2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)

ParaVis: A Library for Visualizing and Debugging Parallel Applications

Andrew Danner, Tia Newhall, Kevin C. Webb
Computer Science Department, Swarthmore College
Swarthmore, PA USA
{*adanner, newhall, kwebb*}*@cs.swarthmore.edu*

***Abstract*—This paper presents ParaVis, a visualization library designed to aid programmers' understanding of their parallel programs and to help them identify bugs with parallelization. ParaVis is particularly targeted for programmers who are first learning parallel programming or learning a new parallel language. It provides easy-to-use C and C++ interfaces to create 2D animations of parallel computation that help programmers understand parallel data decomposition patterns. These visualizations are also helpful in illustrating errors in parallel programs. Additionally, because students often find visualization fun, the use of our library often results in students developing interesting extensions to problems, thus promoting a deeper understanding and richer experience with parallel computing. Currently we provide support and sample implementations for pthreads, OpenMP, CUDA, and sequential applications. To test its effectiveness for parallel computing education, we deployed ParaVis for lab assignments in both intermediate and upper level courses. We present example applications, and evaluate the use of the library across our undergraduate CS curriculum.**

***Keywords*-visualization library; parallel computing; CS Education; CUDA; pthreads; OpenMP**

I. Introduction

In the multi-core era, explicit parallel computing is now *required* to take full advantage of common hardware devices (e.g., multi-core CPUs and GPUs) and the increasingly pervasive parallel and distributed platforms (e.g., clusters and the cloud). At the same time, the era of big data and data-intensive computing has expanded the need for parallel and distributed solutions to applications that span an increasingly broad range of fields. Together these trends motivate the growing importance of teaching parallel and distributed computing throughout the computer science curriculum, as reflected in a new parallel and distributed core knowledge area added in the ACM/IEEE Task Force's 2013 CS Curriculum [2]. The increasing importance of teaching parallel and distributed computing throughout the undergraduate curriculum has led to efforts to create parallel and distributed computing curricular resources. Most notably, the NSF/IEEE-TCPP 2012 Curriculum Initiative on Parallel and Distributed Computing [28] defines detailed curricular requirements and recommendations. It supports projects to help faculty increase the coverage of parallel and distributed computing in their curricula.

Unfortunately, developing and debugging parallel programs can often be challenging, especially for programmers learning a new parallel language or paradigm. To help aid in the initial exploration of parallel computing, we present ParaVis, a visualization library that aims to make debugging of parallel programs easier and more enjoyable through 2D animations of a simple color buffer. The use of visualization provides rapid feedback regarding the correctness of the parallel algorithm, particularly when a large number of parallel workers makes printing to the console too verbose to debug efficiently. Additionally, many students find visualization fun, and the use of ParaVis encourages further exploration, experimentation, and potentially a deeper understanding of the underlying parallel language.

ParaVis is designed to be easy to use so that no experience with graphics or image formats is necessary, allowing students and programmers to focus on learning parallel concepts. It currently supports writing visualizations and animations using pthreads, OpenMP, CUDA, or sequential programming techniques. Such flexibility allows programmers to choose a preferred parallel programming model and also easily enables them to make comparisons across multiple models. The library is primarily written in C++, but it's easily extended to support visualizations in C. In the remainder of this paper, we position ParaVis with respect to related work, characterize the library in the context of small examples, and describe our experiences with integrating it into our curriculum.

II. Related Work

There have been numerous efforts to develop specific parallel and distributed undergraduate curricula [6], [16], [23] and resources for teaching [3], [13]. ParaVis fits into this broad area as a tool for teaching new parallel languages and paradigms, currently targeting languages for single machine parallelism (multi-core CPUs or GPUs).

Previous work has shown that visualization often aids in learning programming and in developing computational thinking skills [1], [5], [26]. Some efforts focus on using visualization specifically to teach parallel programming [12], [25] and to illustrate parallelization and parallel algorithms [17], [21], [24].

Like ParaVis, the thread-safe graphics library (TSGL) project [8] implements a library that can be used for multi-threaded graphics applications. A primary motivation for TSGL is to provide a tool for building visualizations that

978-1-5386-5555-9/19/$31.00 ©2019 IEEE
DOI 10.1109/IPDPSW.2019.00062

can help students understand multi-threaded parallelization. While our work is similarly motivated, we target a broader range of parallel languages and programming paradigms beyond threads. ParaVis also provides a higher-level, simple interface that hides graphics library details from the user.

There is also a large body of work on using visualization to aid in correctness verification, debugging, and performance analyses of parallel programs [9], [15], [18], [20]. Such work in parallel tools highlights the necessity of adopting visualization towards understanding performance and correctness in parallel applications. Furthermore, such tools have also been used as a learning resource to help novice students learn parallel computing techniques and understand their parallel computations [19]. While ParaVis is not in the same category of parallel tools development, it similarly helps students to identify the causes behind many bugs as they learn parallel programming and new parallel programming models.

III. Visualization Library

ParaVis is written in C++ using Qt 5.9 and OpenGL 4.1 for the visualization components. A typical user of our library does not need to know either of these rather large and complex software packages. We abstract many of the details of the visualization framework away through a `QTViewer` class that handles creating the graphical window, driving the animation, and supporting basic keybindings for pausing an animation, taking a screenshot, or quitting an application. Instead, a programmer typically interacts with a `DataVis` class, or more precisely, a class derived from the `DataVis` base class to implement a new visualization or animation. The primary role of this class is to provide a member variable which stores an *image buffer*: a flattened 2D array of red, green, blue tuples that the user will modify through an `update` method of the `DataVis` class. By writing parallel code in `update` and connecting a `DataVis` object to a `QTViewer`, programmers can quickly create parallel 2D visualizations and animations using ParaVis. To illustrate that the library is easy to use, Figure 1 shows the complete main function using the primary classes needed for a sample application. In addition to this short boilerplate, a programmer only needs to write a small user defined class, e.g., `DemoOMPVis`, that writes values to an image buffer. No additional Qt5, OpenGL, or other visualization steps are needed. After providing some more details on how the `DataVis` class works in ParaVis below, we describe the custom components a programmer must write for a particular parallel application.

The `DataVis` base class is extremely lightweight, specifying only an interface that says there should be a pointer to a flattened 2D array of pixels called the *image buffer* and an `update` method to update those pixels. For pthreads and OpenMP applications, the image buffer resides in CPU memory, while for CUDA applications, it is more efficient to store the buffer directly on the GPU. We therefore provide two derived classes; `DataVisCPU`, and `DataVisCUDA`, that allocate space for the image buffer on the CPU and GPU, respectively. Both classes can be initialized by specifying the initial width and height of the data to visualize, or by specifying the name of an image file, e.g, a .png, .jpeg, or .gif file, to pre load into the image buffer.

```
#include <qtViewer.h>
#include <pthreadVis.h>
int main(int argc, char **argv) {
  QTViewer viewer(argc, argv,
       600, 500, "OpenMP Demo");

  int width = 200;
  int height = 200;
  DataVis* vis =
    new DemoOMPVis(2, width,height);

  viewer.setAnimation(vis);
  return viewer.run();
}
```

Figure 1. The full `main` function for a sample OpenMP visualization with ParaVis components shown in bold. A programmer writes specific OpenMP details in `DemoOMPVis::update` to update the pre-allocated image buffer, but no Qt5 or OpenGL knowledge is needed to use this library.

Since the visualization is application specific, the `update` method's implementation is the programmer's responsibility. Once the `update` method is implemented, the programmer connects the visualization to a `QTViewer` using the `setAnimation` method call (see, e.g. Figure 1), and then calls `run`. By default, `run` will repeatedly call the provided `update` method until the user exits the application. Animations can run for a fixed number of steps by providing a number of steps to `run`, with a value of 1 performing only a single frame visualization with no animation. We provide source code, further detailed documentation of the library, and full examples online [10].

A. Writing a new visualization

To visualize a full parallel application using ParaVis in C++, the user first chooses a parallel framework and a writes a new class that derives from `DataVisCPU` or `DataVisCUDA`. Several applications are described in Section IV, and code snippets for all supported frameworks are available online [10]. For simplicity, we describe how a user could write a new OpenMP application. The programmer would first create a new class, e.g., `DemoOMPVis` that inherits from `DataVisCPU`. Since the `DataVisCPU` automatically creates the image buffer on the CPU, the programmer only needs to focus on writing an application-specific `update` method in the `DemoOMPVis` class. An example is sketched in Figure 2. In the OpenMP context, the programmer would likely add pragmas to parallelize a

```
void DemoOMPVis::update() {
   int c, off;
   unsigned char val;
   #pragma omp parallel for \
     private(c, off, val)
   for(int r=0; r<h; r++){
     for(c=0; c<w; c++){
        /* compute offset,
           color val */
        buffer[off].b = val;
     }
   }
}
```

Figure 2. OpenMP code writing to the blue channel of the image buffer by parallelizing over rows

loop that writes a color to each pixel in the image buffer. If the user wants to animate the image over a sequence of steps, this is easy to support by adding a variable to the `DemoOMPVis` class that stores some counter that is incremented once per update call and used to vary the image.

Since the `update` method does not specify how to update the image buffer, any sequential or parallel approach can be used. Currently, we have sample applications using sequential programming, OpenMP, pthreads, and CUDA. It should be fairly straightforward to add other SMP parallel models, e.g, OpenACC and OpenCL. Since Qt5, OpenGL, and the underlying CMake build system are all cross-platform, we have compiled and run ParaVis on Linux, Mac OSX, and Windows operating systems.

B. The ParaVis C interface

In addition to being flexible to parallel programming model choice for C++ programmers, the simplicity of the `DataVis` interface and the underlying image buffer data structure allows us to easily implement C wrappers around the C++ interface to export C interfaces to ParaVis. While we have not yet written C library interfaces for all parallel frameworks, we describe ParaVis's current C interface for pthreads below, and anticipate similar C interfaces for OpenMP and CUDA.

The pthreads C interface to ParaVis is implemented as a thin layer on top of our C++ implementation. C programmers interact with ParaVis by passing a `visi_handle` to library functions. Internally, our library uses the `visi_handle` to refer to a C struct encapsulating the underlying C++ `DataVis` object associated with a visualization.

A C pthreads programmer who wants to use ParaVis, first adds a call to the `init_pthread_animation` function that returns a `visi_handle` that is passed to subsequent library calls. As an example, a partial pthreads application showing ParaVis calls is shown in Figures 3 through 7.

```
/* state to pass to each spawned thread */
struct appl_data {
  /* add image buffer and handle fields: */
 visi_handle handle;
 color3 *image_buff;
};

#define NUMTIDS 4
int main(int argc, char *argv[]) {
  int cols, rows, iters, numtids, i;
  visi_handle myhandle;
  color3 *image_buff;
  static char visi_name[] = "pthreads!";
  struct appl_data thread_info[NUMTIDS];
  pthread_t all_ptids[NUMTIDS];

  /* init common thread info */
  cols = rows = iters = 100;
  numtids = NUMTIDS;
  local_init_state(&thread_info[0],
    rows, cols, numtids, iters);
```

Figure 3. Setting up common information for each thread in a user defined `appl_data` struct.

```
  /* main thread gets handle and
     image buffer from library  */
  myhandle =
     init_pthread_animation(numtids,
                rows, cols, visi_name);
  image_buff =
     get_animation_buffer(myhandle);

  thread_info[0].handle = myhandle;
  thread_info[0].image_buff = myhandle;
```

Figure 4. Getting a C-style handle from the ParaVis library using `init_pthread_animation` and extracting the image buffer data using `get_animation_buffer`.

In Figure 3, a programmer declares some ParaVis variables needed by each thread. The `appl_data` struct is a application-specific struct with thread-specific data passed to every thread on `pthread_create`. The function `local_init_state` that initializes application-specific parts of the the `appl_data` struct. To use ParaVis, the struct should include a `visi_handle` and a pointer to the `color3 *` image buffer (the 2D array of r, g, b tuples that threads modify in an application-specific way).

The main thread calls `init_pthread_animation` to initialize the visualization and to get the `visi_handle`. It then calls `get_animation_buffer` to get a pointer to the library-allocated image buffer, both shared by all application threads (shown in Figure 4.)

Next, the main thread spawns application worker threads,

```
/* create threads and pass a
   copy of handle to each thread
   through its thread_info field */
for (i = 0; i < numtids; i++) {
  // init common fields
  thread_info[i] = thread_info[0];
  // init thread specific fields
  thread_info[i].mytid = i;
  pthread_create(&all_ptids[i], NULL,
     local_thread_main,
     (void *)(&thread_info[i]));
}
```

Figure 5. Creating individual threads in `main()` running user defined `local_thread_main` and using`thread_info`.

```
/* after spawning threads, main thread
   triggers animation on handle */
run_animation(myhandle, iters);

/* wait for exit, cleanup*/
for (i = 0; i < numtids; i++) {
  pthread_join(all_ptids[i], NULL);
}
```

Figure 6. Running a C animation with the `run_animation` library method.

making calls to `pthread_create`, passing each the `visi_handle` and image buffer (shown in Figure 5.) The function `local_thread_main` is the application-specific main function for each spawned thread, which will include code to repeatedly update its portion of the image buffer and inform ParaVis when it is ready to be displayed.

Finally, after spawning worker threads, the main thread makes a call to ParaVis function `run_animation` once before it waits for worker threads to exit, as shown in Figure 6. If the main thread is also a worker thread (i.e. it also calls `local_thread_main`), then exactly one thread needs to make a call to `run_animation` to start the animation. If the number of iterations passed to `run_animation` is a positive integer, the library stops the animation after the number of iterations. If it is passed 0, it runs until explicitly stopped by the user. The application should be designed so that each thread exists for the lifetime of the animation—a thread should not exit before the number of iterations passed to `run_animation`.

The functions `init_pthread_animation`, `get_animation_buffer`, and `run_animation` only need to be called once by the main thread. The application-specific `local_thread_main` function additionally needs to be modified to include code to update portions of the image buffer and to notify ParaVis that is is ready for a new image frame to be rendered.

ParaVis provides a `draw_ready(handle)` function

```
void *local_thread_main(void *args) {
   struct appl_data *myinfo =
       (struct appl_data *)args;
   for(i=0; i < myinfo->numiters; i ++) {
     /* ... */
     my_update(myinfo->image_buff,...);
     draw_ready(myinfo->handle);
```

Figure 7. Each thread updates its part of the shared image buffer and calls `draw_ready` to inform the library to update the figure. Only after every participating thread calls `draw_ready`, will the full image be rendered by the library.

that acts as a synchronization barrier to updates to the image. In the application code, threads update their portion of the image in parallel and each makes a call to `draw_ready(handle)` when it has completed its update (an example is shown as a call to the `my_update` function in the thread iteration loop in Figure 7.) Internally, `draw_ready` contains pthreads barrier synchronization that blocks all application threads until all have called `draw_ready`, signifying that the entire image is ready for rendering on the screen.

ParaVis's C pthreads interface is designed so that programmers can easily add visualization to an existing pthreads application; by just adding a few library function calls to their program, and an application-specific function that sets r, g, b values in the 2D image buffer, pthreads programmers can add visual animation of their applications using ParaVis.

Currently, we support C interfaces for pthreads and sequential applications. We plan to implement C interfaces for the CUDA and OpenMP support to ParaVis soon.

IV. Examples of Using VisiLib

We have used ParaVis in several of our courses, and we describe in detail some examples of our use so far. Additionally, we plan to expand its use in future course offerings.

A. CUDA

The development of ParaVis was inspired by prior work of co-authors Newhall and Danner with CUDA visualizations [22]. We describe two CUDA assignments that use ParaVis in a Computer Graphics and a Parallel and Distributed Computing course below. The primary TCPP [28] concepts covered in both example assignments are: Stream-GPU architectures; GPGPU computing; synchronization; heterogeneous systems; SPMD; and memory management.

1) Transparent Circle Rendering: We used ParaVis to introduce CUDA in our Computer Graphics course [10]. This course emphasizes core 3D modeling and ray tracing elements of computer graphics using a modern OpenGL approach and programmable *shaders* that run in parallel on the GPU. Students learn early in the semester that the GPU is programmable and highly parallel, but OpenGL handles

most of the data decomposition and students only have to write short shader programs that run on a single vertex or single pixel. Students also gain experience using the graphics window as a debugging tool for diagnosing problems with their shader code. Late in the semester we use CUDA to explore parallel data decomposition patterns more deeply.

Using ParaVis, students developed a CUDA visualization that rendered and animated a list of transparent circles in parallel. The assignment was a simplified version of similar project from Carnegie Mellon University [11]. The initial program input consists of a list of circles, each specified by their center, size, and radius. Using multiple CUDA kernels, students determine which circles overlapped which pixels in an image buffer and blended the color of all the circles overlapping a single pixel into one color. A lab requirement was to use a common alpha blending method in computer graphics in which the output color is dependent on the order in which the circles are processed. Students were asked to blend the circle colors in the order that they appear in the original list.

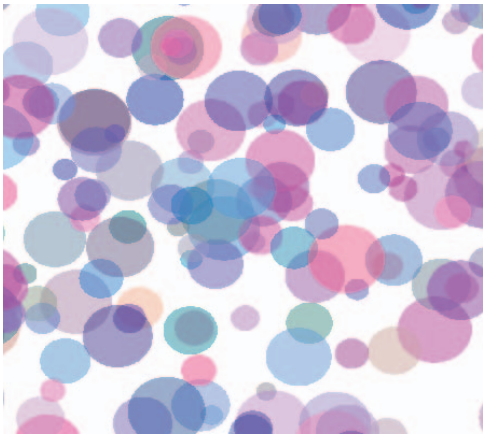

Figure 8. Visualization of transparent circle rendering of 200 random circles using CUDA application in ParaVis.

This assignment presents a challenge when attempting to implement a parallel solution. A first attempt might parallelize over circles and process each pixel that intersects a given circle without any synchronization. However, this results in the blending order being visibly wrong. Instead, parallelizing over pixels, and determining sequentially if each circle overlaps a given pixel yields correct blending results which can be easily verified with our library. Figure 8 shows one such rendering. Once the basic renderer is complete, students can write a separate CUDA kernel that moves all the circles by some amount and re-renders the scene, providing ample opportunities for fun visual effects and scalability experiments.

2) Fire Simulator: The fire simulator lab assignment teaches students CUDA programming in an upper-level undergraduate Parallel and Distributed Computing course (PDC). This is a seminar-style course were students read and discuss research papers, and propose and carry out a large independent project. This lab is assigned early in the course to teach students CUDA prior to their independent course projects. The most recent offering of PDC used a previous version of ParaVis, and this assignment is also discussed in our previous work [22].

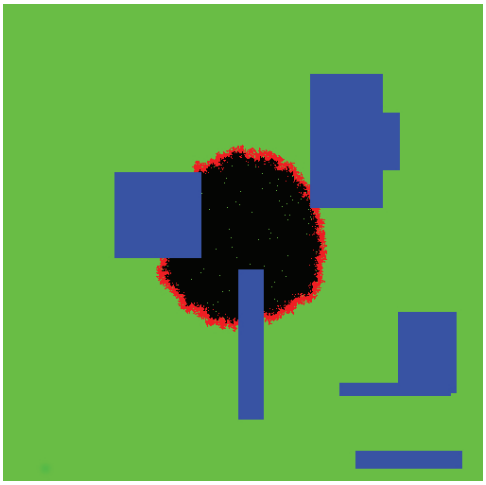

Figure 9. Visualization of CUDA Fire Simulation Program

The application is a parallelization of a discrete event simulation [27] program. It simulates a two-dimensional grid world consisting of forests and lakes. Each grid cell is classified as water, forest, fire, or burnt. Forest cells transition to fire cells based on some probability if one or more of their neighbors are on fire. Fire cells transition to burnt cells after some number of time steps. Each step of the CUDA computation computes in parallel the values for all cells for the next time step. To animate their simulation, students implement a CUDA kernel to update the ParaVis image buffer based on the current grid cell values.

Figure 9 shows a screen dump of one step of the simulation. Off the ParaVis webpage [10], we have videos of this application in action.

B. Pthreads

We recently introduced a pthreads version of ParaVis to our Introduction to Computer Systems course. This course covers a vertical slice through a computer's hardware and software systems, and it's designed to be accessible to students who have only completed a CS 1 course (or AP equivalent). It's required for all majors and minors in our department, and for most of our students, it's their first exposure to C. One of the primary goals of this course is to introduce our students to parallel computing, and we specifically focus on shared memory programming with pthreads. It includes many topics from the TCPP [28] curriculum, with a focus on covering the core skill sets (Architecture, Programming, and Algorithms).

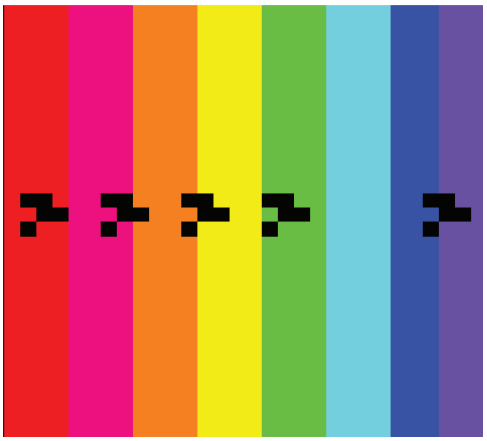

Figure 10. A screen capture of ParaVis illustrating Conway's Game of Life. The foreground shows five gliders (black), and each of the eight threads draws a unique background color according to the region of the board it has been assigned.

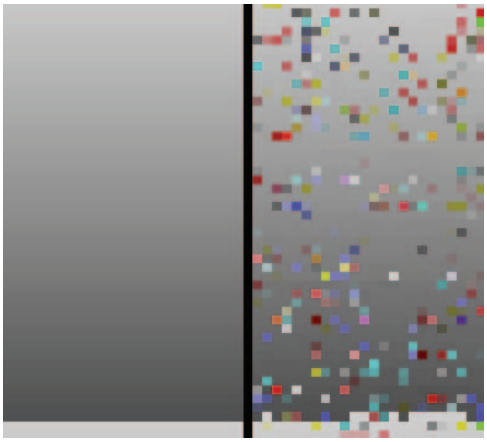

Figure 11. A screen capture of ParaVis visualizing an OpenMP demo of a vertical scrolling gradient. The left half of the image includes necessary `private(...)` variable tags in the parallel for pragma. Without the tags, errors appear visually as noise as shown in the right half.

We assign two forms of Conway's Game of Life to our students during the semester:

1) In the middle of the semester, as a *serial* program. In this version, the student's are primarily concerned with memory management (i.e., allocating and freeing two-dimensional game boards) and the basic rules of the simulation.
2) At the end of the semester, as a *parallel* program. In this version, students are required to extend their serial form of the game by implementing all the necessary thread creation, synchronization, and game board partitioning mechanisms. More details of the assignment are available online [10].

Both versions utilize ParaVis to visualize the simulation, and it's particularly helpful in illustrating how to partition the game board among the threads. Figure 10 shows a screen capture of an eight-thread, column-wise partitioning of the board in which each thread draws a unique background color. We have found that representing the board in this format helps students to comprehend thread boundaries and the regions in which threads require information sharing. Additionally, this depiction facilitates debugging for thread partitioning errors, making it quickly apparent if threads are operating over unintended regions of the board.

C. OpenMP

While we have support for OpenMP, we do not yet have any classroom tested applications that use OpenMP. Anecdotally however, we found ParaVis helpful in debugging a simple demo application using OpenMP. Our demo started from a simple sequential application that updated values in a 2D grid using a basic double for loop. A naïve first attempt might use OpenMP to add a simple `#pragma omp parallel for` ahead of the outer for loop, expecting OpenMP to assign multiple threads to different rows and accelerate our code. This does have the expected effect of accelerating the computation, and a student may think that this speedup indicates that the OpenMP solution works correctly. However, when connected to ParaVis and displaying the results, we find noise in the visualization when we attempt to color a simple scrolling vertical gradient as seen in the right half of Figure 11. With the visual feedback of ParaVis, a programmer can discover that OpenMP only makes private copies of the outer loop variable for each thread. Since the demo code had additional variables inside the body of the loop that needed to not be shared, adding an additional `private(...)` clause to the outer pragma makes the code work correctly as seen on the left half of Figure 11 while still providing speedup.

The examples described in the CUDA and pthreads section could also be adapted to work for OpenMP, but we found that CUDA and pthreads were the preferred choice for the course and application in their respective context. Our library allows the flexibility to choose the parallel programming model that works best for you.

D. Other Applications

Thus far, we found our library to be helpful in three of our courses and we plan to expand its use in future course offerings. We believe the library would have broad appeal in the EduPar community. A quick scan of recent Peachy assignments indicates there are number of examples that combine parallelism and visualization for e.g., fractals [7], image processing [14], and simulations [4]. By separating out the details of rendering and animating images and providing a easy-to-use interface, our library can lower the barrier to developing new parallel visualizations or debugging existing parallel implementations.

V. Evaluation

To evaluate ParaVis, we characterize the experiences of students who have taken the three courses in which we adopted it. In all three courses, students found ParaVis to aid their debugging efforts while making the content more accessible and enjoyable.

We first used ParaVis in the pthreads Game of Life assignment in Fall semester of 2018. The 52 students in the class found ParaVis to be especially helpful for debugging. Most commonly, it has assisted students in tracking down two forms of partitioning problems. In the first problem, students fail to fully divide the board amongst the threads, leaving regions of the board completely unaccounted for. Such regions will clearly appear as uninitialized noise in the visualized output. The other error stems from accidentally assigning multiple threads to operate over the same region (e.g., their start/end rows numbers are off by one). In that case, a region's color may change sporadically during execution as threads compete to update it.

Similarly, students found using ParaVis to be helpful in debugging the CUDA fire simulator lab. The fire simulator lab has been completed by 63 students over three different offerings of the course. The visualization helped students to easily discover errors with CUDA grid, block and thread mappings to application data that missed or overlapped portions of the 2D world simulation. Errors with CUDA thread mapping to data elements are common, particularly when first learning to program in CUDA. The visualization helped students to discover that they had these types of errors, and it often additionally helped them identify the parts of their program that were likely causes of the errors. For example, when students visually saw missing portions of the world not getting updated or saw the fire not correctly spreading into some portions of the world, this immediately pointed them to errors in their CUDA threads mapping logic. Other bugs, such as missing `cudaMemCopy` of initial state, or not correctly implementing the fire spreading function, were also easily identified by the visualization not looking as expected.

In addition to helping with debugging, students really enjoyed this lab, we believe in large part because of the visualization. More than 80% of students implemented extensions to the required parts of this lab assignment, and many implemented impressive extensions that simulate more realistic burning functions, create interesting starting worlds, and visualize the fire simulation in more realistic ways. These "just for fun" extensions resulted in students gaining more expertise and comfort with CUDA programming. After adding this lab assignment to the course, many students use CUDA in their independent course projects. We believe that their learning CUDA in this context helps them to gain enough expertise to view it as another resource for implementing their independent course projects. In the initial offering of this course, we did not included the CUDA fire simulation lab assignment, and only 10% of final course projects used CUDA. We assigned this lab in the three subsequent offerings of the course, and the percentage of final projects that used CUDA increased significantly—in the three semesters we assigned the CUDA fire simulator lab, 50%, 50% and 40% of final projects used CUDA.

In the computer graphics course visual debugging is used throughout the course and the use of ParaVis was essential to verifying the correctness of the students' CUDA lab on rendering transparent circles. Students could quickly verify if the blending order was incorrect on small static test samples, and could also verify proper data decomposition of the 2D array when working with larger sets of moving circles. In a class of 13 students, one listed "Visual debugging" as one of the best things about the course, while a second student listed learning about "CUDA and parallel programming" as the best thing about the course. Though this exercise was just a portion of the only CUDA assignment in the course, one group used it as the starting point for their self-designed final project to build a CUDA renderer for 3D metaballs / iso-surfaces.

VI. Conclusion and Future Directions

ParaVis is a visualization library that is specifically designed to aid in learning new parallel programming languages, understanding parallel computation, and helping with debugging parallel applications. Our initial experiences using it in both intermediate and upper-level courses demonstrated that it is a helpful tool for fostering student understanding and for helping students learn parallel programming. Students also generally enjoy the visualization part of the assignments, often resulting in their spending more time working on the lab assignments and thus gaining further experience and expertise with the parallel language.

Currently, our library targets visualizing shared memory and GPU parallelism on a single machine, with C and C++ interfaces for pthreads, OpenMP, and CUDA (and sequential) applications. It should be fairly easy to extend support to similar SMP parallel models such as OpenCL or OpenACC. Additionally, with community interest and support, we could incorporate more language bindings including python and fortran. Because our library targets single machine parallelism, it exports a single image buffer that is shared amongst all parallel workers. In order to add support for non-shared memory parallelism (such as MPI), a different image buffer abstraction and sharing interface would need to be added.

References

[1] The Alice Programming Environment. https://www.alice.org.

[2] ACM/IEEE-CS Joint Task Force. Computer science curricula 2013. www.acm.org/education/CS2013-final-report.pdf, 2013.

[3] Joel Adams, Richard Brown, and Elizabeth Shoop. Patterns and exemplars: Compelling strategies for teaching parallel and distributed computing to CS undergraduates. In *Parallel and Distributed Processing Symposium Workshops and PhD Forum IPDPSW, 2013 IEEE 27th International*, May 2013.

[4] Joel C. Adams. Using the Monte Carlo pattern to simulate a forest fire. In *Proc. Workshop on Parallel and Distributed Computing Education*, 2018.

[5] L. P. Baldwin and J. Kuljis. Learning programming using program visualization techniques. In *Proceedings of the 34th Annual Hawaii International Conference on System Sciences*, Jan 2001.

[6] Cordelia M. Brown, Yung-Hsiang Lu, and Samuel Midkiff. Introducing parallel programming in undergraduate curriculum. In *Parallel and Distributed Processing Symposium Workshops and PhD Forum IPDPSW, 2013 IEEE 27th International*, May 2013.

[7] Martin Burtscher. Computing a movie of zooming into a fractal. In *Proc. Workshop on Education for High-Performance Computing*, 2018.

[8] Joel C.Adams, Patrick A.Crain, Christopher P.Dilley, Christiaan D.Hazlett, Elizabeth R.Koning, Serita M.Nelesen, Javin B.Unger, and Mark B. Vande Stel. TSGL: A tool for visualizing multithreaded behavior. *Journal of Parallel and Distributed Computing*, 118, Part 1:233–246, August 2018. doi:10.1016/j.jpdc.2018.02.025.

[9] Christopher D. Carothers, Brad Topol, Richard M. Fujimoto, John T. Stasko, and Vaidy Sunderam. Visualizing parallel simulations in network computing environments: A case study. In *Proceedings of the 29th Conference on Winter Simulation*, WSC '97. IEEE Computer Society, 1997.

[10] Andrew Danner, Tia Newhall, and Kevin Webb. https://www.cs.swarthmore.edu/paravis/, 2019.

[11] Kayvon Fatahalian. Assignment 2: A Simple CUDA Renderer. http://15418.courses.cs.cmu.edu/spring2017/article/4, 2017.

[12] J. A. Fulcher. Visualization as a key element in learning. In *Proceedings of IEEE Region 10 International Conference on Electrical and Electronic Technology. TENCON 2001 (Cat. No.01CH37239)*, volume 1, Aug 2001.

[13] Max Grossman, Maha Aziz, Heng Chi, Anant Tibrewal, Shams Imam, and Vivek Sarkar. Pedagogy and tools for teaching parallel computing at the sophomore undergraduate level. *J. Parallel Distrib. Comput.*, 105(C), July 2017.

[14] Julian Gutierrez, David Kaeli, and Fritz Previlon. Optimization of an image processing algorithm: Histogram equalization. In *Proc. Workshop on Education for High-Performance Computing*, 2018.

[15] M. T. Heath and J. A. Etheridge. Visualizing the performance of parallel programs. *IEEE Software*, 8(5), 1991.

[16] David J. John and Stan J. Thomas. Parallel and distributed computing across the computer science curriculum. In *Parallel and Distributed Processing Symposium Workshops IPDPSW, 2014 IEEE International*, May 2014.

[17] M. A. Kuhail, S. Cook, J. W. Neustrom, and P. Rao. Teaching parallel programming with active learning. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2018.

[18] Allen D. Malony, Daniel A. Reed, and David C. Rudolph. Integrating performance data collection, analysis, and visualization. In Rebecca Koskela and Margaret Simmons, editors, *Parallel Computer Systems*. ACM, 1990.

[19] A. Marowka. Think parallel: Teaching parallel programming today. *IEEE Distributed Systems Online*, 9(8), Aug 2008.

[20] Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. The paradyn parallel performance measurement tool. *Computer*, 28(11), November 1995.

[21] Thomas L. Naps and Eric E. Chan. Using visualization to teach parallel algorithms. In *The Proceedings of the Thirtieth SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '99, 1999.

[22] T. Newhall and A. Danner. Fire simulator and fractals: using a visualization library to introduce CUDA. In *Proc. Workshop on Parallel and Distributed Computing Education*, 2018.

[23] Tia Newhall, Andrew Danner, and Kevin C. Webb. Pervasive parallel and distributed computing in a liberal arts college curriculum. *Journal of Parallel and Distributed Computing*, 105, 2017.

[24] Santiago Ontañón, Jichen Zhu, Brian K. Smith, Bruce Char, Evan Freed, Anushay Furqan, Michael Howard, Anna Nguyen, Justin Patterson, and Josep Valls-Vargas. Designing visual metaphors for an educational game for parallel programming. In *Proceedings of the 2017 CHI Conference Extended Abstracts on Human Factors in Computing Systems*, CHI EA '17. ACM, 2017.

[25] Donald P. Pazel and Beth R. Tibbitts. Intentional MPI programming in a visual development environment. In *Proceedings of the 2006 ACM Symposium on Software Visualization*, SoftVis '06. ACM, 2006.

[26] Mitchel Resnick, John Maloney, André Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. Scratch: Programming for All. *Communications of the ACM*, 52(11):60–67, November 2009.

[27] Angela B. Shiflet. Spreading of Fire. http://nifty.stanford.edu/2007/shiflet-fire/, 2007.

[28] The NSF/IEEE-TCPP Curriculum Working Group. NSF/IEEE-TCPP curriculum initiative on parallel and distributed computing - core topics for undergraduates. http://www.cs.gsu.edu/~tcpp/curriculum/, 2012.