LEGION VIS

https://github.com/DocSohl/LegionVis

Process book

Ian Sohl – u0445696

Phil Cutler – u0764757

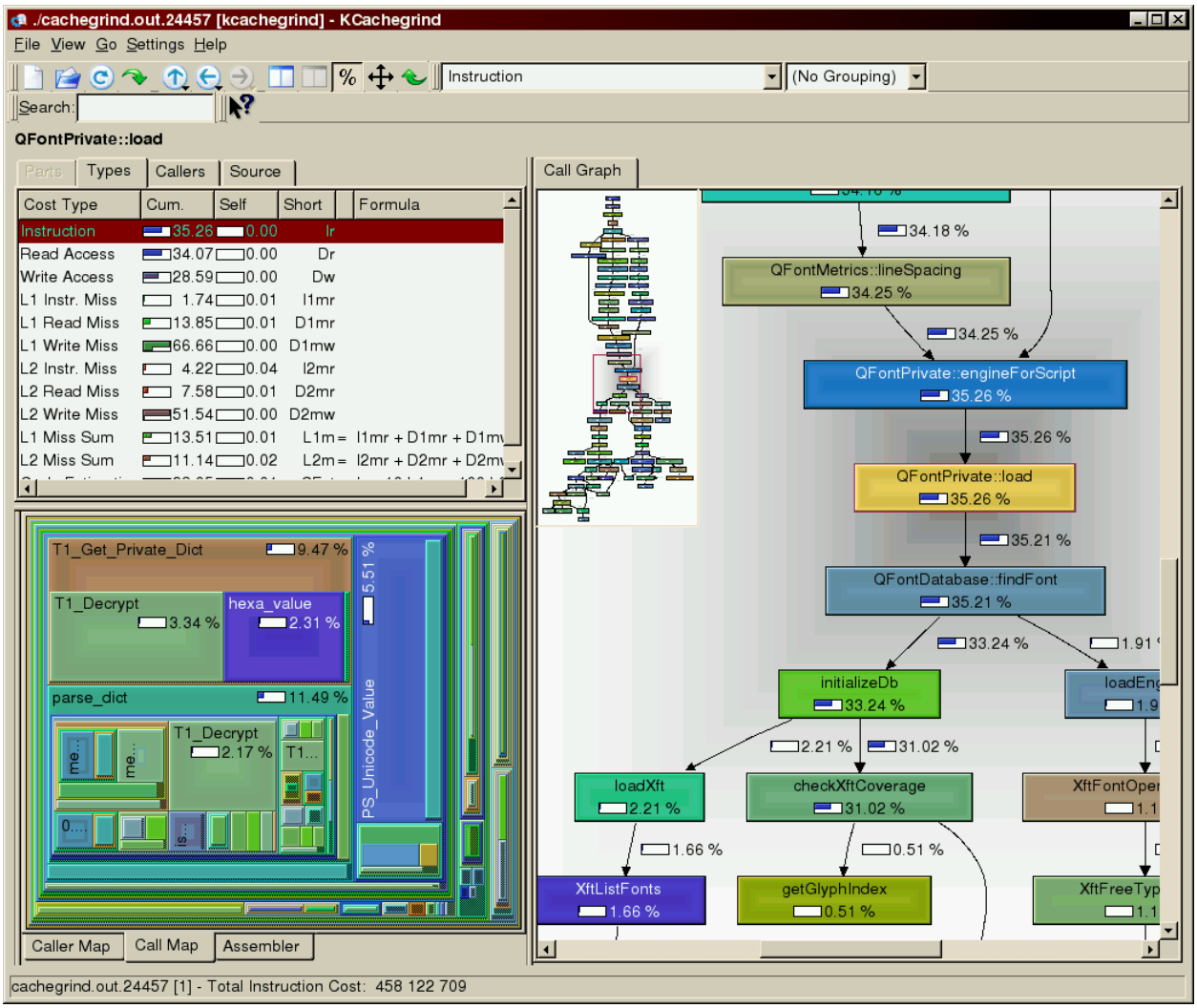
Ariel Herbert-Voss – u0591949

Overview and Motivation

The primary goal of this visualization tool is to allow the user to draw conclusions about the performance of code written in the Legion programming system, which is a data-centric parallel programming system for writing portable high performance programs targeted at distributed heterogeneous architectures. Legion automatically schedules tasks written for high performance applications, meaning that the user doesn't know which processor is running a task or when it starts. Therefore, having a visualization tool that can inform the user of these parameters is very useful for optimizing performance and resources.

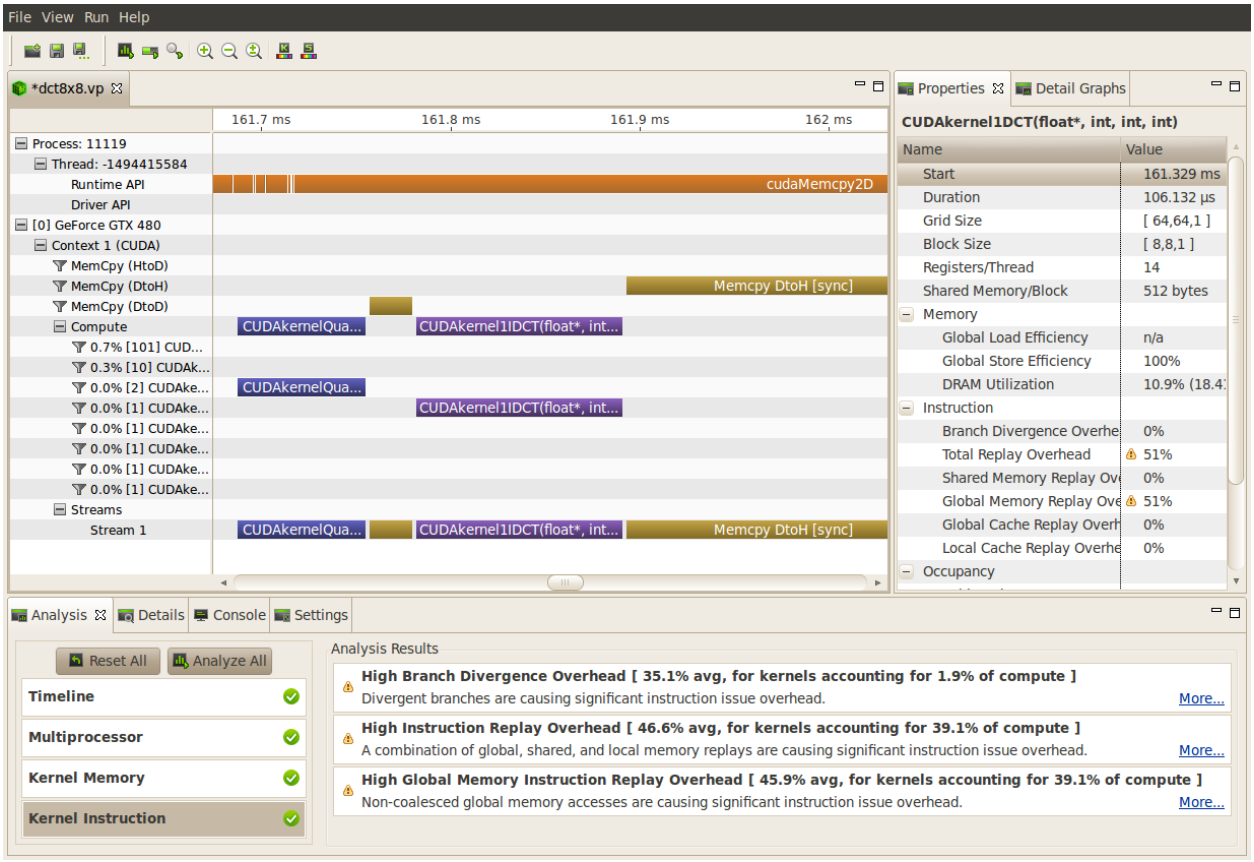
Related Work

Profiler visualization tools are not a new concept and have existed for at least as long as profilers themselves have existed. Perhaps the most well-known example is KCachegrind, which is the visualization tool for the Valgrind programming tool for memory debugging, leak detection, and profiling. Below is the call graph and call map views for KCachegrind.



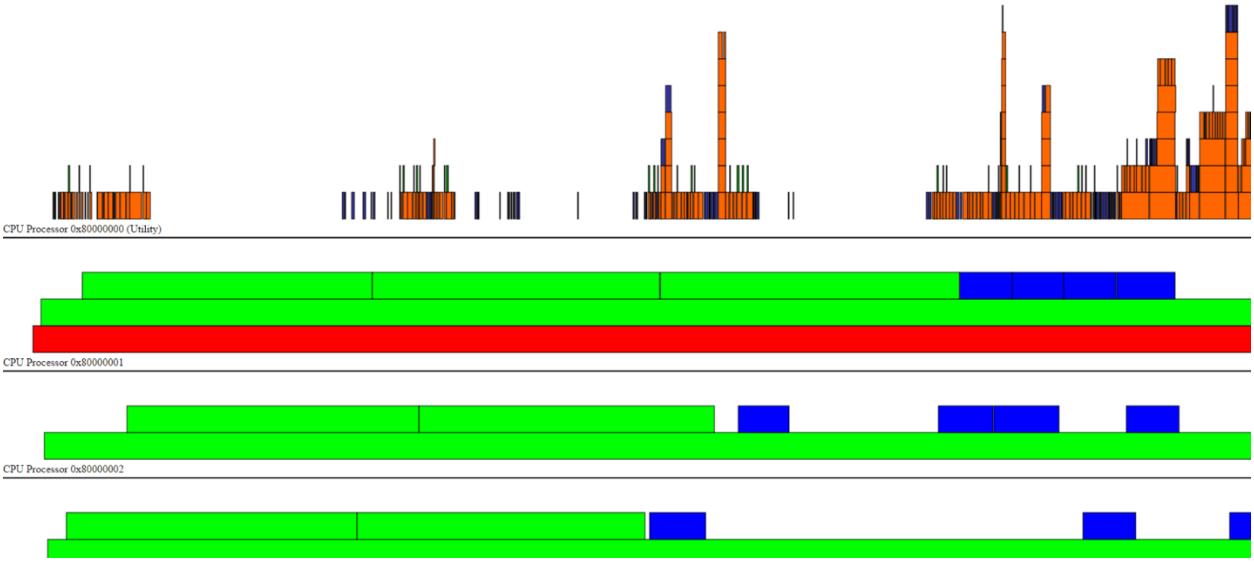
Although not particularly aesthetically-pleasing, KCachegrind is effective in showing the call structure of a particular program and interfaces well with other Linux-based code profilers. However, for highly-parallel code, KCachegrind is inherently insufficient because it relies on the profiled program executing only one thread at a time. Additionally, the tree map graph is ineffective for visualizing call structure in Legion because the user can send data across branches or up the tree due to the unique construction of the system.

With regards to parallel computing, NVIDIA also has a profiler visualization tool for their CUDA programming language. Below is a view of the execution timeline for two host processes in a CUDA kernel.



However, we see that the view is split along function category rather than which processor is running a given function. This sort of tool is not useful for visualizing program execution in Legion because we are unable to identify bottle necks and scaling effects without a proper view of how the processors allocate functions.

Currently, the Legion system has a built in profile visualization tool that produces static images. Below is one such image that shows performance for the first three cores of a 32-core system.



Here, the individual tasks are differentiated by color, rather than a positional axis, as they are less important. This visualization selects for processor parallelism, and task execution time, since tasks in Legion run in an arbitrary ordering and distribution based on the internal scheduler. However, this visualization performs poorly as the number of processors or time increases, which can easily happen in the high performance environments that Legion is intended for.

User Goals

Our target user base for our visualization is Legion programmers who want to be able to identify problems with their code using a profiler. We aim to meet the following user goals:

1- Bottleneck Identification

The main question that our users want to answer with this visualization is whether they have bottlenecks or other major performance issues with their code. A bottleneck can occur when many tasks are waiting upon the completion of a few tasks, thus resulting in irregular load balancing. This load balancing is not only inefficient for the time and computational resources available to the user, but can also have adverse physical effects on very high performance clusters due to the fluctuations in power consumption.

Given that bottlenecks occur when small numbers of tasks dominate execution, we can identify them relatively easily with the right data abstraction.

2- Understanding Control Flow

Due to its properties as a distributed parallel computing language, Legion has a fundamentally different control flow than other HPC languages, making it tricky for users to understand and communicate how information moves in Legion programs.

Being able to see which processes spawn smaller processes is an important feature in traditional profiling tools, so this is also an interaction our users will expect.

3- Memory Performance

Being able to see how a program performs with respect to memory is an important task in any programming language, especially for HPC systems. Legion is written in C++ and therefore does not have garbage collection, which unfortunately allows memory leaks and other rogue memory behaviors, and the parallel nature of the language makes them difficult to detect.

Similar to bottleneck identification, we can identify memory leaks and other unusual program memory behavior with the right data abstractions.

Useful Goals Beyond Our Scope

Although not currently a feature in our visualization, another goal we identified that would be important to our users is the ability to identify how well their programs scale. Since much of computer science and algorithm design uses asymptotic complexity as a metric of performance, our users will want to scale their executions in both data size and processor allocations. To tackle this, future iterations of this project should include a visual summary of the total performance for a given run of a program and compare it with runs on different machines and data.

Data

Data for this visualization will entirely originate from the end user. We want Legion programmers to be able to upload their own profiling files and visualize them using our system, making the visualization very personal and informative.

These profiling files consist of large plaintext files that detail various operations and actions during the code execution. The primary events that we are interested in are the execution start and stop times of the tasks, as well as which tasks were spawned by other tasks. Data processing happens on both the client and server sides. The server will perform the parsing and most computationally expensive components of data processing, such as scrubbing for task start and stop times. This information is then sent to the client side to perform the visualization-centric calculations, such as ordering on the screen, and computation of histograms.

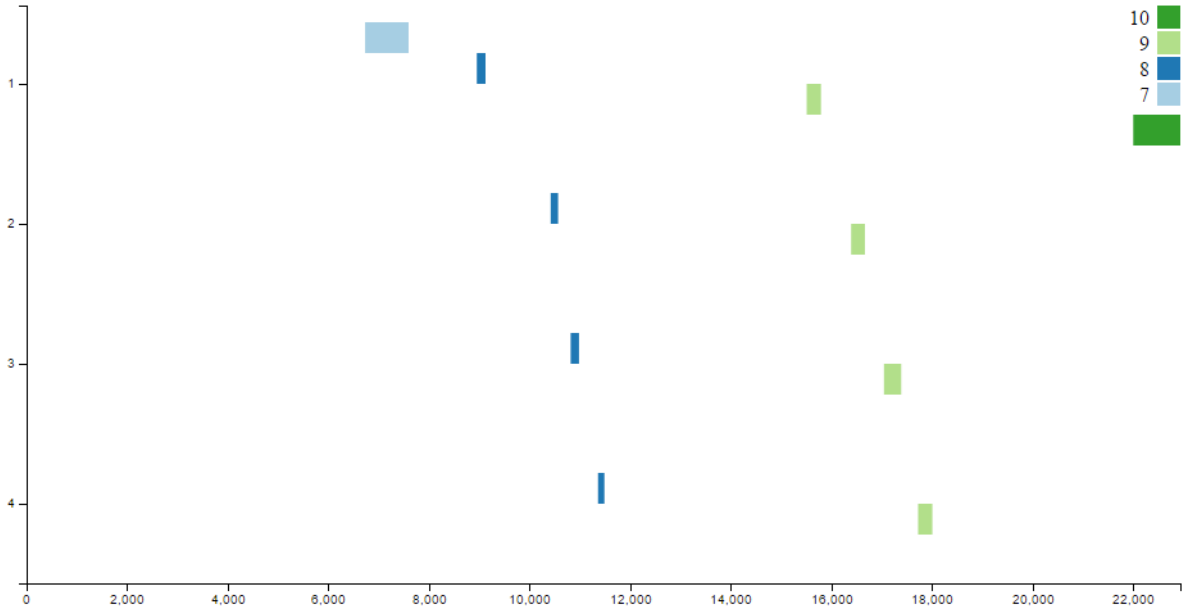
The server will maintain a copy of user submitted files, and allow the user to access them via a unique URL. This allows users to upload the data in one location, and view it from other devices, such as mobile devices, or a coworker's computer for collaboration.

Exploratory Data Analysis

For the initial analysis and selection of test data, we used the built-in visualization offered by Legion. This allowed us to select datasets that varied from simple to complex, and compare our visualizations to an established metric. The other advantage of using the packaged Legion system as a baseline is it's handling of less common cases, such as heterogeneous systems.

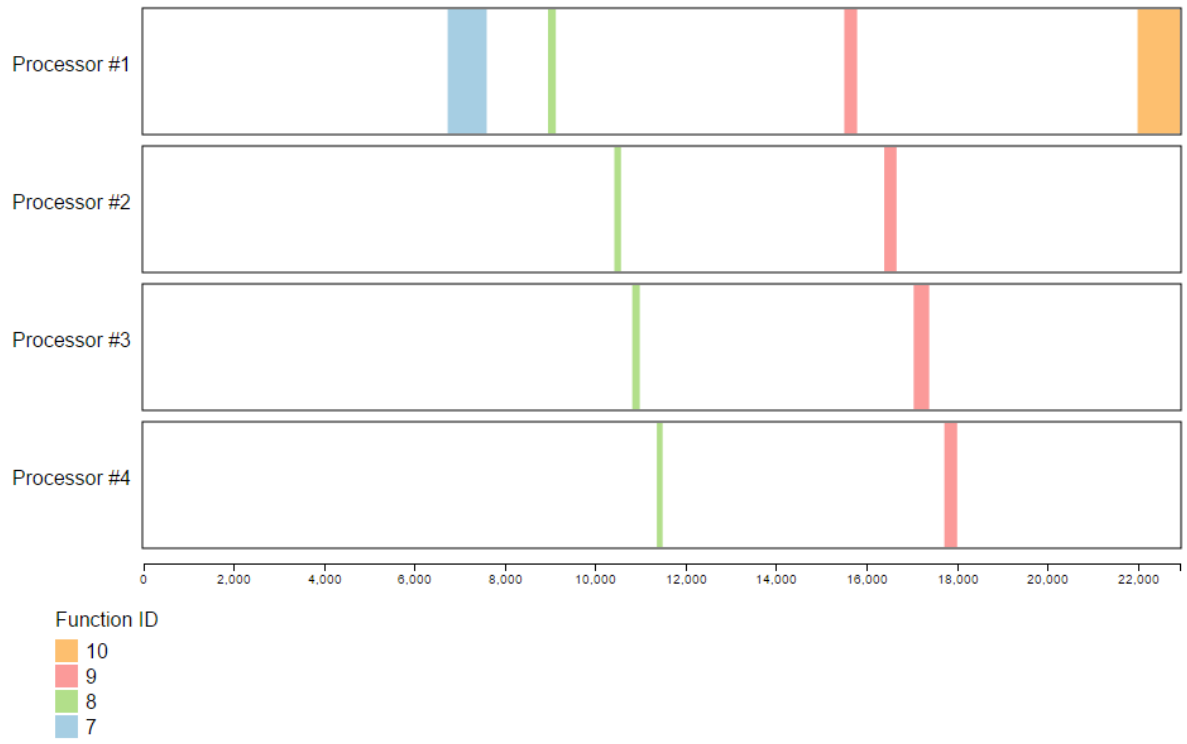
Design Evolution and Implementation

Our first iteration drew very heavily from the Legion default profiler as inspiration.



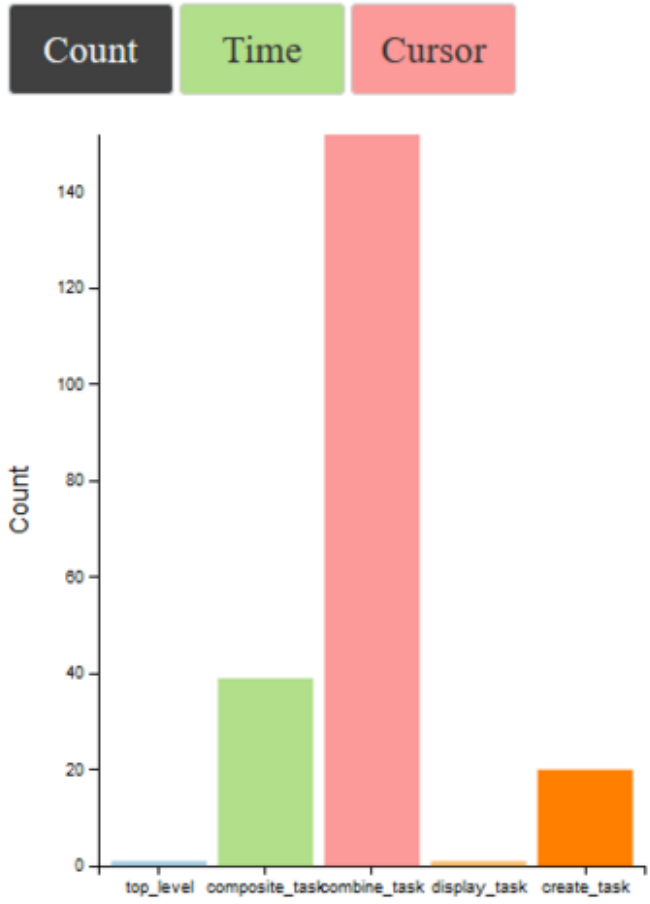
However, we decided that the spacing between components and color system made it difficult to identify features and relationships in the execution. Additionally, we opted for a more compartmentalized view of different processors rather using the numbers on the y-axis, as that erroneously implied some measure of meaning to the ordering of processors.

Our second iteration removed the axis bars and did away with a lot of unnecessary whitespace to better show patterns of processor use. This is seen below.

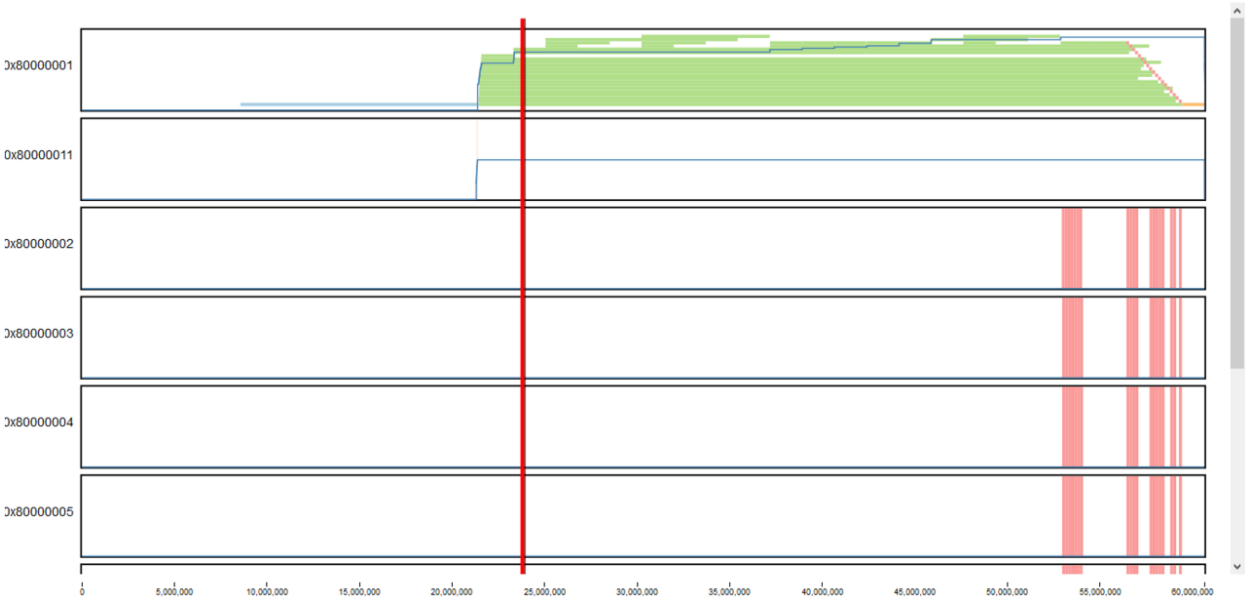


This design allowed the user to see individual processor lines much more clearly, and featured a more useful and intuitive color scale. It also utilized interactivity by providing tooltips upon mouseover of each task. This tooltip provided the ID of the function, and the visualization also featured zooming and panning along the x-axis using the mousewheel and by dragging with the mouse button, respectively.

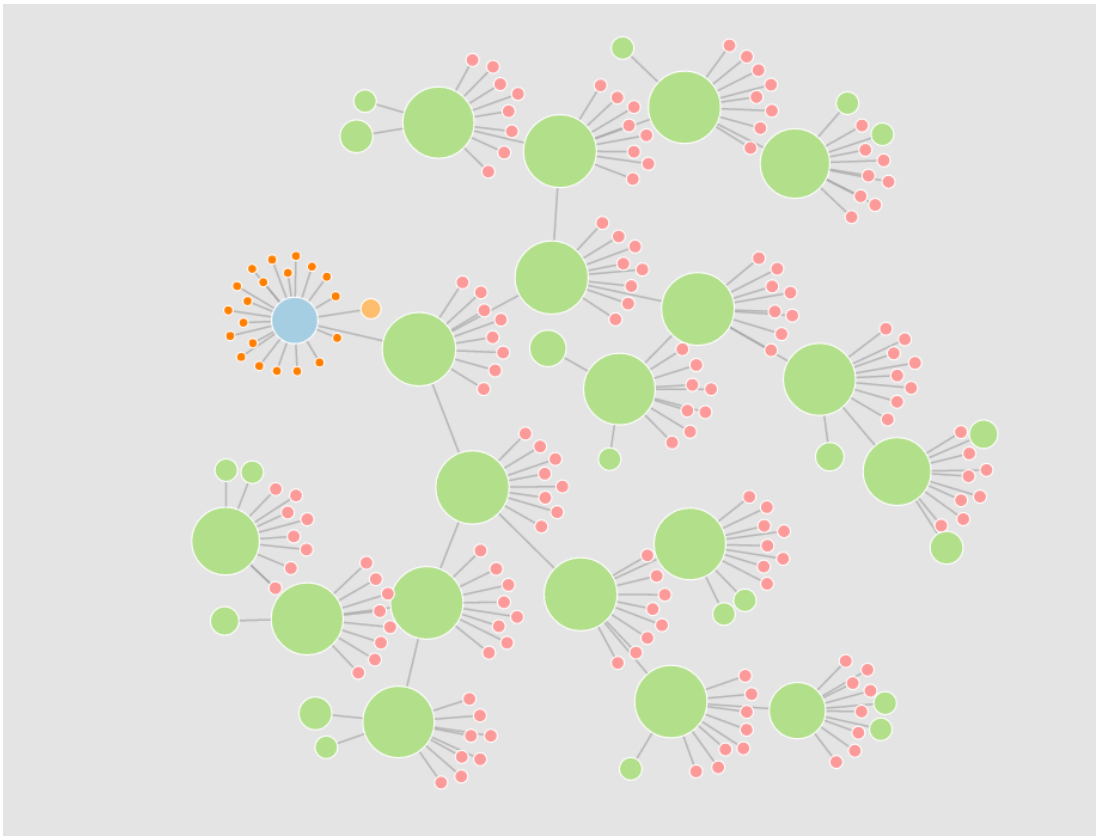
While this view provided more information about how tasks are distributed across each processor, we discovered that it didn't scale very well and didn't provide enough detail about the temporal distribution of task categories. We implemented a histogram view that tells the user the total amount of time each program task takes during the duration of the run, and also toggles to tell the user the total count for each program task across the entire run. Below is a screen shot of the histogram displaying the total count for a run.



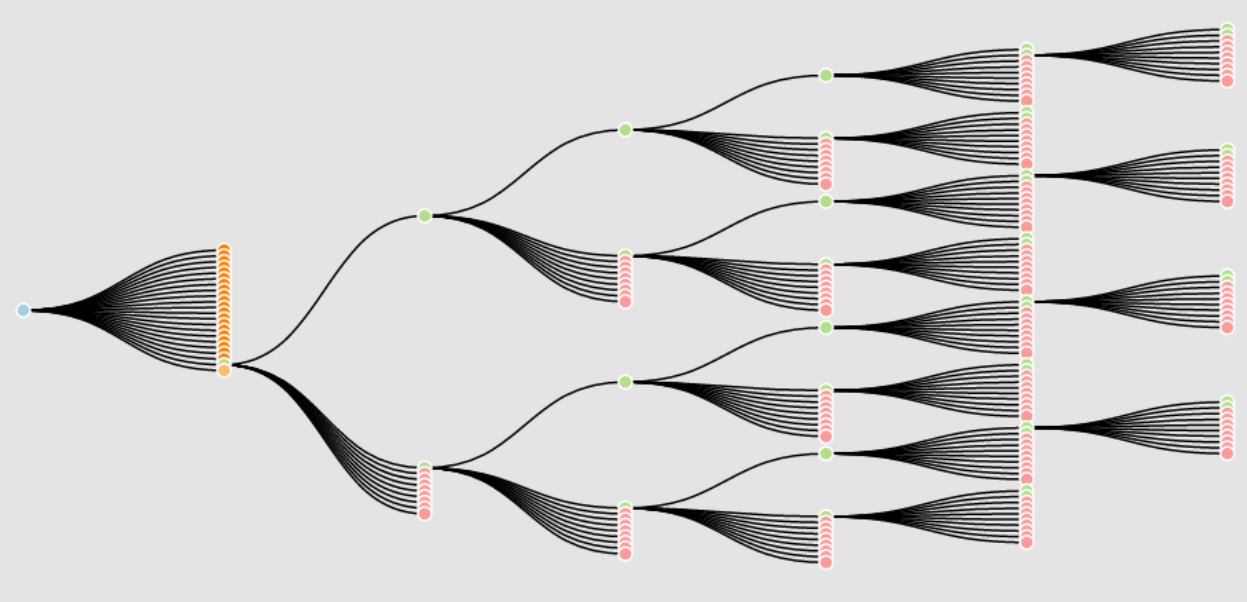
This particular dataset is for a compositing program, and we can clearly see that the combine_task occurs the most, which is to be expected. This view is helpful for identifying anomalies in task spawns. We also implemented a vertical line brush to let the user select a time slice to view a specific time. This can be selected by toggling the "Cursor" button. Below is the first version of the time slice brush.



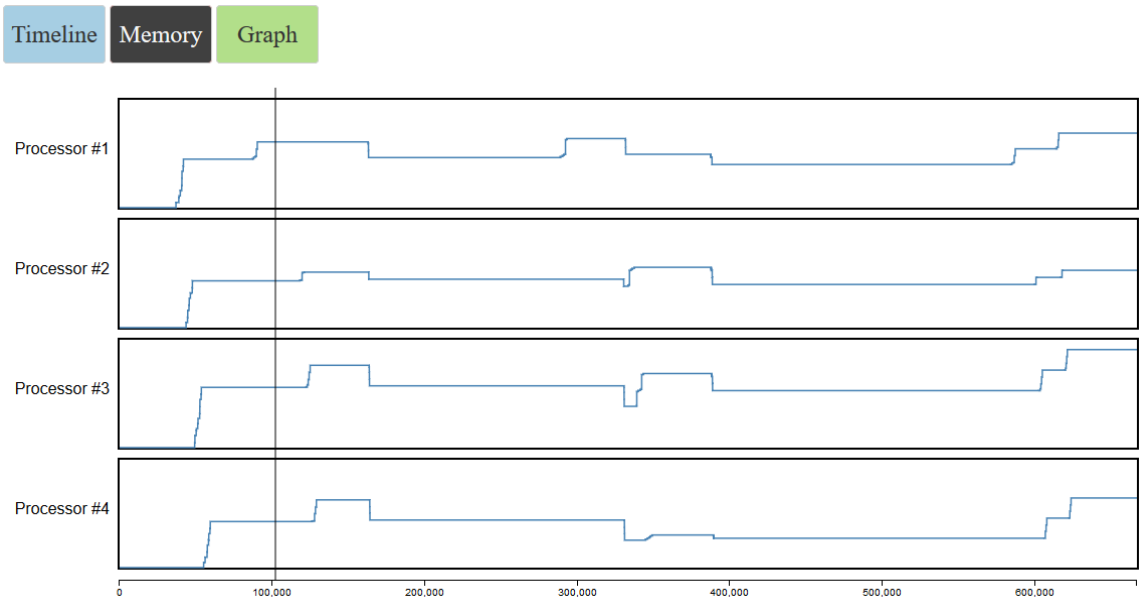
We also wanted to implement a call graph structure similar to other profilers, and started with a simple force-directed visualization, as seen below.



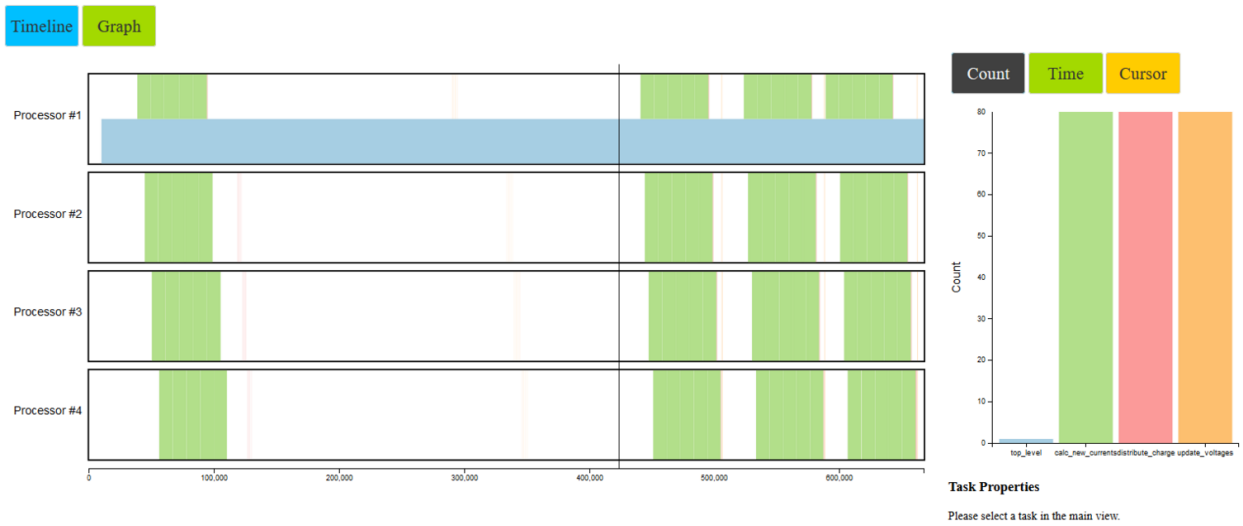
This design was quickly scrapped, as it quickly became cluttered and the bouncing circles proved to be distracting. We then tried a tree structure as shown below, which was much easier to follow.



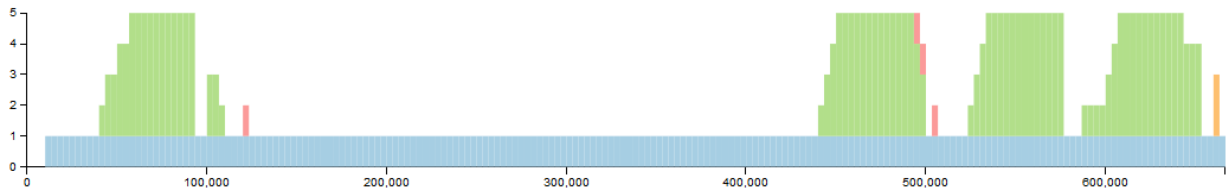
After identifying the user interaction goals, we realized that it would be useful to allow deeper exploration into memory behavior. We implemented another view to allow the user the ability to see the amount of memory allocation. We started by implementing it as a separate view, as seen below.



We see that allocation is not dramatically different across the processors and processor performance has been stacked on the bottom. We realized that memory allocation is most informative while in the context of processor tasks, so we switched the view to have stack the processor tasks and memory allocations in the individual processor windows. This view is shown below.

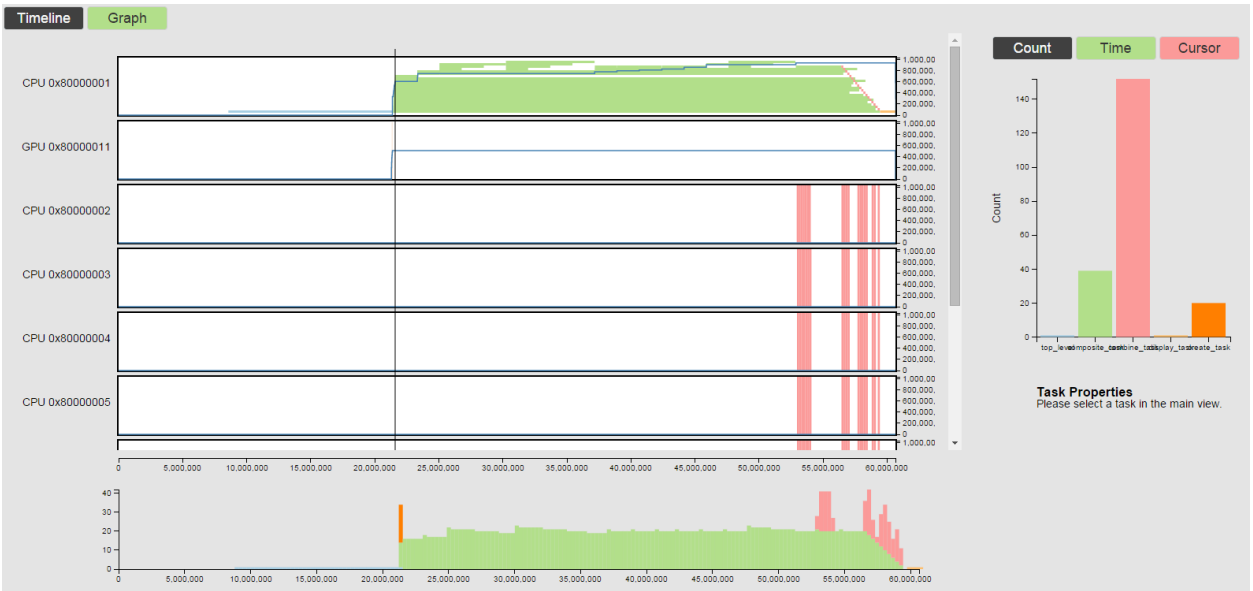


We also recognized that being able to summarize a run into a compact view would be valuable for future work on comparing performance across different runs, so we added a summary view at the bottom of the timeline view. We implemented a scrubbing feature connected to the timeline view so that users can select a time segment on either view and see changes in both. This is shown below.



Implementation

The final version of the visualization appears below with a discussion about each feature.



Timeline

The timeline shows an individual view for each processor with the tasks represented as blocks and memory allocation represented by a line graph. A movable stylus allows the user to scrub across the view and update the histogram for a specified time. Tasks that run concurrently on a single processor stack vertically to show instances of multi-threading. Zooming with the mouse scroll wheel zooms on the x-axis (time) only, allowing the user to see greater detail on certain tasks. For situations with a large number of processors, the main view is limited to seeing 6 at one time, and a scroll bar is available for viewing further processors. Clicking on a task in the timeline will show more information about that task under the task properties section.

Histogram

The histogram view has three main forms. The count and time forms both represent data from across the entire runtime. In particular, the count view shows the raw number of times that each task was executed through the life of the program. The time view displayed the net runtime of all tasks of a single kind added together. This is useful in finding long running tasks.

The third view for the histogram shows the task counts at a particular time, indicated by the user's stylus, which follows the mouse while on the main view. The histogram will then update live to represent the structure of execution at the specified time.

Call Graph

Our second main view is a graph representing the execution structure of the program. Because Legion operates in a task-based model, the connections between individual tasks can be represented in a tree format. This view can be zoomed and panned using the mouse, and the user can interact with individual nodes by hovering or clicking to see properties.

Evaluation

Our visualization has been well-received by the developers of Legion. One developer was able to use the combined task timeline and memory graph to identify a memory leak within their program. Users have universally stated that our visualizer represents the data more accurately and intuitively than other systems they have used. When asked about the user interaction goals outlined in an earlier section, the developers felt that our tool met all of these goals to a good standard. The only complaints we received were about minor aesthetic details.

This project was technically difficult due to its freeform nature as a tool. Because our major system goal is to allow any Legion user to upload their own profiling results and use the visualizer to identify patterns, our system has to be capable of adapting to a wide variety of data conditions and types. A major challenge was in addressing scalability issues, as users could possibly approach the visualizer with datasets containing thousands of individual processors, or very long run times. This has to be balanced with the capability of succinctly representing the information for users with smaller amounts of data, which we believe will be typical of the majority of cases.

While we believe that we have a useful tool for our visualization system, there are still some major improvements that we plan to make. Our next major goal is to complete a summary comparison view to allow users to compare performance of different runs and explore the scalability of Legion programs for

both data size and processor number and type. We recognize that this project has good potential for publication and we would like to polish it to a level appropriate for submission.