# Noise Inspector Tool

Gladys Utrera*, Jordi Fornes*
*Department of Computer Architecture
Universitat Politècnica de Catalunya
c/ Jordi Girona, 1-3
08034 Barcelona, Catalunya, Spain
Email: gutrera,jfornes@ac.upc.edu

Jesus Labarta*†
†Barcelona Supercomputing Center
c/ Jordi Girona, 31
08034 Barcelona, Catalunya, Spain
Email: jesus.labarta@bsc.es

***Abstract*—The operating system noise can interfere with normal execution programs. This behavior is becoming especially important when scaling parallel programs and amplified with global synchronizations. This work presents a tool to detect in a non-intrusive way the alien programs that share resources with current running applications in a multicore cluster.**

***Keywords*-Operating system noise; Tracing tools; Profiling tools; Performance analysis**

## I. Introduction

The prediction of the behavior of a parallel program is not an easy issue. One may need to inspect the source code of the program to understand it. Usually the behavior of a parallel program is analyzed through tools that can gather information regarding the application performance like the amount of time spent in different parts of the program or the occurrence of certain events during the execution.

However, runtime behavior may vary not only depending on the interaction of the components of the parallel program but on external factors to the program such as network traffic, resource occupancy and sharing. These factors will not be explained by analyzing the internal behavior of the program.

We are centered on the impact caused by alien programs like operating system services. These services can arise as a consequence of the program execution like virtual memory services and I/O accesses, or because of regular operating system maintenance. In the literature, the execution of such programs, called noise or jitter, have been widely analyzed and characterized [1]–[5]. However, from a user point of view it is not easy to detect, or even to quantify. Moreover, it is not obvious at all how to predict or even analyze such behavior or the impact and scalability on the currently running programs.

The total time inverted in such external activity represents less than $1\%$ [5] of the total execution time of a user application. However, the real impact comes when such activity is constantly making micro-interruptions in a regulary often synchronized application. The scalability of such applications degrades dramatically (see Figure 3).

In this work we present a user-level tool that is able to extract accurate information at runtime of any activity that is being co-scheduled with the currently running program in a multicore cluster and without any root intervention.

We present a methodology to analyze the current system activity as well as its impact on regular running applications. The applications depending on their communication type may be affected in different proportion. In addition, system activity may have some inherent characteristics that can be taken into account when doing the CPU allocating applications within nodes, to avoid as much as possible the interference.

To our knowledge, this is a first effort to visualize and identify any system activity that arises while running applications at user level and transparent to their execution. In addition, the tool provides information compatible with the Paraver tool [6] format input traces to make post-mortem analysis.

## II. Implementation description

Operating system activity, is named as noise, since it interferes with current program executions by performing micro-interruptions. Our final objective is to generate information at runtime, that describes the use of CPUs in a group of nodes during a period of time.

The activity information must be collected in user-mode. There is no way of triggering an event when the CPU is being used by a different program. For this reason the only way to gather such information is by doing polling. The monitor tool does polling in the $/proc$ folder (linux-like operating systems register activity there) inspecting each program to see if they have had activity since the last inspection. If so, the information registered is the name of the program, amount of time with activity, the number of context switches and the CPU on which it last executed.

The information gathered is saved under $/tmp$ in a special subfolder. In this way we minimize I/O access time. However this could generate extra memory swapping. For this reason is necessary to control the size of this file to save it to disk from time to time if it exceeds a pre-defined maximum size. In practice the information gathered per node for an hour of monitoring is in the order of a few megabytes, so no swapping problems are arisen for such durations.

At the end of the inspection period of time, the monitor generates a report with a summary for each of the programs that registered use of CPU. The accuracy of the information gathered depends on the polling interval. As the system activity has very short running times, even less than a time slice, we make a continuous monitoring, so a dedicated CPU is needed to not interfere to user programs execution.

 DOI 10.1109/PDP.2017.52

Finally, after completing the period of monitoring time, we have a different file per node, which must be combined, synchronized, and formated to be visualized using the Paraver tool [6]. In order to synchronize the trace generated by the $NIT$ tool with a trace generated by the Extrae [7] library of the user program, a special Extrae event is inserted in the user program at the beginning of it. In this way it is possible to see if there is a correlation between any performance problem of the user program with any system activity that occurred by that time. We also use this methodology as part of the validation of the monitoring tool.

## III. Tool validation

We show how the data generated correlates to what happens in the system. To that end we compare the traces generated by our tool and by the Extrae library with the data collected from a running user application.

Experiments were run on the MarenostrumIII supercomputer [8], which is based on Intel SandyBridge processors with Linux Operating System and Infiniband FDR10 interconnection networks. The MPI library used is OpenMPI 1.8.1-mellanox [9]. The performance analysis were made using the Extrae library and Paraver tool [6].

The experiments were performed using two types of benchmarks: 1) an MPI parallel iterative asynchronized application; 2) an MPI parallel iterative synchronized application. Both types use the FWQ [10] for the calculation part, ensuring in this way no cache misses and no page faults are generated as a result of the program execution. Both benchmarks are iteratives. The difference is that in the first one, each process perform its iterations without any intermediate synchronization, while the second one performs a global synchronization at the end of each iteration ($MPI\_Allreduce$).

With the information gathered we compare how system activity varies in the different scenarios. This would give a clue about performance improvement not only to the programmer, also to system administrators. In addition this would enables programmers to differentiate between *external system activity*, that is independent of the existence of the application, and *internal system activity*, which is system activity generated because of the application itself (i.e. I/O intensive).

### A. Asynchronous work

Any external activity to the asynchronous benchmark will make iterations to have different durations. By inserting Extrae events at the end of each iteration, any variation (a longer interval duration) reveals stolen CPU cycles.

We can see in figure 1 the histogram of the execution generated by the Extrae library. Most of the iterations fall down in a common region with approximately 33 milliseconds interval duration, except for some outliers which fall apart with intervals duration between 48 and 52 milliseconds.

Once the the outliers are identified in the Extrae trace we can refer to the external activity trace generated by the $NIT$ library. After synchronizing both traces we can see what external activity was executing at that moment (figure 2). A sequence of states (red/dark grey in this case) delimited with flags (begin/end daemon events) are marked in the trace. The different activities are labeled with the name of the daemon and have an associated color.

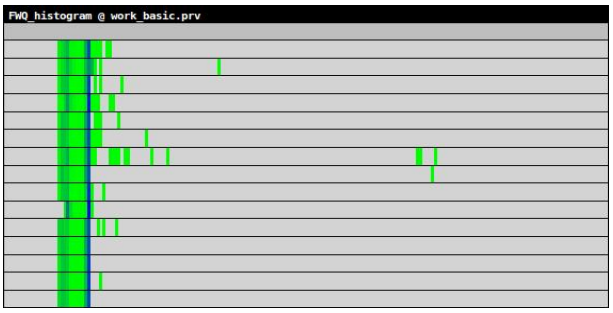


Fig. 1. Zoom trace visualization with Paraver Tool of the histogram of a sequence of the FWQ benchmark delimited with Extrae events.

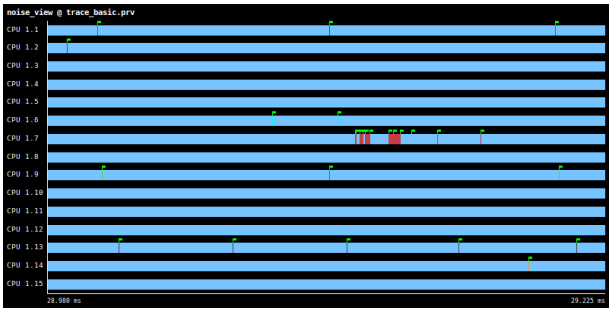


Fig. 2. Zoom trace visualization with Paraver Tool of the execution of external activity trace generated by the NIT library.

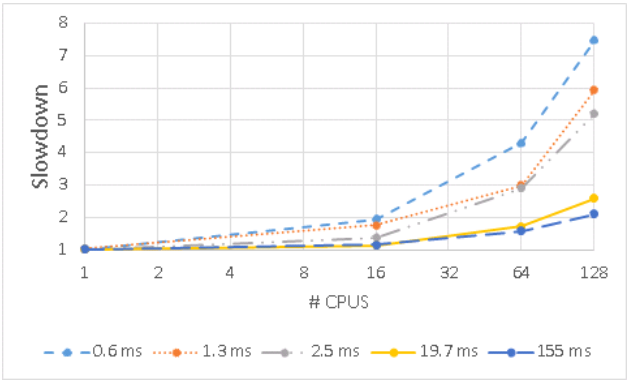


Fig. 3. Slowdown when varying interval between global synchronizations.

### B. Synchronized work

Figure 3 shows the slowdown of the synchronized benchmark when varying the number of CPUs and interval durations between consecutive global synchronizations points. We can observe that as the time interval between synchronizations diminishes and the total number of CPUs increases, the total execution time increases dramatically. As the interval between synchronizations becomes smaller, the probability of

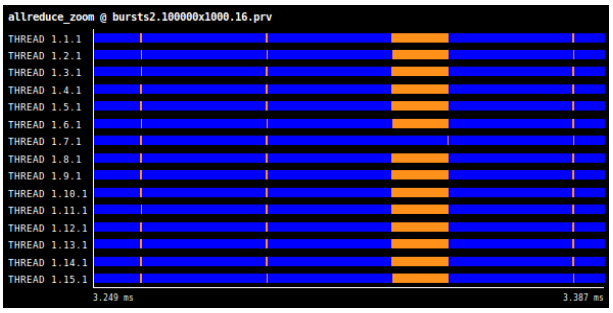


Fig. 4. Zoom trace visualization with Paraver Tool of the execution of the synchronized benchmark extracted with the Extrae library.

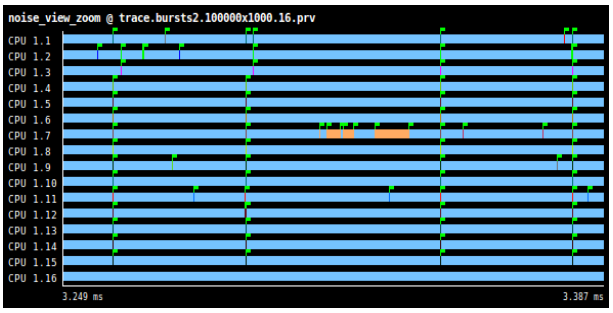


Fig. 5. Zoom trace visualization with Paraver tool of the system activity during the execution of the synchronized benchmark extracted with the $NIT$ tool.

having external activity concentrated in the same iteration across different processes diminishes. In consequence, external activity steals CPU cycles to processes at different intervals, amplifying in this way, the delay effect. Then, the total execution time is incremented.

Our NIT tool can help us to understand how the impact is performed in applications and also identify the services that provoked such behavior in the execution.

Figure 4 shows three complete iterations from the execution of the synchronized benchmark visualized with the Paraver tool. The trace was generated at runtime using the Extrae library. The x-axis represents the time in milliseconds, while the y-axis represents the MPI processes of the application. The colors represent the states of the execution, in particular blue (dark-grey) means running or calculation state, and the orange state (light-grey) represents group communication ($MPI\_Allreduce$). We can observe that during the third synchronization, all the processes except one, arrive to the synchronization point and are stuck waiting for process 7 to arrive. Looking just at this trace, there is no explanation for such delay. We could think the calculation is unbalanced, which is not true (FWQ has fixed duration). Figure 5 visualizes the trace with all the external activity generated with the NIT tool and synchronized with the original trace (figure 4). Light blue (light-grey) states means no external activiy. We can observe narrow states of activity (in the order of magnitude of microseconds) synchronized with the group communication function calls. However, what calls the attention is a light-orange state in CPU 7. This activity corresponds to a daemon, which periodically wakes up and executes during some milliseconds. This daemon steals CPU cycles from the currently running application, provoking process 7 to arrive late to the synchronization point, and consequently delaying the whole application. Using the Paraver tool it is also possible to filter a specific daemon and look at its execution pattern.

## IV. Related work

Performance monitoring tools for cluster environments can be classified into two types according where we put the focus. When interested in dynamic control of the execution of a parallel program we need online monitoring tools, with rapid feedback such as Unix command line `top`. The amount of data collection is critical and can make impractical a postmortem analysis.

On the contrary, to perform a global analysis it is needed as information as possible to make a post-mortem study, but with the lowest possible intrusion. In order to do that, parallel programs monitoring use two different approaches: clock driven (like Unix command line `mon`) or even driven monitoring (timing, counting or tracing). Inside the last group, the most general, portable and cheapest technique is software tracing, although it has to deal with the lack of a global clock [11]. KappaPI 2 [12], EXPERT [13] and Scalasa [14] perform offline postmortem analysis using tracing.

Our tool can be used with other co-monitoring applications for HPC. In this sense, $NIT$ is a good allied for Paraver [6]. Graphical trace browsers constituted a topic by themselves. An interactive 3D vision was incorporated in the TAU ParaProf tool [15] and Paraver displays its views in a 2D array of pixels scalable like a photo.

In fact, the question about how performance analysis tools present their results (time-based or not) can be used to classified the tools between profile-based tools, where profiled data comes from statistics aggregated over the time dimension and it is structured in tables or trees; and timeline visualisation tools, that show data, defined by time and processes, in 2D space keeping the time dimension. Although these two approaches are usually disconnected, they both have clear strengths and some authors have proposed to take advantage of the benefits of both approaches, interfacing profilers like KOJAK (an automatic performance evaluation system for MPI, OpenMP and SHMEM, now part of the Scalasca toolset) or PeekPerf (the control center of the IBM High Performance Computing Toolkit [16]) with analysis tools like Paraver, although the proposed methodology could be applicable to any tool that fits the functionality requirements [17].

Finally, there are tools for understanding and tune different scheduling algorithms on parallel systems. Jedule is a software tool which can visualize schedulings of parallel applications [18] and VizzScheduler is a tool also dedicated to develop and evaluate scheduling algorithms for the LogP cost model [19].

We have shown the huge impact of our tool dealing with

the application noise. At all events, it is not the target of this paper to characterize jitter, nevertheless there is a specially inspiring work on system noise that influenced our work [20]. The authors analyzed the influence of OS noise on applications with analytical modelling and simulation.

## V. Conclusions and future work

We present the $NIT$ tool which traces at execution time any activity external to the currently running user application. In order to benefit from the $NIT$ tool the user doesn't need any system privilege. The user application does not need to be instrumented. The tool runs on a separate processor in each of the nodes the user application is running to not interfere with the normal execution of the user applications.

We have validated the tool by running user applications with certain characteristics like iterative fixed work time quantum (FWQ) where any CPU cycles leakage is detected because of longer work intervals. At this point, this type of intervals are correlated with the trace generated by our tool, and the external activity is then identified. Other way of putting in relevance CPU cycles leakage is when doing regular synchronizations. The impact of any external activity on a process is amplified due to the global synchronization. This shows the importance of detecting and identifying such external activity especially for scaling purposes.

The tool proved to be very effective for detecting and identifying external activity of any user application. In order to visualize the trace we used a format compatible with the Paraver tool, which allows us to perform statistics and operations on the trace like filtering events, zooming, histograms generation. The minimum granularity of the information generated depends on the number of cores per node and on the platform used (network and memory access latencies,. . .). In the experimented platform is in the order of a few milliseconds. This is enough for system activity detection purposes.

We think that it is essential to have a global view of the activity of the system which includes not only the user applications but also any service which can be executed as a result of the operating system maintenance or as a reaction from the application system calls. Finally the tool is able to generate a unified and synchronized trace with the information from a set of nodes, not just within a node.

To our knowledge this is the first approach to trace information about any activity external to user applications in user mode in multinode-cluster. The trace generated provides such amount of information to make post-mortem interesting analysis like to deduce system activities patterns, characterizations and statistical information.

In the future we plan to apply the tool to other contexts, like job scheduling or virtual machines.

## Acknowledgment

The authors acknowledge the support of the BSC (Barcelona Supercomputing Centre). We would like to thank the anonymous reviewers for their comments, which helped us to improve the manuscript. This work has been supported by the Spanish Ministry of Science and Innovation (contract TIN2015-65316), the Spanish Ministry of Economy, Industry and Competitiveness (HAR2014-57776-P) and the Generalitat de Catalunya (2014-SGR-1051).

## References

[1] R. Mraz, "Reducing the variance of point to point transfers in the ibm 9076 parallel computer," in *Proceedings of the 1994 ACM/IEEE conference on Supercomputing*. IEEE Computer Society Press, 1994, pp. 620–629.

[2] J. C. Phillips, G. Zheng, S. Kumar, and L. V. Kalé, "Namd: Biomolecular simulation on thousands of processors," in *Supercomputing, ACM/IEEE 2002 Conference*. IEEE, 2002, pp. 36–36.

[3] F. Petrini, D. J. Kerbyson, and S. Pakin, "The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of asci q," in *Supercomputing, 2003 ACM/IEEE Conference*. IEEE, 2003, pp. 55–55.

[4] T. Jones, S. Dawson, R. Neely, W. Tuel, L. Brenner, J. Fier, R. Blackmore, P. Caffrey, B. Maskell, P. Tomlinson *et al.*, "Improving the scalability of parallel jobs by adding parallel awareness to the operating system," in *Supercomputing, 2003 ACM/IEEE Conference*. IEEE, 2003, pp. 10–10.

[5] D. Tsafrir, Y. Etsion, D. G. Feitelson, and S. Kirkpatrick, "System noise, os clock ticks, and fine-grained parallel applications," in *Proceedings of the 19th annual international conference on Supercomputing*. ACM, 2005, pp. 303–312.

[6] Barcelona Supercomputing Center. Paraver: a flexible performance analysis tool. [Online]. Available: http://www.bsc.es/computer-sciences/performance-tools/paraver/general-overview

[7] ——. Extrae instrumentation library. [Online]. Available: https://www.bsc.es/computer-sciences/extrae

[8] ——. (2016) Marenostrum 3. [Online]. Available: http://www.bsc.es/marenostrum-support-services/mn3

[9] OpenMPI Open Source HPC. OpenMPI Open Source HPC. http://www.open-mpi.org/. [Online]. Available: http://www.open-mpi.org/

[10] Lawrence Livermore National Laboratory. Asc sequoia benchmark codes. https://asc.llnl.gov/sequoia/benchmarks. [Online]. Available: https://asc.llnl.gov/sequoia/benchmarks

[11] J. C. De Kergommeaux, E. Maillet, and J. Vincent, "Monitoring parallel programs for performance tuning in cluster environments," *Parallel Program Development for Cluster Computing: Methodology, Tools and Integrated Environments" book, P. Kacsuk and JC Cunha eds*, 2001.

[12] J. Jorba, T. Margalef, and E. Luque, "Performance analysis of parallel applications with kappapi 2." in *PARCO*, 2005, pp. 155–162.

[13] F. Wolf and B. Mohr, "Automatic performance analysis of hybrid mpi/openmp applications," *Journal of Systems Architecture*, vol. 49, no. 10, pp. 421–439, 2003.

[14] M. Geimer, F. Wolf, B. J. Wylie, E. Ábrahám, D. Becker, and B. Mohr, "The scalasca performance toolset architecture," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 702–719, 2010.

[15] A. D. Malony, S. Biersdorff, W. Spear, and S. Mayanglambam, "An experimental approach to performance measurement of heterogeneous parallel applications using cuda," in *Proceedings of the 24th ACM International Conference on Supercomputing*. ACM, 2010, pp. 127–136.

[16] D. Klepacki, "The ibm high performance computing toolkit," *Trace Library Documentation, IBM*, 2004.

[17] J. Giménez, J. Labarta, F. X. Pegenaute, H.-F. Wen, D. Klepacki, I.-H. Chung, G. Cong, F. Voigtländer, and B. Mohr, "Guided performance analysis combining profile and trace tools," in *European Conference on Parallel Processing*. Springer, 2010, pp. 513–521.

[18] S. Hunold, R. Hoffmann, and F. Suter, "Jedule: A tool for visualizing schedules of parallel applications," in *2010 39th International Conference on Parallel Processing Workshops*. IEEE, 2010, pp. 169–178.

[19] W. Löwe and A. Liebrich, "Vizzscheduler-a framework for the visualization of scheduling algorithms," in *European Conference on Parallel Processing*. Springer, 2001, pp. 62–66.

[20] T. Hoefler, T. Schneider, and A. Lumsdaine, "Characterizing the influence of system noise on large-scale applications by simulation," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–11. [Online]. Available: http://dx.doi.org/10.1109/SC.2010.12