# Chapter 9
# New Analysis Techniques in the CEPBA-Tools Environment

Jesus Labarta

**Abstract** The CEPBA tools environment is a performance analysis environment that initially focused on trace visualization and analysis. Current development efforts try to go beyond the presentation of simple statistics by introducing more intelligence in the analysis of the raw data.
The paper presents an overview of three recent developments in this area. First, we show how spectral analysis techniques can be used to isolate sufficiently small regions of a trace that characterize the behavior of the whole run. Second, we describe how clustering analysis techniques can be used to identify temporal and spatial structure in parallel programs, an essential component to ease the job of the analyst, but also to automatically derive a broad range of both precise and focused metrics from a single run of a program. Then we describe how sampling and tracing data acquisition techniques can interoperate to generate with very low overhead extremely precise metrics about the temporal behavior of a program.
The development rests upon the trace based CEPBA-Tools environment, using the Paraver visualization capabilities to check the quality and usefulness of the techniques. Once identified, they can be implemented on-line aiming at maximizing the amount of information obtained from a run. We report the work being done on top of MRNET in this direction.
We consider that by applying and combining these and other techniques from various data analysis and mining fields, performance analysis tools will be able to effectively address the huge challenge posed by future exascale systems.

Jesus Labarta
Barcelona Supercomputing Center and Technical University of Catalonia,
Jordi Girona 29, Barcelona, Spain,
e-mail: jesus.labarta@bsc.es

## 9.1 Introduction

As larger and larger systems are being developed and applications run on them, the issue of understanding how they behave and how efficiently our applications use the available resources is more and more important.

Performance analysis tools rely on hooks injected into programs to capture relevant events and derive the metrics that quantify and explain their behavior from the acquired data. Traditionally the focus of performance analysis tools has been centered on the monitoring or data acquisition mechanisms. The algorithms used for processing the raw data before presenting results to the analyst are typically very simple. Profilers are the more widely used type of tools and they just present simple statistics like time in each routine, total count of invocations or the accumulated instructions. By aggregating over the time and processor dimensions and focusing on a limited set of predefined metrics, profilers reduce the amount of data that has to be emitted and then presented to the user. This advantage comes at the expense of loosing detail on the variability of system activity and results in a lot of relevant information being discarded.

Intermediate approaches with different amounts of precomputed profile data have been used but the question arises as to how raw data should be processed to maximize the relevant information obtained from it while minimizing the amount of data emitted.

Other areas of science and engineering have developed elaborated techniques to extract useful information out of the raw data. Signal and image processing and data mining techniques are widely used in different fields with such purpose. We have the perception that performance analysis lags far behind other areas in the actual use of those techniques as well as the conviction that they could be successfully used in our field.

In this paper we describe some of the usages of signal processing and data analysis techniques within the CEPBA-tools environment. Section 9.2 briefly describes the environment used to develop and validate the approaches described in successive sections. Section 9.3 then focuses on the use of spectral analysis techniques, section 9.4 on the use of clustering techniques and section 9.5 on the combined use of instrumentation and sampling. Section 9.6 describes current work in integrating the above described techniques in an automatic on-line analysis environment and section 9.7 presents some views on future directions.

## 9.2 The CEPBA-Tools Environment

The development of the CEPBA-tools environment started in 1996 [5] with the objective to better understand the detailed interactions that could take place in a multiprogrammed message passing machine based on the Transputer chip. Three main components constituted the environment: a set of tracing packages (now MPITrace) for message passing programs, a simulator (Dimemas) for message passing machines also modeling the time sharing behavior within a node and a visualizer (Paraver) capable of displaying traces produced by the simulator. The traces capturing the actual behavior of a run of the parallel program could also be directly generated by the instrumentation package and visualized with Paraver. The usage of the tools then evolved to support detailed analysis of single applications and prediction of the impact of different architectural parameters in their performance.

Paraver is a flexible browser for traces that contain sequences of timestamped records of three types: events, states and communication. The Paraver trace format describes the structure of these records but their semantic is essentially undefined, which gives the possibility to apply the tool in very different areas, areas far beyond those initially targeted. This is certainly the case for the records that represent a punctual event with two attributes (type and value) and for state records that represent an interval between start and end for which one attribute is given. The attributes are integer values in which the tracing package can encode the information as desired. Each record applies to one object in a hierarchical structure of three levels which when instrumenting parallel programs are typically mapped to application, process and thread. Communication records actually relate two such objects and have two additional attributes.

The core of the Paraver engine [12] is called the *semantic module*. It provides through its GUI a very flexible algebra to specify how functions of time can be generated out of the records and the numerical values of their attributes. One such function of time is generated for each object. The fact that internally Paraver considers the data it handles as functions of time leads naturally to some of the techniques described through the paper. Finally, a simple but flexible rendering mechanism translates the functions of time to colored timeline plots. Typically a palette of colors is used to translate categorical valued functions of time such as identifier of the MPI call, or user function. A gradient color map is used for continuous valued functions, using light green for low values up to dark blue for large values. Areas where the function value is above a specified range are highlighted in orange and if the function value is zero, the background color is used. Non linear rendering is used to expose information to the analyst in cases where many values map to a pixel. This technique addresses the scalability issues faced when displaying traces with many objects or representing long time intervals.

The *analysis module* implements a single mechanism to compute tables. A very generic approach is used, able to not only report statistics but also histograms and correlations between any of the functions generated by the *semantic module*.

Complex expressions can be defined in the *semantic* and *analysis modules* and saved along with the display setup in configuration files for later reuse. The lack of semantics in the trace format plus the flexibility of these two modules makes Paraver an extremely powerful and versatile browser. It has been used to analyze MPI and MPI+OpenMP programs but also operating system activity, multicore architectures, or file system behavior. Other time series not having any relationship to parallel programming such as stock sensor data or exchange rates could be analyzed in Paraver without requiring any modification of its source code and without requiring convoluted mappings of concepts in these areas to the concepts handled by visualizers too specialized in just parallel program.

## 9.3 Spectral Analysis

Many applications tend to have an iterative structure, originating from the time stepping process they often simulate. The behavior of such iterations tends to be very repetitive or at most slowly varying as the simulated system evolves. This means that a few iterations are sufficient to describe the behavior of applications during long intervals of time. Spectral analysis techniques can be used to determine the periodic structure of a program. One of the applications of such analysis is to automatically select the time interval to be traced such that at least one whole period is captured.

Other sources of repetitive behavior are the iterative nature of the numerical algorithms, the need to process a large number of particles or elements, and so on. These iterative patterns may be nested but for a global performance analysis purpose we are mostly interested in the outermost levels. In [4] we showed how the iterative behavior at different levels can be identified on traces from large runs of a program. The main usage in that work addressed the possibility of reducing the size of the traces required to still be able to do very detailed analyses.

The spectral analysis can be applied to signals representing the evolution with time of some metric for the whole applications, such as average instantaneous instructions per cycle (IPC), or actual number of processes inside MPI calls. The paper also revealed that it is not necessary to use signals representing a metric meaningful from the performance point of view. In fact, the sum at each point in time of the duration of the computation burst of all the processes is a signal with no real meaning that captures pretty well the structure of an application. A computation burst is the time interval between exit of an MPI call and entry to the next. During the whole burst, a process contributes with its duration to the global function. While a process is inside MPI no contribution is made to the global signal.

In the same study we also showed how other techniques such as mathematical morphology can be used to clean-up signals. This non linear filtering technique was applied to signals identifying regions where certain type of perturbations occurred while obtaining the trace. One example of such a signal is the number of processes flushing their trace buffer to disk. Although with sufficiently large buffers this will not be very frequent, it will certainly perturb not only the process doing the flush but also other processes communicating with it. Furthermore, it is frequent that different processes flush their buffers at about the same time. By applying dilation and erosion filters to such signal it is possible to separate regions in the influence area of the perturbations from large regions without such perturbations.

Other technique to obtain general structural information of the trace is the Wavelet transform. This can be used to automatically separate the non iterative phases of an application such as initialization and termination from the core computation phase. The Wavelet transform produces information about the spatial localization of energy at different frequencies. When applied to signals like the sum of the useful duration described previously, initialization and termination phases tend to have much lower energy at high frequencies. By applying again mathematical morphology techniques to the high frequency outcome of the wavelet transform we can identify regions of major program activity.

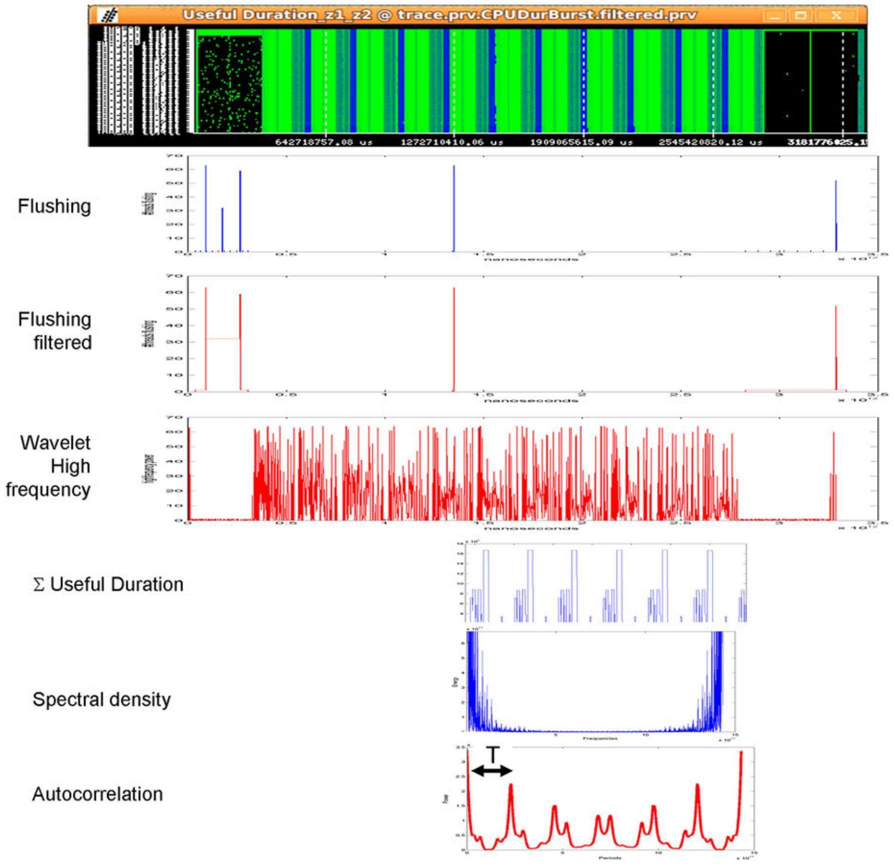

Fig. 9.1: Process of automatic period detection

The whole process is described in figure 9.1. The timeline on top represents the duration of the computation bursts for each of the 128 processes of a run of the WRF weather modeling code. Dark blue represents long computation bursts, light green short computation bursts and black corresponds to time inside MPI. The two signals below represent when some process is flushing data to disk. At the scale shown there is no appreciable difference between them, but the second one corresponds to the outcome of the filtered signal with mathematical morphology. A look at a more detailed scale shows that several flushes from different processes have actually been merged into a single burst. The fourth view from top represents the high frequency components identified by the Wavelet transform when applied to the useful duration signal. The main computation area corresponds to the region with high values. Combining this signal and the outcome of the flush analysis the tool identifies the longest core computation region without perturbation and builds the useful duration signal for that interval. Computing the FFT, squaring it and computing the inverse we obtain the autocorrelation function shown at the bottom of the figure. Peaks in this figure correspond to periodicities in the signal. In our case, the first local maxima different from the origin corresponds to the coarser periodicity. The tool can then be used to cut a region of the trace of one or several periods (depending on a requested maximum trace size).

This functionality was developed as a command line tool to process large traces and is now being integrated both in the Paraver GUI and in the intelligent on-line tracing packages as described in section 9.6.

## 9.4 Clustering Techniques

Clustering techniques have been used in the parallel performance analysis area mostly with the aim of identifying groups of processes of differentiated characteristics. The target has typically been to obtain a representative process for each group and thus reduce the number of processes on which to carry out further analyses.

In [2] we aimed at using clustering techniques with the objective of identifying internal structure at the level of computation bursts within the application. We try to group computation bursts between MPI calls by their similarities in terms of duration and hardware counter derived metrics. In the following sections we describe the relevant data processing and clustering algorithms, usage examples clustering and further work on automatic quantification of the quality of a clustering result.

### *9.4.1 Clustering Algorithms*

Given our objective, the data to be clusterized corresponds to each of the computation bursts between MPI calls. For a typical trace there may be many thousands or millions of records, each of them characterized as a point in an N dimensional space. Possible dimensions include the duration of the region, the number of instructions, cache misses or other captured hardware counters. The number of these dimensions is limited by the number of hardware counters that can be simultaneously read, but derived metrics between several hardware counters such as IPC on miss ratios can also be used.

In order to keep the clustering algorithm in reasonable times and to focus the efforts in relevant regions we filter out bursts that are either very short or have a value of some of the counter below a threshold. The user can specify through an xml file these different thresholds as well as the metrics to consider as dimensions in the clustering algorithm, further data preprocessing transformations (ie. scaling, principal components,. . . ) or other parameters required by the algorithm.

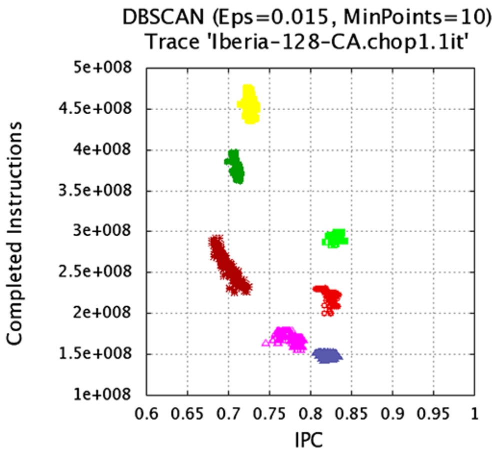

Fig. 9.2: Scatter plot of clustered WRF bursts

We use DBSCAN, a density based algorithm, as we have observed that the assumption made by k-means type of algorithms that data distribution is spherical in nature does not hold with our data. Figure 9.2 shows an example projection of points of a weather forecast run (WRF) on the Instruction and IPC dimensions. We can see how some clusters do have a spherical shape with little variability in both dimensions. Others show a negative correlation between instructions and IPC: the larger the instruction count in the bursts the lower the IPC. The reverse situation may hold on other cases. Clusters where the same number of instructions are executed with a wide range of IPCs are also frequent.

### *9.4.2 Application of Clustering Techniques*

The presentation of the scatter plots such as the one in figure 9.2 does provide a lot of information to the analyst on the behavior of the different regions, but the doubt may arise as to how do the identified clusters distribute over time. The tool can inject new events into the original tracefile labeling each computation burst with its identified cluster. In this way it is possible to visualize the space and time distribution of the clusters. This conveys to the analyst complementary information to the scatter plots, reflecting in detail the structure of the application behavior. Figure 9.3 shows the cluster timeline corresponding to figure 9.2.

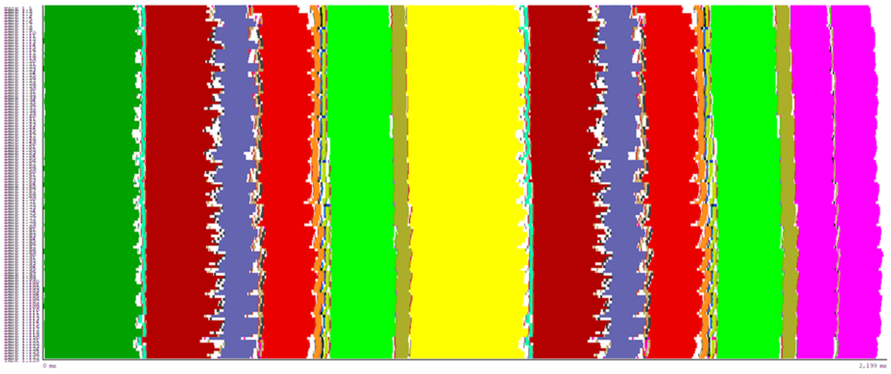
Fig. 9.3: Clustered WRF timeline

Deriving precise metrics and models of the performance of the sequential computation phases is another important use of the clustering techniques. Current processors do have the ability to perform very detailed counts of their internal activity and such information is made available through APIs, of which PAPI [10] is the most widely used. For cost reasons and although the list of potential counts is quite large, the actual number of counters that can be read at the same time is limited and architecture dependent. Being able to read many counters would also introduce significant overheads on the monitoring system. The result is that each data acquisition captures only partial information and in order to obtain values for a large set of counters, either several runs or sampling techniques have to be used. In these approaches, a lot of precision in the ability to correlate counts for individual regions of code is lost.

The use of the clustering techniques we have proposed provides an alternative to achieve higher precision still requiring only a single run of the program. The idea is to shift over time or processes the set of counters acquired with the constraint that a couple of them (typically cycles and instructions) be present in all sets. The acquisition does not even need to be synchronized across processes. The only requirement would be that a sufficiently large run is made such that several acquisitions with different hardware counter sets are made for the relevant computation bursts. If the application does have an SPMD structure, different sets can be used on different processors to reduced the required duration of the acquisition process. By clustering all the points using the two common counters, each point is assigned to one cluster and thus contributes its non common counters to the characterization of such cluster. The derived metrics computed from counts of different instances of a cluster are certainly approximations, but the accuracy is much higher than if the correlation is made through a blind statistical sampling process or derived from two independent runs.

An example use is show in figure 9.4. From a single run of the program, a very large number of hardware counters are obtained for the major clusters in the program. Those counters are fed into a cycles per instruction (CPI) stack model [11]

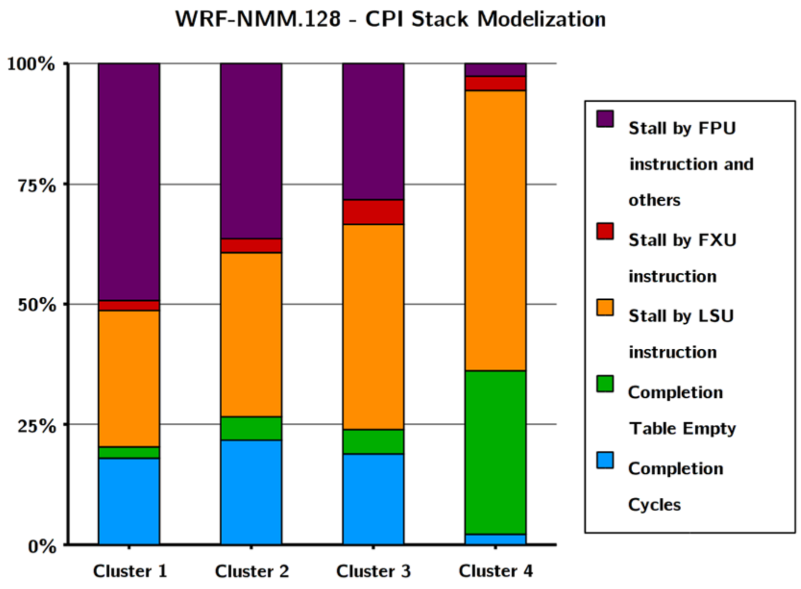

Fig. 9.4: CPI stack model derived from a single run

to obtain a fair quantitative description of which components in the architecture are determining the sequential computation performance of each cluster.

### 9.4.3 *Quantification of the Clustering Quality*

Given that most programs do have an SPMD structure, cluster timelines like the one shown above should display all processes as being in the same cluster at a given point in time. Of course some skew in the time each process enters a cluster is possible. Also different instances of the same cluster may be of different length in different processors, but in general, vertical stripes would be expected. When blindly applying the clustering algorithm it is nevertheless possible to obtain timelines like the one shown in figure 9.5. There we see that some parts do show an SPMD structure while in other intervals different processors are in different clusters. This is caused by real differences in the behavior of processors but the question is what level of granularity we want the clustering algorithm to use when determining what is similar and what different. A noisy plot with a lot of clusters, indicating that everything is different does not actually convey to the analyst useful information on the structure of the application. A plot with just one cluster, where everything is the same does not convey information either. The granularity used by the clustering algorithm is controlled by a parameter provided by the user. Although a wide range of values of such parameters give in general useful results for the DBSCAN algorithm, it is possible to tune it if coarser or finer detail is desired. Typically clustering result with a handful of clusters (less than 10), representing more than 90% of the total execution time and showing an SPMD structure could be considered as a good result for a first analysis.

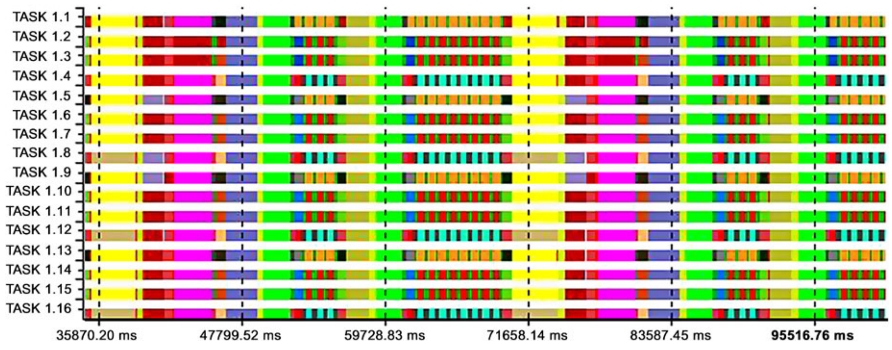

Fig. 9.5: Clustered timeline of WRF run on 16 processors

An interesting question is how the above criteria could be automatized. In [3] we presented an approach to automatically detect the *spmdiness* of a clustering result. The idea builds on the similarity between a process seen as a sequence of clusters and a sequence of aminoacids as handled in the life sciences area. In this field, a lot of tools have been developed to check the alignment and similarities between sequences of such chains. Our approach is to leverage that technology and use available Multiple Sequence Alignment (MSA) codes to properly align the different clusters executed by each process. In doing the transformation of the timeline to a sequence of clusters, the actual duration of the different instances is dropped and the focus shifts to their sequence, which in itself provides yet another way of representing the internal structure of the computation.

Figure 9.6 shows this situation for a run on 16 processors of the NAS LU benchmark (class A). On top we see the timeline of cluster while the bottom figure shows structure in terms of the aligned sequence of clusters as reported by the Kaling2 package. In this case we see that the time dimension has disappeared and each cluster uses only one column, irrespective of whether it was large or small. An asterisk on top of the image shows that the tool has been able to perfectly align that column. Lack of the asterisk means that there is either some hole in that column or not all the clusters are identical.

At the beginning of the timeline we observe several long clusters, with a quite good SPMD structure although task 6 is in a different cluster when all other processors are in the green cluster. In the bottom view of the figure we can see that this outlier causes the first non perfectly aligned column. Towards the end of the timeline each process executes a series of yellow clusters. It is unclear in such view whether all the processes have the same number of instances of the yellow cluster or not, and they are certainly skewed in time. The bottom figure shows how the alignment of those clusters is also quite good although some misaligned positions show up.

From the output of the alignment tool we can compute a metric specific to our purposes that quantifies with a number between 0 and 1 the *spmdiness* of each individual cluster as well as the global *spmdiness* of the clustering result. These metrics tell the analyst how good or bad the clustering result is and will in our future work be used as the quantification method to automatically search for a good granularity for the clustering algorithm. It will be possible to separate sets of points for which a coarse granularity results in good SPMD characterization from other sets where detailed granularity can extract finer detail and still reflect a good SMPD structure.

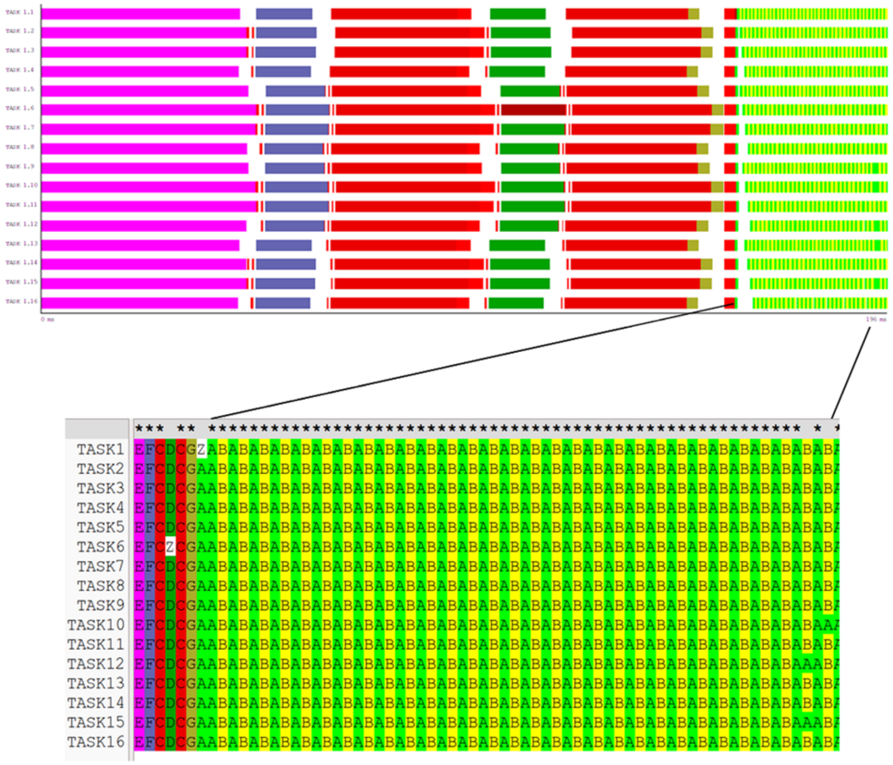

Fig. 9.6: Timeline and alignment of clusters for a 16 processor run of the NAS LU benchmark (class A)

## 9.5 Sampling and Mixed Instrumentation

Statistical sampling is a technique used by profile tools as a way to obtain approximate characterizations of a program without requiring to instrument neither the source nor binary. Sampling uses a mechanism external to the application control flow to fire a probe that from time to time captures information about the program such as the line in the code being executed. From these counts, we estimate the percentage of time spent in each of the lines by assuming it is proportional to such counts. A periodic sampling every few milliseconds is often used. Besides the time based approach, the firing of the probe can be based on the overflow of a hardware counter such as cache misses or floating point operations. In this case, the count for each line is expected to be proportional to the number of cache misses incurred or floating point operations performed by that line.

As mentioned before, the evolution of a program activity can be seen as a signal with spectral components corresponding to the speed of change in the activity of the program. As well known in signal theory, a function can be perfectly characterized by its samples if taken at sufficiently high frequency, in particular above the Nyquist frequency of the signal. In our context, this means that if the sampling frequency is sufficiently high compared to the rate of change in the application, we will be able to get a good picture of the evolution of metrics such as mips, cache misses, or other data captured during the sample.

Tools typically use either an instrumentation or a sampling approach, but typically not both at the same time. Even if they do, it is to obtain two types of information (ie. counts and percentage of time) which are then not correlated, at least as much as we consider it could be done. The following subsections show how the MPITrace package in CEPBA-tools has been extended to handle both instrumentation and sampled probes and the analyses this enables.

### *9.5.1 High Frequency Sampling*

Figure 9.7 shows different metrics obtained by sampling with a period of 1M cycles (roughly 445 microseconds) a 16 processors run of the NAS BT benchmark. The top view represents the main user functions and is derived from instrumentation information. The color scheme is as depicted in the top right corner of the figure. The view can be used as time reference for those below and shows how they represent about 1.5 iterations of the program. One iteration takes in the order of 345 ms, thus the number of samples in it is around 760, a large enough number to capture a lot of detail.

The four views on the bottom show the evolution over time of four metrics: mips, load mix (percentage of load instructions with respect to total number of memory access instructions), memory mix (with respect to total number of instructions) and the ratio of L1 to L2 misses. The figure shows how routines x_solve, y_solve and z_solve have an initial phase with four steps of high mips, and low memory mix.

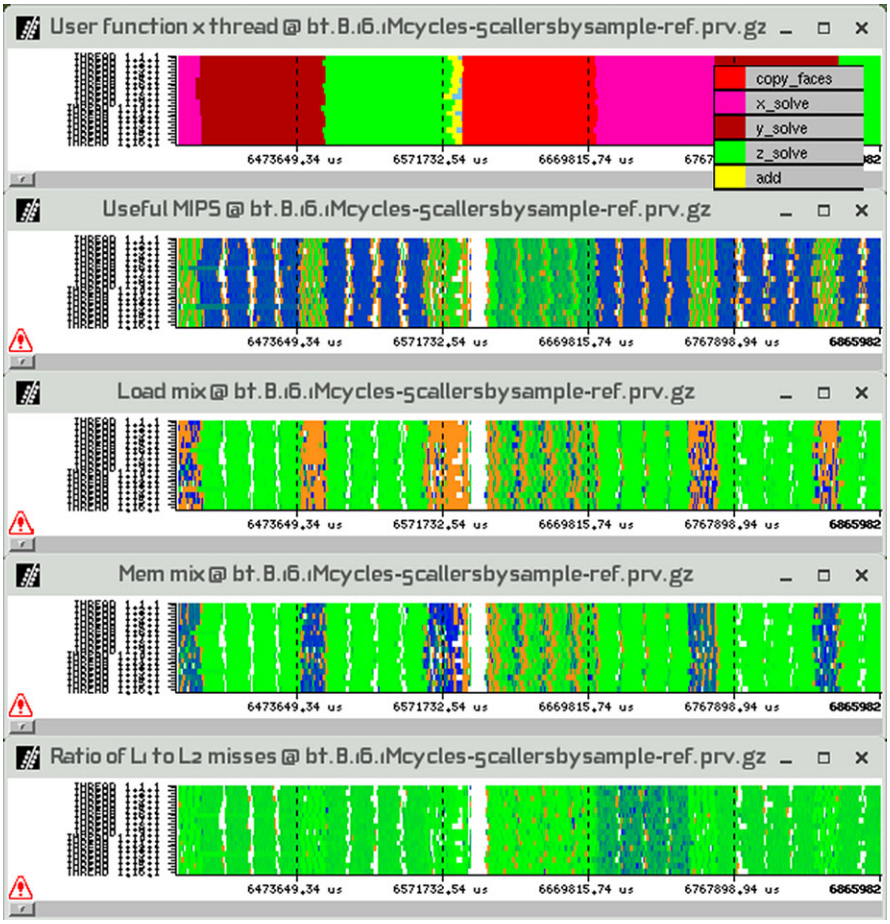
User function x thread @ bt.B.16.1Mcycles-5callersbysample-ref.prv.gz
copy_faces
x_solve
y_solve
z_solve
add
Useful MIPS @ bt.B.16.1Mcycles-5callersbysample-ref.prv.gz
Load mix @ bt.B.16.1Mcycles-5callersbysample-ref.prv.gz
Mem mix @ bt.B.16.1Mcycles-5callersbysample-ref.prv.gz
Ratio of L1 to L2 misses @ bt.B.16.1Mcycles-5callersbysample-ref.prv.gz
6473649.34 us
6571732.54 us
6669815.74 us
6767898.94 us
6865982

Fig. 9.7: Time evolution of metrics captured sampling at high frequency.Light green correspond to a low value of the metric, dark blue to a high value and orange to even higher values

It is interesting to see how routine x_solve has a higher L1 to L2 ratio of misses. Towards the end of each of these three routines, there is a phase where the memory mix, and in particular loads increases a lot, resulting in a lower mips rate.

In the above example, both instrumentation and sampling information are acquired by MPITrace, and they are visually correlated by the analyst. The instrumentation information can provide precise data about the structure of a program such as when a routine is entered and exited and how many instructions are executed in it. The granularity is nevertheless limited to the actual duration of these routines in the user code. The sampling information ensures a granularity (1M cycles in our example) at which data is acquired, even if the program stays within a routine without calling MPI for a long time. The problem is that if the sampling period is larger than the fine grain rate of change of the application it will not produce relevant information. The alternative of reducing the sampling interval to increase the precision is limited, as each sample implies a certain overhead to interrupt the process, capture and store the required information. In order to maintain the total overhead bounded, the sampling period should be a few orders of magnitude above the individual sampling overhead.

### *9.5.2 Hybrid Instrumentation and Sampling*

In this section we address the possibility of obtaining extremely precise information without incurring the overhead of very high frequency sampling. In [1] we developed a method that allows such precise measurements for hardware counter derived metrics under certain conditions in the application behavior, namely the ergodicity (maintaining the same periodic behavior over time). The proposed approach proceeds in three steps.

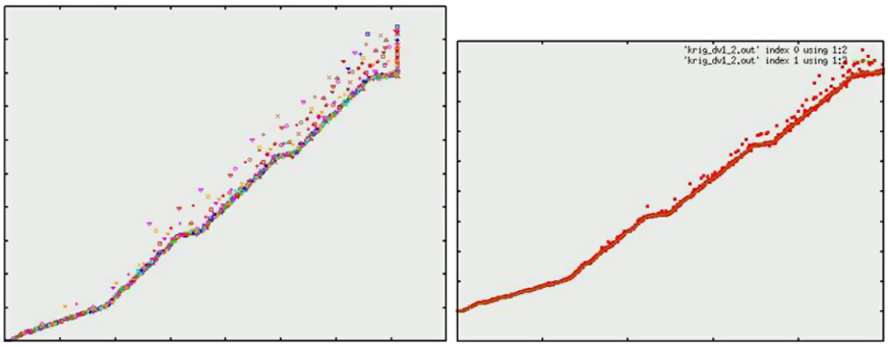

Fig. 9.8: Cumulative instruction count since the beginning of the iteration after the merging process. The left view includes all iteration instances. The right view show the detail when outliers have been discarded.

First we transform the captured hardware counts, which by default correspond to the value since the previous probe where the counters were read. This transformation is done by referring the counts at each sampled point to their nearest previous instrumentation event. By this we mean that we associate with such events the aggregated hardware count (instructions, cache misses,. . . ) of all the previous probes since the reference. The same aggregated value is computed for the instrumentation event at the end of the region (iteration or routine). In this way we obtain for each instances of a region a list of monotonically increasing hardware counts timestamped with respect to the start of the region. The number of such points for one instance may be just two (entry and exit) if no sample fell inside the region or a very large number if the sampling period is much smaller than the duration of the region. In any case, the precision of this data is limited by the sampling period.

The next step is to merge the different instances in order to increase the density of points and thus the precision. If all instances took exactly the same amount of time, just merging and sorting the different lists by the timestamp of the entries would be enough. As this situation will never happen in a real system, we take one instance as reference and scale the timestamps of the lists of all other instances such that the duration of the region matches the reference. After scaling we apply the merge process just described. If the variance in the duration of the different instances is not high, this should result in a thin cloud of points around the actual cumulative distribution of hardware counts since the start of the region. The upper plot in figure 9.8 shows the result of this process for one iteration of one process in the NAS BT benchmark class A run on 16 processes. We can see a certain amount of variability. In order to reduce it, we try to identify region instances that are outliers in terms of total duration and do not merge their points into the final list. The result is shown at the bottom of figure 9.8, with significantly less variability. More restrictive selection of outlyers would reduce variability but also the amount of points and thus precision. Finally, the region can be analytically characterized by a polynomial fit of the cloud of cumulative counts. We tried different fitting models but finally decided for a Kringing method. The analytical expression can then be reported as output of the analysis process. It can also be sampled at periodic intervals and synthetic events injected in the trace. It is also interesting that derivatives can be computed, thus reporting instantaneous rates such as mips or flops.

By using these hybrid sampling and instrumentation techniques, it is possible to compare the instantaneous evolution of metrics reporting how well different parts of the core architecture are being exercised. By correlating them we gain a deep understanding of the behavior of a program. Figure 9.9 shows the normalized instantaneous evolution (from top to bottom and left to right) of mips, store mix, L1 misses and load mix. We can clearly identify an initial phase where the mips rate is high and has a high proportion of store instructions but very low L1 miss rate. After that, four iterations of phases with a fair mips rate are separated by transitions with low mips. Such low performance in the transitions is correlated to a high L1 miss rate also caused by a high proportion of load instructions.

We have described the process to obtain the instantaneous evolution of metrics derived from hardware counter reads. It is also possible to apply the folding process to the call stack information captured by the samples to obtain a timeline of the code line being executed along time. In this case it is not possible to perform an analytical fit of the time function and variability between different iterations will introduce a certain degree of inaccuracy (i.e. backwards control flow). The result is nevertheless extremely useful as shown in figure 9.10. We can identify the four iterations of an outer loop and see how the execution progresses through the code with some source lines making longer contributions to the execution time than others.

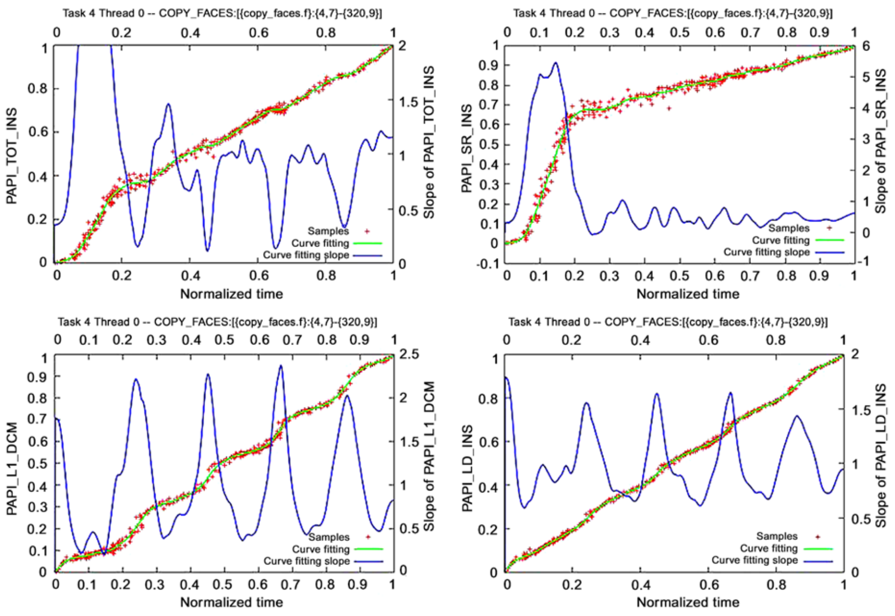

Fig. 9.9: Instantaneous evolution of different metrics for the copy_faces routine in the NAS BT benchmark

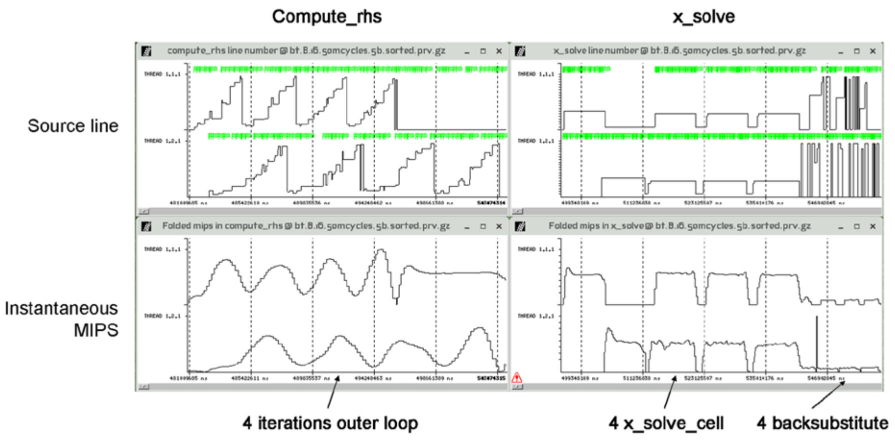
Compute_rhs
x_solve
Source line
Instantaneous MIPS
compute_rhs line number @ bt.B.16.50mcycles.5b.sorted.prv.gz
x_solve line number @ bt.B.16.50mcycles.5b.sorted.prv.gz
Folded mips in compute_rhs @ bt.B.16.50mcycles.5b.sorted.prv.gz
Folded mips in x_solve @ bt.B.16.50mcycles.5b.sorted.prv.gz
THREAD 1.1.1
THREAD 1.2.1
4 iterations outer loop
4 x_solve_cell
4 backsubstitute

Fig. 9.10: Correlation between metrics and folded source line for two processes in a 16 processors run of the NAS BT becnhmark

## 9.6 On-line Techniques

The research activities described in the previous sections heavily used Paraver to visualize traces and validate the results of the different techniques. The question then arises whether those techniques can be applied on-line, to automatically summarize the data captured by the monitoring system and minimize the size of files it generates while maximizing the amount of information emitted.

In [7] we describe ongoing work integrating those techniques into our instrumentation packages and using MRNET [9] as a scalable infrastructure. Such on-line integration does pose new challenges and the need to adapt the basic algorithms accordingly. Two of these extensions are described in [7] and will be summarized in the next paragraphs. The first one addresses the issue of the overhead of the clustering algorithm itself. The second one looks at the stability of the analysis. The objective of this work is to directly identify the clusters, automatically generating their scatter plot and clustered trace of size no larger than a user specified maximum.

The major problem that on-line clustering introduces is the duration of the analysis when a large number of points (typically above 50000) have to be clustered. In order to obtain a faster characterization of the application we sample a subset of the points, cluster them and then perform a nearest neighbor classification of all other points. If the sample is sufficiently small and representative the process should result in a good characterization of the application with a significantly faster execution time than the full clusterization of all the points. In the paper we evaluate different approaches to obtain a representative sample. A good way to achieve the desired characteristics of such sample is to keep all the points of some randomly selected processes plus a random sample of the points of all other processes. A sample of a size around 15% of the original set of points does result in very good characterization of all the relevant computation bursts in the trace.

The second issue addressed in the paper looks at how to detect that an application has entered a stable phase. By monitoring the raw data production rate of the application, the on-line analysis estimates the duration of the interval that would result in a trace of the size specified as target by the user. Such an interval is used by the MRNET root process to drive the instrumentation package at each process to send their captured information. The sampling process can take place in the intermediate nodes in the MRNET tree. The clustering takes place in the root. By comparing two successive clusterings the decision is made whether the application has reached a stable state. If not, a new acquisition period is started. If a stable behavior is identified, the root tells the leaves to dump the trace for the last period. Classification can take place in the leaves and the trace merging process could also be done through the MRNET tree.

It is quite natural that all the techniques described in previous sections are related and complementary. This work started by looking at the on-line use of the clustering techniques but it is clear that future work will further integrate the spectral analysis techniques described in section 9.3 and the sampling based techniques of section 9.5.

Other work in the spirit of on-line analysis is the integration of the sampling based techniques described in section 9.5 in the TAU [8] profile based environment. Figure 9.9 used in section 9.5 actually contains some preliminar results of such work.

## 9.7 Conclusion

This paper presents some attempts to leverage ideas and techniques from different areas such as signal processing and data mining in the area of performance analysis tools. It is based on the believe that a huge body of theory and experience has been developed in other fields that has not yet been applied to enable very precise and fine grained automatic analysis of application performance.

We have described the use of spectral analysis techniques and mathematical morphology to identify how long should we trace an application to obtain full detail of its behavior. Clustering has been applied to identify structure within an application and obtain from a single run very complete and precise statistics of all hardware counter metrics. The combined use of instrumentation and sampling has been used to demonstrate that it is possible to obtain extremely precise information of the evolution of instantaneous performance metrics such as mips without incurring overheads.

We do believe that these techniques will be further improved in the future but their potential is huge. Used in combination with many other techniques this will help evolve performance tools in the direction of minimizing the amount of data emitted by on-line monitoring but providing much more information than what is today's practice.

**Acknowledgements** This chapter summarizes the views of the author, but is based on work by many other people in the BSC tools team (specially Judit Gimenez, Harald Servat, Juan Gonzalez, German Llort and Kevin Huck). The work is partially funded by the IBM, through the IBM-BSC Mareincognito collaboration agreement, the USAF grant FA8655-09-1-3075 and the Spanish Ministry of Education under grant TIN2007-60625.

## References

1. H. Servat, G. Llort, J. Gimenez, J. Labarta: Detailed performance analysis using coarse grain sampling 2nd Workshop on Productivity and Performance. PROPER 2009.
2. J. Gonzalez, J. Gimenez and J. Labarta: Automatic Detection of Parallel Applications Computation Phases. Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium (IPDPS'09), (2009)
3. J. Gonzalez, J. Gimenez and J. Labarta: Automatic evaluation of the computation structure of parallel applications. PDCAT 2009.
4. Casas, M.; Badia, R. M.; Labarta, J. Automatic Structure Extraction from MPI Applications Tracefiles. Euro-Par 2007. 3–12

5. J. Labarta, S. Girona, V. Pillet, T. Cortes and L. Gregoris: DiP: A Parallel Program Development Environment. Proc. of 2nd International EuroPar Conference (EuroPar 96) (1996)
6. W. E. Nagel, A. Arnold, M. Weber, H. C. Hoppe and K. Solchenbach: VAMPIR: Visualization and Analysis of MPI Resources. Supercomputer, vol. 12, n. 1, 69–80, (1996).
7. G. Llort, J. Gonzalez, H. Servat, J. Gimenez and J. Labarta. On-line detection of large-scale parallel application's structure IPDPS 2010.
8. S. Shende and A. D. Malony: The TAU Parallel Performance System. International Journal of High Performance Computing Applications, Volume 20 Number 2 Summer 2006. 287–311
9. P. C. Roth, D. C. Arnold, and B. P. Miller: MRNet: A Software-Based Multicast/Reduction Network for Scalable Tools. SC2003, Phoenix, Arizona, November 2003
10. Browne, S., Dongarra, J., Garner, N., London, K., Mucci, P.: A Scalable Cross-Platform Infrastructure for Application Performance Tuning Using Hardware Counters. Proceedings of SuperComputing 2000 (SC'00), Dallas, TX, November 2000
11. A. Mericas et al.: CPI analysis on POWER5, Part 2: Introducing the CPI breakdown model. https://www.ibm.com/developerworks/library/pa-cpipower2/
12. Labarta J., Gimenez J.: Performance Analysis: From Art to Science. In Parallel Processing for Scientific Computing. M. Heroux and R. Raghavan and H.D. Simon Eds. SIAM. 2006. 9–32.