In Situ Visualization of Performance Metrics in Multiple Domains

Allen R. Sanderson
SCI Institute
University of Utah
Salt Lake City, UT, USA
allen@sci.utah.edu

John Schmidt
SCI Institute
University of Utah
Salt Lake City, UT, USA
jas@sci.utah.edu

Alan Humphrey
SCI Institute
University of Utah
Salt Lake City, UT, USA
ahumprey@sci.utah.edu

Michael E. Papka
Department of Computer Science
Northern Illinois University
DeKalb, IL, USA
papka@niu.edu

Robert Sisneros
National Center for Supercomputing Applications
University of Illinois at Urbana-Champaign
Urbana, IL, USA
sisneros@illinois.edu

***Abstract*—As application scientists develop and deploy simulation codes on to leadership-class computing resources, there is a need to instrument these codes to better understand performance to efficiently utilize these resources. This instrumentation may come from independent third-party tools that generate and store performance metrics or from custom instrumentation tools built directly into the application. The metrics collected are then available for visual analysis, typically in the domain in which there were collected. In this paper, we introduce an approach to visualize and analyze the performance metrics in situ in the context of the machine, application, and communication domains (MAC model) using a single visualization tool. This visualization model provides a holistic view of the application performance in the context of the resources where it is executing.**

***Index Terms*—performance visualization**

I. Introduction

As application scientists use supercomputing resources to accomplish their research, it is critical that simulation codes be instrumented to efficiently utilize these resources. This instrumentation provides insight into how codes execute and the bottlenecks that prevent them from efficiently utilizing these resources. These bottlenecks may come from I/O, load balancing, communication, unoptimized code, or other factors.

Many profiling tools are available to instrument a code [1]–[4]. Existing profiling tools focus on specific parts of a machine or code, e.g. the I/O system. Furthermore, these tools typically visualize the performance metrics in the domain that relates to how they were measured without any other context, e.g., MPI metrics are collected for individual processes, but not the machine nodes/cores being utilized. That is, these tools do not give a complete picture of a code's performance. Further, the picture provided may not match the intuition of application scientists who typically think in terms of their computational domains. Providing scientists a complete intuitive picture maximizes their potential to effectively tune codes thereby increasing overall efficient usage of computational resources. Creating a complete picture currently requires using multiple tools and stitching together the resulting images, which is often difficult to do, especially as the data may not be readily accessible.

Recently the authors created an infrastructure within the Uintah framework that collects in situ custom performance metrics that were then visually analyzed in situ in context of the machine and application domains using the VisIt toolkit [5]. Visualizing these metrics in multiple contexts allowed application scientists, computer scientists, and system administrators to work collaboratively to better understand and solve the problems of deploying a simulation code. Further, visualizating performance data in situ decreases the possibility of exacerbating the I/O bottleneck.

In this paper, we describe how the infrastructure developed to tie the application and machine domains together for the results described above as well as expanding it to include the communication domain. We demonstrate how this approach allows one to use a single in situ tool to visualize and analyze performance metrics in the context of the machine, application, and communication domains, which we refer to as the MAC model. This approach provides a holistic view of not only performance data in the contexts of the application but also the resources where it is executing. Our approach is an extension of existing efforts in that we explicitly incorporate not only the results of analysis but also the *mechanisms* for analysis.

II. Background and Related Work

A. Performance Metrics

Profiling tools can broadly be placed into three categories, machine [4], [6]–[10] code performance(application) [1]–[3], [11], and custom profiling tools [5], [12]. Machine and code profiling is done external to the simulation by third-party tools, and in general the resulting data is written to disk and

This material was based upon work supported by the Department of Energy, National Nuclear Security Administration, under Award Number(s) DE-NA0002375. The authors thank the Uintah and VisIt development groups.

analyzed post hoc. Because the profiling is done external to the simulation it does not contain potentially useful information about the simulation. Whereas custom profiling tools are internal to the simulation and can collect not only performance metrics but also information about the simulation.

For the work presented, only custom profiling tools were utilized. However, it should be noted that third-party tools such as Vampir have an in situ interface which allows real-time application monitoring [13]. As such, it is possible to collect performance metrics from third-party sources at runtime.

B. Performance Metrics Across Multiple Domains

In general, performance visualization has focused on the domain in which the metrics were collected, (e.g. process, threads, call paths, etc.). However, some tools such as the Tuning and Analysis Utilities' (TAU) ParaProf can map performance metrics to a user-defined topological view of the underlying machine domain [3]. Efforts have also been made to link TAU's performance metrics to the application domain [14]. Both of these efforts focused on post hoc visualization. More recently, Wood demonstrated the ability to visualize performance metrics from the /proc/[pid] files of each rank in the application domain using an in situ framework [15].

Ideally, performance metrics should be seamlessly mapped to the most intuitive domains. This may be mapping to the domain in which the metrics were collected along side another domain for added context. For instance, an underutilized node in the machine domain can indicate non-optimal load balancing. Mapping that metric into the application domain can give insight as to why the node was underutilized.

Our machine, application, communication (MAC) model is most closely aligned with the work by Schulz et al. who created a hardware, application, and communication (HAC) model [16]. Their hardware domain is similar to our machine domain, consisting of the computational nodes and physical links, but was limited to the nodes being utilized by the application and had a limited core granularity whereas discussed in Section III-A our machine domain consists of the whole machine while having core level granularity. In the case of the application domain, it was limited to performance data being mapped at the grid-level whereas discussed in Section III-B our application domain has a finer granularity and is able to map application performance data to other domains. This seemingly simple distinction belies several complex and/or important performance considerations such as machine/utilization variabilities, memory subsystems/hierarchies, or inter-job interferences. While Schulz et al. at times refer to the hardware domain as the machine domain, we adopt the "Machine" designation so to be consistent with our previous work.

The full HAC model includes a fourth domain based on the MPI process. In our experience viewing performance metrics in this context adds little value when the same metrics are displayed in context of the machine domain. The exception is the rare case of oversubscribing the ranks or threads. While Schulz et al. meaningfully mapped metrics from one domain to another, it was not always possible to visualize the metrics in multiple domains using the same tool, none of which designed for in situ deployment. While indicative of the fact that visualizing performance data spans multiple fields (scientific visualization, information visualization, visual analytics, etc.), these are likely practical limitations due to the direct leveraging of specialized post hoc tools such as Boxfish [17] and MemAxes [18].

III. Machine, Application, and Communication (MAC) Model

We have focused on a three-pronged model consisting of the machine, application, communication domains, Figure 1. Though these three domains are probably the most intuitive they do not fully encapsulate all of the possible performance metric domains, but instead adequately describe the potential developmental efforts required for our single framework. That is, the metrics are typically measured and collected on a MPI process basis regardless of collection method specifics. These metrics may have different granularities such as per-node or per-thread but still have a MPI process basis.

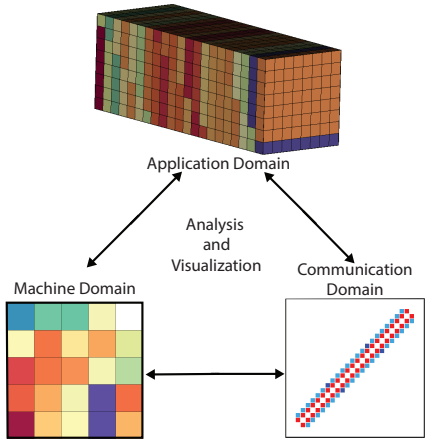


Fig. 1: The Machine - Application - Communication (MAC) model of performance data mapped to their respective domains for visualization and analysis. The simple simulation was run on 1344 patches over 24 ranks on a single node. The machine and application views are of the execution times while the communication matrix view is the MPI point-to-point messaging.

In [5] we created an extended map class to manage the collection of per-process runtime performance data. Maps are associative containers that are a combination of a key, or performance metric, and a mapped value. Our extended class maintained the core map abstraction while allowing for generic index-based access. This mapper was successfully deployed to collect performance metrics on a MPI process basis across different components running within the Uintah framework.

To extend the model and collect threaded performance metrics where a single MPI process controls multiple threads, this map class was extended to be a vector of maps to

work on a per-rank to per-thread basis. Similarly, to collect MPI Communication between ranks the base map class was extended to be a map of maps to create the process associations for a communication matrix. In addition to the extended maps for collecting performance metrics, we also utilized Uintah's Data Warehouse to store performance metrics on a per-patch basis. [1] As will be shown in Section VI this collection allows for a one-to-one mapping between the machine and application domains.

A. Machine Domain

The machine domain describes the physical layout of the computer in terms of a hierarchical view independent of the communication interconnect. For instance, the Mira supercomputer at the Argonne Leadership Computing Facility (ALCF) would be viewed in terms of cabinets (3), chassis (32), slots (16), and nodes (32). Alternately, Theta would be viewed in terms of racks (24), chassis (3), blades (16), and nodes (4). Such a hierarchy allows one to visualize the related performance metrics using a two-dimensional view.

For smaller commodity-level clusters, such as our initial target platforms, the Infiniband networking can be utilized to obtain a switch-level view which is again natural for two-dimensional display. The obvious advantage of using a two-dimensional view for visualization is the lack of occlusion. We also maintain the option to risk the introduction of occlusion in using the remaining spatial dimension for additional profiling metrics, such as cache misses. In practice, we expect frequent cases where such costs (at least for some subset of the represented data) will be minimal.

B. Application Domain

The application domain describes the system being simulated by the application code. This typically corresponds some physical space defined by a mesh along with some physical quantities but could be simply an abstract space such as a graph. For large-scale problems, the application domain is routinely decomposed into sub-domains of patches (elements), each of which is associated with a MPI process.

The collection of application performance metrics is very much dependent on the application code. At the coarsest level, metrics can be collected on a MPI process basis. In the case of Uintah's multi-threaded environment, performance metrics are collected on a per-thread basis. Similarly, we take advantage of Uintah's multi-task environment to collect performance metrics on a per-patch to per-task basis such as different task execution times which are used for dynamic load balancing[2]. We believe the ability to collect performance metrics at different granularities is critical for understanding performance issues.

[1]In Uintah, a patch is the underlying base representation for a group of individual elements.

[2]In Uintah, the fundamental unit of work is the task (ignoring data parallelism).

C. Communication Domain

The communication domain is comprised of two parts. The first represents the communication interconnect for the whole machine. The second represents the MPI communication. The interconnect is a physical attribute of the machine and therefore not only evidence of overlap among model domains but also a reasonable candidate for inclusion in the machine domain.

Communication performance is not only application specific but application *instance* specific; even subtle variations between runs such as actual application layout or interconnect traffic may have a noticeable performance impact. That is, communication domain parameters are often tied to the machine *state*, and performance data is routinely used to address the most difficult challenges in tuning codes, e.g. load balancing. Furthermore, available options to address communication performance issues are routinely interdependent and together form a large parameter space. Any typical display of MPI communication (mapped into a time domain, MPI tracing[3]), serves as both a representative example of display methods as well as of the difficulty in the subsequent utilization thereof.

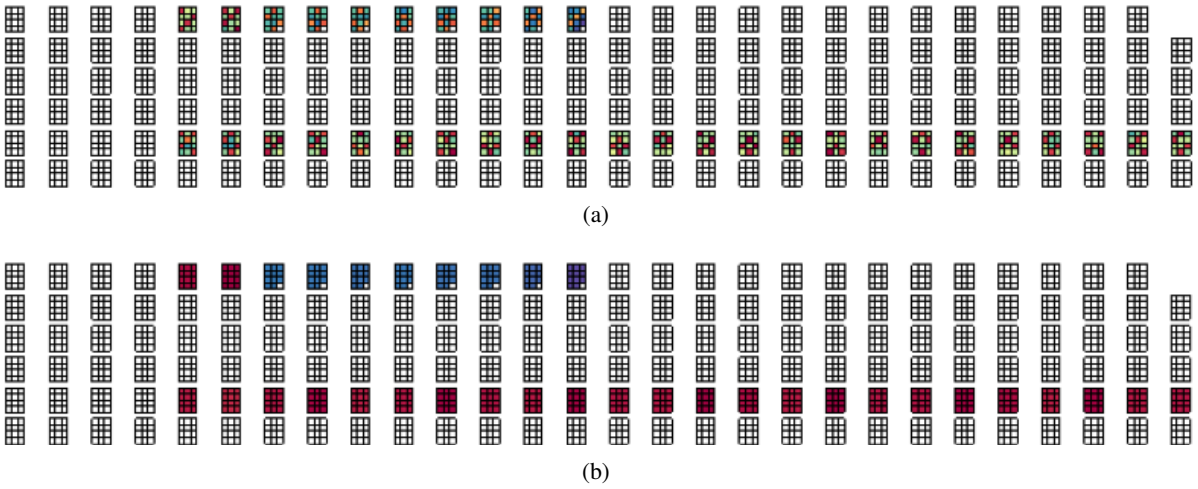
For our initial work the collection of interconnect communication performance metrics was limited to the messages passed between MPI processes using a custom profiling tool. This tool looked strictly at the data passed between MPI processes at the task level, which included the inter-task communication, and more specifically data dependencies that are relevant to the communicating tasks needed to execute (halos). These interconnect communication performance metrics were collected on per-task basis. This granularity is notable, as when trying to understand performance bottlenecks, it can often be the case that a single task is the largest contributor.

IV. Mappings

Mappings are the mechanisms by which a performance metrics are viewed outside their native domains, i.e. the domain in which the data was generated and collected. The metrics are measured and collected on a MPI process basis and are mapped according to their collection granularities. In the simplest case a mapping is one-to-one between domains. These represent the most obvious and observable relationships, such that between an application and the machine it's running on or possibly between machine-level and communication-level performance. We characterize the set of remaining mappings as belonging to one of two simple cases. Practically speaking, we believe most cases are trivially one of the three simple maps or expressible as some combination of simple maps. The three mappings of interest follow:

1:1 Mapping: This mapping is the most important as a performance metric from the domain in which it was collected can be mapped directly to another domain. An example of this mapping is a per-patch performance metric collected in the application domain that is mapped to machine domain

[3]The time domain is also an important domain for understanding the performance of multi-threaded multi-task runtime simulation codes.



(a)

(b)

Fig. 2: Per-core (a) and per-node (b) memory usage in the machine domain (red - high, blue - low). Each row represents a switch (17 total, of which, only five are shown), each block represents a processor node, and each cell within a block represents a core. The colored cores are those being utilized by the application.

where each MPI process (core) is sub-divided based on the number of patches being processed. Subdividing the domain based on the granularity of the performance metric is natural as no information is lost.

1:N Mapping: This mapping takes a performance metric from the domain in which it was collected and maps it to multiple values in another domain. The metric may be replicated or divided using a weighted function. An example of this mapping is the MPI process communication time collected in the communication domain and map to the application domain. In the case the metric would be mapped to one or more patches in the application domain.

N:1 Mapping: This mapping takes multiple performance metrics from the domain in which they were collected and via an aggregation (sum or average) or reduction (min or max), maps them to a single value in another domain. An example of this mapping is the per-patch performance metric collected in the application domain (the same data in the 1:1 Mapping) and mapping it to the machine domain where the total across all patches for each MPI process is mapped. N:1 Mappings can also be used within a domain. For instance, for a multi-threaded application one may be interested in performance metrics on a thread-level or on a node-level.

V. Visualization and Analysis

One of the primary goals of the work presented herein has been to facilitate in situ visualization and analysis of performance metrics. Traditionally the visualization and analysis step has been done post hoc which requires data to be dumped to disk adding to the I/O bottleneck and precludes the ability to make runtime performance adjustments and immediately observe their effects.

A. Middleware

To utilize our MAC model and the associated mappings in situ we created a middleware library that linked directly to the Uintah framework [5]. This library was called after the completion of each iteration, accessed the extended map classes tasked with collecting the performance data within the Uintah framework, applied the mappings for the requested domain view, and communicated the results to Visit through its in situ interface "libsim" [19].

The middleware communicated all of the necessary information needed by VisIt. That is, to VisIt, the performance data was no different than any other scientific data being visualized. This point is important because VisIt could be used natively. Furthermore, the middleware could be augmented to enable additional in situ interfaces such as ParaView's Catalyst [20] at no additional cost to the middleware's user. We point the interested reader to work by Dorier et al. [21] for relevant implementation details in connecting a visualization middleware to multiple in situ interfaces.

B. VisIt

VisIt is a scientific visualization toolkit [22] and as previously noted the visualization and analysis of performance metrics often bridges all three visualization specialties. Information visualization tools, such as DragonView developed for exploring communication metrics in the form of graphs can not be used in situ [23]. At the same time specialized performance visualization tools typically can not view data in multiple domains. For instance, Boxfish can visualize the interconnect communication yet it is not possible to explore the same metrics as a communication matrix [17].

Using a scientific visualization toolkit provides many tools for visualization and analysis that can be adapted. For instance, tools for viewing an application's multi-level grids were adapted for the viewing the application's domain decomposition based on the computational hierarchy: core, node, switch. This ability was important for exploring performance issues related to the domain decomposition. Similar hierarchies were

set up for the machine and communication domains. Other adaptations were also utilized, flat two-dimensional machine views were visualized as geometrical primitives, quads, while the communication matrix used point glyphs. Though some of these adaptations were not ideal and lacked some of the fine interface controls found within a specialized visualization tool, they did allow for in situ visualization of the performance metrics in multiple domains.

VI. Case Studies

We demonstrate some of the capabilities of our MAC model by mapping different performance metrics between the domains. For all of the examples shown, we have utilized our Uintah-VisIt coupled system. For demonstrative purposes test problems running on 1-10 nodes with 12-20 cores each are shown. Though small scale problems, they provide sufficient resolution to demonstrate the efficacy of the system.

A. Memory usage

The first case study is a simulation of a 100KW Oxy-Fuel Combustor running on a heterogeneous HPC system comprised of 253 - 12 core nodes, 164 - 20 core nodes, and 48 - 24 core nodes (7468 total cores) connected using an InfiniBand network. For this particular case 34 - 12 core nodes distributed between two networks switches were utilized. In Figure 2 all of the cores and nodes being utilized are shown in context of the whole machine. This visualization provides contextural information for understanding the application performance in the context of the whole machine. For instance, two runs yielded two different layouts, one runs well, the other does not. The root cause could be the layout.

For this particular case, Figure 2(a) shows the per core memory usage while Figure 2(b) shows the total memory usage per node where a N:1 mapping (sum) was used to get the total node usage. While the per core memory usage appears to be uniform, many of the cores on the top switch are consistency using less memory. This observation is clear in the per node view where eight of the nodes used less memory.

At the same time, this same per core/node memory usage was mapped on to individual patches in the application domain, Figure 3. Because multiple patches were assigned to each core/node a 1:N mapping was utilized. Figure 3(a) shows the per core usage as mapped on to the patches which gives little additional insight. However, when mapped on to the patches assigned to each node, Figure 3(b) it becomes apparent that a region consistently used less memory. The reason was not initially apparent until another performance metric was visualized, the task execution time (shown next), which revealed fewer solver iterations and subsequently less memory was needed.

B. Task Execution

Often one of the more critical performance metrics is the execution times, specifically, the execution time of particular tasks on a per patch basis. These metrics are used to understand many aspects of a code's performance, whether for understanding a solver or the domain decomposition. Using the same simulation as before, a 100KW Oxy-Fuel Combustor running on 100 ranks on five - 20 core nodes on a single switch, the execution time for a specific physics task was collected. In Figure 4 the times were mapped to each patch in the application domain using a 1:1 mapping. Application scientists had previously not seen such a visualization and were able to immediately confirm that the core area of the simulation would indeed take longer (more solver iterations) to execute because of the initial configuration of the combustor.

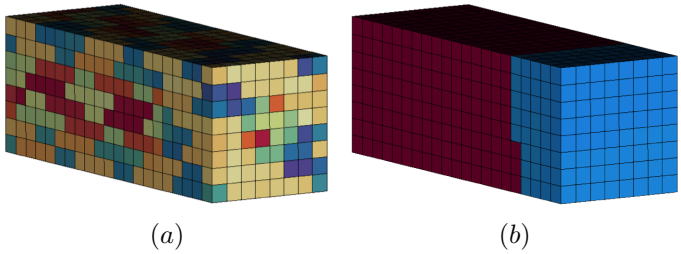


Fig. 3: Per-core (a) and per-node (b) memory usage in the application domain (red - high, blue - low).

This same data was mapped into the machine domain also using a patch level granularity. That is, a 1:1 mapping was used, which for the first time allowed each patch to be overlaid on to the rank/core to which it was assigned, Figure 5. This task's execution time dominated dynamic load balancing and had not previously been visualized resulting in a better understanding of the load balancing limitations in the context of both the machine and application domain. Specifically, the domain decomposition resulted in patches requireing fewer iterations for a solution being assigned to the same core/node which finished its work before other nodes thus resulting in a load imbalance. Had a coarser, per core granularity been utilized it would have been possible to see the imbalance but not easily understand the root cause. This fine granularity was possible because of the close coupling between the application and the collection of the custom performance metrics.

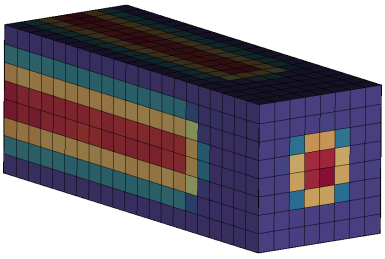


Fig. 4: An image showing the runtime performance data for the execution of a specific physics task on a per-patch basis in the application domain (Blue - fast, Red - slow).

C. Communication

Figure 6 visualizes the MPI communication in all domains of our MAC model. The simulation is from a two-dimensional

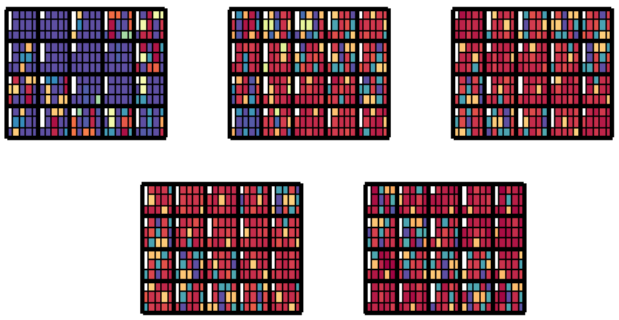


Fig. 5: An image showing same runtime performance data from Figure 4 for the execution of a specific physics task on a per-patch basis (Blue - fast, Red - slow) in the machine domain where five nodes on the same switch each with 20 cores were utilized. Mapped on to each core is the execution time for those patches assigned to that core. An observation from this image is that the faster patches have been assigned to a single node, which completes its execution before the other nodes, thus demonstrating a potential load imbalance.

shock tube with two levels of refinement with twelve MPI processes on a single node. Figure 6a shows the associated 12 by 12 communication matrix which is used to understand the communication patterns between individual MPI processes. The values shown are the total number of messages passed between two processes based on the two initialization tasks. The values are a sum for all tasks running on that process and represent a N:1 mapping. Similar to the execution times, it was also possible to look at the communication on a per task basis as it can be the case that a particular task dominates the communication. This ability is demonstrated in Figure 6b which shows the communication matrix in 3D with the contribution of each individual task shown as a stacked height field, where the first task dominated the communication.

Figure 6c shows similar information from the communication matrix Figure 6a, but for the dominant task and uses a machine view to group each value contiguously within the MPI process on each core (the blank spaces are a result of having an uneven number of communicating processes). Though a 1:1 mapping was used to create the view, the neighbor information was lost. However, one can see that the central processes had the most communicating neighbors. While perhaps not the most intuitive visualization, it demonstrates the ability to maintain a 1:1 mapping between domains.

Figure 6d takes the same information from Figure 6c and shows the total communication for all processes (as opposed to individual processes) for the dominant task in the machine domain. That is the communication from the individual processes for the dominant task was aggregated (summed), a N:1 mapping. Had multiple nodes been used, the aggregation could have been performed at the node level and ideally a hierarchical radial graph visualization would have been used. However, that would have required a specialized visualization tool and would have been outside of our goal to visualize multiple domains using an in situ setting with a single tool. For one process, the patches from the finest grid from the three AMR levels are overlaid on to the rank. These same patches are shown in Figure 6e as part of the application domain.

Figure 6e shows the aggregated communication data from Figure 6d in the application domain, which is the domain most intuitive to the application scientists. This figure demonstrates many visualization features. Similarly to Figure 6d, a N:1 mapping was used to get the total communication for the MPI porces which was then mapped to each patch on the finest AMR level assigned to that process, an 1:N mapping. The resulting visualization shows expected results, the MPI communication of the central region of the simulation (red/orange) is greater than the outer regions (yellow/green/blue), Figure 6e.

In addition, the hierarchical infrastructure was utilized to show the finest AMR level while highlighting those patches assigned to a particular MPI process, which was surprisingly not contiguous. This feature was also used to highlight the same patches in Figure 6d to give context between the different views.

D. Threading

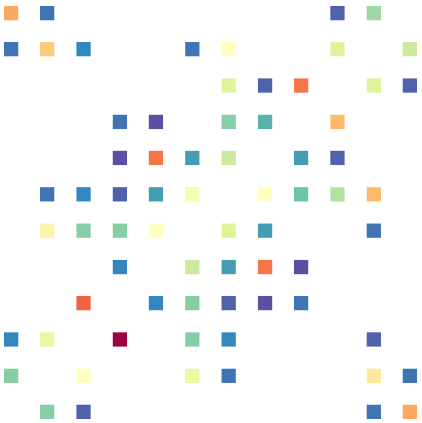
As multithreading on-node is now commonplace, efforts were made to collect and visualize their performance metrics. We consider these metrics to be a combination of the application and machine domains, that is the application controls the threading but their execution relies on the underlying machine. Our starting point was to differentiate threads within a MPI process as we currently do not yet have the necessary granularity to show the socket/core thread binding.

Figure 7 shows the machine domain view for four - 20-core nodes. In this case, the nodes, each with one MPI process controlled 12 threads. One observation is that thread #0, the master thread (upper left corner) as expected, always had fewer tasks than the other threads. It should be noted that these nodes were under utilized as the experiment was assigned to 20 core nodes.

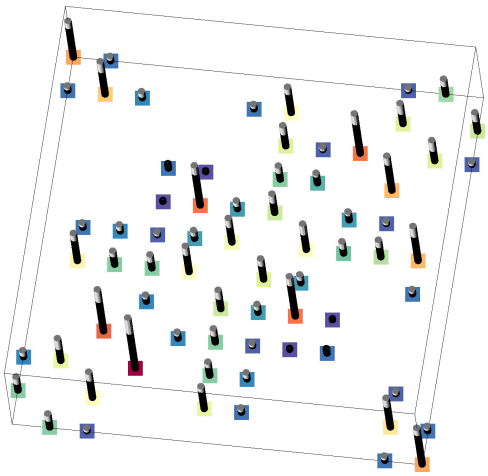
To demonstrate the flexibility of our system we assigned two MPI processes, each controlling six threads to a single node, Figure 8. In this case the job was assigned to a 24-core node (two sockets each controlling 12 cores). Though the visualization shows each of the six threads in context of the first two cores (MPI processes) that is not in fact the underlying binding. That is because the initial implementation did not disambiguate between cores (or sockets). Our furture work is to record the results of a "smp-processor-id" query at runtime and use a table look-up to obtain the corresponding socket and thread bindings.

VII. Conclusion

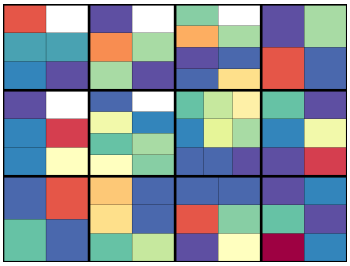
A Machine-Application-Communication model has been demonstrated for in situ visualization of performance metrics in multiple domains using a single tool. While we believe these are the most intuitive domains they are not the only domains for visualizing performance metrics. Our tight in situ coupling to the application gives the ability to map application



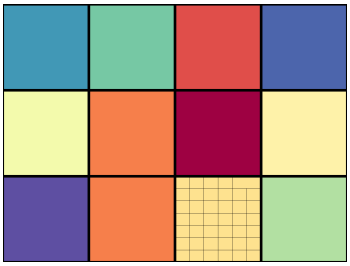
(a) Communication Domain: Pair-wise communication among the twelve cores for all tasks (two tasks total).



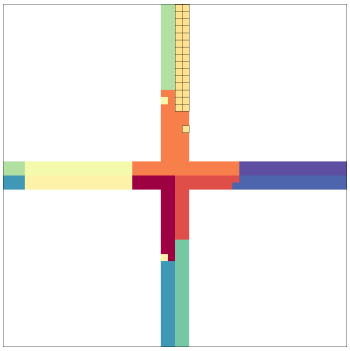
(b) Communication Domain: Pair-wise communication among the twelve cores for all tasks with a stacked height field showing the contribution from the two tasks executed.



(c) Machine Domain: Pair-wise communication among the twelve cores for the dominant task.



(d) Machine Domain: Per-core aggregate communication for the dominant task.



(e) MPI communication corresponding to the simulation's calculation of the finest resolution level for the dominant task.

Fig. 6: Example representations of an AMR simulation's MPI communication (via message counts). The traffic is colored according to the color scale (low - blue, high - red). The simulation distributed AMR patches (across three levels of refinement) on a single node of twelve cores. The same visual "patches" assign to a core comprise the visualizations in 6d and 6a; the same patches are highlighted in 6c and 6e.

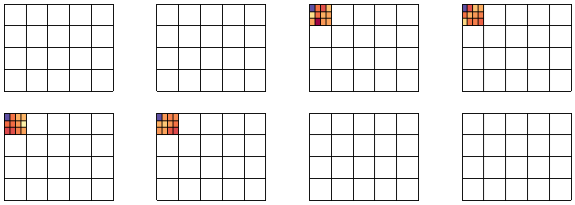


Fig. 7: Example of threading on four - 20 core nodes (the other nodes were utilized by other processes). Shown are the number of tasks assigned to each of the twelve threads controlled by a single MPI process (red - many, blue - few). The thread in the upper left corner of each core was the controlling thread with the least number of tasks.

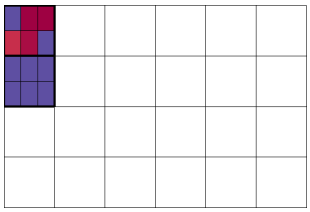


Fig. 8: Example of threading using two MPI processes on a single two socket - 24 core node. Each MPI process controlled 6 threads (under utilized). Shown is the task wait time (red - long, blue - short). The threads in the upper left corner of each process was the controlling thread with the least wait time.

specific metrics to the different domains using garularities not previously demonstrated without having to rely on reductions or aggregations.

As noted the communication domain is ambiguous because it can be seen as a subset of the machine domain. Similarly the thread performance metrics can belong to the machine or application domain. As such, one may not want to think in terms of domains but data. This shift may be important because thirdparty tools such as Vampir have an in situ interface to allow real-time application monitoring. Other tools are currently exploring similar interfaces. The ability to access different, though related data sources at runtime via the application will provide an even closer coupling between the application and performance metrics.

Regardless of whether one thinks in terms of domains or data, one of the drawbacks of using a scientific visualization tool in situ is inability to work with groups and graphs and their hierarchies. Though we have adapted tools for AMR hierarchies with success, lacking this ability greatly impacts the machine and communication domain views. Further, providing a hierarchical topological view of the underlying architecture (e.g NUMA memory nodes, sockets, shared caches, cores and multithreading) would allow for multi-variate data visualization. Despite theses drawbacks, scientific visualization tools are very well suited for in situ usage and large scale data.

Going forward there is a need for more generalized tools that can be used for visualization of performance data from multiple sources working both in situ and post hoc. Ideally such tools would be a merger of all three visualization fields, scientific visualization, information visualization, and visual analytics.

REFERENCES

[1] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. S. Müller, and W. E. Nagel, "The Vampir performance analysis toolset," in *Tools for High Performance Computing*, M. Resch, R. Keller, V. Himmler, B. Krammer, and A. Schulz, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 139–155.

[2] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent, "HPCTOOLKIT: tools for performance analysis of optimized parallel programs," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 685–701, 2010. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.1553

[3] S. S. Shende and A. D. Malony, "The TAU parallel performance system," *Int. J. High Perform. Comput. Appl.*, vol. 20, no. 2, pp. 287–311, May 2006. [Online]. Available: http://dx.doi.org/10.1177/1094342006064482

[4] J. M. Brandt, A. C. Gentile, D. J. Hale, and P. P. Pebay, "Ovis: a tool for intelligent, real-time monitoring of computational clusters," in *Proceedings 20th IEEE International Parallel Distributed Processing Symposium*, April 2006.

[5] A. Sanderson, A. Humphrey, J. Schmidt, and R. Sisneros, "Coupling the Uintah framework and the VisIt toolkit for parallel in situ data analysis and visualization and computational steering," *High Performance Computing*, June 2018.

[6] V. Ahlgren, S. Andersson, J. Brandt, N. Cardo, S. Chunduri, J. Enos, P. Fields, A. Gentile, R. Gerber, J. Greenseid *et al.*, "Cray system monitoring: Successes, requirements, priorities," *Proc. Cray Users Group*, 2018.

[7] J. Jeffers and J. Reinders, *Intel Xeon Phi coprocessor high performance programming*. Newnes, 2013.

[8] M. Kerrisk and P. Zijlstra, "Linux programmers manual," *The Linux man-pages project, version*, vol. 3, 2014.

[9] J. Dongarra, H. Jagode, S. Moore, P. Mucci, J. Ralph, D. Terpstra, and V. Weaver, "Performance application programming interface."

[10] A. Agelastos, B. Allan, J. Brandt, P. Cassella, J. Enos, J. Fullop, A. Gentile, S. Monk, N. Naksinehaboon, J. Ogden, M. Rajan, M. Showerman, J. Stevenson, N. Taerat, and T. Tucker, "The lightweight distributed metric service: A scalable infrastructure for continuous monitoring of large scale computing systems and applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 154–165. [Online]. Available: https://doi.org/10.1109/SC.2014.18

[11] M. Schulz, J. Galarowicz, D. Maghrak, W. Hachfeld, D. Montoya, and S. Cranford, "Openspeedshop: An open source infrastructure for parallel performance analysis," *Sci. Program.*, vol. 16, no. 2-3, pp. 105–121, Apr. 2008. [Online]. Available: http://dx.doi.org/10.1155/2008/713705

[12] M. Schulz, J. A. Levine, P. Bremer, T. Gamblin, and V. Pascucci, "Interpreting performance data across intuitive domains," in *2011 International Conference on Parallel Processing*, Sep 2011, pp. 206–215.

[13] M. Weber, J. Ziegenbalg, and B. Wesarg, "Online performance analysis with the Vampir tool set," in *Tools for High Performance Computing 2017*, September 2017.

[14] K. A. Huck, K. Potter, D. W. Jacobsen, H. Childs, and A. D. Malony, "Linking performance data into scientific visualization tools," in *2014 First Workshop on Visual Performance Analysis*, Nov 2014, pp. 50–57.

[15] C. Wood, M. Larsen, A. Gimenez, C. Harrison, T. Gamblin, and A. Malony, "Projecting performance data over simulation geometry using sosflow and alpine," in *2017 Forth Workshop on Visual Performance Analysis*, Nov 2017, pp. 1–8.

[16] M. Schulz, A. Bhatele, D. Böhme, P. Bremer, T. Gamblin, A. Gimenez, and K. Isaacs, "A flexible data model to support multi-domain performance analysis," in *Tools for High Performance Computing 2014*, Oct 2014, pp. 206–215.

[17] K. E. Isaacs, A. G. Landge, T. Gamblin, P.-T. Bremer, V. Pascucci, and B. Hamann, "Exploring performance data with Boxfish," in *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*. IEEE, 2012, pp. 1380–1381.

[18] A. Gimenez, T. Gamblin, I. Jusufi, A. Bhatele, M. Schulz, P.-T. Bremer, and B. Hamann, "Memaxes: Visualization and analytics for characterizing complex memory performance behaviors," in *Transactions On Visualization And Computer Graphics*. IEEE, May 2016, pp. 1–13.

[19] B. Whitlock, J. M. Favre, and J. S. Meredith, "Parallel in situ coupling of simulation with a fully featured visualization system," in *Proceedings of the 11th Eurographics Conference on Parallel Graphics and Visualization*, ser. EGPGV '11. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2011, pp. 101–109. [Online]. Available: http://dx.doi.org/10.2312/EGPGV/EGPGV11/101-109

[20] U. Ayachit, A. Bauer, B. Geveci, P. O'Leary, K. Moreland, N. Fabian, and J. Mauldin, "Paraview catalyst: Enabling in situ data analysis and visualization," in *Proceedings of the First Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization*, ser. ISAV2015. New York, NY, USA: ACM, 2015, pp. 25–29. [Online]. Available: http://doi.acm.org/10.1145/2828612.2828624

[21] M. Dorier, R. Sisneros, T. Peterka, G. Antoniu, and D. Semeraro, "Damaris/viz: a nonintrusive, adaptable and user-friendly in situ visualization framework," in *Large-Scale Data Analysis and Visualization (LDAV), 2013 IEEE Symposium on*. IEEE, 2013, pp. 67–75.

[22] H. Childs, E. Brugger, B. Whitlock, J. Meredith, S. Ahern, D. Pugmire, K. Biagas, M. Miller, C. Harrison, G. H. Weber, H. Krishnan, T. Fogal, A. Sanderson, C. Garth, E. W. Bethel, D. Camp, O. Rübel, M. Durant, J. M. Favre, and P. Navrátil, "VisIt: An End-User Tool For Visualizing and Analyzing Very Large Data," in *High Performance Visualization–Enabling Extreme-Scale Scientific Insight*. CRC Press, Oct 2012, pp. 357–372.

[23] A. Bhatele, N. Jain, Y. Livnat, V. Pascucci, and T. Bremer, "Analyzing network health and congestion in dragonfly-based supercomputers," *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 93–102, 2016.