NOISEMINER: An Algorithm for Scalable Automatic Computational Noise and Software Interference Detection

Isaac Dooley[*], Chao Mei, Laxmikant Kale
University of Illinois at Urbana-Champaign
Department of Computer Science
Urbana, IL USA
{idooley2,chaomei2,kale}@uiuc.edu

Abstract

This paper describes a new scalable stream mining algorithm called NOISEMINER *that analyzes parallel application traces to detect computational noise, operating system interference, software interference, or other irregularities in a parallel application's performance. The algorithm detects these occurrences of noise during real application runs, whereas standard techniques for detecting noise use carefully crafted test programs to detect the problems.*

This paper concludes by showing the output of NOISEMINER *for a real-world case in which 6 ms delays, caused by a bug in an MPI implementation, significantly limited the performance of a molecular dynamics code on a new supercomputer.*

1. Introduction

Automated analysis techniques are critical when scaling applications to hundreds of thousands of processors. A single person cannot possibly monitor online data streams from an application, or even look through trace logs for all processors in a reasonable amount of time. Therefore performance analysis techniques are useful when they can scale to such volumes of data. One technique for scalably analyzing large volumes of data either online or offline is *stream mining*. Stream mining refers to any data mining algorithm that only makes a single linear pass through its input data, while maintaining a synopsis of the data. The synopsis must have bounded memory requirements, and at any point in the processing of the stream, it can be converted into a useful report. This paper proposes and provides results from a novel scalable stream mining algorithm that detects computational noise that interferes with the performance of parallel applications on large clusters.

[*]This work was funded in part by the DOE High Performance Computer Science Fellowship Program.

2. Motivating Problem

In January 2007, developers discovered that the widely used molecular dynamics code NAMD[15, 16] performed poorly on a brand new large supercomputer cluster named Lonestar at the Texas Advanced Computing Center. After manual analysis of post-mortem performance trace visualizations, they discovered that some events on a specific processor would take 6 ms longer than expected. The 6 ms delays were not limited to a single portion of the computation, but rather occurred in various parts of the computation. Eventually these developers determined that the culprit was a bug in the MPI library whereby some MPI calls would take around 6 milliseconds. The problem was finally fixed by a software update to the MPI library.

The histogram in Figure 1 shows the durations of events from a NAMD run on the new Lonestar cluster, with hundreds of events each taking 6 ms longer than expected. The left portion of the histogram represents events with expected durations less than 1 ms. In the middle of the same histogram are a a group of events with exceptionally long durations. This paper describes an automated scalable algorithm that detects such exceptionally long events. Automatically detecting such exceptional events can increase the productivity of application developers while porting applications to new machines. The method proposed in this paper can detect multiple types of noise even though this paper just shows its use in the presence of a single predominant source as seen in Figure 1.

3. The Problem of Interference

Historically, systems have been plagued with operating system (OS) daemons or other processes that interrupt par-

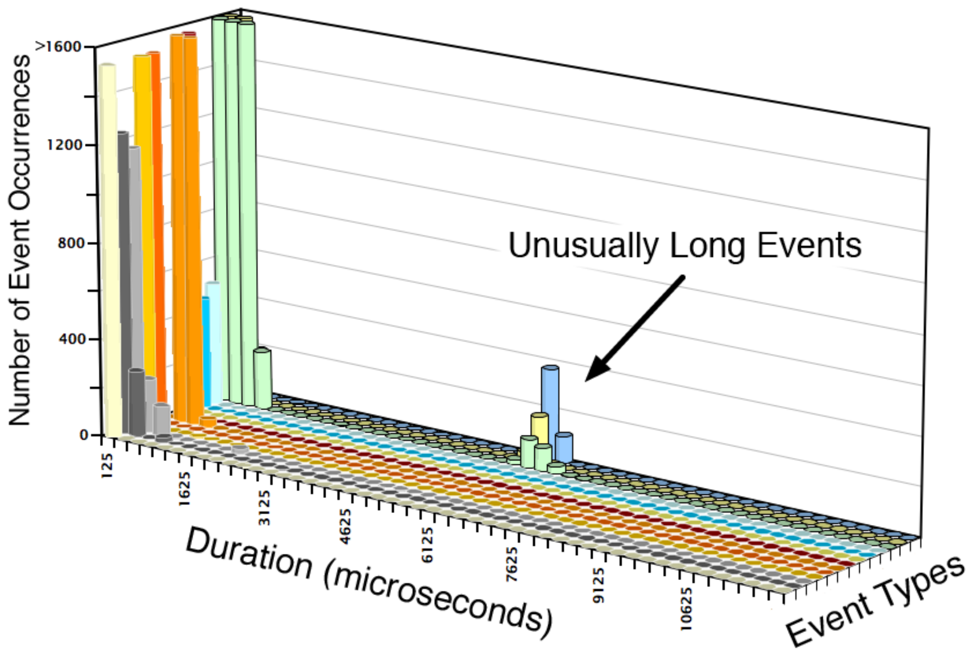


Figure 1. A histogram of event durations for a run of NAMD on the new Lonestar cluster while an MPI bug caused various MPI calls to take about 6 milliseconds longer than expected.

allel applications and cause serious delays[14, 12, 6]. More recently new problems are being discovered where the time taken to execute a single portion of an application's code has significant variability. These new problems can be caused by hardware or software. The hardware causes include dynamic processor speeds, contention for resources, silent slow fault correction, or the use of a slow path instead of an optimized fast path. One example of the last type of hardware interference occurs when subnormal floating-point values are processed more slowly in modern floating-point units[2]. Problems in software libraries can also interfere with the parallel application, as happened with the MPI bug described in Section 2. Many names are used to describe these hardware or software problems, including *OS Interference*, *Software Interference*, and *Computational Noise*. This paper refers to all these problems simply as *Computational Noise*.

Computational noise is a serious problem in parallel systems because a single occurrence of noise may delay all processors, not just the single processor on which the noise occurs [18]. It is therefore critical in large-scale parallel applications to remove or appropriately handle all sources of computational noise because the effects of noise can constrict performance and limit scalability [11]. Solutions such as coordinated-scheduling of OS interference can mitigate such problems if used appropriately [14].

The original published methods for detecting OS noise are to run simple special-purpose benchmarks. Such a benchmark would repeatedly time the duration of a barrier call, performing no other work between barriers. The durations for each barrier call should be constant if noise is not present, but the sampled durations will have high variance if noise is present. Unfortunately such simple benchmarks do not detect all types of noise that can hurt a parallel application. The simple benchmark does not use computational resources that can introduce noise, such as floating-point logic. Because simple benchmarks do not capture all types of noise that may affect a real-world application, a method such as NOISEMINER is valuable for detecting noise affecting complicated real-world applications. The literature contains a further discussion of computational noise[13, 4, 9, 5, 3, 8, 1].

4. NOISEMINER

This section describes the new NOISEMINER data mining algorithm that can detect anomalous patterns indicative of computational noise. Multiple noise sources, each with differing durations and periodicities can be detected. The method is fully automatic, so no input from a user is required. Notably, the method detects expected event durations automatically.

NOISEMINER makes a single pass through parallel application traces while maintaining a synopsis of each processor's trace. It can then generate from the synopses a list of noise components that occur across the processors. Each noise component represents a set of events across one or more processors with a similar duration of noise. Because the synopsis has a small bounded memory requirement, any length parallel trace can be analyzed. If the NOISEMINER technique is implemented in parallel, the combining of synopses can be implemented as a reduction across the set of processors. Such a reduction is scalable to large numbers of processors. Therefore NOISEMINER can analyze arbitrarily large trace logs or continuous online data streams for arbitrarily large numbers of processors.

4.1. Input

The input for NOISEMINER is a trace of the parallel application on each processor. The traces could either be processed online or offline (post-mortem). Each processor's trace contains a list of events $\{e_1, e_2, e_3, ..., e_n\}$. Each event represents a portion of the computation for the application. The events could be specified in a number of different ways, depending on the implementation. For example they could be compute kernel calls, code regions of the application between MPI calls, or Charm++ message driven entry methods. Each event must have an associated start time t_{start}, an end time t_{end}, and a $type$. The durations $(t_{end} - t_{start})$ for events of a single type are expected to be similar if noise is not present. Any event with significantly different duration is considered to be affected by noise.

The initial implementation is incorporated into the `Projections` tracing and performance visualization framework. The events in this system all have start times and durations. In this initial implementation, the events stored in the window are simply references to the native trace event objects. In addition to just timestamps and event types, the native events also contain useful data used to visualize performance characteristics of the object and its associated processor. In the `Projections` framework, the times for each event are produced by a processor local clock, which need not be synchronized across the entire cluster. The tracing framework attempts to correct discrepancies between processor times so that all processor traces have approximately consistent times. Although the NOISEMINER technique can be implemented in a parallel runtime, the initial implementation is incorporated in a post-mortem analysis tool.

4.2. Generating a Synopsis

A data synopsis is maintained for each processor while consuming a stream of trace data in a single pass. The data synopsis for each processor is a set of histograms, one for each type of occurring event. Each histogram bin corresponds to a range of event durations. As shown in Figure 2, each histogram bin contains:

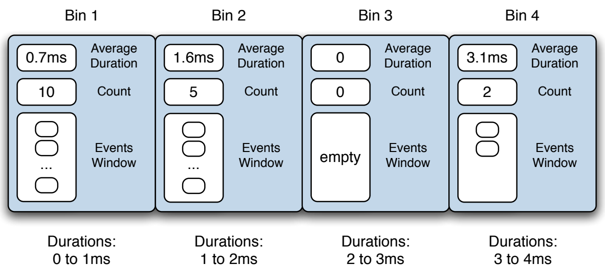


Figure 2. Histogram Structure

- An average duration for all events that have been inserted into the bin,
- A count of the number of events that have been inserted into the bin, and
- A small window of the w most recent events that have been inserted into the bin.

As each event is processed, it is simply inserted into the histogram for its type. Specifically, it is inserted into the histogram bin that matches its duration. To insert an event into a histogram bin, the average duration is updated, the count is simply incremented by one, and the event is added to the head of the window. If the window contains more than w events, the oldest event from the window is discarded. By keeping a fixed size window of events, the synopsis will have a fixed upper memory bound as desired.

The histograms used in the initial implementation each have a fixed number of bins, with a single extra bin for events with durations exceeding the maximum range of the histogram. No sensitivity analysis has yet been performed to optimize the choices for number of bins and width of each bin. So far, using a fixed number of bins in each histogram has not caused problems. It is conceivable however that large input traces could require large amounts of memory. It is important to note that all of the histogram bins will be just null pointers, until an actual event matching the bin duration is encountered. The sparsity of the bins containing data along with the windowing of events keeps memory requirements for the histogram low even if the histogram contains many bins.

This paper assumes that histograms for a processor exhibiting noise will be similar to that of Figure 3. Namely, there will be a group of events representing the expected duration without the presence of noise, and there will be zero or more groups of events that represent occurrences of noise.The majority of the events will have similar durations, and hence be used to determine the expected duration. Each

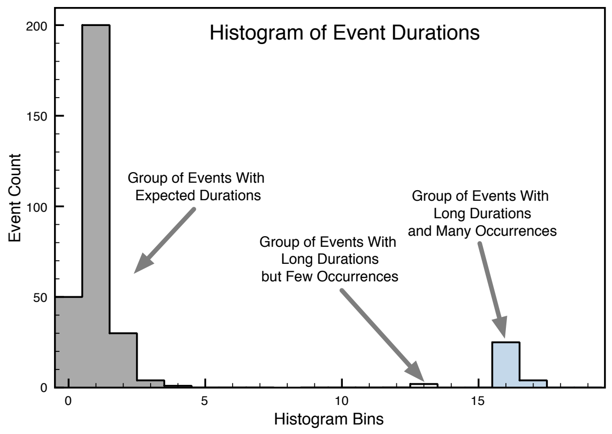


Figure 3. A typical per-processor per-event-type histogram with three groups of events. The rightmost group is likely indicative of a source of noise that causes multiple delays with the same duration

group of many events with similar, but abnormally long, durations is an indication of the existence of a potential noise source. The noise itself must be long enough to differentiate the noisy events from the non-noisy events in the histogram. The initial implementation uses bins of duration $10\mu s$, so any noise affecting the duration of the events by less than $10\mu s$ is undetectable. Although such short duration noise may be undetectable, this type of noise is likely not to cause serious problems for all but the finest grained parallel applications. Noise also will not be detected in the cases where the noise duration is small compared to the variance in the expected durations. In these cases, the peak that would otherwise have been equated to noise is absorbed into the expected duration group.

4.3. Generating the Output Report from the Synopsis

After events for each processor have been inserted into the synopsis for their respective processors, a report can be generated from the data in the synopses without re-examining the original stream of input events.

Noise will interrupt or slow down some events, causing their durations to be longer than those unaffected by the noise. NOISEMINER will examine the histograms for all events across all processors to find occurrences of noise, clustered into groups of events with similar noise duration. We define *noise duration* to be the difference between an event's duration when noise occurred and the event's expected duration. Noise duration can also be considered to be the amount of time by which an event was stretched beyond its expected duration.

Associated with each processor is a synopsis. Each synopsis contains a set of histograms, one for each type of event. To generate a report using the synopses from all processors, first a series of normalization and clustering operations are applied independently to each processor's synopsis, and second the results are merged across processors to produce a final report. This section describes these two steps in detail.

For each event type's histogram on each processor, the bins of the histogram are clustered into groups of events where the groups represent events with similar durations. For example, in Figure 3, three groups of events are shown. Each group of events contains merged data from a set of adjacent bins within a histogram. One group is generated for each bin in the histogram with an event count greater than its two neighboring bins (a local maximum). All other non-empty bins are merged into the same group as whichever neighbor has greater count (bins between a local maximum and local minimum are merged in with the local maximum). When merging bins into a group, the group's average duration is computed by calculating the count-weighted average of the average duration from each constituent bin. The group's count is the sum of the counts from the constituent bins. The group's window contains the w most recent events found in any of the constituent bin windows.

After the groups of events with similar duration have been produced from the histograms, the groups of events contain an average duration and total event count. The group with largest count for each type of event is selected to represent the expected duration for its type of event. The expected duration is used then to compute the noise duration for each of the other groups by subtracting the expected duration from the average duration for the group. Each of the groups with non-zero noise duration represents a *noise component*. Even though negative duration noise components could be produced by this method, the choice was made to use the group with largest count as representative of the expected duration. If the group with shortest duration is used to represent the expected duration, a few outlying short duration events could represent a misleading expected duration. It is difficult to compute a perfect model for expected duration based just on recorded runtimes influenced by noise. Even though the group of maximum count is an approximate model for expected duration, it has so far been sufficient to provide useful results.

After the per-processor noise components are produced they are clustered across processors. Any noise components that are close enough to each other based on a distance function are merged into a cluster. The distance function depends upon the average noise duration for the noise components.

After the previous steps have been performed, a set of cross-processor noise components has been produced. The noise components contain a window of w recent events that exhibited the impact of a similar duration disturbance, along with their average duration t_{noise}, number of occurrences o, and a list of processors on which the noise component was found. Because many parallel programs have variance in the durations of their events, it is necessary to filter and sort the resulting clusters by their relative importances. The importance is based on the potential impact on a parallel run caused by the noise. The worst-case impact depends upon the frequency at which the noisy events occur, and the duration of the noise at each occurrence.

To determine the impact of the noisy events, first a frequency or periodicity p is calculated for each cluster of noise components. The periodicity is difficult to compute in the general case because only a small window of events is available, and the events may not occur uniformly throughout the temporal region covered by the window. The periodicity over the set of windowed events could be computed using an FFT or other spectral methods if the events are distributed uniformly, but in some applications the events are distributed non-uniformly, as shown in Figure 4. The method chosen to compute the periodicities in the first implementation of NOISEMINER is to simply divide the range of the t_{start} times for events in the window, by the number of events in the window.

The noise components are filtered based on the potential worst-case impact the component could make to a tightly synchronized parallel program. Specifically, the noise components are ignored if $\frac{t_{noise}}{p} < c$ for some cutoff value c. The value $\frac{t_{noise}}{p}$ is simply an approximation of the portion of the runtime spent handling the noise events assuming they do not overlap across processors. Finally the noise components are sorted by their t_{noise} values.

On standard contemporary linux installations, the OS timeslice quanta defaults to $100ms$. The quanta can be determined by calling the POSIX function `sched_rr_get_interval()`. If an application is scheduled at the same time as another process, the two processes will likely be scheduled for alternating timeslices. If the application executes continuously for at least the duration of its timeslice, then the other process will cause interference after every $100ms$ executed by the application. If a noise source is inside the program, it could occur many times within a single timeslice. Because these two different types of noise can occur, it may be useful to distinguish automatically between noise generated inside an application and noise caused by external processes. A simple criteria is applied to label each noise component with a hint as to its root cause:

1. **Internal Noise Components** are events with periodicity significantly shorter than the OS timeslice quanta. Specifically $p \leq 80ms$.
2. **External Noise Components** are events with periodicity similar to or longer than the OS timeslice quanta. Specifically $p > 80ms$.

The choice of $80ms$ as the cutoff as opposed to some other value close to $100ms$ is somewhat arbitrary, but the label determined by this simple criterion is not crucial to the final analysis. The label may help guide a user of the tool to noise components of interest, but it is not meant to be a strict classifier.

The initial implementation of NOISEMINER uses some arbitrary values for the various tunable parameters included above. It uses 5000 bins in each histogram, with $10\mu s$ widths for each bin. Thus the dynamic range for the bins is up to $50ms$. Charm++ applications are normally fine grained, but in systems or applications with very long events, the histograms might need to be structured differently. Each event window holds up to 50 events.

4.4. Output

The output from NOISEMINER is a list of detected noise components. This section describes the initial implementation's GUI output of this noise component list. The initial implementation of NOISEMINER is included in the Projections performance visualization tool[6, 7]. An inquisitive reader of this paper could readily download the source code for further examination. The implementation is written in Java, using Generics for the data structures used in the code, along with Java's sort function. The code is written in about 1000 documented source code lines for the algorithm, ignoring the graphical display code.

Figure 5 shows a table view of a list of output noise components. The columns in the table display the t_{noise}, p, o, the label predicting the root cause, a list of processors on which the noise was detected, and a way to drill-down to a noise component view as shown in Figure 6. The noise component view displays many small timelines, each one for a different event found in the noise component's window. Each timeline in this view supports mouse-over tooltips for its events providing even more specific information. Because the view contains many timelines for a noise component, a user can quickly view them at once, quickly determining which events in the application exhibit the noise. The user can then form educated hypotheses about potential causes of noise.

5. Case Study: NAMD on Lonestar

The original motivation for creating a tool such as NOISEMINER was a NAMD run that contained anomalous stretched events. The application performance was poor, but it took a human to locate and to identify the stretched

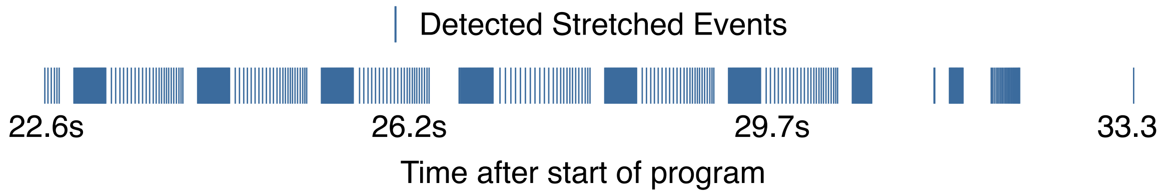


Figure 4. 500 consecutive occurrences of noisy events from the NAMD run on Lonestar described in Section 2. Each tick mark represents an event with noise duration 5.7 ms. The noise occurrences are clearly non-uniform, as there are regions dense with marks, along with regions with no tick marks

events which caused the bad performance. An automated solution was desired to make the performance analysis more effective. This section shows the resulting NOISEMINER automated solution in use on the same problem. The original trace logs, at first analyzed by hand, are analyzed with the initial implementation of NOISEMINER. The traces were produced by NAMD on the Lonestar cluster at TACC shortly after the machine was built.

The screenshot in Figure 5 shows the main result window, which contains a prominent noise component with $t_{noise} = 5.70$ ms. This noise component occurs 1425 times once every 21.34 ms on processor 0. Clicking on the "view" button for this noise component produces a visualization for the corresponding noise component view shown in Figure 6.

Each of the 36 timelines shows an event that is about 5.7 ms longer than expected. The affected events are not just of a single type, which means that a specific portion of the application is probably not the cause. However, a trend is quickly seen across all of the timelines in the view, namely, `MPI_Send` is called (represented by the thin blue bar across tops of the stretched timeline events in the center of each timeline). A knowledgeable NAMD developer and Projections user would also know that the timelines with no displayed events in the middle correspond to times where the Charm++ runtime system could perform activities such as performing MPI calls. Thus in this one view the user could conclude that a likely cause of the 5.7 ms delays in the parallel program is a problem with the underlying MPI library. After a software update, the 5.7 ms delays in the original faulty MPI library were eliminated, and NAMD performance improved to the expected speed.

The NOISEMINER tool allows application developers more easily to detect and to quantify problems such as the one described above because they can automatically find suspect events on which to focus their investigations. In the future more tools such as NOISEMINER will be critical in analyzing performance anomalies because people will not have sufficient time to search for problems within trace logs from hundreds of thousands of processors manually.

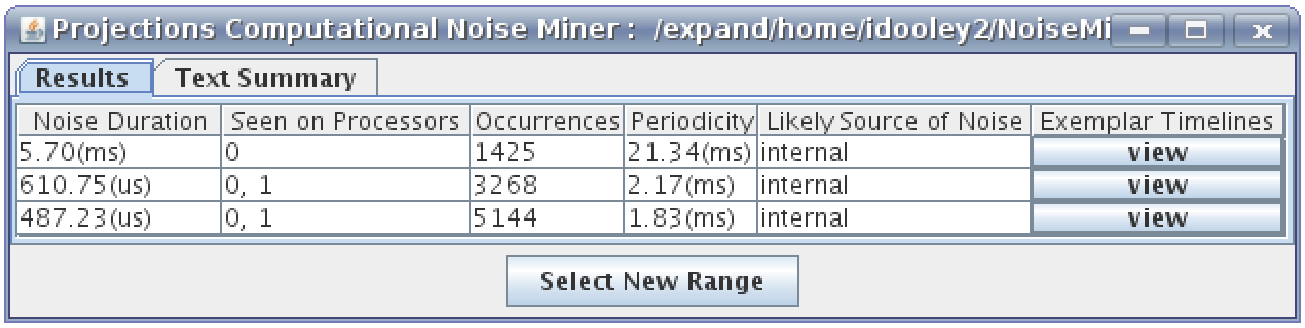
6. Related Work

The NOISEMINER algorithm is a useful tool for mining large parallel application performance traces to detect periodic anomalous patterns indicative of Computational Noise. The tool uses stream mining techniques for classification and clustering to detect abnormal events and cluster them across processors. NOISEMINER and the Sequitur based method of Tabatabaee and Hollingsworth are the first attempts to automatically detect interference in real applications. NOISEMINER uses stream mining techniques to detect parallel performance problems, while Tabatabaee and Hollingsworth uses sequence mining techniques to detect problems indicitive of anomalies relevant to security and system administration[17].

Anomaly detection for administrating grid environments is a related field, where tools attempt to detect network resource contention and intrusion detection. These techniques, however operate on a much larger granularity and attempt to ignore the expected periodic anomalies [10].

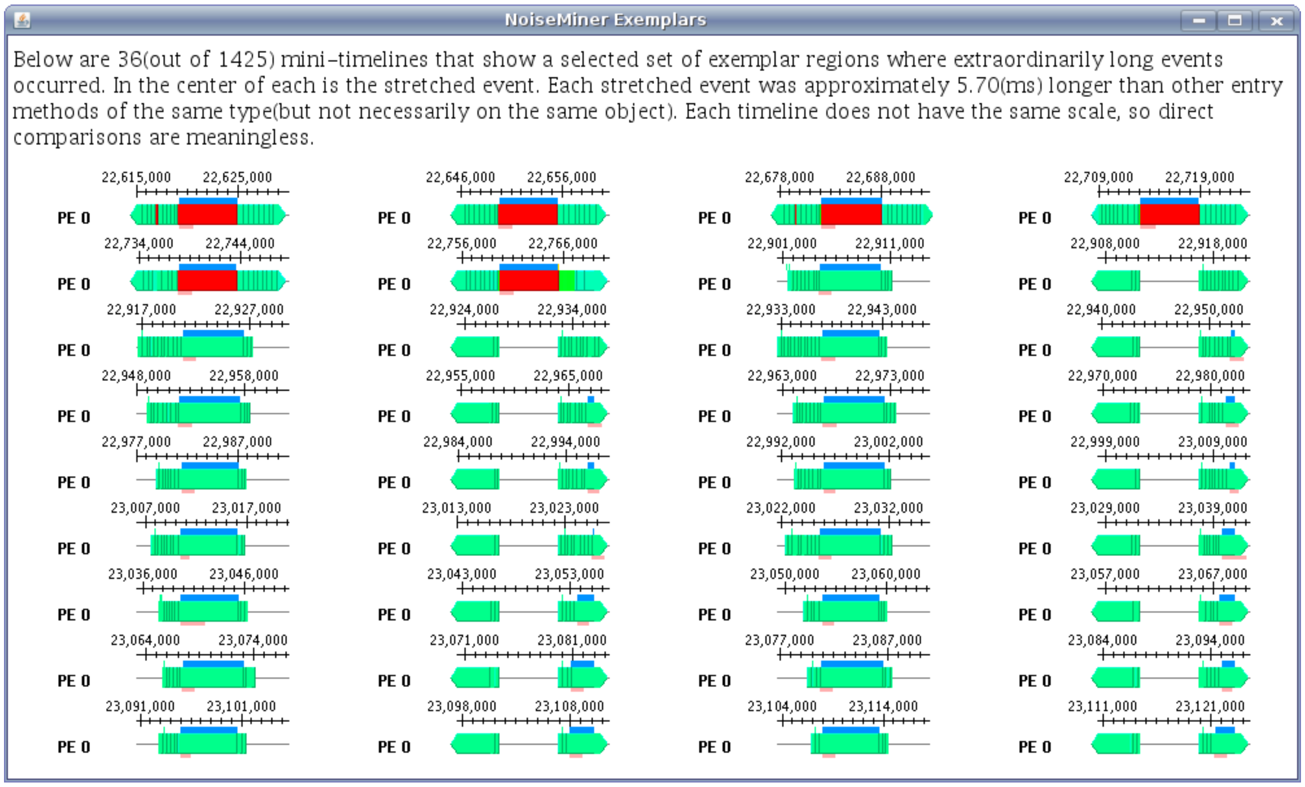
7. Future Work

The initial implementation is only applicable for applications that can be visualized in Projections. This includes Charm++, and MPI applications run with the Adaptive MPI implementation. Future implementations of the generic NOISEMINER method should be implemented within other performance analysis tools or languages. The decomposition of an MPI program into events may not be as simple as it was for the Charm++ case where each entry method is an event. Finally, although the method is described in this paper, it remains future work to implement online versions of NOISEMINER by integrating the method with existing parallel runtime systems.


Projections Computational Noise Miner : /expand/home/idooley2/NoiseMi
Results
Text Summary
Select New Range

Noise Duration	Seen on Processors	Occurrences	Periodicity	Likely Source of Noise	Exemplar Timelines
5.70(ms)	0	1425	21.34(ms)	internal	view
610.75(us)	0, 1	3268	2.17(ms)	internal	view
487.23(us)	0, 1	5144	1.83(ms)	internal	view

Figure 5. Main display of NoiseMiner results. Three noise components are displayed, with the 5.70 ms source of noise occurring in 1425 events on processor 0. Clicking the "view" button for the 5.70 ms line will open the window shown in Figure 6.



Below are 36(out of 1425) mini-timelines that show a selected set of exemplar regions where extraordinarily long events occurred. In the center of each is the stretched event. Each stretched event was approximately 5.70(ms) longer than other entry methods of the same type(but not necessarily on the same object). Each timeline does not have the same scale, so direct comparisons are meaningless.

Figure 6. A resulting visualization of one noise component. 36 small timelines are displayed in one screen, below an explantory text that will help new users decipher the results. The middle third of each timeline is the event affected by the noise or interference. In the cases where the middle third is a rectangle(in our case a Charm++ entry method), a blue bar appears above the event signifying an MPI_Send occurs for the duration of the bar. In the cases where a rectangle is conspicuously missing, the runtime system is performing some action, including a possible MPI_Send.

References

[1] T. E. Anderson, H. M. Levy, B. N. Bershad, and E. D. Lazowska. The interaction of architecture and operating system design. In *ASPLOS-IV: Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, pages 108–120, New York, NY, USA, 1991. ACM Press.

[2] I. Dooley and L. Kale. Quantifying the interference caused by subnormal floating-point values. In *Proceedings of the Workshop on Operating System Interference in High Performance Applications*, September 2006.

[3] A. C. Dusseau, R. H. Arpaci, and D. E. Culler. Effective distributed scheduling of parallel workloads. In *SIGMETRICS '96: Proceedings of the 1996 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 25–36, New York, NY, USA, 1996. ACM Press.

[4] D. G. Feitelson and L. Rudolph. Parallel job scheduling: Issues and approaches. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing – IPPS'95 Workshop*, volume 949, pages 1–18. Springer, 1995.

[5] A. Gupta, A. Tucker, and S. Urushibara. The impact of operating system scheduling policies and synchronization methods of performance of parallel applications. In *SIGMETRICS '91: Proceedings of the 1991 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 120–132, New York, NY, USA, 1991. ACM Press.

[6] L. V. Kalé, S. Kumar, G. Zheng, and C. W. Lee. Scaling molecular dynamics to 3000 processors with projections: A performance analysis case study. In *Terascale Performance Analysis Workshop, International Conference on Computational Science(ICCS)*, Melbourne, Australia, June 2003.

[7] 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*, volume 22, pages 347–358, February 2006.

[8] D. J. Kerbyson, A. Hoisie, and H. J. Wasserman. Modelling the performance of large-scale systems. *IEE Proc.-Softw.*, 150(4):214–221, August 2003.

[9] R. L. Knapp, K. L. Karavanic, and D. M. Pase. Detecting runtime environment interference with parallel application behavior. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, 2007.

[10] C. L. Lingyun Yang, J. M. Schopf, and I. Foster. Anomaly detection and diagnosis in grid environments. In *SuperComputing 2007*, November 2007.

[11] R. G. Minnich, M. J. Sottile, S.-E. Choi, E. Hendriks, and J. McKie. Right-weight kernels: an off-the-shelf alternative to custom light-weight kernels. *SIGOPS Oper. Syst. Rev.*, 40(2):22–28, 2006.

[12] R. Mraz. Reducing the variance of point to point transfers in the ibm 9076 parallel computer. In *Supercomputing '94: Proceedings of the 1994 ACM/IEEE conference on Supercomputing*, pages 620–629, New York, NY, USA, 1994. ACM Press.

[13] J. K. Ousterhout. Scheduling techniques for concurrent systems. In *3rd International Conference on Distributed Computing Systems*, pages 22–30, 1982.

[14] F. Petrini, D. Kerbyson, and S. Pakin. The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q. In *ACM/IEEE SC2003*, Phoenix, Arizona, Nov. 10–16, 2003.

[15] 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*, 26(16):1781–1802, 2005.

[16] K. Schulten, J. C. Phillips, L. V. Kale, and A. Bhatele. Biomolecular modeling in the era of petascale computing. In D. Bader, editor, *Petascale Computing: Algorithms and Applications*, pages 165–181. Chapman & Hall / CRC Press, 2008.

[17] V. Tabatabaee and J. K. Hollingsworth. Automatic software interference detection in parallel applications. In *SuperComputing 2007*, November 2007.

[18] D. Tsafrir, Y. Etsion, D. G. Feitelson, and S. Kirkpatrick. System noise, os clock ticks, and fine-grained parallel applications. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 303–312, New York, NY, USA, 2005. ACM Press.