LEGION VIS

https://github.com/DocSohl/LegionVis

Process book

Ian Sohl – u0445696

Phil Cutler – u0764757

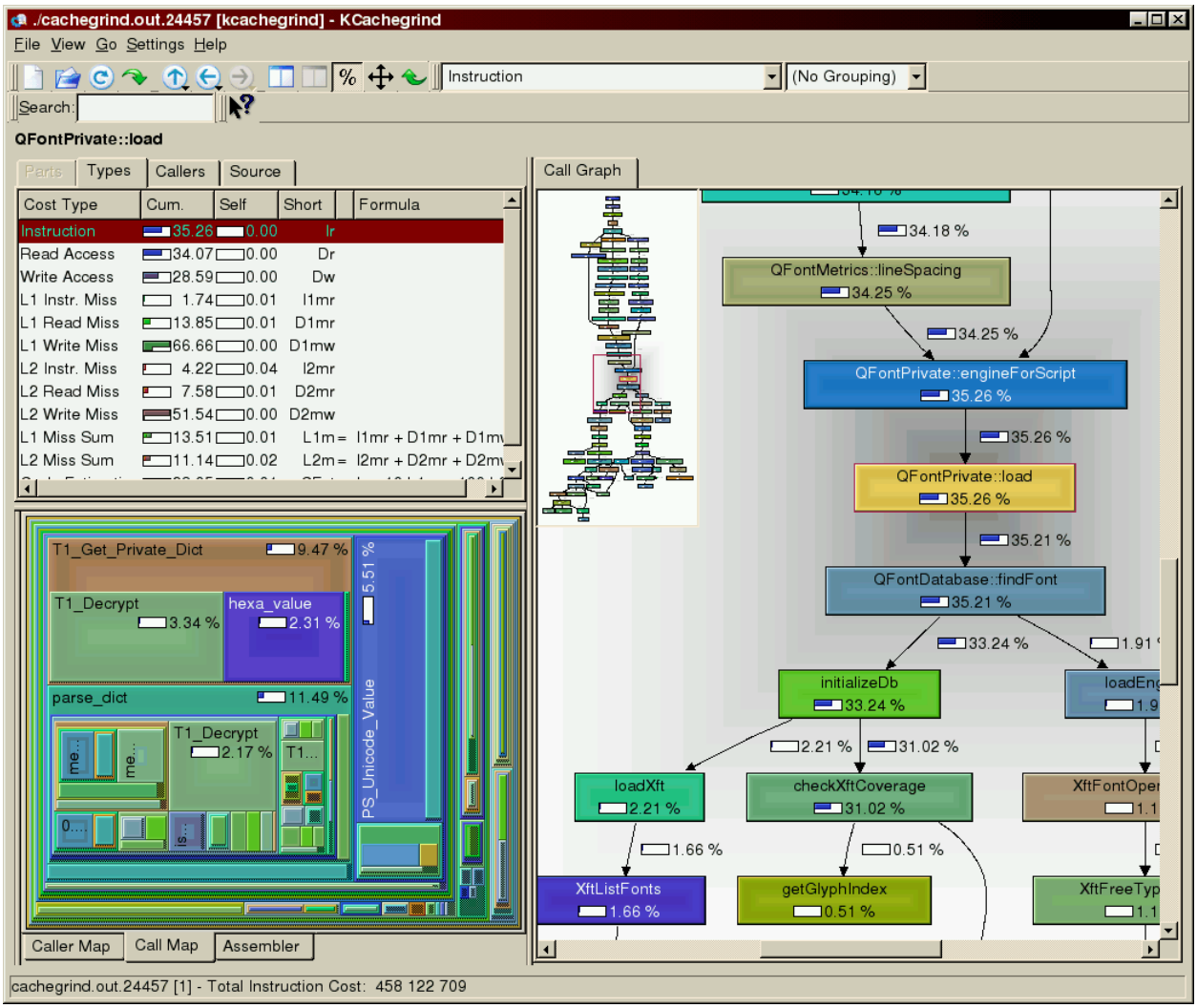
Ariel Herbert-Voss – u0591949

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.

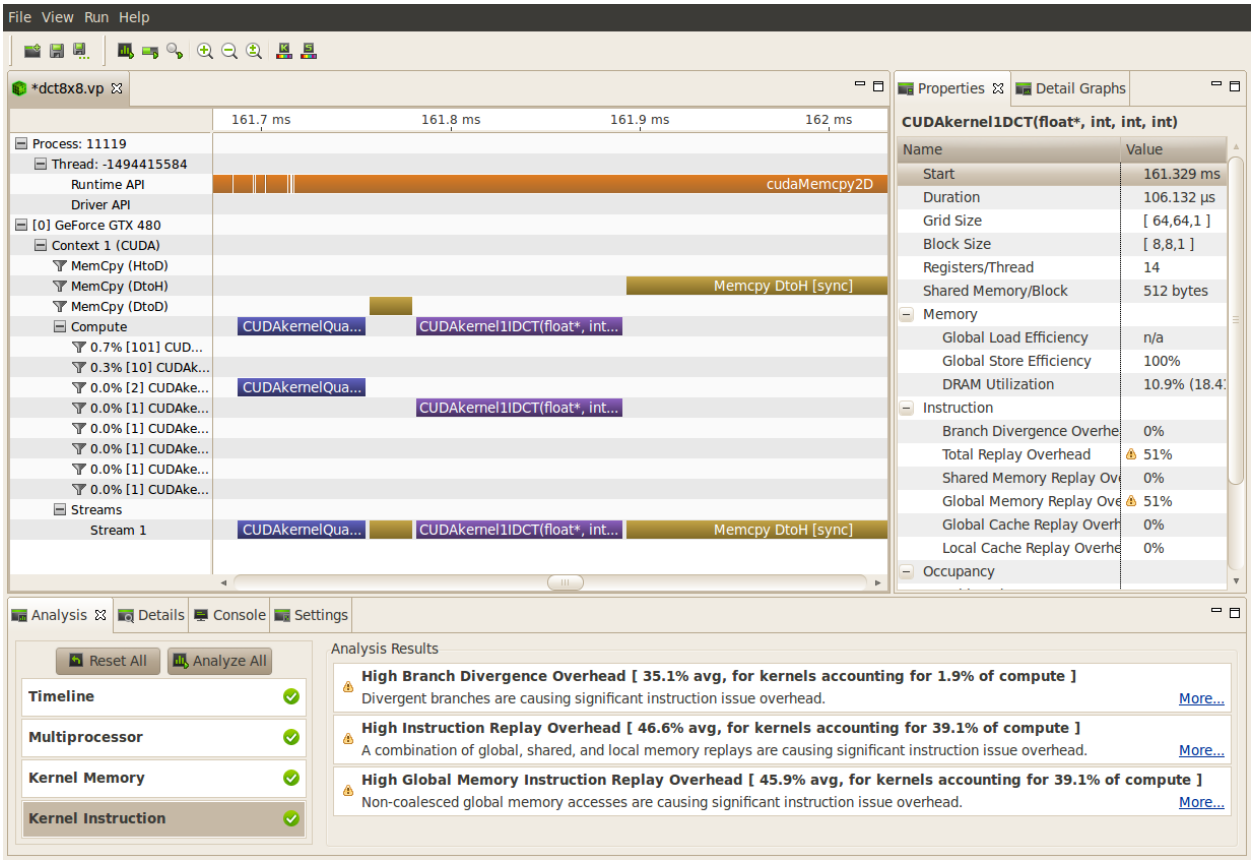
Related Work

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.



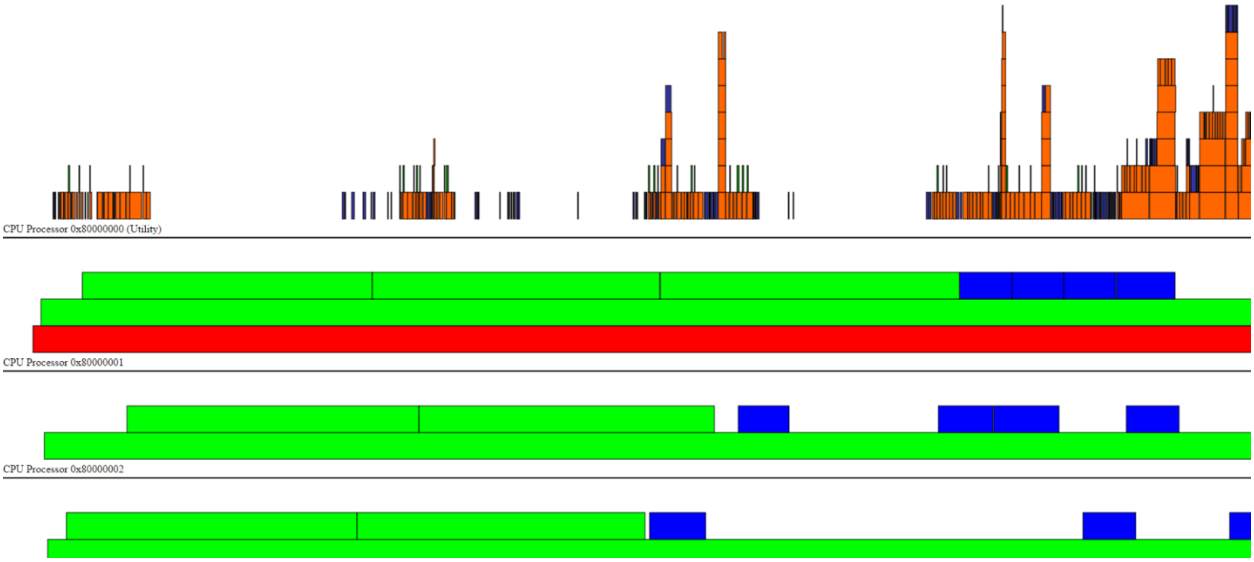
Although not particularly aesthetically-pleasing, KCachegrind is effective in showing the call structure of a particular program and interfaces well with other Linux-based code profilers. However, for highly-parallel code, KCachegrind is inherently insufficient because it relies on the profiled program 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.

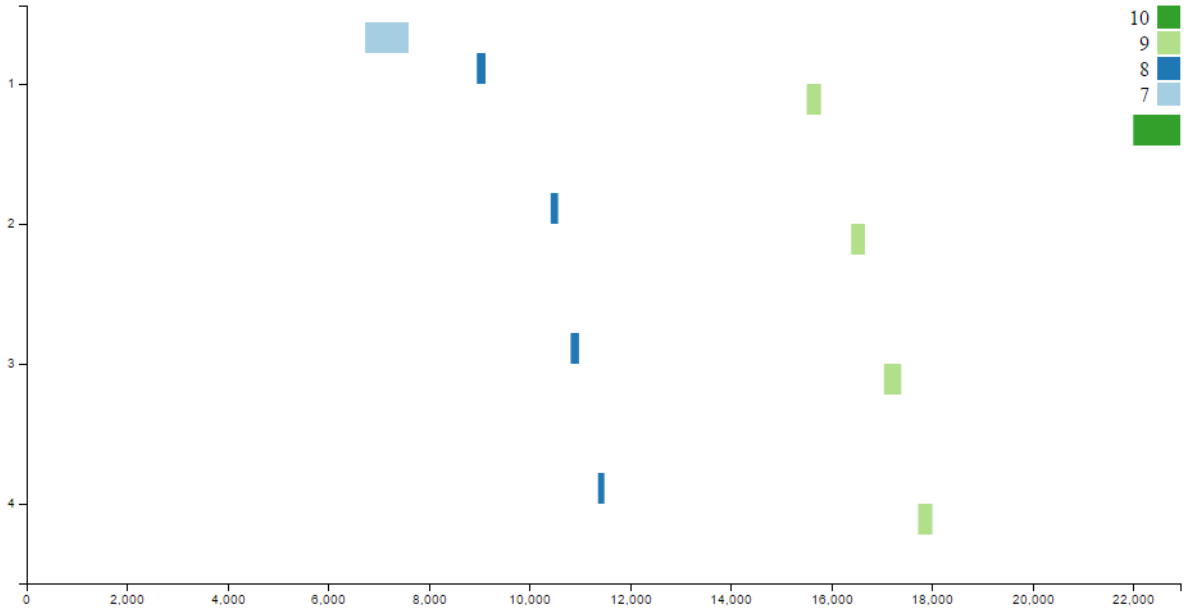
The server will maintain a copy of user submitted files, and allow the user to access them via a unique URL. This allows users to upload the data in one location, and view it from other devices, such as mobile devices, or a coworker's computer for collaboration.

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.

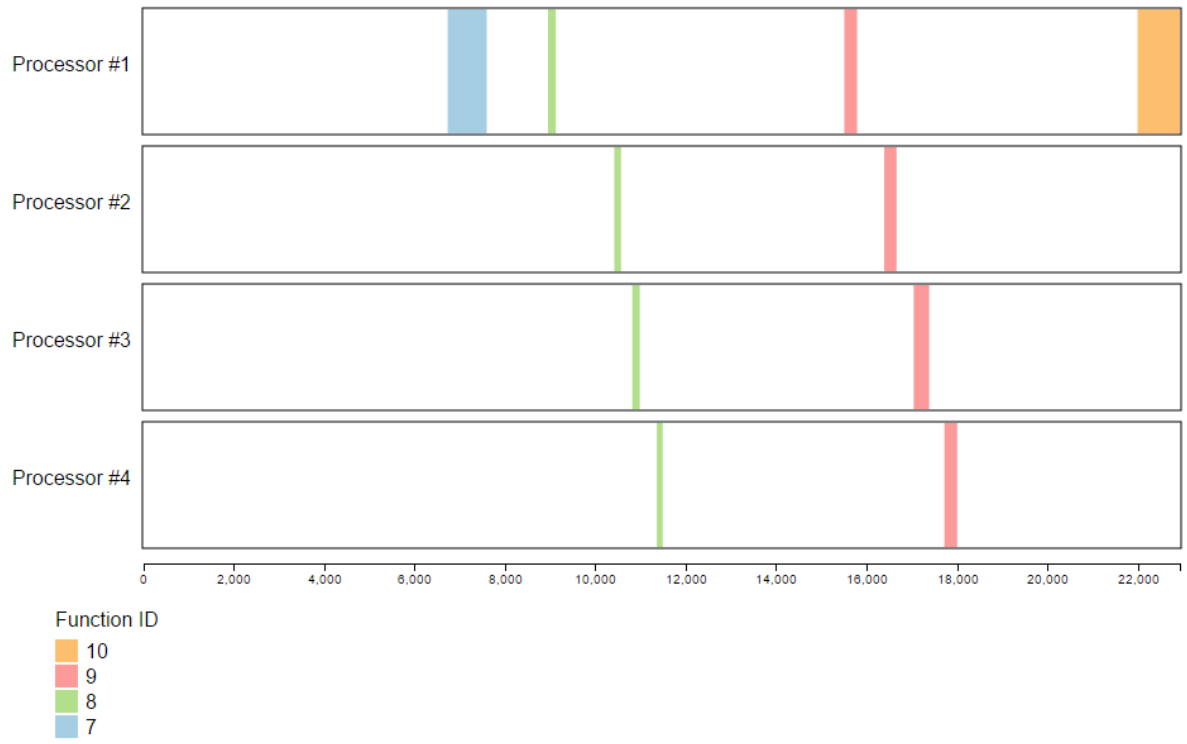
Design Evolution and Implementation

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



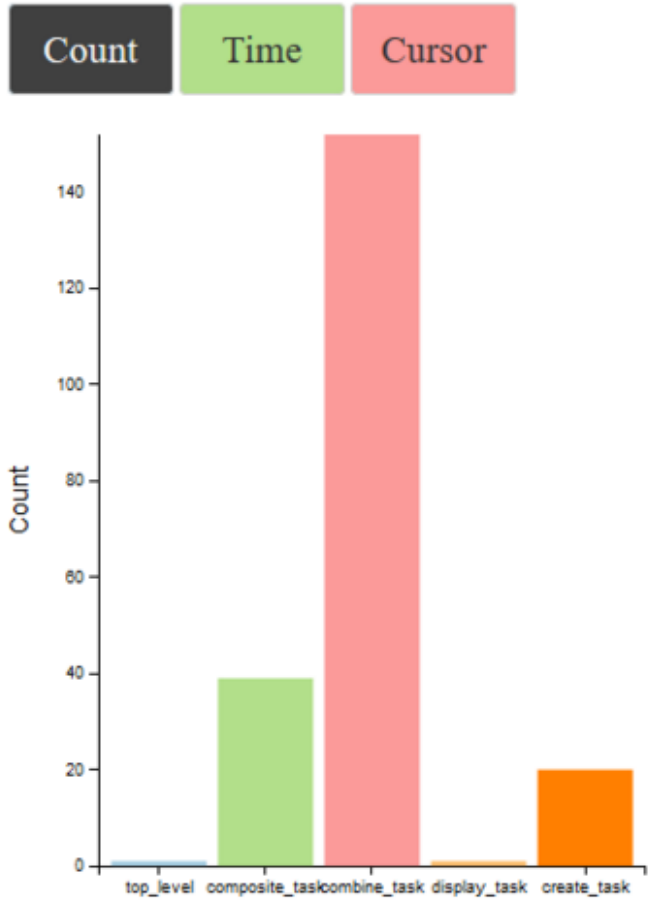
However, we decided that the spacing between components and color system made it difficult to identify features and relationships in the execution. Additionally, we opted for a more compartmentalized view of different processors rather using the numbers on the y-axis, as that erroneously implied some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

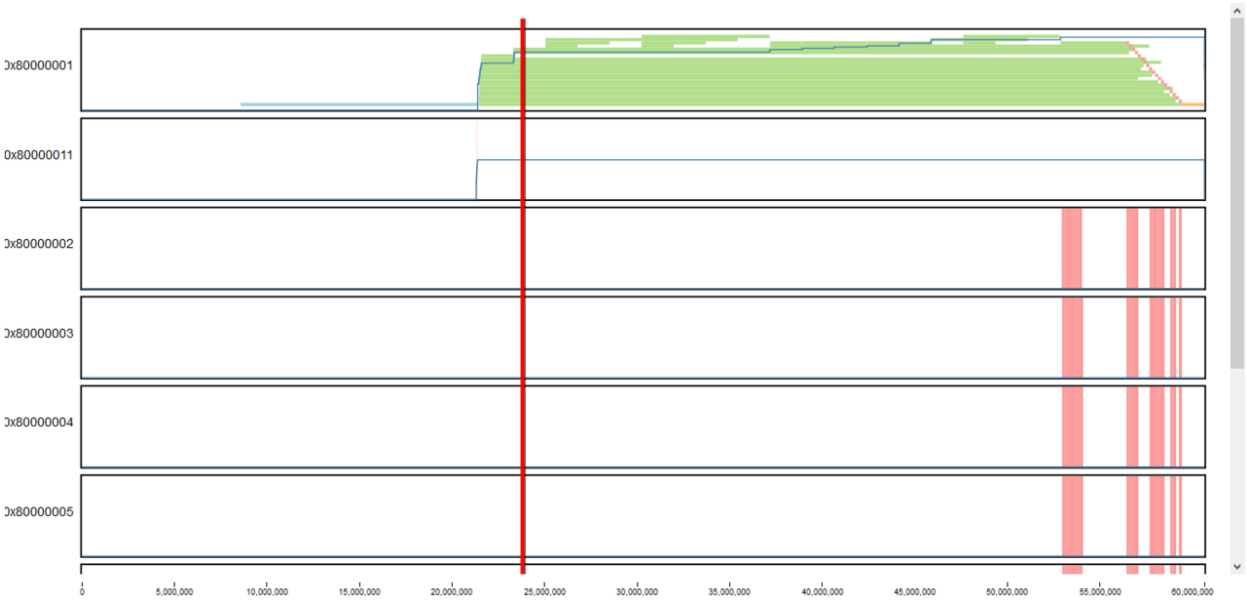


This design allowed the user to see individual processor lines much more clearly, and featured a more useful and intuitive color scale. It also utilized interactivity by providing tooltips upon mouseover of each task. This tooltip provided the ID of the function, and the visualization also featured zooming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

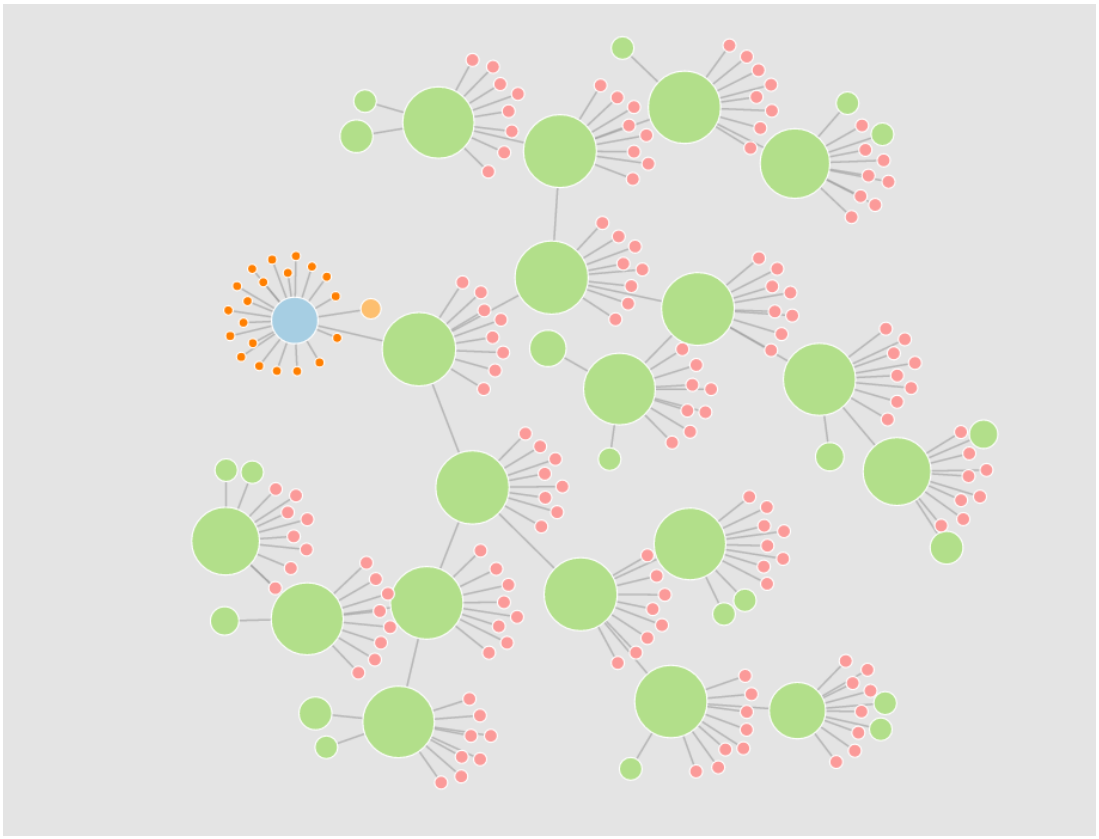
While this view provided more information about how tasks are distributed across each processor, we discovered that it didn't scale very well and didn't provide enough detail about the temporal distribution of task categories. We implemented a histogram view that tells the user the total amount of time 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.



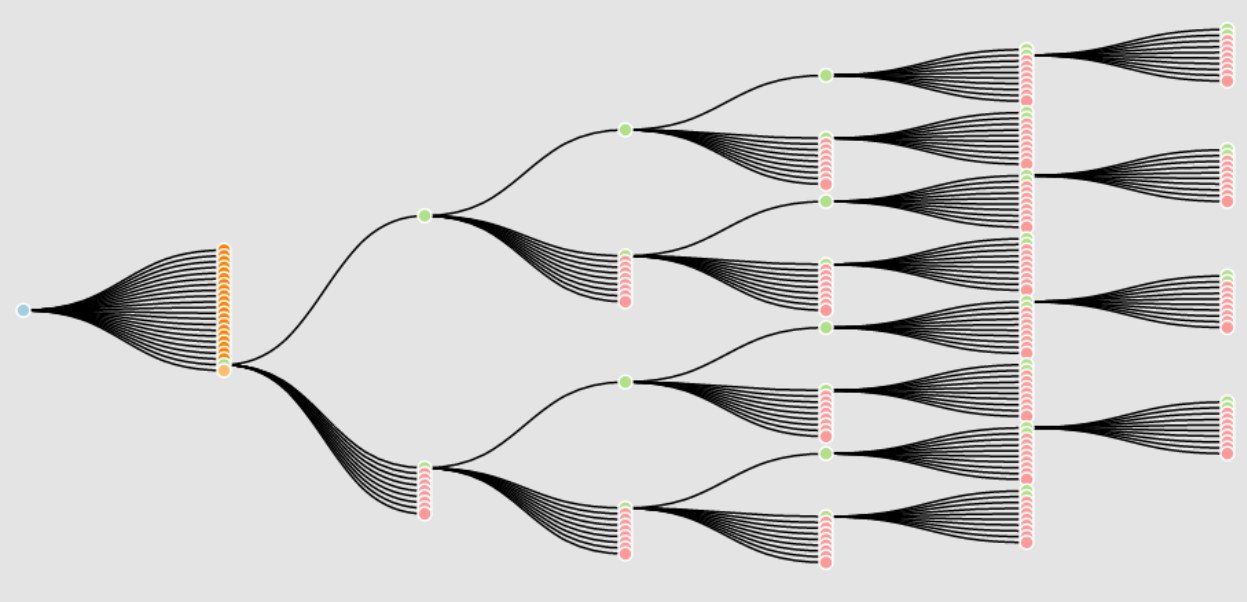
This particular dataset is for a compositing program, and we can clearly see that the combine_task occurs the most, which is to be expected. This view is helpful for identifying anomalies in task spawns. We also implemented a vertical line brush to let the user select a time slice to view a specific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



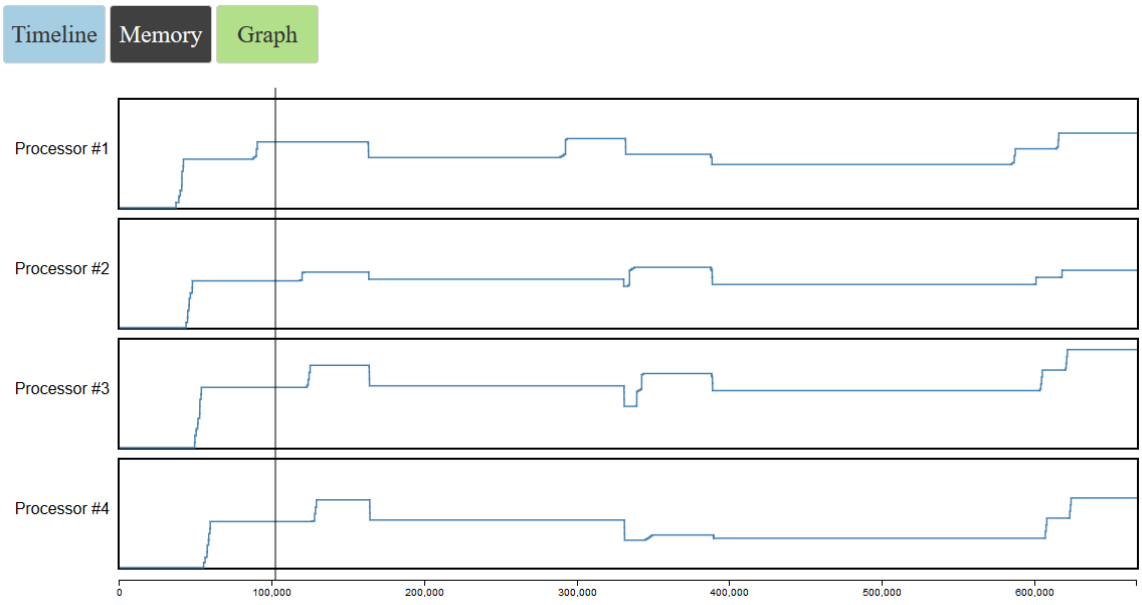
We also wanted to implement a call graph structure similar to other profilers, and started with a simple force-directed visualization, as seen below.



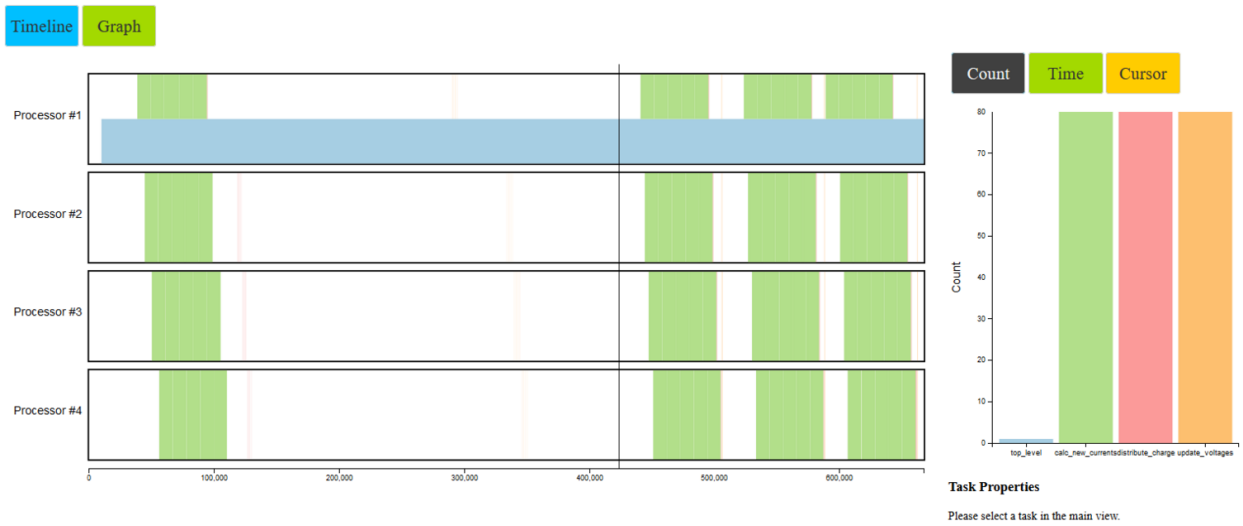
This design was quickly scrapped, as it quickly became cluttered and the bouncing circles proved to be distracting. We then tried a tree structure as shown below, which was much easier to follow.



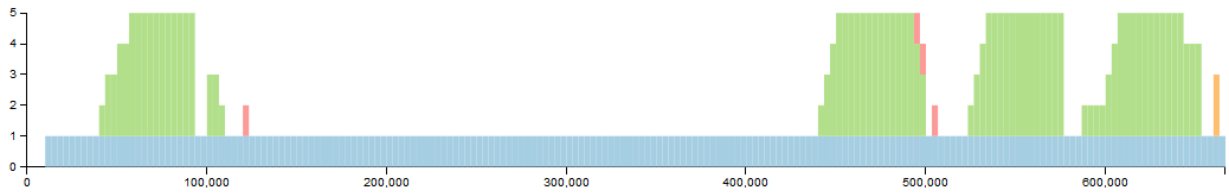
After identifying the user interaction goals, we realized that it would be useful to allow deeper exploration into memory behavior. We implemented another view to allow the user the ability to see the amount of memory allocation. We started by implementing it as a separate view, as seen below.



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 for this class works as

Evaluation

While we believe that we have a good start for our visualization system, there are still many major improvements that are currently in progress. Our next major goal is to complete a histogram that is drawn by the user brushing across the view with the mouse. This histogram allows the visualization to scale with extreme processor counts by showing the distribution of tasks at a particular point in time, as indicated by the user. This allows the user to temporally identify trends in their execution data by scrubbing across the horizontal axis.

The other major visualization component that we wish to add is a second view that represents the execution structure of the program. Similar to the aforementioned KCachegrind tool, Legion programs can be visualized in a directed acyclic graph in the shape of a tree. Although serial programs are forced to be in a strict tree structure due to the execution order, Legion programs can intelligently move information between tasks, but are spawned in a tree structure. This tool would allow users to evaluate and communicate the structure of their program in a manner not currently available for Legion.

Memory leak identification

This project was incredibly challenging from a technical perspective.