Continuous Performance Monitoring for Large-Scale Parallel Applications

Isaac Dooley
Department of Computer Science
University of Illinois
Urbana, IL 61801
Email: idooley2@illinois.edu

Chee Wai Lee
Department of Computer Science
University of Illinois
Urbana, IL 61801
Email: cheelee@uiuc.edu

Laxmikant V. Kale
Department of Computer Science
University of Illinois
Urbana, IL 61801
Email: kale@uiuc.edu

***Abstract*—Traditional performance analysis techniques are performed after a parallel program has completed. In this paper, we describe an online method for continuously monitoring the performance of a parallel program, specifically the fraction of the time spent in various activities as the program executes. Our implementation of both a visualization client and the parallel performance framework that gathers utilization data are described. The data gathering uses a scalable and asynchronous reduction with an appropriate lossless compressed data format. The overheads in the initial system are low, even when run on thousands of processors. The data gathering occurs in an out-of-band communication mechanism, interleaving itself transparently with the execution of the parallel application by leveraging a message-driven runtime system.**

I. Continuous Performance Monitoring

A. Importance of Continuous Performance Monitoring

Postmortem performance monitoring is the norm in parallel computing. In this common performance analysis approach, a parallel program is run. During the run, or at its completion, performance data is written to files which can then be used by analysis tools. This common approach is used primarily due to the lack of availability of continuous performance monitoring tools. Continuous performance monitoring is an approach where performance characteristics of a running parallel application are used as the program runs by a performance analyst. This approach has several benefits when used either by itself on in conjunction with postmortem analysis tools. Continuous performance monitoring, although not yet widely used, could become more popular if it were technically feasible, and if it could be deployed without significantly degrading the performance of the running application. Some of its benefits are discussed below.

One benefit of continuous performance monitoring is that the power of a parallel computer can be applied towards portions of the analysis as performance data is produced. Portions of the analysis can be quickly executed while the performance data is still in memory. One such example of this beneficial scenario would be a monitoring tool that can add up profile information across all processors using reduction operations on the parallel computer itself. In contrast, a postmortem tool would have to spend considerable time processing potentially large per-processor trace files on disk to generate the same information.

For a long running job, a continuous performance monitoring tool can be used to look for unacceptable performance degradation. In some applications, without a built in adaptive runtime system, the evolution of the application may cause its performance to degrade due to gradual or abrupt factors such as adaptive refinement, change in material properties (for finite element structural simulations), etc. If the user can check the performance characteristics from time to time as the program runs, they will be able to terminate the job when such a scenario is reached. Typically, they can restart the simulation from the last saved checkpoint (with a new modified mesh partitioning, for example).

A continuous performance monitoring tool is a prerequisite for online performance steering: visual performance feedback, and specific details emerging from it, may be used by the programmer to tell the application to adjust some parameters or to trigger a runtime load balancing phase, etc.

For long running jobs, a long performance trace or profile, obtained at the end of the run, will take time to transfer all at once at the end via the file systems. A pre-processed efficiently compressed performance stream, coming continuously from the parallel computer throughout the run, could be stored during the program's run, possibly reducing the time spent at the end of the job.

Furthermore, a continuous performance monitoring tool could be used to quickly alert a programmer to an error state in the program. For example when a program hangs, the utilization displayed in a continuous monitoring tool will change. A continuous performance monitoring tool could also help the programmer detect other program specific anomalies. The continuous performance analysis tool could be standalone or integrated into existing or future debugging tool.

B. Types of Existing Performance Analysis Tools

Postmortem performance systems store raw performance data into the memory buffers of a parallel machine while an application executes. The performance data is written out to the filesystem when buffers fill or when the application terminates. The performance logs can then be post-processed, manipulated and displayed by standalone performance tools. The performance log formats generated can be broadly categorized as detailed event logs or profile logs.

Detailed event logs store the faithful, usually chronological, time-stamped recording of performance metrics per pre-defined event (eg. function call, message sends) encountered in the application. These logs tend to be extremely large and care has to be taken to control the generated data volume for effective postmortem analysis. Vampir [1], Jumpshot [2], Paradyn [3], KOJAK [4], Pablo [5] and Projections [6] are examples of tools and systems that can use various forms of detailed event logs.

Profile logs capture some summary of performance metrics, over an application's lifetime. Profiles could be generated based on statistical sampling like the gprof tool [7] or based on direct measurement of performance metrics of pre-defined events, as is done in Projections [6] summaries or TAU profiles [8] using the Paraprof tool. Profile-based tools generally do not capture dynamic performance changes to the application over time. It is possible to take "snapshots" [9] of profiles, allowing the capture of performance changes over time, but at the cost of storing an additional profile for each snapshot in the buffers.

Tools that monitor application performance at runtime include Autopilot [10] and TAU, through a series of development projects leading to *TAUg* and *TAUoverSupermon* [11], [12], [13], [14]. Autopilot attempted to tune the performance of a long-running application through the use of "sensors" and "actuators" to allow a remote client to adjust the performance properties of the application. Its approach is targeted at grid applications whose performance would change unpredictably over a long period of time, requiring the intervention of an external agent. Our focus is, instead, on acquiring as much detailed application performance information as possible in a long running application. Our approach requires techniques for effective low-overhead data collection in the parallel runtime. *TAUg* implements an abstraction of a global view of performance data and introduced an interface for MPI programmers to gather and act on the data. This differs from our approach, which exploits the natural ability of the runtime system to adaptively schedule and interleave the data collection collective operations with application work. Our approach requires no changes to user code. *TAUg* has shown low overheads up to 512 processors. *TAUoverSupermon* takes a different strategy from *TAUg*, using the Supermon cluster monitoring framework [15] as the data aggregation mechanism (referred to in their paper as the "transport"). It too, shows low overhead up to 512 processors.

C. Message Driven Execution Facilitates Continuous Performance Monitoring

To implement the continuous performance monitoring tool described in this paper, we used the Charm++ runtime system. The Charm++ runtime system allows programs to be written in multiple parallel languages including Charm++ [16], SDAG [17], Charisma [18], Multiphase Shared Arrays [19], and AMPI [20]. Different portions of a program can even be written using different languages [21]. In all of the languages supported by the Charm++ runtime system, the execution of the program proceeds primarily through the execution of methods or threads when an arriving message is processed. Each processor contains a scheduler and a queue of messages awaiting execution. The scheduler executes the task specified by each message. Because different messages can result in the execution of different tasks, multiple program modules can interleave their execution and send messages using the same underlying runtime system.

Thus a runtime system such as Charm++ makes it easy for a performance monitoring module to send messages that are independent of the messages being sent by the rest of the parallel program. Such out-of-band communication is useful in implementing continuous performance monitoring tools. We describe an example of such a tool in section II.

II. Utilization Profile Tool

Of the many possible types of continuous performance monitoring that could be implemented, this paper describes a new tool that efficiently streams utilization profiles from a parallel program to a visualization client. The new tool is comprised both of the first generic continuous performance analysis system built into the Charm++ parallel runtime system, and a corresponding visualization client. The new tool works by gathering utilization statistics about a running parallel program, efficiently compresses this utilization data, and merges the compressed data in a reduction from all the processors. A visualization client, run on a workstation, can connect to the running parallel program over a potentially low bandwidth network and retrieve continuously updating utilization profiles from the parallel program. Because the tool is implemented as a Charm++ module, any Charm++ program can be used with the tool without any code modifications.

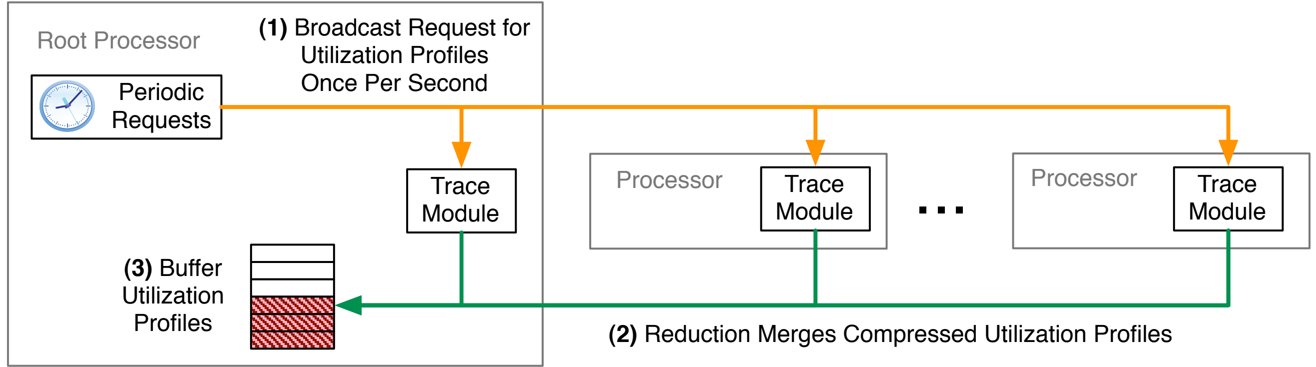
The utilization profile data gathered in this new system describes the fraction of the execution time spent in each activity over some period of time. The utilization profiles produced by the tool contain a number of bins, each representing 1ms slices of the execution of the program. Some fine-grained information about the execution of the activities will be lost, but the resolution of 1ms should be suitable for a variety of analysis tasks, including visualization. The activities themselves, as instrumented by our tool, are *Charm++ Entry Methods*. An entry method is a method that is can be remotely called via an asynchronous method invocation on a migratable Chare object.

Figure 1 shows an overview of the architecture of the utilization profile tool. It shows the two independent mechanism that comprise the utilization profile tool, a performance statistic gathering portion in the parallel runtime system, and a mechanism for the visualization client to retrieve the statistics produced by the first mechanism.

A. Observing Utilization On Each Processor

On the parallel system, each processor creates and updates a utilization profile containing the amount of time spent in each entry method. To observe and record the utilization for each of the entry methods, we created a trace module enabled with

A) Gathering Performance Data in Parallel Runtime System:



B) Visualizing Performance Data:

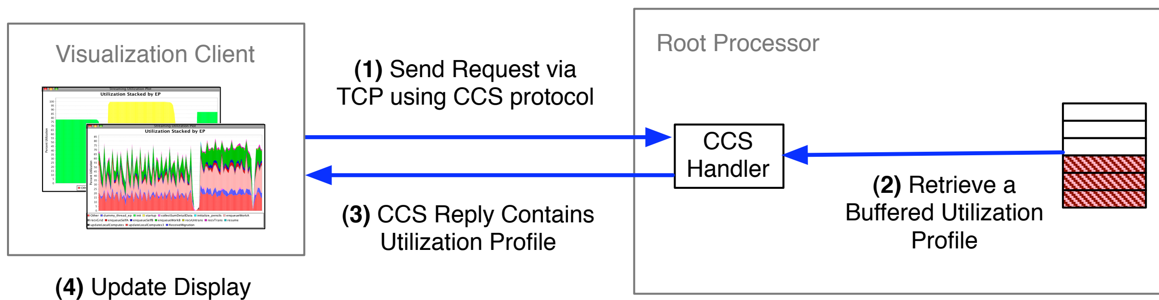


Fig. 1. An overview of the Utilization Profile Tool. The tool is comprised of two separate mechanisms. The first mechanism (A) periodicially gathers performance data in the parallel runtime system. The second mechanism (B) allows a visualization client to retrieve the previously buffered performance data from the parallel program.

a single link time option `-tracemode utilization`. This module is responsible for accumulating the time spent executing each entry method into the appropriate utilization profile bins. Each bin, representing 1ms of walltime, contains a double precision floating-point value for each of the entry method. Hooks in the trace module are called after each entry method completes its execution, at which point the execution time is accumulated into the one or more bins spanning the execution of the entry method.

In memory, the bins are allocated as a fixed-size contiguous array which is treated as a circular buffer. We arbitrarily chose to use a circular buffer with $2^{15} = 32768$ bins which spans about 33 seconds. The circular buffer is not compressed, and hence its size can be large. In typical Charm++ programs there are hundreds of different entry methods, many of which are never executed, and many that are only called at startup. In a run of the NAMD application, with our trace module enabled, there are 371 distinct entry method. The size of the circular buffer allocated on each processor for this program would therefore be about $32768 \text{ bins} \times \frac{371 \text{ entry methods}}{\text{bin}} \times \frac{8 \text{ bytes}}{\text{entry method}} \approx 93 \text{ MB}$. Although this buffer is somewhat large, the sparse data it contains is compressed before being merged across all the processors. Section III shows that the actual cost of this approach is low when used with the NAMD application.

If memory constraints are critical for a program, then the circular buffer could be reduced by a number techniques. The number of bins could be reduced, the bins could cover larger amounts of execution time, and the set of entry methods could be reduced either by reducing the set of entry methods registered by the programs, or by reordering the entry methods and only recording information for the most important ones. A further way to shrink the memory requirement for this buffer would be to use single precision floating-point values instead of the 8-byte double precision values used currently.

B. *Compressing Utilization Profiles*

Although each processor gathers its own utilization statistics in its trace module, the tool described in this paper reports the overall utilization across all processors. In order to efficiently combine the data from all processors, it is important to use an efficient data storage format for the communication intensive part of the data merging and transmission.

We created a compressed utilization profile format to use when transmitting the usage profiles between processors and to the visualization client. This format, which is shown in figure 2, represents the utilization for many timeslice bins for one or more processors. The compressed format has a length that depends upon the number of bins and the number of entry

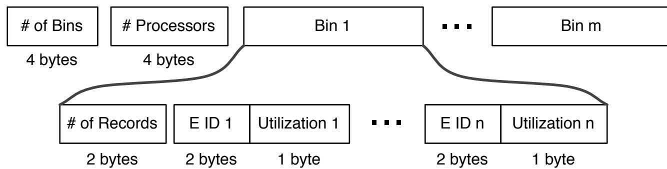


Fig. 2. Compressed utilization profile format.

methods active during the timeslice represented by each of the bins. Within each bin, the utilization for each active entry method is stored using 3 bytes. One byte stores the utilization in a range from 0 to 250, a resolution that is reasonable for onscreen displays. Two bytes store the entry method index (E ID). The records within each bin are in sorted order by the entry method index. As mentioned in section II-A, typical programs have more than 256 entry methods, and hence more than one byte is required to represent them all.

The compressed format has a header which contains 4 bytes specifying the number of bins and it 4 bytes specifying how many processors produced the utilization profile.

Because there will be some entry methods that contribute only tiny amounts to a bin, we decided to compress all such entry methods into a single *other* category which is stored just as any of the other entry methods in the bin but with a reserved entry method index. The entry method stored in this *other* category are any that fail to contribute more than a specified threshold to a bin. The threshold used for the results shown in this paper is 10%. This merging of entry methods that contribute little to the result can reduce the sizes of the compressed utilization profiles.

C. Merging Utilization Profiles from Many Processors

Periodically the compressed utilization profiles from all processors are merged together and stored on processor zero from which the visualization client can retrieve them. At startup, processor zero will instruct the runtime system to call a specified function once per second. This function in turn will broadcast a request to all processors. Each processor, upon receiving such a request, will compress a utilization profile of 1000 bins, and contribute the compressed profile to a reduction. The reduction is a standard Charm++ reduction with a custom reduction operation that merges any number of compressed utilization profiles into a single new compressed utilization profile. The reduction proceeds over a spanning tree, so it is scalable to large numbers of processors. The result of the reduction arrives at processor zero, at which point it is stored in a queue awaiting a request from a visualization client. The incoming reduction message containing the compressed utilization profile is itself saved, so no copying of the usage profile is required at the end of the reduction.

The custom reduction merging operation simply marches through all of the incoming compressed utilization profiles bin by bin, appending resulting merged bins to a new compressed utilization profile. Merging bins is simple because the entry method indices are already in a sorted order, so the minimal entry method index from all the incoming bins is selected and the utilization from any of the incoming bins for that entry method index are averaged. This average is weighted by the number of processors that have contributed to each of the respective incoming bins. The weighted average is important if the reduction tree is not a simple k-ary balanced tree.

Because Charm++ uses an asynchronous message driven execution model, the broadcast, the compression of the utilization profiles, and the following reduction will interleave with the other work occurring in the program. This interleaving could potentially produce adverse performance problems for the running parallel program, but section III shows that the overhead is low for one widely-used example application.

D. Sending Utilization Profiles to a Visualization Client

The communication of the utilization profiles from the running parallel program to the visualization client is performed through the Converse Client-Server (CCS) system [22]. CCS provides a server-side library, and client libraries in multiple languages. The underlying communication mechanism is TCP, so it will work over all common types of networks.

To run the parallel program, a command line option is used to specify the TCP port to be used for the CCS server-side. In our implementation, the parallel program will simply reply to any incoming CCS request of the appropriate type with the oldest stored utilization profile in the queue on processor 0. If no stored utilization profile is available, then an empty message is sent back to the client.

The client will periodically send requests to the running program. Once a compressed utilization profiles is returned, the visualization windows of the client are updated. For convenience, the visualization client also supports the ability for the profiles to be written to a file for future analysis and the ability to play back the profiles saved in a file.

E. Visualization Client

The visualization client created to work with the new Charm++ trace module is implemented in Java, resulting in a portable GUI based tool. The tool provides a both a graphical interface to connect to a running parallel program and some resulting graphical visualizations described below.

The first main display in the visualization client contains a scrolling stacked utilization plot of coarse grained averaged utilizations. This display shows 10 seconds worth of data at a time, scrolling as new data is added. The plot is composed of 100 bars in the x-dimension. Each bar displays the average of 100 bins, or 100ms of execution time. This view provides a high level view of the overall utilization of the program as it runs. A second more detailed view displays data at the finer grain 1ms resolution.

The second display in the visualization client contains a higher resolution plot of the utilization. This view does not scroll as new data arrives, but rather is replaced periodically with a recent view representing 250ms of execution time. Figures 6 and 7 show snapshots of two such views from a

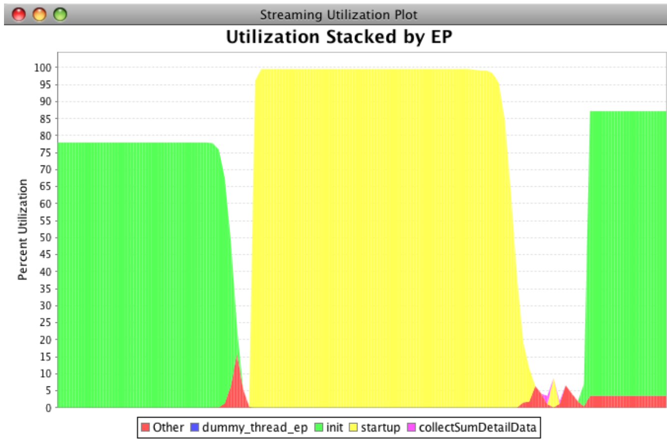


Fig. 3. A screenshot of the streaming view in our tool. This view represents 10 seconds of execution during startup for NAMD on 1024 processors running the STMV molecular system.

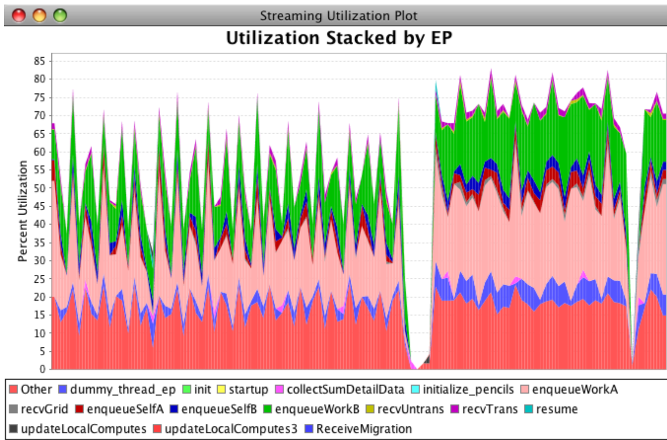


Fig. 4. This streaming view represents 10 seconds of execution during the early steps when load balancing takes place. Two valleys, corresponding to the two load balancing steps, are clearly seen. This screenshot comes later in the execution of the same program run as in figure 3

single run of the NAMD application on 1024 processors. These views are useful when more detailed fine-grain information is required. For example, the plots in figures 3, 4, and 5 show only a high level utilization breakdown while figures 6 and 7 show how each step in the simulation is progressing. Such a detailed view is necessary to identify certain types of problems, such as those that affect the behavior at a sub-timestep basis.

One final display in the initial visualization client shows the sizes of the incoming compressed utilization profiles. This view is useful mostly just when determining the overhead of our system. Figure 8 contains a plot that was generated by the visualization client.

The client uses the CCS Java client-side library to create a `CcsThread` which is used to send requests to the parallel program. When the visualization client receives a CCS reply, it saves the message, and computes the data for the scrolling display. For this display, the values in 100 bins are averaged

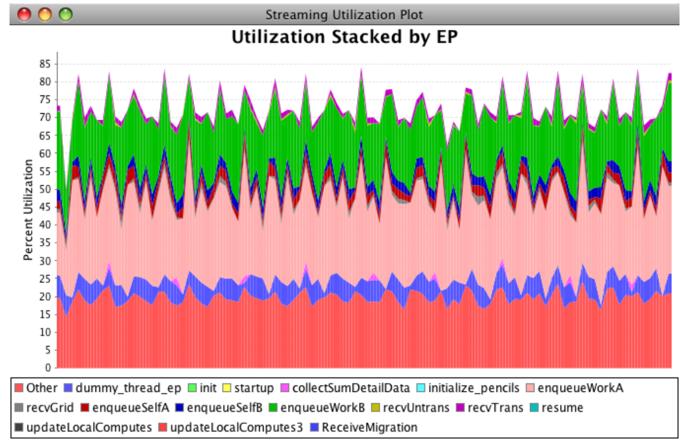


Fig. 5. This streaming view represents 10 seconds of execution during the later simulation steps once a more uniform load balance has been achieved. Each bar represents the average utilization for 100ms of execution time. For this program, the timesteps are shorter than this duration and hence the utilization doesn't reach 100% in this plot. This screenshot comes later in the execution of the same program run as in figures 3 and 4.

together and the plot data is updated. Periodically, the beginning 250ms of the most recent CCS reply is plotted directly in the high resolution display.

The names corresponding to each of the entry method indices is recorded by the trace module into a file. The user of the visualization client can point the visualization client to an appropriate file containing these names, and the legends for the plots will use the names instead of the entry method indices. In the future, the names will be sent through the CCS connection when requested by the client.

III. Performance Impact

To determine the actual performance impact of our performance metric gathering scheme, we ran the parallel NAMD molecular dynamics application [23] on 512 up to 8192 processors of the Cray XT5 system Kraken at the National Institute for Computational Sciences managed by the University of Tennessee. We compared the application performance of a baseline version of NAMD containing no tracing modules to a version using the tracing module that gathers utilization profiles as described in section II. We ran the baseline program once, and on the same allocation of processors we ran the version that gathers utilization profiles twice, once with a visualization client connected, and once without any visualization client connect. The NAMD program ran the Satellite Tobacco Mosaic Virus (STMV) example simulation [24]. It is a large molecular system that is useful for demonstrating scaling to thousands of processors. Timings were analyzed for 400 application timesteps well after the initial load balancing steps.

Our results show that for up to 8192 processors, the overhead of recording the utilization profiles and sending them to a visualization client is at most 1.14%. The results appear to contain a slight amount of noise in that sometimes baseline version is slower than the version that gathers performance

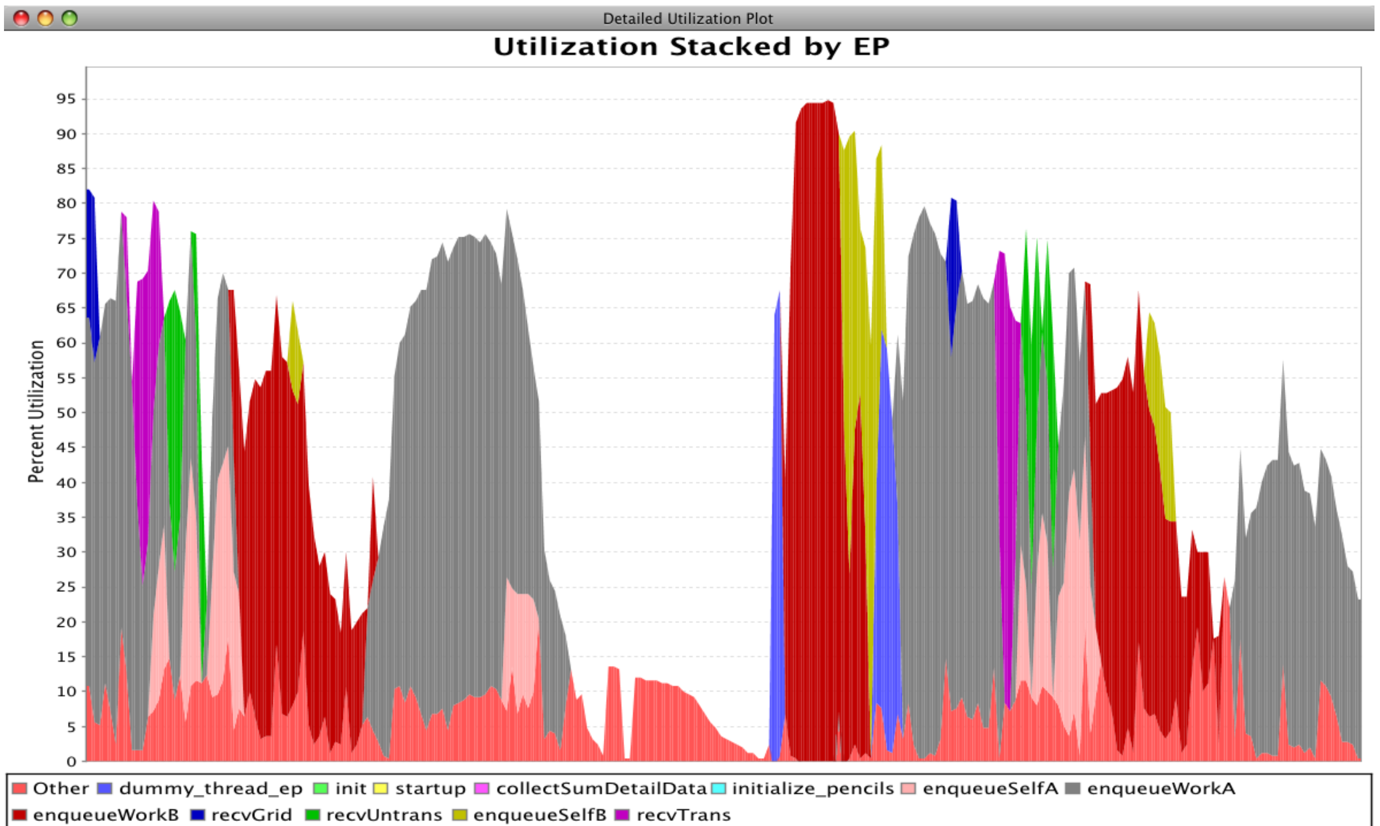


Fig. 6. This detailed view plots the full 1ms resolution utilization profile for 0.25 seconds of execution time for a NAMD STMV run on 1024 processors. This snapshot captures steps with poor load balance.

Processors	512	1024	2048	4096	8192
Overhead Without Visualization Client	0.94%	0.17%	-0.26%	0.16%	0.83%
Overhead With Visualization Client	0.58%	-0.17%	0.37%	1.14%	0.99%

TABLE I
OVERHEAD OF COLLECTING UTILIZATION PROFILE INSTRUMENTATION

stastics. These variations are likely caused by intereference from other parallel jobs sharing the same interconnect and causing contention in the network.

We examined some projections trace logs for a 1024 processor run of NAMD to determine the cost of creating the compressed utilization profiles. The time to create the compressed utilization profile from the uncompressed circular buffer was 9.3 ms. Because the uncompressed buffer is created once per second, we expect the overhead of our approach to be around $\frac{9.3ms}{1s} = 0.93\%$. The cost of the reductions was almost nonexistent in comparison to the compressing of the utilization profiles. The estimated 0.93% overhead seems to correspond well with the results actually obtained, modulo the noise.

The messages sent along the reduction tree when combining the utilization profiles for all processors have sizes that range from 3.5KB up to almost 12KB. Figure 8 shows a plot of the resulting utilization profiles that were received by the visualization client when running NAMD on 1024 processors. This figure shows that the sizes of the messages vary throughout different phases of the application. During startup, the computation involves few entry methods, and hence the message sizes are smaller. When the fine-grained timesteps are executing toward the end of the program, the utilization profile combined from all processors is approximately 12KB in size.

IV. Future Work

The capabilities enabled by this work presents an opportunity for a rich set of features to be deployed by a variety of remote clients, visualization or monitoring tools. Visualizing the time-varying profile of a complex application like NAMD as demonstrated in this paper is just one such feature.

For example, a sorting filter could be applied to the data stream, allowing a user to see in real-time the most significant processor-outliers based on the amount of time they spent being idle. Real time application load balance could also be displayed using the data stream in the form of bar chart of a fixed number of processors demonstrating the highest and

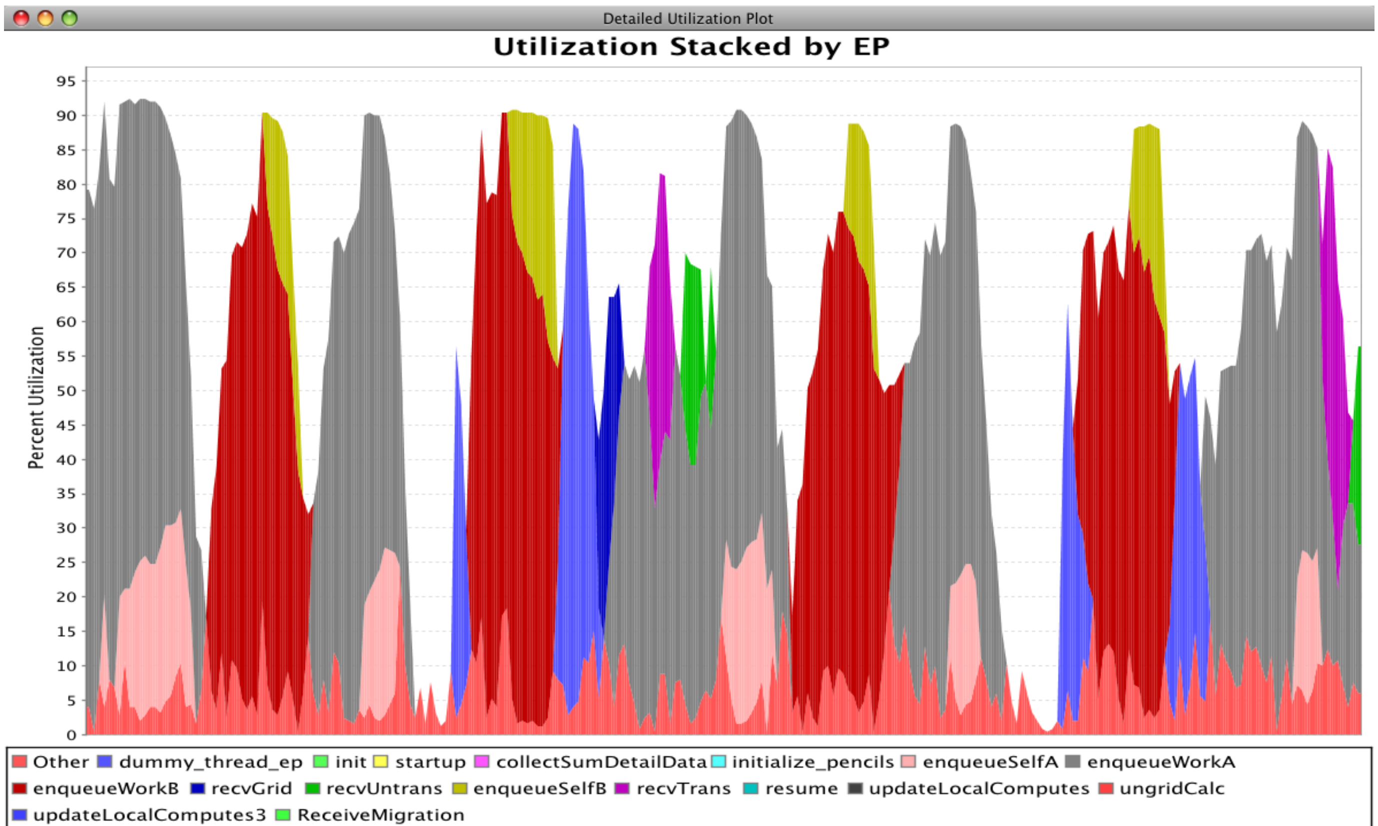


Fig. 7. This detailed view shows a later plot from the same run shown in figure 6, after load balancing improves the performance by shortening the time per step.

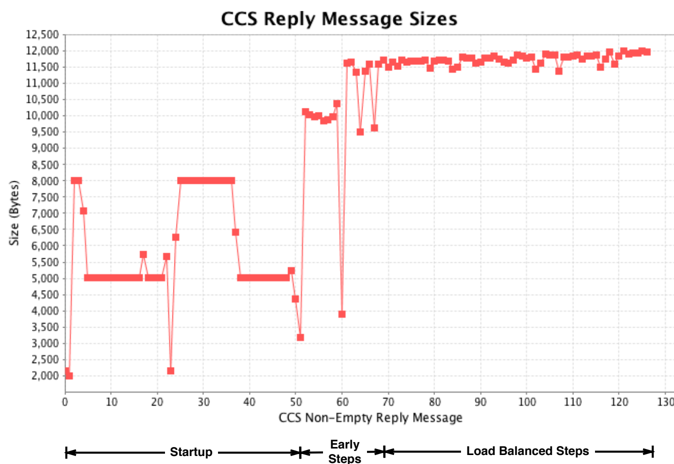


Fig. 8. Sizes of the compressed utilization profiles received by the visualization client. The total bandwidth required to stream the data is thus under 12KB/second for this program, namely NAMD simulating the STMV system on 1024 processors.

lowest loads. Average, maximum and minimum loads can be displayed as animated lines on that chart.

An analyst observing a degradation of performance in a long-running application could choose to send a "termination" signal through the CCS connection. While the traditional purpose of such a signal is to prevent it from consuming further computational cycles inefficiently, we can take this one step further. The "termination" signal could tell the application to terminate after the next application-supported checkpoint. At the same time, the performance framework could respond to the signal by changing from supplying low-overhead data streams to an external client to recording a detailed event trace. The detailed event trace can then be used postmortem to understand the reasons for performance degradation, allowing the analyst to re-tune the application and restart it from the saved checkpoint.

In the future we hope to expand the type of continuously monitored information to incorporate more fine-grained detail. Towards this goal, we would like to be able to selectively gather full trace data for subsets of processors.

Any type of performance data which is represented as scalar values over time could be collected and displayed in an identical manner to the utilization data described in this paper. Such scalar values which are likely to be interesting include: memory usage, CPU performance counters, and network utilization.

V. Conclusion

This paper describes an method for continuously monitoring the performance of parallel programs. Our example tracing

framework uses the Charm++ runtime system. Utilization profiles are generated transparently to the application on each processor and combined across processors in a reduction using a compressed format. We created a new visualization tool that displays both high resolution images of 250ms slices of execution and a lower resolution scrolling view that shows the utilization profile for the previous 10 seconds.

The overheads of gathering the performance measurements are low, around 1%, even when running on thousands of processors. The overheads for the performance measurement collection both with and without a visualization client are shown for runs on 512 up to 8192 processors. The required network bandwidth from the parallel system to the visualization client in our implementation is only 12 KB/second.

ACKNOWLEDGMENT

The authors would like to thank the Department of Energy for its support of this project through the DOE HPCS Fellowship program. This work is also supported in part by grants from the National Institute of Health P41-RR05969. NAMD was developed by the Theoretical and Computational Biophysics Group in the Beckman Institute for Advanced Science and Technology at the University of Illinois at Urbana-Champaign.

REFERENCES

[1] W. E. Nagel, A. Arnold, M. Weber, H. C. Hoppe, and K. Solchenbach, "VAMPIR: Visualization and analysis of MPI resources," *Supercomputer*, vol. 12, no. 1, pp. 69–80, 1996. [Online]. Available: citeseer.ist.psu.edu/article/nagel96vampir.html

[2] O. Zaki, E. Lusk, W. Gropp, and D. Swider, "Toward scalable performance visualization with Jumpshot," *The International Journal of High Performance Computing Applications*, vol. 13, no. 3, pp. 277–288, Fall 1999. [Online]. Available: citeseer.ist.psu.edu/zaki99toward.html

[3] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall, "The paradyn parallel performance measurement tool," *Computer*, vol. 28, no. 11, pp. 37–46, 1995.

[4] F. Wolf and B. Mohr, "KOJAK - A Tool Set for Automatic Performance Analysis of Parallel Applications," in *Proc. of the European Conference on Parallel Computing (Euro-Par)*, ser. Lecture Notes in Computer Science, vol. 2790. Klagenfurt, Austria: Springer, August 2003, pp. 1301–1304, demonstrations of Parallel and Distributed Computing.

[5] D. A. Reed, R. A. Aydt, R. J. Noe, P. C. Roth, K. A. Shields, B. W. Schwartz, and L. F. Tavera, "Scalable performance analysis : The pablo performance analysis environment," in *Proceedings of the Scalable Parallel Libraries Conference*. IEEE Computer Society, 1993, pp. 104–113.

[6] L. V. Kale, G. Zheng, C. W. Lee, and S. Kumar, "Scaling applications to massively parallel machines using projections performance analysis tool," in *Future Generation Computer Systems Special Issue on: Large-Scale System Performance Modeling and Analysis*, vol. 22, no. 3, February 2006, pp. 347–358.

[7] S. L. . Graham, P. B. Kessler, and M. K. McKusick, "GPROF: a call graph execution profiler," *SIGPLAN 1982 Symposium on Compiler Construction*, pp. 120–126, Jun. 1982.

[8] S. Shende and A. D. Malony, "The TAU Parallel Performance System," *International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 287–331, Summer 2006.

[9] A. Morris, W. Spear, A. D. Malony, and S. Shende, "Observing Performance Dynamics Using Parallel Profile Snapshots," *Lecture Notes in Computer Science*, vol. 5168, pp. 162–171, August 2008.

[10] R. L. Ribler, J. S. Vetter, H. Simitci, and D. A. Reed, "Autopilot: Adaptive Control of Distributed Applications," in *Proc. 7th IEEE Symp. on High Performance Distributed Computing*, Chicago, IL, Jul. 1998.

[11] T. J. Sheehan, A. D. Malony, and S. S. Shende, "A Runtime Monitoring Framework for the TAU Profiling System," *Lecture Notes in Computer Science*, vol. 1732, pp. 170–181, December 1999.

[12] A. D. Malony, S. Shende, and R. Bell, "Online Performance Observation of Large-Scale Parallel Applications," in *Proc. Parco 2003 Symposium, Elsevier B.V*, vol. 13, 2004, pp. 761–768.

[13] K. A. Huck, A. D. Malony, S. Shende, and A. Morris, "TAUg: Runtime Global Performance Data Access Using MPI," *Lecture Notes in Computer Science*, vol. 4192, pp. 313–321, September 2006.

[14] A. Nataraj, M. Sottile, A. Morris, A. D. Malony, and S. Shende, "TAUoverSupermon: Low-Overhead Online Parallel Performance Monitoring," *Lecture Notes in Computer Science*, vol. 4641, pp. 85–96, August 2007.

[15] M. J. Sottile and R. G. Minnich, "Supermon: a high-speed cluster monitoring system," 2002, pp. 39–46.

[16] L. V. Kale, E. Bohm, C. L. Mendes, T. Wilmarth, and G. Zheng, "Programming Petascale Applications with Charm++ and AMPI," in *Petascale Computing: Algorithms and Applications*, D. Bader, Ed. Chapman & Hall / CRC Press, 2008, pp. 421–441.

[17] L. V. Kale and M. Bhandarkar, "Structured Dagger: A Coordination Language for Message-Driven Programming," in *Proceedings of Second International Euro-Par Conference*, ser. Lecture Notes in Computer Science, vol. 1123-1124, September 1996, pp. 646–653.

[18] C. Huang and L. V. Kale, "Charisma: Orchestrating migratable parallel objects," in *Proceedings of IEEE International Symposium on High Performance Distributed Computing (HPDC)*, July 2007.

[19] J. DeSouza and L. V. Kalé, "MSA: Multiphase specifically shared arrays," in *Proceedings of the 17th International Workshop on Languages and Compilers for Parallel Computing*, West Lafayette, Indiana, USA, September 2004.

[20] C. Huang, G. Zheng, S. Kumar, and L. V. Kalé, "Performance evaluation of adaptive MPI," in *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming 2006*, March 2006.

[21] S. Chakravorty, A. Becker, T. Wilmarth, and L. V. Kalé, "A Case Study in Tightly Coupled Multi-Paradigm Parallel Programming," in *Proceedings of Languages and Compilers for Parallel Computing (LCPC '08)*, 2008.

[22] *The CONVERSE programming language manual*, Department of Computer Science,University of Illinois at Urbana-Champaign, Urbana, IL, 2006.

[23] J. C. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R. D. Skeel, L. Kalé, and K. Schulten, "Scalable molecular dynamics with NAMD," *Journal of Computational Chemistry*, vol. 26, no. 16, pp. 1781–1802, 2005.

[24] P. L. Freddolino, A. S. Arkhipov, S. B. Larson, A. McPherson, and K. Schulten, "Molecular dynamics simulations of the complete satellite tobacco mosaic virus," vol. 14, pp. 437–449, 2006.