Automatic Phase Detection and Structure Extraction of MPI Applications

Article *in* International Journal of High Performance Computing Applications · August 2010

DOI: 10.1177/1094342009360039 · Source: DBLP

3 authors:

Marc Casas
Barcelona Supercomputing Center
75 PUBLICATIONS **643** CITATIONS
SEE PROFILE

Rosa M. Badia
Barcelona Supercomputing Center
341 PUBLICATIONS **6,260** CITATIONS
SEE PROFILE

Jesús Labarta
Barcelona Supercomputing Center
478 PUBLICATIONS **8,851** CITATIONS
SEE PROFILE

Automatic Phase Detection and Structure Extraction of MPI Applications

Marc Casas, Rosa M. Badia and Jesús Labarta

International Journal of High Performance Computing Applications 2010 24: 335 originally published online 2 February 2010
DOI: 10.1177/1094342009360039

AUTOMATIC PHASE DETECTION AND STRUCTURE EXTRACTION OF MPI APPLICATIONS

Marc Casas[1]
Rosa M. Badia[1, 2]
Jesús Labarta[1]

Abstract

In this paper we present an automatic system able to detect the internal structure of executions of high-performance computing applications. This automatic system is able to rule out non-significant regions of executions, to detect redundancies, and, finally, to select small but significant execution regions. This automatic detection process is based on spectral analysis (wavelet transform, Fourier transform, etc.) and works detecting the most important frequencies of the application's execution. These main frequencies are strongly related to the internal loops of the application's source code. The automatic detection of small but significant execution regions shown in the paper reduces the load of the performance analysis process remarkably.

Key words: message passing interface (MPI), signal processing, spectral analysis, performance analysis

1 Applications of Structure Extraction in Performance Analysis

In recent years, parallel platforms have vastly increased in performance and in the number of nodes and processors. For example, the most recent supercomputers facilities have more than 100,000 processors and many of them have a performance peak near or higher than the megaflop barrier[1]. Thus, the study of the execution of applications on these platforms has involved increasingly hard and tedious work. A complete timetabled sequence of events of an application, that is, a traceable of the whole application, results in a huge file (10–20 GB). It is impossible to handle this amount of data with tools such as Parader (Labarta et al., 1996). Also, it is often the case that some parts of the trace are perturbed, and the analysis of these parts can be misleading. A third problem is the identification of the most representative regions of the traceable. To reduce the size of traceries, the process of application tracing must be carefully controlled, enabling the tracing in the interesting parts of the application and disabling otherwise. The number of events of the traceable (hardware counters, instrumented routines, etc.) must be limited. This process is long and tedious and requires knowledge of the source code of the application.

For these reasons, several authors (Shende and Malony, 2006; Vetter and Worley, 2002) believe that the development and utilization of trace-based techniques has several flaws. However, techniques based on traceries allow a very detailed study of the variations on space (set of processes) and time that could crucially affect the performance of the application. Therefore, there is a need to develop techniques that allow us to handle large event traces.

Our approach is to start from very large traceries of the whole application, allowing simple tracing methodologies, and then analyze them automatically. The underlying philosophy is to use resources that are generally available (disk, CPU, etc.) in order to avoid the use of an expensive resource, namely analyst time. The tool we have developed first warns the analyst about those parts of the trace perturbed by an external factor not related to the application or to the machine itself, and subsequently gives a description of the internal structure of the application, which leads to the identification and extraction of the most relevant parts of the trace. Third, our automatic sys-

[1] BARCELONA SUPERCOMPUTERS CENTER (BSC), TECHNICAL UNIVERSITY OF CATALONIA (UPC), BARCELONA, SPAIN ({MARC.CASAS, ROSA.M.BADIA, JESUS.LABARTA}@BSC.ES)

[2] ARTIFICIAL INTELLIGENCE RESEARCH INSTITUTE (IIIA), SPANISH NATIONAL RESEARCH COUNCIL (CSIC), BARCELONA, SPAIN

The International Journal of High Performance Computing Applications, Volume 24, No. 3, Fall 2010, pp. 335–360
DOI: 10.1177/1094342009360039

Figures 5, 6, 9, 11–16 appear in color online: http://hpc.sagepub.com

tem can be applied not only on off-line traceries, but also on data extracted on-line from an execution of a parallel application.

Our tool is useful for people who work with supercomputers without having a computer science background (chemists, biologists, etc.). Thanks to the tool, these people can get some insights about the performance of scientific applications (reducing the traceable and visualizing it with Parader) without any idea about the source code. Second, it is also useful for experienced supercomputers researchers who have to deal with many scientific applications every day. Most experienced researchers are able to find the main loop of the source code, instrument it, and obtain a traceable of several iterations of this loop. However, if they can obtain this traceable automatically they will have more time to study more interesting issues.

The first step of our approach is to characterize the execution of the application from a particular point of view. There are many approaches to the characterization of the execution of applications, but in our work we use signals to do this. These signals are generated from data contained in the traceable. The information contained in them is the time evolution of a given metric during the execution of the application. For example, we generate a signal that contains how many processes are executing a MPI[2] call in an instance of time. It is possible to generate this kind of signal because traceries are timetabled sequences of events. The characterization of executions of applications is done using signals for several reasons. The first reason is that it is possible to keep the highly detailed information contained in traceries of the variations on space (set of processes) and time. Second, we use signals because it is possible to isolate a given performance metric and perform an analysis based only on it and, therefore, determine the impact of this parameter over the whole execution. Third, the algorithms based on signal processing theory have a low computational complexity, are able to automatically provide relevant information and, finally, are based on a solid and powerful mathematical theory.

We use signal processing techniques (wavelet transform, cross-correlation, morphological filters, etc.) in order to provide a very fast, automatic detection of the phases of MPI application execution. These signal processing techniques are interesting for our analysis because they have several important properties which allow us to build an automatic system based on them. First, morphological filters derived from the theory of mathematical morphology provide a signal processing framework which is able to detect regions that are affected by corrupting factors. Second, wavelet analysis provides a classification of the physical domains of signals according to the occurrence of frequencies on them, that is, the criterion of wavelet transform in order to perform the analysis is to separate regions according to their frequency behavior, i.e. a region with a small iteration which is repeated many times is separated from another region with no periodic behavior. Third, autocorrelation provides us with a means of detecting the value of the iterative period within the regions with strong periodical behavior, that is, it gives the exact value of the time span of the main iterative periods within the periodical regions, which have been detected by wavelet transform. Fourth, cross-correlation makes it possible to detect which of the iterations are the most representative of the whole periodic region. The criterion of cross-correlation to select the most representative iterations is to look for the iterations closest to a perfect periodical signal.

The rest of the paper describes how we apply these techniques to the automatic analysis to find a traceable structure.

In Section 2 we discuss related studies which have been carried out in the last few years. In Section 3 we explain the different metrics we have used and we give some examples. In Section 4, we describe the typical execution phases of high-performance computing applications and we explain how we automatically detect them. In Section 5, we discuss several factors which can corrupt the information contained in traceries. In Section 6, we describe the typical iterative structure within the computing phase of high-performance computing applications and we explain the methodology we use to detect it. Finally, in Section 7 we evaluate our automatic methodology.

2 Related Work

There are various approaches to traceries which attempt to either avoid or to handle large event traces. KOJAK (Mohr and Wolf, 2003) is a tool for the automatic detection of performance bottlenecks. It looks for patterns representing inefficient behavior and quantifies its influence over execution. It focuses on studying every communication contained in the traceable locally. However, KOJAK does not distinguish between execution phases because it does not look for the internal traceable structure. For that reason, it needs large traces in order to amortize the initialization phase or the cost of perturbations. Our automatic system can be used to prepare data that will be given to KOJAK as input. This previous step will help it to focus on the most relevant regions in the traceable.

The DeWiz performance tool (Kranzlmuller et al., 2003) performs an automatic analysis. The idea of DeWiz stems from the fact that most tools rely on graph-based analysis methods or use space–time diagrams for visualization of program behavior. As a result, a unified directed graph may be used to capture different program properties and to represent the basis for analysis activities. This analysis is focused, on the one hand, on the detection of communication failures by pairwise analysis of communication

patterns and, on the other hand, DeWiz identifies repeated patterns in the event graph. It is in this second point that the DeWiz approach and our own is similar. However, DeWiz is unable to find structure on the temporal evolution of the application's execution.

These two tools, KOJAK and DeWiz, also try to distribute data processing in order to analyze large traceries. A parallel extension of KOJAK, SCALASCA (Geimer et al., 2008), has been developed and DeWiz also has a modular design which enables it to be executed on distributed computing infrastructures.

The VAMPIR Next Generation tool (VNG) (Knuepfer et al., 2005; Brunst et al., 2004), an extension of the visualization tool Vampir (Nagel et al., 1996), does not perform an automatic analysis but tries to explore, in a similar manner to SCALASCA and DeWiz, distributed data processing. Furthermore, VNG is composed of a parallel analysis server and a visualization client, where each analysis is executed on a different platform. Another important VNG feature is the utilization of a data structure called a Complete Call Graph (CCG), which holds the full event stream, including time information, in a tree.

Previous studies have used signal processing techniques in order to detect program phases. Shen et al. (2004) used wavelets as a time–frequency analysis method to identify behavior changes on the locality of a program. Next, phase markers are inserted into the program using binary rewriting. When the execution of the instrumented program begins, the first few executions of a phase are used to predict all subsequent executions. This approach uses signal processing in order to predict the locality of a program and, for that reason, it is different from ours. In contrast, Huffmire and Sherwood (2006) use wavelet analysis to perform phase analysis. Wavelet coefficients are used as a similarity metric and *K*-means clustering is applied on these coefficients. The goal of this work is to capture the memory bus behavior of commercial applications. This approach is different to ours because, unlike our approach, it does not use wavelets as a time–frequency analysis method. Gamblin et al. (2008) used wavelets to reduce the amount of data that has to be analyzed to study the load imbalance on high-performance computing applications. This approach achieves several orders of magnitude of data reduction using compression techniques from signal processing and image analysis. In addition, low-error and high-speed reductions are achieved by the approach. The main difference between this work and ours is that we use the information obtained from the wavelet transform to make an analysis about the general structure of executions while, in this work, the wavelet transform is just used to compress data.

Paradyn (Roth and Miller, 2006) is a non-trace-based performance tool which analyzes data in real time, i.e. program instrumentation and performance evaluation are done during the execution. It performs a search which is focused on answering three questions:

1. Why is the application performing poorly?
2. Where is the performance problem?
3. When does the problem occur?

To answer the first question, the system includes hypotheses about potential performance problems in parallel programs. It collects performance data to test whether these problems exist in the program. In answering the second question, it isolates a performance problem to a specific program resource (e.g. a disk system, a synchronization variable, or a procedure). To identify when a problem occurs, it tries to isolate a problem to a specific phase of the program's execution. Finding a performance problem is an iterative process of refining the answers to these three questions.

Finally, mention should be made of Periscope (Gerndt and Kereku, 2007), an automatic performance analysis tool for large-scale parallel systems. Periscope applies an automatic distributed online search in order to detect performance bottlenecks. It consists of a graphical user interface, a hierarchy of analysis agents, and two separate monitoring systems. The graphical interface allows the analyst to begin the analysis process and to study the results. The agent hierarchy performs the actual analysis. The node agents autonomously search for performance problems which have been specified previously. Typically, a node agent is started on each node of the target machine. It is responsible for the processes and threads on that node. Detected performance problems are reported to the master agent.

Several of the approaches discussed above perform an automatic performance analysis. However, most of them do not look for the internal structure of the traceable (and thus they need large traces to amortize the cost of the initialization phases if they want to extract global values of representative metrics), and only a few look for repeated patterns not based on temporal evolution of the application's execution. Thus, we believe that our approach satisfies the need for an automatic performance tool based on the structure of the temporal evolution of the application. While other approaches have attempted to overcome the increasing size of traceries in recent years using distributed computing, our work tries to overcome that problem by using a more analytical methodology and a more sophisticated power analysis. Furthermore, our methodology can be parallelized.

3 Metrics Generation

As we have said above, our approach consists of characterizing executions of parallel applications using signals. The main reasons are that, first, signals keep the highly

detailed information contained in traceries, second, signals make it possible to isolate a given performance metric, and, third, the algorithms based on signal processing theory have a low computational complexity.

In order to summarize important aspects of parallel program performance, we derive signals from data contained in the traceable, capturing the temporal application behavior from a particular point of view. However, these signals can be generated from any kind of timetabled data. For this reason, our approach is not limited to work with traceries. Each metric we generate summarizes one aspect of the parallel program's execution. In this section, we discuss a set of metrics. Some of the metrics put more emphasis on the execution phases than on those associated with communication, while others put more emphasis on communication. The automatic system takes these metrics as input and processes them. Depending on the situation, some metrics show phases of the program's execution more clearly than others.

3.1 Number of Processes Computing

This metric shows the number of processes which are performing computations at each instance of time. It indicates the location of intensive computing regions. Typically, its lower values are located on instances of time when the execution is performing communications. Otherwise, when the execution is performing the main computations, the value of the signal will reach its highest values. Another important qualitative characteristic of this metric is its low variance, explained by the fact that it can only reach a finite set of values, from 0 to P, where P is the number of processes. It is generated from the running states contained in the traceable. For each thread, we generate a signal which contains, for each instance of time, t, value 0 if the thread is not in a running state and 1 otherwise. Finally, signals generated by threads are added, obtaining this metric.

3.2 Sum of Durations of Computing Bursts

This metric shows the instantaneous sum of computing bursts. It will reach its lower values when the execution is performing a few short calculations. Otherwise, high values will appear when a lot of long bursts are being computed by the processors. For this reason, this metric summarizes the kind of calculations we find in execution. Finally, the variance of the metric will be higher than that of the number of processes computing metric because it can reach a wide set of parameters, depending on the duration of running bursts.

For each thread, we generate a signal which contains, for each instance of time t, the duration of the running burst that is being executed. If there is no running burst in this moment, i.e. the process is inside a MPI call, the value assigned to t is 0. Finally, the signals generated for all of the threads are added. Focusing on point-to-point MPI calls provides a way of looking at the application's parallel structure from the point of view of communications between processes. This approach complements the metrics explained above because it takes into account communications performed by the application.

3.3 Number of Point-to-Point MPI Calls

This metric shows the number of processes which are executing a point-to-point MPI call at a given moment. Therefore, the range of its values is from 0 to P, P being the total number of processors. It reaches its minimum values when the application is in a calculation phase because these phases have very few communications. On the other hand, the maximum values of this metric are reached when the occupation of the interconnection network is at its maximum.

To generate this signal, we have to cover all of the traceable and every time we find the beginning of a MPI call we add 1 to the signal. When we find the end of a MPI call, we subtract 1 from the signal. Obviously, the value of the signal at the beginning of the execution is 0.

3.4 Number of Collective MPI Calls

This metric shows the number of processes which are executing a collective MPI call. The range of its values is from 0 to P, P being the total number of processors. Typically, the execution of applications implies a lot of collective calls at the initialization phase in order to provide the information to start the computations. However, in computation phases we can also find collective calls. The difference is that, on the one hand, the collective calls in the initialization phase do not appear following a periodic pattern and, on the other hand, the collective calls on the computation phase have a periodic pattern.

3.5 Instructions per Cycle

This metric shows the average rate between instructions and cycles of the set of processes in a given instance of time. These values are extracted from the hardware counters contained in the traceable. The values of these counters are extracted at the beginning and at the end of computing bursts. For this reason, the granularity of the signal will be equal to the granularity of the signals derived directly from the computing bursts.

3.6 Communications

This metric shows the number of outstanding messages at a given moment. It is very close to the number of point-

to-point MPI calls, but is not exactly the same for two reasons: first, because the transmission of a messages involves two MPI calls; and, second, because the execution of a MPI send or receive does not imply an instantaneous transmission of the message. Typically, if the intercommunication network is under a heavy load these two metrics will show different behavior because the MPI calls need to wait in order to send or receive the message. However, if the network is not **busy**, the behavior of these two metrics is closer.

3.7 Examples

In Figure 1, we show examples of the metrics we referred to above in this section. In this figure we show, for each instance of time of the execution, the value of the metrics. The figures were obtained from a traceable that was generated executing WRF-NMM[3] application with 128 processors in the MareNostrum Supercomputer. At the top of the figure we can see two signals: the sum of durations of computing bursts, expressed in milliseconds, and the number of processes computing. In the middle, there are two more metrics: The number of point-to-point MPI calls and the number of collectives. Finally, at the bottom, we have the instructions per cycle (IPC) and the total number of flying communications.

It is important to state that we see the same kind of behavior in all of the signals. In general, we see an initial phase of low activity, a phase in the middle with strong activity and, finally, a phase with low activity. The exception to this behavior is the collective communications signal. In this signal, we see a strong activity at the beginning of the execution. The reasons for the common behavior of the signals and the exception of the collectives are explained in the next section.

4 Program Phases

Typically, high-performance computing applications radically change their behavior when executed on parallel architectures. For example, in the case of MPI applications, there is little in common between the interval of time in which a MPI collective call is being performed and the interval of time when all of the processors are performing computations. The first is characterized by an intense utilization of the interconnection network and the lack of computations while the second is characterized by great activity focused on computations and a lack of communication between processes. This is only a very simple example of the wide diversity of situations that can be found within the execution of a high-performance computing application. In the next section, we explain the typical structure of a high-performance computing application and how we detect it.

4.1 Description

The execution of scientific applications has several well-defined phases. Typically, the execution has an initialization phase, which consists of the initial communications in order to assign initial values to the variables, domain decomposition, etc. This initialization phase is often not targeted by the analysis since the high cost is typically amortized over a long run. The performance of this initialization phase is not related to the core scientific computation we want to study here. Also, many applications show a final phase, which consists, mainly, of output data operations. As with the initialization phase, this final phase often not targeted in studies of parallelization efficiency.

The core phase that interests us, and that is found in scientific parallel applications, is the computation phase. This phase is characterized by sets of computing bursts alternated by communications, and typically has a periodic behavior prompted by the application's structure, which performs iterations on space (physical domain of the problem) and time. Thus, there is a pattern that we can find repeated many times during this computation phase. Furthermore, it is in this computation phase when the application shows its parallel efficiency, and it is in this phase where it is interesting to study whether the application has a good parallelization or not, whether the load is balanced optimally, etc. Consequently, the parallelization study of the application has to be made here. Although this is a common behavior found in the execution of many scientific applications, it is possible to find different behaviors. Nevertheless, the most important issue for us is whether our automatic system will always be able to detect the execution phases.

In the signals depicted in Figure 1 we see this behavior clearly. The case of the collectives signals is particularly interesting because it shows the strong collective communication pattern in the initialization phase in order to provide the tasks with the initial values to start the computations. Furthermore, the other signals have the typical behavior showing strong activity in the computation phase.

4.2 Overview of Wavelet Transform

We use signal processing techniques to automatically detect periodic phases. The discrete wavelet transform (DWT) is applied to the signal obtained from the *application's execution*. The most important feature of DWT is that it captures information not only about the values of the frequencies which the input signal contains but also about the physical location of these frequencies, that is, using wavelet transform it is possible to know estimations of the values of the main frequencies of the signal and the places within the input signal's domain where these frequencies occur. Remember that the traditional

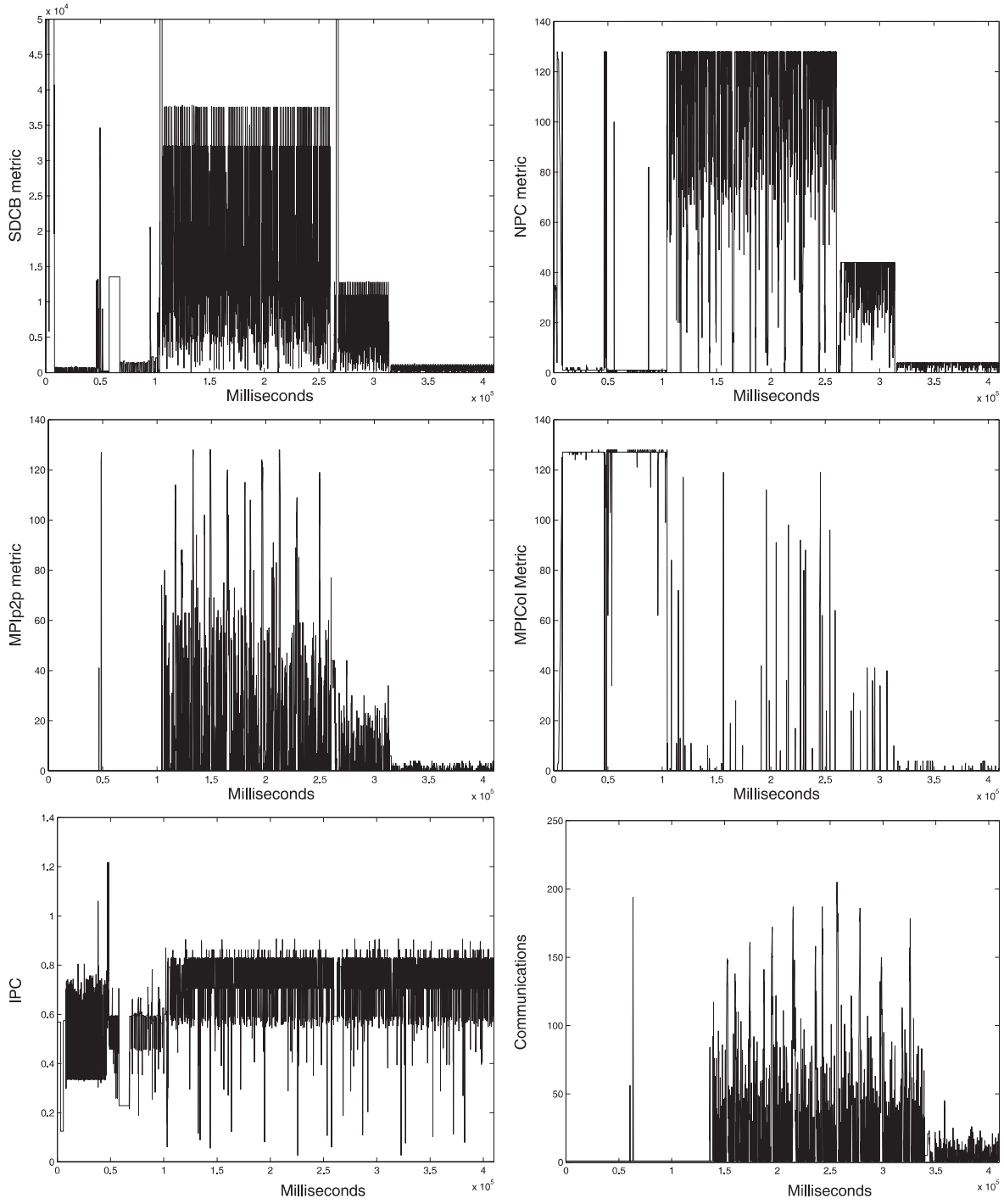

Fig. 1 Visualization of the metrics explained in Section 3. At the top, we show the sum of duration of computing bursts and the number of processes computing. In the middle, we show the number of point-to-point MPI calls and the number of collective calls. Finally, at the bottom, we see the IPC and the number of communications. These metrics have been generated from a traceable of an execution of WRF-NMM application on 128 processors.

Fourier transform does not give information about the physical location of the frequencies, it only gives the spectrum of frequencies as output.

To obtain the DWT of a signal, an algorithm called fast wavelet transform was developed. It has its theoretical basis in functional analysis (Daubechies, 1992; Walnut, 2004). It is recursive and based on two filters: a low-pass filter and a high-pass filter. Each filter is characterized by a set of coefficients: the set $\{a_d\}$ characterizes the low-pass filter and $\{b_d\}$ the high-pass filter. The values of these coefficients are derived from a function $\phi \in L_2(\mathbb{R})$, called scaling function. Each scaling function defines a set of two filters that make possible a quick calculation of the DWT. However, not all of the functions in $L_2(\mathbb{R})$ are able to provide a multi-resolution analysis based on DWT. There are several conditions, described in Kaiser (1994), that the scaling function has to fulfill. In this work, we use the Haart's scaling function (Daubechies, 1992), used widely in signal processing.

DWT is an operation that transforms a set of $N = 2^J$ coefficients, $s_0, s_1, \ldots, s_{N-1}$ into two sets of $N/2$ samples. Each recursive application of the algorithm is called level. The coefficients used as input to the transform are expressed as $s_0^0, s_1^0, \ldots, s_{N-1}^0$. In general, level l coefficients, $s_0^l, s_1^l, \ldots, s_{n-1}^l$ and $d_0^l, d_1^l, \ldots, d_{n-1}^l$, are obtained from level $l-1$ coefficients $s_0^{l-1}, s_1^{l-1}, \ldots, s_{2n-1}^{l-1}$ using two recurrence relations.

When the execution of the fast wavelet transform is finished, we obtain a set of coefficients. First, $N/2$ values of the level 1, $s_0^1, s_1^1, \ldots, s_{N/2-1}^1$, subsequently, $N/4$ coefficients of the level 2, $s_0^2, s_1^2, \ldots, s_{N/4-1}^2$, etc., until two coefficients of level J, s_0^J, d_0^J. According to the theory, the coefficients of level i contain information about frequencies between $[\alpha/2^{i+1}, \alpha/2^i]$ where $1 \le i \le J$ and α is the sampling frequency of the input signal. Note that the higher frequency considered by this multi-resolution scheme is $\alpha/2$. This fact is coherent with the Nyquist–Shannon sampling theorem which states that it is not possible to obtain information about frequencies higher than $\alpha/2$ if the signal's sampling frequency is α.

Since we are interested in the detection of the highest frequencies to identify the computation phase, we base our detection on the 2^{i-1} coefficients of the frequencies belonging to $[\alpha/4, \alpha/2]$. It is important to state that the accuracy of the physical location of frequencies belonging in $[\alpha/4, \alpha/2]$ is the best because while DWT is very good at capturing high-frequency physical locations, it fails to accurately capture their values. In addition, in the case of low frequencies, while it captures the values very well, it fails on the physical locations identification.

If the information contained in level 1 coefficients is not enough to detect significant high-frequency regions, we use the 2^{i-2} level 2 coefficients. Remember that they contain information about frequencies belonging to $[\alpha/8, \alpha/4]$. If in this level there is no significant information, we can use level 3 coefficients, and so on. Remember that we have J levels of coefficients, where J is an initial parameter.

DWT has several desirable properties. First, it can be computed in $O(n)$ operations. Second, it captures not only the values of the frequencies of the input signal but also the physical location where these frequencies occur. These two properties are very useful because they allow us to find different execution phases given a traceable with only $O(n)$ operations.

If the execution that we are analyzing has a periodic structure, the periodic phase will be detected because it has strong high-frequency behavior. If the execution contains multiple periodic phases, we will also detect them, if these multiple periodic phases have high-frequency behavior.

4.3 Detection

As explained in the previous section, the initialization and final phases strongly depend on the input/output system of the machine, the interconnection network, the operating system, and many other factors, they do not represent the application. These considerations explain why we are not so interested in these phases and why we concentrate on the detection of the computation phase in order to guide and focus performance analysis.

The computation phase typically comprises computation bursts followed by communications. Typically, it has periodical behavior; that is, there is an iterative pattern repeated many times. For all of these reasons, we can expect the highest frequencies of the signals to appear in the computation phase. Hence, our criterion for automatically detecting the computation phase of applications is the region where the highest frequencies occur.

The tool used to detect the higher frequencies is wavelet transform, several generalities of which were explained above in Section 4.2. As has been explained, wavelet transform provides a multi-resolution analysis able to identify the locations where the frequencies occur. Given a signal, assume that we sample it in $N = 2^i$ samples, using a sampling frequency α. As was explained in Section 4.2, we obtain from wavelet transform 2^{i-1} coefficients of the locations of frequencies belonging to $[\alpha/4, \alpha/2]$. In general, wavelet transform gives 2^{i-k} coefficients of the locations of frequencies belonging to $[\alpha/2^{k+1}, \alpha/2^k$. Remember that, on the one hand, we are interested essentially in the detection of the highest frequency in order to detect the computation phase and, on the other hand, the precision of the location is best for the highest frequencies. For these reasons, we base our detection on the 2^{i-1} coefficients (level 1) of the fre-

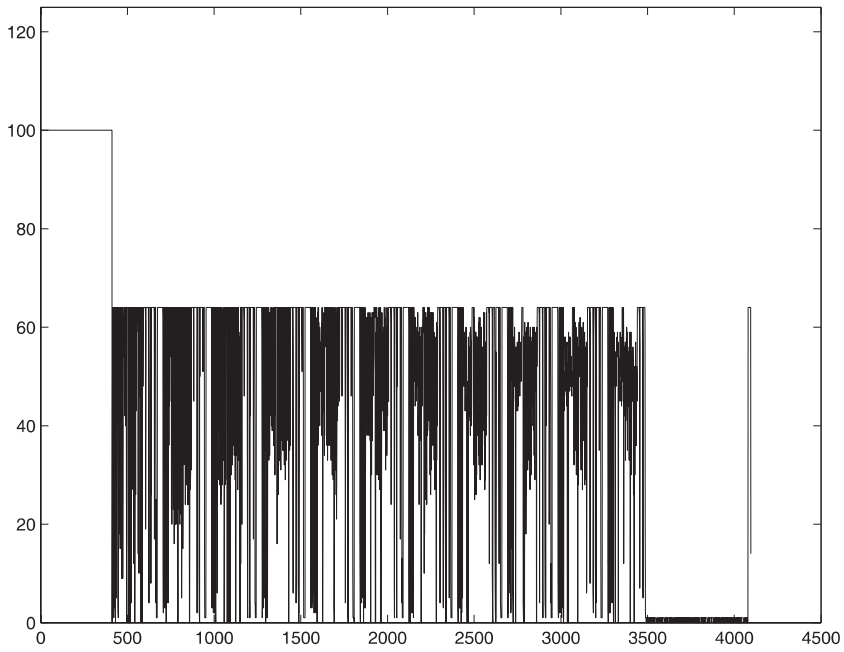

Fig. 2 Input signal. Sampling frequency = α.

quencies belonging to [α/4, α/2]. If at this level there is no significant information, we can use the 2^{i-2} coefficients (level 2) whose frequencies belong to [α/8, α/8], and so on. Remember that we have *i* levels of coefficients, where *i* is an initial parameter.

Once the wavelet transform has provided us with the coefficients, we make a selection of the high-frequency regions based on two parameters: λ and δ. The first represents the threshold from which we select the coefficients, while the second represents the granularity of the selection. More precisely, λ is a number between 0 and 1 that defines the minimum of the selected coefficients. It is defined in terms of the maximum, that is, λ = 0.3 means that we select all of the coefficients equal or greater than the 30% of the maximum coefficient. The second number, δ, says, given a selected coefficient due to λ's criterion, how many neighbor coefficients we select. For example, if λ = 0.3 and δ = 10, we select all of the coefficients equal to or greater than 30% of the maximum and, after that, for each selected coefficient, we take 10 coefficients from the right neighborhood and 10 more from the left neighborhood.

Figure 2 gives an example of a signal. It is clear that this signal has three very defined phases: first, a phase with low-frequency behavior, then, a second phase with high-frequency behavior, and, finally, a low-frequency phase. In Figure 3 we can see how wavelet transform shows the presence of high frequencies. Finally, in Figure 4 we show the selection we made from the wavelet transform. The parameters are λ = 0.3 and δ = 10.

5 Corrupting Factors

By corrupted regions we mean those regions affected by a corrupting factor, that is, regions with distorted relative timing behavior. See Mohr and Traff (2003) for a detailed discussion of these distortions. Furthermore, all of these perturbations have a common characteristic: they are neither caused by the application nor by the architecture. They are caused by external factors such as tracing packages, unknown system activity, etc. Different phenomena or metrics can be identified as being the cause of a significant perturbation of the program behavior. An example of these phenomena is flushing, which is caused by the fact that tracing packages keep individual records in a buffer in memory during the tracing process. The problem is that when the buffer is full, these records have to be flushed to disk. This flushing takes so long that it affects the execution and the statistics derived from the traceable. Also, flushing does not appear simultaneously in all the processes, although it is typical that the flushing of different processes occur in bursts. Other corrupting factors are preemptions, problems with the communication network of the machine, etc. In the next section, we describe the main corrupting factors.

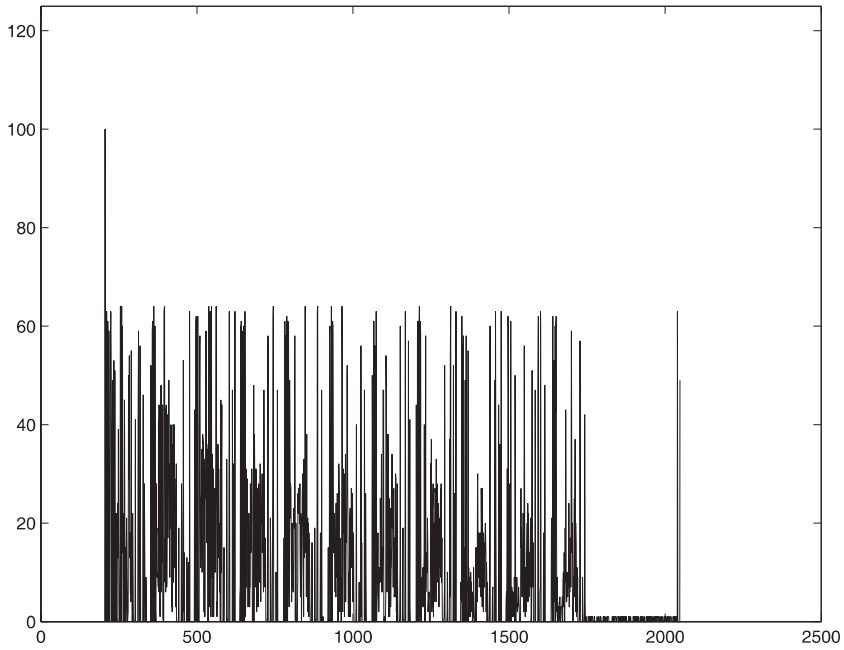

Fig. 3 Wavelet output. Location of frequencies between [α/4, α/2].

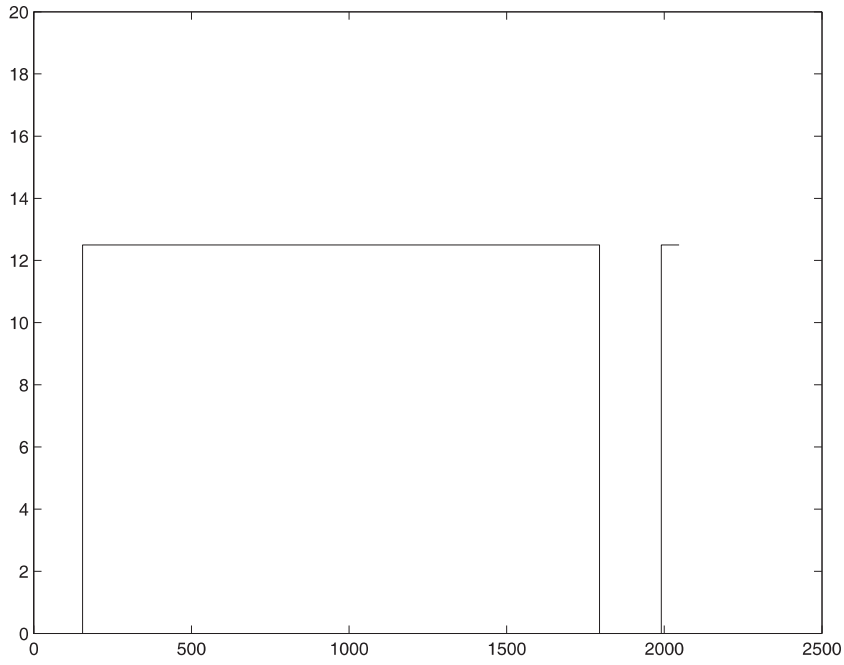

Fig. 4 Selection.

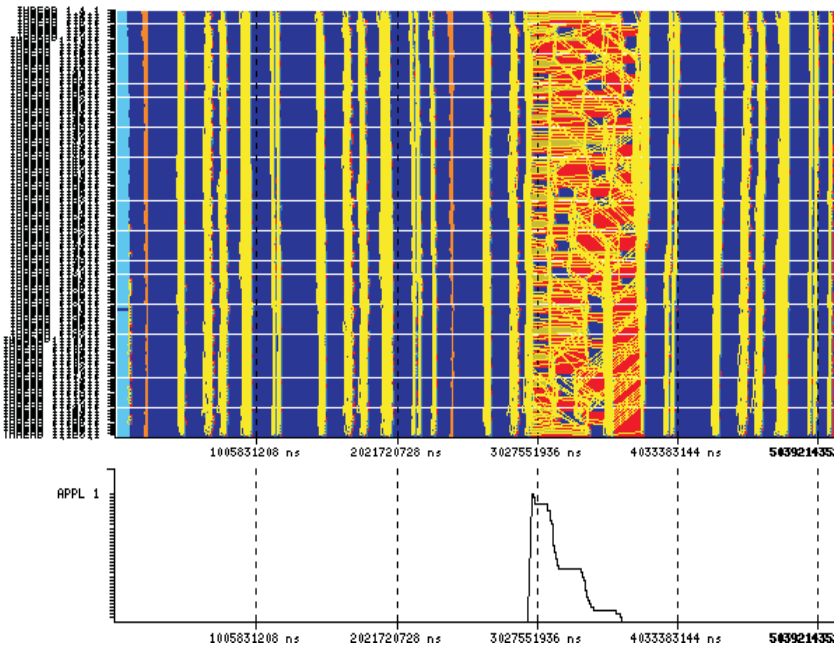

Fig. 5 An example of flushing. Processes are flushing records to disk.

5.1 Description

Different phenomena during execution can be identified as being the cause of a significant perturbation of program behavior. In the following we discuss three of these phenomena.

5.1.1 Flushing Tracing packages[4] keep individual records in a buffer in memory during the tracing process. The problem is that when the buffer is full, the records will have to be flushed to disk. This flushing takes so long that it can affect execution. Furthermore, it can affect other processes as well as the process that is currently flushing, thus generating further delays. If we do not rule out these delays, the statistics will be notably affected. Another important property of flushing is that it does not have to appear simultaneously in all processes, even if, in many applications, flushing of different processes typically occurs in bursts. We can derive this metric from the flushing events contained in the traceable. In Figure 5 we give an example. We can see the impact of flushing records to disk on normal execution behavior. The signal indicates how many processes are flushing data to disk. The application is WRF-NMM[5] executed on 128 processes.

5.1.2 Preemptions Preemptions by daemons or other system activity are another cause of performance degradation in large-scale applications. Identifying preemptions is important in order to calculate their impact and to try to reduce it. Otherwise, if the analyst suspects that a given traceable is affected by preemptions, they will drop all of the file. However, there may be parts of the trace without perturbation and, therefore, good for the analysis. In conclusion, we can say that the identification of regions with preemptions allows the analyst to improve the confidence of the tests. An easy way to identify regions with preemptions is based on the cycles counter available in the trace. The ratio of cycles to time should be equal to the processor clock frequency. If we obtain a value less than the clock frequency, we will infer that for some time the process did not get CPU and, therefore, the cycles counter did not increase.

5.1.3 Clogged System This term is applied to the regions with a large number of messages in transit while a low system bandwidth is achieved. There are several interpretations of this behavior. It may be caused by the preemption of one thread that plays a fundamental role in the critical path of the region where the behavior takes place. Another possible reason is that another system

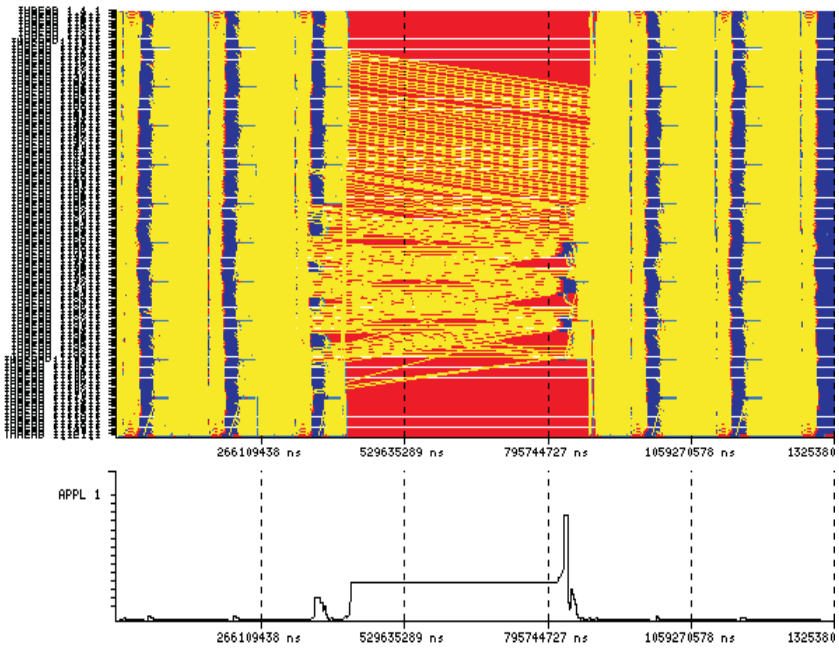

Fig. 6 An example of a clogged system. Many communications are being transmitted with an extremely low bandwidth.

activity stole bandwidth while the execution of the application was running. As we have said, the metric that will identify this behavior is the ratio between the messages in transit and the system bandwidth achieved at a given moment. This metric can be easily derived from the records contained in the traceable. Figure 6 gives an example. The application is NAS-SP (Bailey et al., 1994) executed on 121 processors. At the top we show a typical Parader visualization. In it we can see initially normal execution behavior followed by a region where the transmission of messages is extremely low. Finally, normal execution behavior is resumed by the application. At the bottom, we can see the ratio between the messages in transit and the system bandwidth. This metric is able to characterize the communication problems that the execution is suffering.

5.2 Detection

In this section, we discuss how we detect the corrupting factors discussed above. The phenomenon of flushing is characterized by a signal indicating for each instance of time the number of processors flushing to disk. We derive this signal from a traceable that contains flushing events, indicating when each process starts and finishes the flushing to disk. Figure 7 shows an example of a flushing signal. In this kind of signal we frequently observe interleaved small bursts with flushing peaks and periods without flushing. The traceable is perturbed not only during flushing but also at instances right after flushing peaks. Therefore, we want to consider the bursts of flushing as a single perturbed region. With this objective, we use a set of morphological filters, defined in the context of mathematical morphology. These filters are non-linear and are based on the minimum and maximum operations, aiming at the study of structural properties of the signal. The two basic morphological filters are erosion and dilation. The first has the property of eroding those regions of the signal with values different to zero. The second filter has the property of dilating the regions of the signal different to zero. Both operators have an associated width that has to be specified before the filter is applied. If we combine the two operators performing a dilation followed by an erosion we obtain an interesting result: first, the dilation will merge the small regions with their larger or nearby neighbors and, after that, the erosion will allow us to return towards the initial signal, except in the cases that two different regions have been merged by the dilation. With this combination, we obtain a new morphological operator called closing. Figure 8 shows the result of perform-

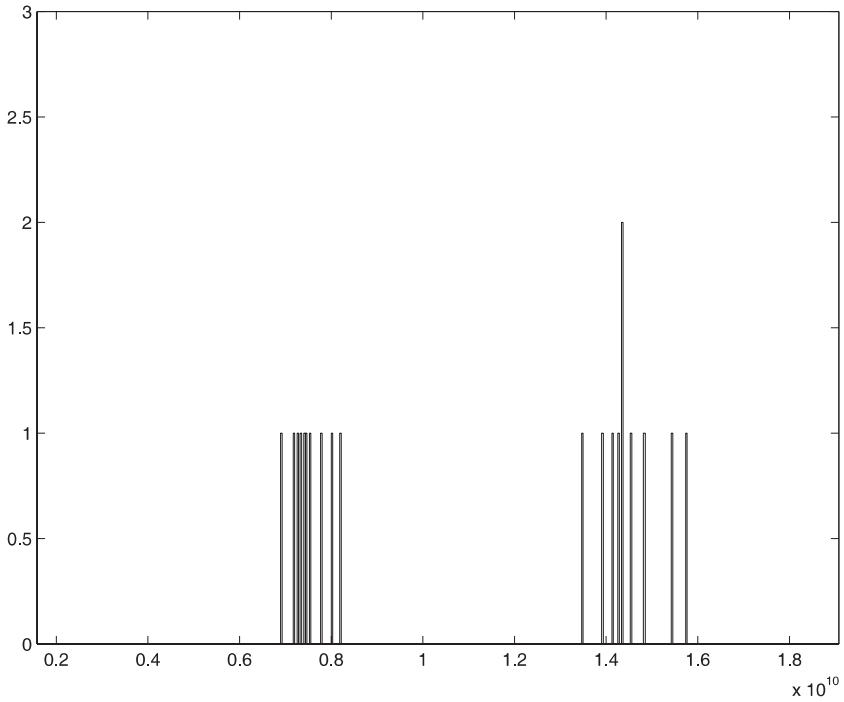

Fig. 7 Signal.

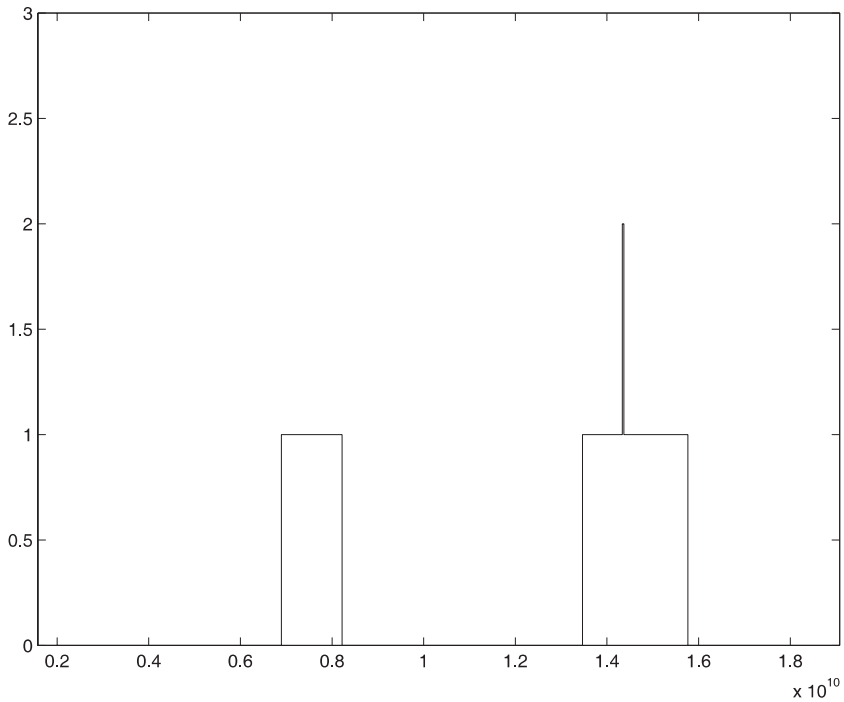

Fig. 8 Closing of the signal.

ing a closing on the signal represented in Figure 7. Note that the small regions that appear in Figure 7 have been merged into a larger region. This scheme is specially useful in analyzing flushing signals because it merges several close regions where some processes are flushing data to disk. As we have said, the execution is perturbed not only during flushing but also at instances right after flushing. The closing operator merges the bursts of flushing into single perturbed regions.

Preemptions are characterized by a signal derived from the cycles counter. The signal shows the ratio of the cycles counter to time. It should be equal to the processor clock frequency. Typically, this signal has a regular behavior and if it has a value less than the clock frequency during a significant span of time, we will infer that for some time the process did not get CPU and, therefore, the cycles counter did not increase. To automatically detect regions perturbed by preemptions, we want to eliminate small oscillations of the signal, because they are not significant, and keep the large intervals where the value of the signal is lower than the CPU frequency. The solution is to perform a closing to the signal because it eliminates small oscillations under the CPU frequency and keeps the significant oscillations. After that, we consider regions with low values as corrupted regions due to preemptions.

The clogged system phenomenon is characterized by a signal indicating, for each instance of time, the ratio between the messages in transit and the system bandwidth. This signal is derived directly from the traceable of the application. Typically, it is characterized by small bursts in which the signal has low values. This is the normal behavior, indicating that there are no problems with the communications. However, if the interconnection network is collapsed and a lot of messages are waiting to be sent, or if the bandwidth is remarkably low, the signal will reach high values in large bursts. Therefore, we are interested in a filter capable of eliminating the sets of short, low bursts and of keeping back long, high bursts. The opening filter is useful for this operation. It is composed of, first, an erosion, which eliminates short bursts, and, second, dilation, which restores the remaining bursts to their original size. Thanks to opening, we are able to automatically detect the location of long and high bursts and eliminate the low values of the signal.

Corrupting factors become more significant as the number of processors on which we execute the application is increased. Instead of this, there are several techniques to deal with corrupting factors in executions made on many cores. In the case of flushing, one of the options is to increase the size of the memory buffer where the information is stored. The other option is to synchronize flushings, that is, force the processes to flush the records to the disk at the same time. If we do this, we will have a non-corrupted region after each synchronization point.

In summary, following the methodology described in this section, we make three steps. First, we generate a signal from the initial traceable, for example the number of processes flushing to disk. Second, we apply a morphological filter to extract the relevant information from the signals. The choice of the morphological filter depends on the characteristics of each signal. For example, in the case of a flushing signal, the closing filter is used in order to merge the pulses of the signal that are too close in a unified burst. The width associated to each filter is based on the minimum span of time we consider useful for the analysis. Third, we rule out the perturbed regions, taking into account the pulses of the resulting signal. They indicate which are the perturbed (thus, useless) regions of the traceable and the areas of the signal which are the non-perturbed regions. Finally, the process of identification of structure explained in the next section is applied to these non-perturbed regions of the trace.

6 Structure

Once we have detected, on the one hand, the regions with relevant information, that is, with high-frequency behavior and, on the other hand, we have detected and ruled out the regions corrupted by factors not related to the application or the architecture, we perform an analysis focused on the detection of the iterative behavior of the largest non-perturbed region with high frequencies.

There are two main characteristics of the structure that we look for in a region. First, the periodic structure is based on the identification of several different smaller regions that are very similar. We say that two execution regions are very similar if the signal that represents a metric is very similar in the two regions. The second main characteristic we look for is the hierarchy of the periodicities. The periodic structure is expressed in different levels: the first level is the structure within the original trace, the second level of periodicities is the structure within one period of the first level, and so on. In order to obtain that hierarchical structure, our algorithm is recursive, that is, when the internal structure of one level is detected, we apply again the algorithm within one period of that level.

The information derived from this hierarchical structure based on periodicities is useful in at least two ways. First, our tool shows the analyst the structure as a first approach to the execution of the application under analysis. Second, the tool provides the user with chunks of traces which are cut from the original trace. These small traces are representative parts of the original trace at different levels of the structure. These chops of the original traceable allow an accurate analysis, but the tool also reports several metrics (percentage of time in MPI, etc.) for each of the regions to give a first approach to the performance obtained by the application.

The analysis is based on signals derived from the region we are studying. As indicated in Section 3, it is possible to select many signals to perform the study of the periodicity behavior of the application. In applications with strong communication activity, signals related to MPI calls can give good information. On the other hand, in applications with strong computations activity, signals derived from computing bursts are a better option. In general, signals related to computing bursts give better results, as we show in Section 7.

6.1 Iterative Pattern Detection

To find the internal structure of the application we apply the autocorrelation function to the signal generated from the traceable. The following is a simple definition of autocorrelation function:

$$A(k) = \sum_{i=0}^{N-1} (x_i)(x_{i+k}), \tag{1}$$

where the set $\{x_i\}$ is generated by sampling the signal obtained from the traceable. The higher values of the function $A(k)$ will be reached when k is equal to one of the main periods of $\{x_i\}$. However, for accuracy reasons (Press et al., 1992), the numerical values of $A(k)$ are not obtained by following (1). It is possible to obtain the value of $A(k)$ function performing, first, a discrete Fourier transform (DFT) and, after that, an inverse discrete Fourier transform (IDFT) taking the square of the modulus of each spectral coefficient obtained with the DFT (Press et al., 1992). This method can be implemented using a fast Fourier transform (FFT) library. An important feature of FFT is its computational complexity, $O(n \log(n))$, which allows us to calculate the values of $A(k)$ in reasonable time. Once we have the values of the autocorrelation function, the principal periodicities are selected (De Chevigne and Kawahara, 2002). We select the maximum of the relative maximums. In other words, the period we select, T, will satisfy the following:

$$A(T) = \text{Max}\{A(k) | A(k-1) > A(k) < A(k+1), k > 0\}. \tag{2}$$

Here, we should point out that the signal may not have meaningful periods, or that has two or three significant periods. Therefore, there is a need for a method to estimate the correctness of the period obtained. The approach taken is the following: assuming that T is the period identified and that M is the set of those k where $A(k)$ has a relative maximum:

$$\forall k, \quad (k \in M \wedge k \neq T) \Rightarrow 0.9 > \frac{A(k)}{A(T)}. \tag{3}$$

If the above formula holds, 90% of the value of $A(T)$ is higher than all of the values in the rest of the maximum values. In that case, we assume that T is a good approximation to the main period. We check the logical formula (3) every time we perform an autocorrelation. If the formula is true, we assume that the correctness of the results. If not, we perform a filtering to the original signal in order to filter the small oscillations that can perturb the results and repeat the process. We perform the filtering in order to obtain a coarse-grained description of the signal. This replacement of a fine-grained description with a lower-resolution coarse-grained model will outline the global signal behavior.

Another good criterion to estimate the correctness of the period obtained is the detection of its harmonics. By harmonics we mean multiples of the maximum of the relative maximums. For example, if the maximum of relative maximums is T, its first harmonic is $2T$. If this first harmonic is the second maximum of the relative maximums, the correctness of the period obtained will be remarkably high because the existence of harmonics indicates a strong iterative behavior whose value is T.

Once a "good" period is identified, we select a region of the signal containing an iteration of this period and apply the methodology again to look for inner structure. At the same time, we cut the original traceable in order to provide the analyst with one period on every periodic zone we found. Finally, this methodology needs the execution of intense processes. In order to perform these executions and take advantage of several concurrent processes, we have implemented the methodology with GRID Superscalar (Badia et al., 2003), a Grid programming environment developed at BC.

6.2 Representative Iterations

Once we have extracted the value of the iterative pattern, T, we have to select n iterations from the execution. This is not a trivial issue, because we can often find iterations which are not good representatives of the whole execution. To detect the best iteration, we perform a cross-correlation. In signal processing, cross-correlation is a measure of similarity of two signals. It reaches its maximum where the first signal is closest to the other signal. In this work, we consider the cross-correlation between the input signal (extracted from the execution of the application) and a test signal. This test signal contains n sinusoid periods of value T. The following is the definition of the test signal:

$$f(x) = \begin{cases} \sin\left(\frac{2\pi x}{T}\right) & \text{if } x \in [0, nT], \\ 0 & \text{otherwise.} \end{cases} \tag{4}$$

The underlying idea here is to detect where the input signal is closest to a sinusoid function of period T. To detect this, we select the point where the cross-correlation between the input signal and the test signal is higher. This point, t, is the beginning of the representative segment.

In summary, at the end of the spectral analysis, a representative subset which contains one iteration is selected. This representative subset starts at time t and finishes at time $t + nT$. If it was possible, we applied the methodology recursively over the subset. The extraction of hierarchical patterns is based on spectral analysis. This detection and the subsequent hierarchical search are detailed in Casas et al. (2007).

6.3 Hierarchy Detection

As seen above, we perform a hierarchical detection of the structure of executions of applications. The objective of this hierarchical detection is to identify the minimal representative region of the execution. Many high-performance computing codes base the computing phase on a succession of nested loops. It is possible to find internal levels of these nested loops that are representative of the whole computing phase because the main calculations are carried out within these internal levels. However, the initial structure detection finds the most external iterative structure, that is, the iterations of the most external loop. If we want to detect the structure related to internal nested loops, we apply the methodology recursively over the selected representative region.

The structure found within the whole execution of the application is called level 1 structure. As we have said, it is the structure related to the most external levels of the nested loops. Subsequently, the iterative structure found within one iteration of level 1 is called a level 2 structure. It is related to the second level of the nested loops. This terminology continues as we apply the searching methodology recursively. The search for hierarchy finishes when no iterative structure is found.

Finally, it is important to state that this internal iterative behavior within the level 1 structure exists only if most of the computations are performed within the internal nested loops. Therefore, the existence of nested loops in the source code does not warrant the detection of level 2 structure, since it is possible that the application performs the main calculation outside the most internal loops.

7 Evaluation

This section is divided into four subsections. In the first, we provide an evaluation of the detection of the complete internal structure of several applications. This subsection is focused on the evaluation of the hierarchical search for structure of applications and on the detection of the internal structure among them. In the second subsection, we provide an evaluation of the automatic analysis on several applications using the metrics explained in Section 3. This subsection is focused on the evaluation of the efficiency of several metrics in detecting internal structure. For this reason, the results depicted in this section are more focused on the differences of the results obtained with the different metrics than on the complete internal structure of the applications. The third subsection is focused on evaluating the size reductions achieved using our technique, that is, we compare the size of the original traceable with respect to the size of two iterations extracted from the original data. Finally, in the fourth subsection we show that the automatic methodology scales linearly with respect to the size of the input.

7.1 Evaluation of Complete Internal Structure and Hierarchical Searching

In this section, we evaluate the complete internal structure detected by spectral analysis and its hierarchical search. In order to do so, we apply the methodology explained in this section to four real applications: Liso (Hoyas and Jimeńez, 2006) with 74 processors, Idris (Teysser, 2002) with 200 processors, Gadget (Springel et al., 2001) with 256 processors, and Linpack[6] with 2,048 processors. These have been executed and traced in MareNostrum[7]. The structure found in these applications is based on one of the metrics explained in Section 3, the MPI point-to-point calls.

In Figure 9 we show graphically a part of the structure of the Liso traceable. This structure is shown with a Parader visualization. In this visualization, the horizontal axis represents the time and the vertical axis the different processes. The color black means that a given process in a given instance of time is not executing any MPI call. On the other hand, a light colored point (green when printing or visualizing in color) indicates that an MPI call is being executed by the process. The figure first shows a visualization of the whole traceable. The flushing regions are also outlined. In the second part of Figure 9 we show the structure of the first region without flushing. In this case, we show, first, a region with a non-periodic structure that corresponds to the initialization phase of the application. The span of the initialization phase is 18,029 ms. After that, there is periodic region with five iterations. The span is 47,306 ms and the period shown is 9,010 ms. Finally, in Figure 9 we show one of the iterations of the periodic zone.

Table 1 shows that the automatic system was able to detect the structure shown in Figure 9. Furthermore, we can see the results of the automatic analysis for the whole

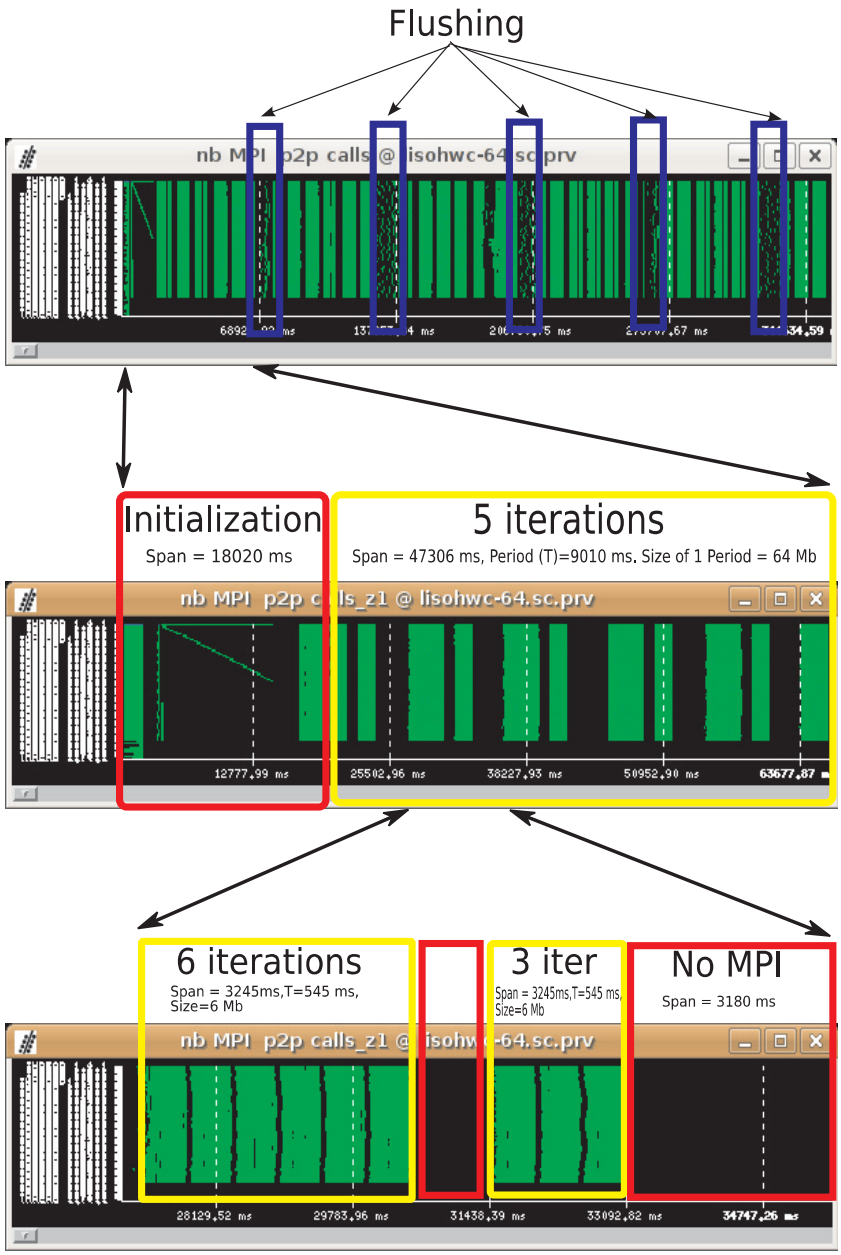

Fig. 9 Top: visualization of the whole Liso traceable and the flushing zones. Middle: the first region without flushing is shown. We highlight a region without periodic structure and a region with five iterations. Bottom: we show one of these iterations.

Liso application. In this table we show two characteristics of the execution: first, from left to right we show the hierarchy; second, from top to bottom we show the temporal sequence.

In the first column, we show the duration of the whole execution in milliseconds. Next, in the second column of Table 1 there is a decomposition of the total elapsed time of execution. In this second column, we show a set of

Table 1
Liso application structure detected by our system. The time units are milliseconds. The first three columns show the hierarchical levels of periodicity. Level 0 column shows the total elapsed time, Level 1 column shows the different phases detected by the automatic system, and Level 2 shows the internal structure of one of the periods of Level 1. In the first three columns, each cell contains three numbers: the total span of the region, the number of the periods found in that region and the duration of each period.

Span/#it/*T*			Flushing
Level 0	Level 1	Level 2	
356,352/1/–	18,020/1/–		
	47,306/5/9,010	3,245/6/545	
		970/1/–	
		1,615/3/550	
		3,180/1/–	
	10,273/1/–		X
	49,166/5/9,105	1,490/3/545	
		1,140/1/–	
		1,585/3/550	
		3,030/1/–	
		1,860/3/535	
	11,880/1/–		X
	60,773/7/9,100	3,215/6/540	
		1,004/1/–	
		1,615/3/550	
		3,266/1/–	
	11,576/1/–		X
	49,253/5/9,145	1,650/3/550	
		2,825/1/–	
		3,400/6/550	
		1,270/1/–	
	12,269/1/–		X
	48,347/5/9,045	3,185/6/550	
		1,015/1/–	
		1,625/3/560	
		3,220/1/–	
	13,312/1/–		X
	24,177/3/8,905	1,875/1/–	
		3,425/6/550	
		1,010/1/–	
		1,635/3/555	
		960/1/–	

Table 2
Idris: table representation.

Span/#it/T			Flushing
Level 0	Level 1	Level 2	
1,097,460/1/–	205,740/1		
	885,955/9/102,870	550/5/110	
		590/1/–	
		540/7/80	
		101,190/1/–	

numbers in each cell. The first number is the total time span of the region, the second is the number of periods found in that region, and, finally, the third is the duration of each period. In the regions where no periodicity was found a dashed line is written.

The second column refers to the first level of the hierarchical structure, i.e. is the structure over the original trace. For example, the MPI point-to-point calls distribution of the first (18,020 ms) and second (47,306 ms) regions of the traceable shown in Figure 9 are represented in the first and second cells of the second column.

The third column contains the second level of the structure, i.e. the structure that can be found in one of the periods of the first level. For example, the second, third, fourth, and fifth cells of the third column are the decomposition of one of the periods of level 1. The MPI point-to-point distribution is shown in Figure 9. Here, the second level of structure can be identified, with six periods (of 545 ms) of communication, a computation period (of 1,005 ms), three more periods of communication (each of 550 ms), and a final computation period (of 3,180 ms).

Finally, the fourth column shows the existence of flushing events in a given region of the traceable.

Note that the first five rows correspond to the structure represented in Figure 9

The output of the tool is basically the information contained in this table plus the names of the files where the chops of the original traceable can be found.

In Table 2 we show the structure detected in the Idris application. In Figure 10 we show another possible representation of the same information. In this representation, the hierarchical structure is depicted from top to bottom and the temporal sequence is drawn from left to right. Finally, in Tables 3 and 4 the structure found in Gadget and Linpack applications is shown.

7.2 Evaluation of Metrics

In this section, we evaluate the automatic system using several applications as input and changing the metrics. The

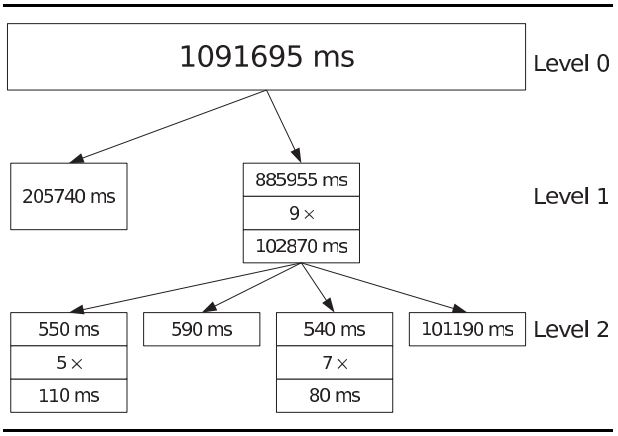

Fig. 10 Idris: tree representation.

Table 3
Gadget.

Span/#it/T		Flushing
Level 0	Level 1	
1,097,460/1/–	55,000/1/–	
	811,126/23/35,095	
	286,334/1/–	X

Table 4
Linpack.

Span/#it/T		Flushing
Level 0	Level 1	
223,168/1/–	82,850/1/–	
	140,318/130/1,130	

Table 5
Evaluation of spectral analysis. Metrics used are SDCB (Sum of Duration of Computing Bursts), NPC (Number of Processes Computing), MPIp2p (Number of Processes in a MPI point-to-point call) and CPI (Cycles per Instruction). Results are depicted in milliseconds.

Name	Processor	SDCB Main period	NPC Main period	MPIp2p Main period	CPI Main period
ALYA	64	11,132	Not found	Not found	Not found
BT	16	1,277	1,277	1,277	1,276
	36	639	639	639	639
	64	357	357	357	357
	121	197	197	197	197
	256	115	115	114	Not found
CG	16	39	39	39	39
	32	23	23	23	Not found
	64	17	17	17	Not found
	128	11	11	Not found	Not found
CPMD	64	229,393	Not found	Not found	Not found
LU	16	1,441	1,439	1,439	Not found
	32	530	539	539	Not found
	64	227	227	209	Not found
MG	16	1,777	1,775	1,773	Not found
	32	982	981	981	Not found
	64	372	372	372	372
RTM	32	1,175	1,175	1,170	1,170
SP	16	1,341	1,341	1,340	1,341
	36	841	840	840	840
	64	330	330	330	330
SPECFEM3D	1,944	10,626	10,356	Not found	Not found
VAC	128	4,644	4,667	4,631	Not found
WRF	128	2,194	2,194	2,194	Not found
	256	1,193	Not found	Not found	Not found
	512	792	Not found	Not found	Not found

applications are ALYA (Houzeaux et al., 2007) executed on 64 processors, some of the NAS benchmarks (Bailey et al., 1994) (NAS-BT class C executed on 16, 36, 64, 121, and 256 processors, NAS-CG class C executed on 16, 32, 64, and 128 processors, CPMD[8] on 64 processors, NAS-LU class C on 16, 32, and 64 processors, NAS-MG class C on 16, 32, and 64 processors, RTM (Baysal et al., 1983) on 32 processors, SPECFEM3D (Komatitsch et al., 2002) on 1,944 processors, NAS-SP class C on 16, 36, and 64 processes), VAC (Saxena et al., 2004) on 128 processors, and WRF-NMM[9] on 128, 256, and 512 processors.

We analyzed these applications using many metrics: Sum of Duration of Computing Bursts (SDCB), Number of Processes Computing (NPC), Number of Processes in a MPI point-to-point call (MPIp2p), Ratio between the Cycles and the Instructions (CPI), Number of Flying Communications (Communications), Number of Floating Point Instructions Executed (FLOPS), Number of Processes in a MPI Collective Call (MPICol), and Number of Instructions Performed (Instructions).

Tables 5 and 6 depict the results. In the first column of these tables, we show the name of the application and in

Table 6
Evaluation of spectral analysis. Metrics used are communications (number of flying communications), FLOPS (number of floating point instructions executed), MPICol (number of processes in a MPI collective call) and instructions (number of instructions performed). Results are depicted in milliseconds.

Name	Processors	Communications Main period	FLOPS Main period	MPICol Main period	Instructions Main period
ALYA	64	Not found	Not found	11,395	Not found
BT	16	1,277	1,277	Not found	1,277
	36	639	639	Not found	639
	64	357	357	Not found	357
	121	197	197	Not found	197
	256	115	114	Not found	116
CG	16	39	Not found	Not found	Not found
	32	23	Not found	Not found	Not found
	64	17	Not found	Not found	Not found
	128	Not found	Not found	Not found	Not found
CPMD	64	Not found	Not found	229,386	Not found
LU	16	1,442	Not found	Not found	1,441
	32	529	Not found	Not found	Not found
	64	227	210	Not found	Not found
MG	16	1,774	1,778	Not found	1,778
	32	982	982	Not found	Not found
	64	372	372	Not found	Not found
RTM	32	1,170	Not found	Not found	Not found
SP	16	1,341	1,341	Not found	Not found
	36	841	841	Not found	Not found
	64	330	330	Not found	Not found
SPECFEM3D	1,944	Not found	Not found	Not found	Not found
VAC	128	4,638	Not found	4,626	Not found
WRF	128	2,194	Not found	2,193	Not found
	256	Not found	Not found	Not found	Not found
	512	Not found	Not found	Not found	Not found

the second column we specify the number of processors used to execute the application. In the third column of Table 5 we show results using the Sum of Duration of Computing Bursts metric. In each cell of this column, we depict the duration in milliseconds of the main periodicity detected within the execution of the application. We do not show information about the whole structure of the application, as was done in Table 1, in order to provide a more friendly representation of data. As we have said, in this section we are more interested in comparing the results of the automatic analysis using many metrics than in a complete description of the results. Then, in the fourth column of Table 5 we show the results using the Number of Processes Running metric, in the fifth we depict the results taking the Number of Processes in a MPI point-to-point call, and, finally, in the sixth column we show the results using the Ratio of Instructions per Cycle.

In Table 6 we show, in the third column, results taking the Number of Flying Communications as input metric,

Table 7
Sizes of all of the representative traces of each level.

Application	Total trace size	Level 1 size	Level 2 size
Liso	2.02 GB	64 MB	6 MB
Idris	2.7 GB	250 MB	25 MB
Gadget	2.7 GB	53 MB	
Linpack	6.7 GB	46 MB	

and, in the fourth column, we depict results using the number of floating point instructions executed. In the fifth column we show results obtained using Number of Processes in MPI Collective Calls as metric. Finally, in the sixth column, we show some results taking the Number of Instructions Performed.

As we can see in Tables 5 and 6, the best results, understanding best result as a correct structure detection, are achieved using the two metrics related to computing bursts (Sum of Duration of Computing Bursts and Number of Processes Computing). Metrics such as the Number of Processes in a MPI Point-to-Point call and Number of Flying Communications also provide good results in general. On the other hand, metrics such as Number of Instructions Performed, IPC and Number of Floating Point Instructions Executed provide poorer results. The justification of these differences is that the regularity of the first set of metrics highlights the iterative structure of executions of applications. Metrics related to hardware counters are subjugated to a lot of irregular oscillations which can cloud the iterative structure. The metric generated from the collective call is also very regular, but the problem is that sometimes there are no collective calls, or they do not have iterative behavior because they only initialize data. Finally, it is important to state that the results are sound, that is, if the system successfully detects the value of the main periodicity of an execution using different metrics as input, the values obtained are very close.

7.3 Evaluation of Traceable Size Reductions

This section evaluates the traceable size reductions achieved using the automatic methodology explained in this section. First, we provide data about the applications used in Section 7.1. In this data we see that strong reductions are achieved when we pass from the original traceable to the first level chops. The hierarchical search reduces the size even more, but this last reduction is weaker than the first.

In Table 7 we show the average size of the chops of the original trace. As observed in Section 6.1, every time the system finds a periodic region, it selects one of the periods of that region and cuts the traceable to provide the analyst with representative chops of the original traceable. The sizes shown in Table 7 are, first, the size of the total traceable and, second, the average size of the periods of the first level. For example, the Liso traceable has six periodic regions, one of these regions in every non-flushing zone. If we take one period of every periodic region and then we cut the traceable, we obtain six small traceries. The average size is the value we show. Finally, the third column is the average size of the second level periods. Note the large reduction in the amount of data to study.

In Figure 11 we represent, first, the size of the whole trace. Next, we show the total sum of the sizes of the first and second level chops. Obviously, if there is only one periodic region, the value shown in Figure 11 is the same as the value represented in Table 7. The first level of Idris application is an example. The most important thing, however, is that the global behavior of the applications, with the exception of the flushing and initialization regions, is contained in level 1 traceries. We have reduced notably, from gigabytes to a few megabytes, the amount of data to be analyzed in order to study the performance of the applications.

We also provide the size reductions of the applications used in Section 7.2 as shown in Tables 5 and 6. Remember that these applications were used to check the behavior of several metrics with respect to spectral analysis. Strong reductions on them are also achieved by using the metric generated from durations of computing bursts.

In Figures 12 and 13 we show the results in terms of the size of the traceries. For each execution of each application on different number of processors, we show the size of the whole traceable and the size of the reduced traceable. This reduced trace contains two iterations. In these cases, the reduced trace is derived from the structure found within the whole execution of the application, that is, level 1 structure. As we can see, we achieve remarkable reductions in all of the cases. The results shown have been obtained using the Sum of Duration of Computing Bursts as input metric.

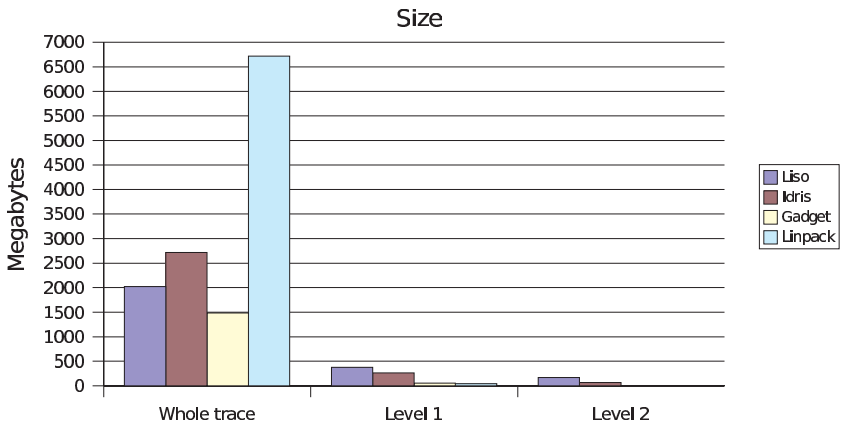

Fig. 11 Sum of the sizes of all of the representative traces of each level.

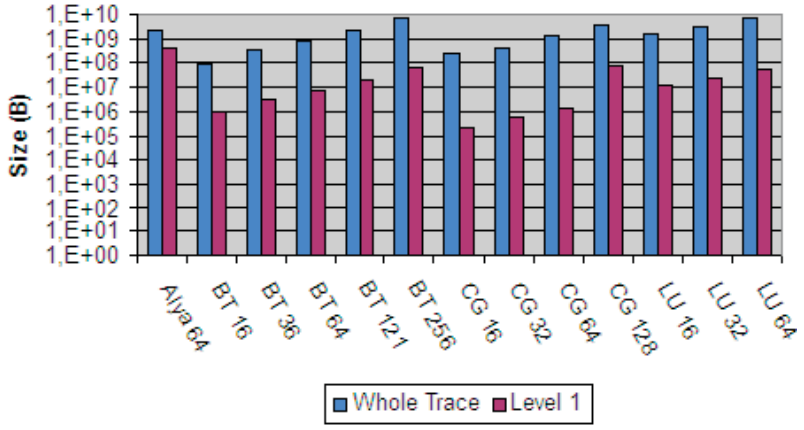

Fig. 12 Representation, on a logarithmic scale, of size reductions of the applications depicted in Tables 5 and 6. Results are depicted in bytes. The metric used to perform the analysis is the Sum of Durations of Computing Bursts.

7.4 Scalability of the Analysis

As has been pointed out in this paper, the analysis we apply to executions of applications consists mainly in applying signal processing algorithms. It is widely reported that these algorithms have very low computational complexity. Therefore, our automatic methodology also has low complexity. In order to prove this statement, in this section we perform a detailed study of the scalability of our analysis with respect to input data size.

In Figures 14 and 15 we depict, for each application, the input data size and the execution time needed to perform the reduction of size. As we can see, the relationship between these two magnitudes is almost linear. This fact becomes clear when looking at the results of BT or LU applications. As we increased the number of processors where we executed these applications, the size of the generated data increases and the reduction time is also increased. The relationship between the input data size and reduction time is clearly linear.

In Figure 16 we depict another view of the scalability of the system. In the *x*-axis, we plot the size of the tracerries. In the *y*-axis, we plot the time needed by the automatic system to perform the reductions. Each point represents an execution of the automatic system. Each point is labeled with the name of the application traceable used as input data. The same conclusions indicated above can be extracted from this figure.

It should be noted that the input data size is higher in the case of WRF executed on 512 processors than in the

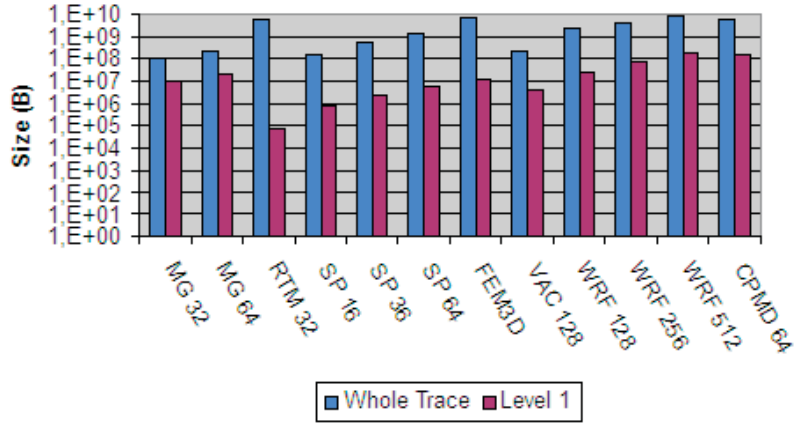

Fig. 13 Representation, on a logarithmic scale, of size reductions of the applications depicted in Tables 5 and 6. Results are depicted in bytes. The metric used to perform the analysis is the Sum of Durations of Computing Bursts.

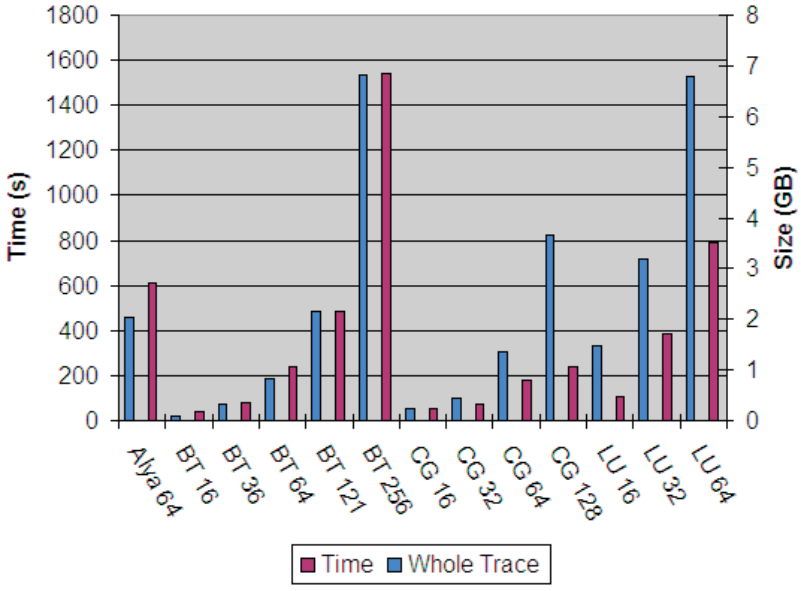

Fig. 14 Scalability of the system.

case of RTM executed on 32 processors. However, the required reduction time is higher in the case of RTM 32 than in the case of WRF 512. The explanation for this is that data extracted from WRF execution on 512 processors has more perturbed regions (flushing, clogged system) than RTM. Since the analysis is applied on non-perturbed regions, the study of RTM requires the analysis of larger data sets and, therefore, the required time to perform the reduction is higher. The same can be said comparing WRF 512 against CPMD 64. In general, input data sets with very few perturbed regions require more analysis time, but they provide more possibilities of finding significant structure within them.

It is important to state that the most remarkable size reductions are the largest reduction processes. Two clear examples of this fact are, first, RTM 32 and CPMD 64. On both cases, we achieve significant reductions, ruling out several gigabytes of redundant or damaged data but

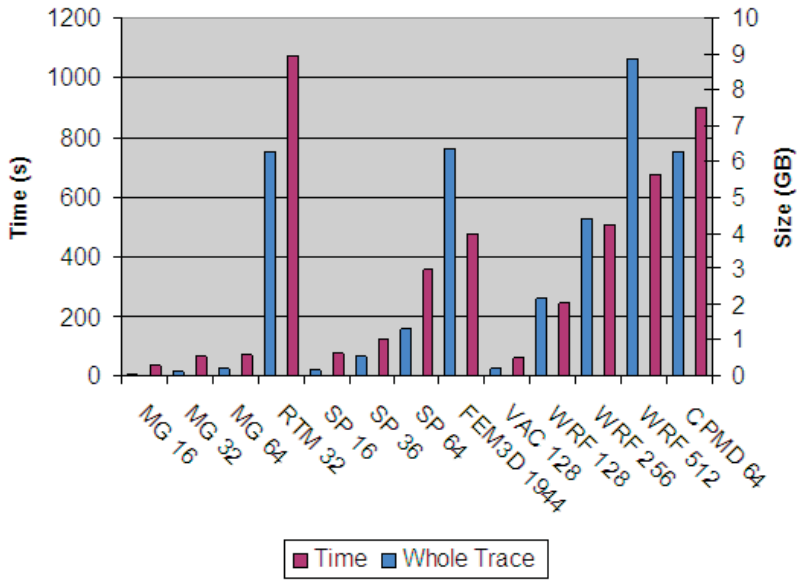

Fig. 15 Scalability of the system.

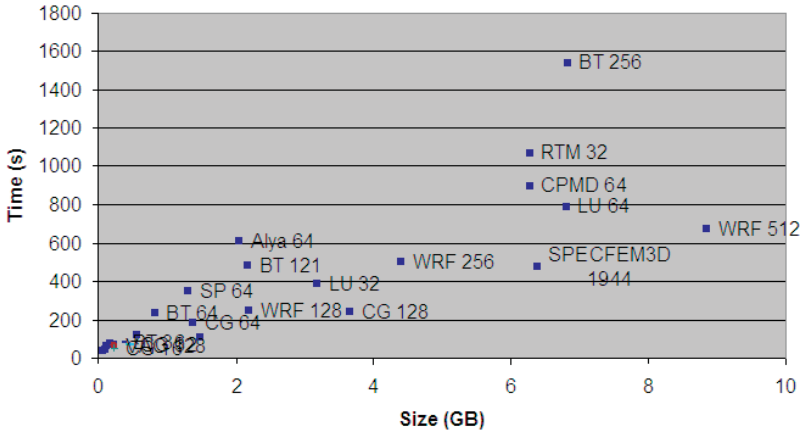

Fig. 16 Scalability of the system.

we need more than 15 minutes to perform the reductions. Even these largest reduction processes can be performed in a few minutes.

8 Conclusions

In this paper, we have explored the possibility of automatically deriving the internal structure of a traceable. This structure has two main properties. First, it is based on periodicities and, second, it is hierarchical. We have shown that it is possible to, first, detect the perturbed regions of the traceable and, second, derive the internal structure of non-perturbed regions. It is useful in many aspects: this makes the process of tracing the application easier, it avoids the need to spend time studying perturbed zones, it gives the internal structure of the traceable and, finally, gives the most representative regions of it. In conclusion, we have reduced the problem of analyzing a huge traceable

(10 or 20 GB) to the study of several hundreds of megabytes.

The other important aspect to take into account is the flexibility of the methodology. We have demonstrated that it is possible to apply the automatic analysis to any kind of metric we can generate from data contained in traceries. This flexibility opens the possibility to study the application from many points of view, ruling out the non-significant metrics and taking into account the useful metrics.

In the future, we will address issues such as what to do if the behavior of the application changes over the time. For example, what to do if the amount of data sent in the communication at the end of each iteration varies. A possible way to handle with these situations is to pre-process the signal obtained from the execution of the application. Such pre-processing consists of taking into account only the values which the signal reaches, not during what time the signal reaches such values. With this pre-process, we can equilibrate the iterations, detect them from the pre-processed signal and go back to the original.

The automatic detection of structure of MPI applications traceries is the first step to automatize, as much as possible, the process of performance analysis of executions of applications in order to reduce the time required to analyze data generated from executions. Automatic structure detections using spectral analysis opens a wide range of possibilities.

Acknowledgments

This work was supported by the Catalan Government (2009-SGR-980), the Spanish Government (TIN2007-60625), and by the IBM/BSC Mareincognito Project.

Authors' Biographies

Marc Casas has a degree in Applied Mathematics (2004) from the Technical University of Catalonia (UPI). Currently, he is finishing his PhD in Computer Science in the Computer Architecture Department at UPI. From 2005 to 2009 he held an academic grant from the Spanish Government to develop his research. He has worked in several R&D projects at Barcelona Supercomputers Center (BC). His research interest are performance modeling of MPI programs and analytical methods for high-performance computing.

Rosa M. Badia has a PhD in Computer Science (1994) from the Technical University of Catalonia (UPI). Since 2008 she has been a Scientific Researcher from the Consejo Superior de Investigaciones Cientificas (CHIC) and manager of the Grid Computing and Cluster research group at the Barcelona Supercomputers Center (BC). She has been involved in teaching and research activities at the UPI from 1989 to 2008, and where she has also been an Associated Professor since 1997. From 1999 to 2005 she has been involved in research and development activities at the European Center of Parallelism of Barcelona (CEPBA). Her current research interest are performance prediction and modeling of MPI programs and programming models for complex platforms (from multicore to the Grid). She has participated in several European projects and currently she is participating in projects HPC-Europa2, Brein, and is a member of HiPEAC2.

Jesús Labarta has been full professor at the Computer Architecture Department at UPI since 1990. Since 1981 he has been lecturing on computer architecture, operating systems, computer networks, and performance evaluation. Since 1995 he has been director of CEPBA and currently he is the director of the Computer Science Research Department as BC. His research interest has been centered on parallel computing, covering areas from multiprocessor architectures, memory hierarchy, parallelizing compilers and programming models, operating systems, numerical kernels, metacomputing tools, and performance analysis and predictions tools. He has lead the technical work of UPI in some 15 industrial R&D projects.

Notes

1 See http://www.lanl.gov/roadrunner/rrposterslanlbooth.shtml

2 See http://www-unix.mcs.anl.gov/mpi/

3 See http://www.dtcenter.org/wrf-nmm/users/

4 See http:www.cepba.upc.es/paraver/docs/OMPItrace.pdf

5 See http://www.dtcenter.org/wrf-nmm/users/

6 See http://www.netlib.org/linpack/

7 See http://www.bsc.es

8 See http://www.cpmd.org

9 See http://www.dtcenter.org/wrf-nmm/users/

References

Badia, R. M., et al. (2003). Programming Grid applications with GRID Superscalar. *J. Grid Comput.* **1**(2): 151–170.

Bailey, D., et al. (1994). The NAS Parallel Benchmarks. *RNR Technical Report RNR-94-007*.

Baysal, E., Kosloff, D. D. and Sherwood, J. W. C. (1983). Reverse time migration. *Geophysics* **48**: 1514–1524.

Brunst, H., Kranzlmuller, D. and Nagel, W. E. (2004). Tools for scalable parallel program analysis—Vampir VNG and DeWiz. *Distributed and Parallel Systems*, pp. 93–102.

Casas, M., Badia, R. M. and Labarta, J. (2007). Automatic extraction of structure of MPI applications tracefiles. *Proceedings of the International Euro-Par Conference on Parallel Computing (Euro-Par)*, pp. 3–12.

Daubechies, I. (1992). *Ten Lectures on Waveletes*. Philadelphia, PA: Society for Industrial and Applied Mathematics.
De Chevigne, A. and Kawahara, H. (2002). YIN, a fundamental frequency estimator for speech and music. *J. Acoust. Soc. Am.* **111**: 1917–1930.
Gamblin, T. et al. (2008). Scalable load–balance measurement for SPMD codes. *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*.
Geimer, M. et al. (2008). The SCALASCA Performance Toolset Architecture. *Proceedings of the International Workshop on Scalable Tools for High-End Computing (STHEC)*, pp. 51–65.
Gerndt, M. and Kereku, E. (2007). Automatic memory access analysis with Periscope. *Proceedings of the International Conference on Computational Science (ICCS)* (*Lecture Notes in Computer Science*, Vol. 4488). Berlin: Springer, pp. 847–854.
Houzeaux, G., Eguzkitza, B. and Vázquez, M. (2007). A variational multiscale model for the advection-diffusion-reaction equation. *Communications in Numerical Methods in Engineering*.
Hoyas, S. and Jimeńez, J. (2006). Scaling of velocity fluctuations in turbulent channels up to Re=2003. *Phys. Fluids* **18**: 351–356.
Huffmire, T. and Sherwood, T. (2006). Wavelet-based phase classification. *Proceedings of Parallel Architectures and Compilation Techniques (PACT)*, pp. 95–104.
Kaiser, G. (1994). *A Friendly Guide to Wavelets*. Basel: Birkhäuser.
Knuepfer, A., Brunst, H. and Nagel, W. E. (2005). High performance event trace visualization. *Proceedings of the 13th Euromicro Conference on Parallel, Distributed and Network-Based Processing*, pp. 258–263.
Komatitsch, D., Ritsema, J. and Tromp, J. (2002). The spectral-element method, Beowulf computing and global seismology. *Science* **298**: 1737–1742.
Kranzlmuller, D., Scarpa, M. and Volkert, J. (2003). DeWiz—a modular tool architecture for parallel program analysis. *Proceedings of the European Conference on Parallel Computing (Euro-Par)*, pp. 74–80.
Labarta, J. et al. (1996). DiP: a parallel program development environment. *Proceedings of the European Conference on Parallel Computing (Euro-Par)*, pp. 665–674.
Mohr, B. and Traff, J. L. (2003). Initial design of a test suite for automatic performance analysis tools. *Proceedings of the IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, pp. 131–140.
Mohr, B. and Wolf, F. (2003). KOJAK—a tool set for automatic performance analysis of parallel programs. *Proceedings of the International Euro-Par Conference on Parallel Computing (Euro-Par)*, pp. 1301–1304.
Nagel, W. et al. (1996). VAMPIR: visualization and analysis of MPI resources. *Supercomputer* **63**: 69–80.
Press, W. H. et al. (1992). *Numerical Recipes in FORTRAN: The Art of Scientific Computing*, 2nd edn. Cambridge: Cambridge University Press, pp. 538–539.
Roth, P. C. and Miller, B. P. (2006). On-line automated performance diagnosis on thousands of processes. *Proceedings of 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pp. 69–80.
Saxena, V. et al. (2004). Vacuum-assisted closure: microdeformations of wounds and cell proliferation. *Plastic and Reconstructive Surgery* **114**: 1086–1096.
Shen, X., Zhong, Y. and Ding, C. (2004). Locality phase prediction. *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 165–176.
Shende, S. and Malony, A. D. (2006). The TAU Parallel Performance System. *Int. J. High Perform. Comput. Appl.* **20**(2): 287–311.
Sherwood, T. et al. (2002). Automatically characterizing large scale program behavior. *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems, (ASPLOS)*, pp. 45–57.
Springel, V., Yoshida, N. and White, S. D. M. (2001). Gadget: a code for collisionless and gasdynamical cosmological simulations. *New Astron.* **6**(2): 79–117.
Teysser, R. (2002). Cosmological hydrodynamics with adaptive mesh refinement—a new high resolution code called RAMSES. *Astron. Astrophys* **385**: 337–364.
Vetter, J. S. and Worley, P. H. (2002). Asserting performance expectations. *Proceedings of the ACM/IEEE International Conference on Supercomputing*, pp. 1–13.
Walnut, D. F. (2004). *An Introduction to Wavelet Analysis*. Boston, MA: Birkhäuser.