This is such a common occurrence, programmers have given it a name — the **90-10 Rule** (also known as the *principle of locality*):

"*90% of the time is spent in 10% of the code.*"

Using parallelism to speed up the processing of a time-consuming loop is a common problem in parallel programming, so parallel professionals have identified a pattern for solving it, known as the *Parallel Loop* pattern. In the simplest form of this pattern, the compiler generates code to (a) identify n, the number of available threads, (b) divide the iteration-range of the loop into n equal-sized chunks, and (c) give each thread one of the chunks to perform.

Explaining this behavior to students can be a challenge, even using a parallel education tool like a patternlet [2]. As we shall see, TSGL makes it possible for students to *see* the behavior of this pattern in operation in near real-time.

3.1.1. Image processing: a balanced-load Parallel Loop

For students who have grown up with smart phones and their built-in cameras, creating a "photoshop-style" effect to process large photographic images can be a motivating way to introduce parallelism [14]. TSGL lets us do this and see the transformation happening in near real-time.

To illustrate, Fig. 3 presents a large and colorful PNG image being displayed using a TSGL *Canvas*.

As an example "photoshop-style" effect, we will process this image using the *color inversion* transformation.

There are different algorithms for inverting a color image, depending on how the RGB color information is stored. If a color's RGB components are integers between 0 and 255, one pseudo-code algorithm is as follows:

Fig. 3. A colorful PNG image. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

```
Canvas canvas1, canvas2;
canvas1.loadImage(imageFile);
for each y in canvas1.getRows() {
   for each x in canvas1.getColumns() {
      Pixel p = canvas1.getPixel(x,y);
      newR = 255 - p.getR();
      newG = 255 - p.getG();
      newB = 255 - p.getB();
      canvas2.setPixel(x, y, newR, newG, newB);
   }
}
```

If the image being processed is sufficiently large and a single thread performs the algorithm, then that thread's progress may be slow enough that it can be seen in real time. For smaller images, TSGL lets us slow the processing as needed for a person to see the thread's progression.

In this algorithm, each pixel's value is retrieved and modified independently of all other pixel values. Since each of the loop's iterations is independent of the others, we can use the *Parallel Loop* pattern to parallelize this algorithm. The following pseudocode shows how we might do so using OpenMP's built-in mechanism for this pattern:

```
Canvas canvas1, canvas2;
canvas1.loadImage(imageFile);
#pragma omp parallel for
for each y in canvas1.getRows() {
   for each x in canvas1.getColumns() {
      Pixel p = canvas1.getPixel(x,y);
      newR = 255 - p.getR();
      newG = 255 - p.getG();
      newB = 255 - p.getB();
      canvas2.setPixel(x, y, newR, newG, newB);
   }
}
```

Suppose that our *Canvas* has 800 rows, and that we are running our program on a quad-core CPU. Then when execution reaches the **#pragma omp parallel for** directive, OpenMP will divide the 800 iterations of the outer for loop into four chunks (0–199, 200–399, 400–599, and 600–799), and give each chunk to a different thread.

Fig. 4 is a screenshot of four threads inverting the image from Fig. 3, with the computation about two thirds done.

The four gray bands in Fig. 4 are the portions of the image that were unprocessed at the time the screenshot was taken; the other areas' pixels have been inverted. Note that the four gray bands are all equal in size, indicating that each thread has an equal amount of work remaining. Conversely, each thread has made equal progress on its chunk of the image. TSGL thus lets a student see: (i) *what* work each thread is doing; (ii) *when* that work is being done, in relation to the other threads' work; and (iii) *how fast* each thread is working, compared to its peers.

At the end of the loop, we have each thread draw a uniquely-colored rectangle around its chunk of the image, so that its contribution toward the overall computation can be clearly seen, as shown in Fig. 5.

TSGL makes it possible to provide students with a sequential, graphical version of any of the common image transformations (e.g., color-to-grayscale, sepia tinting, resizing, brightening, sharpening, blurring, etc.) and have the students time the operation. If they then parallelize the operation's processing loop and rerun the program, they will see and experience the difference in speed and behavior between the original sequential version and their parallel version. By interactively varying the number of threads in such a parallel program and seeing the result, first-year students can develop an intuitive understanding of abstract concepts like the *Parallel Loop* pattern, scalability, speedup, and so on.

3.1.2. Numerical integration: a different balanced-load Parallel Loop

For CS students who have had integral calculus, integration should be a familiar concept, and they may be interested in learning how integration can be performed computationally. While there are a variety of methods that can be used, one that is easy for students to understand is to compute the area between the function's graph and the x-axis for the specified range of x values. A pseudo-code algorithm to compute the integral of `f(x)` from `a` to `b` using the "rectangle method" might be given as follows:

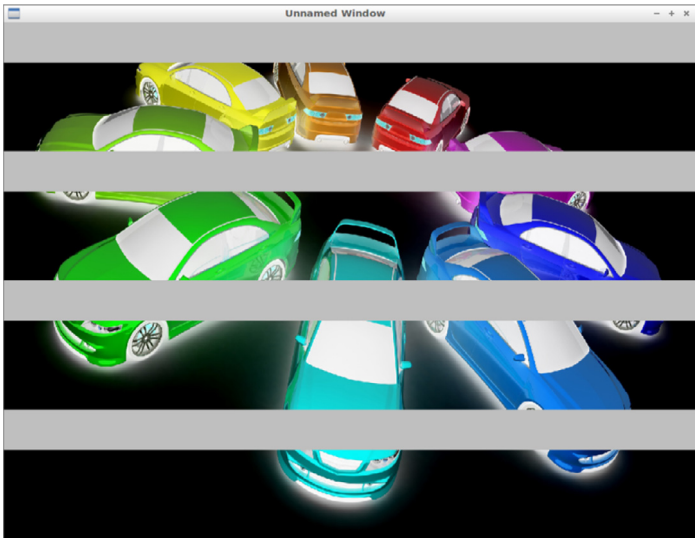

Fig. 4. Color inversion using four threads: In progress. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

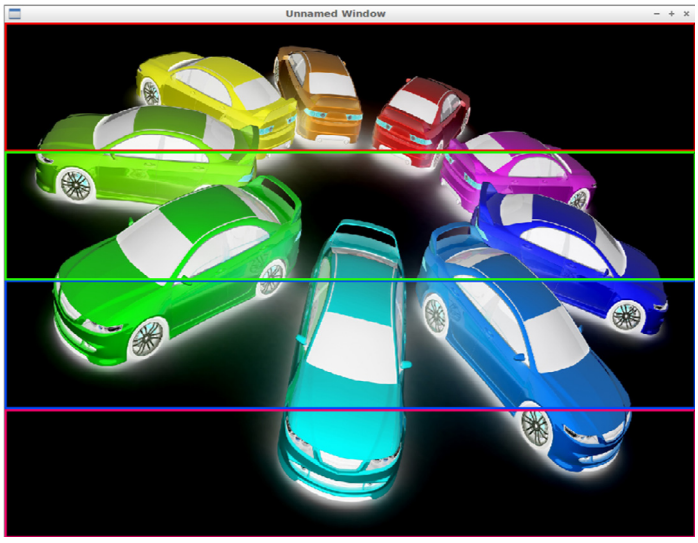

Fig. 5. Color inversion using four threads: Finished. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

```
heights = 0.0;
recWidth = (b-a) / NUM_RECTANGLES;
halfRecWidth = recWidth / 2.0;
for (i = 0; i < NUM_RECTANGLES; i++) {
   xLo = a + i * recWidth;
   xMid = xLo + halfRecWidth;
   y = f(xMid);
   heights += y;
}
return heights * recWidth;
```

For each rectangle, the algorithm's loop accumulates the sum of the rectangles' "heights" (from the rectangle's midpoint on the x-axis up to the curve) in the variable `heights`. When the loop is completed, we multiply those accumulated "heights" by a rectangle's width to compute the return value.

To convert this algorithm into a parallel algorithm, we might again use OpenMP and the *Parallel Loop* pattern. To help students see how the algorithm works, we can use TSGL to (i) give each thread a color, and (ii) have the thread draw its rectangles, as shown in the following pseudo-code:

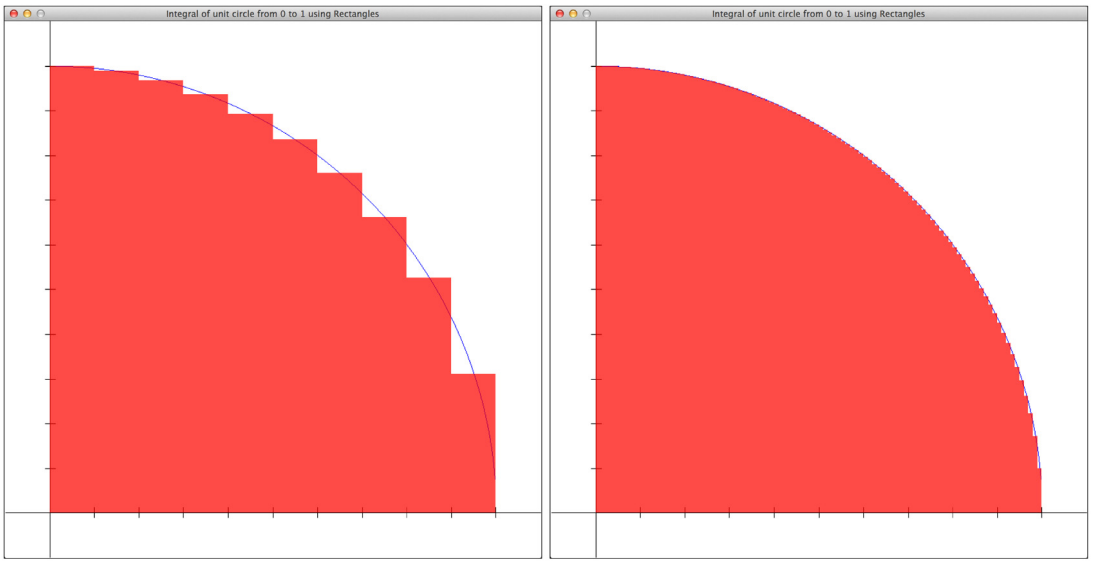

Fig. 6. Integration with one thread: (a) 10 rectangles; (b) 100 rectangles. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

```
heights = 0.0;
recWidth = (b-a) / NUM_RECTANGLES;
halfRecWidth = recWidth / 2.0;
CartesianCanvas canvas(MAX_X, MAX_Y);
canvas.showAxes();
canvas.drawFunction(f);
#pragma omp parallel reduction(+:heights)
{
 threadID = getThreadID();
 Color color = canvas.getMyColor(threadID);
 #pragma omp for
 for (i = 0; i < NUM_RECTANGLES; i++) {
   xLo = a + i * recWidth;
   xMid = xLo + halfRecWidth;
   y = f(xMid);
   canvas.drawRec(xLo, 0, xLo+recWidth, y, color);
   heights += y;
 }
}
return heights * recWidth;
```

That is, we (1) create a *CartesianCanvas* object that all threads will share; (2) tell that canvas to display its axes; and (3) tell that canvas to draw the function we are integrating. We then (4) direct OpenMP to create a parallel block, launching new threads; (5) have each thread retrieve its id number; (6) use that id number to give each thread a unique color; and (7) direct OpenMP to divide the iterations of the for loop among the threads launched in step 4. Within the loop, (8) each thread draws its current rectangle on the canvas, using its unique color.

Fig. 6 shows two screenshots of a running implementation of this algorithm, using a quarter of the unit circle function for `f()` and one thread. In the left shot, `NUM_RECTANGLES` is 10; in the right shot, its value is 100.

Since a single thread is performing the integration, each rectangle is drawn using the same color (red). This program lets us specify the number of rectangles and threads from the command-line, so Fig. 7 shows the same computation using 10 rectangles with two vs. four threads.

As before, TSGL lets us see how the *Parallel Loop* pattern works. Fig. 7a shows that for two threads, the pattern divides the iteration range into two contiguous "chunks"; Fig. 7b shows that for four threads, it divides the range into four such "chunks". Since each thread is coloring the rectangles in its "chunk" using its unique color, we can infer (and verify) that for n threads, the pattern divides the iteration range into n contiguous "chunks". It is also easy to see how this pattern divides the iterations when they are not evenly divisible by the number of threads, as shown in Fig. 7b.

By letting us color-code each thread differently, TSGL lets us readily see how a computation's workload is being divided among its threads.

3.1.3. The Mandelbrot set: an imbalanced-load Parallel Loop

The Mandelbrot set is a well-known fractal figure. A pseudo-code algorithm to draw it might be simplistically given as follows:

```
Canvas canvas(MAX_X, MAX_Y);
for (y = 0; y < MAX_Y; y++) {
   for (x = 0; x < MAX_X; x++) {
      Color color = mandelColor(x, y);
      canvas.drawPoint(x, y, color);
   }
}
```

In this algorithm, we have hidden the details of computing whether a given (x, y) point is in the Mandelbrot set within a `mandelColor()` function. This function is sufficiently time-consuming (i.e., it contains another loop) that on many computers, one can see the individual rows of the figure being drawn. On newer and faster computers, the figure may be drawn too quickly to see this, but TSGL lets us slow the computation sufficiently to see this occur.

As with the image processing and integration examples, the *Parallel Loop* pattern can be used to divide the iterations of this algorithm's outer loop into "chunks" performed by different threads. However, unlike our previous examples, the time to process an (x, y) point in the Mandelbrot figure can vary widely. TSGL lets

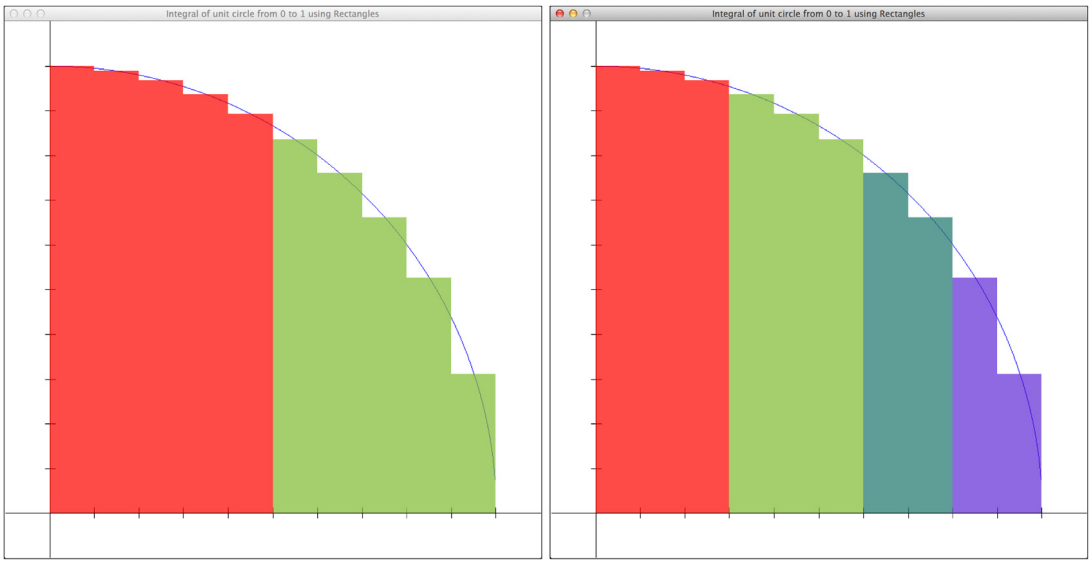

Fig. 7. Integration with 10 rectangles: (a) Using two threads; (b) Using four threads. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Fig. 8. Mandelbrot using eight threads; In progress.

students see this time-variance; if we use the *Parallel Loop* pattern and eight threads, we observe behavior like that shown in Fig. 8.

Like Figs. 4, 8 shows the computation in an intermediate state. However, where the threads in Fig. 4 had made equal progress, the eight threads in Fig. 8 have not. More precisely, threads 0 and 1 have completed their chunks (the top two eighths of the figure), and threads 6 and 7 have completed their chunks (the bottom two eighths), but threads 2, 3, 4, and 5 are still working on their respective eighths, as indicated by the four gray bands in the middle of the figure.

This behavior occurs because points within the Mandelbrot set (the black portion of the figure) take longer to compute than points outside the set. Since the threads drawing the middle rows have more inside-the-set points to compute, it take them longer to complete their "chunks". TSGL's near real-time drawing can thus let students *see* non-uniform workloads as they occur, which can be used to motivate the introduction of other parallel design patterns that do a better job of balancing the different threads' work-loads.

When finished, we have each thread draw a colored rectangle around the portion of the figure it drew, as shown in Fig. 9.

3.2. The actor pattern and synchronizing accesses to shared resources

Another parallel design pattern is the *Actor* pattern, in which autonomous actors perform their prescribed behaviors like actors

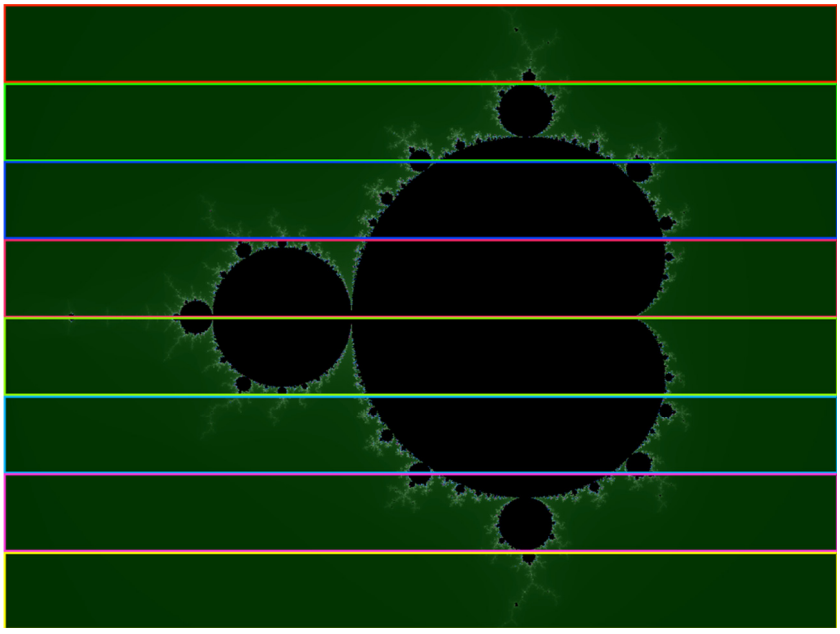

Fig. 9. Mandelbrot using eight threads; Finished. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

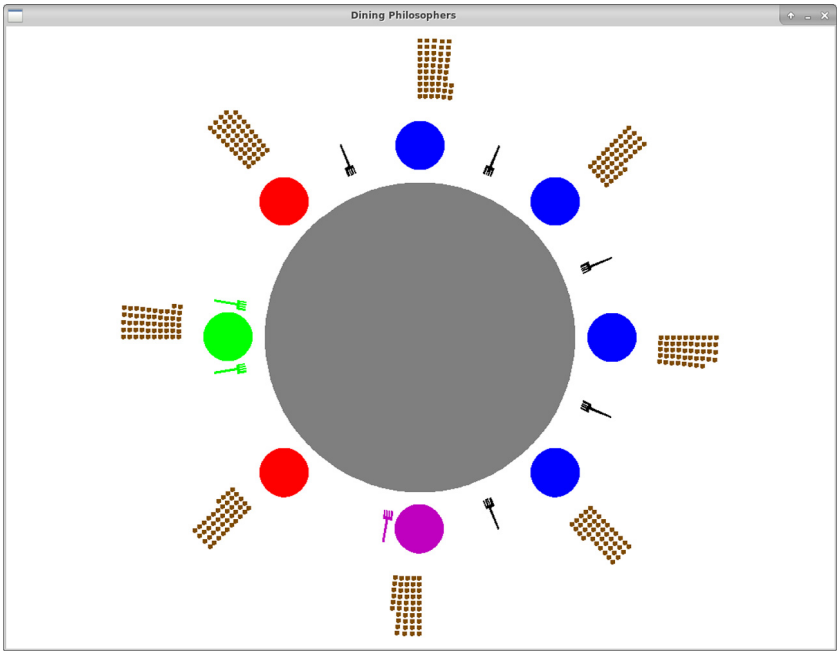

Fig. 10. Visualizing the Dining Philosophers: Eight philosophers. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

in a play. This pattern can be used to solve a variety of problems, including the classic Dining Philosophers problem. In this problem, n silent philosophers sit around a table, with a large bowl of food in the middle. There are n forks on the table, each positioned between a pair of philosophers. A philosopher may be thinking, hungry, or eating, and may think for an arbitrary length of time before becoming hungry, but in order to eat, he or she must acquire both of the adjacent forks. As shared resources, the forks represent a potential source of race conditions. The problem is to devise a strategy that all philosophers can follow, that ensures: (a) no deadlock occurs, (b) no livelock occurs, (c) no philosopher starves, and (d) that a philosopher who is thinking does not prevent a philosopher who is hungry from eating.

Each actor in the *Actor* pattern has its own thread; to use this pattern to solve the Dining Philosophers problem, each philosopher has its own thread that performs the strategy in parallel with the others. TSGL makes it possible to augment a strategy with graphics calls, using color-coding to represent a philosopher's state, and thus create a visualization that lets students see that strategy running in near real-time. Fig. 10 shows our visualization of this problem using eight philosopher-threads, in which the large central gray circle represents the table, the eight circles around the table represent the philosophers, the eight forks can be seen

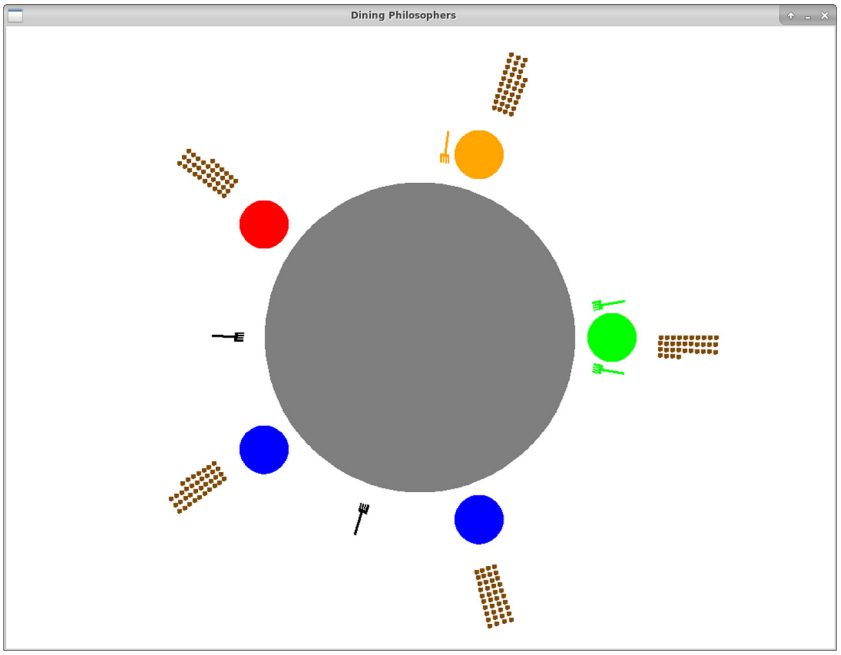

Fig. 11. Visualizing the Dining Philosophers: Five philosophers.

between the philosophers, and the small brown dots "behind" each philosopher tally the meals she has eaten.

A separate "Legend" window (not shown) indicates the strategy being used and the state each color represents: blue indicates thinking, red indicates hungry with no forks, purple indicates hungry with the left fork but not the right, orange indicates hungry with the right fork but not the left, and green indicates eating. When a fork is free, it is equidistant between two philosophers; when a philosopher has obtained a fork, the fork is adjacent to that philosopher.

If we use the eight compass points to denote the philosophers in Fig. 10, the north, northeast, east, and southeast philosophers are all thinking, as indicated by their blue colors. The south philosopher is hungry and has obtained its left fork, but has not yet picked up its right fork. The southwest philosopher is hungry but cannot eat until her neighboring philosophers release their forks. The west philosopher has obtained both forks and is eating, as indicated by her green color. The northwest philosopher is hungry but has not yet picked up the fork between her and her northern neighbor. The brown meal dots behind each philosopher let us see at a glance that no philosopher is starving.

The number of philosophers can be specified when the program is run; Fig. 11 shows it running with five philosophers.

In Fig. 11, one philosopher is eating, two are thinking, one is hungry but has acquired no forks, and one is hungry but has acquired its right fork. Again, the brown "meal" dots behind each philosopher indicate that no philosopher is starving, and that each is eating about as often as its peers. Some variance is normal, as each philosopher thinks for a random length of time.

This visualization also lets students interactively explore incorrect strategies. For example, Fig. 12 shows a version of the program in which the philosophers follow a strategy that leads to *deadlock*.

In Fig. 12, each philosopher has acquired its right fork and is waiting to acquire its left fork. In this (incorrect) strategy, a philosopher thinks for a while and when she gets hungry: (1) attempts to grab her right fork, (2) attempts to grab her left fork, (3) eats, and (4) releases her fork when she is done eating. If each philosopher happens to get hungry at the same time, a circular wait happens, resulting in deadlock.

Relatedly, this program lets students interactively specify a different incorrect strategy that produces *livelock*. In this strategy, a philosopher thinks for a while and when she gets hungry, she: (1) attempts to grab her right fork, (2) attempts to grab her left fork and if it is not available, releases her right fork, (3) eats, and (4) releases her fork when she is done eating. Again, if each philosopher happens to get hungry at the same time, each picks up her right fork, puts down her right fork, picks up her right fork, puts down her right fork, …endlessly. The visualization lets us see that behavior, as the philosophers endlessly cycle between the two red and orange states shown in Fig. 13.

TSGL thus makes it possible to create visualizations for "classic" synchronization problems like the Dining Philosophers problem. We are currently working on visualizations of other "classic" problems: the Producers–Consumers problem, the Readers–Writers problem, the Sleepy Barber problem, and so on.

4. Assessment

Since 2012, we have devoted week 12 (of our 15-week semester) in our CS2 (*Data Structures*) course to OpenMP multithreading [1]. This week includes three classroom sessions, plus a hands-on lab session in which students use the *Parallel Loop* pattern to speed up slow operations, and measure their runtimes using 1, 2, 4, 6, 8, 10, 12, 14, and 16 threads. To assess our students' long-term recall of this material, our final exam includes four questions that explore their understanding of:

1. What OpenMP is.
2. How an OpenMP parallel block behaves.
3. Which thread behaviors produce race conditions.
4. How an OpenMP thread can discover its identity number.

The 52 students in our Spring 2015 course were spread across two lab sections of 27 and 25 students. To assess the effect of TSGL-visualization on our students' long-term recall of the *Parallel Loop* pattern, we ran an experiment using these two lab sections as the control and experimental groups, respectively:

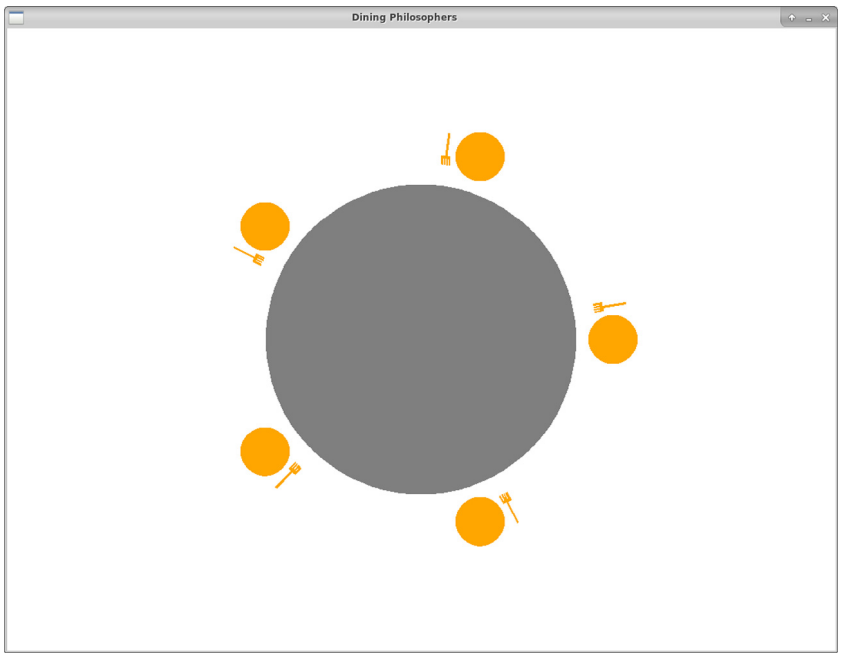

Fig. 12. Visualizing the Dining Philosophers: Deadlock.

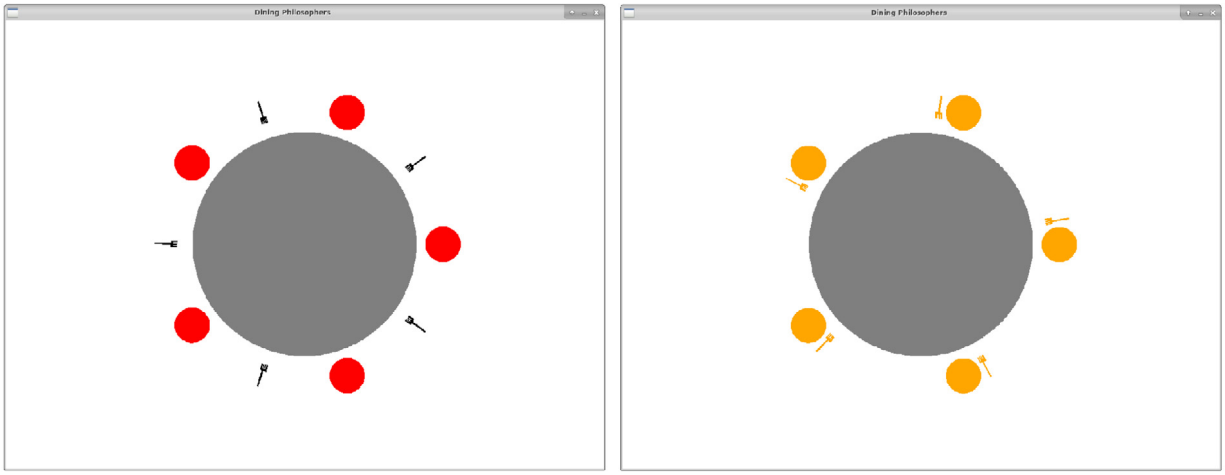

Fig. 13. Visualizing the Dining Philosophers: Livelock. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

- Group 1 (the control group) completed the same exercise as the students in preceding semesters, using OpenMP to speed up the addition and transpose operations for large matrices.
- Group 2 (the experimental group) used OpenMP to speed up the inversion of a large image, using an interactive TSGL visualization to see the running computation.

The research question we were seeking to answer with this experiment was as follows:

Does visualizing the behavior of the Parallel Loop *pattern improve a student's long-term recall of that pattern's behavior?*

Both lab exercises used the *Parallel Loop* pattern in almost identical ways:

- Both exercises' programs use nested for loops to process a two-dimensional data structure.
- Both exercises have students parallelize the computation by adding the same `#pragma omp parallel for` directive before the outer for loop.
- Both exercises' have the students modify the programs to output the time required to perform the operation.
- Both exercises have the students run the program multiple times, interactively varying the number of threads; recording the resulting times in a spreadsheet, and creating a chart that plots the execution time against the number of threads, so that the students can see how the run-times decrease as more threads are used.
- Both exercises' programs output the resulting data structure.
- Both exercises included static figures depicting the parallel loop pattern's behavior. In the control group, these figures showed how the pattern would divide the rows of a matrix among the threads; in the experimental group, these figures showed how the pattern would divide the rows of an image among the threads.

The primary difference in the two exercises was that in our experimental group, the program's output let students to *see* the image being inverted in near real-time (see Fig. 4) and *see* this behavior change as they varied the number of threads. By contrast, the output in our control group was simply the final matrix, after the operation had completed.

To answer our research question, we added a fifth question to the final exam (but we did not count it toward a student's exam score). Using the final exam to assess the results of our experiment had two benefits:

1. Since all students in the course take the final exam, using the exam question as our assessment instrument guaranteed us a high response rate. Given the relatively small numbers of students in our control and experimental groups, a high response rate was important to maximize the chances of producing statistically significant results.
2. Using the final exam as our assessment instrument maximized the time between our experiment and our assessment. This was important because we were evaluating the effect of the visualization on our students' long-term recall.

The question we added to our final exam is shown below:

5. Suppose the following for loop is performed by two threads:

```
vector<int> v(8);
#pragma omp parallel for
for (int i = 0; i < v.size(); i++) {
   cout << v[i] << endl;
}
```

Which of the following is true?

A) Thread 0 will output the items in v[0] through v[3];
thread 1 will output items in v[4] through v[7].

B) Thread 0 will output the items in v[0], v[1], v[6], v[7];
thread 1 will output the items in v[2], v[3], v[4], v[5].

C) Thread 0 will output the items with even index values (0,2,4,6);
thread 1 will output the items with odd index values (1,3,5,7).

D) Thread 0 will output the items with odd index values (1,3,5,7);
thread 1 will output the items with even index values (0,2,4,6).

E) Thread 0 will output four items with random index values;
thread 1 will output the remaining items.

F) None of these is true.

Note that where the exercises performed by our control and experimental groups both used OpenMP's *Parallel Loop* pattern to process a two-dimensional data structure, question 5 requires the students to recognize how that same pattern will process a one-dimensional data structure. It thus strips away all contextual overlap except the behavior of the *Parallel Loop*.

Our students' final exam scores formed an approximately normal distribution. Since our goal was to determine whether the visualization had a positive effect for the students in our experimental group, we only needed a test that would evaluate differences in one tail (i.e., the positive end) of the distribution. We accordingly used a 1-tailed T-test to examine the significance of any difference in the mean of the students' responses to our five questions. Our null hypotheses were that students in the two groups would do equally well on each of the five exam questions. Fig. 14 summarizes our two groups' performances on the five exam questions, as well as the statistical significance (p-values) of their differences.

These results led us to accept the null hypothesis for the original four questions, but led us to reject the null hypothesis for question #5. Note that both groups interactively explored the use of the *Parallel Loop* pattern to speed up the processing of a 2-dimensional

Exam Question	Group 1 (Control)	Group 2 (Experimental)	p-value
1	77.8%	85.0%	*0.26764*
2	85.2%	80.0%	*0.32762*
3	18.5%	30.0%	*0.19133*
4	88.9%	85.0%	*0.23201*
5	**7.4%**	**40.0%**	***0.00685***

Fig. 14. Percentage of correct question–responses by group.

	Control Group		Experimental Group	
Question 5 Answers	# of Responses (out of 27)	Percentage of Responses	# of Responses (out of 25)	Percentage of Responses
A	2	7.4%	10	40%
B	0	0%	0	0%
C	11	40.7%	5	20%
D	1	3.7%	4	16%
E	12	44.4%	5	20%
F	1	3.7%	1	4%

Fig. 15. Table of student responses to question 5.

structure, but the experimental group could see the behavior occurring where the control group could not. The difference in the two groups' performances on question 5 thus strongly suggests that seeing a multithreaded behavior like the *Parallel Loop* pattern can significantly improve a student's understanding of such abstractions, compared to a non-visual exercise in which a student must build his or her own mental model of the parallel behavior.

Fig. 15 provides a table that breaks down the students' responses to question 5.

As can be seen in Fig. 15, the plurality of the students in the experimental group answered question 5 correctly; most of this group's incorrect responses are almost equally distributed between answers C (20%), D (16%), and E (20%). It is unclear exactly why these students were confused (especially those who chose D), but we discuss some possibilities below.

By contrast, 44.4% of the students in the control group chose answer E, which is surprising, given its "random" aspect. During our lecture sessions that week, we live-demoed OpenMP *Parallel Loop* patternlets [2] that, for each iteration i, cause the thread performing that iteration to display its id number and the value of the iteration number i. In the patternlet for the *Parallel Loop* 's default behavior, each thread is assigned a contiguous equal-sized "chunk" of the iteration range (as reflected in answer A), but each thread writes its output to `stdout`, where these outputs are interleaved non-deterministically. After the final exam, we talked to some of the students in the control group. From these conversations, it appears that in the absence of a visualization that lets one see the default "chunking" behavior, these students (along with 20% of the students in our experimental group) attributed the nondeterministic interleaving of the different threads' outputs to the *Parallel Loop* pattern, and so never understood the essence of the "chunking" behavior.

Nearly as many control group students (40.7%) chose answer C. In addition to live-demoing the default "chunking" *Parallel Loop* patternlet in our lectures, we also live-demoed the "striping" version of the *Parallel Loop* patternlet, in which a loop's iterations are divided in a round-robin fashion (as reflected in answer C). We hypothesize that, lacking a visualization that lets them see the default, behavior, these students (along with 20% of the students in

our experimental group) misremembered and thought that "striping" was this pattern's default behavior.

The reader may be wondering why so few students answered question 3 correctly, compared to questions 1, 2, and 4? The reason is that the best answer to that question was an "all of the above" choice. Nearly all students chose an answer that was at least partially correct, but only the percentages shown in Fig. 14 chose the best answer. Note that 30% of our experimental group chose the best answer compared to 18.5% of the control group, but this improvement was not significant ($p = 0.19133$).

We also compared our two groups' aggregate responses to questions 1–4. There were no significant differences in the correctness of their responses to these questions ($p = 0.36166$).

We also compared our two groups' responses on questions 1–4 to the responses of the 183 students who took the course (and did the matrix exercise) the previous six semesters. There were no significant differences between the correctness of the responses of students from previous semesters and our control group ($p = 0.19232$), nor our experimental group ($p = 0.42134$).

Our TSGL visualization thus had no measurable significant effect on student mastery of the parallel concepts assessed by our four original exam questions, but it did significantly improve student understanding of the *Parallel Loop*'s default behavior. Because of this, we now have all students complete our visual image-processing exercise instead of our non-visual matrix-processing lab exercise.

Both groups used the same *Parallel Loop* pattern to accelerate an operation on a 2-dimensional data structure, but the fact that our control group's structure was a *matrix* while our experimental group's structure was an *image* represents a potential confounding factor: perhaps that difference is somehow responsible for the improved learning shown on question 5? This question might be resolved by conducting a follow up experiment in which two groups both parallelize an operation on an image, but where one group gets to see the parallelization happening in real-time (i.e., while the operation is running); the other group only gets to see the transformed image after the processing has completed. We conjecture that such an experiment will produce results similar to those reported here; if we conduct such an experiment, it will be the subject of a future report.

5. Discussion

In this section, we discuss other aspects related to TSGL and multithreaded visualization.

5.1. Visualizing other parallel patterns

We have seen that TSGL makes it possible to visualize parallel computing abstractions like the *Parallel Loop* and the *Actor* patterns. This raises the question: Can TSGL be used to create visualizations of other parallel abstractions?

We believe the answer to this question is "Yes" and that the potential of TSGL is mainly limited by our creativity. For example, we have begun work on a visualization of the *Task Queue* pattern, as follows:

a. Open a *Canvas* whose background is white and whose width is proportional with m, the length of the queue;
b. On the *Canvas*, draw m black rectangles, each representing one of the queue's tasks, saving a reference to each rectangle in a shared queue.
c. Each time a thread gets a task from the task queue, use the shared queue to change the color of the corresponding rectangle from black to white.

At the outset, the *Canvas* will show m black rectangles, but as tasks are removed from the task queue, the corresponding rectangles will 'disappear' into the white background. The result will be a kind of "reverse progress bar" that grows shorter as the length of the task queue decreases. Alternatively, in step (c), a thread could change the color of the rectangle to that thread's unique color. At the end, the distribution of colored rectangles on the *Canvas* would reflect the distribution of the tasks among the threads. We plan to use usability testing to decide which of these two approaches is preferable.

As a second example, we are working on a TSGL visualization of the parallel Merge Sort algorithm. This and similar visualizations will let students see how the sequential and parallel versions of an algorithm differ in behavior.

5.2. Guidelines for creating effective visualizations

To create our multithreaded visualizations, we have adopted the following guidelines:

1. The visualization must work *correctly* using 1 or more threads.
2. The visualization must be *interactive*, allowing one to change the number of threads being used without recompiling.
3. The visualization must exhibit *scalability* — its behavior must change as the number of threads is changed, in appearance (e.g., see Figs. 7, and 10 versus 11) and where appropriate, execution-time.

Each example described in Section 3 follows these guidelines, allowing a student to explore the program's behavior using differing numbers of threads and experience the result. TSGL's near real-time graphics let a student actually *see* a scalable program run faster as the number of threads is changed from 1 to 2 to 3 to 4 to …We have found that *seeing* a program run faster motivates many students to quantify the speedup and calculate precisely how much faster the program is running. TSGL appears to have great potential for creating visualizations that let students viscerally experience multithreaded behavior.

5.3. Testing and debugging

Visualizing an algorithm's behavior provides an alternative means of testing one's code. As one example, the small brown "meals" circles in Figs. 10 and 11 indicate that none of our dining philosophers are starving using that particular strategy. By contrast, Fig. 12 shows how the use of a different strategy leads our philosophers to a deadlocked state, and Fig. 13 shows how the use of another strategy leads our philosophers to a livelocked state; the lack of any "meals" dots in these executions show that no philosopher has eaten, thus showing how these strategies lead to starvation.

A second example occurred when we first implemented the integration example in 3.1.2. Integration is an inexact computation: the precision of the result varies with the number of rectangles being used, so our unit test checked that the area computed by our function was computing the correct result plus or minus a certain threshold. In our initial implementation, our for loop contained a "off-by-one" logic error that caused it to use one more rectangle than it should. This extra rectangle was very short so the area of this extra rectangle was too small to exceed our unit test's threshold,

and our function still passed our unit test. When we added the TSGL calls to visualize the algorithm's operation, this "off-by-one" error was immediately visible, as we could *see* the extra rectangle being drawn. TSGL and visualization thus let us find and fix a logic error that we were unable to detect via a unit test checking our function's return-value. Testing and debugging programs was not a design goal of TSGL, but it appears to be a beneficial side effect of visualization.

5.4. Controlling execution speed

In Section 2.1, we indicated that one of our design goals for TSGL was to be able to slow down or delay a thread, if that was desirable. For example, if run on the hardware in a typical laptop without slowing the computation, each of the examples presented in Section 3 will happen too quickly to be useful as instructional examples:

1. In both the image-processing and the integration examples, a single-threaded program will display the figure so quickly that a student will be unable to see any differences between it and any of the multithreaded versions.
2. In the Mandelbrot example, the figure will be displayed too quickly for a student to see the need for load-balancing — that the threads drawing the rows at the top and bottom of the figure finish much faster than those drawing the middle rows.
3. In the Dining Philosophers example, the philosophers will go through their thinking–hungry–eating cycle too quickly for a student to see the philosophers change from one state to another.

For situations like this, TSGL lets the creator of a visualization control the rate at which a visualization is drawn. More precisely, each TSGL *Canvas* has a **drawTimer** instance variable, whose value is used to delay the *Canvas*'s render-thread each of its draw-cycles. The TSGL *Canvas* constructors each have a **timerLength** parameter whose value is used to initialize this instance variable. The **timerLength** parameter has a default argument of 0.0, so that if no explicit argument is passed, the **drawTimer** is initialized to 0.0, the *Canvas*'s render-thread is delayed 0.0 s each draw-cycle, and thus operates at its maximum possible speed. But the creator of a visualization may pass an explicit value to this parameter to delay the render-thread each of its draw-cycles. For convenience, TSGL defines two global constants: **FPS** (frames per second) equal to 60 and **FRAME** set to **1.0/FPS**. By passing **FRAME** to the *Canvas* constructor's **timerLength** argument, the visualization's frame-rate can be set to 60 frames per second:

```
Canvas canvas(MAX_X, MAX_Y, FRAME);
```

By passing different values to the **timerLength** parameter, one can control the frame rate at which a visualization displays.

The TSGL *Canvas* class also provides a **sleep()** method that causes a thread executing it to delay until the *Canvas*'s render-thread has completed its current draw-cycle. This can be used to control the speed of a visualization's threads. For example, to slow the execution of the numerical integration example from Section 3.1.2, we would invoke this **sleep()** method within the for loop that draws the rectangles, as shown below:

```
heights = 0.0;
recWidth = (b-a) / NUM_RECTANGLES;
halfRecWidth = recWidth / 2.0;
CartesianCanvas canvas(MAX_X, MAX_Y, FRAME);
canvas.showAxes();
canvas.drawFunction(f);
#pragma omp parallel reduction(+:heights)
{
 threadID = getThreadID();
 Color color = canvas.getMyColor(threadID);
 #pragma omp for
 for (i = 0; i < NUM_RECTANGLES; i++) {
   canvas.sleep();
   xLo = a + i * recWidth;
   xMid = xLo + halfRecWidth;
   y = f(xMid);
   canvas.drawRec(xLo, 0, xLo+recWidth, y, color);
   heights += y;
 }
}
return heights * recWidth;
```

Since the **#pragma omp for** directive divides the iteration-range of the for loop among the threads, the first thing the threads will do upon entering the loop-body is sleep until the *Canvas*'s render-thread has finished its current draw-cycle. When that happens, the threads will wake, compute and draw the first rectangle, return to the top of the loop, and then sleep again. The effect is thus to slow the threads' executions, synchronizing them with the render-thread's draw-cycle. Since we can control the length of that draw-cycle by passing an argument to the *Canvas* constructor's **timerLength** parameter, we can control the speed at which a visualization runs.

As a second illustration, the following parallel version of the Mandelbrot algorithm from Section 3.1.3 will cause each thread to sleep when it computes a given row of the Mandelbrot figure:

```
Canvas canvas(MAX_X, MAX_Y, delay);
#pragma omp parallel for
for (y = 0; y < MAX_Y; y++) {
   canvas.sleep();
   for (x = 0; x < MAX_X; x++) {
      Color color = mandelColor(x, y);
      canvas.drawPoint(x, y, color);
   }
}
```

By inserting a call to the *Canvas*'s **sleep()** method between the two loops, a thread sleeps before it draws each row of the Mandelbrot figure. Since this happens for each row, but the outer rows take less time to draw than the center rows, this will slow the visualization sufficiently for an observer to see the non-uniform workloads of the threads.

To provide more fine-grained control, the *Canvas* class also provides a **sleepFor**(***seconds***) method. This method causes the thread that calls it to sleep for the specified number of (double precision) ***seconds***, regardless of the value of the *Canvas*'s **drawTimer** member, thus delaying a thread independently of the visualization's draw cycle.

For more information about using TSGL, we encourage the reader to consult the tutorials that are available at the TSGL website [4].

6. Conclusions

To extend the adage "*A picture is worth 1000 words*", we believe that an effective way to teach students about parallelism is to take a working multithreaded program and augment it with graphical calls that show what each thread is doing, to create an interactive visualization of the program's parallel behavior. The behavior of such visualizations appears to help the students build mental models of the underlying parallelism, and the interactivity lets them explore and alter that parallel behavior, improving their mastery of abstract parallel concepts.

To help ourselves and others create such visualizations, we have built the thread-safe graphics library (TSGL). TSGL is an object-oriented library whereby C++11, POSIX, and/or OpenMP threads can safely draw on one or more common *Canvas* object, and one can see the results in near real-time.

To illustrate the use of TSGL, we have presented several examples, including image processing, numerical integration, the Mandelbrot Set, and the Dining Philosophers Problem. We have also described others that are currently under construction.

We have presented evidence that the use of a TSGL visualization improved student understanding of the precise behavior used by the *Parallel Loop* pattern. By helping students build mental models of parallel behavior, we believe that TSGL visualizations hold great potential to help students understand abstract concepts, whether parallel or sequential.

Finally, we have discussed different ways TSGL can be used, including the unexpected discovery that visualization can aid testing and debugging. We also presented our guidelines for creating multithreaded visualizations: they should be *correct*, *interactive*, and *scalable*, so that students can see how the behavior of the parallel algorithm changes as they vary the number of threads.

For those who would like to try it, the current release of TSGL may be freely downloaded from GitHub (see [4]) under the GNU Public License (v. 3). It was developed on Ubuntu Linux 14.0.4 and has been successfully tested on both MacOS X Yosemite and Windows 7. The project's Github site includes installation instructions, nine tutorials on how to use the library, and its API. We look forward to seeing the new and exciting visualizations that others will create using TSGL.

Acknowledgments

We wish to thank the National Science Foundation, whose support through grant DUE-1225739 made TSGL possible; our collaborators on the CSinParallel project [6], who have provided valuable feedback during the development of TSGL; and the reviewers of initial drafts of this paper, whose feedback significantly strengthened it.

References

[1] J. Adams, Injecting parallel computing into CS2, in: Proc. 45th SIGCSE Tech. Symposium on Computer Science Education, Mar. 2014, pp. 277-282. http://dx.doi.org/10.1145/2538862.2538883.

[2] J. Adams, Patternlets: A teaching tool for introducing students to parallelism, J. Parallel Distrib. Comput. 105 (2017) 31–41. http://dx.doi.org/10.1016/j.jpdc.2017.01.008.

[3] J. Adams, R. Brown, E. Shoop, Patterns and exemplars: Compelling strategies for teaching parallel and distributed computing to CS undergraduates, in: EduPar-13, 27th IEEE International Parallel and Distributed Processing Symposium, Boston, MA, May 2013, pp. 1244–1251. http://dx.doi.org/10.1109/IPDPSW.2013.275.

[4] J. Adams, P. Crain, C. Dilley, M. VanderStel, TSGL. Online: github.com/Calvin-CS/TSGL (Accessed 12.06.17).

[5] S. Bogaerts, Hands-on exploration of parallelism for absolute beginners with scratch, in: EduPar Workshop, Proc. 27th IEEE International Parallel and Distributed Processing Symposium, IPDPSW 2013, May 2013, p. 1263. http://dx.doi.org/10.1109/IPDPSW.2013.63.

[6] R. Brown, E. Shoop, J. Adams, S. Matthews, CSinParallel: Parallel Computing in the Computer Science Curriculum. Online: csinparallel.org (Accessed 12.06.17).

[7] E. Fouh, M. Akbar, C. Shaffer, The role of visualization in computer science education, Comput. Schools: Interdiscip. J. Pract. Theory Appl. Res. 29 (2012) 95–117. http://dx.doi.org/10.1080/07380569.2012.651422.

[8] D. Garcia, et al. Beauty and Joy of Computing Curriculum: Concurrency. Online: http://bjc.berkeley.edu/bjc-r/topic/topic.html?topic=berkeley_bjc/areas/concurrency.topic (Accessed 30.11.17).

[9] Intel Corp. Intel Xeon Phi Processors. Online: www.intel.com/content/www/us/en/products/processors/xeon-phi/xeon-phi-processors.html (Accessed 12.06.17).

[10] Intel Corp. Intel Xeon Processors. Online: www.intel.com/content/www/us/en/products/processors/xeon.html (Accessed 12.06.17).

[11] C. Kehoe, J. Stasko, A. Taylor, Rethinking the evaluation of algorithm animations as learning aids, Int. J. Hum.-Comput. Stud. 2 (54) (2001) 265–284. http://dx.doi.org/10.1006/ijhc.2000.0409.

[12] K. Keutzer, B.L. Massingill, T.G. Mattson, B.A. Sanders, A design pattern language for engineering (parallel) software: merging the PLPP and OPL projects, in: Proceedings of the 2010 Workshop on Parallel Programming Patterns, New York, NY, USA, 2010, pp. 9:1–9:8.

[13] K. Keutzer, T. Mattson, Our pattern language (OPL): A design pattern language for engineering (parallel) software, in: ParaPLoP Workshop on Parallel Programming Patterns, 2009.

[14] S. Massung, C. Heeren, Visualizing parallelism in CS2, in: Third NSF/TCPP Workshop on Parallel and Distributed Computing Education, EduPar-13, May 2013. Online: grid.cs.gsu.edu/~tcpp/curriculum/sites/default/files/VisualizingParallelisminCS2_0.pdf (Accessed 12.06.17).

[15] T. Mattson, B. Sanders, B. Massingill, Patterns for Parallel Programming, Pearson Education, 2005.

[16] T. Naps, G. Rosling, V. Alstrum, W. Dann, R. Fleischer, C. Hundhausen, A. Korhonen, L. Malmi, M. McNally, S. Rodger, J. Valezquez-Iturbide, Exploring the role of visualization and engagement in computer science education, ACM SIGCSE Bull. 35 (2) (2003) 131–152. http://dx.doi.org/10.1145/782941.782998.

[17] S.K. Prasad, A. Chtchelkanova, F. Dehne, M. Gouda, A. Gupta, J. Jaja, K. Kant, A. La Salle, R. LeBlanc, A. Lumsdaine, D. Padua, M. Parashar, V. Prasanna, Y. Robert, A. Rosenberg, S. Sahni, B. Shirazi, A. Sussman, C. Weems, J. Wu, NSF/IEEE-TCPP Curriculum Initiative on Parallel and Distributed Computing - Core Topics for Undergraduates, Version I, Online: http://www.cs.gsu.edu/~tcpp/curriculum/index.php, 2012, p. 55.

[18] M. Sahami, et al., "Computer Science Curricula 2013," http://dx.doi.org/10.1145/2534860. Online: www.acm.org/education/CS2013-final-report.pdf (Accessed 12.06.17).

[19] C. Shaffer, M. Cooper, A. Alon, M. Akbar, M. Stewart, S. Ponce, S. Edwards, Algorithm Visualization: The State of the Field, ACM Trans. Comput. Educ. 10 (3) (2010). http://dx.doi.org/10.1145/1821996.1821997. Article no. 9.

[20] S. Torbet, U. Vishkin, R. Tzur, D. Ellison, Is teaching parallel algorithmic thinking to high school students possible? one teacher's experience, in: Proc. 41st ACM Technical Symposium on Computer Science Education, March 2010, pp. 290-294. http://dx.doi.org/10.1145/1734263.1734363.

Joel C. Adams is professor and Chair of the Department of Computer Science at Calvin College. He has been teaching his students about parallel and distributed computing since 1997. He has designed four Beowulf clusters including Microwulf, the first cluster to break the $100/GFLOP barrier. He is one of the PIs on CSinParallel.org, an NSF-funded project to create and distribute high quality pedagogical materials for teaching students about parallel and distributed computing. He is a two-time Fulbright scholar (Mauritius, 1998; Iceland, 2005) and is an ACM Distinguished Educator.