Visualizing Execution Traces with Task Dependencies

Blake Haugen
Innovative Computing Laboratory
University of Tennessee
Knoxville
bhaugen@utk.edu

Stephen Richmond
Innovative Computing Laboratory
University of Tennessee
Knoxville
srichmo1@utk.edu

Jakub Kurzak
Innovative Computing Laboratory
University of Tennessee
Knoxville
kurzak@icl.utk.edu

Chad A. Steed
Computational Sciences and Engineering
Oak Ridge National Laboratory
csteed@acm.org

Jack Dongarra
Innovative Computing Laboratory
University of Tennessee
Knoxville
Oak Ridge National Laboratory
University of Manchester
dongarra@eecs.utk.edu

ABSTRACT

Task-based scheduling has emerged as one solution to the complexity of parallel computing. When using these tools, 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 easily 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.

Keywords

Task-Based Scheduling, Execution Trace, Data Movement, DAG

1. INTRODUCTION

In the last decade, the microchip industry has shifted to a multicore paradigm and consequently altered the path of software development. Prior to this shift, developers could expect their software to see performance improvements with each new generation of computing architecture because the clock frequency of the new chip would boost the performance. During this era, modifications to software were not necessary to increase performance. Eventually, the frequency of new microprocessors stabilized while the number of cores began to increase. As a result, developers must now modify their software to make performance gains on new hardware [25]. However, adding parallelism to software is generally a non-trivial task.

Unix platforms provide POSIX threads (Pthreads) that can be used to develop software that utilize multiple cores on multicore hardware. Unfortunately, developing code that efficiently maps a computational problem to Pthreads is a challenging task requiring expert level knowledge for most problems. In order to deal with the challenges associated with Pthreads, several higher level APIs are available to simplify the development of parallel software. One of the most common software tools is OpenMP. This library employs compiler directives to parallelize relatively simple loops across the cores available in the system [14]. Early versions of this paradigm were sufficient for many problems but lacked several features necessary for more complex algorithms as well as the increasingly heterogenous computing systems.

Task-based parallel computing emerged as an alternative to the simple fork-join parallelism employed in early versions of OpenMP. In the task-based paradigm, the developers are re-

sponsible for expressing their computation as a set of tasks. These tasks can be scheduled in sequence on a single CPU. However, it is often possible to execute multiple tasks in parallel, so long as the task and data dependencies are satisfied. 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, 6, 5, 15, 22], StarPU [1, 3, 2] and QUARK [18], 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.

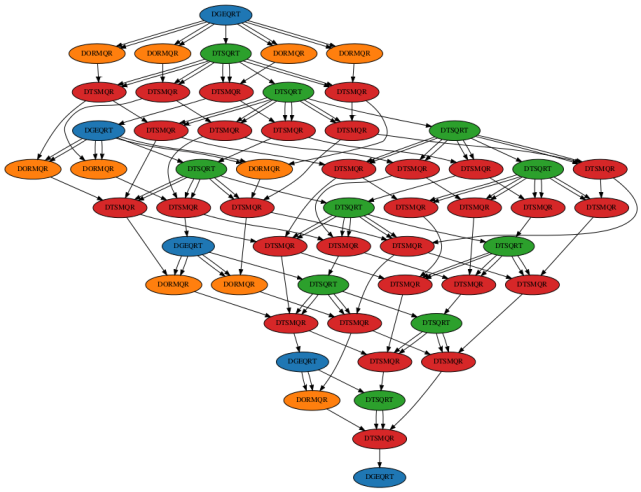

Figure 1: An example of a small DAG from a numerical linear algebra application.

Figure 1 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 is labeled with the type of task it represents.) 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 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. SMPSs and StarPU also have utilities to generate a file describing the DAG.

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 also provides tools to assist with debugging tasks.

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.

EZTrace [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 an execution trace. The data in the PARSEC Trace Table (PTT) can be read and analyzed using a Python library or converted to a Pajé trace file which can be viewed using ViTE.

3. VISUALIZATION DESIGN

The concept of visualizing communication in an execution trace is not new and has been implemented in many trace environments. However, the current methods can be improved. The current tracing methods often instrument the code automatically for the user. Each invocation of a function is recorded with a starting and stopping time as well as information about the computational element performing the computation. In the context of an MPI program, it is also possible to instrument all of the communication

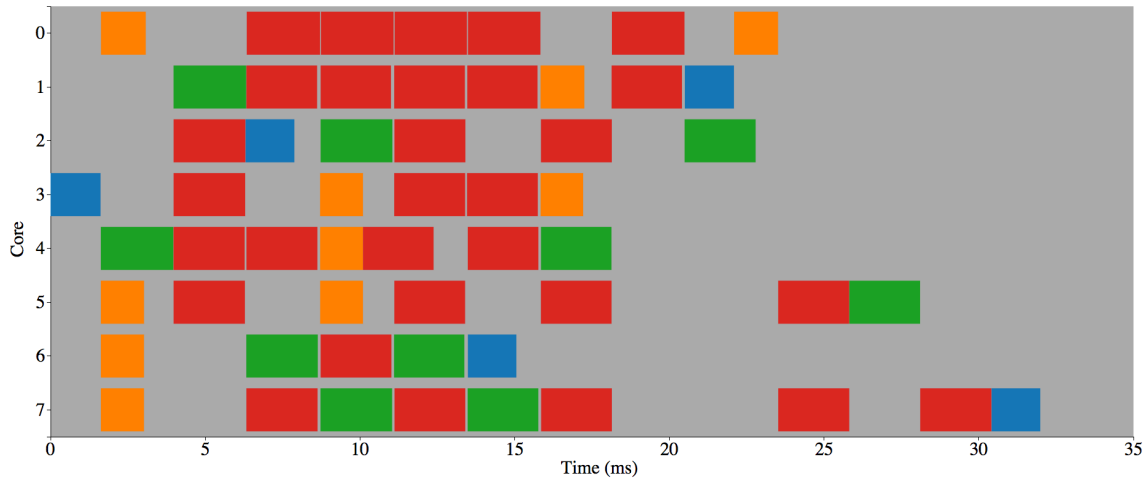

Figure 2: An example of a small trace from a numerical linear algebra application.

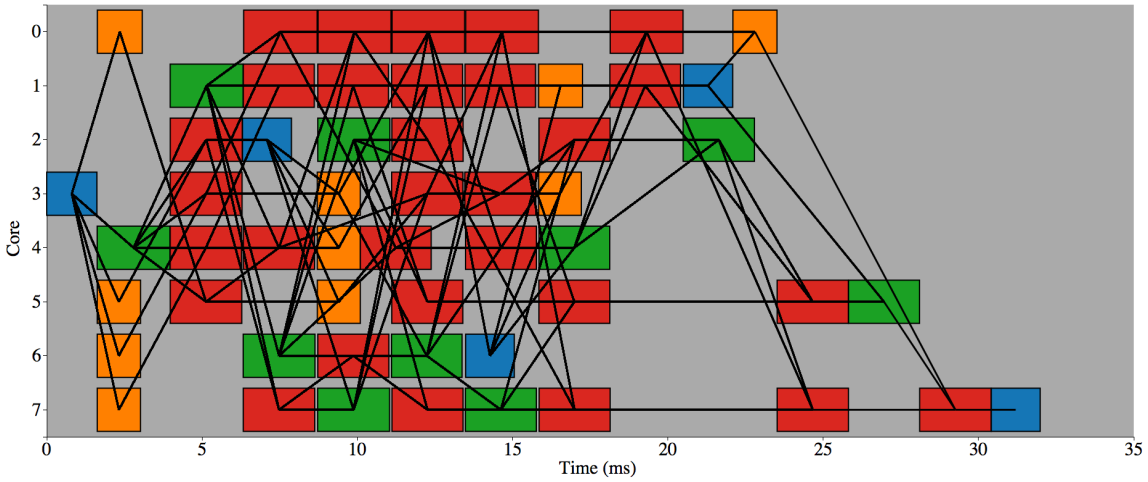

Figure 3: The first version of the tracing utility employing lines to depict the dependencies in the parallel workload

functions. The communication functions are traditionally represented with a line between the two nodes on the execution trace. This depiction clearly communicates that data movement has occurred, but it is often overwhelming to the user.

This communication visualization method is perfectly suited for software which uses MPI because each time the program moves data it must call an MPI communication function that can easily be tracked. In a shared memory setting, however, this methods breaks down. There is no communication function which can easily be instrumented to log data movement. The user must have knowledge of the algorithmic structure and what data movement must occur. It is hard to know exactly how the data transfer takes place but it must occur in order for the computation to continue. The task-based schedulers can provide information about where the computations happen and where the data was before it was performed.

It should be noted that a dependency between tasks implies the later task must wait until the earlier task has completed. This means the second task is waiting for some piece of data from the earlier one. If these tasks are executed consecutively on the same 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 performed on a different core, device, or node the scheduler must move data or communicate. As a result, when considering task-based scheduling, a dependency implies the requirement of communication unless the tasks are computed on the same core. Even if the tasks are computed on the same node it is possible data will have to move through the memory hierarchy if the data has 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 the DAG. The opposite could also be true. However, the size of the DAG and trace quickly grow to extremely large

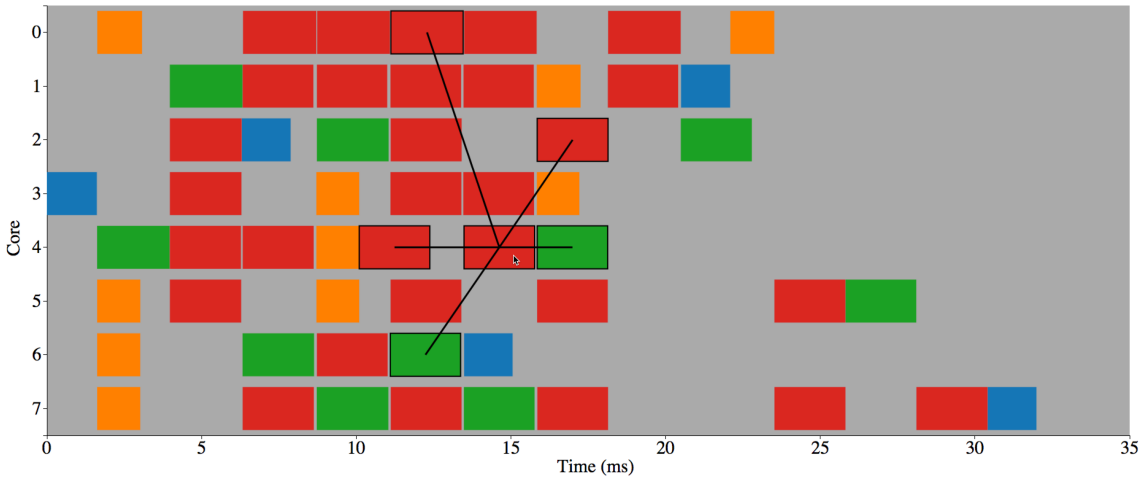

Figure 4: The execution trace shows dependencies before and after the task highlighted in the middle

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 1, 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 4 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 users 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 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 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 which can be used to match the tasks in the DAG with the execution trace. PARSEC is slightly more challenging because it lacks a single explicit task id. Instead, PARSEC uses three data points to uniquely identify each task in the DAG and the PTT (PARSEC's execution trace format).

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 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 allows the user to see what data movement has occurred in the process of the computation. It would have been difficult to gain knowledge of these data movements without a visualization technique like this.

5. FUTURE WORK

There are several ways this work can be expanded upon in the future. The most obvious would be support for more schedulers and data formats. This will likely require collaboration with the developers of the other task-based schedulers in order to determine whether the DAG and trace information is available and can be joined.

While this technique scales to several thousand tasks (computationally and visually), it is not capable of displaying enormous traces. Even if they could be rendered, they may contain too much information for the user to comprehend. For example, a trace with several million tasks across hundreds or even thousands of multicore nodes provides too much data for the user to mentally process or render. New methods for aggregating trace data will be necessary to give developers a higher level overview of the trace than is currently available. In terms of the visual information-seeking mantra, the overview of trace visualizations must be improved to give a higher level overview than is currently available.

6. CONCLUSIONS

Task-based scheduling of computational workloads is going to be an important part of the future of high performance computing. The work presented here expands the common execution trace techniques to include task dependency information. The fusion of these two important data sets allows scheduler developers to analyze the performance characteristics of their scheduler while providing developers a new tool to gain understanding of how their computations are mapped to parallel computing hardware.

7. ACKNOWLEDGMENTS

This work was supported by the National Science Foundation under grant ACI-1339822.

8. REFERENCES

[1] C. Augonnet and R. Namyst. A unified runtime system for heterogeneous multicore architectures. In *Proceedings of the Euro-Par 2008 Workshops - Parallel Processing*, Lecture Notes in Computer Science, pages 174–183, Las Palmas de Gran Canaria, Spain, August 2008. Springer. DOI: 10.1007/978-3-642-00955-6_22.

[2] C. Augonnet, S. Thibault, R. Namyst, and P. Wacrenier. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency Computat. Pract. Exper.*, 23(2):187–198, 2011. DOI: 10.1002/cpe.1631.

[3] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. In *Proceedings of the 15th International Euro-Par*

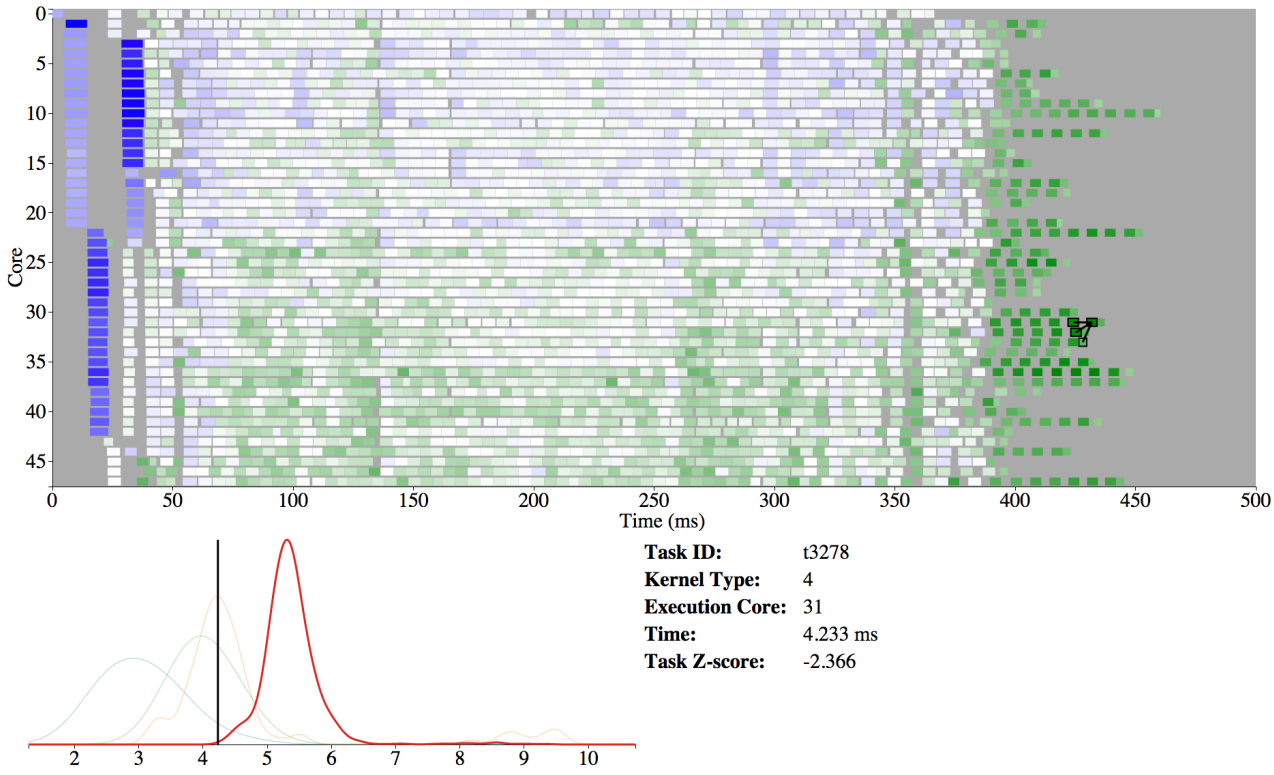

Figure 5: An example of the trace utility applied to a linear algebra workload. The tasks are colored based on their relative speeds.

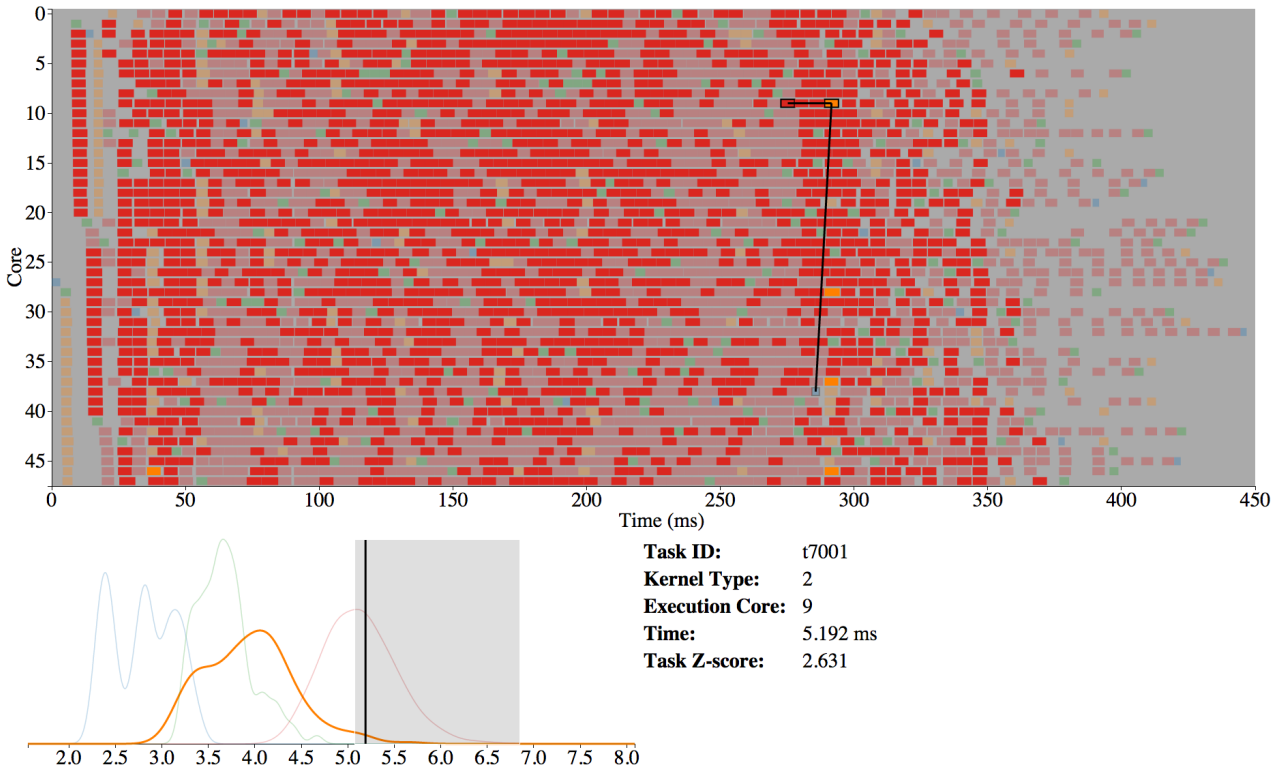

Figure 6: An example of the trace utility applied to a linear algebra workload. The KDE plot was used to highlight tasks in the trace based on execution time.

Conference on Parallel Processing, Euro-Par '09, pages 863–874, Berlin, Heidelberg, 2009. Springer-Verlag.

[4] C. Aulagnon, D. Martin-Guillerez, F. RuÃl', and F. Trahay. Runtime function instrumentation with eztrace. In I. Caragiannis, M. Alexander, R. Badia, M. Cannataro, A. Costan, M. Danelutto, F. Desprez, B. Krammer, J. Sahuquillo, S. Scott, and J. Weidendorfer, editors, *Euro-Par 2012: Parallel Processing Workshops*, volume 7640 of *Lecture Notes in Computer Science*, pages 395–403. Springer Berlin Heidelberg, 2013.

[5] E. Ayguadé, R. M. Badia, F. D. Igual, J. Labarta, R. Mayo, and E. S. Quintana-Ortí. An Extension of the StarSs Programming Model for Platforms with Multiple GPUs. In *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, pages 851–862. Springer-Verlag, 2009.

[6] R. M. Badia, J. R. Herrero, J. Labarta, J. M. Perez, E. S. Quintana-Orti, and G. Quintana-Orti. Parallelizing dense and banded linear algebra libraries using SMPSs. *Concurrency Computat. Pract. Exper.*, 21(18):2438–2456, 2009. DOI: 10.1002/cpe.1463.

[7] R. M. Badia, J. Labarta, J. Gimenez, and F. Escale. Dimemas: Predicting mpi applications behavior in grid environments. In *Workshop on Grid Applications and Programming Tools (GGF8)*, volume 86, pages 52–62, 2003.

[8] R. M. Badia, J. Labarta, R. Sirvent, J. M. Perez, J. M. Cela, and R. Grima. Programming grid applications with GRID Superscalar. *J. Grid Comput.*, 1(2):151–170, 2003. DOI: 10.1023/B:GRID.0000024072.93701.f3.

[9] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz. Minimizing communication in numerical linear algebra. *SIAM Journal on Matrix Analysis and Applications*, 32(3):866–901, 2011.

[10] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, A. Haidar, T. Herault, J. Kurzak, J. Langou, P. Lemarinier, H. Ltaief, P. Luszczek, A. YarKhan, and J. Dongarra. Flexible Development of Dense Linear Algebra Algorithms on Massively Parallel Architectures with DPLASMA. In *Proceedings of the 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops*, IPDPSW '11, pages 1432–1441, Washington, DC, USA, 2011. IEEE Computer Society.

[11] S. Brinkmann, J. Gracia, C. Niethammer, and R. Keller. TEMANEJO - a debugger for task based parallel programming models. *CoRR*, abs/1112.4604, 2011.

[12] A. Chan, W. Gropp, and E. Lusk. An efficient format for nearly constant-time access to arbitrary time intervals in large trace files. *Scientific Programming*, 16(2-3):155–165, 2008.

[13] K. Coulomb, A. Degomme, M. Faverge, and F. Trahay. An open-source tool-chain for performance analysis. In H. Brunst, M. S. MÃijller, W. E. Nagel, and M. M. Resch, editors, *Tools for High Performance Computing 2011*, pages 37–48. Springer Berlin Heidelberg, 2012.

[14] L. Dagum and R. Menon. OpenMP: An Industry Standard API for Shared-Memory Programming. *Computational Science Engineering, IEEE*, 5(1):46 –55, 1998.

[15] A. Duran, E. Ayguade, R. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas. OmpSs: A proposal for programming heterogeneous multi-core architectures. *Parallel Process. Lett.*, 21(2):173–193, 2011. DOI: 10.1142/S0129626411000151.

[16] H. Gelabert and G. Sánchez. Extrae user guide manual for version 2.2. 0. *Barcelona Supercomputing Center (B. Sc.)*, 2011.

[17] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. S. Müller, and W. E. Nagel. The vampir performance analysis tool-set. In *Tools for High Performance Computing*, pages 139–155. Springer, 2008.

[18] J. Kurzak and J. Dongarra. Fully dynamic scheduler for numerical scheduling on multicore processors. Technical Report LAWN (LAPACK Working Note) 220, UT-CS-09-643, Innovative Computing Lab, University of Tennessee, 2009.

[19] J. M. Pérez, R. M. Badia, and J. Labarta. A dependency-aware task-based programming environment for multi-core architectures. In *Proceedings of the 2008 IEEE International Conference on Cluster Computing, 29 September - 1 October 2008, Tsukuba, Japan*, pages 142–151. IEEE, 2008.

[20] J. M. Perez, P. Bellens, R. M. Badia, and J. Labarta. CellSs: Making it easier to program the Cell Broadband Engine processor. *IBM J. Res. & Dev.*, 51(5):593–604, 2007. DOI: 10.1147/rd.515.0593.

[21] V. Pillet, J. Labarta, T. Cortes, and S. Girona. Paraver: A tool to visualize and analyze parallel code. In *Proceedings of WoTUG-18: Transputer and occam Developments*, volume 44, pages 17–31. mar, 1995.

[22] J. Planas, R. M. Badia, E. Ayguadé, and J. Labarta. Hierarchical task-based programming with StarSs. *Int. J. High Perf. Comput. Applic.*, 23(3):284–299, 2009. DOI: 10.1177/1094342009106195 .

[23] S. S. Shende and A. D. Malony. The tau parallel performance system. *International Journal of High Performance Computing Applications*, 20(2):287–311, 2006.

[24] B. Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *Visual Languages, 1996. Proceedings., IEEE Symposium on*, pages 336–343. IEEE, 1996.

[25] H. Sutter. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's Journal*, 30(3), 2005.

[26] F. Trahay, Y. Ishikawa, F. Rue, R. Namyst, M. Faverge, and J. Dongarra. Eztrace: a generic framework for performance analysis. In *Cluster, Cloud and Grid Computing (CCGrid), 2011 11th IEEE/ACM International Symposium on*, pages 618–619. IEEE, 2011.