ABSTRACT
Task-based scheduling has emerged as one method to reduce the complexity of parallel computing. When using task-based schedulers, developers must frame their computation as a series of tasks with various data dependencies. The scheduler can take these tasks, along with their input and output dependencies, and schedule the task in parallel across a node or cluster. While these schedulers simplify the process of parallel software development, they can obfuscate the performance characteristics of the execution of an algorithm.

The execution trace has been used for many years to give developers a visual representation of how their computations are performed. These methods can be employed to visualize when and where each of the tasks in a task-based algorithm is scheduled. In addition, the task dependencies can be used to create a directed acyclic graph (DAG) that can also be visualized to demonstrate the dependencies of the various tasks that make up a workload. The work presented here aims to combine these two data sets and extend execution trace visualization to better suit task-based workloads.

This paper presents a brief description of task-based schedulers and the performance data they produce. It will then describe an interactive extension to the current trace visualization methods that combines the trace and DAG data sets. This new tool allows users to gain a greater understanding of how their tasks are scheduled. It also provides a simplified way for developers to evaluate and debug the performance of their scheduler.
Developers could express their tasks and various dependencies while allowing a library to schedule the tasks in parallel while respecting the dependencies presented by the developer.

There are a number of scheduling libraries that supply the infrastructure for task-based computation. They all allow the developer to concentrate on the details of the algorithm while a runtime system schedules the tasks in parallel. The user never has to explicitly schedule a task or move the necessary data for each task. While this certainly makes the development of parallel software easier, it also obfuscates when and where the tasks are executed.

Execution trace visualizations have been employed to provide users and developers a greater understanding of their software. However, these tools are relatively static and can be improved for the workloads of task-based schedulers. The visual information-seeking mantra of “overview first, zoom and filter, then details on demand” [24] certainly applies to trace visualizations. The work presented here extends the current methods to provide users with more “details on demand” about their computational workloads.

The paper is organized as follows: Section 2 presents background information about data movement, task-based scheduling, and current trace visualization methods. Section 3 describes the new visualization technique and how it is implemented. Section 4 shows a few examples of the tool and why it is useful for software developers. Finally, Section 5 discusses potential questions which may be addressed in future work.

2. BACKGROUND AND RELATED WORK

2.1 Data Movement

Data movement is expensive in terms of time and energy. As a result, it is important for software developers to be mindful of the data movement and communication patterns. There are research and software development efforts which intentionally avoid communication in order to optimize the speed of computations [9]. Communication can generally be classified as intra-node communication or inter-node communication.

2.1.1 Inter-node Communication

Inter-node communication is the most obvious form of data movement within a distributed memory system. This is the movement of data from one node to another across the network. Inter-node communication is also the most visible form of communication to developers because they must explicitly coordinate the movement from one node to another. This communication is often implemented using explicit communication functions provided by a library such as MPI.

2.1.2 Intra-node Communication

Intra-node communication can be easily overlooked because developers do not need to explicitly coordinate any intra-node communication. This communication is transparently provided to the developers by the hardware that moves data through the memory hierarchy. This type of communication should not be overlooked if high performance software is desired. Non-uniform memory access (NUMA) machines are a common type of shared memory machines. NUMA machines have multiple CPUs and NUMA nodes. The memory appears to be uniform to the user but the underlying hardware is responsible for providing the correct data from the correct memory location. This means the time it takes a CPU to access a segment of memory can vary based on the NUMA node or CPU cache currently containing the requested data. In practice, this means a computation performed using data on the “other side” of the machine can take longer than if the computation is performed on the CPU closest to the current location of the data.

2.2 Task-based Scheduling

Several software systems have been developed to focus on task-based scheduling. Systems such as SMPSs, StarPU, QUARK, PARSEC, and OpenMP 4.0 allow the developers to define their algorithms in terms of a series of tasks. The schedulers are generally classified as either task-superscalar schedulers or dataflow schedulers.

Task-superscalar schedulers allow the developer to define each task with input and output parameters. The scheduling library uses the order of the task definitions in conjunction with the input and output parameters to determine task dependencies. These dependencies are used to build a directed acyclic graph (DAG) of dependencies. The underlying runtime then uses the DAG to schedule the tasks across the resources while respecting the dependencies inferred based on the input and output parameters for each of the tasks.

SMPSs [8, 20, 19, 9, 5, 15, 22], StarPU [1, 3, 2], QUARK [18], and task-based scheduling in OpenMP 4.0 are considered task-superscalar schedulers. Several of these have extensions allowing them to support distributed workloads. However, the inherent bottleneck of unrolling the DAG and scheduling the tasks has limited the scalability of this solution. This means this class of schedulers is primarily used in the context of shared memory systems.

PARSEC [10], however, is a dataflow scheduler requiring explicit dependencies from the developer but it provides much greater scalability. The computation is represented in a job description format (JDF) file defining the tasks and dependencies in a compact format. This format allows the runtime to determine dependencies without unrolling the entire DAG. The ability to determine dependencies independently makes the runtime far more scalable. PARSEC provides the underlying scheduling and runtime for a scalable dense linear algebra library called DPLASMA.

2.3 DAG

Task-based schedulers ultimately rely on the dependencies between tasks. Whether the developer explicitly states the task dependencies or the scheduling library infers them, the data dependencies must be observed in order to ensure accurate computation. These dependencies are often represented by a DAG.

Figure 4 shows the DAG for a small linear algebra problem that only has 55 tasks. Each vertex in the graph represents a task and is depicted in the figure by an oval. (Each oval
Figure 1: This graph is an example of a small DAG from a numerical linear algebra application. The tasks are labeled and colored by class and each arrow represents a data dependency. Each of the edges in the graph represents the dependencies that must be observed when scheduling the tasks. The data set produced by a small numerical linear algebra algorithm from the the PLASMA library was used to generate Figures 1, 2, 3, and 4. The algorithm was executed on a single 8-core CPU. This small problem size was selected to illustrate the underlying structure of the problem rather than a real world application. Larger, more realistic problem sizes are used later in the paper.

The SMPSs, StarPU, PARSEC, and QUARK libraries generate the DAG in a DOT file which can be used by many applications and libraries to visualize and interact with the DAG. Figure 1 was produced from the execution of the workload using QUARK. The resulting DOT file was visualized using the GraphViz toolkit.

The TEMANAJO project [11] also aims to visualize the task dependency graphs for task-based parallel computing. The project gives the developer a visualization of the dependencies but it is primarily used for debugging.

2.4 Trace

Execution traces have been used to visualize parallel computing for many years. Figure 2 shows a trace for a small problem. (This is the same problem used to create the DAG in Figure 1. The same color scheme is used for the tasks in the DAG as well as the trace.) The workload was executed on 8 cores of a shared memory system. The x-axis is used to depict the time (in milliseconds) while each row is used to represent a single core on the system. Each of the rectangles represents one of the tasks comprising the parallel workload. The rectangles in this figure are colored to convey the type of task represented. However, the color and texture of the boxes can be used to depict any number of task properties.

Unfortunately, the wide variety and complex interoperability of trace collection, analysis, and visualization tools make it difficult to accurately describe the landscape of the field briefly. There are several trace collection tools producing intermediate data formats which can often be converted to use a variety of analysis and visualization tools to analyze the execution trace.

SLOG-2 and Jumpshot [12] were developed at Argonne National Laboratory for trace collection and analysis. The focus of the work was to provide a file format and viewer that could scale to very large trace sizes. The trace information is stored in the file hierarchically which provides efficient access to any portion of the trace.

The TAU performance system [23] focuses on providing an instrumentation toolkit (Program Data Toolkit or PDT) that collects the event data. TAU also provides ParaProf and PerfExplorer for detailed analysis and visualization of many of the performance characteristics of an algorithm. The tracing information can also be converted to a variety of common formats for viewing with several event trace viewers.

Researchers at the Barcelona Supercomputing Center have also developed an ecosystem of tools for collecting and analyzing event trace data. Extrae [16] is used to instrument a parallel program and collect the event trace. Paraver [21] is used to visualize the trace while Dimemas [7] is used to manipulate it and simulate execution under a variety of conditions.

EZETrace [26, 4] was built on top of the Generic Trace Generator [13] library which is capable of producing various trace file formats including Open Trace Format (OTF) and Pajé. These traces can be viewed with the ViTE trace viewer or Vampir.

Arguably the most common trace viewer and analysis toolkit in the field is Vampir. This viewer has the ability to view trace files in Open Trace Format (OTF) or OTF2 which can be collected using a variety of instrumentation toolkits. Vampir also provides a number of features and tools allowing the developer to interact and analyze the event trace [17].

Finally, the PARSEC project [10] has implemented an embedded execution data collection framework creating a binary file with a variety of performance information including
tasks are computed on the same core. Even if the tasks are performed on a different core or device, the data should already be in cache and the communication cost should be relatively low. On the other hand, if the tasks are executed consecutively on the same core or device, the data must have already been evicted from the processor cache.

Perhaps the most obvious way to depict the execution trace and the task dependencies is to visualize the trace and the DAG simultaneously. Adding interactivity with mouseovers or mouseclicks would allow the user to select a task in the trace which would also highlight the corresponding task in the DAG. The opposite could also be true. However, the size of the DAG and trace quickly grow to extremely large datasets which make it difficult for the user to comprehend the information on a problem of any reasonable size. As a result, the two visualizations needed to be combined into a single visual representation.

The first version of the tool employed the depictions used by many common MPI tracing tools. The tasks were represented using the same methodology as Figure 2. A simple line between tasks, a common visual representation for MPI communication, was used to represent the dependencies and data communications required by the algorithm. This representation, applied to the same data used in Figures 2 and 4, can be seen in Figure 3.

Figure 3 now shows all of the tasks and their dependencies in one visual space. However, even for small problems the number of dependencies quickly overwhelms the user. The visual “hairball” shown in Figure 3 can be greatly improved by making the dependency lines an interactive feature.

Figure 3 demonstrates a visualization of the small problem presented earlier, but with interactive features. Without having the mouse hover over any of the tasks in the diagram, the users sees a trace that looks identical to the trace in Figure 2. When the user’s moves the mouse over one of the tasks, however, the trace highlights the task as well as the tasks for which it is waiting. It also highlights any tasks that are waiting for it to complete. The tasks are also connected to the task in focus to represent the dependencies. In terms of the DAG, each of the lines represents the edges that are connected to the highlighted task. Lines connected to tasks earlier in the trace are edges directed into the highlighted node. Conversely, lines connected to tasks in the future represent edges leaving the highlighted node in the DAG.

Many of these features can be configured to allow the user to adjust the behavior of the visualization. For example, the user may want to only highlight (add a black border to the task) dependencies without drawing the lines. The user may want to only show the tasks in the past or only show the tasks waiting in the future. The user may also want to see tasks more than one step away from the task in focus. These are all features the user can configure in order to make the visualization useful.

### 3.1 Implementation

In order to create the visualization, two separate data sets must be combined. The first data set is from the execution trace. This data generally contains information about each task including when the task started, when the task ended, on what core it executed, and likely the type of task. This data set may also contain other information about the
Figure 4: This trace demonstrates the basic interactivity of the software. The user has selected the red task in the middle by placing the cursor over it. The trace highlights the dependencies on either side of that task with a black border and a line to depict the dependency.

task. For example, the code may be instrumented with PAPI counters which collect information about cache misses or instruction counts.

The second data set is the DAG of tasks and dependencies. QUARK and PARSEC currently provide the DAG for the workload in a DOT file. This information can be used to visualize the DAG using any number of software libraries. The file can also be parsed to identify the dependencies (edges) of the graph.

The challenging part of combining these data sets is finding the tasks in the execution trace corresponding to each of the nodes in the DAG. The earlier discussion about tracing collection and storage utilities highlights the challenge of dealing with data produced by different schedulers and instrumentation libraries. The code currently supports data from QUARK and PARSEC, although it could be extended to support various other data formats and schedulers in the future.

QUARK provides a task id that is unique for each task in the execution. The task id is included in each node of the DOT file that contains the DAG representing the computation. The trace was collected by instrumenting each task with a start and stop time stamp as well as the corresponding task id. Once the two files are produced, the tasks in the DOT file are matched to the corresponding tasks in the trace.

PARSEC, however, automatically collects and records all of the necessary data to match tasks from the DAG with tasks from the trace. The DAG is collected in pieces on each node and post-processed to generate a single DOT file. The trace is collected in the PTT file discussed earlier. The distributed nature of the PARSEC execution makes it difficult to have a single task id that is unique for each task. Therefore, each task has three id properties (hid, did, and tid) that combine to create a unique identifier for each of the tasks. The three ids are present in the DOT and PTT files created by PARSEC and are used to match the tasks from the two data sets.

There is no standardized method for uniquely identifying tasks in a trace or DAG at this time. In order to port this method to the data provided by other schedulers, the user must be able to uniquely identify tasks in the trace as well as the DAG.

The visualization is implemented as a client-server architecture. The server is implemented in Python while the client is implemented using Javascript. The Python server is better suited to do the heavy computational tasks and performing various analytical tasks. Javascript (and associated libraries) are well suited for making interactive visualizations.

This architecture also gives developers a flexible way to improve, adjust, and expand the capabilities of this system. The next section will demonstrate one such extension using a kernel density estimation (KDE) plot in conjunction with the trace visualization.

4. APPLICATION & ANALYSIS

The visualization method presented in section 3 can be applied independently like the example in Figure 4, but it is also possible to employ this technique in conjunction with other data visualization techniques. Figures 5 and 6 present two examples of the visualization library applied to a real world problem while demonstrating how it can be used.

Figure 5 demonstrates the new trace visualization on the same linear algebra application presented earlier. However, this example uses a larger problem size which more closely resembles a real world problem. The tasks in the trace have been colored based on their relative speeds. Tasks colored in white are close to the average for a particular class of task. Blue tasks are slower than average and green tasks are faster than average. The intensity of the color is scaled based on how far the runtime is from the average. This representation allows the user to quickly determine which tasks are slower or faster than average. There are several blue tasks at the
Figure 5: An example of the trace utility applied to a linear algebra workload. The tasks are colored based on their relative speeds.

Figure 6: An example of the trace utility applied to a linear algebra workload. The KDE plot was used to highlight tasks in the trace based on execution time.
beginning of the trace. This is likely due to library and data initialization costs at the start of the algorithm.

Several of the tasks near the end are green indicating they are faster than average. It is likely this is caused in part by the smaller number of tasks being executed and the resulting reduction in memory contention. By selecting one of the brightest green tasks, the visualization also shows the user the data dependencies all come from the same CPU. Therefore, the data is likely to be in cache instead of the main memory or the cache on another chip. As a result, the data movement is likely to be much faster than other tasks.

The plot at the bottom left shows four KDE curves for the four types of tasks. The red KDE curve is highlighted which indicates the selected task is part of this density estimator. The black vertical line indicates where the selected task falls in relation to the distribution of task times. In this case, the selected task is likely one of the fastest of its kind.

One of the elements of the visual information-seeking mantra is the ability to filter the data and make it easier to focus on information deemed most interesting by the user. Figure 6 demonstrates how the KDE plot can be used to highlight tasks in a specific range with a filter based on execution times. In this case, the user is interested in the relatively slow tasks. The tasks in the trace which have execution times falling within the range of the gray box on the KDE plot are highlighted, while the others tasks have been obscured by a reduction in opacity.

The user has selected an orange task in order to determine why it was relatively slow. The visualization shows two dependencies for the selected task. One is on the same core while the other is on another CPU. However, closer inspection reveals two other tasks were executed on the same core between the two tasks linked in the trace. Thus, the data from the dependency has likely been evicted from the cache. As a result, the task likely had to load two dependencies from memory or another CPU which caused an increase in task execution time.

The new dependency visualization technique is intended for people who develop task-based schedulers as well as the developers who use them. The developer of a scheduler can use this method to evaluate the performance characteristics of their software. The execution traces and the DAGs produced by task-based schedulers have been available for many years, but the two data sets have never been combined. This fusion of data allows developers to access performance information that is not available from each of the data sets individually.

Interactivity allows the user to access all available trace data without being overwhelming. Figure 3 demonstrates how quickly the tracing data sets can overwhelm the user. By allowing the user to access dependency information on demand, we have made the entire data set available without the overwhelming visual “hairball” that can occur when the dependencies are visualized simultaneously.

Task-based scheduling of computational workloads is going to be an important part of the future of high performance computing. The work presented here will provide developers with improved tools for analyzing the performance characteristics of their software.

6. CONCLUSIONS
The work presented here has provided two primary contributions.

Task-based scheduling provides rich data sets that can be used to give developers a greater understanding of the performance characteristics of their software. The execution traces and the DAGs produced by task-based schedulers have been available for many years, but the two data sets have never been combined. This fusion of data allows developers to access performance information that is not available from each of the data sets individually.

The use of the KDE plot in a coordinated view shows how the trace visualization presented in this paper can be used in conjunction with other visual representations of the data set. The development of other complementary representations will give the user the ability to filter the data set and decrease the time spent searching for insights.

7. ACKNOWLEDGMENTS
This work was supported by the National Science Foundation under grant ACI-1339822 and the Department of Energy under award #DE-SC00010682.

8. REFERENCES


