On-line detection of large-scale parallel application’s structure.

Germán Llort, Juan González, Harald Servat, Judit Giménez, Jesús Labarta

July 16, 2009

Abstract

With larger and larger systems being constantly deployed, trace-based performance analysis of parallel applications has become a daunting task. Even if the amount of performance data gathered per single process is small, traces rapidly become unmanageable when merging together the information collected from all processes.

In general, an efficient analysis of such a large volume of data is subject to a previous filtering step that directs the analyst’s attention towards what is meaningful to understand the observed application behavior. Furthermore, the iterative nature of most scientific applications usually ends up producing repetitive information. Discarding irrelevant data aims at reducing both the size of traces, and the time required to perform the analysis and deliver results.

In this paper, we present an on-line analysis framework that relies on clustering techniques to intelligently select the most relevant information to understand how does the application behave, while keeping the trace volume at a reasonable size.

1 Introduction

The availability of high computational power keeps on enabling the study of more and more complex problems which could not be tackled before due to the lack of resources. Such ambitious problems tend to last for hours or even days of intensive computation before delivering results, and demand hundreds of processors. It is clear that, the longer the execution takes and the more processes are interacting, the larger amount of data will be collected by a tracing tool.

The detailed analysis of such an application poses a challenge when long runs and so many processes are involved. First, just saving and handling the trace might be unfeasible due to storage limitations. Second, the vast amount of data that has to be analyzed sharply degrades the responsiveness of the analysis tools, having a negative impact on the analyst’s ability to draw a conclusion, and ultimately, on the user’s interest.

However, most large scale scientific applications are highly iterative in nature. This results in large amounts of redundant data emitted to the trace that serve no more useful purpose than storing just a few representative iterations. Moreover, there are dozens of potentially interesting performance metrics to inspect (e.g. IPC, cache misses, network bandwidth, etc.), but the choice has to be done without prior knowledge of the application, and those selected might not be relevant enough to unravel why the application behaves as it does. An excess of irrelevant data can be counterproductive, as it can distort the results and hinder the understanding of the application’s performance.

In the case where the trace could be stored, an effective treatment of a large volume of data must undergo some kind of filtering process in order to reduce it to a lesser extent. Our previous efforts were tailored to the development of semi-automatic post-processing mechanisms to generate summarized views of the general structure of the application, and then allow the user to concentrate on all relevant de-
tails [8]. This approach has several advantages over profilers, since microscopic effects that might be important to get complete understanding of the problem are not lost in the summarization process, and the analysis is not limited to predefined averaged metrics.

Once these mechanisms are available, it is possible to apply them at runtime to intelligently focus on what is relevant for understanding the observed application behavior and thus notably reduce the amount of data generated. In this paper we propose a novel alternative to tackle the trace scalability problem in the direction of intelligent selection of the traced information. We apply clustering techniques [7] to accurately identify the application’s structure as it runs, and track its evolution over time. We are able to detect at runtime a fairly small region that faithfully represents its overall behavior. Performance data is stored in the trace with a wealth of detail for this particular region only, keeping the trace at a manageable size while still offering all the advantages of a flexible and detailed trace-based analysis.

Our objective is to provide an automatic on-line analysis framework that minimizes the amount of data emitted to the trace while maximizing the amount of relevant information presented to the analyst. Such a mechanism fosters the use of trace-based approaches in large scale scenarios, where so many limitations have to be dealt with.

So far, a variety of real applications and benchmarks have been used to test the system: Gromacs, UMO, SPECFEM3D, WRF, Gadget, NAS Parallel Benchmarks and SPEC MPI2007.

The rest of this document is structured as follows: section 2 describes the on-line analysis framework based on clustering. In section 3, we compile the conclusions and outline the future directions in our research.

## 2 On-line analysis framework

The architecture of the system is structured in three major components, as depicted in Figure 1. Performance metrics of the application being analyzed are obtained with MPItrace [2], a tracing suite that automatically intercepts calls to most common parallel programming API’s and records the values of several performance counters (selected beforehand) that PAPI library [4] makes available.

Periodically, all data is aggregated in a master process, taking advantage of MRNet [9]. This software interconnects a network of processes in a tree-like topology, and allows data traversing the tree to be summarized upon its way to the master, so as to preserve the system’s scalability.

The application structure is analyzed centrally using a clustering mechanism [7]. This tool identifies the different trends in the pure computation phases (CPU bursts, henceforth) of parallel applications. Precisely, a CPU burst is considered to be the region comprised between two consecutive communications in a message-passing application. All CPU bursts are grouped according their similarity in terms of duration and performance counters, obtaining a fine-grain characterization of the application’s structure.

Combining these tools, we have built an automatic analysis framework that is able to determine at runtime a relevant region of the execution that illustrates the overall behavior of the application, and produce a full detailed yet small trace for this time interval.

Required user intervention is limited just to specify an approximate size for the resulting trace, and
the system parameters will be automatically adjusted to produce the desired amount of information. Free control over this setting allows the user to adapt to the limitations of the analysis tools that are going to be used afterwards, so as not to generate more data than what can be conveniently handled. However, we are working on several strategies to determine the most suitable value automatically.

The following sections describe each of these components and how do they interact.

2.1 Data acquisition

MPItrace [2] is an instrumentation package to trace programs that use the message-passing programming model. It relies on the PMPI profiling interface to intercept calls to MPI. Additionally, it also provides information about OpenMP and pthreads routines by replacing the runtime references through dynamic library interposition. If the source code or a runtime code patching mechanism such as Dyninst [5] is available, any other user function can also be instrumented. Resulting traces can be visualized with Paraver [3].

Performance data is collected whenever any entry or exit event of the registered calls occur. Most of this data comes from the performance counter hardware found in most modern microprocessors, which is accessed through PAPI [4]. Typical information gathered at these points would include, but not be limited to: processor cycles, instructions completed, cache misses, etc. These performance values are stored per task into separate memory buffers rather than files. Once they have been filled, every new event overwrites the oldest. In this way, the information comprised in the buffers always reflects the most recent activity of the application.

It is worth pointing out that some processes can generate performance data at a faster pace than the rest. In order to generate a representative trace, all data retrieved for the analysis belongs to a time region where all processes are active simultaneously.

2.2 Data transmission

MRNet [9] is a communication system for scalable distributed tools with a master/worker architecture. MRNet uses a tree of processes to interconnect the tool’s front-end and back-ends, providing efficient group data transmissions between them. Intermediate processes of the tree may apply filters to synchronize and aggregate data sent to the tool’s front-end. Using filters to manipulate data in parallel as it passes through the network allows complex aggregations and keeps the tool’s front-end loads manageable.

Resources needed to create this network are additional to those used by the parallel application itself. In our experiments, all communication processes ran in a separate set of processors in order not to introduce overhead to the application. Though depending on their computation loads and resources availability, sharing resources is also a viable option.

When the program being traced with MPItrace starts executing, a back-end thread is created for every process of the parallel application, and they all connect to the tool’s front-end through the MR-Net network. These threads stall, not interfering at all with the traced application execution flow. Periodically, the front-end broadcasts a control message awakening them all, and they will then transfer all performance data (relative to the analysis that is going to be computed) that is currently stored in the buffers of the tracing tool. Alternatively, these transfers could trigger independently once a buffer fills, rather than centrally orchestrated.

In this way, the front-end gets a global view of the status of the application and how every task is performing. Aggregated data is examined and, based on the results of the analysis, the tracing tool running within the back-ends is instructed in what is the most interesting performance data to focus on.

2.3 Data analysis

The analysis conducted consists of a computation structure detection using a clustering tool [7]. The main purpose of this mechanism is to detect computation regions (i.e., CPU bursts) with similar behavior and, eventually, to identify phases of the application.
Every CPU burst is defined by its duration and a set of performance metrics read at the beginning and end of the region. The clustering analysis can use an arbitrary number of these available metrics to characterize the application (e.g., 'instructions' combined with 'cache misses' to reflect the memory access pattern impact on performance). Amongst all traced metrics, we select a combination of 'IPC' and 'instructions completed' to execute the clustering analysis with. These two metrics are useful to bring insight into the overall performance of the application. Trends in the 'instructions completed' counter reflect regions with different computational complexity. In combination with 'IPC', it is possible to differentiate between regions with the same complexity but different performance.

As a result of the analysis we obtain a fine-grain characterization of the computing regions of the application, grouped by performance trends. The tool presents a numerical report with the average values for the different metrics that were used, and a scatter plot depicting the general structure of the application. Figure 2 shows an example of the structure of Gromacs [1]. It can be seen that the first three clusters (those that represent most computation time) exhibit different distributions. While cluster 2 is rather compact, clusters 1 and 3 present a significant dispersion in IPC and instructions respectively, which is a sign of unbalance.

In our current implementation, the analysis is carried out centrally at the front-end of the communication network. Nevertheless, the MRNet filters facility enables us to perform this complex computation in the intermediate communication processes of the network. In this way, data is incrementally reduced as it flows towards the front-end, which ends up receiving the results rather than a whole aggregation of inputs. We are working on a parallel version of the clustering tool that would run in these nodes in order to keep the system scalable.

As previously discussed in [7], it is advisable to apply some reductions on the amount of data to cluster in order to accelerate the process. While data is being retrieved from the tracing buffers and sent through the communication network, those CPU bursts whose duration is negligible are directly discarded. In this way, the clustering tool needs not to process irrelevant data, and the communication network is not flooded with useless traffic. Experiments in [7] show that the 99% of the total computation time is covered by only the 20% of all CPU bursts.

Even after having filtered all meaningless data, the cost of the clustering process grows more than linearly with the input size. In the applications tested (Gromacs, Specfem3D, NAS BT, etc.) more than 50,000 bursts can be generated in roughly 30 seconds on average, which can take up to 10 minutes to analyze. Among those bursts that make the cut, only a small subset is actually clustered to speed up the process. The rest is classified based on the results of a previous clustering with the selected training set. Different strategies have been tested to select a representative set:

- **Sampling across time.** 10-15% of CPU bursts of every process are randomly selected.
- **Sampling across space.** Select all CPU bursts from a few representative processes only.

In this way, a reduction of the input size to a few thousand of samples drops the analysis time to 5-10 seconds. Obviously the less samples are taken, the quickest but less precise results are produced. However, this does not imply a bad characterization of
the application at all, but just a different one. Typically, a single large cluster can split into two or more (or even disappear) because samples that would have fallen in there were not taken, and subclusters were not close enough to merge. This is still a valid clustering that shows structure, though with different fineness. Depending on the application (e.g., if it is purely SPMD or not), one approach might be more convenient than the other, and their suitability responds more to the expert’s objective than to any strict rule. In our experiments, the combination of both methods (i.e., include all bursts from a few processes and sample the rest) has proven to work fairly well for all situations, as it provides a better chance to capture both time and space variations.

Figure 3 corresponds to the same analysis than the one shown in Figure 2. Instead of having all data clustered, we took all bursts from just 8 processes (out of 64), and sampled the 15% of bursts from the rest. In comparison, 75% less data was processed (2,500 bursts down from 10,000) and the analysis finished 20 times faster (6 seconds down from 2 minutes). Although resulting clusters are less populated, and the former cluster 4 was split into two subclusters, the same overall structure can be identified.

The analysis described above is repeatedly computed whenever the application produces new volumes of performance data. Multiple clustering results are used to track the evolution of the application behavior. It is only when certain conditions are met that a representative region of the whole execution is selected to produce a full detailed trace. The following section explains this process in detail.

2.3.1 Tracking the application evolution

Whenever the application produces a new volume of performance data, equivalent in size to the amount of data specified by the user for the final trace, a subsequent clustering analysis triggers. This mechanism produces an ordered sequence of clusterings that illustrates not only the structure of the application in their respective execution regions, but its evolution over time.

By means of the study of the similarities between multiple clusterings, we aim at detecting at runtime a single region of the execution that faithfully represents the overall application behavior. We consider a representative region to be one where the application behavior is stable. By stability we understand the convergence of the algorithm into an iterative pattern over which the achieved performance presents minor fluctuations. Given the iterative nature of the vast majority of scientific codes, such a region typifies the whole execution almost in its entirety. Any performance flaw that can be detected there and optimized will result in a strong positive impact all over the execution.

The application is considered stable when several clusterings in a row are equivalent. This is to say the clusters found in the current analysis have similar shape, size and position in space than those detected in previous steps. Using the precise numbers based on previous tests experience, for every cluster we check that the values of the extremes of every dimension are within a ±5% margin of variance compared to the same dimension in the previous clusterings. In the case of using just two dimensions (‘instructions’ and ‘IPC’ in our case studies), this can be interpreted as inscribing every cluster into a rectangle and matching...
Figure 4: Timeline showing the distribution of clusters over 2 iterations of Gromacs application.

those that overlap. Two clusterings are considered equivalent if the matching clusters represent at least the 85% of the total computation time. In case the application would keep on changing behavior and the stability criterion could not be met, the requisites are gradually lowered to attempt to find the best possible region. The inability to find a stable region can also be the consequence of too small trace buffers which can not hold an entire representative region. A possible alternative is to dynamically increase their size and check whether the results improve.

The current heuristic leaves room for improvement. For instance, it could easily take into account how populated the clusters are. Non-regular shapes could be inscribed into complex geometrical figures more accurately, and image superposition techniques could be used to calculate the overlapping area between two clusters. Yet as simple as it is, it offers a quick and versatile approach to the problem that deals well with the spectrum of scientific applications, whose behavior does not tend to radically change all of a sudden.

Once a stable region has been detected, clustering results are transferred back to the back-end threads, and every CPU burst is labeled with the cluster to whom it belongs to. Along with the clusters distribution, all performance data within the same time interval is flushed from the tracing buffers in order to produce a detailed trace of that region. Figure 4 depicts a view of 2 iterations of Gromacs showing the distribution of clusters over time (clusters colors match with the scatter plot Figure 3).

In addition, we obtain periodic reports and plots that help the analyst to understand the evolution of the application until then. At this moment, the system does not necessarily have to stop, but can work the other way round. Taking the stable region as representative, subsequent clusterings can be compared looking for significant differences in structure. This might be a reflection of the application entering a different computation phase or undergoing significant perturbations, and detailed snapshots of these regions can be stored along with the trace.

3 Conclusions and future work

This paper presents an on-line analysis framework based on clustering techniques that is able to minimize the amount of data emitted to the trace while maximizing the amount of relevant information presented to the analyst. This proposal tackles the trace scalability problem and allows tracing to be used even in large-scale scenarios.

Performance measurements collected by a tracing facility are periodically aggregated through a software communication network upon a master process, where they are analyzed using clustering techniques so as to detect the application’s structure at run-time. The combined use of these tools enables us to produce a compact full detailed trace of a representative region of the whole execution that can be analyzed afterward with Paraver. In addition, periodic performance reports and trend plots are produced, showing the structure of the application at the respective time intervals and its evolution over time.

Unlike other approaches, we do not perform an on-line analysis that tries to answer why the application behaves as it does. Instead, we produce a fully detailed yet small trace, so that the analyst can conduct the analysis not being limited by predefined knowledge-based rules. Although we detect iterative patterns, our objective is not to find and filter periods for a mere data compression. We do actually identify the application structure as it runs and decide which are the most interesting regions to present to the analyst.

Although the work presented relies on clustering, the underlying infrastructure enables the use of other analysis mechanisms towards the intelligent selection of performance data. A downside of this approach
is that there is not precise control over which part of the code corresponds to the traced region. We are incorporating a mechanism based on spectral analysis techniques [6] to precisely correlate them and delimit more accurately the traced regions.

For the moment, we are taking advantage of the MRNet filters facilities to perform rather simple data reductions to keep the system scalable, but they offer a far greater potency. To make the most of it, we plan to develop a parallel version of the clustering algorithm, that would run on the intermediate processes of the communication network, so as to perform the analysis incrementally while data traverses the network upon its way to the master process.

References


