On the scalability of tracing mechanisms[1]

Felix Freitag, Jordi Caubet, Jesus Labarta

Departament d'Arquitectura de Computadors (DAC)
European Center for Parallelism of Barcelona (CEPBA)
Universitat Politècnica de Catalunya (UPC)
{felix,jordics,jesus}@ac.upc.es

Abstract. Performance analysis tools are an important component of the parallel program development and tuning cycle. To obtain the raw performance data, an instrumented application is run with probes that take measures of specific events or performance indicators. Tracing parallel programs can easily lead to huge trace files of hundreds of Megabytes. Several problems arise in this context: The storage requirement of the high number of traces from executions under slightly changed conditions; visualization packages have difficulties in showing large traces efficiently leading to slow response time; large trace files often contain huge amounts of redundant information. In this paper we propose and evaluate a dynamic scalable tracing mechanism for OpenMP based parallel applications. Our results show: With scaled tracing the size of the trace files becomes significantly reduced. The scaled traces contain only the non-iterative data. The scaled trace reveals important performance information faster to the performance analyst and identifies the application structure.

1 Introduction

Performance analysis tools are an important component of the parallel program development and tuning cycle. A good performance analysis tool should be able to present the activity of parallel processes and associated performance indices in a way that easily conveys to the analyst the main factors characterizing the application behavior. In some cases, the information is presented by way of summary statistics of some performance index such as profiles of execution time or cache misses per routine. In other cases the evolution of process activities or performance indices along time is presented in a graphical way.

To obtain the raw performance data, an instrumented application is run with probes that take measures of specific events or performance indicators (i.e. hardware counters). In our approach every point of control in the application is instrumented. At the granularity level we are interested in, subroutine and parallel loops are the control points where tracing instrumentation is inserted. The information accumulated in the hardware counters with which modern processors and systems are equipped is read at these points.

[1] This work has been supported by the Spanish Ministry of Science and Technology and by the European Union (FEDER) under TIC2001-0995-C02-01.

Our approach to the scalability problem of tracing is to limit the traced time to intervals that are sufficient to capture the application behavior. We claim it is possible to dynamically acquire the understanding of the structure of iterative applications and automatically determine the relevant intervals. With the proposed trace scaling mechanism it is possible to dynamically detect and trace only one or several iterations of the repetitive pattern found in scientific applications. The analysis of such a reduced trace can be used to tune the main iterative body of the application.

The rest of the paper is structured as follows: In section 2 we describe scalability problems of tracing mechanisms. Section 3 shows the implementation of the scalable tracing mechanism. Section 4 evaluates our approach. Section 5 describes solutions of other tracing frameworks to the trace scalability. In section 6 we conclude the paper.

2 Scalability issues of tracing mechanisms

Tracing parallel programs can easily lead to huge trace files of hundreds of Megabytes. Several problems arise in this context. The storage requirement of traces can quickly become a limiting factor in the performance analysis cycle. Often several executions of the instrumented application need to be carried out to observe the application behavior under slightly changed conditions.

Visualization packages have difficulties in showing large traces effectively. Large traces make the navigation (zooming, forward/backward animation) through them very slow and require the machine where the visualization package is run to have a large physical memory in order to avoid an important amount of I/O.

Large trace files often contain huge amounts of redundant trace information, since the behavior of many scientific applications is highly iterative. When visualizing such large traces, the search for relevant details becomes an inefficient task for the program analyst. Zooming down to see the application behavior in detail is time-consuming if no hints are given about the application structure.

3 Dynamic scalable tracing mechanism

3.1 OpenMP based application structure and tracing tool

The structure of OpenMP based applications usually iterates over several parallel regions, which are marked by directives as code to be executed by the different threads. For each parallel directive the master thread invokes a runtime library passing as argument the address of the outlined routine. The tracing tool intercepts the call and it obtains a stream of parallel function identifiers. This stream contains all executed parallel functions of the application, both in periodic and non-periodic parallel regions.

We have implemented the trace scaling mechanism in the OMPItrace tool [2]. OMPItrace is a dynamic tracing tool to monitor OpenMP and/or MPI applications available for the SGI Origin 2000 and IBM SP platforms. The trace files that OMPItrace generates consist of events (hardware counter values, parallel regions entry/exit, user functions entry/exit) and thread states (computing, idle, fork/join). The traces can be visualized with Paraver [5].

3.2 Pattern detection

We implemented the periodicity detector (DPD) [3] in the tracing mechanism in order to perform the automatic detection of iterative structures in the trace. The stream of parallel function identifiers is the input to the periodicity detector. The DPD provides an indication whether periodicity exists in the data stream, informs the tracing mechanism on the period length, and segments the data stream into periodic patterns. The periodicity detector is implemented as a library, whose input is a data stream of values from the instrumented parameters.

The algorithm used by the periodicity detector is based on the distance metric given in equation (1).

$$d(m) = sign\sum_{i=0}^{N-1} | x(i) - x(i-m) | \qquad (1)$$

In equation (1), N is the size of the data window, m is the delay ($0<m<M$), $M<=N$, x[i] is the current value of the data stream, and d(m) is the value computed to detect the periodicity. It can be seen that equation (1) compares the data sequence with the data sequence shifted m samples. Equation (1) computes the distance between two vectors of size N by summing the magnitudes of the L1-metric distance of N vector elements. The sign function is used to set the values d(m) to 1 if the distance is not zero. The value d(m) becomes zero if the data window contains an identical periodic pattern with periodicity m.

4 Evaluation

We evaluate the following aspects of the scalable tracing mechanism: 1) Trace size reduction; 2) Improvements of the ease of visualization and application structure identification; 3) Tracing overhead; and 4) Trace completeness.

All experiments are carried out on a Silicon Graphics Origin 2000 with 64 processors running with the Irix 6.5.8 operating system. The OpenMP applications are executed in a dedicated environment with 8 CPUs. We configure the scaling mechanism such that after having detected 10 iterative parallel regions it stops writing trace data to the file until it observes a new program behavior. The parameters contained in the trace file are the thread states and OpenMP events, which include two hardware counters.

We use the applications given in Table 1: Four applications from the NAS benchmark suite: Bt (class A), Lu (class A), Cg (class A) and Sp (class A); and five applications of the SPECFp95 suite: Swim, Hydro2d, Apsi, Tomcatv, and Turb3d, all with ref data set. The applications Ep, Ft, Mg from the NAS benchmarks are not used, since their trace files are small and/or they have very few periodic patterns. Column 2 of Table 1, periodicity length, indicates that the NAS benchmarks and Apsi, Swim, and Tomcatv have only one periodicity, while Hydro2d and Turb3d have nested iterative parallel structures (N). The periodicity length is the size of the periodic pattern measured in terms of parallel regions. The number of times each periodic pattern repeats is given in column 3. For instance, the function stream of the Bt application exhibits the same pattern 201 times. In column 4 the data stream length is shown. The data stream length is approximately the product of the periodicity length by the number of iterations, due to the iterative structure of the applications. Additionally, the data stream length includes the number of functions, which do not belong to a periodic pattern.

Table 1. Evaluated benchmarks.

Application	Periodicity length	Number of iterations	Data stream length
NAS Bt	9	201	1827
NAS Cg	4, 106	72,13	1702
NAS Lu	8	251	2021
NAS Sp	14	401	5636
Apsi	6	961	5762
Hydro2D	1(N), 24(N), 269	16, 8, 196	53814
Swim	6	900	5402
Tomcatv	5	751	3750
Turb3d	1(N), 12(N), 142	24, 17, 10	1580

4.1 Reduction of the trace size

We study how much the trace file size reduces when using the scalable dynamic mechanism. Figure 1 shows the size of the trace files obtained with and without scaling mechanism. It can be seen that with scalable tracing the trace files reduced significantly. The NAS Lu trace file, for instance, reduces from 173 Mb to 8 Mb, which is a reduction of 95%.

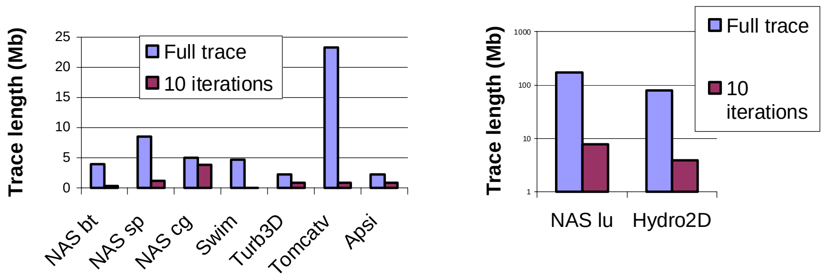

Figure 1. Comparison of the trace file size with full and scalable tracing.

4.2 Improvement of the ease of visualization and application structure identification

We want to show that scaled traces are more visualization-friendly than complete traces. In addition, by selecting the Hydro2d benchmark, we want to illustrate how the scalable tracing mechanism restarts tracing once it observes a change in the application behavior. In Figure 2 we show the visualization of the scaled trace of the Hydro2d application with Paraver. The visualization of the thread states is shown. The activity of the threads is encoded in dark color for actively computing, gray color if the thread is in the idle loop and bright color if it is in a fork/join activity.

In Figure 2 we can easily identify that there is a periodic pattern (period boundaries tagged with flags). It can be observed in the middle part that after a certain number of repetitions this pattern changes and that a new periodic pattern is then repeated. The flags in Figure 2 identify the period, so with the scaled trace it is immediate to zoom to an adequate level to see the actual pattern of behavior. In the visualization of the scaled trace the data describing iterative application behavior is not shown (black area), since the tracing mechanism did not write it to the trace file.

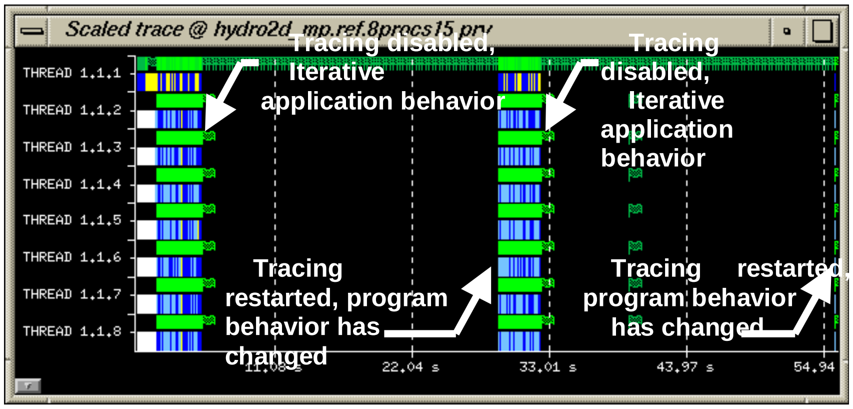
Scaled trace @ hydro2d_mp.ref.8procs15.prv
THREAD 1.1.1
THREAD 1.1.2
THREAD 1.1.3
THREAD 1.1.4
THREAD 1.1.5
THREAD 1.1.6
THREAD 1.1.7
THREAD 1.1.8
Tracing disabled, Iterative application behavior
Tracing disabled, Iterative application behavior
Tracing restarted, program behavior has changed
Tracing restarted, program behavior has changed
11.08 s
22.04 s
33.01 s
43.97 s
54.94

Figure 2. Visualization of the Hydro2D execution with scalable tracing.

4.3 Overhead of the scalable tracing mechanism

We evaluate the overhead introduced by the scaled tracing mechanism. We compare the execution time of the different benchmarks with and without tracing (Figure 3). The first bar (light gray) shows the execution time of the application without tracing, the second bar (dark gray) shows the execution time when the applications are traced with the original tracing mechanism, and the third bar (white) shows the execution time when the application is traced with the scalable tracing mechanism. The original tracing mechanism adds about 1%-3% to the execution time. With the scalable tracing mechanism, the overhead is about 3%-6%. It can be observed that the overhead introduced by the tracing tool is small in terms of execution time.

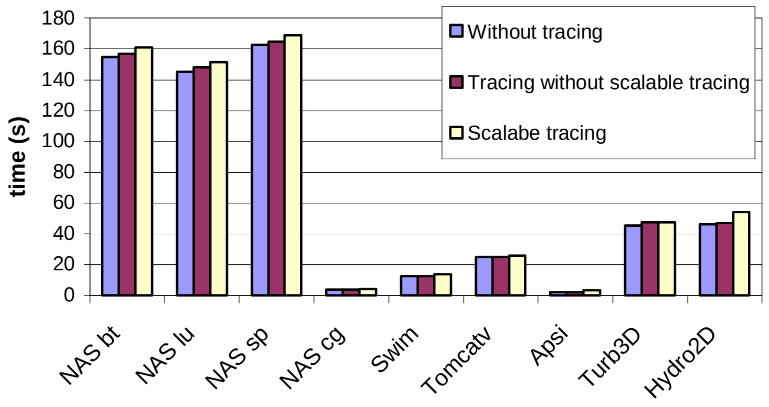

Figure 3. Overhead evaluation.

4.4 Trace completeness: Case study NAS BT

We want to show that the same quantitative performance results are obtained when analysing the full and scaled trace. With Paraver we obtain performance indices such as global load balance, duration of parallel loops, data cache misses, and TLB misses. We wish to demonstrate that using the scaled trace leads to the same conclusions on the application performance as the full trace. In a case study we compare performance indices computed from the scaled trace containing 10 iterations and from the full trace of the NAS BT application. In [1] the comparison of the scaled and full traces of the other applications given in Table 1 can be found.

We compute the load balance of the BT application from the scaled and the full trace. The values obtained from both traces for % running state per thread are very similar. On average we obtain for this application 91 % running state from both traces. Next we compare the TLB and L2 data cache misses in the parallel functions of the periodic pattern. On average the difference in TLB misses between the two traces is less than 1 %, and in L2 data cache misses it is 4%. In Figure 4 we show the execution time of the parallel functions computed from the two traces. This is the average value for all the executions of each routine during each trace. We show the 95% confidence interval. It can be seen that the function mpdo_z_solve_1 has the longest execution time (0.22 seconds approximately), followed by mpdo_x_solve_1 and mpdo_y_solve_1 (0.155 seconds approximately). This information useful to the analyst can be clearly identified from both traces.

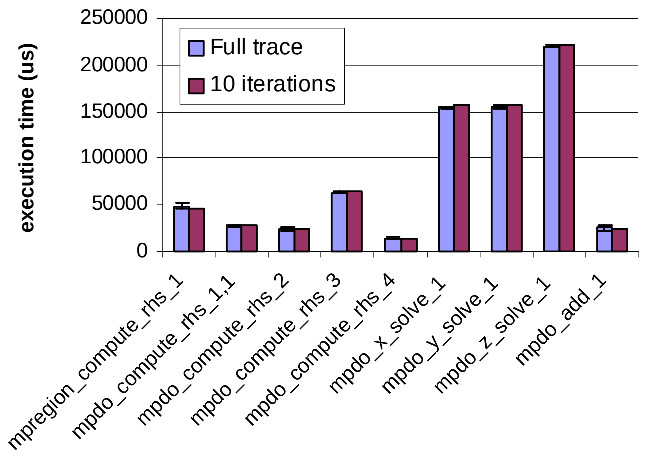

Figure 4. BT application from the NAS benchmarks. Comparison of the execution time of the parallel functions reported by Paraver from the full and the scaled trace.

5 Related work

The most frequent approach to restrict the size of the trace in current practice is to insert calls to start and stop the tracing into the source code. Systems such as VampirTrace [7], VGV [4] rely on this mechanism. OMPItrace [2] also provides this mechanism to the programmer. This approach requires the modification of the source code, which may not always be available to the performance analyst. The totally automatic way we propose in this paper is certainly more convenient as the analyst gets the structure of the application from the trace and can start identifying problems without any need of the source code.

In IBM UTE [8] an intermediate approach is followed to partially tackle the problem that very large traces pose to the analysis tool. The tracing facility can generate huge traces of events. Then, some filters are used to extract a trace that focuses on a specific application. To properly handle the fast access to specific regions of a large trace file the SLOG format (scalable logfile format) has been adopted. Using a frame index the Jumpshot visualization tool [9], for instance, improves the access time to trace data.

The Paradyn project [6] developed an instrumentation technology (Dyninst) through which it is possible to dynamically insert and take out probes in a running program. Although no effort is made to automatically detect periods, the methodology behind this approach also relies on the iterative behavior of applications. The automatic periodicity detection idea we present in this paper could also be useful inside the dynamic analysis tool to present to the user the actual structure of the application.

6 Conclusions

We have presented a number of scalability problem of tracing encountered in current performance analysis tools. We described the reasons for large traces and why this is a problem. We showed different approaches currently used to reduce the trace file size. In our approach we implemented a dynamic scalable tracing mechanism, which records only the non-iterative trace data during the application execution and stops writing the redundant data to the trace file. Our results show: 1) The size of the trace file becomes significantly reduced; 2) In order to achieve a reduced trace, our approach does not limit the granularity of tracing, nor the number of read parameters, nor the problem size; 3) The scaled trace lets the analyst faster observe relevant application behavior such as the application structure; 4) The overhead of the scaled tracing tool is small and it can be used dynamically; 5) The scaled trace can substitute the full trace in several performance analysis tasks, since the performance analyst can reach the same conclusions from the application performance indices as when using the full trace.

References

[1] J. Caubet et al. Comparison of scaled and full traces of OpenMP applications. *Tech. Report UPC-DAC-2001-31,* Dep. d'Arquitectura de Computadors, UPC, 2001.
[2] J. Caubet, J. Gimenez, J. Labarta, L. DeRose, J. Vetter. A Dynamic Tracing Mechanism for Performance Analysis of OpenMP Applications. In *International Workshop on OpenMP Applications and Tools*, pages 53-67, July 2001.
[3] F. Freitag, J. Corbalan, J. Labarta. A dynamic periodicity detector: Application to Speedup Computation. In *Proceedings of International Parallel and Distributed Processing Symposium (IPDPS 2001),* April 2001.
[4] J. Hoeflinger et al. An Integrated Performance Visualizer for MPI/OpemnMP Programs. In *In. Workshop on OpenMP Applications and Tools*, pages 40-52, July 2001.
[5] J. Labarta, S. Girona, V. Pillet, T. Cortes and L. Gregoris. DiP: A Parallel Program Development Environment. In *Proceedings of 2nd International EuroPar Conference (EuroPar 96)*, August 1996.
[6] B. P. Miller et. al. The Paradyn Parallel Performance Measurement Tools. *IEEE Computer* 28(11):37-46, November 1995.
[7] Pallas: *Vampirtrace. Installation and User's Guide.* http://www.pallas.de
[8] C. E. Wu et al. From Trace Generation to Visualization: A Performance Framework for Distributed Parallel Systems. In *Proceedings of SuperComputing (SC2000),* November 2000.
[9] O. Zaki, E. Lusk, W. Gropp, and D. Swider. Toward scalable performance visualization with Jumpshot. In *International Journal of High Performance Computing Applications,* 13(2): pages 277-288, 1999.