A Trace-Scaling Agent for Parallel Application Tracing[1]

Felix Freitag, Jordi Caubet, Jesus Labarta
Computer Architecture Department (DAC)
European Center for Parallelism of Barcelona (CEPBA)
Universitat Politècnica de Catalunya (UPC)
{felix,jordics,jesus}@ac.upc.es

Abstract

Tracing and performance analysis tools are an important component in the development of high performance applications. Tracing parallel programs with current tracing tools, however, easily leads to large trace files with hundreds of Megabytes. The storage, visualization, and analysis of such trace files is often difficult.

We propose a trace-scaling agent for tracing parallel applications, which learns the application behavior in runtime and achieves a small, easy to handle trace. The agent dynamically identifies the amount of information needed to capture the application behavior. This knowledge acquired at runtime allows recording only the non-iterative trace information, which drastically reduces the size of the trace file.

1. Introduction

Performance analysis tools are an important component of the parallel program development and tuning cycle. To obtain the raw performance data of an application, an instrumented version of the application is run with probes that take measures of specific events or performance indicators (i.e. hardware counters, subroutines, parallel loops).

The obtained trace data can be summarized on-line by the tracing tool. More often, however, it is stored in trace files for off-line analysis. We focus our interest in tracing packages for parallel programs, where all the acquired data is stored in trace files for a detailed analysis at a later time. Tracing parallel programs with such tracing tools easily leads to huge trace files with hundreds of Megabytes, which has several problems concerning storage, visualization and analysis of such traces.

We propose a trace-scaling agent for tracing tools of parallel applications. In runtime the agent learns the periodic structure in the application behavior exhibited by many scientific programs. The caption of the application behavior allows storing only the non-iterative trace information, which drastically reduces the storage requirement for trace files. We show that the agent can obtain such an understanding automatically at runtime without programmer intervention or support.

The remainder of the paper is structured as follows: In section 2 we describe scalability problems of tracing mechanisms. Section 3 shows the implementation of the trace-scaling agent. Section 4 describes some applications and results of scaled tracing. Section 5 contains further disussion of our approach. In section 6 we conclude the paper.

2. Scalability of tracing mechanisms

2.1. Problems associated to large traces

The performance analysis of parallel programs easily leads to a large number of trace files, since often several executions of the instrumented application are carried out in order to observe the application behavior under slightly changed conditions. Another reason why several traces are needed is to study how the application scales. All these traces for the different configurations of the application and the environment (number of processors, algorithmic changes, hardware counters, …) require storage space.

Visualization packages have difficulties in showing such 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. Otherwise, the response time of the tool increases significantly, strongly affecting the motivation of the programmer to carry out the performance analysis.

The high amount of redundant trace information in large trace files hides the relevant details of the application behavior. When visualizing such large traces, zooming down to identify the application structure becomes an inefficient task for the program analyst. Often, the analyst needs to have a certain understanding of the application in order to carry out an efficient performance analysis.

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

2.2. Related work

The most frequent approach to restrict the size of the trace in current practice is to insert calls into the source code of the application to start and stop the tracing. Systems such as VampirTrace [6], VGV [4], and OMPItrace [1] provide this mechanism. This approach requires the modification of the source code, which may not always be available to the performance analyst. Even if the source code is available, it is necessary to have a certain understanding of it before being able to properly insert the tracing control calls.

The Paradyn project [5] 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 be useful inside such a dynamic analysis tool to present to the user the actual structure of the application.

In IBM UTE [7], an intermediate approach is followed to partially tackle the problem, which large traces pose to the analysis tool. The tracing facility can generate huge traces of events, containing information with a lot of detail down to the level of context switches and global system activities. Then, filters are used to extract a trace that focuses on a specific application, summarizing information in record formats more amenable to visualization and better describing the application behavior. 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 [8] improves the access time to trace data.

2.3 Our approach

Our approach to the scalability problem of tracing is to adapt dynamically the traced time. We propose a trace-scaling agent, which learns in runtime the structure of the application. It automatically determines the relevant tracing intervals, which are sufficient to capture the application behavior. With the trace-scaling agent it is possible to trace only one or several iterations of the dynamically detected repetitive pattern in the application behavior. Our approach does not require limiting the granularity of tracing, nor the number of parameters read at every tracing point, nor the problem size. Due to the dynamic interception of the calls to runtime libraries in the tracing tool, our implementation does not require the source code of the application to achieve the scaled trace. In runtime the redundant trace information is identified and only the non-iterative application behavior is stored in the trace file. The analysis of such a reduced trace allows tuning the main iterative body of the application.

3. Trace-scaling agent

3.1 Recognition of iterative patterns

The trace of the application is a data stream containing the values of several parameters. If the application contains loops, then it has segments with periodic patterns. We apply a periodicity detection algorithm to the data stream in order to segment the data stream into periodic patterns. The used algorithm is frame based and requires a finite length of past data values to compute the periodicity.

We implement the periodicity detector from [3] in the trace-scaling agent in order to perform the automatic detection of iterative structures in the trace. The stream of parallel function identifiers from the trace is the input. The output of the agent is the indication whether periodicity exists in the data stream and its period length.

The algorithm used by the periodicity detector is based on the distance metric given by the equation

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

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.

If the periodicity m in the data stream is several magnitudes less than the size N of the data window, then the value d(m) may become zero for multiples of m. On the other hand if the periodicity m in the data stream is larger than the data window size N, then the detector cannot capture the periodicity. The periodicity length we found in the used applications was usually small (between 5 –20) and less than 300. For an unknown data stream, the window size N of the periodicity detector can be set initially to a large value, in order to be able to capture potentially large periodicities. Once a satisfying periodicity is detected, the window size can be reduced dynamically.

3.2. Implementation

In our implementation of equation (1), we store a finite number of previous data values of the data stream including the most recent value in a data vector. On this data vector the algorithm performs periodicity detection. This data vector can be implemented as a FIFO buffer of length M+N. This type of implementation uses the least amount of memory, but requires a higher number of operations at every instant *i* than other implementations.

Applying equation (1) on the data vector requires M x N operations to compute the values of d(m) at the instant *i* of the data stream. It can be observed, however, that some operations are done with the same data values several times at different instants *i*. The previously computed distances between vector elements could be stored to reduce the number of computations made by the algorithm at instant *i*. There is a trade-off between the number of computations made at instant *i*, and the amount of memory needed by the algorithm. In order to reduce the amount of computation we implement a FIFO organized matrix of size MxN where previously computed distances are stored. Using this distance matrix we compute at each instant *i* the value of $d_i(m)$ for all values of m, where x(i) is the value of the data value at the current sample *i*, and x(i-m) is the data value obtained m samples before. The computed values of $d_i(m)$ are written in the column corresponding to the instant *i* in the matrix.

In case of using the distance matrix to store previously computed distances, then only M instead of M x N operations need to be made at instant *i* to obtain the new distances $d_i(m)$. The value of d(m) for all values m is obtained by summing the elements of each raw of the distance matrix, i.e. the previously computed distances and each most recent distance $d_i(m)$. This means that at every instant *i* new values are written in a column of the distance matrix, and d(m) is computed as the sum of the values in each raw using the previously computed values of the other columns. Using the distance matrix the size of the data vector can be reduced to M, since at instant *i* only the distances between the new data value and the M-1 past data values need to be computed. It can be seen in equation (1) that if the data window size N and the delay M is increased, larger iterative structures can be detected. Then, the number of computations to obtain d(m) also increases. However, when increasing N and M and the previously computed distances are re-used, then the increase of operations is only linear.

In order to reduce the number of shifts of the FIFO operations, the data structures of the periodicity detector conceptually working as FIFO organized matrix and FIFO organized data vector are programmed as circular lists. At each instant *i* the pointer to the current list element shifts by one such that it points to the oldest values in the list. The new sample overwrites the column containing the oldest values with the new distance. The implementation with circular lists avoids moving the data values. The number of operations at instant *i* are reduced, which leads to a small overhead of this implementation.

3.3. OpenMP and tracing tool integration

The structure of OpenMP based parallel applications usually iterates over several parallel regions. For each parallel directive the master thread invokes a runtime library passing as argument the address of the outlined routine. The tracing mechanism intercepts the call and 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 agent in the OMPItrace tracing tool [1]. The tracing tool generates trace files, which consist of events (hardware counter values, parallel regions entry/exit, user functions entry/exit) and thread states (computing, idle, fork/join).

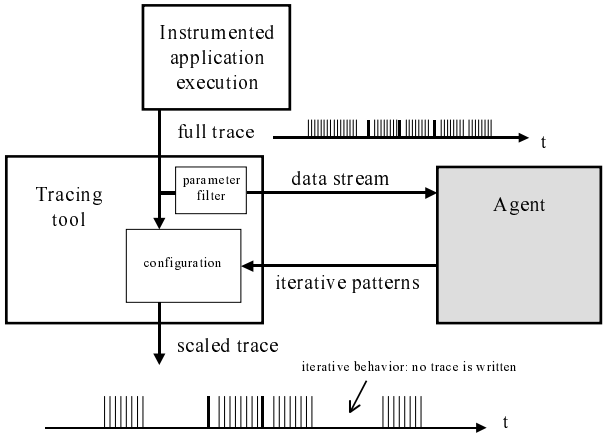


Figure 1. Interaction of the agent in the tracing tool.

In Figure 1 the interaction between the instrumented application, the tracing tool, and the agent is illustrated. It can be seen that the agent receives a data stream from the tracing tool. The data stream contains the values of a traced parameter such as the identifiers of the executed functions in parallel regions. The agent learns the application behavior. Having this indication the tracing tool knows, which is the non-iterative information to write to the trace file.

4. Applications of scaled tracing

4.1. Experimental setup

We trace 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 SPEC95 suite: Swim, Hydro2d, Apsi, Tomcatv, and Turb3d, all with ref data set.

All experiments are carried out on a Silicon Graphics Origin 2000. The OpenMP applications are executed in a dedicated environment with 8 CPUs. We configure the trace-scaling agent 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.

Table 1. Evaluated benchmarks.

Benchmarks	*Application*
NAS benchmarks	NAS Bt
	NAS Cg
	NAS Lu
	NAS Sp
SPEC95fp benchmarks	Apsi
	Hydro2D
	Swim
	Tomcatv
	Turb3d

4.2. Application structure identification

The trace-scaling agent allows inserting information about the detected application structure in the trace records, which indicates the start/end of an iterative pattern. This information is highly useful for the analyst because one of the first activities when facing a large trace is to zoom down, trying to identify an area of a few periods that can be taken as reference for looking at details. The tracing tool writes these events indicating periodic patterns to the trace even if the writing of all other trace information is suspended.

In Figure 3 (see final page) two iterative regions of the NAS Bt benchmark with their thread states are shown. The boundaries of the iterative regions are represented as flags, which reveal the application structure. The number of periodic patterns and their duration can easily be computed from the periodicity event in the trace file.

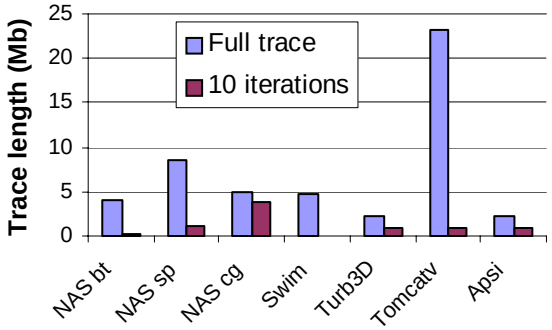
4.3. Improvement of the ease of visualization

Considering the full trace in Figure 4 (see final page) a first visual perception of the program behavior can be quite misleading. For example, it seems that there is a lot of fork/join activity in the first thread (white color) while this is only an effect of the display precision. The reason is that at the scale that had to be used to display the whole trace, each pixel represents a large time interval (152 ms) within which one thread can perform many changes of activity.

In Figure 5 (see final page) we can easily identify that there is a periodic pattern (period boundaries tagged with flags). It can be observed that after a certain number of repetitions this pattern changes and that a new periodic pattern is then repeated. The direct look at the full trace of Figure 4 hardly reveals that there is a special behavior in the middle part. The flags in Figure 5 identify the period. 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 iterative trace information is not shown (Figure 5 black area), since the tracing mechanism did not write it to the trace file.

4.4. Reduction of the trace file size

We examine how much the trace file size reduces when using the trace-scaling agent. Figure 2 shows the size of the trace files for the NAS and SPEC95 benchmarks obtained with and without using the agent. It can be seen that with scalable tracing the trace files are reduced significantly. The NAS Lu trace file, for instance, reduces from 173 Mb to 8 Mb, which is a reduction of 95%. Had we traced less than 10 iterations, the trace size would reduce more.



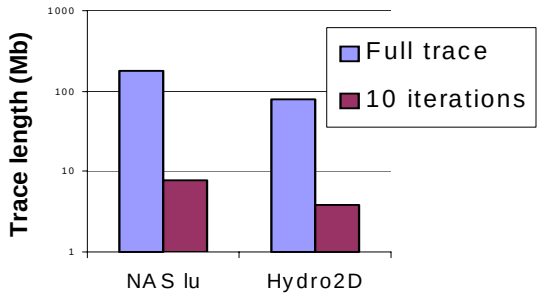


Figure 2. Comparison of the trace file size with full and scalable tracing.

5. Discussion

The overhead of an implementation is an important performance factor of real-time tools. In [2] we have evaluated the overhead produced by the trace-scaling agent. It was observed that the overhead introduced by tracing is small in terms of the execution time. The original tracing tool adds 1% - 3% to the execution time. With the trace-scaling agent, the overhead is 3% - 6%.

In applications with a periodic pattern we expect to reach the same concluysions on performance when analysing a subset of the iterations, i.e. the scaled trace. In [2] we have compared the performance indices computed from the scaled and full traces. Our results show that the same performance conclusions can be obtained when analysing the scaled trace of the applications.

The agent learns the application behavior from the stream of function identifiers. It could be possible that the agent detects iterative behavior in the executed functions, but at the same time the performance of the other indices (cache misses, TLB misses, …) could differ significantly from one iteration to another. If such a case occurs in an isolated parallel region, the agent would not detect this situation.

Our tool relies on the iterative behavior of applications, where loops are executed many times. Many scientific applications have such a structure. The studied NAS and SPEC95 benchmarks, which mostly perform numeric computations, exhibit iterative application behavior. In case of having the trace-scaling agent activated with another class of applications, which are non-iterative, simply no periodic behavior would be detected and the whole trace would be written to the file.

6. Conclusions

We have described the some scalability problems of tracing in current performance analysis tools and why these are a problem for storage, visualization, and efficient analysis. We have proposed a trace-scaling agent, which allows storing data for a complete analysis while achieving a small trace file. We have implemented the agent, which learns the application behavior in runtime and allows storing only the non-iterative trace data. We have shown that the size of such a scaled trace file becomes significantly reduced, while in the traced interval the relevant application behavior is captured. We observed that the scaled traces are easy to handle by visualization tools and the scaled trace lets the analyst faster observe relevant application behavior such as the application structure. Our implementation of the trace-scaling agent has a small overhead and it is used in runtime. The scaled trace can substitute the full trace in several performance analysis tasks, since it allows the performance analyst to reach the same conclusions on the application performance as when using the full trace.

7. References

[1] 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 (WOMPAT 2001)*, July 2001, pp. 53-67.

[2] J. Caubet, F. Freitag, J. Labarta. “Comparison of scaled and full traces of OpenMP applications.“ Tech. Report UPC-DAC-2001-31.

[3] F. Freitag, J. Corbalan, J. Labarta. “A dynamic periodicity detector: Application to speedup computation.” In *International Parallel and Distributed Processing Symposium (IPDPS 2001),* April 2001.

[4] J. Hoeflinger, B. Kuhn, W. Nagel, P. Petersen, H. Rajic, S. Shah, J. Vetter, M. Voss, and R. Woo. An Integrated Performance Visualizer for MPI/OpenMP Programs. In *International Workshop on OpenMP Applications and Tools (WOMPAT 2001)*, July 2001, pp. 40-52.

[5] B. P. Miller, M. D. Callaghan. The Paradyn Parallel Performance Measurement Tools. *IEEE Computer* 28(11): 37-46, November 1995.

[6] Pallas: *Vampirtrace. Installation and User’s Guide.* http://www.pallas.de

[7] C. E. Wu, A. Bolmarcich, M. Snir, D. Wootton, F. Parpia, A. Chen, E. Lusk and W. Gropp. From Trace Generation to Visualization: A Performance Framework for Distributed Parallel Systems. In *Proceedings of SuperComputing (SC 2000),* November 2000.

[8] 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.

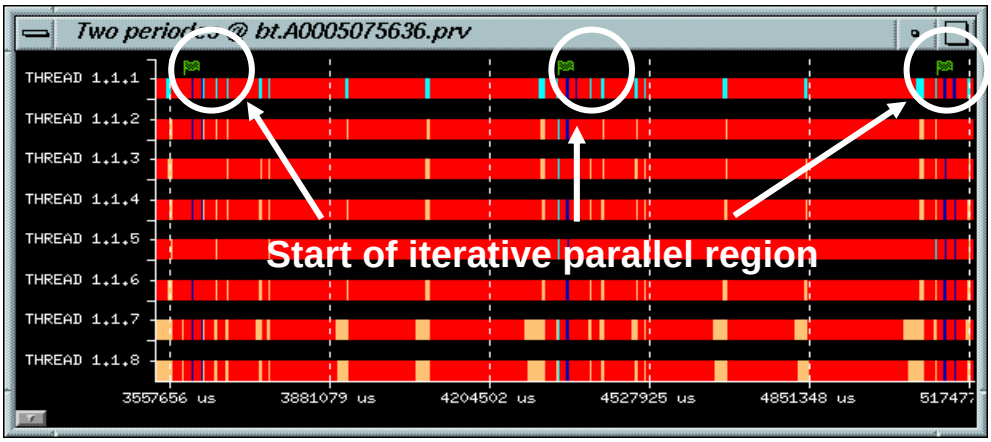


Figure 3. Visualization of the thread states in the NAS Bt application in 2 iterative parallel regions. Light color=idle, dark color=computing.

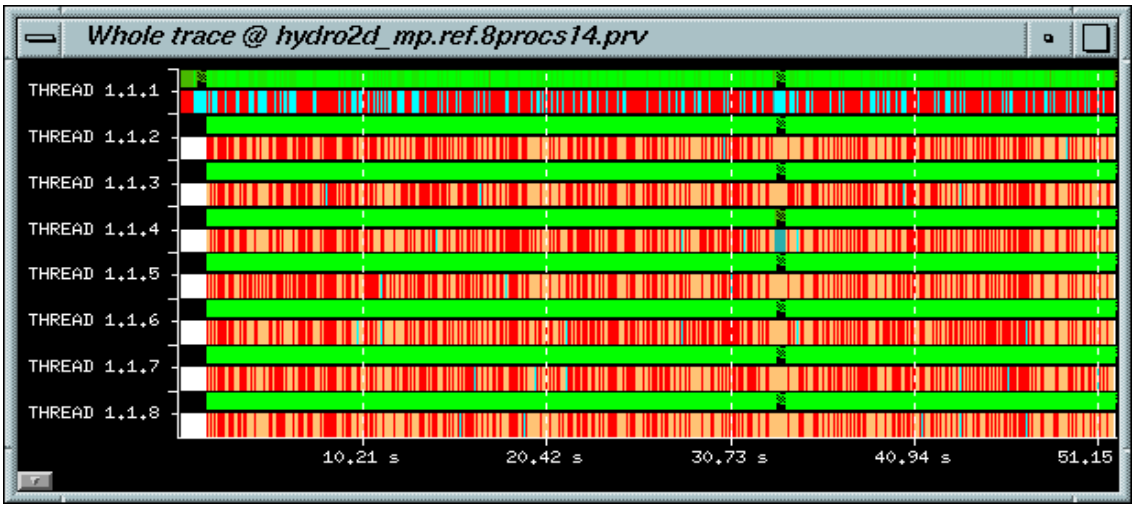


Figure 4. Visualization of the whole Hydro2D execution trace (full trace).

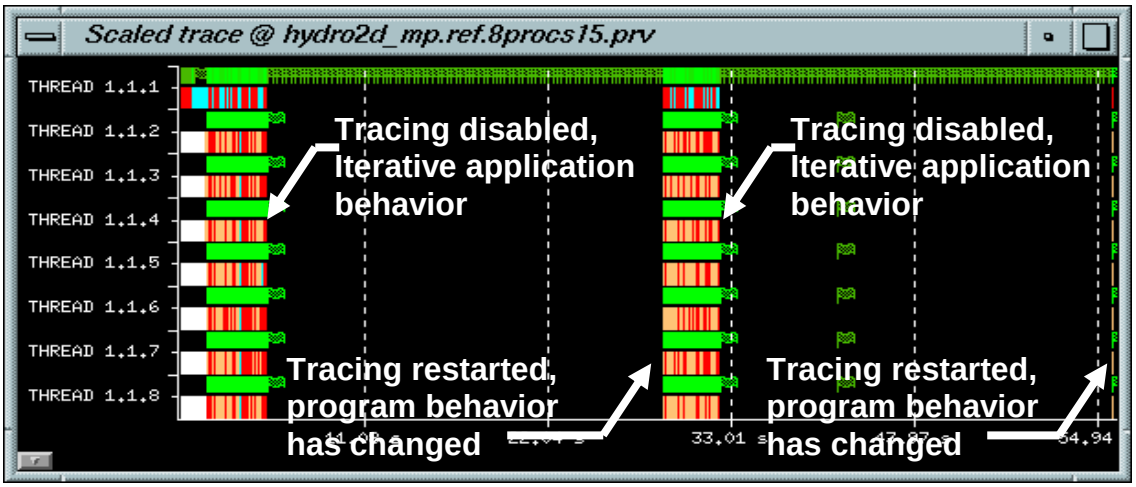


Figure 5. Visualization of the Hydro2D execution with scaled tracing (scaled trace).