Application of clustering analysis and sequence analysis on the performance analysis of parallel applications

Author:
Juan González García

Advisor:
Prof. Jesús Labarta Mancho

A THESIS SUBMITTED IN FULFILMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
Doctor per la Universitat Politècnica de Catalunya
Departament d'Arquitectura de Computadors

Barcelona, 2013

Abstract

High Performance Computing and Supercomputing is the high end area of the computing science that studies and develops the most powerful computers available. Current supercomputers are extremely complex so are the applications that run on them. To take advantage of the huge amount of computing power available it is strictly necessary to maximize the knowledge we have about how these applications behave and perform. This is the mission of the (parallel) performance analysis.

In general, performance analysis toolkits offer a very simplistic manipulations of the performance data. First order statistics such as average or standard deviation are used to summarize the values of a given performance metric, hiding in some cases interesting facts available on the raw performance data. For this reason, we require the Performance Analytics, i.e. the application of Data Analytics techniques in the performance analysis area. This thesis contributes with two new techniques to the Performance Analytics field.

First contribution is the application of the cluster analysis to detect the parallel application computation structure. Cluster analysis is the unsupervised classification of patterns (observations, data items or feature vectors) into groups (clusters). In this thesis we use the cluster analysis to group the CPU burst of a parallel application, the regions on each process in-between communication calls or calls to the parallel runtime. The resulting clusters obtained are the different computational trends or phases that appear in the application. These clusters are useful to understand the behaviour of computation part of the application and focus the analyses to those that present performance issues. We demonstrate that our approach requires different clustering algorithms previously used in the area.

Second contribution of the thesis is the application of multiple sequence alignment algorithms to evaluate the computation structure detected. Multiple sequence alignment (MSA) is technique commonly used in bioinformatics to determine the similarities across two or more biological sequences: DNA or proteins. The Cluster Sequence Score we introduce applies a Multiple Sequence Alignment (MSA) algorithm to evaluate the *SPMDiness* of an application, i.e. how well its computation structure represents the Single Program Multiple Data (SPMD) paradigm structure. We also use this score in the Aggregative Cluster Refinement, a new clustering algorithm we designed, able to detect the SPMD phases of an application at fine-grain, surpassing the cluster algorithms we used initially.

We demonstrate the usefulness of these techniques with three practical uses. The first one is an extrapolation methodology able to maximize the performance metrics that characterize the application phases detected using a single application execution. The second one is the use of the computation structure detected to speedup in a multi-level simulation infrastructure. Finally, we analyse four production-class applications using the computation characterization to study the impact of possible application improvements and portings of the applications to different hardware configurations.

In summary, this thesis proposes the use of cluster analysis and sequence analysis to automatically detect and characterize the different computation trends of a parallel application. These techniques provide the developer / analyst an useful insight of the application performance and ease the understanding of the application's behaviour. The contributions of the thesis are not reduced to proposals and publications of the techniques themselves, but also practical uses to demonstrate their usefulness in the analysis task. In addition, the research carried out during these years has provided a production tool for analysing applications' structure, part of BSC Tools suite.

Agradecimientos

En primer lugar, me gustaría agradecer la oportunidad que me ha brindado mi director, Jesús Labarta, para poder realizar esta tesis en un centro de investigación único como es el BSC. Tampoco puedo olvidarme de mi co-directora moral, Judit, que aunque no pueda constar de forma oficial en todo el papeleo, ocupa sin duda un papel primordial en este trabajo. A los dos, muchas gracias por vuestra paciencia, consejos y guía.

Seguimos con los Tools, CEPBA y BSC, ese equipazo ecléctico donde los haya: Germán, Harald, Eloy, Pedro, Xavi(P). Han aguantado, sin rechistar, mis dudas, problemas, lloros, impertinencias y demás desvaríos. Tampoco debería olvidarme de los ex-Tools, Xavi(A) y Kevin. Pasar por este equipo es algo que se recuerda para siempre. Y también tenemos miembros honorarios, como Javi, con nuestros jueves de cañas.

Fuera del entorno laboral, estos últimos años nada habría sido igual sin la banda: tPR. Javi, Marcelo, Carral (y demás baterías) hemos hecho cosas muy guapas juntos. Gracias a vosotros he podido encontrar un espacio para poder expresarme con algo que no sean cifras y letras. Espero que sigamos adelante por mucho tiempo.

Tampoco puedo olvidarme de la gente de esgrima. Una afición desconocida para mi a la que me lancé en un mal momento, y ha resultado un descubrimiento mayúsculo. Por el deporte y por los amigos: Carles (capitán!), Jordis, Àngel, y David. A este último el agradecimiento es por partida doble, compañero de iniciación en el deporte y segundo co-director moral de esta tesis. Grandes conversaciones hemos tenido en el corto camino que une el Campus Nord con la Sala de Armas.

También debo citar aquí al ático de la calle Equador. He vivido en esa casa durante todo el transcurso de esta tesis, por lo que guarda una relación estrecha con este trabajo. Curiosidades de la vida, me mudo a la vez que finalizo la tesis. Justamente mientras escribo estas líneas me he enterado que ya ha sido vendido. Aquí debo acordarme de toda la gente que ha pasado, que no ha sido poca: Carli, David, Juanjo, Ramón, Valentina, Marina (a la que habría que dedicar un capítulo propio), Triinu, Anna, Laura, Lucia, Laia y David. Yo he sido la constante, vosotros lo variable.

Antes de acabar me gustaría meter en un saco a todos aquellos amigos, que son tan amigos que si nos los pones no se enfadan. Los de "la colla". Los primos. Los Erasmus. Ellos quizá no saben lo mucho que los quiero y que me importan. Si tuviera que agredecerselos uno a uno, no acababa.

Y por último, tengo que agradecer sin mayor duda el soporte de mis padres, mi hermana y su respectivo. Otra gran dosis de paciencia infinita. Supongo que de eso va el juego, pero ellos lo demuestran con creces. Suerte que la familia no se elige.

Esta tesis ha sido financiada por una beca FPI del Ministerio de Educación, contrato BES-2005-7919 (proyecto TIN2004-07739-C02-01), el proyecto de colaboración BSC/IBM MareIncognito y el proyecto de colaboración UE/Rusia HOPSA.

Contents

List of Figures

List of Tables

Part I.

Introduction and Related Work

1. Introduction

In this thesis we present novel techniques to characterize the performance of parallel applications. The work is mainly focused on those parallel applications that run on high performance computing (HPC) systems. This chapter presents the motivation for the research and also introduces the Performance Analytics field, where this thesis is included. Finally, the chapter also contains a list of the contributions of the thesis as well as the book organization.

1.1. Motivation

High Performance Computing and Supercomputing is the area of computing science that studies and develops the most powerful computers available. For example, the current #1 supercomputer in the world, according to the Top500.org list is Titan, a Cray supercomputer located at the Oak Ridge National Laboratory, Oak Ridge, Tennessee, United States. It has 560,640 processing cores (the smallest hardware unit capable of running one parallel process or thread) and its computation power is 17.590 PFlop/s. These figures are a good start to illustrate the challenges in this scenario. To achieve this computing at this scale, supercomputers include a huge number of factors that affect their performance: processors, mathematical accelerators, more or less sophisticated interconnection networks, I/O systems.

In the same way, the applications that run on these kind of computers also have many factors that affect their performance. First, to take advantage of the huge amount of compute power available, the applications must be parallel. Essentially, a parallel application is an application where parts of its code can be executed at the same time, *concurrently*, producing partial results that later combine solve a given problem. In practice, the design and implementation of these parallel applications involve many elements that affect the performance: the sequential algorithms implemented, the distribution of the data it uses, the communications patterns of the different parallel parts, etc.

As a result, to achieve the maximum performance that a supercomputer offers, the theoretical *peak* performance of the hardware, the developer of a parallel application must take into account this huge number of factors to know and understand how its application behave on it. This is the mission of the parallel performance analysis. In an application-centric approach, the performance analysis is a cyclic process consisting of observing the behaviour of the application so as to hypothesize the possible problems that affect its performance and finally translate these hypotheses to improvements in the application re-starting the cycle to validate them. Obviously, the less number of iterations of this cycle the less time wasted and also the less money spent.

The observation of the behaviour relies on capturing a certain amount of performance

data, that will be later analysed to define the hypotheses and validate or discard them. At the scales defined previously (hundreds of thousands cores, petaflops of computing power) the performance data generation during an application execution is huge: up to millions of performance events per second. Thus, it is necessary to increase the power and intelligence of the performance tools available to deal with this such amount of performance data and then reduce the analysis iterations. That represents the main motivation of this thesis: how maximize the knowledge of a parallel application performance when dealing with an enormous amount of performance data produced and also how to translate this knowledge to a performance tool that presents this information in an understandable way to the application analyst or developer.

1.2. Performance Analytics

Data Analytics is the science of examining raw data with the purpose of drawing conclusions about that information. Performance Analytics is Data Analytics applied to performance analysis data. As presented in the next chapter, there are a relatively short list of works included in the different analysis toolkits under the umbrella of this term.

In general, current performance analysis toolkits offer a simplistic manipulations of the performance data. First-order statistics such as average or standard deviation are used to summarize the values of a given performance metric, hiding in some cases interesting facts available from the raw data. We consider that Performance Analytics techniques are necessary so here we introduce the techniques we propose in this field.

1.2.1. Cluster Analysis in the Parallel Performance Scenario

When analysing a parallel application, the amount of performance data that is generated is huge. Summarizing the information according to a given criteria is then a necessity. Application profiling is the common technique to summarize the performance information. A profile is a simple accounting of first order statistics over a set of metrics associated to an application-level abstraction, for example the application subroutines. The major drawback when using profiles is that the time-varying behaviour is hidden. For example, a given subroutine could have different timings depending on duringo which phase of the application it is called. A profile will not make this distinction.

Considering the necessity to summarize the information in a more intelligent way that profiles do, we found cluster analysis to be better suited. As defined in [9], cluster analysis is the "unsupervised classification of patterns (observations, data items or feature vectors) into groups (clusters)". The fact that is a classification and not an aggregation of the information is a key factor to overcome the intrinsic problem of the profiles, so the different trends in the data are not hidden.

For example, a classic use of cluster analysis is to reduce the amount of information generated taken into advantage the repetitive patterns of parallel applications. In works such as [10, 11, 12], the authors exploited the structure of the Single Program Multiple Data (SPMD) paradigm that the vast majority parallel applications follow. In a SPMD applica-

tion it is expected that all processes/tasks involved perform the same sequences of computations/communications. In this context, cluster analysis is demonstrated to be effective to group those of processes/ tasks that behave similarly. Using a representative per cluster, the authors easily reduce the amount of performance data.

Our novel approach in the application of cluster analysis has a different target: determine the computational structure of the application. Instead of grouping the processes/tasks that behave similarly, we focus on the identification of the phases observed in the computation regions, i.e. those regions between communication primitives or calls to the parallel runtime, of a given parallel application. With our approach, we do not just take advantage of the SPMD pattern but also of the iterative applications design, for example in the wide variety of numerical methods used to solve equation systems. As a result, we obtain an small number of clusters characterize these repetitive structural computation phases, providing the developer/analyst an useful insight of applications computation behaviour.

Figure 1.1 compares these two different approaches. Picture 1.1a represents the different processes in a parallel application, where we have distinguished the computation (light grey) and communication (dark grey). Picture 1.1b represents the results of the approaches presented in [10, 11, 12]: considering for example of the duration of the processes, they detect two different clusters that group two processes each. Picture 1.1c represents our approach, where we group the different computation parts that appear in the processes, obtaining three different clusters according to the duration of the computation regions.

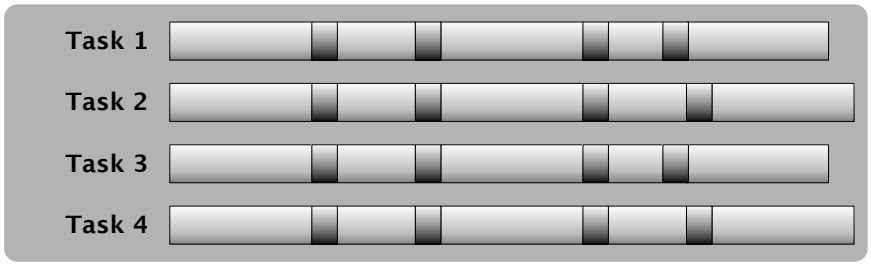
1.2.2. Sequence analysis in Parallel Performance Scenario

In bioinformatics, sequence analysis [13] is the process of extracting useful information from biological sequences, such as DNA chains or proteins. Multiple Sequence Alignment (MSA) is a sequence analysis technique able to determine the similarities across two or more sequences to define functional, structural or evolutionary relationships across them. These alignments are also used for non-biological sequences, such as those present in natural language [14] or in geographical studies [15].

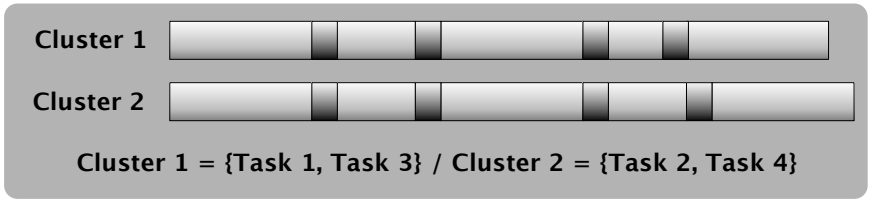
Using the brief description of a parallel application provided previously, we can make this simple analogy: each activity (compute or communicate) that each process/task of the application performs in parallel to solve the desired problem could be seen as a DNA base or amino-acid element, so the sequence of activities would be the equivalent to a DNA chain or a protein. Using a MSA algorithm will tell us how similar are the different processes/tasks in terms of the sequences they perform.

That is especially relevant when the application follows the SPMD paradigm, because it is expected to have the same sequence of activities for all processes/tasks involved. Our contribution to the Performance Analytics in this aspect is a quality score that measures the *SPMDiness*, i.e., how well it follows the SPMD paradigm of the structure of a given parallel application.

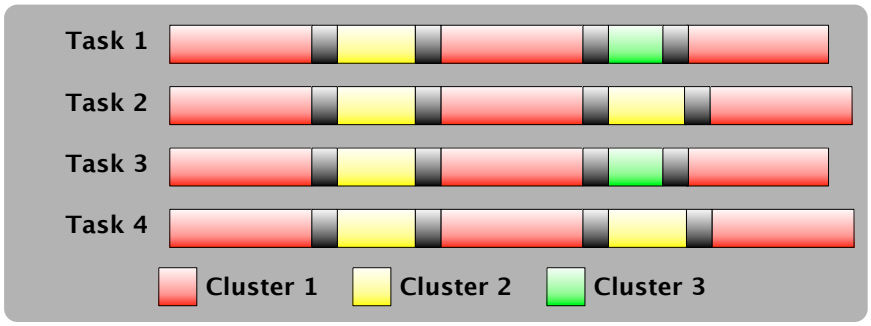
This quality score is linked with the computation structure characterization obtained using the cluster analysis. For example, we can use the sequence of clusters of each process/task as the input of a MSA algorithm that will evaluate if the computation phases detected represent the desired SPMD structure.



(a) Original sequence of computation/communication to analyse



(b) Cluster analysis to detect tasks/threads with similar behaviour



(c) Cluster analysis to detect computation regions with similar behaviour

Figure 1.1.: Comparison of cluster analysis targets

1.3. Contributions

As a summary of this chapter, we want to emphasize that this thesis introduces two new techniques in the Performance Analytics field, with the target of improving the knowledge of parallel application performance. More precisely, we present cluster analysis for detecting the computational structure and the application of sequence analysis to evaluate the SPMD structure.

The contributions of this thesis can be classified into two main categories. The first one corresponds to the technical contributions to create and validate the new techniques. The second category corresponds to the concrete examples of *applying* the techniques to different areas of performance analysis.

1.3.1. Technical Contributions

Demonstration of the suitability of density-based cluster algorithms to detect computation structure

We demonstrate the suitability of density-based cluster algorithms, specifically DBSCAN, when characterizing the computation structure of parallel applications. We show that this kind of algorithm surpasses the clustering algorithms based on K-Means when grouping performance hardware counters (HWC) data, and work more effectively than hierarchical clustering algorithms because they do not require an user interaction.

Design and validation of a cluster quality score for SPMD computation patterns detection

To fulfil the necessity of quantitative evaluate the computation structure a given application, we introduced the Cluster Sequence Score. This score is calculate by means of a Multiple Sequence Alignment (MSA) algorithm to evaluate the *SPMDiness*, i.e. how well the different sequences of actions in a parallel application follow the SPMD paradigm.

Design and validation of a new cluster algorithm to refine the computation structure detection

To improve the quality of the structure detection and focusing on producing a purely unsupervised cluster algorithm, we propose the Aggregative Cluster Refinement algorithm. It is a combination of a density-based algorithm and a hierarchical algorithm that maximizes the quality of a score. Using the Cluster Sequence Score, the Aggregative Cluster Refinement is able to detect SPMD computation regions in message-passing parallel applications at different levels of granularity, without user intervention.

1.3.2. Examples of applications of the techniques presented

Design and validation of a performance data extrapolation methodology using structure detection

In some cases, we need to execute an application several times to obtain different metrics that cannot be read simultaneously, for example when using processor hardware counters. We present the design and validation of a performance data extrapolation methodology that solves this problem by multiplexing the data acquisition along the (repetitive) application phases, and then extrapolates the average values per phase of a wide number of performance metrics.

Design and validation of a methodology to minimize the volume of the input data in a multi-level simulator of parallel applications

Detailed simulations of large scale message-passing parallel applications are extremely time consuming and resource intensive. We present the design and validation of a methodology where the structure detection is combined with signal processing techniques capable of reducing the volume of simulated data by hundreds to thousands orders of magnitude. This reduction makes possible detailed software performance analysis and accurate predictions in reasonable time.

Parallel applications what-if studies

Using the structure detection, we present a set of *what-if* studies of four production-class parallel applications. To perform these studies, we use the structure detection combined with an application level simulator. As a result, we show how what will be gained by improving the applications, without the need to actually changing the code, and also what will be the expected performance when porting the applications to different hardware, without requiring the real machines.

1.4. Dissertation Organization

The rest of this book is organized as follows. The Chapter 2 contains a discussion of the previous work in the parallel performance field, including the major work in the Performance Analytics field. Chapter 3 is an introduction to cluster analysis algorithms and multiple sequence alignment algorithms, important to understand the Performance Analytics techniques developed. In Chapter 4, we demonstrate the suitability of density-based clustering algorithms to effectively determine the computation structure of message-passing parallel applications. In Chapter 5, we present the Cluster Sequence Score, a score to evaluate the SPMD structure of a parallel application. In Chapter 6, we present Aggregative Cluster Refinement, a density-based clustering algorithm that detects the computation structure at fine grain, without user interaction. In Chapter 7, we present a methodology to extrapolate performance data using a structure detection to maximize the information extracted in a single

application run. In Chapter 8, we use the computation structure obtained combined with signal processing techniques to minimize the input data in a multi-level simulator. In Chapter 9, we present an analysis of four parallel applications by using the Aggregative Cluster Refinement. In Chapter 10, we discuss the contributions presented as a whole, describing the research and development opportunities that it offers to the community. In Appendix A, we detail the BSC-Tools suite, the performance analysis toolkit used to developed and validate the work presented in previous chapters. Appendix B contains the software engineering foundations of the `CluteringSuite` package, the software piece included in the BSC Tools suite that contains the implementation of the ideas and methodologies presented.

1.5. Publications

[16] J. Gonzalez, J. Gimenez, and J. Labarta. **Automatic Detection of Parallel Applications Computation Phases**. In *IPDPS '09: Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium*, Rome, Italy, May 2009

[17] J. Gonzalez, J. Gimenez, and J. Labarta. **Automatic Evaluation of the Computation Structure of Parallel Applications**. In *PDCAT '09: Proceedings of the 10th International Conference on Parallel and Distributed Computing, Applications and Technologies*, Hiroshima, Japan, December 2009

[18] J. Gonzalez, J. Gimenez, and J. Labarta. **Performance Data Extrapolation in Parallel Codes**. In *ICPADS '10: Proceedings of the 16th International Conference on Parallel and Distributed Systems*, Shanghai, China, December 2010

[19] J. Gonzalez, J. Gimenez, M. Casas, M. Moreto, A. Ramirez, J. Labarta, and M. Valero. **Simulating Whole Supercomputer Applications**. *IEEE Micro*, **31**:32–45, 2011

[20] J. Gonzalez, K. Huck, J. Gimenez, and J. Labarta. **Automatic Refinement of Parallel Applications Structure Detection**. In *LSPP '12: Proceedings of the 2012 Workshop on Large-Scale Parallel Processing*, Shanghai, China, May 2012

2. The Parallel Performance Analysis Field

Parallel performance analysis is a process that starts by extracting the raw performance data that will be later analysed to determine the potential problems of the application. In this context, we want to distinguish three important aspects that compose the whole process: when the analysis is done; the data used to characterize the performance; and finally, the different methods, techniques and applications to analyse this data to draw the conclusions.

2.1. Analysis Placement

The analysis placement refers to when the analysis is going to take place. Here we distinguish between *on-line* analyses, where the measurements and hypothesis are done during the application execution, and the *post-mortem* where the information is collected and stored to be later analysed when application finishes. We do not want discuss the detailed implications of the analysis location in the performance analysis process because, in most cases, the techniques or methods are applicable to both scenarios.

2.2. The Performance Data

To understand the behaviour of a parallel application on a given machine, we need to measure different elements that reflect its performance. The most simple and recurrent element measured is the application execution time. Undoubtedly, the execution time is a good indicator of the application performance, and in most cases is the objective value to minimize.

In the performance analysis scenario, it is also common to use time, but a finer grain, for example measuring the time spent on each application subroutine. To perform these measurements, we can make use of the timing mechanisms provided by the operating system (OS), for example the programmable clock interrupts. In addition, this example points to a second performance metric: the application code location. Location information is required to relate the metrics to the application source code. This location can be a single position where the measurement is taken, accessed via the Program Counter (PC) of the CPU, or the *call path*, that includes the list of active subroutines in the application, accessed unwinding the call stack, for example using `libunwind` [21].

Performance hardware counters (HWC) values are a unique metrics to understand the behaviour of the application in a given hardware. Hardware counters are available in almost

all modern processors, and count micro-architectural events such as the total cycles elapsed or the number of instructions executed. Hardware vendors provide the hardware and software interface to access these counters, but the PAPI [22] library, a homogeneous application programming interface to access the counters in most of the hardware, is the most common way to read them. More recently, due to the need to better understand new hardware, we can also find performance hardware counters in other components beyond CPUs, such as the Infiniband network hardware [23] or the Nvidia CUDA GPUs [24]. In both cases, the counters are also available via PAPI.

The metrics regarding the parallel programming model are varied, as well as the mechanisms to access them. For example, in the message-passing applications, the ones we analyse in this thesis, it is normal to gather the number of messages sent or received, the size of these messages, etc. In this case, if the application uses MPI, we have available the MPI profiling interface (PMPI) [25] to intercept the MPI calls and extract these values. In some other cases, access to the run-time of the programming models requires more sophisticated mechanisms, as detailed in next section when talking about instrumentation.

Apart from this list, there are a variety of metrics to evaluate specific performance elements (I/O, power consumption, etc.). It is not the aim of this section to present every performance metric available, but those commonly available in most performance toolkits.

2.2.1. Data Acquisition

We can distinguish two different methods to obtain performance data: *sampling* the application or *instrumenting* it.

Sampling consists of taking measures of the application status at regular intervals or when a certain condition happens. In this way, the measurements are taken independently from the application behaviour. Usually, for each sample, the information collected is the application location plus a set of quantitative metrics, such as the performance counters. Using sampling, the specific metrics of the programming model used are difficult to access.

When using sampling, the precision of the samples is directly related to the intrusiveness of the method: higher sample rates imply more precision but more overhead and perturbation of the results.

Application instrumentation consists of adding additional instructions or probes to the regular application code to capture those events and metrics that provide insight of the application behaviour, essentially entry or exit to application subroutines or the programming model run-time. As opposed to sampling, it is the application implementation that drives the data acquisition. In this way, the location is implicit, but sometimes the full call path is also recorded. As in sampling, other metrics such as the performance counters metrics are recorded when an event is captured.

The mechanisms to instrument an application are many. Instrumentation can be done manually by modifying the source code but also using more sophisticated methods: code to code translation with instrumentation injection, such as the OPARI [26] translator for OpenMP; rewriting application binaries as DynInst [27]; or dynamically modifying the application at run-time, also offered by DynInst and Intel's PIN [28]. It is also possible to use

the profiling interfaces of some libraries, such as PMPI, that provide access to those metrics of the programming model not available when using sampling.

2.2.2. Emitted Data

Before starting the actual analyses, there is classical dichotomy describing how the information is emitted and therefore stored: application profiles or event traces. The way the information is stored partially determines the kind of analyses available.

Application Profiles

An application profile is a summarization of a metric, or a set of metrics, that characterize an application-level abstraction. A typical example of a flat profile is the number of calls and the time spent in each application subroutine. In the process of summarization, the temporal component information regarding when the data was collected is lost.

`gprof` [29] is the classic tool for profiling sequential codes. This tool relies on compiler assisted instrumentation to define the bounds of the application subroutines and the POSIX timers and signalling to take the samples. It generates partial call graph profiles, more detailed profiles that express the caller-callee relationships across the application subroutines.

OpenSpeedshop [30] and the HPCTOOLKIT [31] extend `gprof` capabilities focusing on parallel applications. Both are able to manage the information generated by all tasks/threads involved in a parallel application, so as to present a single profile. In addition, they provide the ability to profile not only application subroutines but also basic blocks of code, or even single individual source code lines.

`gprof`, OpenSpeedshop and HPCTOOLKIT heavily rely on sampling mechanisms to perform the data acquisition. On the other hand, TAU [32] from the University of Oregon provides a profiling toolkit based on instrumentation mechanisms to allow the user a fine control on what is going to by profiled, including MPI run-time accesses and performance hardware counters. TAU also gathers *phase profiles*, partial profiles extracted at different *states* of the application execution. We can find in [33] a study of these different states or phases, into the IPS-2 analysis toolkit. In the case of TAU, the states are defined by the user and can be understood as the logical steps in the application evolution, for example the different time-steps in a weather forecast simulation. The phase profiles provide an approach to observe the time-varying behaviour of an application.

The Scalasca toolkit [34] also offers the collection of regular profiles with metrics from MPI, OpenMP and hardware counters using instrumentation. In this toolkit ecosystem we also find an interesting effort gathering *time-series call-path profiles* [35], call-graph oriented version of the previously mentioned phase profiles.

Event Traces

An event trace is the collection of all information gathered during the application execution, consisting of a log or trace-file, of the actions that a parallel application performed. These actions, emitted as time-stamped events, include the entry and exit to the subroutines or to

the parallel libraries used, values of the hardware counters or the call-path that lead to the point of interest. Tipically, the generation of event traces relies on instrumentation packages to define the points of interest, but we can also find sample traces. Event traces offer a highly detailed view of the performance, at the cost of high storage space requirements. They are especially interesting to analyse the time-varying behaviour of the application, information that is totally lost when using profiles.

In general, almost all parallel performance analysis toolkits offer the extraction of event traces as the base to the further analysis. For example the BSC Tools package, the toolkit used in this thesis, is based on Paraver trace [36]. This trace is a structured text file whose main characteristic is that it is semantic free: the events contained are essentially time-stamped key/value pairs. A complementary (optional) file is the responsible for linking the semantics to the tuples contained on the trace file. A detailed description of the Paraver trace is available in Appendix A.

Recently, as a part of the Score-P initiative [37], the analysis toolkits Scalasca, Vampir [38], Periscope and TAU decided to adopt the Open Trace Format version 2 (OTF2) [39], the second generation of the Open Trace Format [40]. OTF2 is structured in a collection of multiple binary files accessible via an API. The main concern in the design of this trace is the scalability: the trace format definition includes a series of encoding techniques to reduce its size [41], and the access API uses techniques to reduce the memory footprint.

As mentioned before, there are also some efforts to produce *sample traces*. As opposed to the regular traces whose events are associated to instrumentation points, sample traces contain a series of time-stamped samples taken regularly during the execution. The information in each sample mainly includes the call-path, to correlate to application source code, and performance metrics, such as the hardware counters. We find an early approach to the generation of sample traces inside the Sun Studio Performance Tools (now Oracle Studio) [42]. Recently, the HPCTOOLKIT and the BSC Tools also added sample traces on their toolkit ecosystems, [3] and [6] respectively. It is interesting to highlight that the addition of samples to the Paraver trace described in [6] did not imply any modification of the trace format.

2.3. Analysing the Performance Data

In a post-mortem scenario, once the data has been collected, the analysis step consists of exploring the information gathered, manually or assisted, to detect patterns or trends that reflect anomalies or performance losses in the application behaviour and correlate them with the possible causes.

At this point, we distinguish between two different elements of the analysis itself. The first is how the information is presented to the analyst to make the analysis process understandable and manageable. The second is the collection of Performance Analytics techniques available, i.e. the different techniques developed to automate the processing of the raw data so as to detect anomalies and correlations.

2.3.1. Data Presentation

Data presentation is a key element of the analysis process. The way the information is offered to the analyst defines, to some extent, the possible observations and hypotheses they could make about the performance achieved by the application. Undoubtedly, the way data is emitted imposes a series of restrictions or opportunities on its presentation.

Again, due to the (potentially) huge volume of data generated, this step may include a series of transformations or manipulations to present the information in a clear and concise manner.

Presentation of Profiles

Application profiles are an explicit example of this data manipulation *per se*. The different measurements taken during the execution are accounted in first order statistics such the sum or the average at the selected application-level abstraction (subroutines, application phases, etc.). Using profiles, the exploration for the unusual situations can be done easily. For example sorting total time spent in the different subroutines will point us where are the hotspots. The weak point of profiles is that the aggregation hides the potential variability across the instances accounted, but for a initial look of the performance, profiles are a good choice.

In general, profiles are presented as human-readable plain text files. The text is indented to categorize the elements accounted more easily and they are sorted with respect to one of the metrics used to focus the analysis. In Figure 2.1 we can see a flat profile produced by `gprof`. It contains the different metrics calculated for each subroutine, right-most column, sorted by the percentage of total application time spent each one represent, left-most column. The rest of the metrics include the exclusive/inclusive times (time inside the subroutine excluding or including the calls it makes) and number of calls. The call-graph profile in Figure 2.2 contains similar metrics, but the right-most column presents the caller-callee relationship using the indentation. These call-graph profiles generated by `gprof` can also be visualized in a interactive GUI, using a tree representation, with the IBM's Xprofiler [43].

In contrast to the simple plain-text document, we find tools such as the `hpcviewer` [44], an evolution of the HPCVIEW [45], which is the profile visualization tool of the HPCTOOLKIT. The `hpcviewer` is a GUI that offers a clean presentation of the different metrics gathered in the HPCTOOLKIT profiles, with the ability to link the metrics with the source code, shown in Figure 2.3, or present the metrics using charts for a better comprehension, shown in Figure 2.4.

ParaProf [46], the profile analysis tool of the TAU package, offers similar functionality to the `hpcviewer`, in the areas of source code correlation and the generation of different chart types. In Figure 2.5, we can see an example of a bar-chart presentation of the most time consuming application subroutines, including the MPI primitive calls. In addition, ParaProf is also capable of comparing profiles obtained from different executions of the same application. For this purpose, TAU uses their PerfDMF [47], a framework to manage profiles from different executions using a database that eases the comparison of profiles. Figure 2.6 shows a ParaProf window comparing the most time-consuming subroutines of three threads from

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
time   seconds   seconds    calls  ms/call  ms/call  name
33.34      0.02     0.02     7208     0.00     0.00  open
16.67      0.03     0.01      244     0.04     0.12  offtime
16.67      0.04     0.01        8     1.25     1.25  memccpy
16.67      0.05     0.01        7     1.43     1.43  write
16.67      0.06     0.01                             mcount
 0.00      0.06     0.00      236     0.00     0.00  tzset
 0.00      0.06     0.00      192     0.00     0.00  tolower
 0.00      0.06     0.00       47     0.00     0.00  strlen
 0.00      0.06     0.00       45     0.00     0.00  strchr
 0.00      0.06     0.00        1     0.00    50.00  main
 0.00      0.06     0.00        1     0.00     0.00  memcpy
 0.00      0.06     0.00        1     0.00    10.11  print
 0.00      0.06     0.00        1     0.00     0.00  profil
 0.00      0.06     0.00        1     0.00    50.00  report
```

Figure 2.1.: Example of flat profile produced by `gprof`. Extracted from the `gprof` manual[1]

```
granularity: each sample hit covers 2 byte(s) for 20.00% of 0.05 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.00    0.05                 start [1]
                0.00    0.05       1/1           main [2]
                0.00    0.00       1/2           on_exit [28]
                0.00    0.00       1/1           exit [59]
-----------------------------------------------
                0.00    0.05       1/1           start [1]
[2]    100.0    0.00    0.05       1         main [2]
                0.00    0.05       1/1           report [3]
-----------------------------------------------
                0.00    0.05       1/1           main [2]
[3]    100.0    0.00    0.05       1         report [3]
                0.00    0.03       8/8           timelocal [6]
                0.00    0.01       1/1           print [9]
                0.00    0.01       9/9           fgets [12]
                0.00    0.00      12/34          strncmp <cycle 1> [40]
                0.00    0.00       8/8           lookup [20]
                0.00    0.00       1/1           fopen [21]
                0.00    0.00       8/8           chewtime [24]
                0.00    0.00       8/16          skipspace [44]
-----------------------------------------------
[4]     59.8    0.01        0.02       8+472     <cycle 2 as a whole> [4]
                0.01        0.02     244+260         offtime <cycle 2> [7]
                0.00        0.00     236+1           tzset <cycle 2> [26]
-----------------------------------------------
```

Figure 2.2.: Example of call-graph profile produced by `gprof`. Extracted from the `gprof` manual[1]

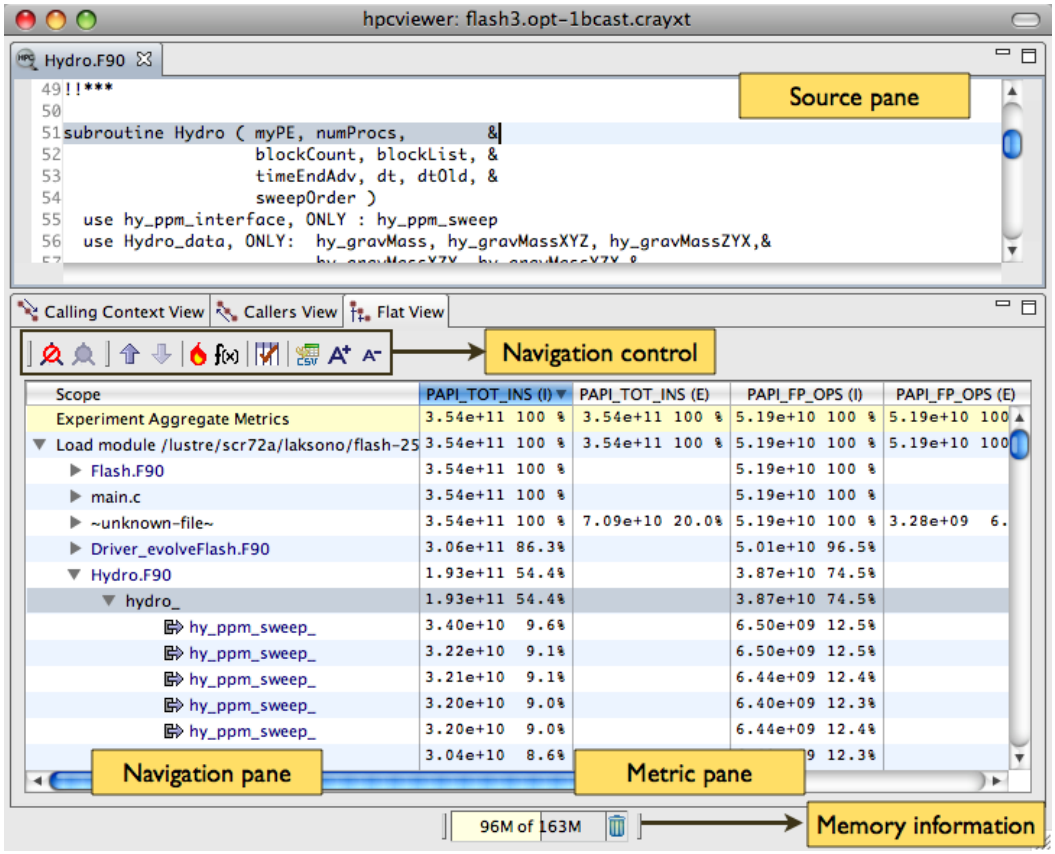


Figure 2.3.: Annotated HPCTOOLKIT's `hpcviewer` main interface. Image obtained from HPCTOOLKIT manual[2]

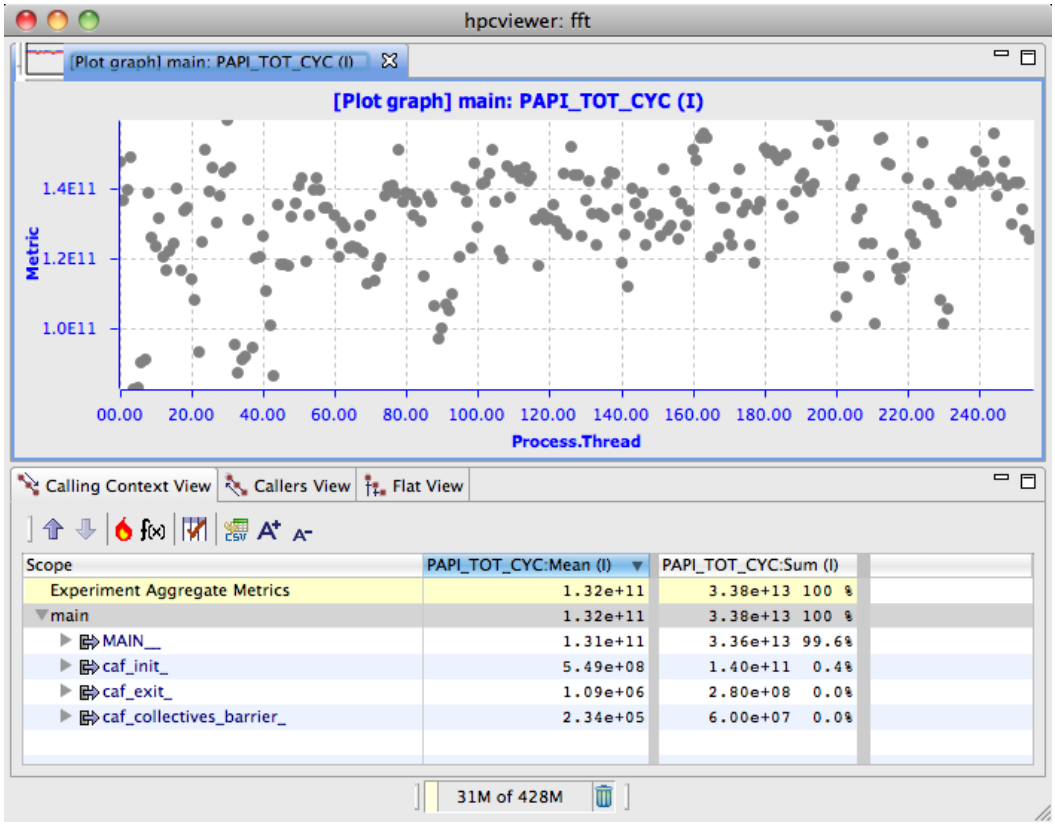


Figure 2.4.: Example of scatter plot of Completed Instructions counter produced by HPCTOOLKIT's `hpcviewer`. Image obtained from HPCTOOLKIT manual[2]

three different executions of the same application with different configurations.

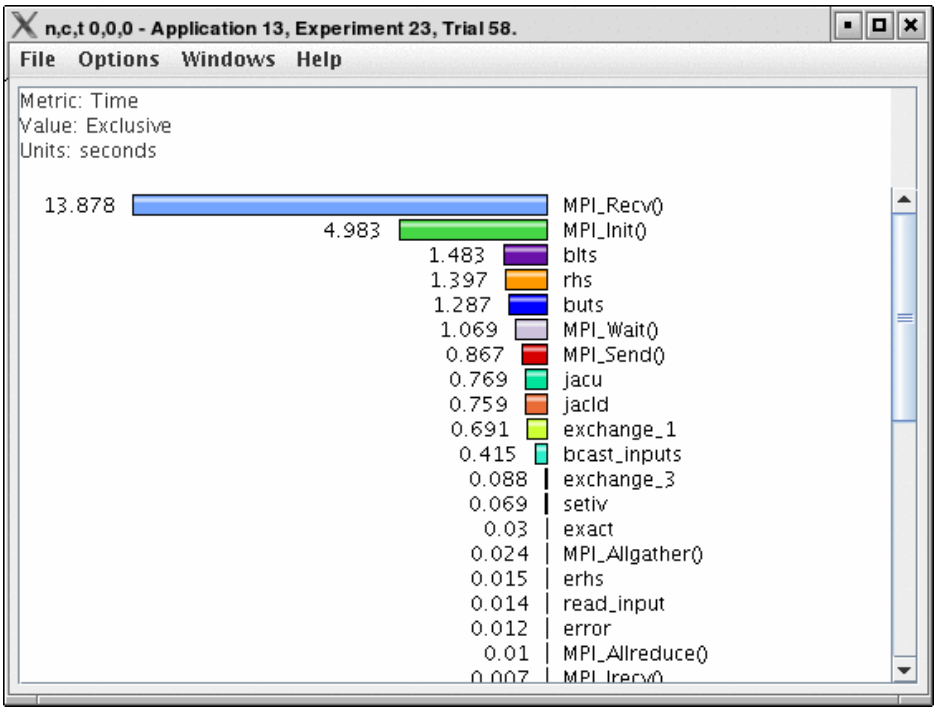


Figure 2.5.: TAU's ParaProf thread hotspots profile window

Finally, Scalasca's CUBE [48] visualizer has an original way to visualize and interact with profiles. The main display of this GUI is divided in three parts (see Figure 2.7). The left part shows the different metrics gathered; the central part is the code location; and the right part shows the system location, i.e. the accounting of each metric in the physical hardware. Thanks to the colouring, the user can see quickly the severity of a given metric, viz. low, regular or high values. In the example shown, the metric selected was the time spent in 'Point-to-Point' message-passing primitives (left column), which has a relatively low severity (yellow). This metric appears in several subroutines in the call-path tree (central column) with medium severity, having the selected one, `hsmoc`, a medium high value in the visible range. Finally, the value of "Point-to-Point" time spent in the `hsmoc` subroutine is distributed uniformly in all the system (right column), with relatively high severity.

Event Traces Presentation

As opposed to profiles, event traces have greater potential to detect elements at finer granularity, as they keep track of all the actions performed by the analysed application, including the spatial and temporal distribution of them. On the other hand, traces require more effort both in the manipulation and the exploration to detect the anomalies and correlate them to the root causes.

To aid the analyst, there are available several interactive GUIs such as Vampir [38], Paraver or the `hpctraceviewer`. In terms of exploration, all these tools provide a unique visualization, impossible to obtain when using profiles: the time-line. A time-line is a representation

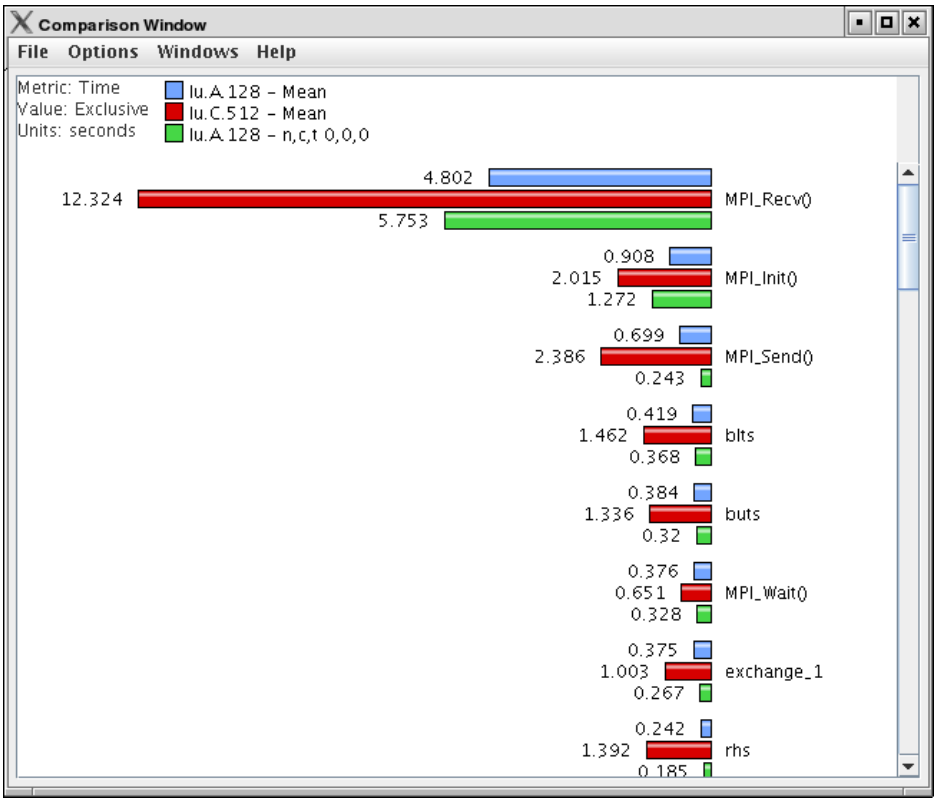


Figure 2.6.: TAU's ParaProf thread profiles comparison

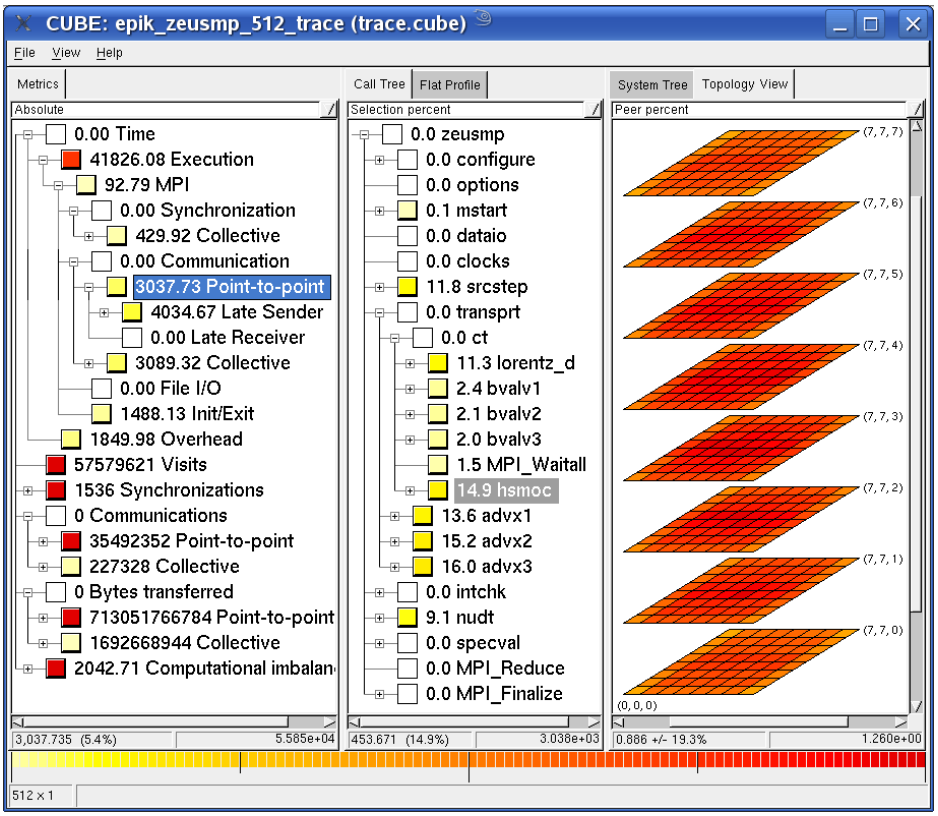


Figure 2.7.: Scalasca's CUBE main interface

where the information available in the trace is organized in a bi-dimensional plot, applications abstractions such as tasks or threads versus time, and coloured according to a metric.

Figure 2.8 contains an example of the Vampir master time-line, presenting an OTF trace. In this case, the Y axis represents the processes of a message-passing application, and the colouring indicates the subroutine executed. We can also observe black lines that represent the point-to-point messages passed between the different processes. In the top left part of the window we can see a general view of the whole trace, being the main-time line just a zoom of the central part.

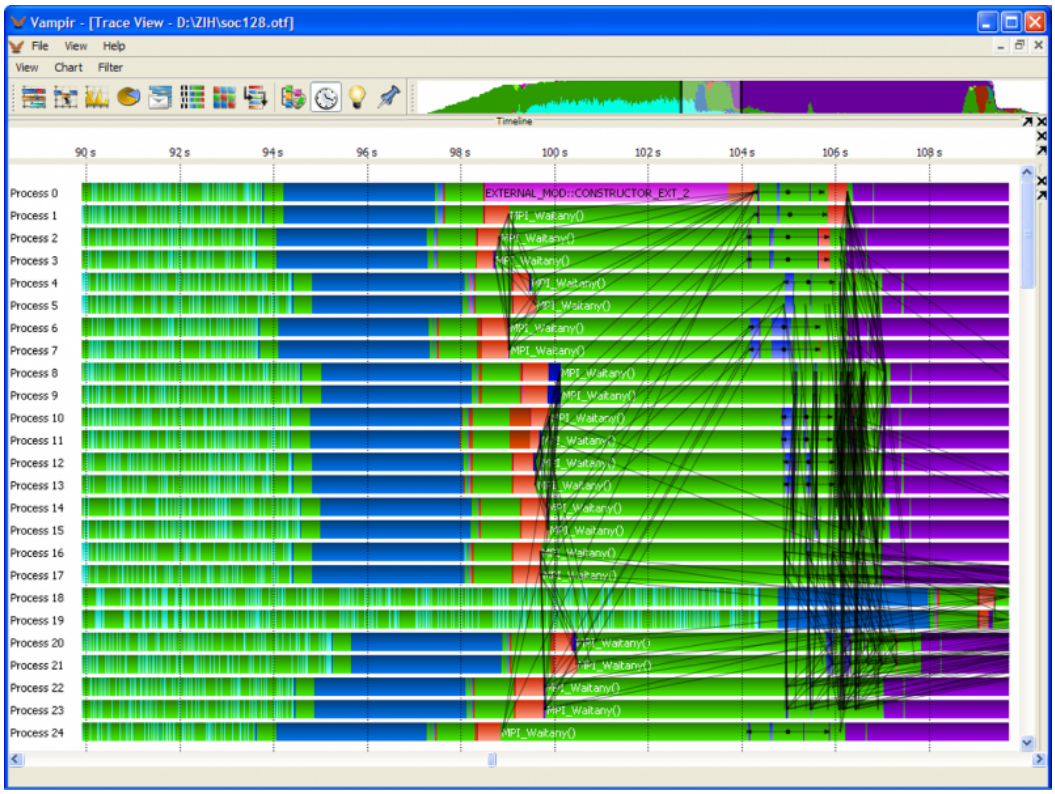


Figure 2.8.: Master time-line of VAMPIR trace analyzer

Figure 2.9 contains the equivalent window of the `hpctraceviewer`. In this case, since sample trace it presents does not have information regarding the point-to-point communications, the time-line does not include communication lines. On the other hand, in the bottom part of the window it present the depth of the call-path obtained on each sample of the trace.

Finally, in Figure 2.10 we can see an example of Paraver time-line. In this example, the metric selected was the Instructions per Cycle (IPC), derived from the Completed Instructions and Total Cycles hardware counters present on the trace. The colouring is a gradient from light green (low IPC) to dark blue (high IPC). Paraver traces also contain point-to-point messages information, and these are depicted as yellow lines in the time-line.

Even though the time-line representation is useful to explore the time distribution of a raw (or derived) metrics at different levels of granularity using zooming, there are limitations to this representation: first, current screen resolutions limit the amount of data that can be

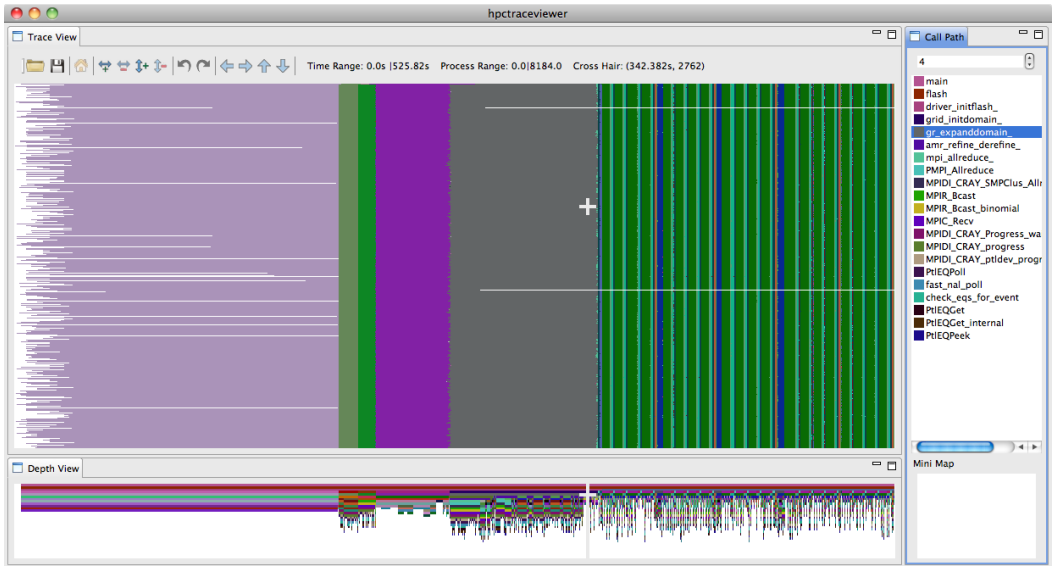


Figure 2.9.: `hpctraceviewer` interface. Picture obtained from [3]

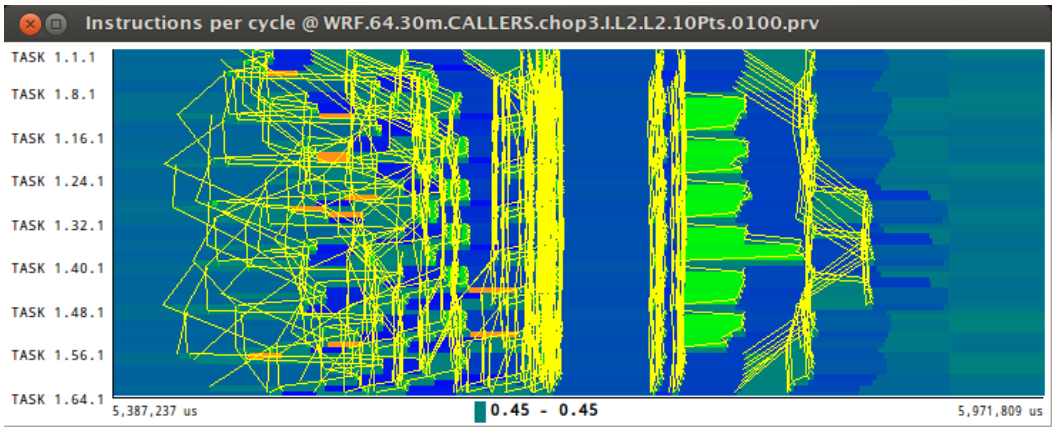
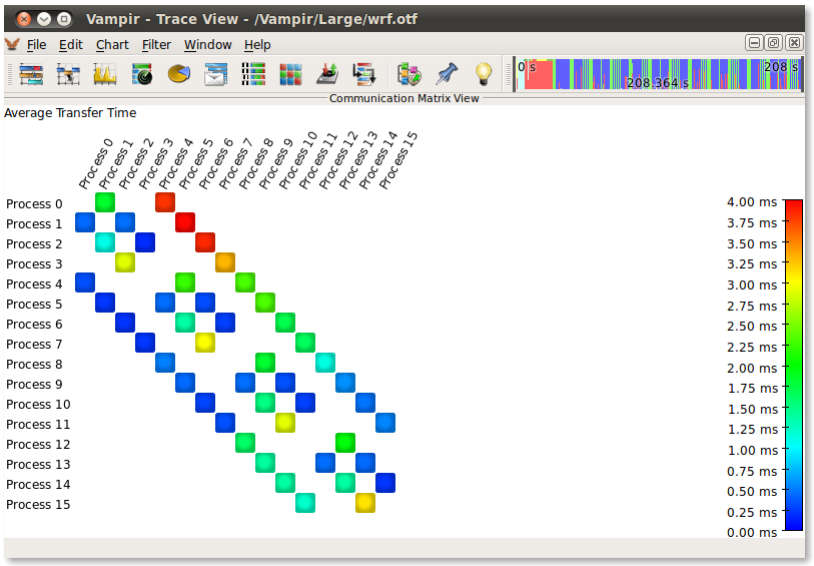
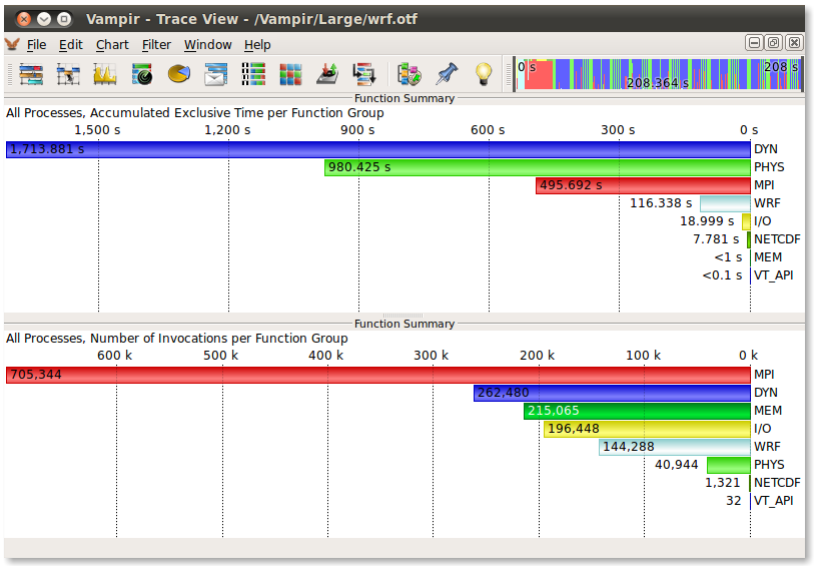


Figure 2.10.: Paraver time-line window

presented; second, there is a biological limit to distinguish differences the colour hue. These limitations imply that the representation requires a big effort in processing the input data to render how each pixel of the bitmap is filled. For example, a single pixel of the screen may represent more than one object or a wide range of time, so an algorithm is required to decide how depict this the actual value. For example, non-linear renderings of the data ranges are used to clarify the representation of the different values presented.



(a) Communication statistics summary

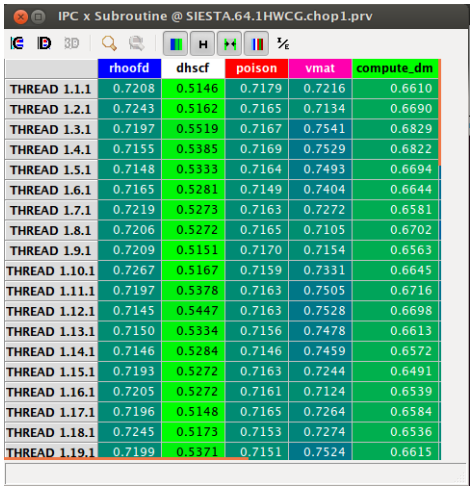


(b) Subroutine profiling summary

Figure 2.11.: Vampir summary windows

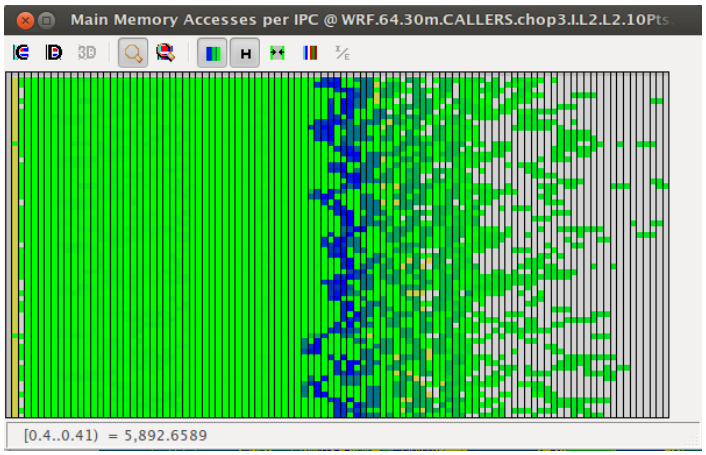
To overcome these constraints, the trace analysers, mainly Vampir and Paraver, also include a set of features to manipulate the information available in the event trace. In the case

of Vampir, it offers a battery of predefined summaries that include the generation of communications profiles, Figure 2.11a, and also flat-profiles, Figure 2.11b. Paraver also includes a profiling view, but extends the regular summarization with the ability to freely combine different metrics available, to detect the possible correlations across them. For example, in Figure 2.12a, there is a window of a profile showing the average IPC obtained by each application subroutine (columns) for each application thread (rows). In Figure 2.12b, we can see a complex histogram where the X axis represents ranges of values of the IPC, the Y axis represents the different threads of the application and the colouring expresses the number of L2 data cache misses (using the green-to-blue gradient). Even these two tools differ in the features to combine performance metrics, both are able to compute a wide range of statistics, similar to profiles: sums, averages, standard deviations, etc.



	rhoofd	dhscf	poison	vmat	compute_dm
THREAD 1.1.1	0.7208	0.5146	0.7179	0.7216	0.6610
THREAD 1.2.1	0.7243	0.5162	0.7165	0.7134	0.6690
THREAD 1.3.1	0.7197	0.5519	0.7167	0.7541	0.6829
THREAD 1.4.1	0.7155	0.5385	0.7169	0.7529	0.6822
THREAD 1.5.1	0.7148	0.5333	0.7164	0.7493	0.6694
THREAD 1.6.1	0.7165	0.5281	0.7149	0.7404	0.6644
THREAD 1.7.1	0.7219	0.5273	0.7163	0.7272	0.6581
THREAD 1.8.1	0.7206	0.5272	0.7165	0.7105	0.6702
THREAD 1.9.1	0.7209	0.5151	0.7170	0.7154	0.6563
THREAD 1.10.1	0.7267	0.5167	0.7159	0.7331	0.6645
THREAD 1.11.1	0.7197	0.5378	0.7163	0.7505	0.6716
THREAD 1.12.1	0.7145	0.5447	0.7163	0.7528	0.6698
THREAD 1.13.1	0.7150	0.5334	0.7156	0.7478	0.6613
THREAD 1.14.1	0.7146	0.5284	0.7146	0.7459	0.6572
THREAD 1.15.1	0.7193	0.5272	0.7163	0.7244	0.6491
THREAD 1.16.1	0.7205	0.5272	0.7161	0.7124	0.6539
THREAD 1.17.1	0.7196	0.5148	0.7165	0.7264	0.6584
THREAD 1.18.1	0.7245	0.5173	0.7153	0.7274	0.6536
THREAD 1.19.1	0.7199	0.5371	0.7151	0.7524	0.6615

(a) Profile Metrics window (showing average IPC per subroutine per application thread)



(b) Metrics Combination Histogram (showing Main Memory Accesses per IPC range per application task)

Figure 2.12.: Paraver statistics windows

2.3.2. Performance Analytics

As a summary of previous section we can conclude that application profiles provide a coarse-grain knowledge of the application but do not require a big expertise when looking for possible hotspots. So, when going into detail, event traces are necessary, at the cost of requiring big expertise to manipulate the available information in order to detect the performance problems and correlate them to their causes. The goal of the Performance Analytics is to join the best of both worlds. In other words, to be able to get the deep insight provided by event traces without requiring the expertise to manipulate the *huge amount* information.

The Performance Analytics group the techniques that translate the expert knowledge into algorithms or methodologies able to automatically extract the most valuable information about applications performance. In general, these techniques rely on the search patterns or trends that characterize the performance losses. The search can be done at different granularities and at different levels of abstraction obtaining different analysis results.

Expert Systems

A methodology widely exploited is the use of a rule-based inference system to detect well-known problems of parallel applications, [49]. An example of rule can be: "IF the time spent by a task waiting to send a message send is bigger than x because the partner task have not executed the receive operation THEN *late receiver* problem detected". Usually, the rules include a *severity* value, indicating the importance of the detected problem into the performance of the whole application.

As a part of the Scalasca toolkit, EXPERT [50, 4, 51] is a sequential post-mortem rule-based system to identify the wait states in message-passing applications, using the categorization depicted in Figure 2.13. In this work, the set of rules of known wait states, i.e. when the application is wasting time without advancing in the resolution of the problem, is defined using the EARL script language and are applied to the performance metrics stored in OTF traces. As a result, EXPERT generates profiles that can be visualised in CUBE. In the profiles, EXPERT accounts the occurrences of each problem in the knowledge base on each of the subroutines. Further research on EXPERT system include a parallel implementation [52] and an extension of the rule base in [53] to track the *root cause* of the wait states detected, i.e. those regions or points in the application that later provoke wait states when application communicates.

KappaPI [54] and KappaPI 2 [55], are two versions of the same rule-based tool with the difference that KappaPI includes the set of rules hard-coded, while KappaPI 2 brings the user the possibility of writing his own rules, using the APART Specification Language (ASL) [56]. This language has been adopted in some of the rule-based due to to its power to easily express the performance problems. An interesting feature of both tools is the recommendation system, that shows possible solutions to the detected problems.

The SCALEA toolkit [57] also provides similar features than Scalasca toolkit, with the support of a multi-experiment environment based in a database storage as PerfDB. Aksum [58, 59] makes use of the data extraction mechanisms provided by SCALEA to implement its own rule-based system. In this case, the rule set used has to be specified in JavaPSL,

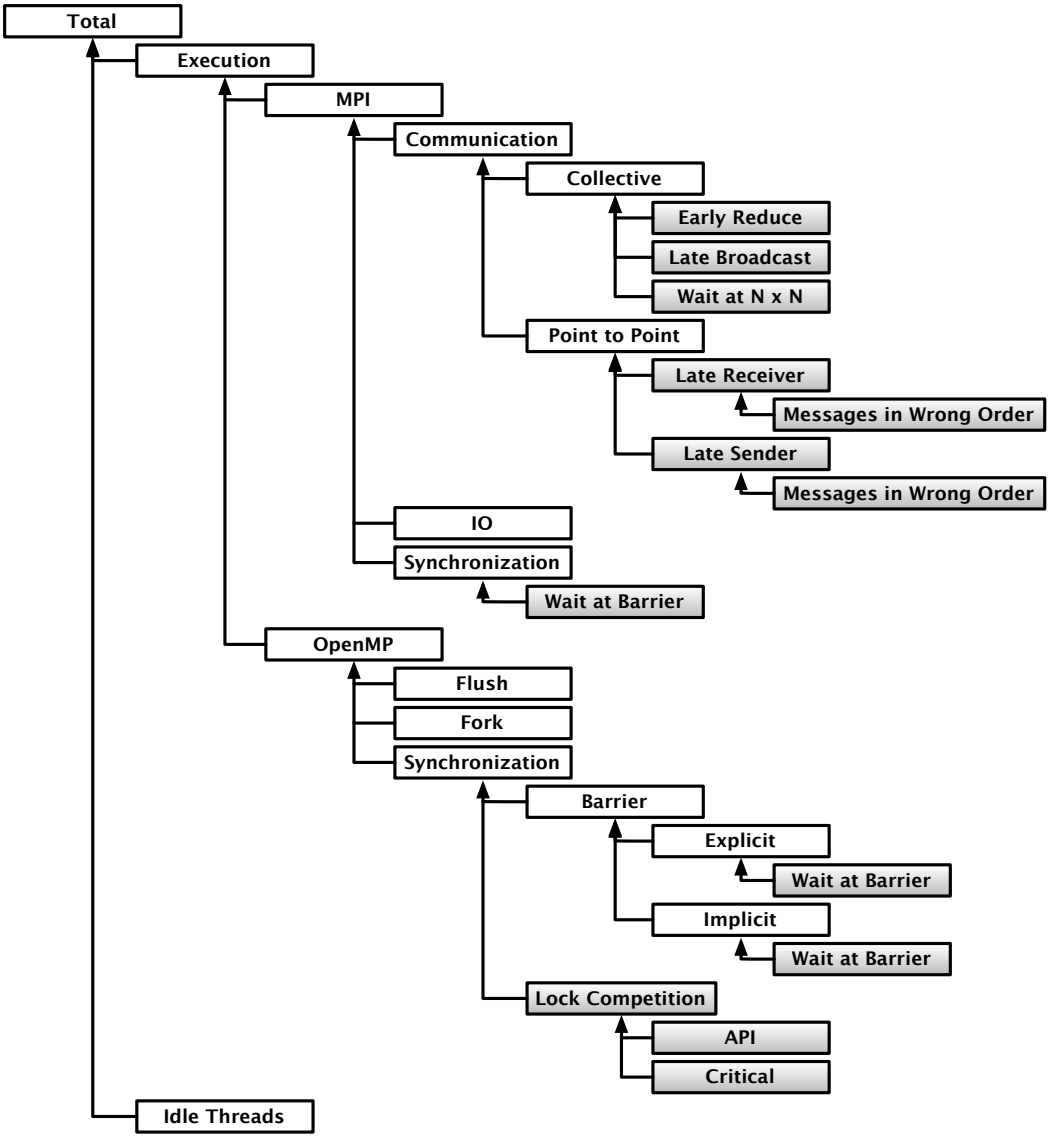


Figure 2.13.: Categorization of wait states defined by the EXPERT system. Figure adapted from [4], p.427.

a Java version of the previously mentioned ASL.

While all works mentioned above do post-mortem analysis there are also two major works of rule-based on-line analysis. The first one is the Performance Consultant Module, part of the Paradyn [60, 61]. Periscope [62, 63] is essentially a Paradyn clone, with the possibility of define custom rules using ASL.

Pattern Recognition

Other interesting methods to add intelligence to the analysis rely on the fact that parallel applications usually have a repetitive structure to detect patterns on application phases. For example, Casas et. al. [64] apply signal processing techniques to Paraver traces to detect coarse-grain iterations of the application algorithms. The resulting tool produces a summary of the application defined by the patterns found and also generates partial traces containing only the repetitive patterns, orders of magnitude smaller than the original, that can be used for later analysis at detail.

Freitag et. al. [65] present the *Dynamic Periodicity Detector* (DPD) an on-line analyser of OpenMP parallel function patterns used in the BSC Tools performance data extraction library, `Extrae`. It uses the stream of OpenMP parallel function identifiers to detect repetitive sequences on the subroutine calls, to generates smaller Paraver traces where the repetitive sequences had been compressed.

A third work project in pattern recognition can be found in [66]. In this case, the recognition is done at visualization time in a Vampir module. This approach tries to solve the problem of visualizing large amounts of data, showing the repetitive patterns on the trace as "boxes" on a time-line view. The pattern detection is based on *Compressed Complete Call Graphs* (cCCG) [67], an optimization of application call tree representation to save space.

Cluster Analysis

In this dissertation we make use of cluster analysis techniques. These techniques, whose target is to classify elements into groups, has also been exploited previously in the performance analysis area, but with different objectives.

Nickolayev et al. [10], based on [68], propose the application of K-means clustering algorithm in an on-line analysis to determine the similarities among processors involved in a parallel application execution. The authors use coarse-grain granularity metrics such as processor idle or running times to describe the behaviour of each individual processor. The work was developed as part of the Pablo Performance Environment, with the target of reducing the event traces generated. Instead of flushing the performance data of all processors, the output trace just includes the metrics of a representative processor per cluster detected, reducing the output data up to 4.5 times.

In [11], Ahn and Vetter develop a deep statistical analysis of event traces containing processor performance hardware counters that characterize application subroutines. In this work, hierarchical clustering and K-means clustering are used to determine the similarity across the processes, MPI tasks and OpenMP threads, at the level of subroutines. The authors demonstrate the utility of clustering to automatically distinguish master-slave patterns

of the processes and also application algorithm structural patterns such as the organization of the processes depending the problem decomposition. In addition, the authors use other multivariate statistical methods, Principal Components Analysis (PCA) [69] and Factor Analysis [70], to highlight the high correlations between some of the performance counters. These two techniques are useful to select those metrics that provide more information, reducing the dimensionality of the collected data.

The framework PerfExplorer [12], developed as a part of the TAU toolkit, offers similar features to [11] with major emphasis on describing the detected clusters. In PerfExplorer, K-means and hierarchical clustering, PCA and Factor Analysis are applied to profiling information stored in a PerfDMF database, that include a wide variety of metrics, from high level idle or running times to low level processor hardware counters. As in [11], the clustering algorithms are used to find the parallel processes, both MPI tasks and OpenMP threads, that behave similarly. The major contribution of this work is the correlation of the groups found with the profiling information available, that provides the analyst a clear understanding of the behaviour of the clusters. In addition, PerfExplorer implements a rich GUI to navigate through this information.

In [71], we find a totally different use of cluster. The goal of the work is also to reduce the information, in this case the instructions needed to an accurate micro-architectural simulation, but the characterization tries to find similar application phases. The execution phases are regions of 100 million instructions and the metrics use to characterize them are basic block vectors, unidimensional vectors that account how many times each of basic blocks of the application has been executed in the given region. The authors demonstrate both that K-means is able to correctly distinguish the application phases and that simulating a set of representatives, not just the centroid, provides high quality simulations.

3. Introduction to Cluster Analysis and Multiple Sequence Alignment

In this chapter we present a brief introduction to cluster analysis and multiple sequence alignment, required to fully understand the technical contributions of this thesis.

3.1. Cluster Analysis

Cluster analysis consists of assigning a set of objects into groups, the clusters. The objects assigned to a same cluster are similar in terms of a distance or dissimilarity metric. In this thesis, these objects are d-dimensional points where each dimension is a performance metric. The distance measure used is the Euclidean Distance.

This clustering task can be implemented in different ways obtaining a wide set of distinct clustering algorithms. Surveys by P. Berkhin [72] and by Xu and Wunsch [73] review a high number of these algorithm, but in this section we focus on algorithm families most commonly used.

3.1.1. Centroid-based clustering

In the centroid-based clustering algorithms family, the resulting clusters are represented by a central vector, that can be part of the data set (a *medoid*) or not (a *centroid*). The rest of the objects in the data set are assigned to the nearest cluster centre.

K-means [74] algorithm is the classic example of centroid-based clustering algorithm. Consists on dividing the whole set of data points in *k* clusters C_j. Each cluster is represented by the mean value (or weighted average) c_j, the *centroid*. The sum of discrepancies between a point and its centroid is used as objective function in iterative optimization schema. The usual distance to minimize is the Euclidean distance.

The algorithm requires the user to supply the desired number of clusters k, and the the optimization schema is composed by three simple steps, as can be seen in Figure 3.1:

1. Compute the centroids. In the first iteration, they can be computed randomly or by several other methods. In further iterations, the centroids are the mean value of the points assigned to each cluster.

2. Compute the distances of each point to all the centroids.

3. Group the points to the closest centroid, to minimize the discrepancies.

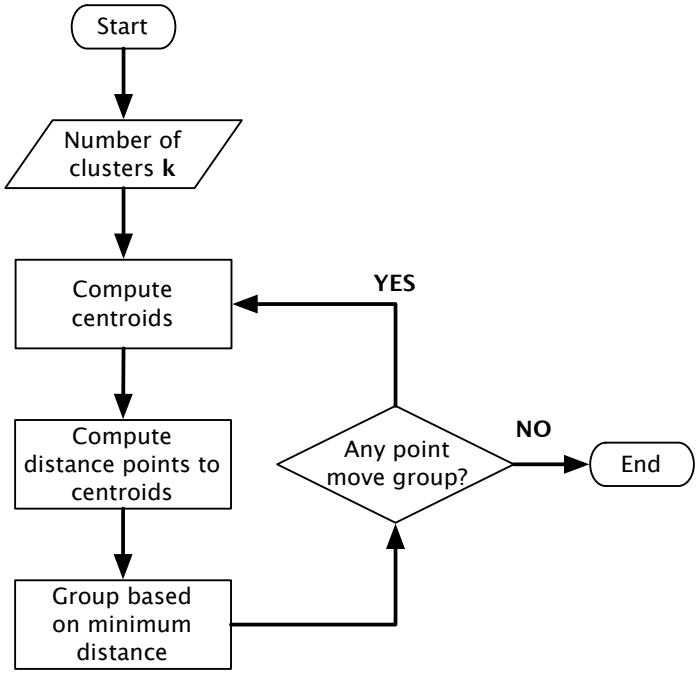


Figure 3.1.: K-means algorithm workflow

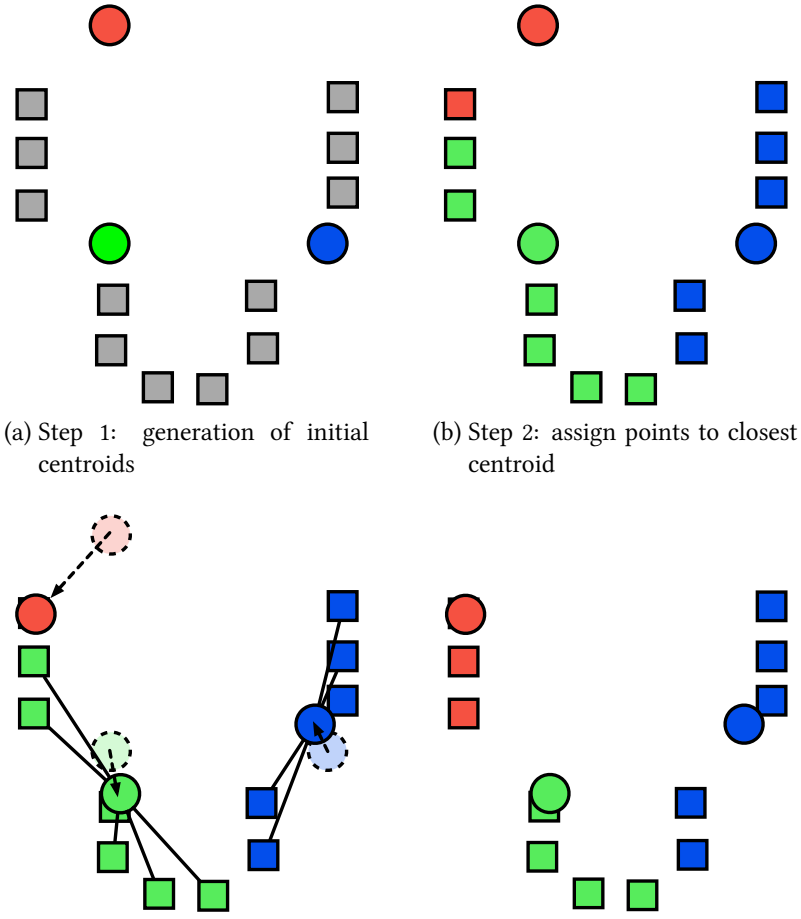
When each iteration finished, the algorithm checks if there has been any change in the points assigned to each cluster. If the clusters have not changed, the algorithm finishes.

Figure 3.2 illustrates this process. The first plot 3.2b is the election of initial centroids, the round shaped points. Plots 3.2b and 3.2c are the iterative nucleus of the algorithm, where points are assigned to their nearest centroid and then the centroids are updated. The algorithm converged when objects assigned to each cluster do not change in two consecutive iterations, plot 3.2d.

The selection of *k* is one of the major drawbacks of K-means/medoids algorithms as is not always possible to guess how many different clusters are present in the data set. The algorithm X-means [75] tackles this problem by applying the Bayesian Information Criteria (BIC) that evaluates the quality of clusters in terms of how well they represent Gaussian distributions. Basically, X-means iteratively executes K-means increasing the value of *k*, verifying on each iteration if the resulting clusters increase the BIC score.

The equivalent algorithm but using actual elements in the dataset, the *medoids*, is called K-medoids being Partition Around Medoids (PAM) [76] the reference implementation.

By far, centroid-based algorithms are the most widely used clustering algorithms. The major advantage of these algorithms is the easy implementation and also its fast execution. Unfortunately, these algorithms lack some important aspects: they are not robust against outliers and they the assume an hyper-spherical structure of the data distribution.



(a) Step 1: generation of initial centroids

(b) Step 2: assign points to closest centroid

(c) Step 3: recompute the centroids

(d) End: Convergence state

Figure 3.2.: Graphical example of K-means algorithm using $k = 3$. After generating the initial centroids in Step 1, points of the data set are assigned to the closest one on Step 2 and centroids are recomputed on Step 3. Step 2 and 3 are executed until the algorithm converges.

3.1.2. Connectivity based clustering

The target of connectivity based clustering [77], also called hierarchical clustering, is the construction of a hierarchy of individuals in a data set. This hierarchy is represented as a *dendrogram*, a tree where the leaves are the individuals and the root represents the whole data set. Intermediate nodes represent groups of two or more individuals whose height with respect to the leaves expresses the value of a linkage metric required to conform such a group. This linkage metric is based on a dissimilarity function and can be computed in different ways: single linkage uses the minimum value of the dissimilarity metric between two points/groups; or average linkage, the average of the distance; full linkage uses the maximum value of the dissimilarity metric.

Figure 3.3 contains an example of a small data set, plot 3.3a, and the resulting dendrogram obtained with a single linkage of the Euclidean distance, plot 3.3b. Points B and C, and D and E, have the same distance, so they merge at same height in the dendrogram. Next, the group of D and E merges with F at slightly higher height. The rest of the dendrogram express a merge of the group formed by B and C and the group formed D, E, F. Finally the whole data set is merged at top level.

The strategy to build the dendrogram results in two types of hierarchical clustering: aggregative, a bottom-up approach, merging the individuals/groups from leafs to root; or divisive, a top-down approach, separating the individuals/groups from the root to the leafs.

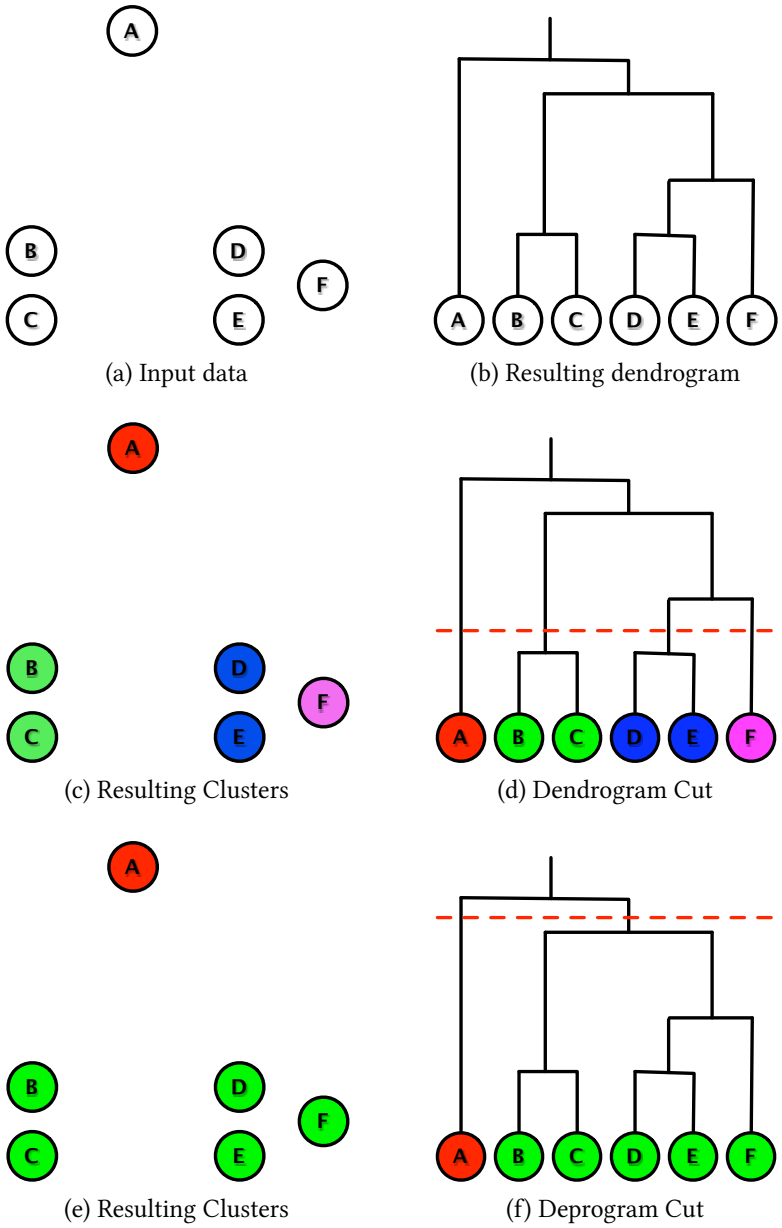
To obtain a data partition in a hierarchical clustering, it is required to perform a *horizontal cut* at some height in dendrogram. In Figure 3.3 there is an example of two different partitions: plot 3.3c shows a partition obtained by cutting the dendrogram at the height marked in 3.3d; analogously, plot 3.3e correspond to the clusters obtained by cutting at the heights marked 3.3f. We can see that a cut close to the leaves produce a big number of clusters more compact. On the other hand, a cut close to the root of the dendrogram produce less clusters with more variability.

As opposite to centroid-based clustering, where the algorithms generate Gaussian clusters around a centroid/medoid, connectivity based clustering does not assume the underlying model of the data. In other words, the construction hierarchy of the individuals according to the dissimilarity metric is orthogonal to how the individuals are distributed. On the other hand, deciding which level of the dendrogram is expressing the most valuable division of the data to perform the cut is a hard task. This problem worsens when dealing with large data sets due to difficulty to depict and analyse the dendrogram.

3.1.3. Density-based clustering

Density-based clustering algorithms are partition algorithms, as K-means, but they share some features with connectivity based clustering. The aim of density-based clustering is group points which are linked by a particular connectivity property and their density is big enough to be considered as a real cluster. These algorithms are widely used in the image recognition area.

DBSCAN [78] is a classic example of density-based algorithm. The inputs of DBSCAN are two parameters, the radius Epsilon (*Eps*) and minimum number of points (*MinPoints*). This



(a) Input data (b) Resulting dendrogram

(c) Resulting Clusters (d) Dendrogram Cut

(e) Resulting Clusters (f) Deprogram Cut

Figure 3.3.: Graphical example dendrogram construction used in hierarchical clustering and the possible partitions obtained from cuts a different heights

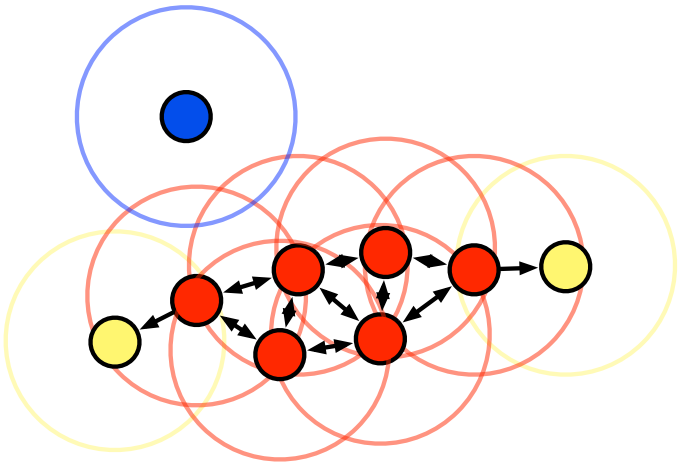


Figure 3.4.: Graphical example of a resulting cluster using DBSCAN density-based clustering. Yellow points are border points, red points are core points. Blue point is a noise point

algorithm is based in two basic definitions:

1. A point p is *directly density-reachable* from a point q if their distance is less or equal to Eps, and the Eps-neighbourhood of q (neighbours in a distance less or equal than Eps) is greater or equal to $MinPoints$. This relation is not symmetric.

2. Two points p and q are *density reachable* if there is a sequence $p_1, ..., p_n$, where $p = p_1$ and $q = p_n$ where each p_{i+1} is directly density-reachable from p_i.

3. Two points p and q are *density connected* if there is a point o such that both p and q are density reachable from o.

The resulting clusters obtained are those subsets C_i of the data where all points are mutually density connected. The points that do not fall in a cluster are considered noise. Note that the definitions lead to two types of points inside a cluster: *core* points, inner points in a cluster that serve to fulfil the *density connectivity* across all points in the cluster, and *border* points, in the edges of the cluster, from where there are no directly density-reachable points available. These definitions are clarified with Figure 3.4: red points are the core points of the cluster formed by red and yellow points; yellow points are border points; arrows represent the direct density reachability relation and coloured circumferences are the range of the Eps-neighbourhood; blue point is a noise point that do not belong to the cluster.

As hierarchical clustering, density-based clustering algorithms do not do any considerations about the data structure or model. In addition, there are robust against outliers and noise. An important disadvantage is the lack of interpretability of the resulting clusters.

3.2. Sequence Analysis

Sequence analysis is a set of processes and methodologies used in bioinformatics to understand features, structure or evolutionary elements. of DNA, RNA or protein chains. One of the most exploited and well-known techniques is the sequence alignment. The aim of the sequence analysis is to arrange two or more sequences in order to identify regions of similarity across them. Furthermore, (multiple) sequence alignments are also used for non-biological sequences, such as for those present in natural language or in financial data.

To exploit the multiple sequence alignment (MSA) in the parallel performance analysis scenario, we represent each task/thread involved in a parallel application as the sequence of different phases executed to solve a given problem. This parallelism has never been used before in the parallel performance analysis literature. We demonstrate in chapter 5 its utility in this scenario to determine the similarity of the work done across the different parallel processes or threads.

Here we briefly introduce three families in of MSA. To those interested readers, see the complete and up to date survey by Cedric Notredame [79].

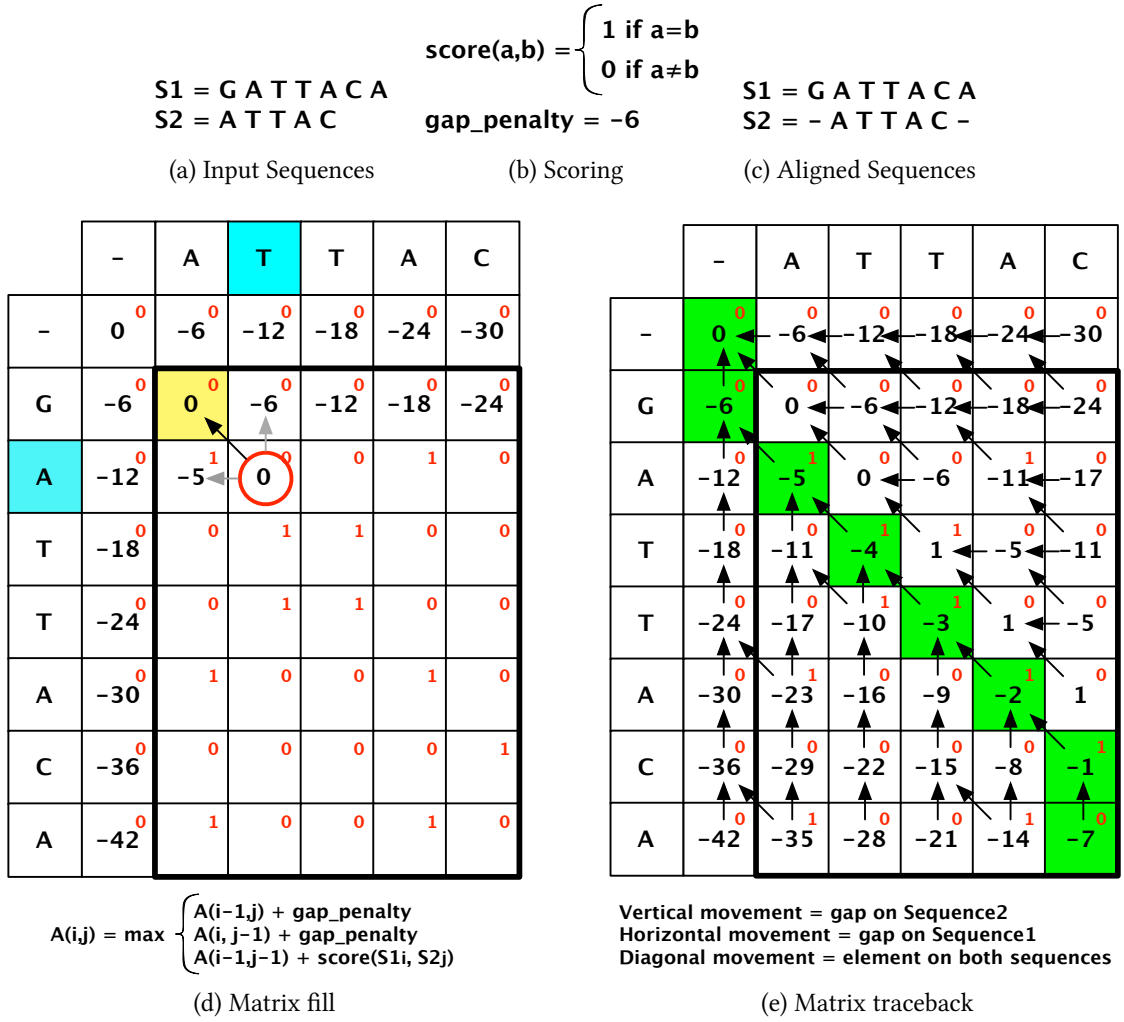
3.2.1. Dynamic programming

The dynamic programming methods are based on the classical Needleman-Wunsch [80] and Smith-Waterman [81] algorithms, originally designed to perform a pairwise alignment. Needleman-Wunsch algorithm is used to detect global alignments, when initial sequences are quite similar and have similar length. On the other hand, Smith-Waterman is used to detect local alignments, when it is known that the two sequences have certain regions in common. Both algorithms use *dynamic programming* [82] methodology to break the main alignment problem into simpler sub-problems, obtaining optimal solutions. The algorithmic scheme in these algorithms consists on building a m-by-n matrix A, where m and n are the lengths of each input sequence, and each position $a_{i,j}$ is computed according a match/mismatch score depending on the i and j elements of each sequence and also previous positions. Once the matrix is constructed, a trace-back will define the resulting alignment. The way to fill and trace-back the matrix what distinguish them. Figure 3.5 illustrates an example of the Needleman-Wunsch algorithm: using DNA sequences in 3.5a and the scoring defined in 3.5b, the algorithm produces the alignment 3.5c; in any case, the most interesting part of the process is the matrix fill 3.5d and the matrix trace-back 3.5e. The trace-back simply 'follows the sequence' of the matrix filling to obtain the alignment that produce the best score.

The translation of this basic algorithms to apply to multiple sequences consists of building an n-dimensional matrix, with *n* being the number of sequences. This generalization of the dynamic programming has been demonstrated to be an NP-complete problem, so these algorithms become impractical due to their computational cost.

3.2.2. Progressive methods

The alternative to avoid the limitations of dynamic programming is using heuristics that produce sub-optimal alignments, both global or local.



(a) Input Sequences (b) Scoring (c) Aligned Sequences

(d) Matrix fill (e) Matrix traceback

Figure 3.5.: Matrix construction and trace-back in the Needleman-Wunsch pairwise alignment algorithm

The progressive heuristic technique consists of computing the alignment of two of the N sequences, set this alignment as a fixed entity and then combine it with one of the $N-2$ remaining sequences, repeating the process until all the sequences are gathered into a single alignment.

In general, the solutions that implement the progressive method differ in the way to select the sequences, but is commonly accepted the principle of "the closer the earlier". To implement this principle, most of the algorithms build a guide tree based on the distances calculated from pairwise alignments.

The most referenced progressive MSA is CLUSTAL [5] and its different variants [83, 84, 85, 86], that include both global and local alignments. Figure 3.6 depicts the flux diagram of the original CLUSTAL algorithm that represents the general scheme of a progressive method. In this case, the progressive selection of sequences is based on a hierarchical cluster analysis, using an Unweighted Pair Group Method with Arithmetic Mean (UPGMA) tree as a dendrogram, where distance between pairs of clusters A and B is the average of all distances between pairs of sequences $x \in A$ and $y \in B$.

T-Coffee [87] is a plus novel approach, also a widely used in the bioinformatics community. Basically, it extends the generic CLUSTAL scheme depicted in Figure 3.6, adding an initial step where different MSA algorithms (including CLUSTAL) are used to generate an initial alignment structure, the *primary library*. The sequences and scores in the primary library are weighted and processed to generate an secondary structure, the *extended library*, the serves as the basis to build the guide tree and finally perform the progressive analysis. The combination of external algorithms leads T-Coffee to obtain excellent results, at the cost of being one of the most expensive MSAs.

In this category we also have to mention, Kalign [88], and its different variants [89, 90], that thanks to the use of a different methods to compute the scoring for the sequences pairs, it has become the fastest MSA algorithm.

3.2.3. Iterative methods

This set of methods is similar to progressive methods. In general, the progressive approach is also applied, but while progressive methods do not recompute the initial pairwise alignments nor the guide tree, iterative methods revisit some of the stages to reconstruct or adapt previous decisions. For example, the MUSCLE algorithm [91, 92] three fixed refinement steps. In the first two steps it varies the score distance to evaluate a pairwise alignments, using less accurate but fast distance measures. Next it generates a progressive alignment and finally in a (iterative) third step it a final alignment based on combinations of sub-tree prunes. Even using a more complex scheme than a progressive method, MUSCLE obtains similar results than CLUSTAL being faster and less memory consuming, but in any case, not being able to beat Kalign.

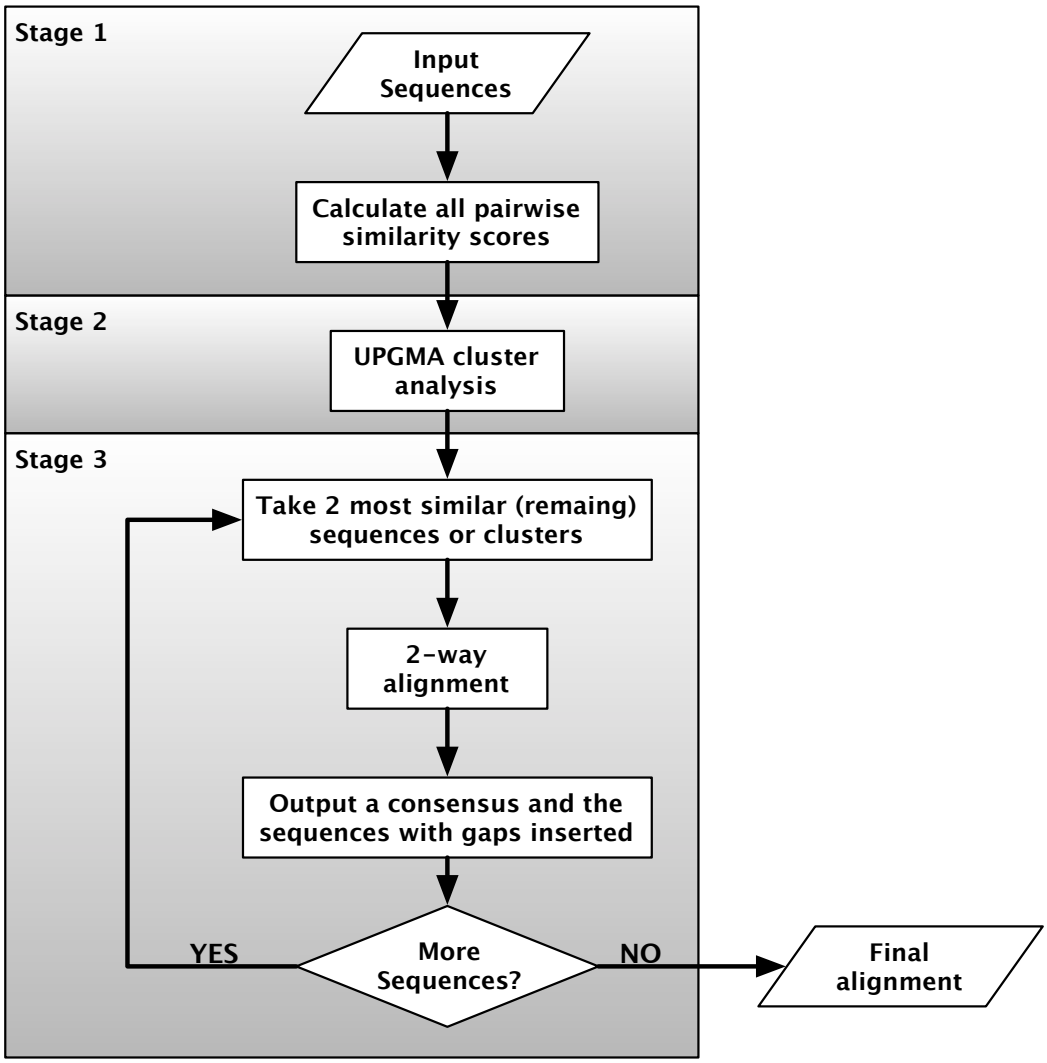


Figure 3.6.: Flux diagram of CLUSTAL multiple sequence alignment algorithm. It represents the foundations of a progressive multiple sequence alignment algorithm scheme. Figure adapted from [5], p.238.

Part II.

New Performance Analytics Techniques

4. Computation Structure Detection using Cluster Analysis

The first of the contributions contained in this thesis is a technique to determine the computation structure of parallel applications applying cluster analysis.

In this chapter we detail all the elements necessary to correctly characterize the computation regions of a parallel applications. We also justify the suitability of the DBSCAN clustering algorithm among others for this purpose.

4.1. Computation bursts and cluster analysis

In order to characterize parallel applications structure, the computation bursts are the minimum analysis abstraction used in our analyses. We define a computation burst, also named *CPU burst* or simply *burst*, as the sequential region of the application between communications primitives or calls to a given parallel runtime. This definition is based on the dichotomy that a parallel application can only be performing a parallel primitive or processing data, i.e. *computing*. Figure 4.1 depicts a simple time-line of two processes of a message-passing application, that perform some computation and the interchange information, the communication phases.

Cluster analysis is used to group the different behavioural trends CPU bursts exhibit along the application execution. To do that we associate to each burst a feature vector of different performance data. In all the studies presented in this thesis, the processor performance hardware counters represent the most interesting piece of information when analysing the CPU bursts behaviour as they provide an unique insight of the CPU performance at very fine grain.

It is interesting to emphasize that while we use the cluster analysis to identify the different trends of the computation bursts, following the idea presented by Sherwood et. al. in [71], previous works that used cluster analysis in the parallel performance analysis scenario followed a different approach. All of them [10, 11, 12], focused on identifying those processes/tasks/threads that behave similarly.

4.2. Data Preparation

Before executing the clustering algorithm, the input data set is manipulated in different ways to reduce the volume processed by the cluster analysis and also increase the quality of the results.

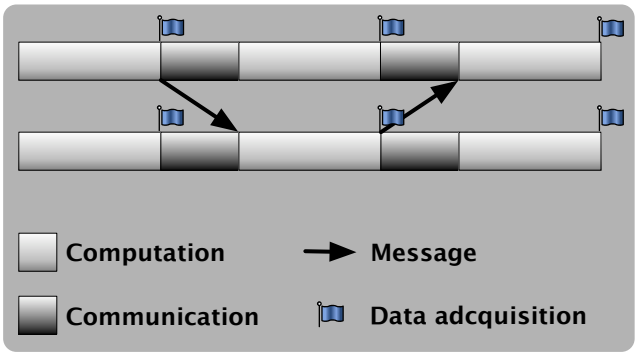


Figure 4.1.: Computation regions and communication regions in a message-passing parallel application

First, we apply two pre-processing techniques to the data. Then we carefully select the dimensions which will be used by the clustering algorithm to group the points.

4.2.1. Pre-processing

The first pre-process of the data we apply is a filter. The filtering stage simply consists of discarding those bursts whose duration is negligible in the application execution. In this way, a vast amount of irrelevant data is directly discarded when applying the clustering algorithm. As can be seen in Table 4.1, the filtering process is been able to discard up to 80% of processed bursts maintaining the 99% of application time. Additionally, to fine tune the filtering, we can define range filters to the metrics used by the cluster algorithm.

In second place, a normalization is applied to ensure that when using counters with different data ranges, none of them bias the clustering results. In fact, two different normalizations methods are used. First, logarithmic normalization is used when the dynamic range of the performance metric is large. Reducing the dynamic range guarantees that the results of clustering are not displaced to the higher values of a counter. Additionally, range normalization, simply scaling the data to $[0, 1]$ range ($\forall a_i \in A, a_i \leftarrow (a_i - min(A))/(max(A) - min(A))$), ensures that all factors have a similar weight in the multi-dimensional clustering.

4.2.2. Dimensionality Reduction

A common problem when using clustering algorithms is related to the dimensionality of the data. With the performance counters data we have up to 8 different counters for each CPU burst. Our proposal to address this problem is to reduce the dimensionality by selecting counters or derived metrics with "physical" meaning to the analyst, as those proposed by Joshi et. al. in [93]. In the studies developed, we found the following groups of counters represent a useful way to determine the application structure:

- Completed Instructions combined with Instructions per Cycle (IPC). This combination focuses the clustering on the "performance view" of the application.

- Completed Instructions, Level 1 (L1) and Level 2 (L2) cache misses. This combination reflects the impact of the architecture on the application structure, via the cache misses counters.

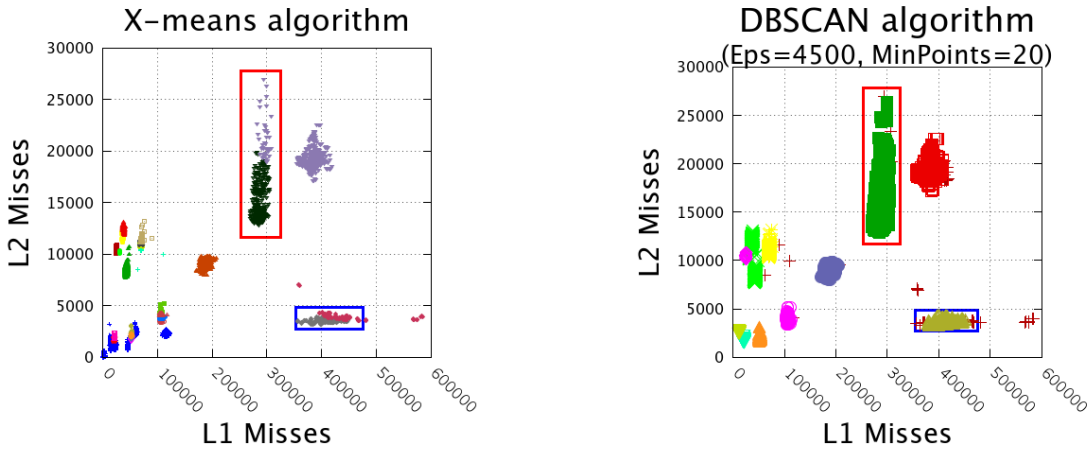
These two combinations of counters result in two characterizations of the computation bursts where we take into account the computational complexity and the performance observed. In both cases, Completed Instructions is used to represent the computational complexity, while the performance observed by these regions is represented by the IPC, in first combination, or the memory accesses behaviour (in terms of cache misses) in the second combination.

In previous works such as the one by Ahn et. al. in [11] or the presented by Huck et. al. in [12], Principal Components Analysis (PCA) is used to reduce the dimensionality of the data by creating a new space with a lower number of dimensions, that are the principal components of the original data. Each point is projected to this new space, and the clustering algorithm is applied to the transformed set of points. This technique succeeds in reducing the dimensionality, as an aid to the clustering algorithm. In our experiments, we tested this technique obtaining similar results to our manual selection of attributes. To the contrary, the resulting dimensions have no direct interpretation (they are combinations of the original ones). As we consider the scatter plot with the cluster information is essential to validate the results (as is explained in 4.6), we conclude that using PCA is not strictly necessary in this scenario because it adds no significant benefits to the process.

4.3. Clustering algorithm selection

The initial cluster algorithm used was a K-means-like algorithm. As we explained in chapter 3, K-means-like algorithms always suppose a Gaussian model of the data. However, the performance hardware counters data is not distributed following this Gaussian model so the results obtained were not satisfactory. Finally, we selected DBSCAN as representative of density-based clustering as the due to its no assumption of underlying model. Even hierarchical clustering also shares this property with density-based clustering, we discarded it due to the difficulty to manipulate dendrograms with high number of points.

To demonstrate the aforementioned inability of K-means-like algorithms to correctly detect clusters which are non Gaussian, in Figure 4.2 we show the result of applying X-means and DBSCAN to the same application data, a section of `CPMD` [94], a molecular-dynamics application that uses the Car-Parrinello method, executed with 128 tasks, and using L1 vs. L2 cache misses parameters. The scatter plots show some clouds of points that have a spherical shape, but others can be elliptical with different principal component directions. In this case, X-means tends to partition the ellipses. The red box (on the central area) marks a region with a strong vertical component where X-means, 4.2a, detected two clusters, but DBSCAN, 4.2b, detected only one cluster. The blue box (on the lower-right area) shows another equivalent region where X-means also detected two clusters and DBSCAN only one, in this case having a strong horizontal component. In terms of the structure detection, the homogeneous shape detection done by DBSCAN is better than the X-means cluster assignment.



(a) X-Means Resulting Clusters

(b) DBSCAN Resulting Clusters

Figure 4.2.: X-means and DBSCAN algorithms comparison. Boxes highlight clouds of points with strong components, vertical and horizontal, where X-means has divided the isolated group.

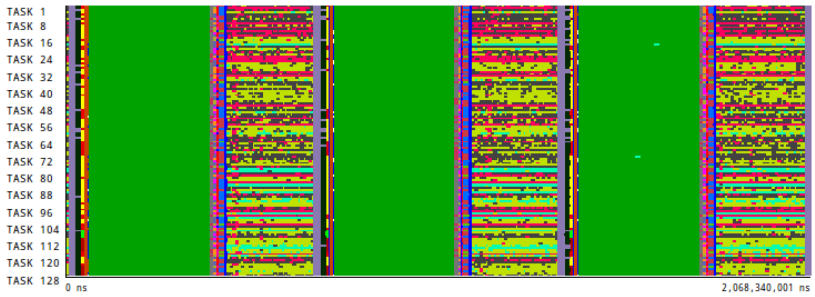
To demonstrate this affirmation we have to take into account that almost all message-passing parallel applications follow a *Single Program, Multiple Data* (SPMD) model. In this model, the tasks involved to solve a problem execute the same program using different parts of the total data to process. According to this model, the structure we want to observe in a time-line visualization corresponds to succession of phases, longer or shorter, where the bursts of all tasks are assigned to the same cluster, i.e. the different phases of the same program.

Figure 4.3 contains the clusters time-line obtained by each algorithm. In this Figure, the X axis of the time-lines is the time axis, the Y axis represents the tasks involved in the parallel application execution, and the colour indicates the cluster identifier assigned to each CPU burst. Both time-lines of the figure contain three iterations of the application.

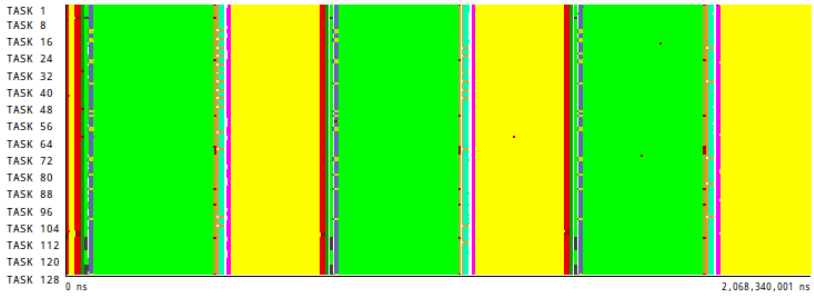
In the time-line obtained using X-Means clusters, Figure 4.3a, we can see a pattern that repeats three times, the three iterations share the same structure. On each iteration we see an initial SPMD phase dark green clearly detected while the second part of each iteration does not present the regular SPMD structure we expected. On the other hand, the time-line containing the clusters detected by DBSCAN, Figure 4.3b, does not show this irregular pattern, and each iteration is detected cleanly as a repetition of two big clusters, light green and yellow.

4.4. DBSCAN parameters

DBSCAN algorithm is very sensitive to its parameters, specially to Eps, the radius of the search. Figure 4.4 and Figure 4.5 illustrate how changing the Eps parameter affects to the granularity of structure detected. In this case, the figures show two different analysis using



(a) X-Means clusters time-line distribution



(b) DBSCAN clusters time-line distribution

Figure 4.3.: X-means and DBSCAN algorithms comparison. Clusters time-line distribution

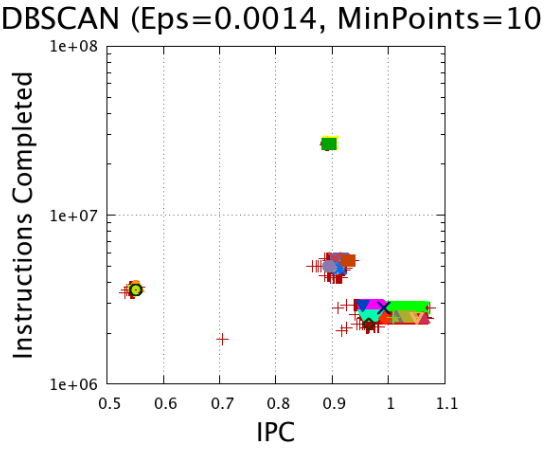
the same input data, obtained from the NPB BT class A benchmark, class A, a block tri-diagonal solver included in the well-known NAS Parallel Benchmarks [95], executed using 64 MPI tasks.

Figure 4.4 presents a cluster analysis using a *restrictive* (small) value of Eps (0.0014) and $MinPoints$ of 10. An small Eps value means that the radius of search of the algorithm is shorter, and indeed restrictive, so the results will a big number of clusters, more compact and dense, and more noise points, as can be seen in 4.4a. In terms of the structure detected, the computation regions each cluster represents will be highly detailed, showing internal behaviours that not always reflect clear stages in the application execution. As an example, in the time-line 4.4b we can observe how the application structure is detected in fine detail: in this case a staggered computation pattern (similar to a pipeline) between different tasks.

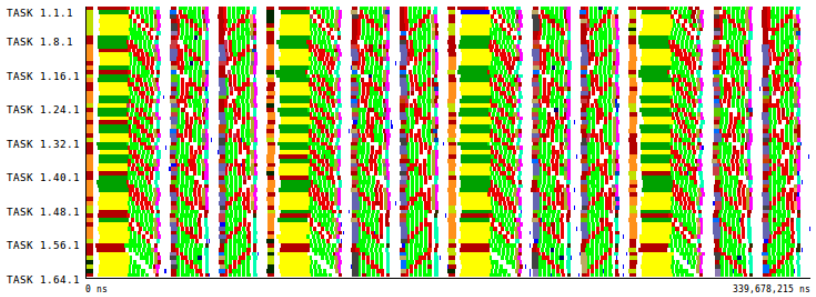
On the other hand, the cluster analysis of Figure 4.5 presents a different approach. Using a higher Eps value, 0.0400, and the same value for $MinPoints$ we obtain a small number of clusters, that aggregate more number of points each, 4.5a. In this case, the time-line 4.5b shows the detected application structure at a coarser granularity, and the typical SPMD structure appears.

Parameter selection

The technique we apply for the parameter selection is also described by Ester et. al. in [78], the same paper where DBSCAN is introduced. It consists of generating a histogram with the sorted k-neighbour distance, being k the desired value of $MinPoints$. Then this distance is

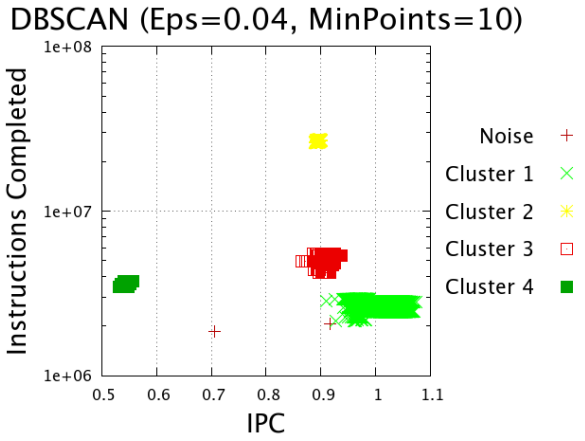


(a) Discovered clusters using a restrictive *Eps*

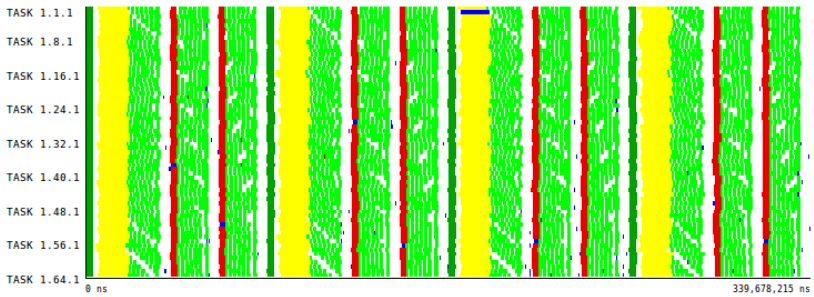


(b) Distribution of clusters in the time-line

Figure 4.4.: Cluster analysis of NPB BT class A Benchmark using clustering algorithm over Completed Instructions and IPC. Due to the use of a restrictive Eps value the structure is detected at fine grain



(a) Discovered clusters using a less restrictive *Eps*



(b) Distribution of clusters in the time-line

Figure 4.5.: A second cluster analysis of NPB BT class A Benchmark using the cluster algorithm over Completed Instructions and IPC. Using higher value of Eps produces a coarser grain detection, showing a SPMD structure

sorted (descending) and plotted. The histogram will show a descending curve. In [78], the authors suggest that the optimum value of Eps is the distance where the curve makes its first inflexion (or "valley"). The points located on the left of this "valley" will be noise in the resulting partition and the rest will be present on one cluster. In [78], the authors ensure that choosing 4 as the default value of $MinPoints$ produces the best results in 2-dimensional clusterings. In our experiments higher values, usually 10, obtained a better characterization of applications structure.

Application	Duration Filter (μs)	Discarded bursts (%)	Discarded time (%)	Analysed bursts	Analysis time (s)	Clusters found[1]
CPMD	1000	34.03	7.28	53,454	3067.790	16 (2)
NPB BT	500	64.26	1.10	24,850	746.160	9 (3)
WRF	1000	88.70	0.44	13,917	30.237	23 (4)
Hydro	5000	61.79	5.96	3,438	1.830	13 (3)

[1] In parenthesis, the number of clusters that cover more than 10% of application time

Table 4.1.: Clustering tool statistics for a set of applications used in the examples. The values regarding the NPB BT benchmark correspond to the execution with higher value of Eps

4.5. Cluster analysis results

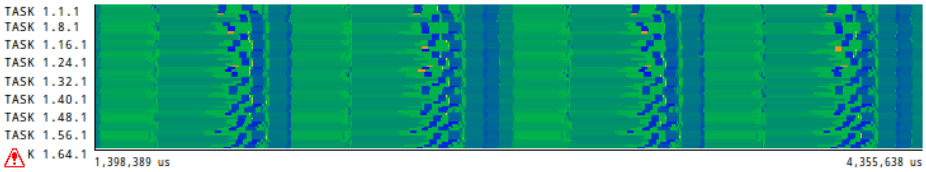
Up to this point, we have seen the ability of cluster analysis to detect the structure of message-passing parallel applications. In this section we highlight the usefulness of this technique to ease the characterization of a message-passing parallel application and the ability to detect the behaviour structure in contrast to the classic syntactic structure proposed by other techniques such as profiling.

In Table 4.1 we present some statistics about cluster analysis filters, execution times and detected clusters of the different applications used along this chapter.

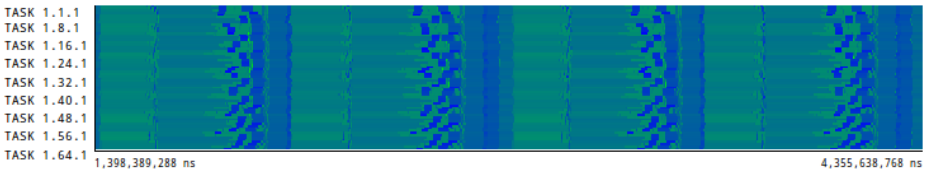
4.5.1. Ease of the computation structure analysis

When analysing a parallel application, the analyst/developer has to deal with lots of different information. For example, in Figure 4.6 we present the time-lines of four performance obtained for WRF application[1] executed with 64 tasks. The metrics are Million of Instructions per Second (MIPS), time-line 4.6a, Instructions per Cycle (IPC), time-line 4.6b and Level 1 (L1) and Level 2 (L2) data cache misses per 10^3 instructions, time-lines 4.6c and 4.6d respectively. On each of these time-lines, the X axis is the time, the Y axis are the application tasks

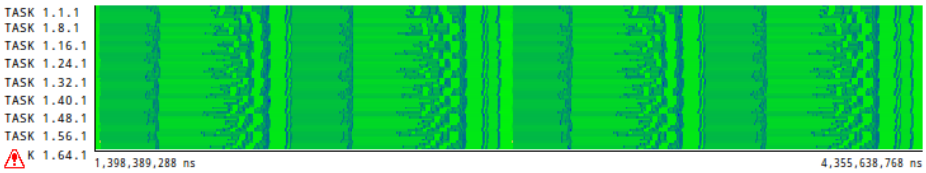
[1] A description of this application is available in chapter 9



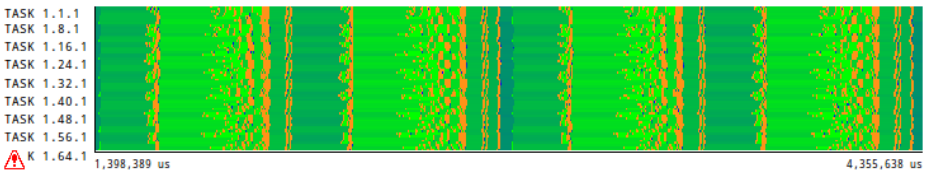
(a) Millions of instructions per second (MIPS)



(b) Instructions per cycle (IPC)



(c) Level 1 Data Cache misses per 10^3 instructions



(d) Level 2 data cache misses per 10^3 instructions

Figure 4.6.: Time-lines of different performance hardware counter metrics of `WRF NMM` application executed with 64 tasks

and the colour is a gradient from green to blue expressing the magnitude of the given metric (we do not indicate the ranges of each metric), being the orange zones the peak values.

Observing the time-lines, the analyst could certainly detect a structural pattern: a series of two wide regular phases followed by and irregular region of small and imbalanced phases (in dark blue on 4.6a and 4.6b and orange in 4.6d). This structural pattern repeats four times, so the analyst could conclude that he or she is observing four iterations of the main loop typically observed in the message-passing applications.

Using the cluster analysis we can draw different conclusions. In this example we applied DBSCAN to group the CPU bursts characterized with Completed Instructions, L1 and L2 data cache misses. Figure 4.7 contains the results of this analysis. The scatter plot 4.7a represent the clusters discovered projected to Completed Instructions and L1 data cache metrics. In this plot we can observe three isolated clusters below 1×10^8 and a region on the top right with three more clusters that present high number of instructions, but also more L1 data cache misses. It is interesting to note the Cluster 2 and 3 seem to be overlapped, so the algorithm

was able to distinguish them using the L2 data cache metric, not depicted on the plot.

However, the most interesting results of the cluster analysis can be observed in the clusters time-line 4.7b. What we wrongly consider four iterations of the application main loop, thanks to the cluster analysis is clearly depicted as two repetitions of two different structures. The first one starts with Cluster 3 (red), followed by Cluster 1 (light green) and finished with the succession of Clusters 6 (purple), 5 (pink) and 4 (dark green). The second structure starts with Cluster 2 (yellow), share the pattern of the previous one, Clusters 1, 6, 5 and 4, finishing with Clusters 8 (orange) and 7 (light brown). The application developers confirm that the simulation we run computed the status of one variable of the model just every other iteration, so the structure detected by the cluster algorithm was correct.

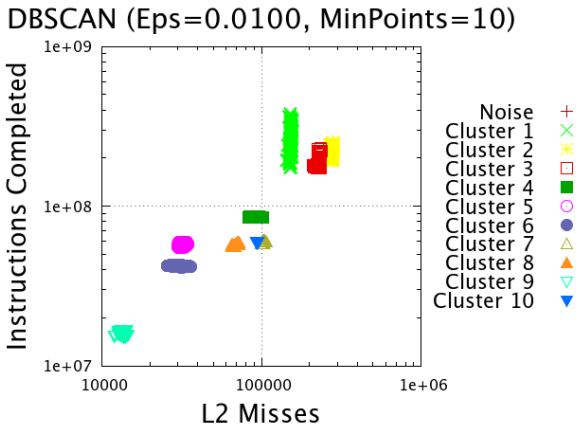
Another interesting information we can extract from the clusters time-line are the regions marked with a black box. In this region Cluster 10 (blue) was assigned to a subset of tasks, meanwhile the majority of tasks at this point of time were assigned to Cluster 7 (light brown). This situation indicates that we found an imbalance in the behaviour of this phase. According to the scatter plot 4.7a, this distinction in two different clusters was caused by the higher number of L2 data cache misses occurred in Cluster 10.

In terms of analysis, it is interesting to remark that usually few clusters represent the most time-consuming phases of the applications. As can be seen in Table 4.1, in our examples the number of clusters that cover more than 10% of applications time moves between 2 and 4 (number in parenthesis in the "Clusters found" column). This is a useful property of the structure detection proposed, as the analyst could focus on the analysis of these regions because their performance will have a bigger impact in the global application performance.

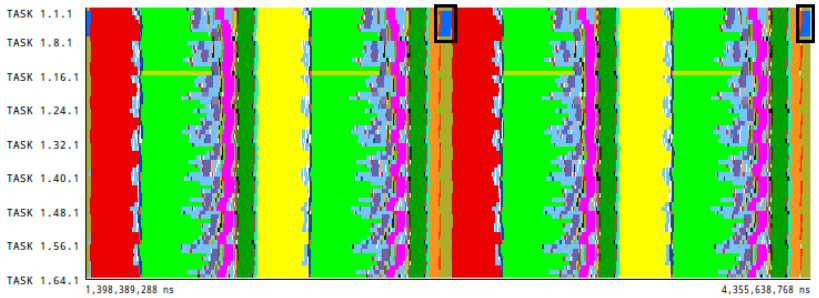
Cluster	Total computing time (%)	Average burst duration (ms)	IPC	MIPS	L2 misses per 10^3 instructions	Memory bandwidth (MB/s)
1	36.23	219.56	0.536	1214.60	0.570	88.67
2	15.95	190.32	0.505	1145.04	1.262	184.96
3	13.83	165.07	0.487	1190.18	1.170	178.23
4	10.19	60.81	0.619	1402.93	1.076	193.30

Table 4.2.: Performance characterization of the 6 clusters that aggregate more than 90% of the WRF application computing time of application analysis depicted in Figure 4.7

In the case of WRF, four clusters represent more than 10% of the total computation time. So as to analyse the regions they represent, we computed a set of statistics for these clusters using the hardware counters information available in the input data. This information is contained in Table 4.2. Using this table the analyst can observe the performance of each of the regions detected, and take decisions regarding which parts of the application should be improved in first term. For example, we can see that Clusters 2 and 3 (yellow and red regions in time-line 4.7b), represent a major section of application computation time, 15.95% and 13.83% respectively. In terms of performance, they have the lowest IPC of all regions



(a) Discovered clusters projected to L1 Misses and Instructions Completed



(b) Time-line distribution of the discovered clusters

Figure 4.7.: Cluster analysis with DBSCAN algorithm using Complete Instructions, L1 Data Cache Misses and L2 Data Cache Misses of WRF application executed executed with 64 MPI tasks

detected, 0.505 and 0.487. This small IPC figures could be caused by poor cache usage, as they exhibit the higher L2 data cache misses per 10^3 instructions and also the higher memory bandwidth. These observations present an useful starting point to the analyst/developer to study what is causing this potential memory problems and improve the regions detected, whose location in the actual source code is listed in Table 4.3.

Cluster	Code Section
1	`solve_nmm.f:[2037 - 2310]`
2	`solve_nmm.f:[1478 - 1782]` `solve_nmm.f:[2030 - 1782]`
3	`solve_nmm.f:[1241 - 1345]`
4	`solve_nmm.f:[2771 - 2865]` `solve_nmm.f:[2388 - 2489]`

Table 4.3.: Code linking of the 6 main clusters of WRF application

4.5.2. Applications syntactic structure and behaviour structure

We define the application syntactic structure as the abstract structure of the application's source done during the application design and implementation. On the other hand, we define the behaviour structure as the performance an application exhibited in an actual execution. In this section, we study the relationship between both structures, comparing the subroutines an application execute to the computation structure obtained by using cluster analysis.

The aim of this study is to demonstrate how this relationship is not always a bijection. In the following examples we show how a given computation phase can represent a type of computation that appears in different routines of the application, and also how a given subroutine could be detected as different computation phases depending on its parameters.

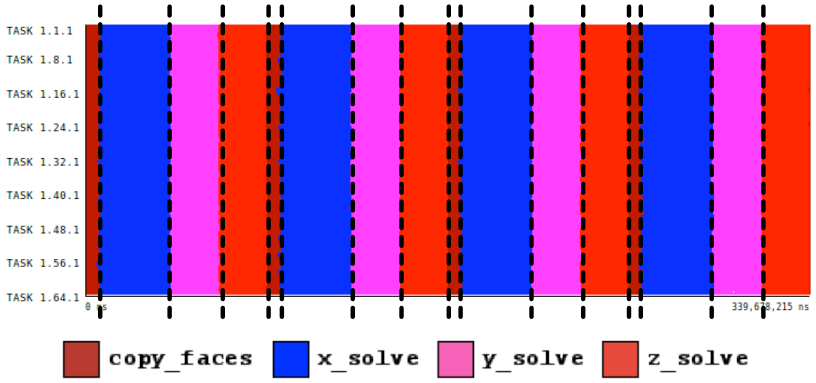
The two examples reflect how the structure detection based on cluster analysis provides an unique point of view of the applications behaviour structure. This technique surpasses other approaches such as the classical profiles, where all of these variability we are able to capture is hidden as they aggregate the performance information at the level of subroutines.

Subroutines with common behaviour

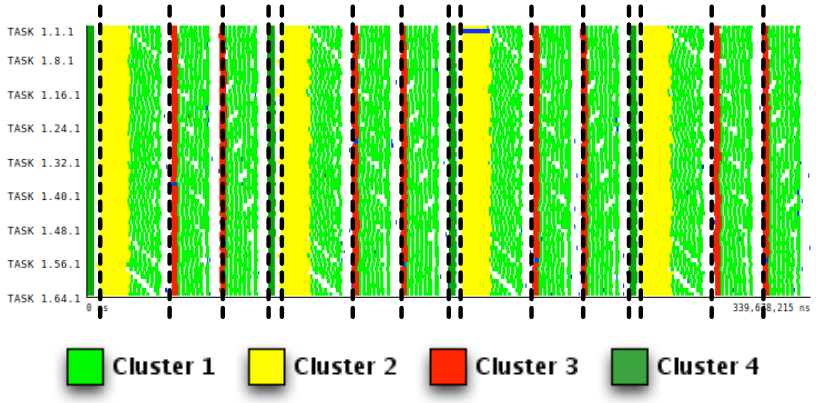
In this first example we used the cluster analysis of the NPB BT benchmark presented in Figure 4.5. We used this structure detection to illustrate a good parameter selection of DBSCAN algorithm, showing a clear SPMD structure.

Now, on Figure 4.8 we compare the main subroutines of this application, time-line 4.8a with the SPMD phases detected, time-line 4.8b. In both time-lines we added vertical lines to indicate the subroutines bounds.

Observing both time-lines, we can establish the following relationships: `copy_faces` subroutine corresponds to unequivocally to Cluster 4 (red); similarly Cluster 2 (yellow) always



(a) Time-line of main user functions



(b) Time-line of discovered clusters using a high *Eps* value

Figure 4.8.: Comparison of the clusters discovered on NPB BT class A Benchmark presented in Figure 4.5 and the main user functions of the application

corresponds to the initial part of the sub-routine `x_solve`; Cluster 3 (dark green) is the initial part of `y_solve` and `z_solve`; and, finally Cluster 1 (light green) is a common section of all solve subroutines.

The relationships listed demonstrate that although the syntactical structure of the application distinguished three different solvers, the actual behaviour of them its quite similar: Cluster 2 and Cluster 1 phases in case of `x_solve`, and Cluster 3 and Cluster 1 phases in case of `y_solve` and `z_solve`.

If we compare these observations to a regular profile output what we will obtain a is just the characterization of the three solve routines, independently. A further comparison could led to affirm that `y_solve` and `z_solve` behave similarly, but in no case would be able to affirm that all three have a big region where they behave similarly, unless we go through the source code.

Multi-modal subroutines

For this second example, we used Hydro [96] a solver of the compressible Euler equations derived from the cosmology hydrodynamics code RAMSES [97]. We performed an small execution using 24 tasks.

In Table 4.4 we present the correspondence between application subroutines and detected clusters. As we can see, Clusters 1 to 9 correspond unequivocally to fixed subroutines in the application, that could suggest a bijection between syntactic structure and an the observed behaviour. But the interesting point appears with subroutine `updateConservativeVars`. This subroutine has been detected as two different clusters, 10 and 11.

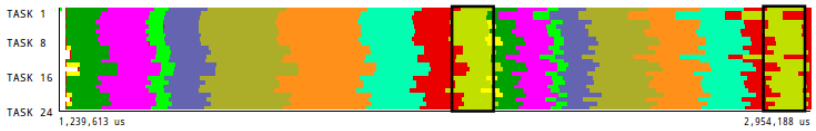
Cluster	Subroutine
1	`vtkfile`
2	`trace`
3	`qleftright`
4	`constoprim`
5	`riemann`
6	`slope`
7	`hydro_godunov`
8	`gatherConservativeVars`
9	`equation_of_state`
10	`updateConservativeVars`
11	`updateConservativeVars`

Table 4.4.: HydroC clusters/subroutines correspondence

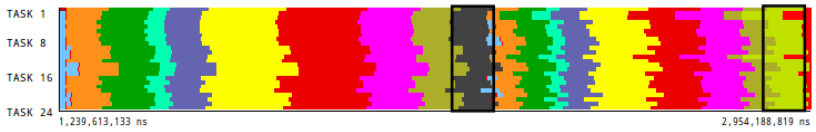
This detection could be understood as an imbalance of some of the tasks as the observed in the previous example of WRF. We can check if this reasoning is correct by observing the time-lines of Figure 4.9. The time-lines present two inner iterations of the Hydro solver. In the subroutines time-line 4.9a we marked the occurrences of `updateConservativeVars` (soft green yellow). In the clusters time-line 4.9b we marked the same regions occupied by this subroutine. What we see is that in the first iteration, this subroutine has been detected as Cluster 11 (dark grey) while in the second iteration it has been detected as Cluster 10 (soft green yellow). According to scatter plot detail presented in 4.10, this two clusters differ in around 1.4 million instructions.

Actually, this distinction reflects the actual behaviour of the solver: it works with a 2D domain which is treated in vertical or horizontal directions on successive iterations. Most of the inner subroutines present the same behaviour indifferently to direction selected, but the update of the conservative variables of the equations are sensitive to this change.

To sum up, the table and time-lines presented demonstrate that the behaviour structure we produce by using the cluster analysis is capable of detect multi-modal subroutines, that correspond to a fixed syntactical structure. Comparing again with regular profiles, this situation



(a) Time-line of main user functions



(b) Time-line of discovered clusters

Figure 4.9.: Comparison of the clusters discovered on HydroC and the main user functions of the application

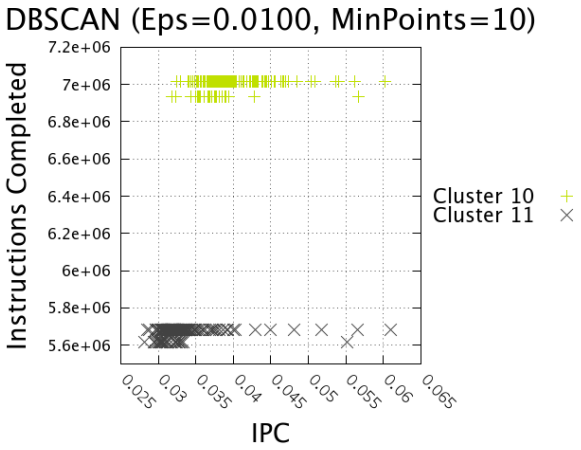


Figure 4.10.: Detailed plot of the clusters that represent the bi-modal `updateConservativeVars` of HydroC solver

could have never been detected.

4.6. Clusters quality evaluation

In previous studies in this area, the output of the clustering algorithm applied is assumed to be correct, and no other analysis of the clusters is done. The quality of the clustering is assessed informally by checking that it displays some of the expected differences between processes. An exception is found in [71]. where the authors use the Bayesian Information Criterion (BIC), as is defined in the X-means algorithm [75], to evaluate the compactness of the clusters detected, using a measure which is closely related to the assumption of a Gaussian model of the clusters.

Knowing that the density based cluster does not assume any model in the resulting clusters, the quantitative evaluation of the results is a difficult task. Our first approach consists of use the different to manually check the desired SPMD described previously, the so called “expert

criterion" [98] (also known as *gold standard* [99]). With the methodology provided we can make use of three complementary methods to perform the "expert validation":

1. Using the scatter plots to examine the cluster assigned to each point as a result of a good clustering, the scatter plot would show that isolated groups of points are detected as different clusters.

2. If the application is purely SPMD, we would expect the application time-lines to show that all processes execute the same cluster at the same time. In addition, we expect to find a repetitive pattern among clusters in the time-line, due to the common iterative structure of parallel codes.

3. If the original trace includes code linkage (user added events or back-traced caller events on MPI calls), a good clustering should show that every cluster region corresponds to a fixed sections of code. Have we have seen in the examples, these sections of code could not be strictly subroutines but inner parts of them or subroutines with multiple behaviours.

5. Evaluation of the computation structure quality

We can make an analogy between the sequences of different actions a parallel Single Program Multiple Data (SPMD) application performs and biological sequences such as DNA or proteins. Then, using a Multiple Sequence Alignment algorithm we can quantitatively measure how the application follows the SPMD pattern. We call this the *SPMDiness*. In this chapter we introduce the Cluster Sequence Score, an score that evaluates the SPMDiness of a computation structure characterization obtained using a cluster analysis. This score is also applicable to evaluate the SPMDiness of any other structural characterization.

5.1. Cluster Sequence Score Motivation

As explained in the previous chapter, the fact that density-based cluster does not assume any model of the data, made us consider the "expert criterion" as a first approach to evaluate the quality of a structure detection using DBSCAN. In fact, there are some validity indexes to assess the quality of density-based cluster algorithm results, as the ones presented in [100] or [101] but, as can be seen in the experiments presented later in this chapter, those are not able to correctly evaluate what we consider a good structure detection. Independently of the cluster algorithm used, or even the methodology to differentiate the computation phases, the results should group those computation bursts that detect conform SPMD phases at finer granularity. We call this a check for the *SPMDiness* of the resulting clusters. It is obvious to say that this reasoning only makes sense when the application to analyse follows a SPMD design, but in practice the vast majority of message-passing parallel applications follow it.

To check the *SPMDiness* the most useful output we consider is the application time-line, where the clusters information is used to colour the CPU bursts. In this time-line one should observe a succession of vertical regions assigned to the same cluster. In addition, one should also check that the pattern of phases repeats in case the application performs different iterations. The score we propose, Cluster Sequence Score, is a translation of this expert validation into an automatic process. Our main consideration in implementing this *criterion* is the following: if an application has an SPMD design, the sequence of actions in all tasks should be the same. Consequently, the sequence of clusters detected should also be the same for all tasks.

5.2. Multiple Sequence Alignment (MSA)

The reasoning regarding the similarity of the clusters in the sequence of each task/thread directly points to a clear analogy of our *criterion* and the DNA and protein alignments, a highly developed area in bioinformatics. The parallelism with our data is simple: each task/thread is represented as a DNA sequence or a protein where DNA bases or amino-acids, respectively, are the cluster assigned to each computation bursts of the task/thread. Using a MSA algorithm to align the sequences that represent each task/thread we can easily evaluate the desired *SPMDiness* of each cluster.

Following the guidelines of MSA selection presented in [102] we selected Kalign2 [89] from the list of different MSA packages detailed in chapter 3. Due to the availability of the source code of Kalign2 we performed the modifications so as to extend this tool to work with an arbitrary alphabet. This is a very limiting element because almost all MSA packages only allow using DNA or protein alphabets, which limit the possible alignments to 27 different elements, the clusters in our case.

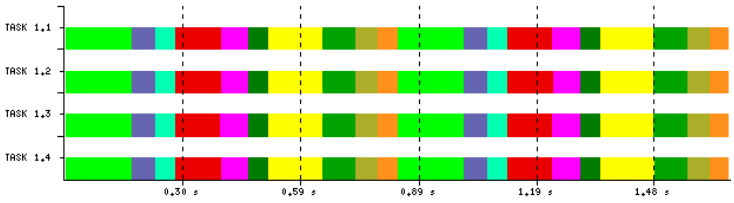
In addition, we also modified the algorithm core to dynamically generate the score matrix, used to prioritize which clusters of the sequence should be aligned together. This manually tuned matrix is a variation of the identity matrix, where those clusters with a higher percentage duration have a higher alignment score. In this way, Kalign2 always tries to align the most important clusters first. This tune guarantees serves to improve the aligment results, by using those clusters that represent more computation time as reference points in the alignment process.

An important fact is that after applying the MSA in the output alignment all sequences have the same length. This property is required to define the sequence score described in next section, as it iteratively evaluates each position of all sequences to quantify the SPMDiness.

Figure 5.1 shows an example of an alignment obtained using Kalign2 using the sequences that represent a trace of NPB BT class A with four tasks. In 5.1a the time-line with the cluster information is presented. Next, in 5.1b we present the alignment obtained using Kalign2, using the ClustalX [86] application. Note that in 5.1b there are asterisks on top of each position of the sequences. These are marks introduced by ClustalX to indicate that all sequences in the given position have the same element of the alphabet.

5.3. Cluster Sequence Score

Once the sequence that represent each task/thread of the parallel application have been aligned, the actual Cluster Sequence Score is computed. The score has two parts: the percentage of total alignment for each cluster, the *score per cluster*, and the *global score*, the weighted combination of the scores per cluster, using the cluster aggregate duration. To compute the *score per cluster* we create a matrix $S_{m \times n}$ using the aligned sequences. m is the number of tasks and n the length of the aligned sequences. Each cell $s_{i,j}$ represent the cluster identifier assigned to computation burst of task/thread i in the position j of the sequence.



(a) Time-line

(b) Kalign2 alignment

Figure 5.1.: An example of alignment of the NAS BT Class A with 4 tasks. (a) shows the cluster distributation in the time-line; (b) presents the proposed alignment computed by Kalign2 and depicted by ClustalX.

Then the score per of each cluster $C_x \in C$, $Score(C_x)$, is defined in the following equations:

$$Score(C_x) = \frac{Alignment(C_x)}{Appearances(C_x)} \tag{5.1}$$

$$Alignment(C_x) = \sum_{i=1..n} \frac{\sum_{j=1..m} Hit_{C_x}(s_{i,j})}{n} \tag{5.2}$$

$$Hit_{C_x}(s_{i,j}) = \begin{cases} 1 & if\ x = s_{i,j} \\ 0 & otherwise \end{cases} \tag{5.3}$$

$$Appearances(C_x) = \sum_{j=1..n} SingleAppearance_{C_x}(j) \tag{5.4}$$

$$SingleAppearance_{C_x}(j) = \begin{cases} 1 & \exists i; 1 \leq i \leq m; x = s_{i,j} \\ 0 & otherwise \end{cases} \tag{5.5}$$

Conceptually, the *score per cluster* can be understood as "for each position in the sequence where the cluster C_x appears, measured as the $Appearances(C_x)$, which percentage of the tasks are executing this cluster, computed by $Alignment(C_x)$". The score is expressed indistinctly in percentages or in range 0 to 1, being 100% or 1 a perfect score which means that every time a cluster appears in a sequence position, it appears in all tasks/threads.

$GlobalScore(C)$, the *global score* fore the whole set of the resulting cluster set RC is

defined as:

$$GlobalScore(C) = \sum_{C_x \in C} Score(C_x) \times PercentageDuration(C_x) \tag{5.6}$$

In this case, the *global score* is a weighted average of the *scores per cluster* using the percentage duration of each cluster as a weight. It is a single percentage value which shows the quality of the total structure detected. A larger percentage signifies a higher quality.

This formulation is implemented in an algorithm which evaluates each matrix column sequentially to compute the score of each cluster. Using the output of the cluster duration from the structure detection tool, the *global score* is computed.

Experiment	Application	Eps	MinPoints	Total Clusters
bt_a_16_0057	BT Class A	0.0057	10	43
bt_a_16_0150	BT Class A	0.0150	10	17
ft_a_16_0090	FT Class A	0.0090	4	16
lu_a_16_0175	LU Class A	0.0175	4	7
wrf_16_0100	WRF	0.0100	10	21
wrf_16_0200	WRF	0.0200	10	16
wrf_16_0300	WRF	0.0300	10	13

Table 5.1.: Summary of the different applications used in the experiments, as well as the DBSCAN parameters used and the total number of clusters obtained

5.4. Validation

The validation of the Cluster Sequence Score consists on manually checking that it correctly evaluates different results in terms of *SPMDiness*. We used 3 different NAS Parallel Benchmarks: BT, FT and LU, all of them using Class A inputs. We also analysed the results of a simulation of WRF. In all cases, the applications were executed using 16 tasks. We chose not to scale the test to a high number so as to present more understandable time-line and alignment figures.

As we mentioned in the motivation of this chapter, the Cluster Sequence Score is intended only to evaluate structure detections of SPMD applications. All the applications we use here follow this pattern. In fact, this is an *a priori* knowledge that we must have regarding the application design, to guarantee the applicability of the score. This check can be easily guaranteed by consulting the application developers, but as we pointed before, the vast majority of message-passing parallel applications are SPMD.

Table 5.1 summarizes the different experiments, the parameters of DBSCAN used and the total number of clusters obtained for each case. Figures from 5.2 to 5.8 contain the cluster

time-lines, the aligned sequences, and also the Cluster Sequence Score plus the CDbw value of each experiment.

The CDbw value is the value of the validity index for clustering algorithms presented by Halkidi et. al. in [101]. This index evaluates the quality of a given partition of the data based on distance metrics and the density distribution of the data. To do so, each cluster is modelled by a set of r representative points, instead of a centroid. It defines three different values for the set of clusters C detected:

- $Compactness(C)$: the average value of the number of individuals of each cluster $C_i \in C$ that are within one standard deviation from its closest representative.
- $Cohesion(C)$: the value of the compactness of each cluster in the clusters set, weighted by the density variations observed around the cluster representatives.
- $Sep(C)$ (Separation): the average distance of between the clusters detected, weighted to the maximum density observed between them. This inter-cluster density is proportional to the number of the element in the data set within one standard deviation from the mid-point of the closest representatives in different clusters.

Finally, the CDbw index of a set of clusters C is defined as:

$$CDbw(C) = Cohesion(C) \times Sep(C) \times Compactness(C) \tag{5.7}$$

We use this validity because is one of the few approaches that takes into account the density distribution of the clusters, a basic characteristic in out data sets. We want to note that due its definition higher values of the index represent better results.

The different DBSCAN parameters used on BT and WRF serve as example to show the ability of the *global score* to capture the *SPMDiness* of structure detection. The best way to determine if the results are correct is to visually compare the time-lines, sequences and scores in Figures 5.2 and Figure 5.3 to evaluate the BT benchmark. Identically, same evaluation could be done for WRF, using the Figures 5.4, 5.5 and 5.6.

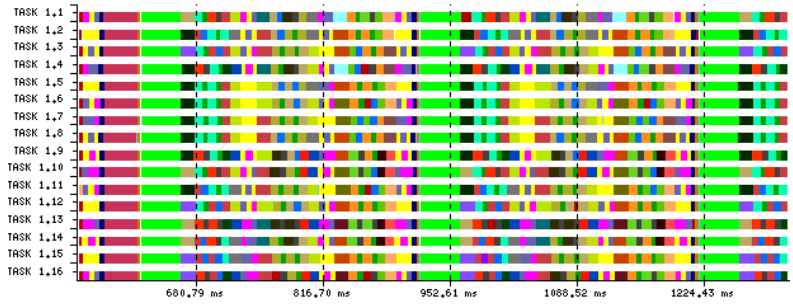
NPB BT

In the first experiment of BT, `bt_a_16_0057`, presented in Figure 5.2, we used a Eps value (0.0057), obtained using the technique proposed by DBSCAN authors in [78]. As we explained in previous chapter, this technique does not provide good choices of Eps parameter to our purposes, as it usually suggests too restrictive values. It is easy to see that the structure detected in the time-line, 5.2a, does not show any SPMD structure, nor the aligned sequences, 5.2b. Even having Cluster 1 (the green region) well aligned, the rest of the structure is not clear, and one can see the high number of gaps ($-$ symbol) that the MSA introduced. In terms of the score, 5.2c, this is translated into a good score for Cluster 1 (also for Cluster 2), but a poor global score of 0.341.

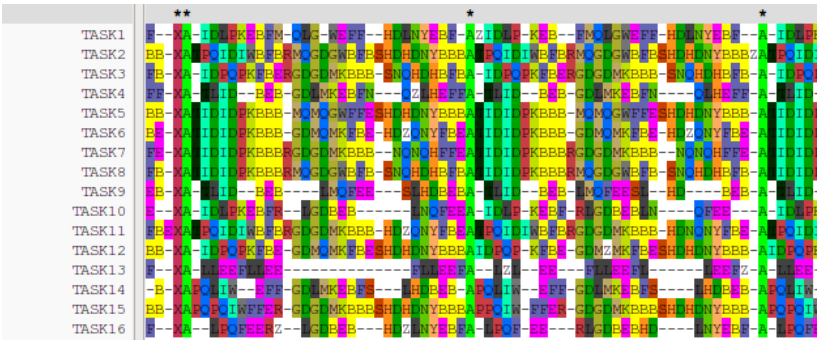
In contrast, the experiment `bt_a_16_0150`, presented in Figure 5.3, shows the results using a higher value of Eps (0.0150). As we commented in previous chapter, higher values of Eps detect better the SPMD structure. In this experiment, one can easily see the different steps

inside the iteration of this code, 5.3a, as well as the high precision of the alignment, 5.3b. The Cluster Sequence Score correctly evaluates this situation giving a perfect score (1.000) for first 4 clusters and a global score of 0.969.

Comparing the CDbw scores obtained by the two experiments we demonstrate the affirmation that this validity index does not apply in our context. It obtained higher value for the poorest structure detection, 41.063 for `bt_a_16_0057` experiment and 36.236 for `bt_a_16_0150` experiment, the opposite of our purpose.



(a) Time-line of resulting clusters

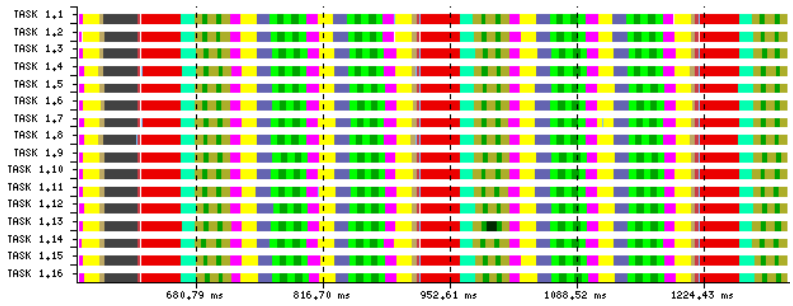


(b) Aligned sequences

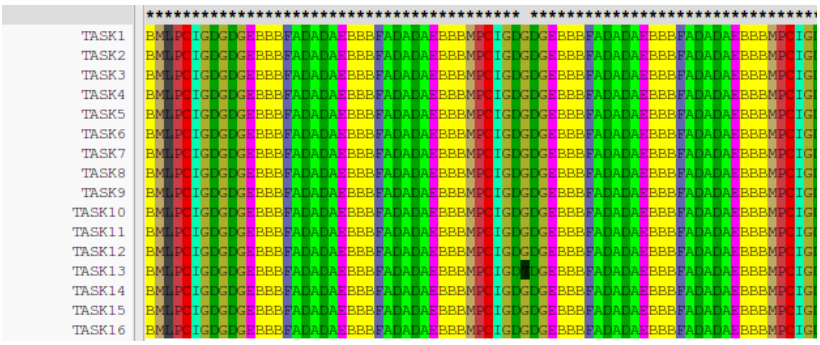
Score per Cluster (% Duration)				Global	
Cluster 1	Cluster 2	Cluster 3	Cluster 4	Score	CDbw
1.000 (13.66%)	0.414 (8.39%)	1.000 (7.25%)	0.311 (5.91%)	0.341	41.063

(c) Cluster Sequence Score and CDbw

Figure 5.2.: Results of the experiment `bt_a_16_0057`. Time-line (a), and alignment (b) correspond to two iteration detail of the whole application



(a) Time-line of resulting clusters



(b) Aligned sequences

Score per Cluster (% Duration)				Global	
Cluster 1	Cluster 2	Cluster 3	Cluster 4	Score	CDbw
1.000 (15.56%)	1.000 (15.45%)	1.000 (12.56%)	1.000 (11.34%)	0.969	36.236

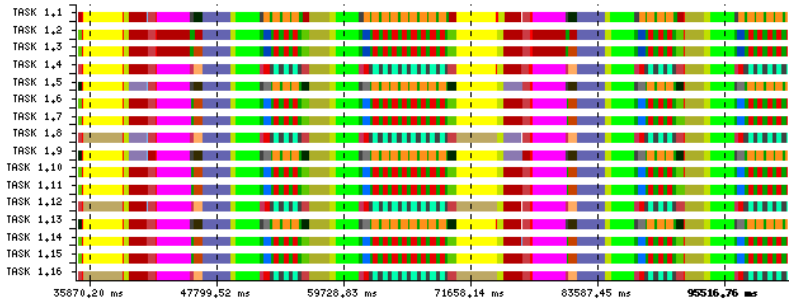
(c) Cluster Sequence Score and CDbw

Figure 5.3.: Detected structure, sequences alignment and Cluster Sequence Score results of `bt_a_16_0150` experiment. The figures corresponds to two iterations of the whole application

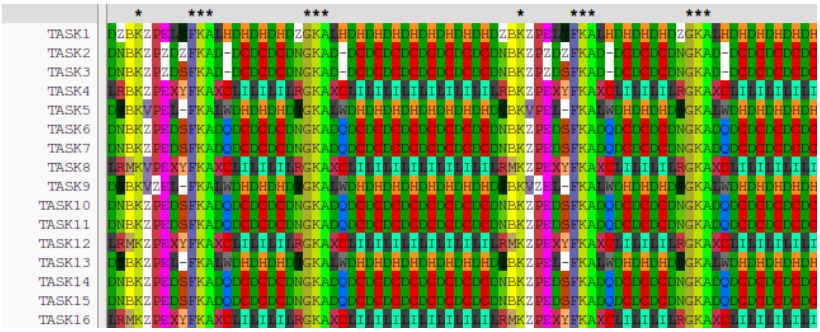
WRF

The same discussion is applicable for the experiments using WRF. In this case, we used three different values of Eps. The first one was the suggested by the DBSCAN parameter selection technique, 0.0100, and whose results are depicted in Figure 5.4. The second experiment used an Eps double of the first value, 0.0200, collected in Figure 5.5. The last one Eps was three times the original value, 0.0300, presented in Figure 5.6.

Looking and comparing the time-lines and alignment plots, we can observe that the SPMD structure result clear using higher Eps values than the suggested by the original technique. The structure is not as clear as the best one obtained with the BT benchmark, as we did not exhaustively search for the *best* Eps value. The Cluster Sequence Scores obtained, 0.661, 0.736 and 0.887 respectively, capture the increase of the *SPMDiness* observed informally. In this case the CDbw again produced the opposite results we expected.



(a) Time-line of resulting clusters

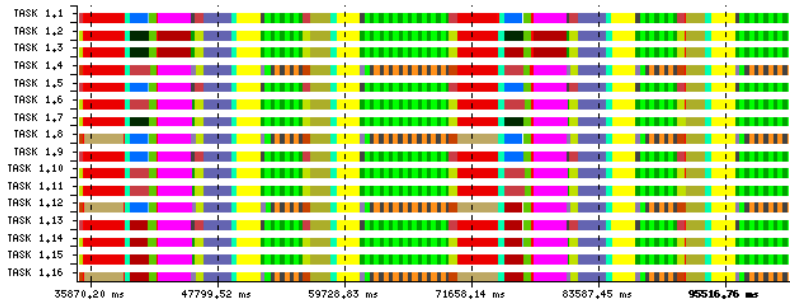


(b) Aligned sequences

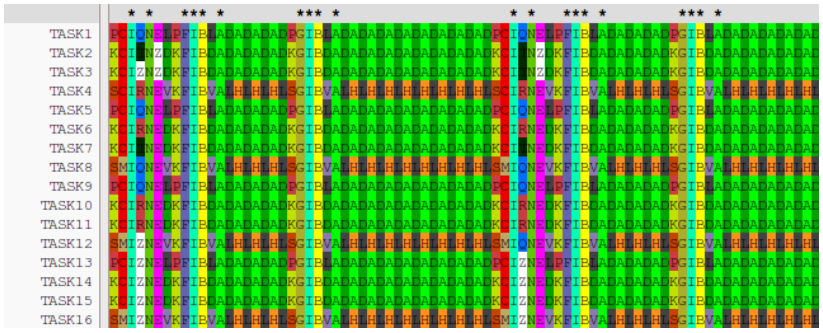
Score per Cluster (% Duration)				Global	
Cluster 1	Cluster 2	Cluster 3	Cluster 4	Score	CDbw
1.000 (14.31%)	0.800 (11.17%)	0.462 (11.04%)	0.703 (9.93%)	0.661	52.769

(c) Cluster Sequence Score and CDbw

Figure 5.4.: Detected structure, sequences alignment and Cluster Sequence Score results of `wrf_16_0100` experiment. The figures correspond to two iterations of the whole application



(a) Time-line of resulting clusters

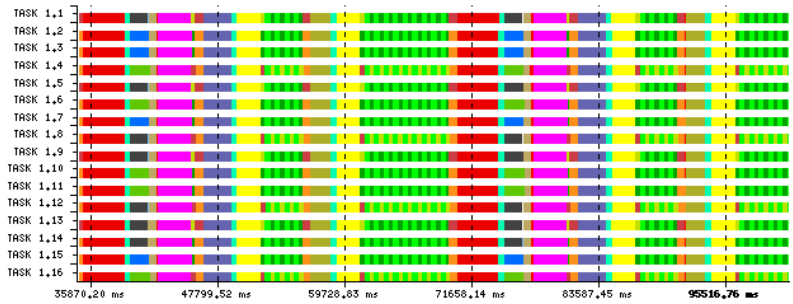


(b) Aligned sequences

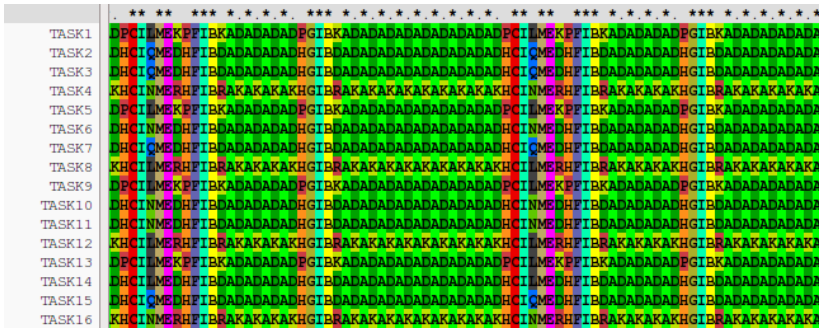
Score per Cluster (% Duration)				Global	
Cluster 1	Cluster 2	Cluster 3	Cluster 4	Score	CDbw
0.788 (17.64%)	1.000 (13.42%)	0.800 (10.48%)	0.703 (9.31%)	0.736	22.154

(c) Cluster Sequence Score and CDbw

Figure 5.5.: Detected structure, sequences alignment and Cluster Sequence Score results of `wrf_16_0200` experiment. The figures correspond to two iterations of the whole application



(a) Time-line of resulting clusters



(b) Aligned sequences

Score per Cluster (% Duration)				Global	
Cluster 1	Cluster 2	Cluster 3	Cluster 4	Score	CDbw
1.000 (22.50%)	1.000 (13.49%)	0.988 (13.01%)	0.703 (9.36%)	0.887	16.120

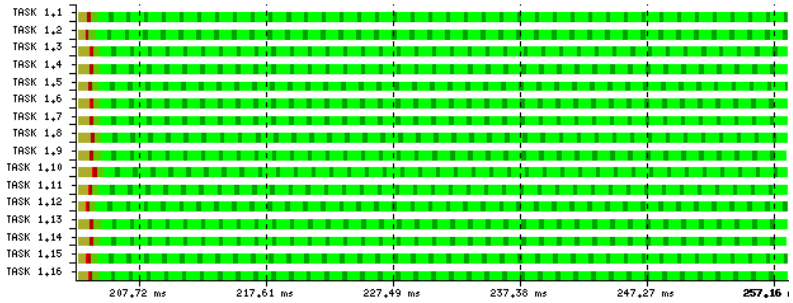
(c) Cluster Sequence Score and CDbw

Figure 5.6.: Detected structure, sequences alignment and Cluster Sequence Score results of `wrf_16_0300` experiment. The figures correspond to two iterations of the whole application

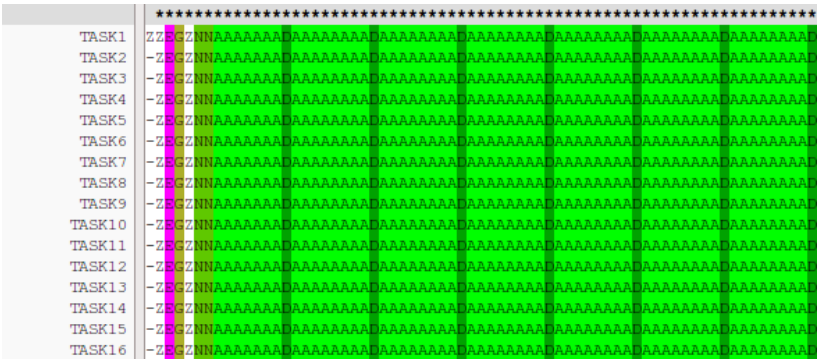
NPB FT and LU

The two other NAS Benchmarks tested, FT and LU, resulted in good structure detection using the initial parameter approximation as shown in Figure 5.7 and Figure 5.8, respectively. In both cases, we get a global score higher than 0.9, as can be seen in Tables 5.7c and 5.8c. This was caused by the total application time of the four main clusters summing to 71% and 95%, respectively. Being well aligned, the global score (and, obviously, the scores per cluster) results in excellent values.

As these experiments just used a single Eps, we do not compute the CDbw index.



(a) Time-line of resulting clusters

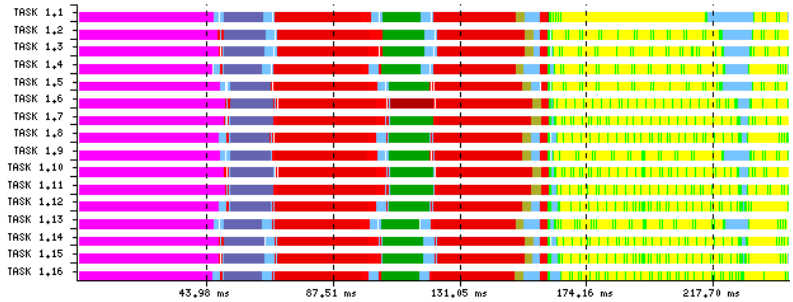


(b) Aligned sequences

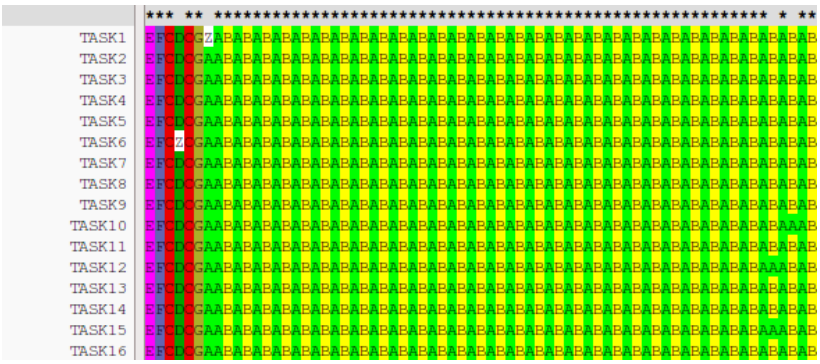
Score per Cluster (% Duration)				Global
Cluster 1	Cluster 2	Cluster 3	Cluster 4	Score
0.999 (47.08%)	1.000 (14.58%)	0.972 (5.61%)	0.996 (5.03%)	0.963

(c) Cluster Sequence Score

Figure 5.7.: Detected structure, sequences alignment and Cluster Sequence Score results of `ft_a_16_0090` experiment. The figures correspond to a single iteration of the whole application



(a) Time-line of resulting clusters



(b) Aligned sequences

Score per Cluster (% Duration)				Global
Cluster 1	Cluster 2	Cluster 3	Cluster 4	Score
0.977 (36.85%)	0.997 (28.01%)	0.995 (22.08%)	0.994 (9.03%)	0.988

(c) Cluster Sequence Score

Figure 5.8.: Detected structure, sequences alignment and Cluster Sequence Score results of `lu_a_16_0090` experiment. The figures correspond to a single iteration of the whole application

6. Automatization of the Structure Detection

In this chapter we present the Aggregative Cluster Refinement, a new cluster algorithm that automatizes the fine-grain structure detection of SPMD parallel applications. It overcomes the limitations of DBSCAN and also avoids the user to face the, sometimes hard, problem of selecting the DBSCAN parameters.

6.1. Limitation of the structure detection based on DBSCAN

Two major problems can be associated to the DBSCAN algorithm. The first one is the sensitivity to the parametrization. As we shown in chapter 4, a given combination of Eps and $MinPoints$ could lead to structure detections that do not show any SPMD pattern. Even DBSCAN authors proposed a technique to tune the parameters, our experience points that the user require a series of trial-and-error analyses to detect the precise values.

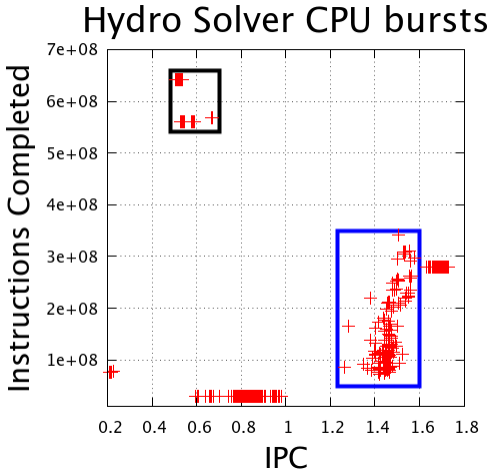


Figure 6.1.: Example of data set that require multiple Eps parameters. This data was obtained from the Hydro solver executed with 128 tasks

The second problem is the inability of DBSCAN to correctly detect clusters when the density varies across the data space. This problem is directly related to the use of a single Eps value. In Figure 6.1 we present a practical example. The black box in this Figure marks a region of the data set where compact clouds of points appear. To correctly detect these

clouds as different clusters we require using a small Eps value. On the other hand, the region inside the blue box contains a group of points that exhibit more variability. To detect this second region as a single cluster we require a higher Eps. The fact that DBSCAN just can use one Eps implies that if we use a small value, the algorithm will detect correctly the clouds present inside the black box, but the cloud inside the blue box will be detected as multiple clusters and also noise points. On the other hand, if we used a high Eps value, the cloud inside the blue box will be detected as a single cluster, but the clouds of points inside the black box will be merged together in a single or two clusters (we call this an over-aggregation of clusters).

6.2. The Aggregative Cluster Refinement algorithm

With these two problems in mind, we present the Aggregative Cluster Refinement algorithm. This is a hierarchical algorithm based on merging the density components that conform a final cluster. It has been inspired by previous cluster algorithms and presents two major properties:

1. Is able to detect clusters that present different densities in the data space.
2. Minimizes the number of input parameters needed to correctly detect the desired clusters.

6.2.1. Aggregative Cluster Refinement foundations

As explained in chapter 3, one of the most basic algorithms used in cluster analysis is the hierarchical clustering [77].

An implicit parallelism exists between hierarchical algorithms and DBSCAN. Using a given data set and using a fixed value of $MinPoints$, each Eps value we use can be understood as a cut at different heights of the dendrogram. In this way, if we use a lower (restrictive) Eps value, the "cut" will be close to leaves in the dendrogram, keeping isolated points that will be classified as noise. On the other hand, using a less restrictive Eps will produce a "cut" close to the root of the dendrogram, where big clusters group large number of individuals with more variability across them.

The second fundamental part of the Aggregative Cluster Refinement algorithm is the scheme to detect the clusters. In this case, it is inspired by X-Means. X-Means guesses the number of clusters k using K-means algorithm by iteratively increasing the value of k. On each iteration it runs the K-means algorithm, and evaluates the resulting clusters using the Bayesian Information Criterion, to decide if a new iteration is required.

The Aggregative Cluster Refinement is an iterative algorithm that builds a pseudo-dendrogram, also named *refinement tree*, similarly to the aggregative hierarchical cluster algorithms. On this refinement tree the leaf nodes represent clusters obtained using DBSCAN with an initial low Eps value. The intermediate nodes represent clusters obtained using progressively larger Eps values. In these intermediate nodes, DBSCAN is only applied to the individuals that belong to those clusters that do not pass the a cluster quality criterion in the previous

level. In this way, the Aggregative Cluster Refinement algorithm uses the scheme defined by X-Means. Those nodes that pass the criterion do not take part in further levels and are reported as an actual cluster by our new algorithm.

As the target of our studies is to detect the SPMD structure of parallel applications, we use the Cluster Sequence Score as the criterion to evaluate the clusters quality.

6.2.2. Algorithm Description

The Aggregative Cluster Refinement pseudo-code is listed in Algorithm 1. The inputs of the algorithm are the data set composed by the counters associated to each CPU burst, $CPUBurstsSet$; the number of different Eps we want to use, N; and the number of tasks present in the application we want to analyse, $ApplicationTasks$. The outputs of the algorithm are a partition of the data, $FinalPartition$, with the cluster identifiers assigned to each burst in the input set, the Cluster Sequence Score of the last partition, $FinalScores$, and the pseudo-dendrogram of the hierarchical refinement, $RefinementTree$. Additionally, the user can also retrieve the intermediate partitions and scores, stored in $Partitions$ and $Scores$ lists.

The algorithm starts setting $MinPoints$ as a quarter of the total tasks the application has. This value is selected because we consider that the minimum acceptable SPMD region should cover, at least, 25% of the total tasks. Next, using the data set the N different Eps are generated in `ComputeEpsilons`, sorted increasingly and stored in the $EpsList$ list. This generation chooses N values using a technique described later.

On each iteration, the clusters which pass the criterion are discarded, and just the burst that belong to clusters with poor score take part in further DBSCAN executions with higher Eps values (the $CPUBurstsSubset$). This bound is performed in `GenerateCandidatePoints`. To our purposes, we consider that a cluster passes the criterion if it gets an cluster score of 100%, using the Cluster Sequence Score defined in the previous chapter.

The algorithm converges if all clusters pass the criterion before exploring all N different Eps values. If that does not happen in N iterations, in `ProcessLastPartition` we merge those clusters that do not have perfect score but occur simultaneously according to the aligned sequences used by the Cluster Sequence Score.

The outputs the algorithm produces are the $FinalPartition$ of the data, that contains the cluster identifier assigned to each input point and a the $RefinementTree$. In this tree, each level contains the clusters generated on a given iteration of the algorithm. Each clusters of an intermediate level is connected to those clusters of immediately previous level that merged to generate it as a result of increasing the Eps value, or to the noise cluster node if it has *absorbed* points previously classified as noise. It is interesting to note that the final clusters produced by the Aggregative Cluster Refinement are those nodes not connected to clusters at further levels. Actually, the refinement tree is a collection of trees, where each of the clusters present on the final partition of the data corresponds to the different tree roots in this collection.

Additionally to these two outputs, the algorithm can also save each intermediate partition of the data, named $Partition_i$ in the algorithm pseudo-code, to complement the information of the refinement tree to study of the hierarchical formation of the clusters.

As a conclusion of this description, we want to remark that the properties of the Aggregative Cluster Refinement listed at the beginning of this section are accomplish due to following facts:

1. The user just has to provide the data set and the maximum number of refinement iterations N he or she wants to execute.

2. As it uses multiple Eps values it is able to detect clusters that present different densities.

Input: CPUBurstSet = points representing the CPU Bursts of the application
Input: N = number of maximum refinement steps
Input: ApplicationTasks = number of application tasks
Output: FinalPartition = resulting data partition obtained
Output: FinalScores = Cluster Sequence Scores obtained by the last partition
Output: RefinementTree = pseudo-dendrogram of the refinement process

MinPoints = ApplicationTasks/4;

`ComputeEpsilons` (CPUBurstSet, EpsSet, N);

CPUBurstSubset = CPUBurstSet;
i = 1;
while *CPUBurstSubset* $\neq \emptyset$ **and** $i \leq N$ **do**
    `RunDBSCAN` (CPUBurstSubset, MinPoints, EpsSet_i, Partitions_i);
    `ComputeScores` (CPUBurstSet, Partitions_i, Scores_i);
    `UpdateTree` (CPUBurstSet, Partitions_i, Scores_i, RefinementTree);
    `GenerateCandidatePoints` (CPUBurstSet, Scores_i, CPUBurstSubset);
    i = i + 1;
end

`ProcessLastPartition`(CPUBurstSet, Partitions_i, Scores_i, PartitionsPostProcessed, ScoresPostProcessed);

`UpdateTree` (CPUBurstsSet, PartitionsPostProcessed, ScoresPostProcessed, RefinementTree);

FinalPartition = PartitionsPostProcessed;
FinalScores = ScoresPostProcessed;

Algorithm 1: Aggregative Cluster Refinement

Algorithm example results

In Figures 6.2 and 6.3 we present example to illustrate how the algorithm works. The refinement tree 6.2 is depicted top-down (first iteration on the top of the figure), the fill nodes represent the clusters of the final partition. Each node is decorated with the cluster name and the *score per cluster* obtained at this iteration.

In this example, we used $N = 10$ but the algorithm converged in 7 iterations as can be seen in the refinement tree in Figure 6.2. Different time-lines in Figure 6.3 correspond to the intermediate partitions obtained on those iterations where the algorithm detected clusters that will be part of the final partition (marked with a black dot), i.e. clusters that passed the score. The time-line 6.3d corresponds to the computation structure detected by the final partition the algorithm produced.

It is important to note that the information of the intermediate partitions the algorithm provides is an useful piece of information to understand the application structure at fine-grain. In section 6.3 we illustrate how these intermediate partitions plus the refinement tree can be useful to detect detect the performance variability that can appear in a given SPMD region.

Epsilon selection

As we mentioned before, the selection of the different Eps plays a crucial role in this methodology, as the separation or aggregation of clusters directly depend on this parameter.

In the original paper where DBSCAN is described [78], the authors propose an interesting technique to choose a good Eps candidate for a given data set. The technique starts selecting the value of the $MinPoints$ parameter. Then computes the distance of each individual of the data set to its k neighbour, being k the selected value of $MinPoints$. These distances are sorted decreasingly and presented in a graph. In Figure 6.4 *sorted k-dist* graph is shown as the red curve. Finally, the way to choose the ideal Eps proposed in [78] is to manually select it around the right-most "valley" of these series.

We explained in chapter 4 that when we applied this technique to the CPU burst hardware counters data, the resulting Eps selected tended to be small, producing high number of clusters and structure detections that do not follow the SPMD pattern. For this reason, we define the following technique to select the N different Eps values required, where we use the sorted k-dist graph to detect a lower bound of the Eps values to generate.

1. First, being n the number of points in the data set, we define the line that contain the points $(0, max\_k\_dist)$, the first point in the *sorted k-dist* graph, and $(n/2, 0)$, the middle point in the x range. This line is depicted in blue in Figure 6.4.

2. For each value of the x in interval 0 to $n/2$, we compute the difference between its *sorted k-dist* and its projection in line defined.

3. We select as the lower bound of Eps, Eps_N the value of the *sorted k-dist* whose is distance to the defined line is the maximum. As the *sorted k-dist* is a monotone decreasing curve, this maximum difference to the defined line represents a "valley" near to the described in [78].

4. The $N - 1$ remaining Eps values are selected dividing the range between $x = 1$ (the point with the second maximum sorted k-distance) and x projection of the selected Eps_{min} by $N - 1$ times.

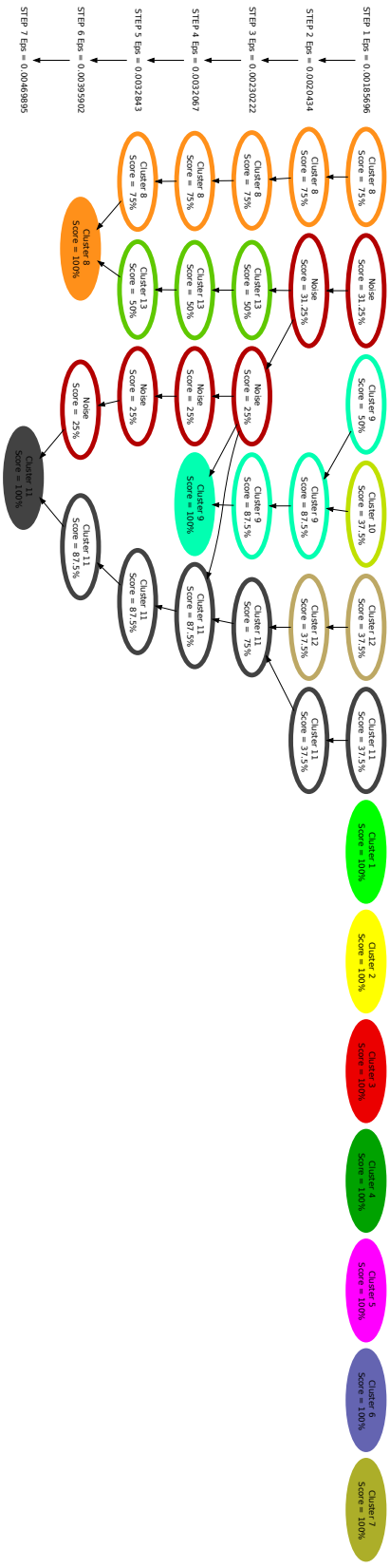
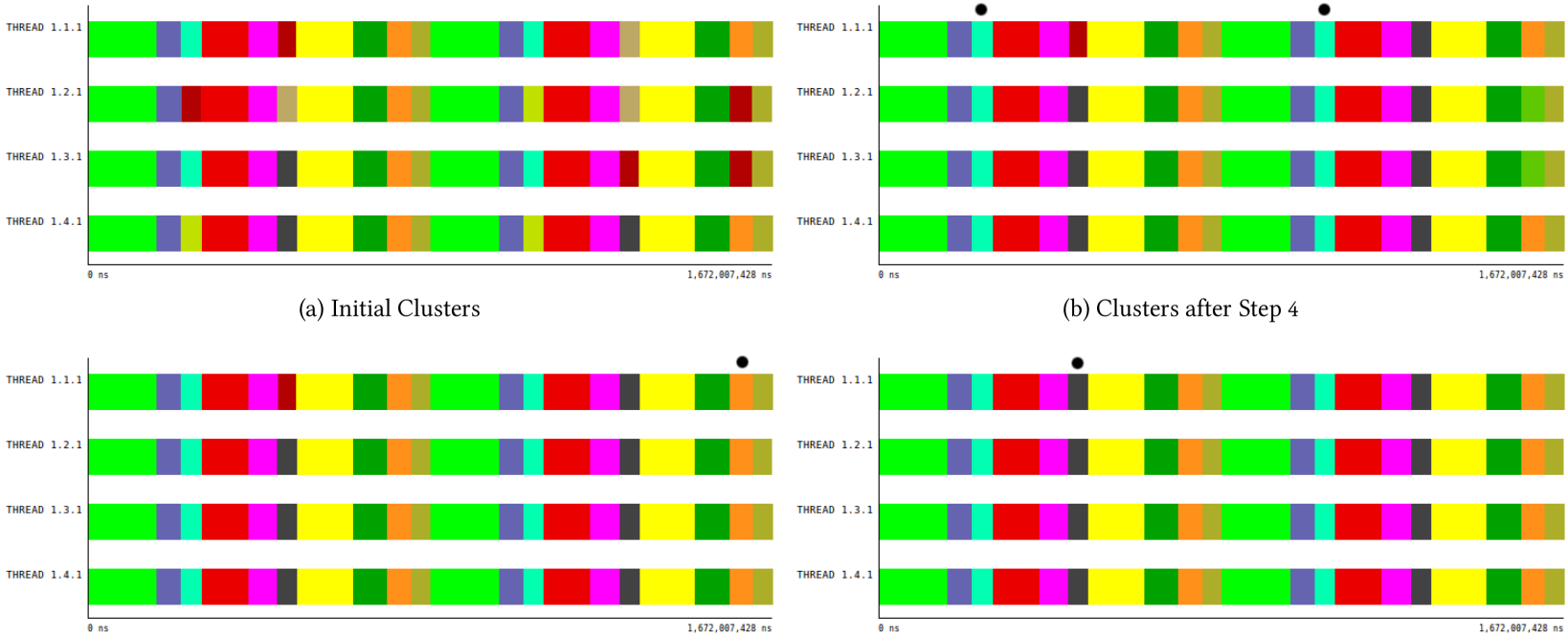


Figure 6.2.: Complete aggregative refinement cluster analysis tree obtained from NPB BT class A executed with 4 tasks. The empty nodes of the tree depict those clusters that are discarded because of poor SPMDiness and thus need to be merged. Filled nodes are those selected in the final partition of the data. In this case, due to the convergence, all selected nodes got 100% score. Each layer represents one step in the refinement loop.



(a) Initial Clusters

(b) Clusters after Step 4

(c) Clusters after Step 6

(d) Final Clusters

Figure 6.3.: Application time-lines expressing the clusters found at different steps of the Aggregative Refinement corresponding to Figure 6.2. (a) is the initial clustering, Step 1, where most of the main SPMD phases have already been discovered. (b) Time-line of the Step 4, where Cluster 9 (light green) represents a SPMD phase not previously detected. (c) Time-line of the Step 6, where Cluster 8 (orange) the one that represents a SPMD phase not correctly detected before. (d) Time-line of the final partition of the data, Step 7 in the tree, where Cluster 11 represents the last SPMD phase found. Dots on the top of the time-lines serve as guide to see the clusters mentioned.

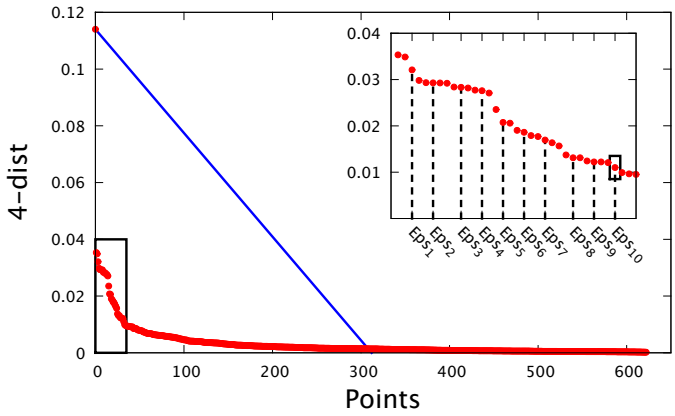


Figure 6.4.: *sorted 4-dist* graph obtained from the Socorro application. The red dots are represent the distance to the 4th nearest neighbour for each point in the dataset. The blue line is the line defined by the points (0, max_k_dist) and ($number\_of\_points/2$, 0). We use it to compute the different Eps values

In Figure 6.4, the black box indicates the range where we select the different Eps values. This range is is shown in the inner plot where the point corresponding to Eps_{10} is the point lower bound computed using the distances to the blue line, and the rest of Eps_i values correspond to the selected range. It is important to note that we never use as Eps the sorted k-dist corresponding to the first point, the maximum value, because it tends to provoke an over-aggregation when merging the clusters.

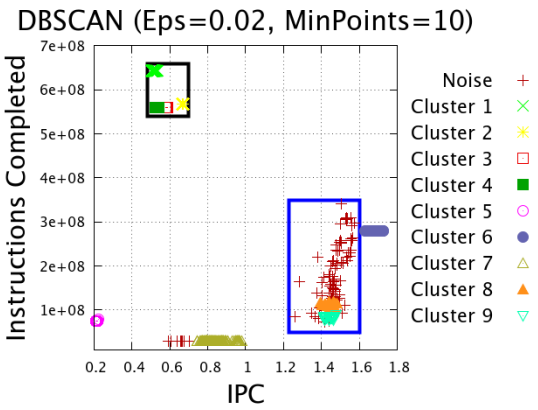
6.3. Aggregative Cluster Refinement results

In this section we demonstrate the utility of the Aggregative Cluster Refinement algorithm in two aspects. First, and most important, its ability to capture the desired SPMD structure of parallel applications even when the data present different densities, surpassing the structure detection DBSCAN provides. Second, the information of about the the hierarchical formation of the clusters detected using the refinement tree and the intermediate partitions it produces.

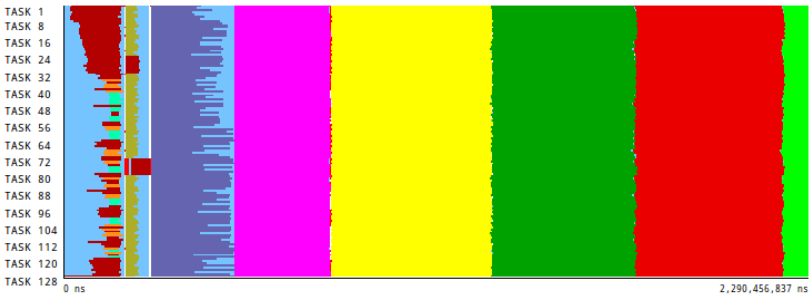
6.3.1. SPMD structure detection

We present the computation structure detected in two production class applications by using the Aggregative Cluster Refinement algorithm. These examples illustrate the ability of the Aggregative Cluster Refinement to capture the desired SPMD structure of these applications even when they show different densities in the data space of the performance metrics used. In addition, we compare the detected structure of each application to the structures obtained by using DBSCAN with different Eps values. These comparisons reflect how the new algorithm we propose is able to overcome the DBSCAN limitations observed.

We want to remark that to obtain the computation structures using the Aggregative



(a) Scatter-plot of discovered clusters



(b) Clusters time-line distribution

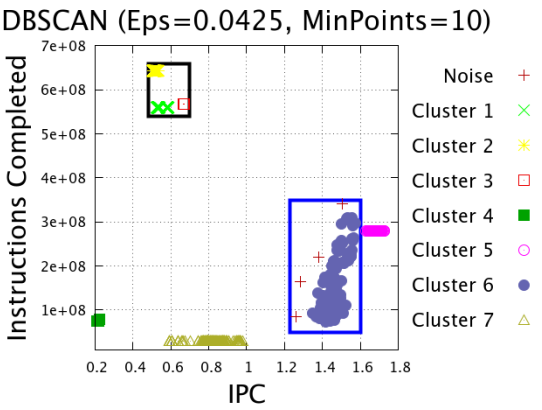
Figure 6.5.: Computation structure detection of Hydro solver, using DBSCAN cluster algorithm with the parameters $MinPoints = 10$ and $Eps = 0.0200$

Cluster Refinement, we just provide a single parameter N, the number of refinement steps. This parameter was always set to 10. As opposite, when using DBSCAN we need to fine-tune the Eps parameter by running several times the algorithm, so as to detect the computation regions that clearly reflect the SPMD structure.

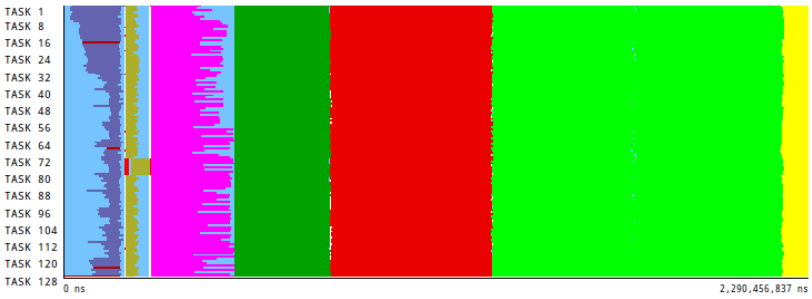
Hydro Solver

The application selected for the first example is the Hydro Solver. In this example we use the same data set we presented in Figure 6.1 to illustrate the problem of DBSCAN when having a dataset with multiple densities problem.

In Figure 6.5 we present the results obtained applying DBSCAN with the parameters $MinPoints = 10$ and $Eps = 0.0200$. We can see in the plot 6.5a that the four clouds inside the black box have been detected as four different clusters. On the other hand, the cloud inside the blue box contains two Clusters, 8 (orange) and 9 (light blue), and high number of noise points (brown points). Observing the computation structure these clusters provide (time-line 6.5b) we detect that the clusters contained in the blue box reflect pure SPMD regions at the end of the time-line while the clusters inside the blue box clusters appear in the initial section of the time-line not reflecting the actual SPMD behaviour.



(a) Scatter-plot of discovered clusters



(b) Clusters time-line distribution

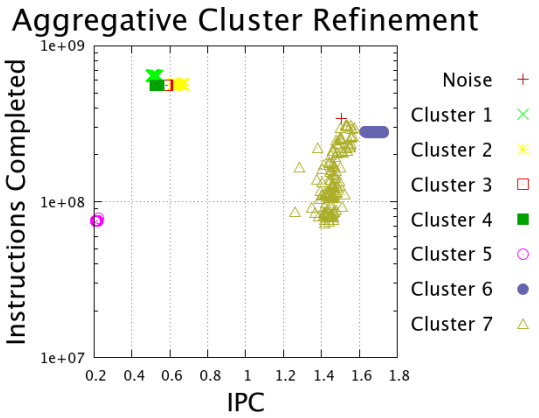
Figure 6.6.: Computation structure detection of Hydro solver, using DBSCAN cluster algorithm with the parameters $MinPoints = 10$ and $Eps = 0.0425$

To capture the variability of the cloud inside the blue box we need to increase the Eps to 0.0425. The results of using this Eps are contained in Figure 6.6. We see in the scatter plot 6.5a that this cloud is detected as single Cluster 6 (purple). Unfortunately, the higher value Eps provokes an over-aggregation of the clouds in the black box. In this plot Clusters 3 and 4 of the previous analysis now are merged in Cluster 1. Looking at the time-line 6.6b, we confirm these observations: in the initial section of the time-line we can see the SPMD region represented by the Cluster 9 (purple), while the four SPMD regions detected in the previous example, now are depicted as just three regions.

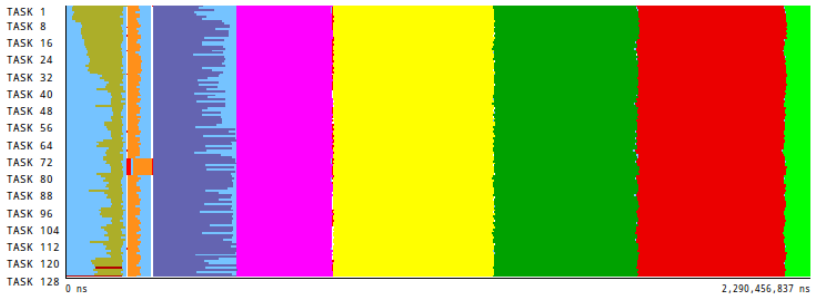
Applying the Aggregative Cluster Refinement to this performance data, we obtain the results depicted in Figure 6.7. We can see in the plot 6.7a how the algorithm detected as four different clusters the clouds present in the black box and a as single cluster the cloud inside the blue box. With the time-line 6.7b we can confirm that all SPMD regions have been clearly distinguish, surpassing the results obtained with DBSCAN.

Weather Research Forecast (WRF)

For this second example, we used the Weather Research Forecast (WRF) application, a weather simulator. More information about this application is detailed in chapter 9. We

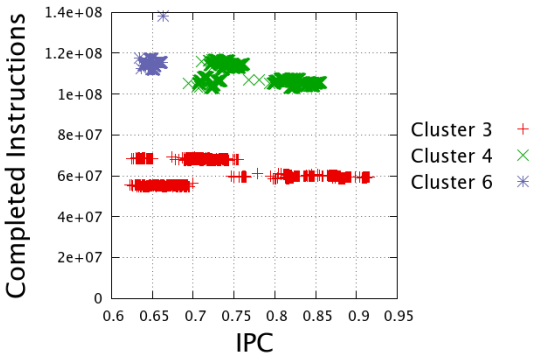


(a) Scatter-plot of discovered clusters

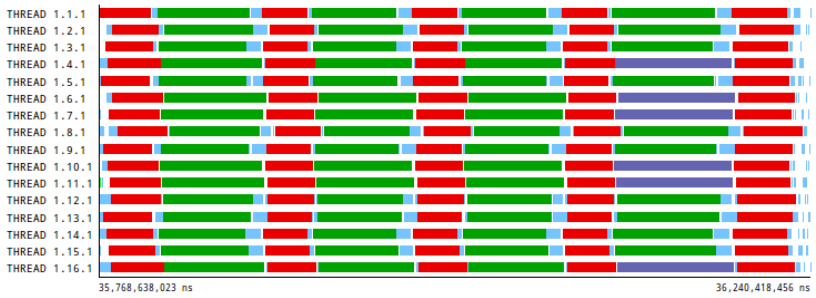


(b) Clusters time-line distribution

Figure 6.7.: Computation structure detection of Hydro solver, using Aggregative Cluster Refinement algorithm



(a) Scatter-plot detail



(b) Time-line zoom

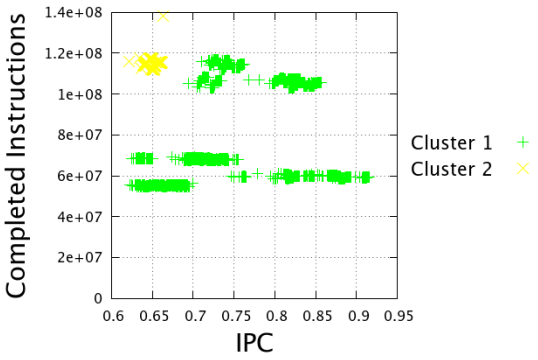
Figure 6.8.: Detailed results of a DBSCAN cluster analysis of same phases present in Figure 6.9 of WRF application. The parameters used were $MinPoints = 4$ and $Eps = 0.0470$

used a small execution of 16 tasks.

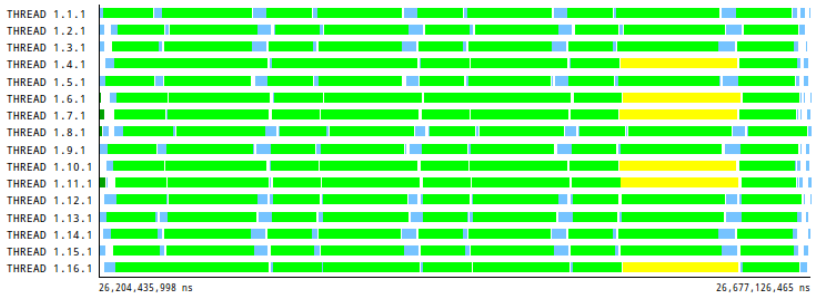
In the different the scatter plots used to illustrate this example, we only present the regions of the dataset where DBSCAN cannot detect correctly the different clouds while the Aggregative Cluster Refinement can. In the application time-lines we only show a detail of the whole execution where these clusters appear, using the grey colour to mark those CPU bursts not interesting in the study.

Figure 6.8 contain the results obtained using DBSCAN with $Eps = 0.0470$ and $MinPoints = 4$. The scatter plot 6.8a present three different clusters. Cluster 3 (red) appears at bottom, and merges different clouds within the same range of instructions. Cluster 4 (green) and Cluster 6 (purple) also appear in the same range of instructions, but have not been merged. Looking at the time-line 6.8b, we see that Cluster 3 detects a clear SPMD phase. Clusters 4 and 6 are two parts of the same phase, hiding SPMD pattern.

In a configuration of DBSCAN we use the parameter $Eps = 0.0896$, so as to merge these two clusters, 4 and 6, that represent the same SPMD phase. Figure 6.9 presents the results of this experiment. As can be seen in plot 6.9a, instead of the desired merges, DBSCAN merged Clusters 3 and 4 of the previous experiment into Cluster 1 (light green), while Cluster 6 of previous experiment, now is detected as Cluster 2 (yellow). The time-line 6.9b reflects that



(a) Scatter-plot detail



(b) Time-line zoom

Figure 6.9.: Detailed results of a DBSCAN cluster analysis of WRF application. The parameters used were $MinPoints = 4$ and $Eps = 0.0896$

the two different SPMD phases detected in previous experiment now are just detected as a single one, Cluster 1, and Cluster 2 groups the tasks that disturb the SPMD pattern.

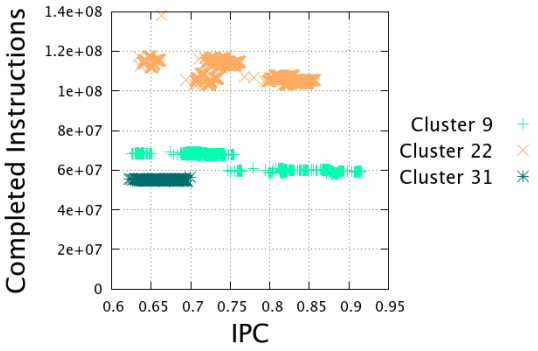
Using the Aggregative Cluster Refinement algorithm, we obtain the results present in Figure 6.10. The scatter plot 6.10a shows that the algorithm detected three clusters: 9 (pale orange), 22 (turquoise) and 31 (dark green) [1]. We can see that this algorithm was able to merge the the clusters where DBSCAN failed. But it is more remarkable that it was also capable to separate in two different clusters the individuals in the bottom cloud. Checking the time-line 6.10b we can see that the three clusters represent three different SPMD phases.

This detection could have never been possible using DBSCAN. As we have seen using a small Eps value, DBSCAN separated in different clusters sub-SPMD structures. On the other hand, using a higher Eps, DBSCAN merged clusters that represent the SPMD regions, but was still unable to merge a cluster that not reflects the SPMD pattern.

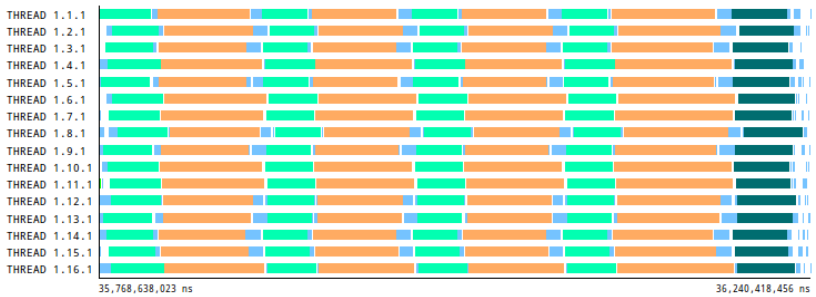
6.3.2. Study of the refinement tree

Even the Aggregative Cluster Refinement algorithm was design to overcome the DBSCAN limitations, the refinement tree it produces offers us an interesting information hint to un-

[1] We did not change the cluster numbering, for this reason we present these high cluster identifiers



(a) Scatter-plot detail



(b) Time-line zoom

Figure 6.10.: Detailed results of the Cluster Aggregative Refinement of the same phases of WRF application presented in Figures 6.9 and 6.8

derstand how the different clusters it generates are formed. In this section we study the three different cluster formations we observed the most in our experiments with the algorithm.

For the present example we applied the Aggregative Cluster Refinement, using $N = 10$ to the CPU bursts data extracted from an execution of the Versatile Advection Code (VAC) [103] using 128 tasks. VAC is a hydrodynamic and magneto-hydrodynamic simulation software used in astrophysics.

We illustrate each formation pattern by observing the structure of one of the clusters present in the final partition. For each cluster we present the following information:

- The section of the refinement tree that contains the formation of the cluster we want to study. In other word, the *sub-tree* of the refinement tree where the given cluster is the root. In the examples we usually refer to this sub-tree simply as tree.
- The scatter plots of intermediate partitions of the algorithm showing of the different sub-clusters the cluster studied contains. These intermediate partitions are selected by observing the clusters' relationships expressed in the refinement tree at the different levels.
- The time-lines of the clusters distribution corresponding to the intermediate partitions previously selected.

Comparing this three different outputs, we can stablish a relationship between the formation pattern of the given cluster and the internals of the SPMD phase it represents.

Formation pattern 1: clusters refined from noise

This cluster formation pattern is the most observed in the different analyses we performed. It consists of a cluster that appears in the initial iterations of the algorithm and absorbs individuals previously classified as iterations proceed.

In Figure 6.11 we can observe the tree corresponding to the formation of Cluster 1 of VAC. It is an small tree, as it only presents two iterations of the algorithm. In first iteration, Cluster 1 got an score of 99.3%. It required an small increase of the Eps so as to be perfectly detected in the second iteration.

If we check the plots of these two iterations, Figure 6.12, we can observe that using the small Eps generated by the Aggregative Cluster Refinement, plot 6.12a, the algorithm detected almost the whole cloud as Cluster 1, except for a small region at top right, which was detected as noise. At second iteration, using a slightly higher Eps, these few individuals previously detected as noise now where absorbed this cluster, as can be seen in plot 6.12b.

Figure 6.13 contain the time-lines of the SPMD region detected by Cluster 1 at this two iteration, showing one repetition of this phase. We can observe that in the first iteration, time-line 6.13a, the SPMD regions was detected nearly perfect, and just two of the tasks where detected as noise. In other repetitions of this SPMD phase we observed similar situations. In time-line 6.13b, we observe that the SPMD region is perfectly detected.

With respect to the SPMD structure detected, this formation pattern indicates that the SPMD region presented a strong core behaviour, i.e. high number of CPU bursts close together detected in the initial iteration, while some burst it contains present some variability

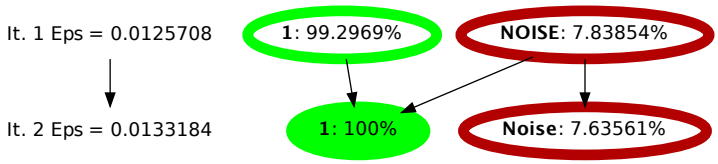
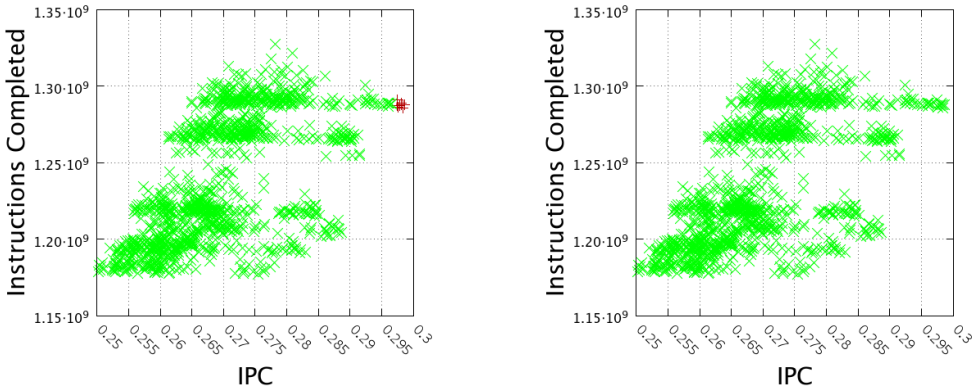
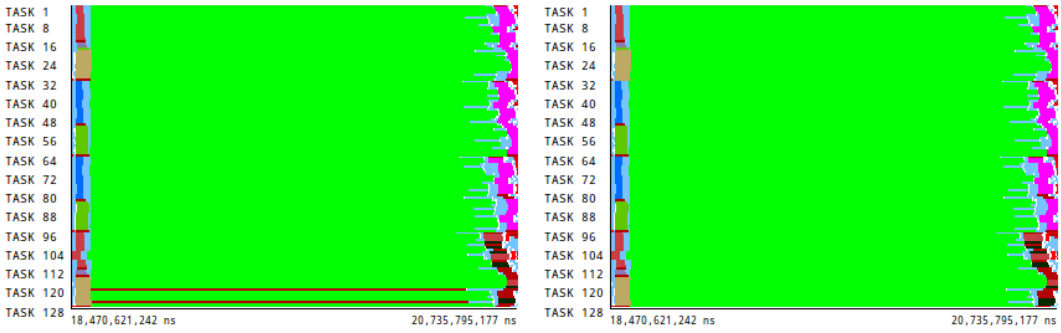


Figure 6.11.: Refinement tree depicting the formation pattern of Cluster 1 of VAC application



(a) First iteration (b) Iteration 2

Figure 6.12.: VAC Cluster 1 formation scatter plots. This plots correspond to the two iterations of the refinement tree of Figure 6.11.



(a) First iteration (b) Iteration 2

Figure 6.13.: Cluster 1 distribution time-line of iterations iterations presented in Figure 6.11. The time-lines contain one repetition of the SPMD region detected

with respect to this core. The number of bursts that present this variability is not high enough or close enough to consider them as clusters.

Formation pattern 2: clusters merges

This is the most interesting formation of clusters when using the Cluster Aggregative Refinement, corresponding to those clusters that appear as a merge of two or more cluster at previous levels. The merge of clusters reflects that the resulting computation region shows some dispersion along the bursts it contains. This dispersion is captured in the internal sub-clusters with less variability.

In Figure 6.14 we can observe the tree corresponding to the formation of Cluster 4 of VAC. This tree contains ten iterations the algorithm executed, as the formation of the cluster required the highest Eps value to be completely merged, to obtain a perfect cluster score. In the tree we can see that in the first iteration, when using the lowest value of Eps, the individuals detected as Cluster 4 were separated in up to 6 clusters (4, 6, 7, 8, 9 and 10) and also noise points. Next, at further iterations/different levels of tree, this initial clusters merged so as to obtain the definitive one. For example, in level 4, Clusters 8 (light orange) and 9 (electric blue) merged. Is it interesting to note that the cluster numbering is generated at first iteration according to the amount of computation time each cluster aggregates, as in previous chapters, lower numbers indicates more aggregated computation time. In further levels, the clusters resulting from merges take the number of the parent with lower number.

Figure 6.15 presents the scatter plots of intermediate partitions of the data obtained at first level, initial Eps, and those iterations where the cluster algorithm merged two or more clusters, detected by observing the refinement tree. This plots show how the initial 6 clusters mentioned, plot 6.15b, starts merging depending on their distance and density, as the algorithm increases the Eps, reflecting the hierarchical formation we mentioned in point 6.2.1.

Finally, Figure 6.16 we can see the clusters obtained from the partitions presented in the previous plots conform internal sections of a SPMD region. For example, in the first iteration, time-line 6.16a, it is clear that this sections correspond to ranges of consecutive tasks that present some variability on its duration inside the final SPMD phase represented by Cluster 4.

In contrast to the previous formation pattern, in this case, the number of CPU bursts that present the variability is high enough to be detected independently, pointing that they could be taken into account for further analysis. For example, the SPMD region represented by Cluster 4 has a temporal imbalance, as can be seen in the time-lines of Figure 6.16. Thanks to the plots of Figure 6.15, the imbalance of each of the groups can be quantified in terms of instructions and IPC. This is a valuable information to perform a deeper analysis to explain what are the causes of the imbalance observed and how can be solved.

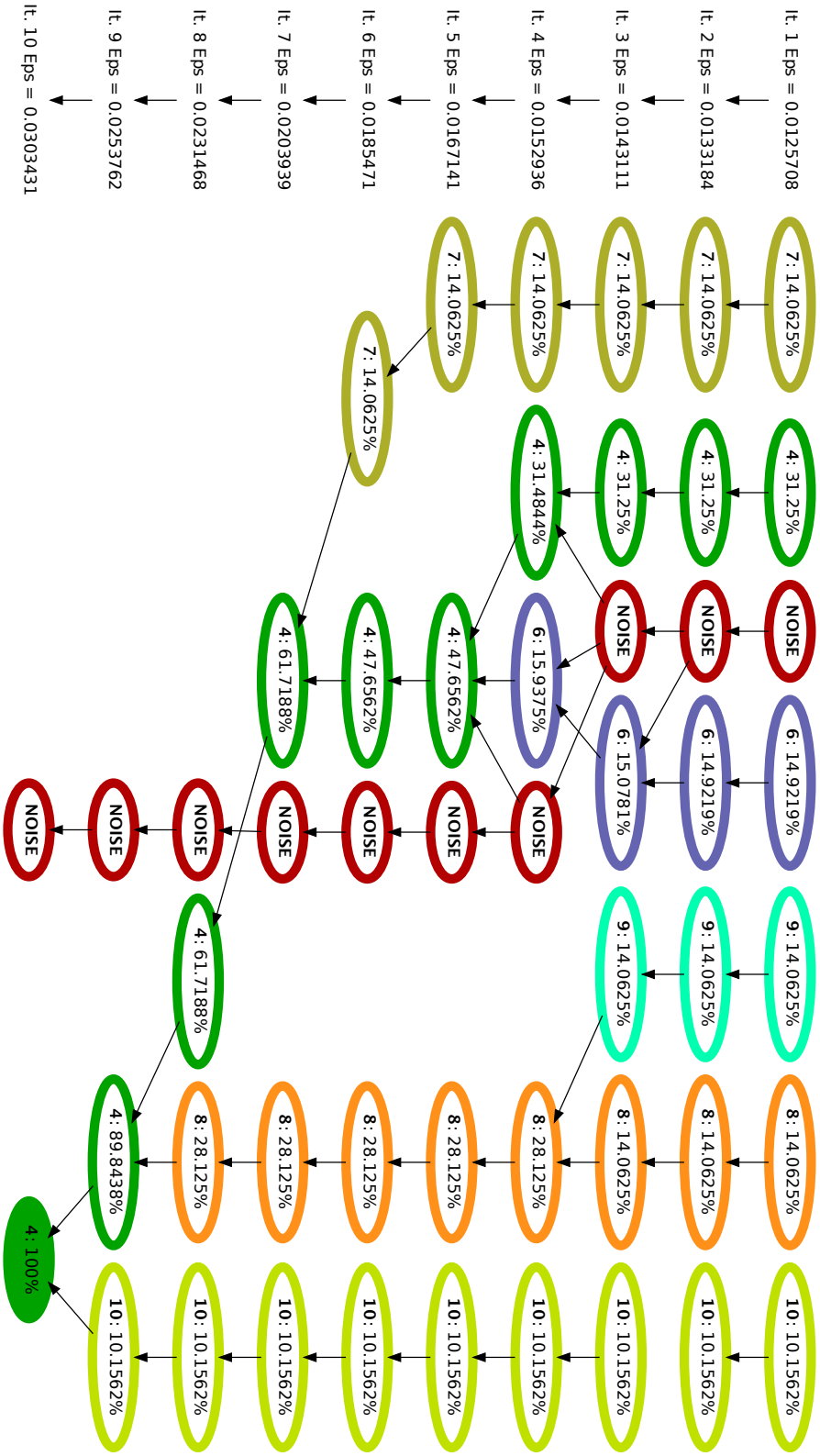
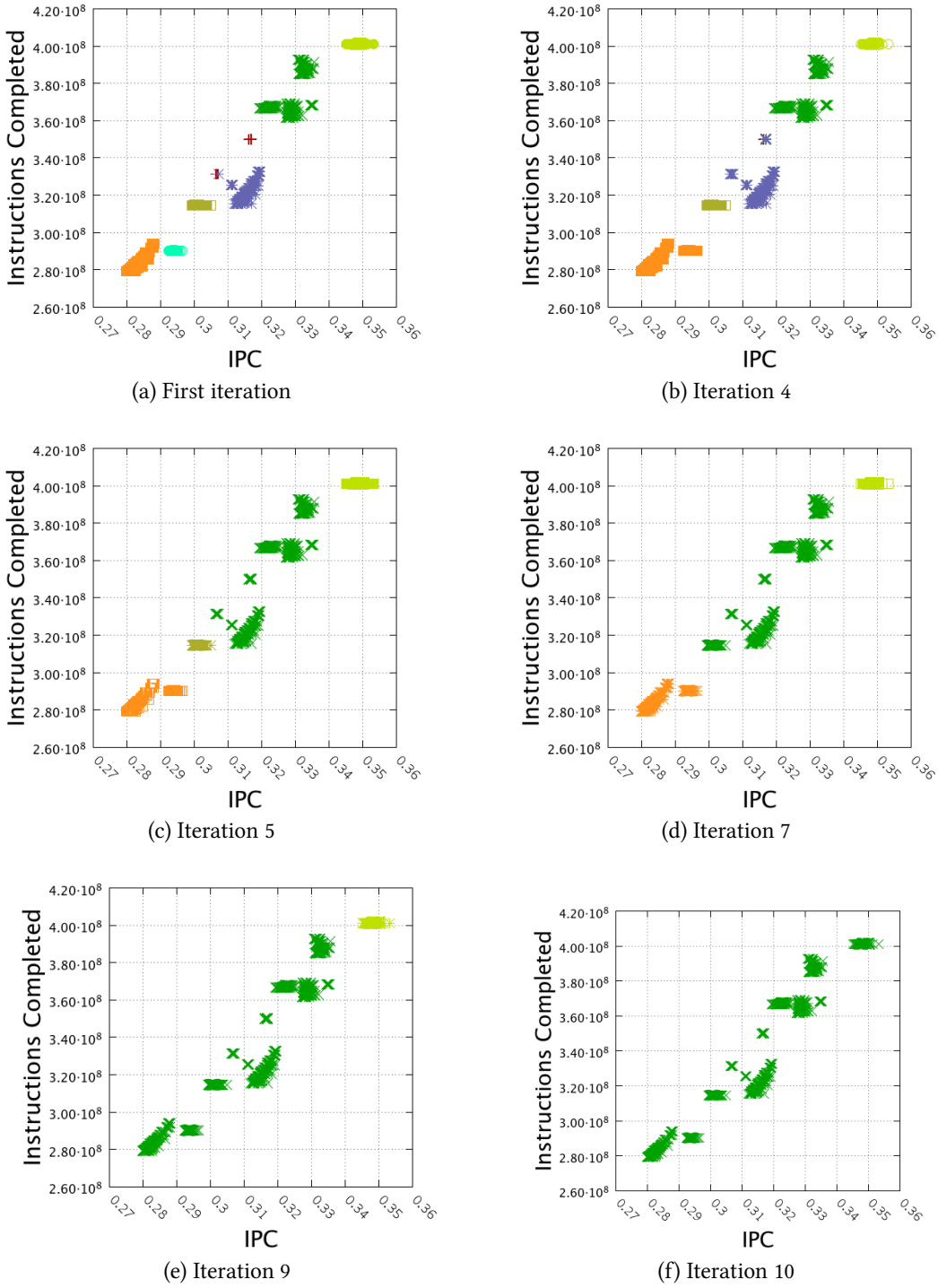


Figure 6.14.: Refinement tree depicting the formation pattern of Cluster 4 of VAC application



(a) First iteration

(b) Iteration 4

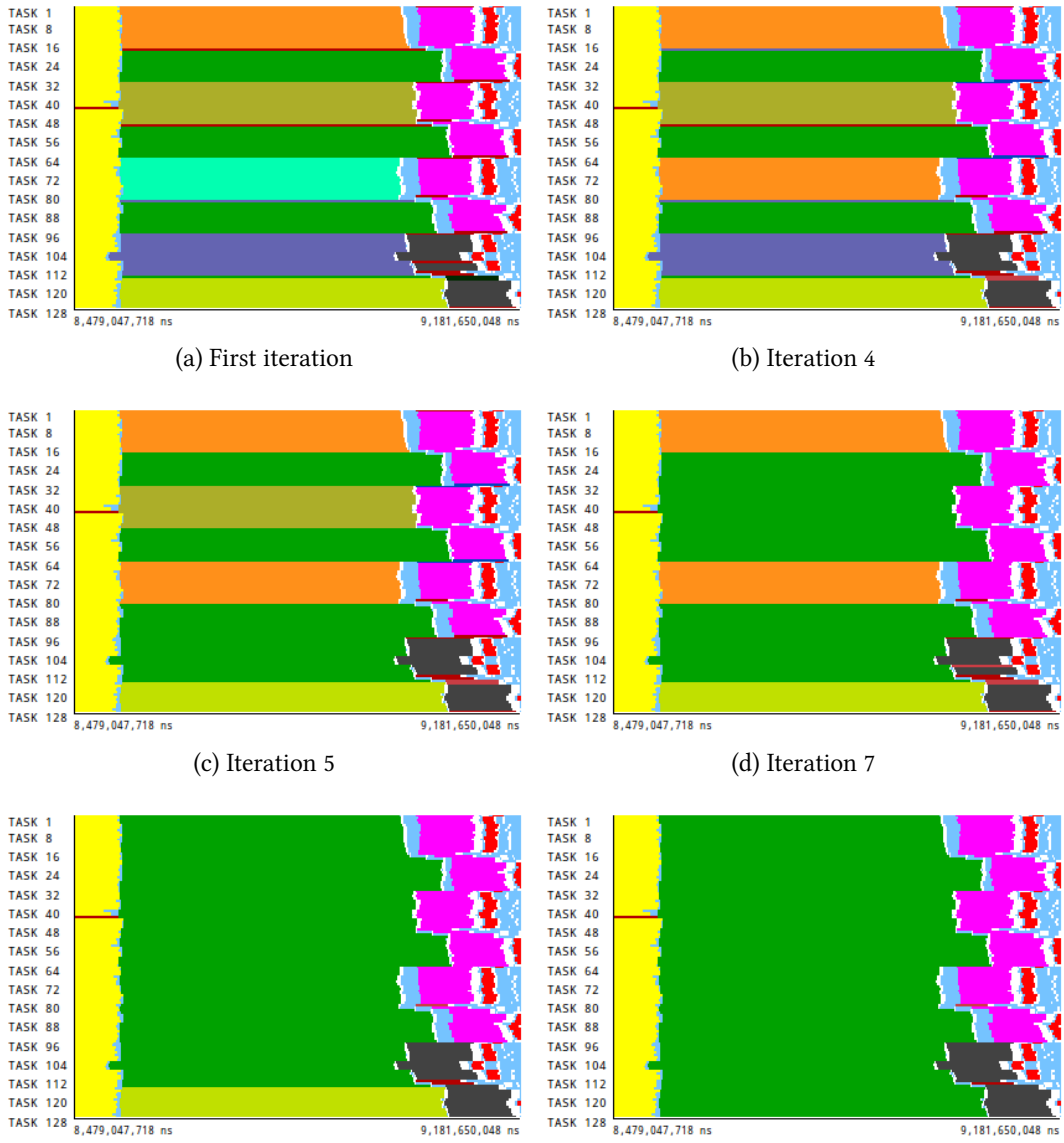
(c) Iteration 5

(d) Iteration 7

(e) Iteration 9

(f) Iteration 10

Figure 6.15.: VAC Cluster 4 formation scatter plots. These plots correspond to the the first iteration of the algorithm and iterations observed in the refinement tree of Figure 6.14, where the Aggregative Cluster Algorithm merged two or more clusters.



(a) First iteration

(b) Iteration 4

(c) Iteration 5

(d) Iteration 7

(e) Iteration 9

(f) Iteration 10

Figure 6.16.: Cluster 4 distribution time-line of iterations presented in Figure 6.15. The time-lines contain one repetition of the SPMD region detected

Formation pattern 3: sequence based merge of clusters

The third formation pattern is an special case of the cluster merge. It illustrates case illustrate the effect of the *sequence based merge* of clusters that acts as a post-process of the algorithm, the routine named `ProcessLastPartition` in the Algorithm pseudo-code. This process merges those clusters that are not close enough in the data space, but represent substructures into a single SPMD phase.

The sequence based merge uses the aligned sequences of clusters produced by the use of the Cluster Sequence Score criterion in the last step of the algorithm. Traversing the these sequences, it detects the clusters that appear simultaneously along the execution. Those groups of clusters that *always* appear simultaneously are merged, independently from its geometrical description.

An example of this formation pattern is illustrated by the refinement tree of Figure 6.17. We see in this tree that Cluster 5 appears in the final level by the merge of two different clusters of level tenth, 5 and 11, where the highest Eps was used.

We can also observe in the refinement tree that in previous level, the *definitive* Cluster 5 is also decomposed in a series of sub-cluster, but for this example we want two focus of the last two levels of the tree. The scatter plots of the partitions corresponding to these two levels are depicted in Figure 6.18. Scatter plot 6.18a shows that the algorithm did not merge Clusters 5 and 11 using the highest Eps due to the empty band, with null density, that separates both clouds. Thanks to the sequence based merge, we obtain the partition corresponding to the plot 6.18b where these two regions are merged.

Finally, Figure 6.19 shows that the two clusters that where separated by the empty band correspond to two groups of tasks, time-line 6.19a. These two groups present a difference in terms of duration, but are in fact two inner parts of the same SPMD phase, see time-line 6.19b.

In terms of analysis, this formation pattern differs from the previous in the strong variability across the metrics to apply the cluster analysis. So the sub-SPMD structures they represent present a higher imbalabce.

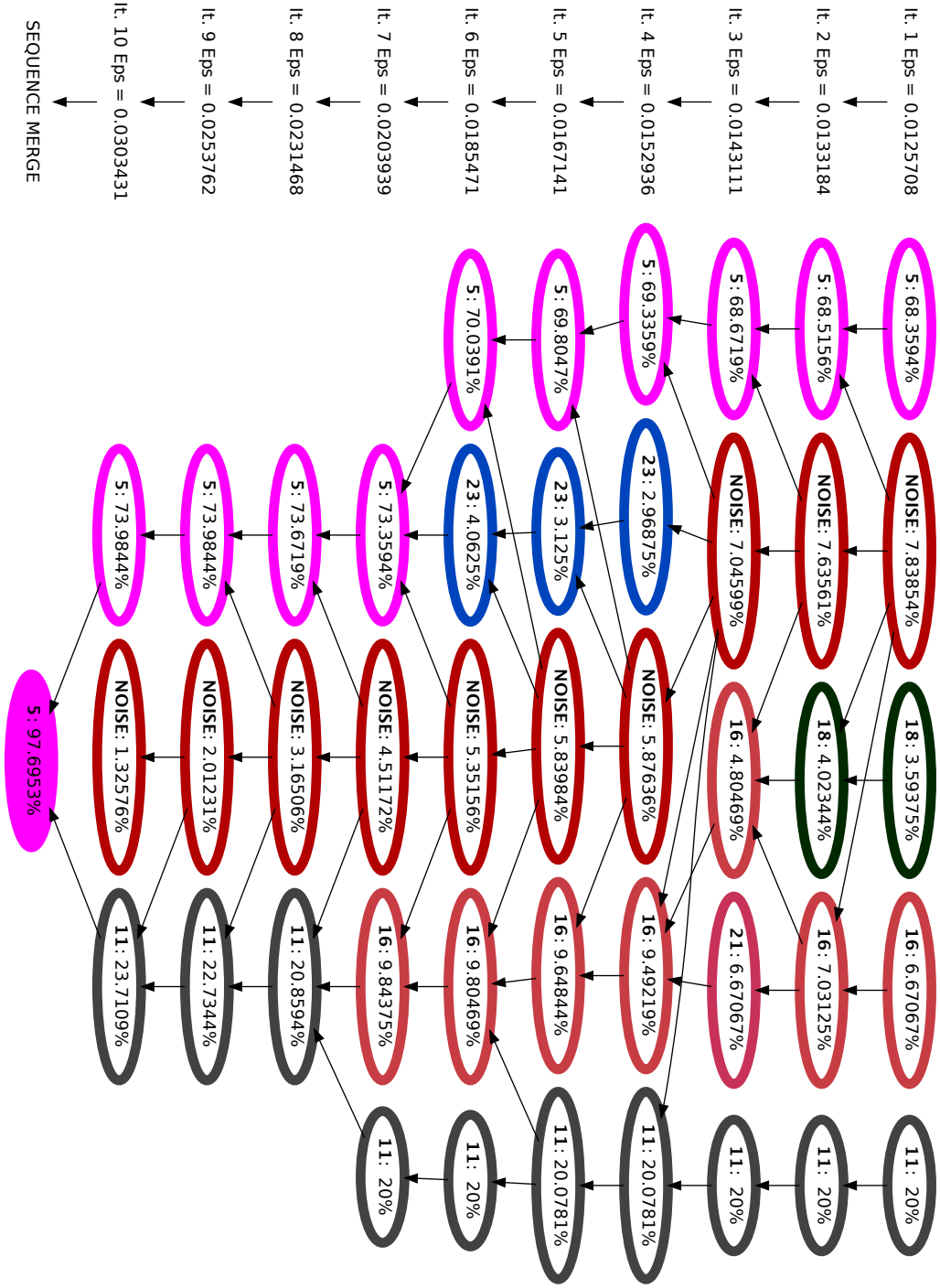
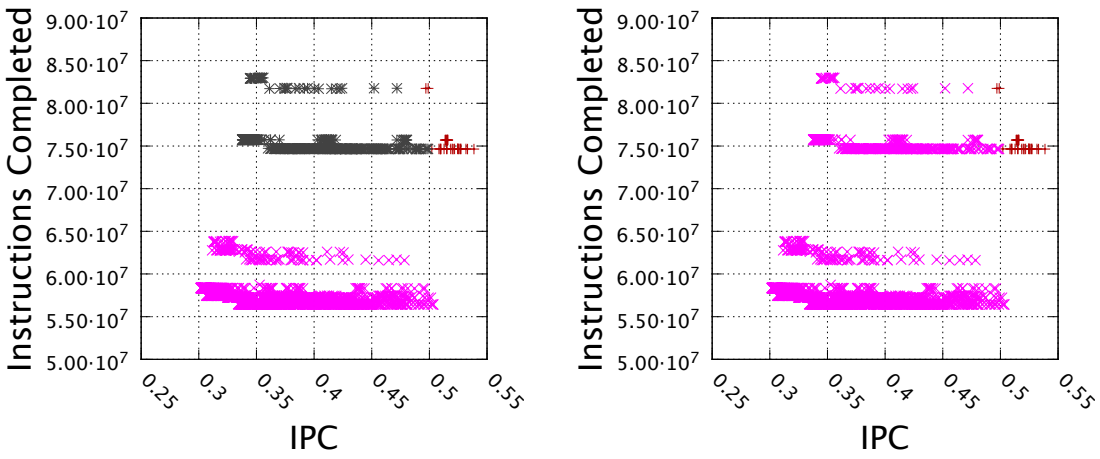


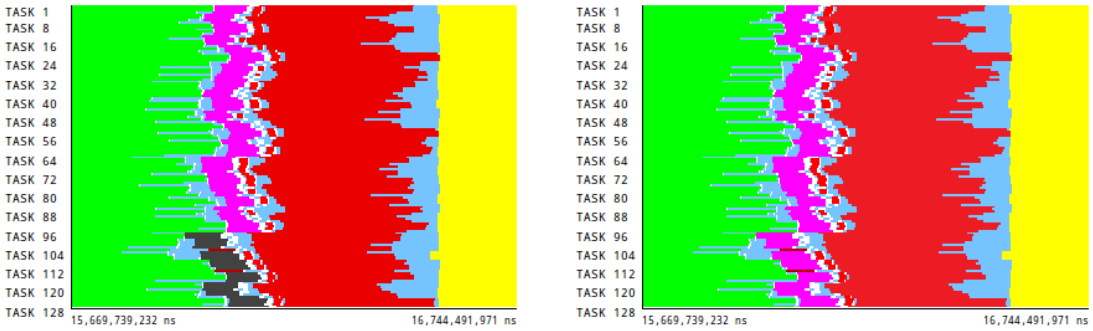
Figure 6.17.: Refinement tree depicting the formation pattern of Cluster 5 of VAC application



(a) Last iteration (10)

(b) Sequence based merge

Figure 6.18.: VAC Cluster 4 formation scatter plots. These plots correspond to the the two bottom levels of the refinement tree of Figure 6.17.



(a) Last iteration (10)

(b) Sequence based marge

Figure 6.19.: Cluster 5 distribution time-line of partitions presented in Figure 6.18. The time-lines contain one repetition of the SPMD region detected.

Part III.

Practical Uses

7. Performance Data Extrapolation

In some cases, when analysing an application the availability of the performance data is limited. The case of performance hardware counters is paradigmatic: as they are part of the hardware resources usually its availability is restricted by design so as not to occupy or interfere with regular purpose of the hardware element they account.

In this chapter we present a methodology to maximize the performance information obtained from a single run of a parallel application. It relies in the structural detection performed by the cluster analysis so as to extrapolate the average value of a performance hardware counters set, larger than the offered by the processor.

7.1. Performance Data Extraction Limits

It is obvious to say that more performance issues can be detected as more information is made available. For example, if we want to know the memory behaviour of our application, we will require information regarding memory accesses or cache hits and misses, meanwhile if we target on how the computational part of application behaves, data regarding to fixed point and floating point operations will be essential.

Hence performance hardware counters have become an invaluable aid to performance analysis. However, as pointed by Sprunt in [104], there are always limitations on the total amount of counters that can be read at the same time and the combinations of counters available. For example, current processors range the available registers used to performance accounting from 4 to 8, and the way to access them are in prefixed groups of counters, defined at processor design.

7.2. Extrapolation Methodology

The performance data extrapolation process consists of determining the average value of a large set of hardware counter groups for each structural region of the application. The definitions of the structural regions can be done by multiple methods, but in the scenario of this thesis we use the SPMD phases obtained by using cluster analysis. It is an extrapolation because the number of groups computed is larger than the number of registers available in the processor that can be read simultaneously. In addition, the resulting set can contain two mutually exclusive counters, coming from combinations fixed by the processor design.

7.2.1. Performance hardware counters multiplex

The basic idea of this method is to multiplex of the hardware counters groups read during the application execution to account them in the different structural regions defined.

This approach has been widely used in the literature such as by the works by May [105] or the `hpmcount` tool [106] by IBM. In these two works, the results are a linear projection of the multiplexed counters values for the whole (sequential) application execution. Further works such as [107, 108], proposed different projection techniques so as to capture the time-varying behaviour of the counters. The granularity of projected values in these two works was defined by the sampling frequency used to read the counters. This fact provoked relatively high errors in some extrapolations.

Our methodology differs from [105] and [106] as it characterize much more finer regions of the application, the SPMD regions. On the other hand, our target is totally different to the works presented in [107] and [108], as we do not study the time-varying behaviour of the counters, but the behaviour of the different SPMD phases in terms of these metrics.

In addition, we can take advantage of the fact that we analyse parallel applications, in order to tune how we multiplex the data acquisition. In this way, we propose three different multiplexing strategies of the hardware counter groups: first, space multiplexing that consists of assigning a different hardware counters group to each task of the parallel application; second, time multiplexing, where all tasks read the same group of counters at the same time, but the group changes with a user defined frequency; and third, time-space multiplexing, a combination of both.

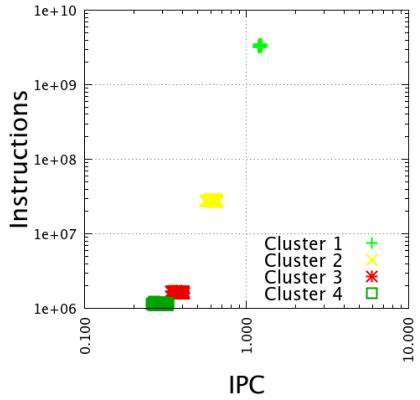
7.2.2. Extrapolation Steps

In brief the extrapolation methodology acts in three steps: a data acquisition where different counters groups are multiplexed; a process to define the different structural regions, the cluster analysis in our case; and finally a projection of the values to describe each region.

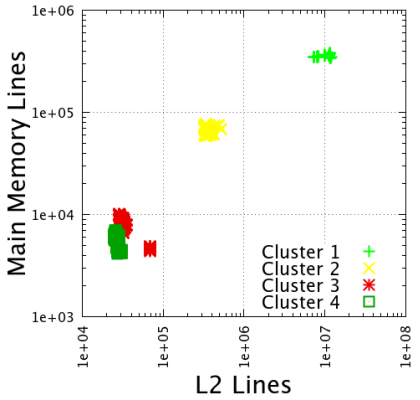
Data Extraction

First decision before executing the application and extracting the data is which groups of counters do we need. Even more important than the selection, is to ensure that all groups have a common subset of counters so as to guarantee that the cluster analysis could be computed for all points extracted. This fact depends on the processor design, but all major processor vendors today include Completed Instructions and Cycles as fixed values in all possible combinations of counters in the available sets.

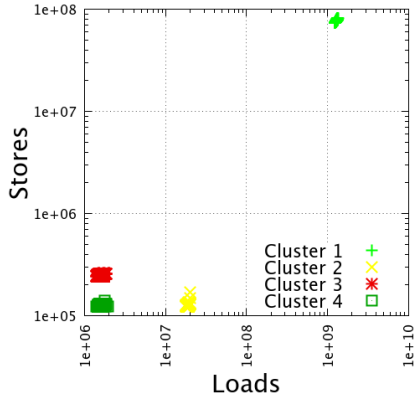
In order to have measures of all groups for all application phases or regions, then we multiplex the read of all groups during the application execution. We already mentioned that we can make use of three different multiplexing schemes: time multiplexing, space multiplexing and time-space multiplexing.



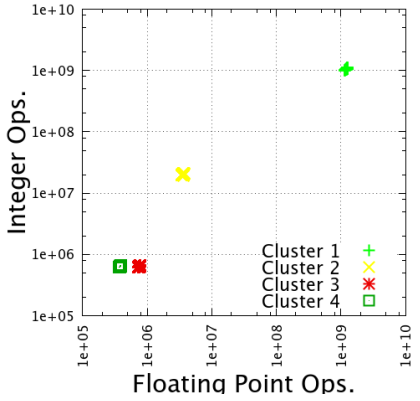
(a) Instructions vs. IPC



(b) Main memory accesses vs. L2 data cache acessess



(c) Stores vs. Loads



(d) Integer instructions vs. Floating point instructions

Figure 7.1.: Example of a clustering of GAPgeofem application using the Aggregative Cluster Refinement with Instructions Completed and IPC. Upper left plot depicts the metrics used by the clustering algorithm. The rest of plots show the clusters found in terms of other pairs of metrics not used during the cluster analysis

Cluster Analysis

Having read the desired data, then we apply the Aggregative Cluster Refinement to the common metrics that appear in all groups. As demonstrated, Completed Instructions and IPC produce a good computation characterization, but the most interesting fact is the representativity of these counters. In other words, the clusters found using these two metrics detect groups present when using other metrics. In Figure 7.1 we can see the results of the Aggregative Cluster Refinement to GAPgeofem application using the mentioned combination of metrics, plot 7.1a. Rest of the plots depict show the groups found using other metrics combiations to represent the CPU bursts. Except for the combination of Main Memory Lines and L2 Lines, plot 7.1b, where Cluster 3 (in red) got divided, the clusters found using Completed Instructions and IPC represent isolated groups in the different metrics.

After applying the cluster analysis, we obtain is a set of clusters that represent different

SPMD regions of the application. On each cluster there must be points that include information from the different counters groups multiplexed.

To guarantee this, 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 when using time multiplexing. In SPMD codes the space multiplexing approach can avoid running the application for a long time, but it is most advisable to use the combination of both. Space multiplexing is able to capture the variances of the intra-nodes, and time multiplexing focuses on the variances across nodes, hence the combination of both results in a better way to guarantee that these variances are considered when extrapolating the counters values.

Data Projection

Finally, once we have the points divided into clusters, the final step consists of computing the average of all counters. That is just a weighted average of each counter for each cluster. In this way, each point contributes its non common counters to the characterization to such cluster.

Obviously, for a given cluster, the average of some of the metrics will be computed using less individuals than others. For example, the average value of Completed Instructions and Total Cycles will be computed using all individuals on each, as these counters appear in all groups, not the case for thosethose counters that appear in few groups. While it will be reasonable to study the potential error these metrics computed with less individuals suffer, the measurements we present in section 7.3 suggest the errors observed are related to those metrics that account events that occur few times.

7.3. Validation

The perform the validation of the extrapolation methodology presented, we followed these steps:

1. First, we defined a set of hardware counters that are present in different counters groups.

2. We ran each application analysed three times, using the three different multiplexing of the counters groups required to gather the data.

3. We applied the extrapolation methodology to data obtained using the three different multiplexing schemes. At this point, we have three different *projected* or *extrapolated* values for each counter and cluster detected.

4. Next, we run the application as many times as counters groups we require, gathering the information of a single group on each execution, without using multiplexing.

5. We applied a cluster analysis to each *non-multiplexed* execution and computed the *actual* values of each metric and cluster.

6. Finally, for each metric and cluster, we compared the three extrapolated values obtained applying the extrapolation methodology to actual values.

To perform this comparison, it strictly required that all clusters represent the same regions across all executions of the application we use to validate the technique. To assess this requirement we compared manually the plots and application time-lines on each application run.

To quantify the difference between actual and extrapolated values on each case, we use a weighted error metric described below.

7.3.1. Experiments data

All the experiments presented here were carried on in an IBM JS21 cluster, using PowerPC 970MP processors. Table 7.1 contains the list of all counters we used as well as the description of them provided by PAPI command `papi_native_avail` and the group defined by the vendor where each cluster is available. We select these counters because they can be used later to build a CPI breakdown model described in [109, 110]. As a result, we obtained a set of 6 different counters groups. In this way, we need to run each application up to 9 times to perform the comparisons: three using the different multiplexing schemes and 6 more to obtain the required counters groups without multiplexing them.

We used four applications to perform the validation of the extrapolation technique. First one, PEPC (described in chapter 9) was executed using 32 tasks. The rest of applications were selected from the SPECMPI2007 benchmark suite: TERA_TF, a 3D eulerian hydrodynamics application; GAPgeofem, a finite-element for thermal conduction; and ZEUS-MP/2, an astrophysical fluid dynamics simulator. These three applications were executed using 16 tasks.

In all cases, we analysed the group clusters that aggregate more than 90% of the application total computation time.

7.3.2. Weighted error

We observed an interesting fact that occurred while performing the validation experiments: using the classical relative error metric, those clusters that accounted events that appear in very few occasions produced the highest error values. For example, the counter that measures the processor cycles stall due to an instruction cache miss, a uncommon situation in the applications we analysed, produced high relative errors the between actual and the projected values. That was caused by the small absolute figures these counters reach, where an small variation of the projected value produced a big relative error.

So as to avoid this misleading information the relative error metric could provide, we propose a weighted error metric to validate the extrapolation technique. This error measures the differences between the actual and extrapolated value of a counter in a given cluster, taking into account the counter *relevance* in such cluster.

We define the relevance of a counter in a cluster as its relative value with respect to actual value of total cycles counter or completed instructions counter in such cluster. Using total cycles or completed instructions depends of the semantics of the counter itself: for example,

#	Counter Name	Description	Groups
1	PM_CYC	Processor cycles	All
2	PM_GRP_CMPL	A group completed. Microcoded instructions that span multiple groups will generate this event once per group	1
3	PM_GCT_EMPTY_CYC	The Global Completion Table is completely empty	1, 2
4	PM_GCT_EMPTY_IC_MISS	GCT empty due to I cache miss	2
5	PM_GCT_EMPTY_BR_MPRED	GCT empty due to branch mispredict	2
6	PM_CMPLU_STALL_LSU	Completion stall caused by LSU instruction	3
7	PM_CMPLU_STALL_REJECT	Completion stall caused by reject	4
8	PM_CMPLU_STALL_ERAT_MISS	Completion stall caused by ERAT miss	3
9	PM_CMPLU_STALL_DCACHE_MISS	Completion stall caused by D cache miss	4
10	PM_CMPLU_STALL_FXU	Completion stall caused by FXU instruction	5
11	PM_CMPLU_STALL_DIV	Completion stall caused by DIV instruction	5
12	PM_CMPLU_STALL_FPU	Completion stall caused by FPU instruction	6
13	PM_CMPLU_STALL_FDIV	Completion stall caused by FDIV or FQRT instruction	6
14	PM_CMPLU_STALL_OTHER	Completion stall caused by other reason	4
15	PM_INST_CMPL	Number of Eligible Instructions that completed	All

Table 7.1.: List of all hardware counters used in the experiments to verify the extrapolation technique

the relevance of the counters that measure processor stall cycles caused by a given reason is computed using total cycles, while the relevance of the counters that measure the different types of instructions executed is computed using the completed instructions.

As a summary, the weighted error we define focus the evaluation of those counter that obtain higher values, according to its relevance. We consider these counters more interesting from the analyst point of view. For this reason, we use the weighted error metric more appropriate to quantify the errors in the projection methodology.

In the experiments, with the exception of completed instructions, `PM_INST_CMPL`, all the hardware counters we used (Table 7.1) relate to processor cycles stalled caused by different hardware elements. In this way, the counter relevance was computed by using the total cycles counter value. In the case of completed instructions, we set its relevance to 100% (relevance over itself). Obviously, total cycles also has a relevance of 100%.

7.3.3. Validation Results

In the first experiments of the results we illustrate the issues described when using the relative error metric. In Figure 7.2 we present the relative errors, the charts on left column, and weighted errors, charts on right column, for each metric and cluster using the three extrapolation schemes for PEPC application. In these plots, the X axis represent the different counters listed in table 7.1, while the Y axis is the respective error. In Figure 7.3 we present the relevance of each counter and cluster with respect to cycles.

If we look at the charts containing the relative errors, 7.2a, 7.2c and 7.2e, we can see that results for the vast majority of counters are low. Contrarily, few of them obtained high errors. For example, the processor cycle stalls due to integer divisions (11) obtained a relative error of 45% in Cluster 4 when using the space multiplexing scheme (chart 7.2a and the processor cycle stalls due to floating point divisions (13) had a relative error of -42% in the same cluster when using the time multiplexing scheme (chart 7.2c).

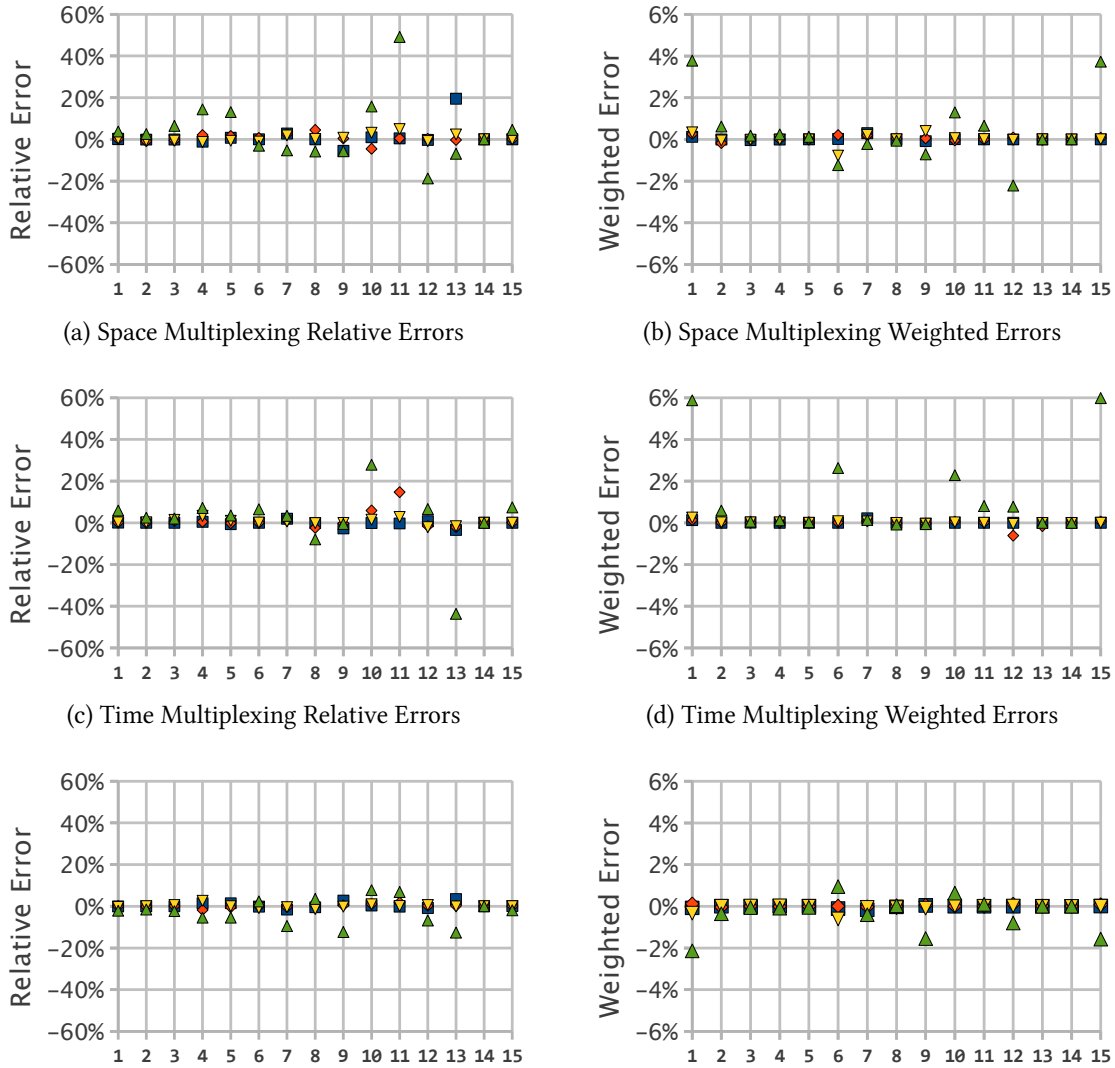
Now, if we observe the relevance of these counters for the mentioned cluster in Figure 7.3, what we can detect that their value is nearly 0%. In this way, the relative error does not reflect an actual issue with the extrapolated value, as their relevance is negligible. This observation is applicable to the rest performance counters available that present high relative errors.

Using the weighted error metric, depicted in charts 7.2b, 7.2d and 7.2f we totally avoid the biased evaluation the relative error provides. The charts show that the performance extrapolation methodology is able to correctly extrapolate the different values, as the weighted error is bounded between $\pm 6\%$

In Figures 7.4, 7.5 and 7.6 present the charts of the weighted errors using the different multiplexing schemes with the applications TERA_TF, GAPgeofem and ZEUS-MP2 respectively. As can be seen in these charts, the weighted errors are always bound between $\pm 5\%$, confirming the accuracy of the extrapolation technique we present.

7.3.4. Multiplexing scheme selection

In the previous section we have not evaluated the quality of the different multiplexing schemes used. Here we discuss which approach is results the most interesting in terms



(a) Space Multiplexing Relative Errors

(b) Space Multiplexing Weighted Errors

(c) Time Multiplexing Relative Errors

(d) Time Multiplexing Weighted Errors

(e) Time-Space Multiplexing Relative Errors

(f) Time-Space Multiplexing Weighted Errors

Figure 7.2.: PEPC extrapolation errors for CPI breakdown model counters using different multiplexing strategies. Left column represents the relative error comparing the extrapolation and the actual value from non-multiplexed execution. Right column represents these errors weighted in terms of the relevance of each cluster with respect to the total cycles on the non-multiplexed run

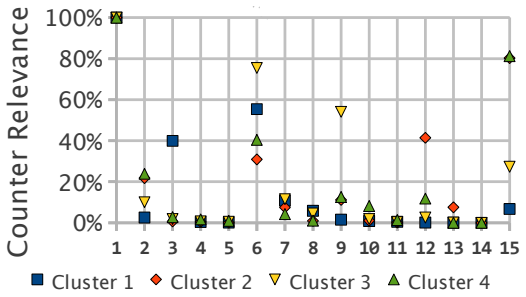


Figure 7.3.: PEPC counters relevance with respect to Total Cycles counter

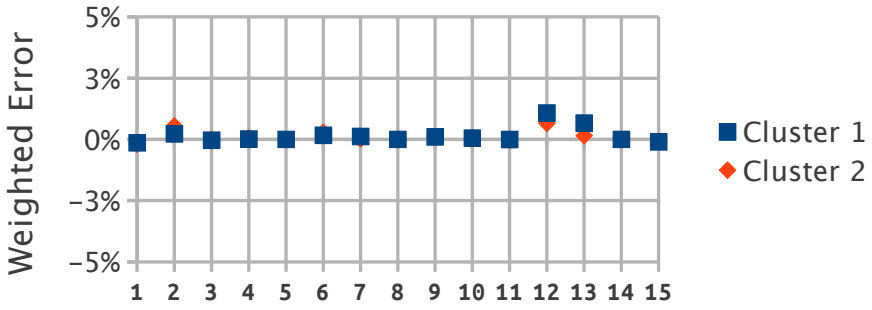
	PEPC	TERA_TF	GAPgeofem	ZEUS-MP2
Space multiplexing	0.14%	0.12%	0.32%	0.06%
Time multiplexing	0.35%	0.02%	-0.09%	0.01%
Time-space multiplexing	-0.11%	0.01%	0.19%	0.07%

Table 7.2.: Average value of weighted errors using different multiplexing schemes

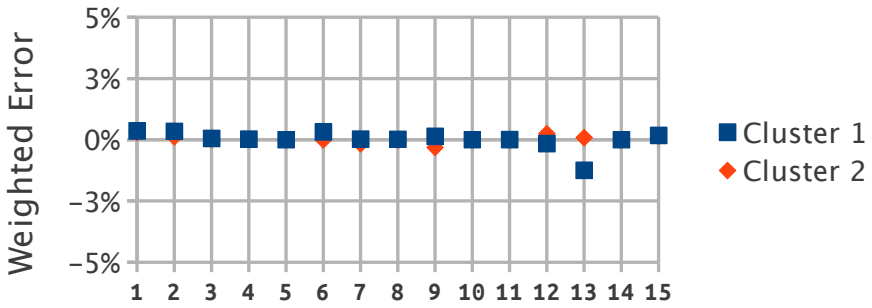
of the errors they produce.

If we look at the different plots of weighted errors presented, it is difficult to observe high variations between the errors produced by the different schemes metrics. To ease its comparison, we present the average value of the weighted errors for each counter and cluster in Table 7.2 obtained by using the different schemes.

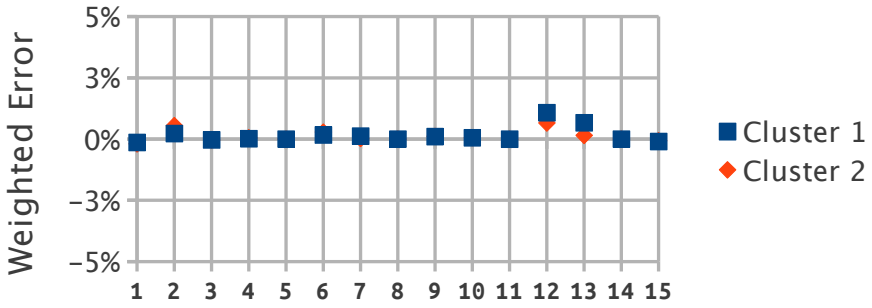
In this table, we observe that the schemes that produced the lowest errors where time multiplexing and time-space multiplexing, both in two cases. This points that the variations of the counters values are more likely to appear across the different temporal periodicities of the structural phases of the application than across the processors involved. Even the numbers present in Table 7.2 are not very conclusive, we suggest using the time-space multiplexing scheme. This multiplexing scheme is able to capture the counters variability in both dimensions.



(a) Space multiplexing weighted errors

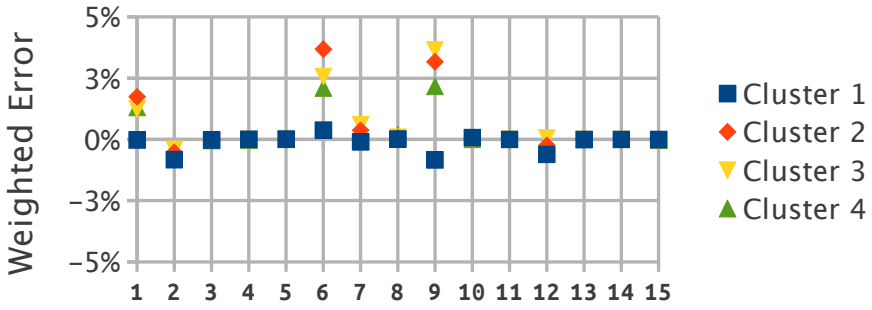


(b) Time multiplexing weighted errors

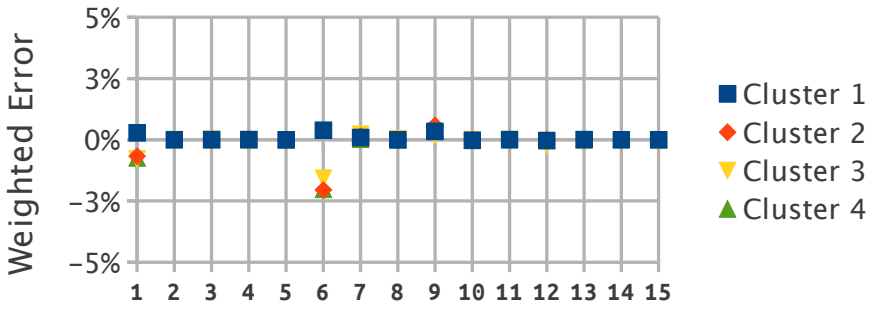


(c) Time-space multiplexing weighted errors

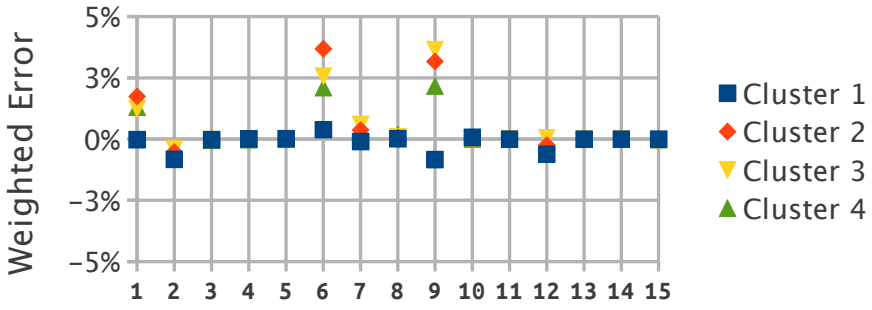
Figure 7.4.: TERA_TF extrapolation errors of counters listed in Table 7.1 model counters using different multiplexing schemes.



(a) Space multiplexing weighted errors

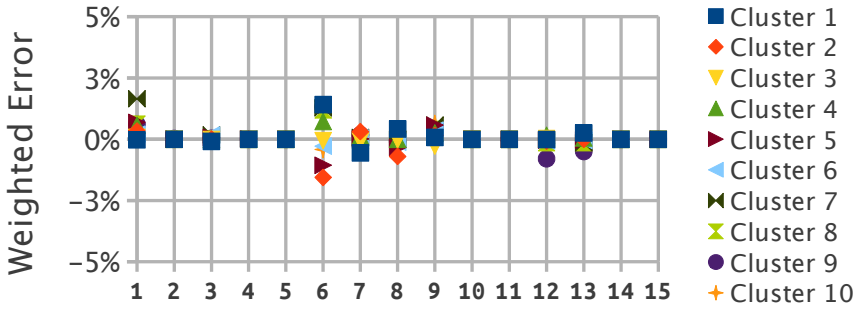


(b) Time multiplexing weighted errors

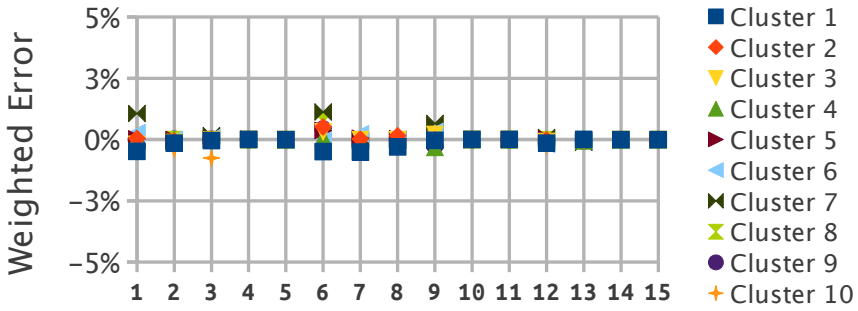


(c) Time-space multiplexing weighted errors

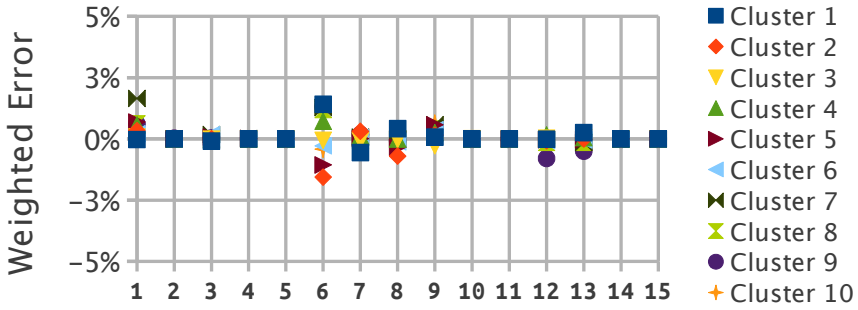
Figure 7.5.: GAPgeofem extrapolation errors of counters listed in Table 7.1 model counters using different multiplexing schemes.



(a) Space multiplexing weighted errors



(b) Time multiplexing weighted errors



(c) Time-space multiplexing weighted errors

Figure 7.6.: GAPgeofem extrapolation errors of counters listed in Table 7.1 model counters using different multiplexing schemes.

7.4. A Use Case: construction of CPU breakdown models per cluster

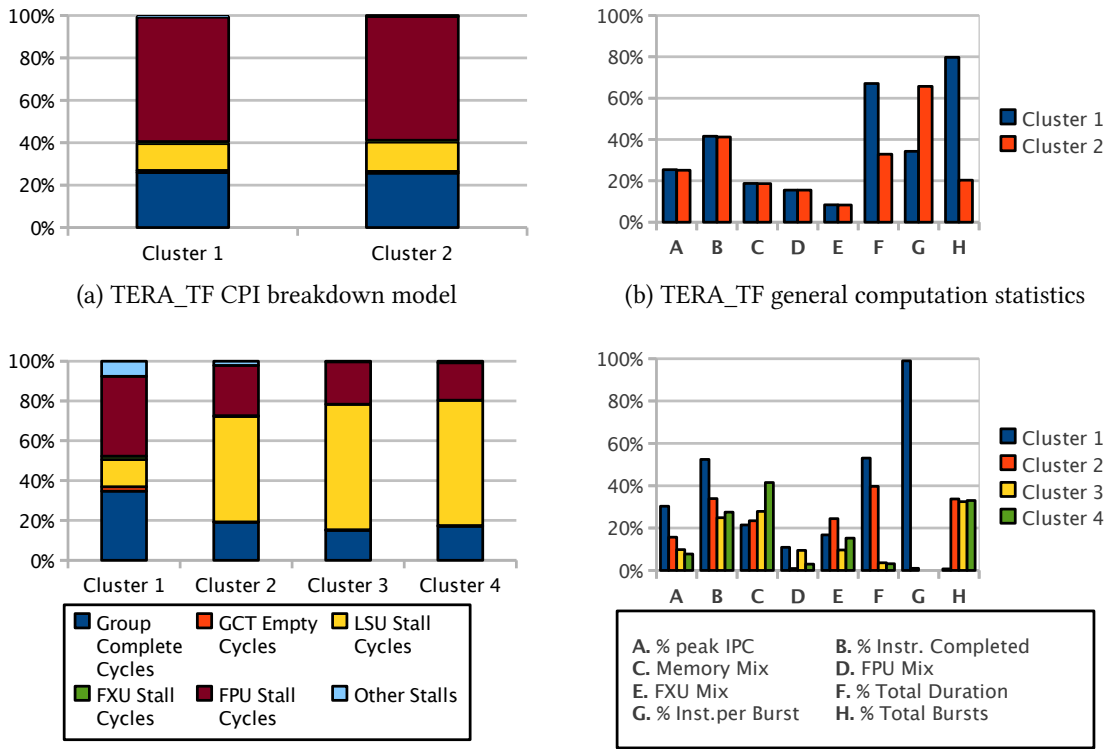
In Figure 7.7 we show an analysis model, left charts, and statistics characterization, right chart, of the major cluster of TERA_TF and GAPgeofem. The information contained on these charts has been extracted from a single run of each application.

The analysis model is a simplification of the CPI breakdown mentioned previously, using just five of the 15 counters extracted to improve its legibility. It shows the major stall causes for each cluster. In brief, 'Group Complete Cycles' refers to those cycles when the processor actually finishes instructions, 'GCT Empty Cycles' explains the stalled cycles due to re-order buffer issues, for example branch miss-predictions, 'LSU (Load-Store Unit) Stall Cycles' are those cycles stalled by memory issues, 'FXU Stall Cycles' and 'FPU Stall Cycles' are the cycles stalled waiting to integer and floating point computations respectively and, finally, 'Other Stalls' refers to stalls caused by situations not captured by the hardware counters.

The plots on the right column are a brief statistical description of some interesting computational characteristics, as the percentage of the peak IPC (considering 4 as the maximum IPC possible in a PowerPC 970), different instructions mix (percentage of total instructions that corresponds to memory 'Memory Mix', floating point 'FPU Mix' and integer instructions 'FXU Mix'), and also some numbers related to total duration, instructions per burst, and number of bursts per cluster.

A good approach to understand the figure is comparing both plots of each application. For example, in TERA_TF, both regions detected perform in the same way, having a IPC around the 22% of the peak (metric A on sub-figure 7.7b), which is actually a good value for this processor. In both cases, most of the stalls are caused by floating point operations (CPI breakdown 7.7a), but FPU Mix (metric D on chart 7.7b) is just around 16%. What is more, we can see that Memory Mix (metric D on chart 7.7b) presents the highest values, around 20%. These facts make that both clusters could be improved by trying to solve the dependencies of floating point data computations, and the access to the data structures that contain them. Finally, we can also see that the only difference between both regions is that the Cluster 1 has less instructions per burst, but it appears four times more often than Cluster 2 (metrics G and H on chart 7.7b)

Regarding GAPgeofem, we observe that Cluster 1 dominates the computing part, up to 50% of total time (metric F on chart 7.7d), having an IPC around 30% of the theoretical peak. It is interesting to note that the frequency of this cluster is nearly 100 times bigger than the rest of the clusters (metric G on chart 7.7d). In any case, this is a good situation, because this cluster groups big computation bursts, which obtained the best performance in the applications tested. So, in this case, developer/analyst should tackle the problems of Cluster 2, because the rest of clusters represent a small amount of total time (metric G on chart 7.7d). Cluster 2 is mainly dominated by memory stalls (LSU stalls on chart 7.7c). The main recommendation here should be to analyse the memory access patterns of this cluster.



(a) TERA_TF CPI breakdown model

(b) TERA_TF general computation statistics

(c) GAPgeofem CPI breakdown model

(d) GAPgeofem general computation statistics

Figure 7.7.: General CPI breakdown models of all applications presented in the paper. These models are a general view of the major categories, not using all 15 counters extrapolated to clarify its legibility. In all cases, they were computed the time-space multiplexing extrapolation method

8. Information Reduction for Multi-level Simulation

An important use of the structure detection based on clustering is to summarize the performance information generated be a parallel application. In this chapter we present methodology that uses this feature in a multi-level framework to simulate supercomputing applications. A combination of a coarse-grain characterization of the application phases using a spectral analysis plus the fine-grain characterization offered by cluster analysis is able to minimize the total amount of information used by a simulator, with no compromise of the simulation quality.

8.1. Scenario

Researchers have widely used simulation tools to verify, analyse, and improve computer systems. Simulation is used at different levels of detail, depending on the particular target system to study. The trade-off between simulation speed and accuracy is always present in these studies. In this area, no other current simulation infrastructure allows the simulation of large-scale computing systems such as supercomputers at the same level of detail that our methodology provides, without compromising the total simulation time.

Functional simulators emulate the target system's behaviour, including the operating system and the different system devices (such as memory, network interfaces, and disks). These simulators let designers verify system correctness and develop software before the system has been built, but they can't estimate the real system performance with the simulators. Some examples are SimOS [111], QEMU [112], and SimNow [113].

Micro-architecture simulators model in detail the processor's architecture and can estimate the performance of an application with different processor configurations. However, these kinds of simulators normally don't model the interaction between the architecture and the operating system and other system devices, and they tend to be expensive in terms of time. Researchers have used three major approaches to tackle this problem. The first approach is using statistical sampling to reduce the volume of instructions to simulate, as SIMPoint [114] and SMARTS [115] do. The second approach is working on the parallelization of the simulator itself, such as in Proteus [116] or the more recent Graphite [117]. The third approach is using FPGAs to implement the simulator itself (such as in RAMP Blue [118]), reducing the simulation times but also limiting the flexibility of the micro-architecture to simulate.

Full system simulators, such as Simics [119] and COTSon [120], include the features of functional and micro-architecture simulators at the cost of simulation time. COTSon can

model a cluster supercomputer from the micro-architecture up to the operating system, but it's clearly oriented to hardware design, whereas our approach focuses on performance analysis of parallel applications.

Other authors have proposed simulation methodologies to evaluate the performance of large-scale parallel applications [121, 122, 123]. Carrington et al. use the same network simulator that we do, but the micro-architecture simulation is based on signatures of all computation regions [121]. León et al. present a parallel network simulator combined with a regular micro-architecture simulator [122]. They obtain good simulation speed-ups, but the simulator parallelization adds new problems such as the high variability in the accuracy of the results across different runs. Neither approach uses any information reduction process to reduce the simulation time, resulting in time-consuming simulations. Finally, Zheng et al. focus on selecting the computation regions that drive the application execution [123], as we also do. However, they require the intervention of an expert (that is, the application developer) to describe the most important computation bursts, while in our project, that part is automated.

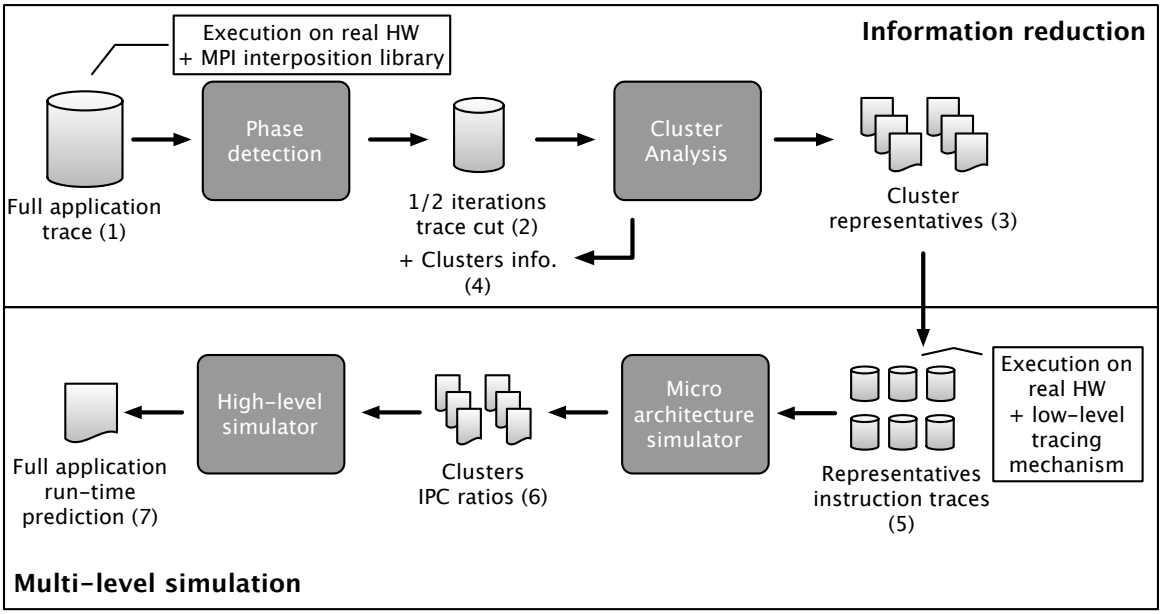


Figure 8.1.: Simulation methodology cycle for a whole supercomputing application. Starting with a trace of a parallel application (step 1), we produce a sub-trace (or trace cut) containing information of just two iterations (step 2). A cluster analysis is applied to the information of the computation regions present on this reduced trace, and a set of representatives per cluster is selected (step 3), adding cluster information to the trace cut (step 4). The set of representatives is traced (step 5) and simulated using a low-level simulator to obtain the ratios on other possible processor configurations (step 6). Finally, using a full-system-scale simulator, we combine the communication information present in step 3 and the cluster instructions per cycle (IPC) ratios (step 6) to predict the total runtime of the whole application (step 7).

8.2. Methodology

As a whole, the basic idea of the methodology we propose consists of combining two simulators to predict the behaviour of message-passing parallel applications. We use a high-level simulator to predict the network behaviour of the application and a micro-architecture simulator to predict the behaviour of the application computation regions. This combination of simulators is to the approach presented in [121].

The main contribution of the methodology we propose is the the characterization the computation regions. We dramatically reduce the required time to produce the full-application prediction by simulating only the most representative CPU bursts of each computation region detected in the micro-architecture simulator, the most time consuming simulator we use. We call this the *information reduction* process.

The top of Figure 8.1 depicts this information reduction process. It is decomposed into a phase detection analysis, which can distinguish the iterations present in the parallel applications we work with, and the cluster analysis presented in previous chapters, which groups detects the CPU bursts of an iteration that compose the different computation phases. Finally, a representative set of CPU bursts per each phase are selected.

Then, the selected CPU bursts are simulated at the micro-architecture level to obtain their performance in the target machine to study. This information is provided to a high-level application simulator that predicts the whole parallel application's execution time. This combination of two simulators with different abstraction levels composes the multilevel simulation process, depicted at the bottom of Figure 8.1.

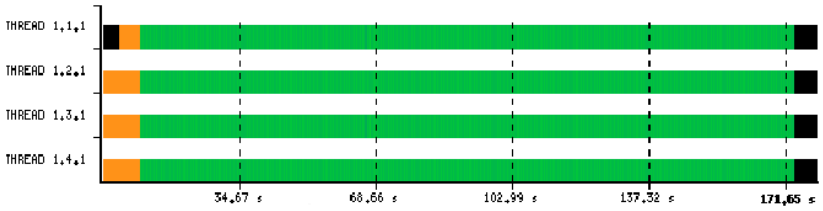
The combination of the information reduction process with the multi-level simulation process permits software performance analysis beyond what current performance counters would allow. Furthermore, it lets us predict the performance of an application running on a future system for which no performance data can be used as a reference machine. All these performance analyses can be done while maintaining high accuracy in performance predictions and without needing exhaustive simulations.

8.2.1. The Information Reduction Process

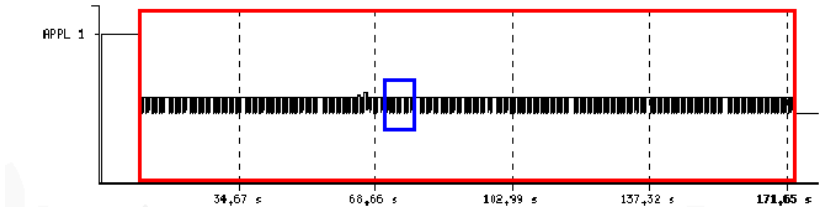
As a whole, the information reduction process focuses on selecting the minimum number of CPU bursts that represent the different computation phases present in the application. To start this process, we run the whole application we want to analyse to obtain a Paraver application-level trace (Figure 8.1, step 1).

Phase Detection

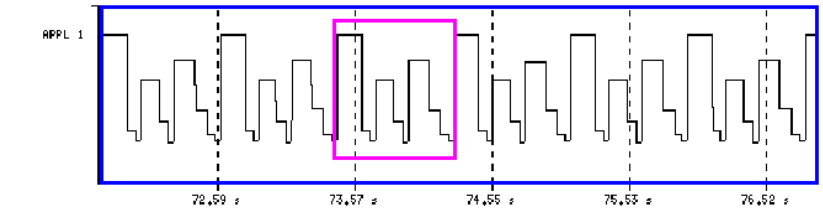
The underlying idea of phase detection is that we can benefit from the iterative nature of high-performance computing (HPC) applications. In almost all HPC applications we find a periodic region composed by sets of computations and communications that follow an iterative pattern, the application's main loop. In this way, we can reduce the data to analyse and simulate by selecting a representative segment of the whole application run that contains few iterations of this main loop.



(a) Phase detection analysis input data time-line



(b) Signal derived from input data. Red box marks the periodic region detected by the DWT



(c) Zoom detail of the blue box of previous figure. Pink box marks an iterative phase detected using the autocorrelation function

Figure 8.2.: Input data and the two stages of the phase detection analysis: periodic region detection using the DWT and the iterative phase detection using the autocorrelation

The phase detection is based on signals: we express an application's behaviour in terms of time-varying functions generated from different performance metrics present in the trace. Once we have obtained the signal, we apply to it the discrete wavelet transform (DWT). The DWT is able to capture the values of the input signal's frequencies and also the physical location where those frequencies occur. This property let us find the location of the periodic region HPC applications contain, as these regions have a strong high-frequency behaviour within the signal's domain. In addition, the DWT gives an approximate value of the main frequencies in each periodic execution phase.

Using an autocorrelation function with these main frequencies, we automatically detect the the iterative pattern present in such periodic region detected. Assuming each periodic region detected by the DWT has an iterative pattern of T seconds and since there are no significant differences between the repetitions of the pattern, it is possible to select several such regions, notably simplifying the subsequent analysis in terms of the amount of data that must be analysed.

In Figure 8.2 we illustrate the two steps of the phase detection process using the NPB BT benchmark with 4 tasks. Time-line 8.2a presents the input data used by the DWT. In this

time-line, X axis represent the time and Y axis represents the tasks involved in the parallel execution. The colour expresses the duration of the CPU bursts, in a gradient from green to blue, where black and orange represent those burst with duration below and above the gradient intervals. Using this input data, we derive the signal 8.2b, which is used by the DWT to detect the periodic region of the application, marked with a red box in this picture. Finally, the signal presented in 8.2c corresponds a zoom of the region marked with a blue box in signal 8.2b, where the pink box marks the iterative phase pattern detected by the autocorrelation function.

At the end of this step we generate a sub-trace (or trace cut), which is a portion of the original trace, containing information of n iterative phases of the periodic region. Typically, using just one or two iterations of the main loop (Figure 8.1 , step 2) is enough to keep the structure of the application clearly. In addition, this analysis also gives us the cut factor, so as to approximate the application's total runtime by multiplying it per the sub-trace duration. We use this cut factor in the final step of our methodology. (For further information on phase detection, see previous work by [64])

Cluster Analysis and Representatives Selection

After selecting one or two iterations of the application, we must characterize the different CPU bursts present in these iterations. In this step of the information reduction process, we identify the inner computation structure of each iteration by using the cluster analysis techniques presented in previous chapters. This information is added to the trace cut, as a pair of events, wrapping each computation burst (Figure 8.1, step 4).

After the CPU bursts characterization, we select a reduced number of representatives from those regions detected (the *clusters*) that represent a significant percentage of the application's execution time. Only these cluster representatives will be simulated at the micro-architecture level. We consider the minimum number of clusters that cover more than 80 percent of the total execution time of the application —usually less than six clusters.

Selecting the cluster representatives implies two decisions: first, select a small subset of tasks (from 1 to 5) where representatives will be taken, and second, select the CPU bursts themselves to be traced. Our experiments show that there is no significant difference between the selection schemes, so we chose the representatives at random.

This selection results in a reduced set of CPU bursts (no more than 10 in our validation experiments) that precisely represent the different computation behaviours in the application trace. Furthermore, we also provide the exact location of these CPU bursts in the application execution (Figure 8.1, step 3) in order to obtain the instructions trace needed to start the second part of the methodology, the multi-level simulation process.

8.2.2. Multi-level Simulation

The parallel application's simulation process comprises two steps with different levels of detail. Once the cluster representatives have been selected, we proceed with the micro-architectural simulation (Figure 8.1, step 4). These simulations let us predict the behaviour of all the computation regions in the target machine we plan to evaluate. These results are

provided to the application-level simulator (Figure 8.1, step 5), which estimates the whole application's execution time (Figure 8.1, step 6) using the cut factor obtained before (Figure 8.1, step 2).

Micro-architecture simulation level

To obtain a micro-architecture trace at the instruction level, we must rerun the application with a low-level tracing mechanism. This trace describes in detail the source and target operands of each instruction, the instruction code, and the addresses of the memory accesses. To obtain this trace, we use `valgrind`. To identify where the different cluster representatives begin and end, we count the number of message-passing interface (MPI) calls performed before the selected CPU burst begins.

We use MPsim, a cycle-accurate simulator in which each simulated core comprises at least eight pipeline stages, although we can modify the pipeline depth by adding decode or execution stages. To reduce computational costs, MPsim provides a trace-driven front end. However, it also supports simulating the impact of executing wrong-path instructions (when a branch miss predictions occurs), as it has a separate basic block dictionary containing the information of all static instructions of the trace.

The MPsim memory subsystem is accurately modelled, having a complete cache hierarchy with up to three levels of caches and main memory. The simulator also models bus conflicts to access shared levels of cache and main memory. All caches are multi-banked and multi-ported, offering a range of configurations to the user. MPsim optimistically assumes that main memory is perfect and, thus, all memory accesses will hit.

Application Simulation level

Using the application trace cut produced after phase detection with the clusters information (Figure 8.1, step 4) and the performance predictions per cluster (Figure 8.1, step 6), we can rebuild the entire application's performance in the target machine to study.

To perform this high-level simulation, we used the Dimemas simulator. Dimemas reconstructs the time behaviour of a parallel application on a machine modelled by performance parameters. In Appendix A we present detailed information of Dimemas model.

CPU ratios. Dimemas offers a way to tune computation regions simulation by applying a CPU ratio, a divisive factor to the duration of the bursts that will run in a given processor. Using the CPU ratio we can modify the CPU time request to simulate different CPUs. In this methodology, we apply different ratios to the clusters found using the timings produced by the micro-architectural simulation, we can accurately predict the timings of the application's computation parts. Because we select only those clusters that cover 80 percent of the computation time, we have the ratio for just a subset of all CPU bursts. To consider those CPU bursts that don't belong to the main clusters, we apply to them the weighted average of the ratios we actually computed. This decision is based in the consideration that even these burst do not represent a significant time of the application computation phases the predicted time could be affected by keeping their original durations.

Full application runtime projection. Once we've simulated the iterations present on the trace cut using the CPU ratios, we have the runtime prediction for this segment of the trace. To obtain the full application runtime, we multiply this runtime by the cut factor we obtained during phase detection.

8.3. Validation

The main purpose of experimental validation is to quantify which errors are introduced in each step of the methodology. Because it's divided into two processes, we distinguish two types of potential errors:

- representativity errors, or evaluations of the quality of the representatives obtained in the information reduction process, with respect to the information present on the original trace; and
- simulation errors, or the errors introduced by the two different simulators.

Finally, we must also consider the errors resulting from applying the whole methodology along with the reduction in simulation time from applying our simulation methodology.

We conducted the validation experiment using two applications: the Versatile Advection Code (VAC) and the Weather Research Forecasting (WRF) model. In both cases, the applications executed with 128 tasks on the MareNostrum supercomputer. In the case of WRF, initialization and finalization phases covered a significant part of the application's execution time. This behaviour was detected with our simulation methodology, which suggests that we should focus only on the application's computation phase. In contrast, VAC shows a more balanced behaviour, and we included initialization and finalization phases in the reported performance results.

8.3.1. Information Reduction Quality

To report the error that the information reduction process introduced, we analysed the error introduced by each decision taken in this part of the methodology. Because simulation time is mainly dominated by the micro-architecture simulation time, and this simulation time is proportional to the number of instructions to simulate, we will use the number of total instructions to simulate in order to estimate simulation time. Alternatively, we also use the number of CPU bursts as an indicator of our selected representatives' quality. Finally, we use the average IPC of the computation phases to measure the error introduced by the information reduction process. Even if IPC isn't the most adequate metric to measure parallel applications' performance, it adequately represents computation phases, which in our case are free of communications.

Tables 8.1 and 8.2 summarizes the results from the information reduction process of WRF and VAC applications respectively. The complete trace of WRF, Table 8.1, comprises a total of 8.29×10^{12} instructions and 4.1 million CPU bursts, with an average IPC of 0.551. Simulating 10^{12} instructions is unreasonable in current micro-architecture simulators, because

they normally simulate 10^5 instructions per second, which implies nearly 10^{10} simulated instructions per day. The situation is similar with VAC, Table 8.2, because the total number of instructions to simulate is 3.46×10^{13}. In this case, the number of CPU bursts is half a million, which indicates that the computation phases are longer than in the case of WRF. The average IPC of its computation phases is 0.274.

	All Trace	Two-iteration Cut	Clusters	Representatives
No. of Bursts	4137194	51008	1274	10
Reduction Factor	–	81.1	3247	413719
No. of Instructions	8.29×10^{12}	1.00×10^{11}	8.98×10^{10}	7.30×10^{8}
Reduction Factor	–	82.4	92.3	11345
IPC	0.551	0.549	0.555	0.554
Error	–	0.37%	0.60%	0.40%

Table 8.1.: Reduction factors and quality evaluation of the different parts of the Information Reduction step of WRF application

After applying the periodic phase detection mechanism, we identify the periodic behaviour of WRF and extract two out of 160 iterations. This new trace comprises 51,000 bursts (an $82\times$ reduction factor) and a total of 10^{11} instructions to simulate. Even if we have an $82\times$ reduction factor in the number of instructions to simulate, the average IPC of these CPU bursts is 0.549, which represents a 0.37 percent error. In the case of VAC, we extracted two out of the 100 iterations of the detected periodic behaviour. This new trace has a $50\times$ reduction factor in CPU bursts and instructions to simulate, whereas the average IPC is 0.276, which represents a 0.91 percent error. The low error obtained in both applications is coherent with the iterative nature of HPC applications.

Next, the cluster analysis gathers the different CPU bursts of the trace into different clusters. This characterization process first filters a significant amount of CPU bursts (identified as noise) and then selects the clusters that cover most of the trace's execution time. In the case of WRF, we find five clusters, covering 88 percent of the total execution cycles. These clusters represent 1,274 CPU bursts (a $3,200\times$ reduction factor) with a total of 8.98×10^{10} instructions to simulate (a $92\times$ reduction factor). The reduction in CPU bursts is around one order of magnitude larger than the reduction in instructions to simulate. This fact indicates that the cluster analysis effectively filters the CPU bursts that aren't representative in terms of execution time and IPC. Despite the large reduction in instructions to simulate, the average IPC of these five clusters is 0.555, which represents a 0.60 percent error. We obtained similar conclusions in the case of VAC, where only two clusters cover 81 percent of execution time, with a $700\times$ and $70\times$ reduction in CPU bursts and instructions to simulate, respectively, with an average IPC of 0.260, which represents a 5.08 percent error.

Finally, we must select a set of representatives from the identified clusters of CPU bursts. In the case of WRF, we select two representatives per cluster at random. Consequently, the

	All Trace	Two-iteration Cut	Clusters	Representatives.
No. of Bursts	523616	10495	746	8
Reduction Factor	–	49.9	702	65452
No. of Instructions	3.46×10^{13}	6.80×10^{11}	4.78×10^{11}	6.20×10^{9}
Reduction Factor	–	50.7	72.5	5587
IPC	0.274	0.276	0.260	0.253
Error	–	0.91%	5.08%	7.57%

Table 8.2.: Reduction factors and quality evaluation of the different parts of the Information Reduction step of VAC application

number of selected CPU bursts is 10, and the number of instructions to simulate is 7.3×10^8 (a $11,300\times$ reduction factor). The average IPC of these CPU bursts is 0.554, which represents only a 0.40 percent error with respect to the original trace. Because the number of identified clusters in VAC is lower, we select four representatives per cluster. Here, the total number of CPU bursts is eight, whereas the number of instructions to simulate is 6.2×10^9 (a $5,587\times$ reduction factor). The average IPC of these CPU bursts is 0.253, which represents a 7.57 percent error with respect to the original trace.

To summarize, the combination these three techniques in the information reduction process leads to a dramatic reduction in the input data to simulate, between four and five orders of magnitude less instructions, with a small error in the global application's IPC. This huge reduction and high accuracy is due to the natural repetitive behaviour of HPC applications and the intelligence of the successive techniques used in this information reduction process. Considering the instruction simulation rates introduced previously (10^{10} instructions per day), a parallel application that would require several years of simulation time can be simulated in a few hours, obtaining at the same time a detailed analysis at the micro-architecture and application levels.

8.3.2. Multi-level Simulation Quality

Next, we evaluate the error introduced by the two simulators involved to simulate a whole supercomputer application. We first focus on *self validation* of the methodology, or predicting the execution time of the application in the same system in which it had been executed. We then focus on *cross validation* of the methodology, or predicting the execution time of the parallel applications on a different system configuration.

Self Validation

As we described earlier, we use MPsim, an in-house micro-architecture simulator with a detailed pipeline and cache hierarchy. MPsim has been developed to perform research in

Features	Specification
Architecture	2 cores, 2-way SMT, superscalar architecture
Fetch/Issue/Retire width	8/5/5 instructions per cycle
Fixed-point and load/store issue queue	36 entries
Floating point issue queue	20 entries
Branch instructions issue queue	12 entries
CR-logical instructions issue queue	10 entries
Vector instructions issue queue	36 entries
Reorder buffer	100 entries
Branch predictor	16K-entry gshare[1]
L1 instruction cache	64KB, direct mapped, 128B line, 1 cycle hit
L1 data cache	32KB, 2-way, 128B line, 2 cycle hit, LRU
L2 unified cache	1MB, 8-way, 128B line, 15 cycle latency, LRU
Memory latency	250 cycles
Peak memory bandwidth	2 GBps per GHz

[1] Two-level adaptive predictor with globally shared history buffer and pattern history. Commonly used in most current processors.

Table 8.3.: Baseline MPSim processor configuration

the processor's micro-architecture. Thus, this simulator's target architectures are future architectures that will appear in the market in about 10 years. For this reason, the accuracy when modelling a particular existing micro-architecture isn't as important as the relative differences when projecting the performance of future architectures.

To predict the performance of the chosen HPC parallel applications running in a real machine, we carefully chose the simulator parameters to model a PowerPC 970-like processor. Table 8.3 summarizes this machine's main characteristics.

Table 8.5 shows the IPC values obtained with MPsim and those measured on our real supercomputer. In the case of WRF, the IPC predictions obtained with MPsim are always within 40 percent and, on average, 25.16 percent different from the real IPC. In the case of VAC, the error per cluster remains within 40 percent, whereas the average error increases to 33.1 percent. We observe that the performance of CPU bursts over 100 million instructions is normally overestimated, whereas the performance of shorter CPU bursts is underestimated. For short traces, recovering the state of the cache hierarchy implies a significant portion of simulation time. On the real machine, part of this data is already on the cache hierarchy, which reduces the execution time of these computation phases. We've measured that it takes more than 5 million cycles. In contrast, this initialization time is less significant for longer traces. In this case, the overestimation is due to some structures optimistically modelled in MPsim: memory is assumed to be perfect (we assume that we will not access the disk). Also, some implementation details of the PowerPC 970 processor aren't public, and we've followed

Feature	Specification
Number of nodes	2560
Processors per node	4
Input links per node	1
Output links per node	1
Number of buses	∞[1]
Memory bandwidth per node	600 MB/s
Memory latency	$4\mu s$
Network bandwidth	250 MB/s
Network latency	$8\mu s$

[1] Contention is only defined by input and output links

Table 8.4.: Baseline Dimemas cluster configuration

a best-effort approach.

Reducing the average IPC error of MPsim is outside the scope of this work. More accurate simulators exist, but they're normally industrial simulators developed by the company selling the processor. These simulators are more detailed (and slow), but show IPC estimations within a 1 percent error. Because selecting the micro-architecture simulator is orthogonal to the presented simulation methodology, we report the error in execution time when using the IPC obtained with MPsim or the real IPC that would be obtained with an industrial simulator.

Finally, we simulate the original trace with Dimemas using the micro-architecture simulator's feedback. Table 8.4 shows the reference parameters used in the different experiments. This configuration models the MareNostrum supercomputer, composed of clusters of IBM JS21 server blades. Each node has two PowerPC 970MP processors (four cores total) and 8 Gbyte of RAM memory. The nodes are connected using a Myrinet network. In this table, memory and network latency refer to the time added by the simulator to each communication, in terms of library initialization, not the actual latency of these units.

Figure 8.3a shows the error in the total execution time of two iterations of the parallel application predicted with Dimemas, as well as of the whole application error after applying the cut factor. Using the IPC provided by MPsim, we obtain 7.67 percent error in the execution time prediction of two iterations of WRF. This error is reduced to 6.25 percent when predicting the execution time of the whole application. Using the IPC values measured on the real machine (or a highly accurate industrial simulator), the error is reduced to just 0.3 percent for two iterations and 1.62 percent for the whole application. This error is computable to the error introduced by Dimemas. In the case of VAC, the error increases to 16.2 percent (two iterations) and 21.7 percent (full application) with MPsim values, and 0.54 percent (two iterations) and 7.2 percent (full application) with the real IPC values. Thanks to the combination of communication and computation phases, the initial error of MPsim in

	WRF			VAC		
Cluster Number	Real IPC	MPSim IPC	Error (%)	Real IPC	MPSim IPC	Error (%)
1	0.529	0.703	32.77	0.289	0.380	31.71
2	0.497	0.466	-6.39	0.251	0.340	35.38
3	0.618	0.429	-30.72	–	–	–
4	0.755	0.468	-38.04	–	–	–
5	0.811	0.522	-35.60	–	–	–
Weighted Average Error (%)			25.16	Weighted Average Error (%)		33.12

Table 8.5.: Real IPC vs. MPSim predicted IPC comparison in self validation experiment, using two threads per core configuration

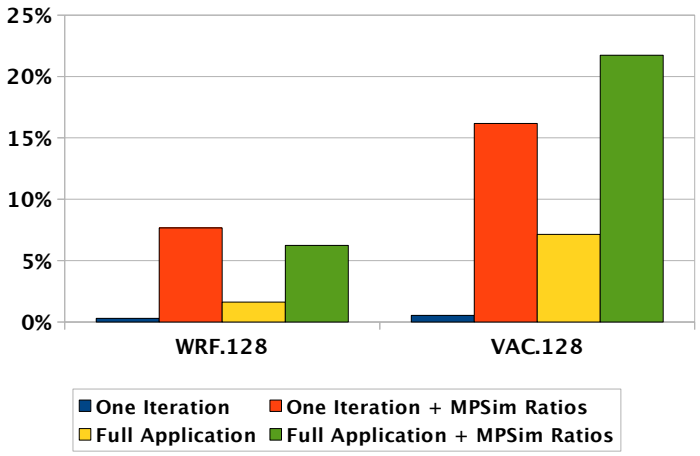
IPC predictions is nearly divided by 3 in the final prediction of execution time.

Cross Validation

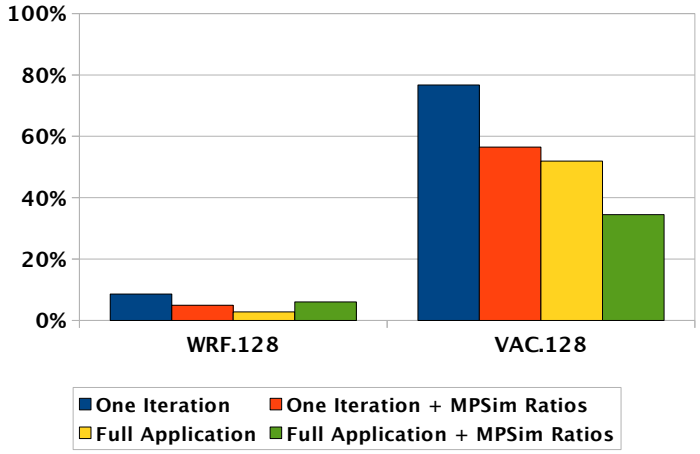
We can use our simulation methodology to predict the performance of a parallel application when running on a different system configuration. More specifically, we can predict the parallel application's execution time when running one task per node (single-thread configuration) instead of four tasks per node (CMP configuration). Nodes in our supercomputer infrastructure comprise two dual-core processors with a shared Level 2 (L2) cache. Furthermore, the two chips share the 8 GB of memory. We've run the application using four tasks per node configuration and predicted the performance when using one task per node configuration. To cross validate the results, we reran the application using one task per node and compared its runtime with the prediction.

First, we used MPsim to simulate a dual-core PowerPC 970-like machine, as described in Table 8.3, with just one representative of each cluster running. Then we ran two representatives of the same cluster in the same processor configuration. The performance improvement per representative is used as the required CPU ratio to the feedback Dimemas simulator. Finally, Dimemas simulations are done with the cluster parameters of the new configuration (we have just one processor per node instead of four as in Table 8.4).

Figure 8.3b shows the average error results we obtained. Without using the CPU ratios derived from MPsim, we obtain 8.6 percent and 76.7 percent errors in the execution time of two iterations of WRF and VAC, respectively. When predicting the whole application's execution time, the error is reduced to 2.78 percent and 51.9 percent for WRF and VAC, respectively. The difference in the error is due to the fact that VAC performance is much more affected than WRF when moving from a configuration with one task per node to four tasks per node. When using the CPU ratios from MPsim, we reduce the error to 4.94 percent and 56.5 percent for two iterations of WRF and VAC, respectively. In the case of WRF, the measured CPU ratios for the five representative clusters are between 1.05 and 1.21, as Table



(a) Self Validation Errors



(b) Cross Validation Errors

Figure 8.3.: Execution time prediction error for Versatile Advection Code (VAC) and Weather Research Forecasting (WRF) parallel applications, for both self validation (a) and cross validation (b) experiments. The figures show the error when estimating the execution time of two iterations of the application or the full application execution time with the measured real IPC and the IPC provided by MPsim.

	Task 07			Task 09					
Cluster	IPC CMP	IPC ST	Ratio	IPC CMP	IPC ST	Ratio	Representatives average ratio	Real Ratio	Error (%)
1	0.705	0.717	1.017	0.700	0.712	1.017	1.017	1.047	-2.90
2	0.494	0.505	1.022	0.437	0.445	1.018	1.020	1.083	-5.82
3	0.428	0.479	1.119	0.429	0.481	1.121	1.120	1.275	-12.18
4	0.466	0.512	1.099	0.469	0.517	1.102	1.101	1.251	-12.04
5	0.518	0.647	1.249	0.526	0.655	1.245	1.247	1.169	6.68
							Weighted average error (%)		-4.62

Table 8.6.: WRF cross validation MPSim Ratios, comparing the IPC running two-threads per core (CMP) and a single thread per core (ST) and the differences with the ratios in real configuration

Cluster	Task 51 Ratio	Task 79 Ratio	Task 85 Ratio	Task 104 Ratio	Representatives average ratio	Real Ratio	Error (%)
1	1.048	1.049	1.053	1.049	1.050	1.756	-40.22
2	1.042	1.042	1.042	1.039	1.041	1.937	-46.28
					Weighted average error (%)		-42.54

Table 8.7.: VAC cross validation MPSim Ratios and the differences with real ratios. In this case, we just express the ratios and not the IPC on single thread and CMP configurations because of the higher number of representatives

8.6 shows. The predicted ratios are between 1.02 and 1.25, with an average error of 4.62. As a result, the final error in the whole application's execution time prediction is just 6.04 percent.

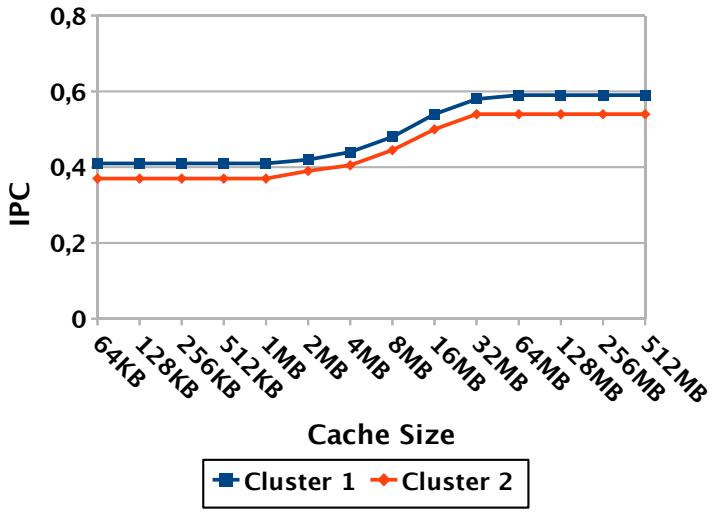
In the case of VAC, the measured CPU ratios for the two representative clusters are 1.75 and 1.92 (see Table 8.7). The predicted ratios are 1.04 and 1.05, with an average error of 58.3 percent in the CPU ratios. This high error is due to the optimistic model of RAM memory implemented in MPsim. As we mentioned earlier, memory is assumed to be perfect, with no misses. VAC is suffering between nine and 11 L2 misses per kilo instruction, whereas WRF suffers only between one and two L2 misses per kilo instruction. Apart from that, VAC suffers much more TLB misses than WRF (a $2.5\times$ increase). As a result, the final error in execution time prediction of the whole application is 34.5 percent. Even if this accuracy is acceptable for our experiments, a more detailed model of the memory hierarchy is required in the micro-architecture simulator to obtain a more accurate prediction.

8.4. A Use Case: Performance Prediction

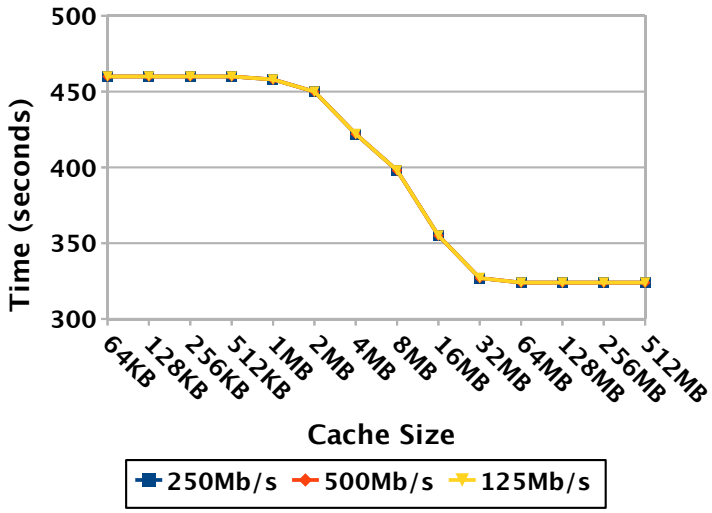
This last experiment aimed to illustrate our simulation methodology's potential. In particular, we studied in detail the evolution of the behaviour of the applications as the size of the L2 cache and network bandwidth increased. This kind of performance analysis is not feasible if we must simulate the whole application at the micro-architecture level, because simulating just one configuration with a given L2 size would require several years of simulation time. Instead, we obtained our results with less than 1 day of simulation.

Figure 8.4a shows results extracted from the VAC application in terms of IPC. In this case, we've divided the computing time into two clusters. From the perspective of the L2 cache size, we can see the same behaviour in these two clusters—that is, a remarkable increase of the application's IPC if the L2 cache is larger than 2 MB. The most important conclusion we can extract from this is that we can improve this application's performance by executing it on a supercomputing infrastructure whose processors have an L3 cache level that can reduce the cycles dedicated to access principal memory due to L2 misses. In Figure 8.5a, we depict the same results for the WRF application. In this case, we can see a strong improvement of the performance in all the clusters until 1 MB of the L2 cache size is reached. This behaviour is due to the small size of the input data. Thus, WRF will have enough with a small L2 cache to reach its maximum performance.

Figures 8.4b and 8.5b show the execution times obtained with Dimemas after the cluster IPC ratios. This approach lets us study the impact of architectural parameters (in this case, L2 cache size) and network parameters (in this case, bandwidth) simultaneously. According to our results, the impact of the network is negligible in the case of VAC and WRF. For these applications, the dominant performance factor is IPC, which significantly changes with the L2 cache size.

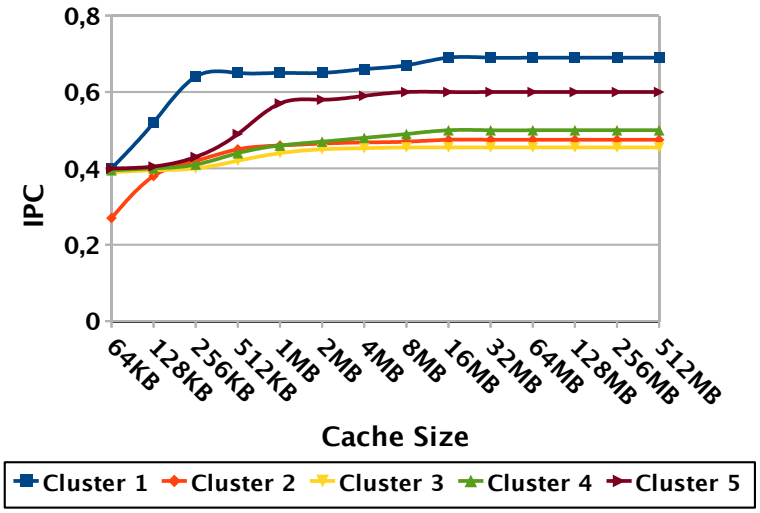


(a) IPC prediction using different cache sizes

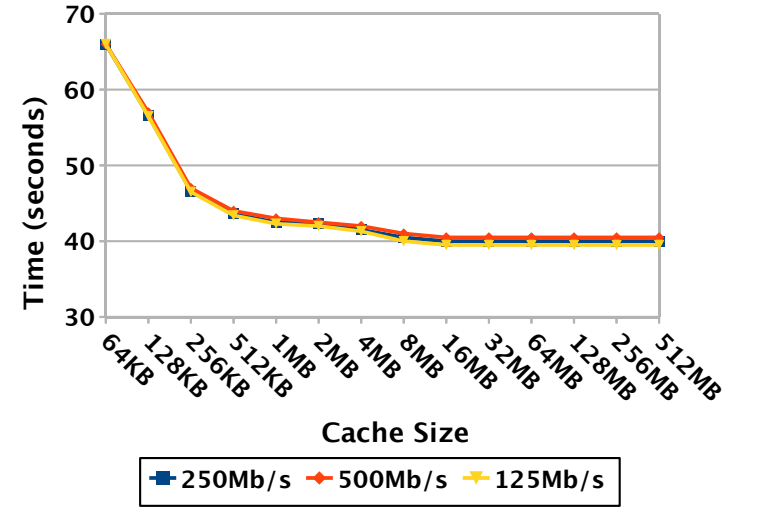


(b) Execution time prediction using different cache sizes and network bandwidths

Figure 8.4.: Performance predictions of VAC using different cache sizes and network bandwidths



(a) IPC prediction using different cache sizes



(b) Execution time prediction using different cache sizes and network bandwidths

Figure 8.5.: Performance predictions of WRF using different cache sizes and network bandwidths

9. Analysis of Message-Based Parallel Applications

In this chapter we demonstrate the usefulness of the structure detection technique using it to analyse four different state-of-the-art applications. We first characterize the phases detected and then we provide different *what-if* analyses to measure the potential application gain when improving these phases and also how the application will behave in new hardware configurations. These *what-if* analyses can only be performed by using the structure detection.

9.1. Applications Analysed

We selected four applications to perform the analyses. These applications are commonly used in HPC facilities in different research areas.

- PEPC [124]. PEPC is a parallel tree-code for rapid computation of long-range (1/r) Coulomb forces for large ensembles of charged particles. The heart of the code is a Barnes-Hut style algorithm employing multi-pole expansions to accelerate the potential and force sums, leading to a computational effort $O(N \log N)$ instead of the $O(N^2)$ which would be incurred by direct summation. Parallelism is achieved via a 'Hashed Oct Tree' scheme, which uses a space-filling curve to map the particle coordinates onto processors. The kernel (tree routines and force computation) is separated from the application 'front-end' so that the code can be easily adapted to both electrostatic and gravitational problems.

- WRF [125]. The Weather Research and Forecasting (WRF) Model is a mesoscale numerical weather prediction system designed to serve both operational forecasting and atmospheric research needs. It features multiple dynamical cores, a 3-dimensional variational data assimilation system, and a software architecture allowing for computational parallelism and system extensibility. WRF is suitable for a broad spectrum of applications across scales ranging from meters to thousands of kilometres.

- GADGET [126]. A code for cosmological simulations of structure formation, is a freely available code for cosmological N-body/SPH simulations on supercomputers. It computes gravitational forces with a hierarchical tree algorithm (optionally in combination with a particle-mesh scheme for long-range gravitational forces) and represents fluids by means of smoothed particle hydrodynamics (SPH). The code can be used for studies of isolated systems, or for simulations that include the cosmological expansion of space, both with or without periodic boundary conditions.

- SU3_AHiggs [127]. SU3_AHiggs is a lattice quantum chromodynamics (QCD) code intended for computing the conditions of the Early Universe. Instead of the "full QCD", the code applies an effective field theory, which is valid at high temperatures. In the effective theory, the lattice is 3D. For this reason, SU3_AHiggs stresses different parts of the architecture than the conventional QCD applications using 4D lattices.

9.1.1. Data gathering

All applications analysed were executed with 256 tasks the in MareNostrum supercomputer, allocating 4 MPI tasks per node.

For the PEPC application we applied the performance data extrapolation technique described in chapter 7 extracting the counters required to build the CPI breakdown model presented in the Section 7.4 of that chapter. To extract the counters we used the time-space multiplexing scheme.

For the rest of applications we gathered the information of a set of 8 performance hardware counters, described in table 9.1. We chose this set as it contains the counters we commonly use to perform the computation structure as well as memory related counters.

Counter Name	Description
PM_CYC	Processor cycles
PM_INST_CMPL	Number of Eligible Instructions that completed
PM_INST_DISP	The ISU (Instruction Sequencer Unit) sends the number of instructions dispatched
PM_LD_REF_L1	Total DL1 (Level 1 Data Cache) Load references
PM_ST_REF_L1	Total DL1 Store references
PM_LD_MISS_L1	Total DL1 Load references that miss the DL1
PM_DATA_FROM_MEM	Data loaded from memory
PM_GCT_FULL_CYC	The ISU sends a signal indicating the GCT (Global Completion Table) is full

Table 9.1.: List of all hardware counters used in the experiments

In all cases, we reduce the amount of total data by manually selecting a region for the analysis that contains few iterations of the main application loop.

9.2. Analyses description

For each application we present a report that includes two different analyses, first, the structure characterization; second, a set of *what-if* to illustrate how the structure detected could

be useful to the analyst/developer so as to understand the scalability of his or her application.

9.2.1. Structure characterization

For each application, we first perform a structure detection using the Aggregative Cluster Refinement algorithm. To perform this detection, we combine the metrics Completed Instruction and IPC except for WRF application, where this pair of metrics lead to an over-aggregation of the clusters, and we used Completed Instructions and Main Memory Accesses counters

For each application, we present the refinement tree, the clusters time-line distribution as well as the scatter plot of the metrics used in the cluster.

Once we have the computation structure, we select those regions that have an important weight in the total application computation time. This selection considers just the clusters representing regions that cover more than 5% of the total application computation time. For each of these most representative clusters, we show a set of metrics useful to observe the performance of the regions these clusters represent. Further *what-if* analyses will use these clusters to evaluate the scalability of the different applications.

In addition, we also provide an insight of the applications' parallel efficiency, by using the speedup model presented in [8]. We discuss the factors of this model in Section 9.3. Thanks to the structure detection, we can measure how the performance cluster affects to the global speedup.

9.2.2. What-if analyses

The target of these *what-if* analyses is to present possible studies that we could perform thanks to the structure detection. These analyses focus on giving a detailed information about the application behaviour and also the possible impact certain implementations could obtain. In this way, we help to focus the parallelization efforts in those directions that seem more productive.

We define two *what-if* analyses: first, what would be the benefits if we improve the application?; second, what would be the sensitivity of the application to hardware improvements?

Application improvements

Typically, parallel applications exhibit two main problems, first the imbalance across the different tasks it comprises and second, poor sequential performance in some of the regions they contain.

In these analysis, we evaluate two possible improvements the developer could apply to its application to solve these problems. First, a duration balance of the most time consuming computation clusters. Second, a sequential performance improvement of the clusters that show poor IPC.

In practice, the duration balanced improvement could be addressed by tuning the data distribution across the different application processes. To model it, we consider that all the

burst that conform a given cluster will have the same duration. This duration will be the average burst duration observed for such cluster in the real execution.

In the case of the sequential performance improvement, it could be increased for example by exploiting the memory locality if the application does not take into account the cache sizes or unrolling loops to avoid the loop overheads. To model the sequential performance improvement, we scale the duration of the bursts present in the clusters that obtain an IPC lower than 0.8. We select this IPC because in the MareNostrum supercomputer this is the minimum value observed in the computation regions of those applications that perform well. The scale factor used is the ratio of the average IPC obtained for such cluster and 0.8.

As a whole, comparing these two improvements would indicate where is more interesting to devote the efforts so as to obtain the maximum gain in the overall performance of the application. It is interesting to remark that in these two improvements we evaluate the results considering the modifications just in the selected clusters. This will provide a finer insight of the scalability different applications exhibit.

Use of different hardware

In this analysis we evaluate how the application will perform when porting to a faster machine, both in terms of network bandwidth and/or CPU speed. The different networks we define present bandwidths in the range 256 MB/s to 16384 MB/s in powers of two. In terms of the CPU, we define two scenarios:

1. Using general purpose processors. In this scenario, we define a set of processors that can reach performance ratios in range 1 to 64 in powers of two, i.e. using a processor that can execute the computation as fast as the original platform, to a processor which can execute the computation 64 times faster.

2. Using acceleration hardware. In this scenario, we define a set of accelerators that can reach performance ratios in range 1 to 64 in powers of two. The different from the previous scenario is that in this acceleration hardware we only execute those bursts that belong to the representative clusters. In this way, we can model a realistic situation where we can only port to these acceleration hardware those regions that have a important weight in the computation, i.e. big subroutines or kernels of computation.

As a whole, this experiment will show us where could be the behaviour of the application in different supercomputers, some of them corresponding to currently available hardware components and some other with future parameters, without the need to having them. With this information, we can anticipate which could be the problems of the application in these different machines.

Applications simulation

To measure the impact of the modifications we contemplate in these three scenarios we use the Dimemas simulator, introduced in chapter 8. Detailed information about this simulator is presented in Appendix A.

To model the modifications defined in the bursts, we use a feature of the simulator that can adapt the execution times of the CPU burst contained on each of the clusters selected. To simulate the sequential improvement scenario and the use of new hardware scenario we the *CPU ratio*, a divisive factor applied to the bursts duration. When evaluating the duration balance, we use a simulator feature able to change the duration of these bursts inside a cluster by a fixed value during the simulation.

To model the different bandwidths used in the hardware porting scenario, we directly modify the network bandwidth parameter included in the simulator.

In all the *what-if* simulations we compare the gains in total execution time with respect the times obtained in a nominal simulation using the simulator parameters listed in table 9.2. These parameters describe the MareNostrum architecture, where the applications where executed to gather the data used in the experiments.

Feature	Specification
Number of nodes	2560
Processors per node	4
Input links per node	1
Output links per node	1
Number of buses	∞[1]
Memory bandwidth per node	600 MB/s
Memory latency	4μs
Network bandwidth	235 MB/s
Network latency	8μs

[1] Contention is only defined by input and output links

Table 9.2.: Baseline Dimemas cluster configuration

9.3. Analyses results

In Table 9.3 we present the results of the cluster analysis of the four applications analysed. Basically, we present the total number of clusters found, and the number of clusters that represent more than 5% of total application computation time. We also include in this table the characterization of the four applications in terms of the parallel efficiency, a model to numerically evaluate the speedup of parallel applications defined in [8]. The parallel efficiency of an application is defined by the following equation:

$$\eta = CommEff \times LB \tag{9.1}$$

The parameter η is a measurement of the parallelization efficiency, in other words, which are the losses in the application due to its parallel implementation. For example, a η of 0.7

	PEPC	GADGET	WRF	SU3_AHiggs
Cluster Analysis				
Total Clusters	2	16	9	17
Computation Time (%)	98.00	96.55	89.03	92.64
Clusters > 5%	2	4	7	5
Computation Time (%)	98.00	75.50	81.45	60.70
Efficiency model factors				
η	0.711	0.582	0.712	0.981
$CommEff$	0.837	0.612	0.793	0.989
LB	0.849	0.951	0.898	0.992
IPC	0.705	0.649	0.493	1.008
$\#Inst$	2.743×10^{11}	2.168×10^{13}	5.583×10^{11}	3.309×10^{12}

Table 9.3.: Cluster analysis results and factors of the speedup model defined in [8] obtained by the four applications analysed

indicates that the application spends 30% its time in communications and synchronizations or imbalances caused by its parallelization.

The parallelization efficiency is decomposed in two factors, first the communications efficiency, $CommEff$, which measures which part of the efficiency losses are caused by in effective communications regions, i.e. when all the processes interchange information. The second factor is the load balance, LB, which measures the differences in the durations of the computation regions observed across all processes. This second factor serves to evaluate the time spent in communications caused by the load imbalance observed across the different processes. It will be useful to indicate which are the possible improvements in the duration balancing experiments.

Finally, in the table we also include the average IPC and the total number of instructions executed, $\#Inst$, within the computation time of each application. These two figures serve to measure the efficiency of the sequential part of the different applications.

Using these factors, we can have a clear idea of how well the application have been parallelized and which are the parts of the parallelization itself that provoke the bigger resource losses.

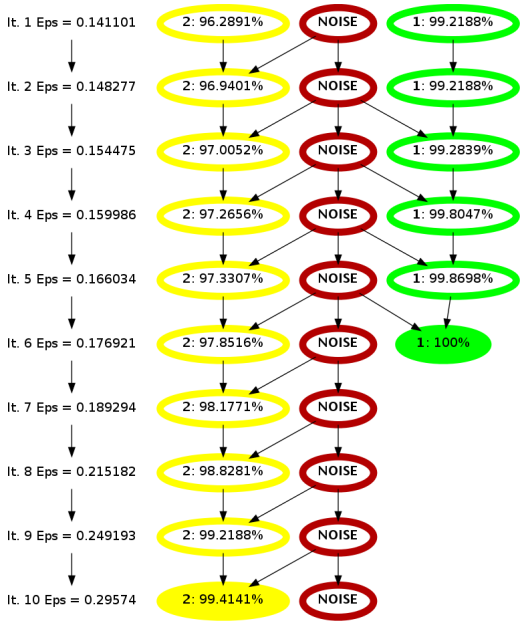


Figure 9.1.: PEPC refinement tree

9.3.1. PEPC

Structure detection

This first application we analyse presents a parallel efficiency of 0.711, as can be seen in Table 9.3. This relatively low value is decomposed in a communications efficiency of 0.837 and a load balance of 0.849, indicating that there is no single element that affect the most to the parallelization efficiency.

In this table we can also see that the Aggregative Cluster Refinement was able to detect just two clusters, which represent 98% of the total application. We use both cluster in the rest of the analysis. The outputs of the clustering algorithm are presented in Figures 9.1 is the refinement tree, Figure 9.2 shows the scatter plot of clustering metrics, and Figure 9.3 contains the clusters time-line distribution (of 5 iterations of the periodic region). Finally, in Table 9.4 we present the statistics obtained by each cluster.

The refinement tree, Figure 9.1, shows that both clusters have a core structure detected in initial levels, and only noise points are added to them along the different iterations of the clustering algorithm. This structure indicates that the SPMD regions the clusters represent are do not have clear sub-structures.

In terms of the sequential performance, Table 9.3 shows that the average IPC of the whole application is 0.705. In Table 9.4 we can see that the average IPC of Cluster 1 is 0.619 and the average IPC of Cluster 2 is a little bit higher, 0.870. This difference is the main reason to distinguish them as two clusters (see plot 9.2). The elements that affect the average IPC obtained by each cluster can be explained the CPI breakdown model we introduced in chapter 7, depicted in Figure 9.4. Using this model we detect that most of the stall cycles that occur in

Cluster 1 are due to floating point unit causes: the basic latency of the unit (dark purple) and also the latency of divisions and square root floating point instructions (blue). In the case of Cluster 2, this model indicates us the it is mostly dominated by memory operations: the basic latency of the Load Store Unit (LSU, dark green) and more precisely that volume of data cache misses (light green).

In terms of the load balancing, the value obtained by both clusters is around 0.850 (LB_{dur} in Table 9.4). These values are similar to the overall application load balance (LB in Table 9.3), which means that the overall imbalance is uniformly distributed across the two main computation regions. The effects of this imbalance can clearly be seen in time-line 9.3, as the white communications regions that appear just after the clusters.

In the case of Cluster 1 its duration imbalance can be mainly associated to instructions load balance (LB_{inst}), 0.885. In Cluster 2, the duration imbalance can also be associated to instructions load balance, 0.846, and also the IPC balance, 0.942. This variability measured in the different factors can also be observed in the scatter plot (Figure 9.2).

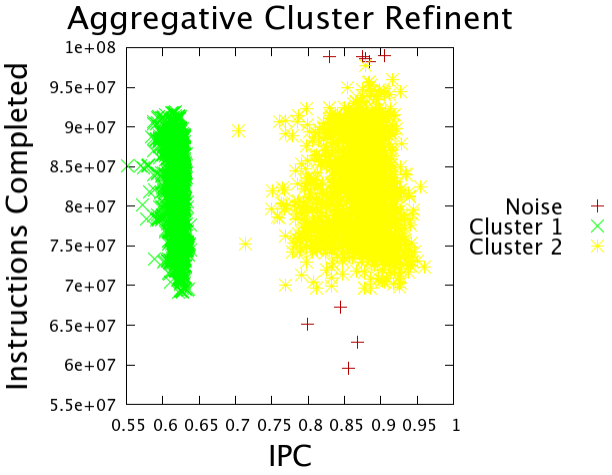


Figure 9.2.: PEPC scatter plot of discovered clusters

Metric	Cluster 1	Cluster 2
Total Computation Time (%)	57.02	40.98
Avg. Duration (ms)	57.954	41.892
LB_{dur}	0.859	0.841
Avg. Completed Inst.	8.130×10^7	8.260×10^7
LB_{inst}	0.885	0.846
Avg. IPC	0.619	0.870
IPC Balance	0.974	0.941

Table 9.4.: PEPC clusters characterization

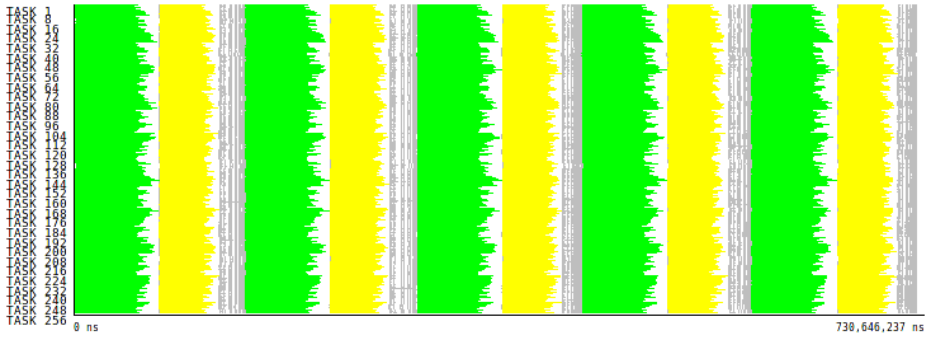


Figure 9.3.: PEPC time-line distribution of clusters

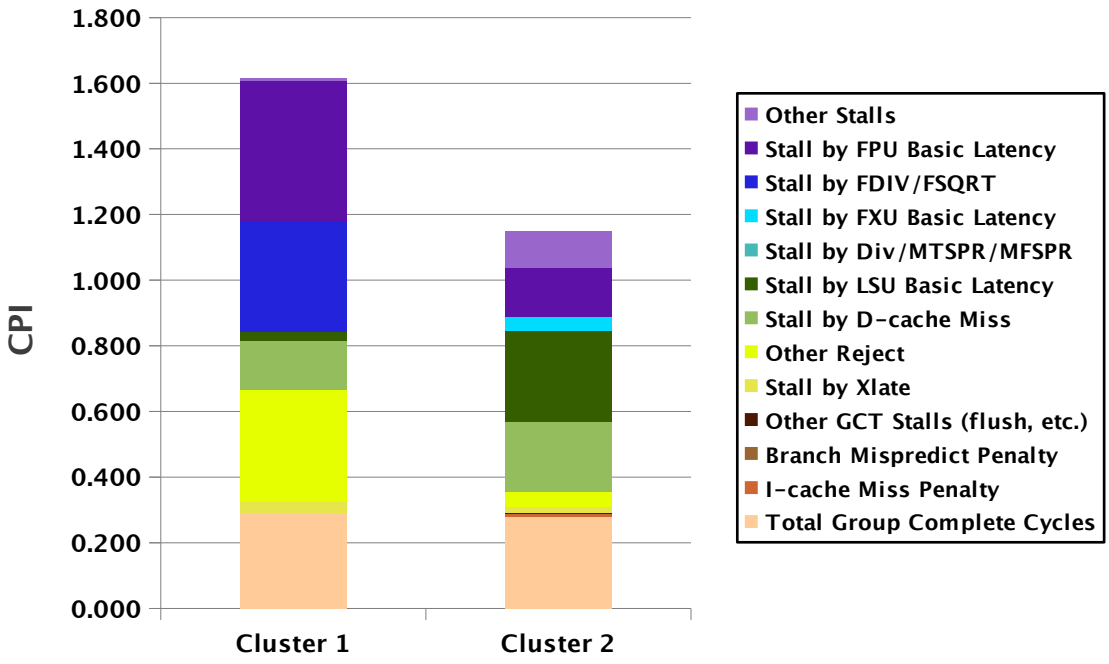


Figure 9.4.: PEPC CPI breakdown model of the clusters found

Configuration	Execution time (s)	Speedup
Nominal	0.835	–
Duration balancing (All Clusters)	0.715	1.168
Algorithmic Improvement	0.740	1.128

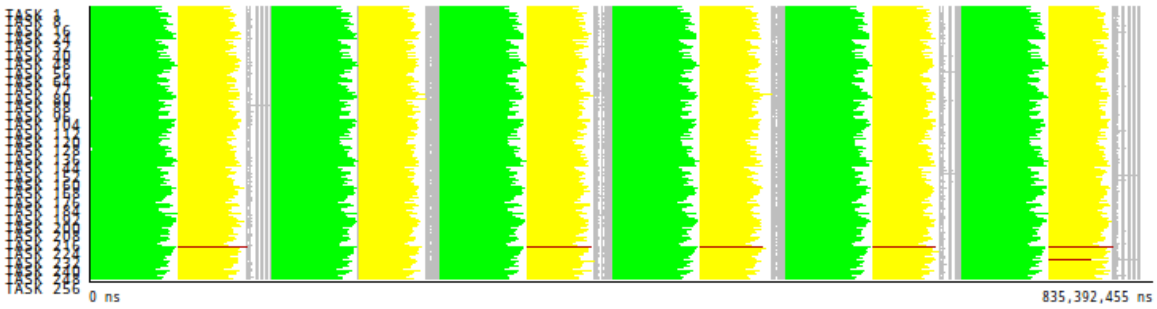
Table 9.5.: PEPC speedups of application improvement analyses

Application improvement analysis

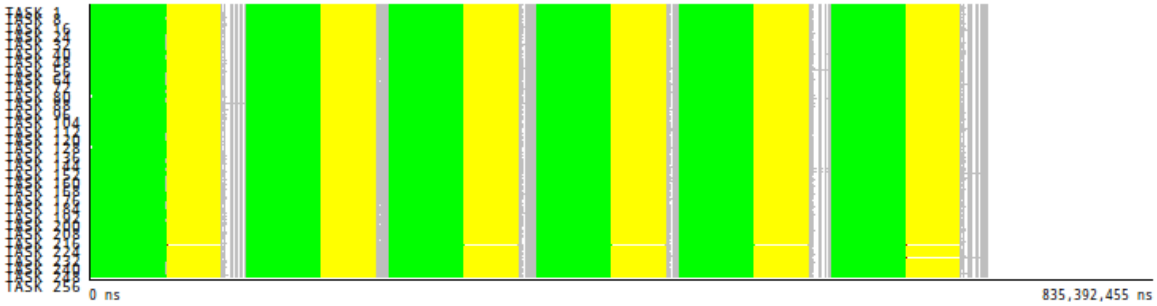
Table 9.5 contains the execution time of the nominal simulation, and the execution times and speedups obtained in the duration balancing simulation and sequential performance improvement simulations. In Figure 9.5 we can see the cluster time-lines of these three simulations presented at the same scale, so as to compare them visually. We want to note that the difference execution of the nominal simulation vs. the real execution of this application was just 3.85%.

We have seen that both clusters detected in PEPC present a relatively low duration load balancing, so we simulated the balancing of all of them. As a result of this improvement, we obtained an speedup of 1.168. To understand how the application reached this speedup, we can observe the time-line of the duration balancing simulation (9.5b) where the communication region (white) after Cluster 1 (light green) in time-line 9.5a have disappeared due to the balance of this cluster. These regions corresponds to a synchronization between all the processes involved in the application. Due to the perfect balance of the clusters, the wait times of the shorter processes disappear. Similarly, a synchronization that appears just after Cluster 2 (yellow) also disappears in the different iterations of the application's main loop.

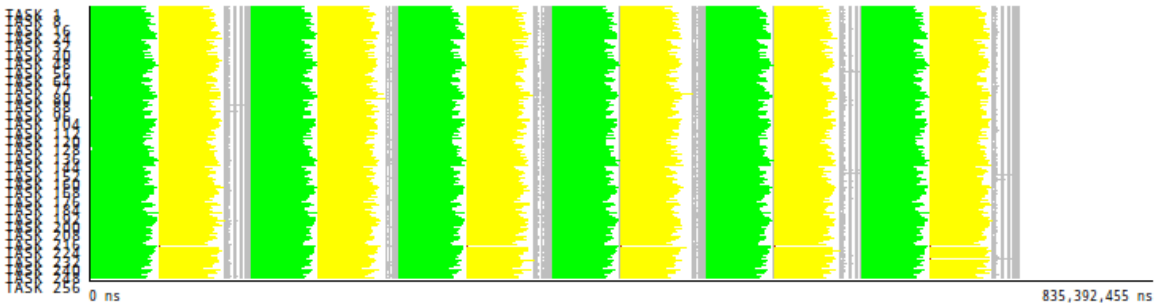
In the case of the sequential improvement, we quantified only the improvement of Cluster 1, as its average IPC is 0.619 is far from the target value of IPC we consider, 0.8. We model an improvement of the bursts present by reducing their duration by a 30%. As a result of this reduction, the speedup obtained was 1.128, a lower gain than the one obtained when balancing the clusters. We can observe in time-line 9.5c that the improvement obtained corresponds almost to the Cluster 1 reduction. In addition, there is also a gain to the reduction collective communication that appears just after this cluster. Even the duration load balancing of the cluster is not modified in this experiment, the waiting times it produces (grouped in the collective communication) are shortened in absolute terms.



(a) Clusters time-line distribution of nominal simulation



(b) Clusters time-line distribution of duration balancing simulation



(c) Clusters time-line distribution of Cluster 1 algorithmic improvement

Figure 9.5.: Clusters distribution time-line of nominal simulation, duration balancing simulation and the algorithmic improvement of PEPC application

Use of new hardware analysis

We present the results of the acceleration scenario using surface plots. In these surface plots, the X axis contains the different CPU ratios applied to the bursts, the Y axis contains the different Bandwidths used and finally, the Z axis represents the speedup obtained by this combination with respect to the nominal simulation.

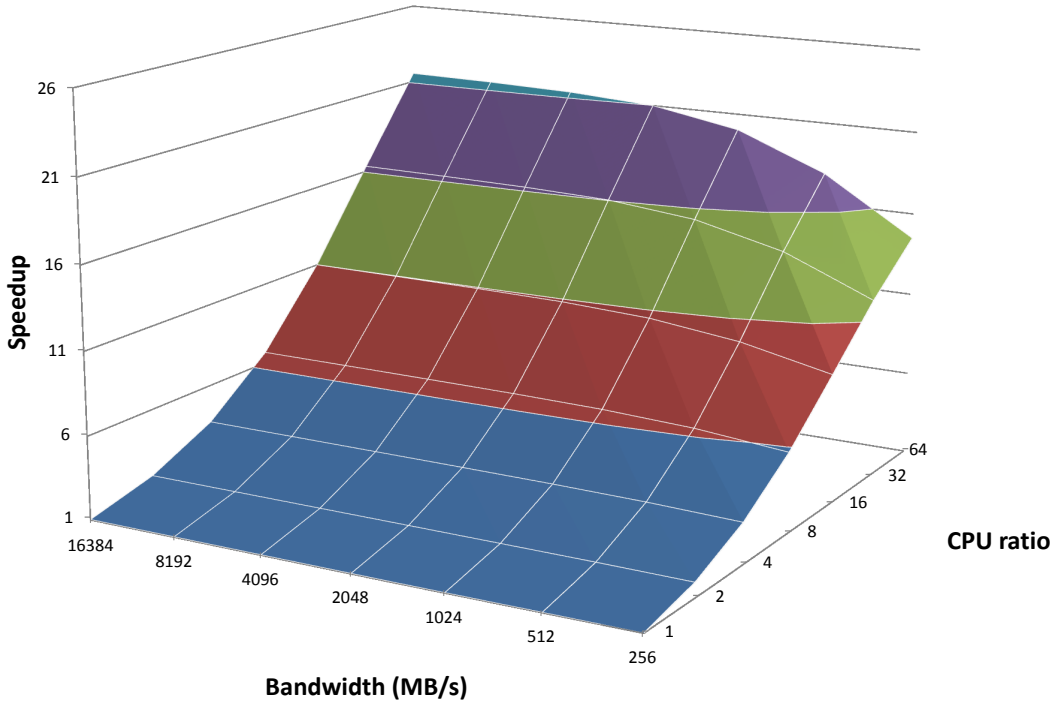
Figure 9.6 contain the two surface plots obtained for PEPC. Plot 9.6a present the results of applying the CPU ratios to all the computation of the application, i.e. the use of general purpose processors. Plot 9.6b present the results of applying the CPU ratios to the two clusters detected but not to the filtered CPU bursts, i.e. supposing a port of these clusters to acceleration hardware.

The first observation we can do in plot 9.6a is that the maximum speedup observed has an upper bound around 21. This value is far from a theoretical speedup around 64 in the extreme case, where all the computation is reduced 64 times and the message transmissions are reduced 64 times by using a network with 64 times more bandwidth. A second observation we can do of this plot is a network bandwidth bound: the speedup does not improve after using a network of 2,048 MB/s. This shows that PEPC will not benefit by using networks with higher bandwidth. On the other hand, we can see that in terms of the CPU ratio, there is still a margin of improvement, as the slope in this axis is steep.

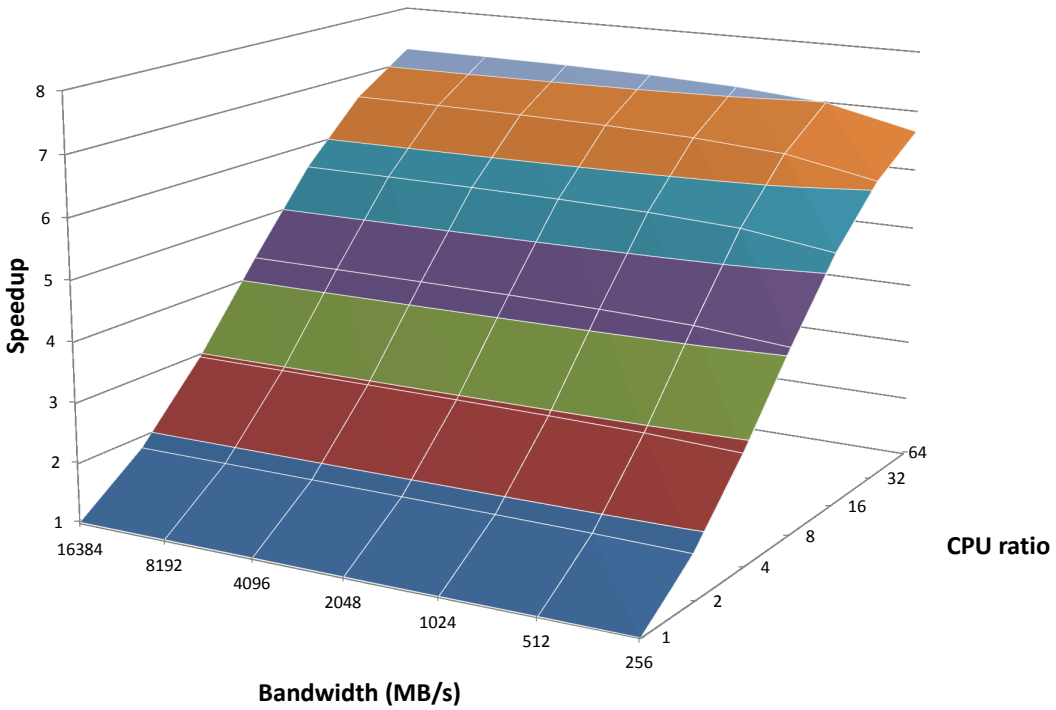
In the case scenario of only "accelerating" the two clusters detected, plot 9.6b, we can quickly detect that the top speedup in just around 8. This value is 3 times smaller than the obtained in the previous configuration. In this case, the upper bound of the bandwidth appears at a lower value, around 512 MB/s. The speedup due to the CPU ratio seems to have a margin of improvement.

If we compare both plots, we can see that this maximum speedup observed when using accelerators, around 8, is reached when using a general purpose CPU of ratio 8 (plot 9.6a). This is a really interesting result because the baseline system we use the experiments uses an old PowerPC 970MP processor. Current CPUs could reach this 8x ratio easily, while having accelerators of ratios higher than 32x its not that common.

In Figure 9.7 we present the time-lines of five iterations of the application when using the extreme hardware configurations: a network of 16,384 MB/s and a general purpose CPU 64 times faster (9.7a) and the same network bandwidth but using an "accelerator" 64 times faster (9.7b). Notice that those time-lines are at different time scale, so as to show the weight of the different application phases. In time-line 9.7a we see that when the small regions become shorter, and the distance between them is approximately the minimum cost of a communcation call (latency time in Dimemas terminology, set to 4 μs), this is the cause that the application do not take in advantage higher network bandwidths. On time-line 9.7b these small computation regions between the communication primitives dominate the iteration execution. Even though these smallest computation regions represent just 2% of the computation in the initial execution, the imbalance of a minimum part of them provoke a dramatic loss of performance when porting the application to a computer based on acceleration hardware.

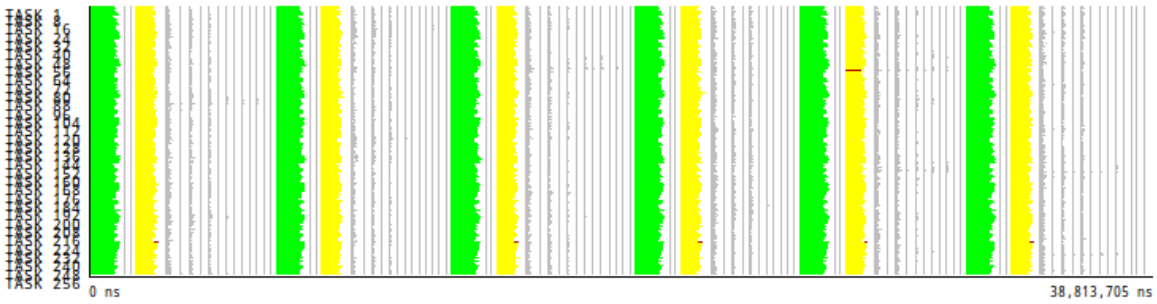


(a) Speedups using general purpose CPUs

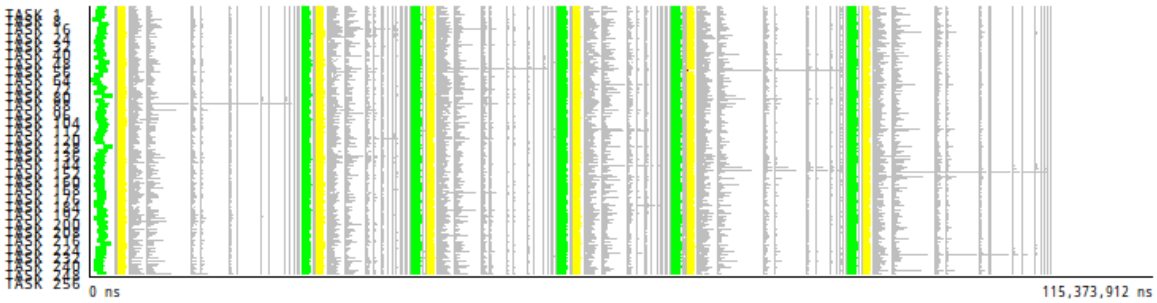


(b) Speedups porting the most consuming clusters to accelerators

Figure 9.6.: Results of the simulation when using PEPC in a different hardware



(a) Clusters distribution time-line using a general purpose CPU



(b) Clusters distribution time-line using an accelerator

Figure 9.7.: PEPC clusters time-lines using 16384 MB/s network bandwidth and general purpose CPU 64 faster and accelerator 64 faster

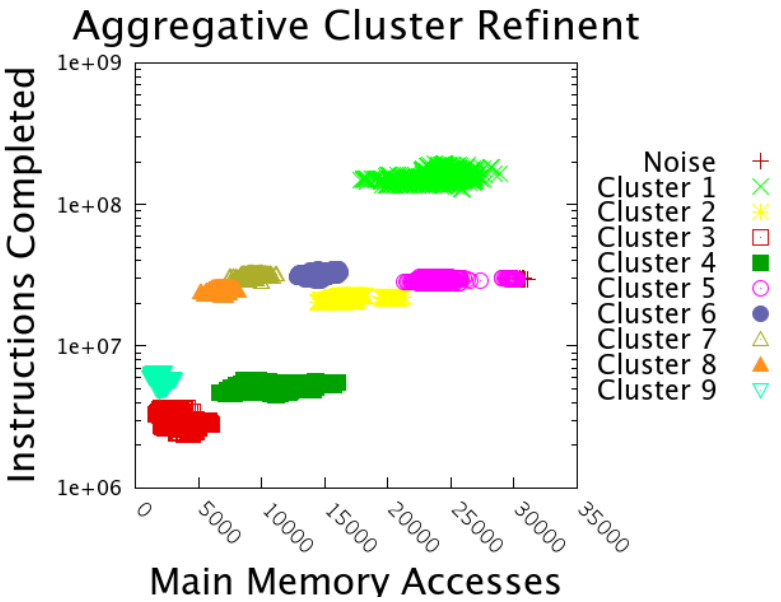


Figure 9.8.: WRF scatter plot of discovered clusters

9.3.2. WRF

Structure detection

This second application has a parallel efficiency of 0.712, similar to the previous one (Table 9.3). On the other hand, the parameters that define this efficiency have a certain difference, being the communications efficiency 0.793 and the load balancing 0.898. The communications efficiency is the factor that has more impact on the total parallel efficiency.

WRF is the only application where we used a different hardware counter metrics to detect the computation structure. We combined Completed Instructions counter with the Main Memory Accesses counter (also named Last Level Cache Misses). We used this counter combination instead of Completed Instructions and IPC because we obtained a clearer phase detection. In the scatter plot of Figure 9.8 the different clusters detected using this pair of metrics, while Figure 9.9 contains the time-line distribution of the clusters. The refinement tree obtained generated by the Aggregative Cluster Refinement is depicted in Figure 9.10. Finally, in Table 9.6 we present the statistics obtained by each of them. In this application, we characterize a cut of 2 iterations of the periodic region detected.

In this application, the Aggregative Cluster Refinement detected 9 different clusters (Table 9.6. Only 7 of these clusters cover more than 5% of the total computation time, aggregating around 81% of the total computation time of the application. The rest of the clusters represent and additional 8% of the computation time, while the filtered bursts aggregate the remaining 11%.

In the time-line of Figure 9.9 we can observe how the different computation phases detected are distributed during two iterations the application execution. The time-line shows that the structure present sequences long computation, detected as clusters, and short computations, the filtered bursts in grey, interleaved with communications. That can be clearly

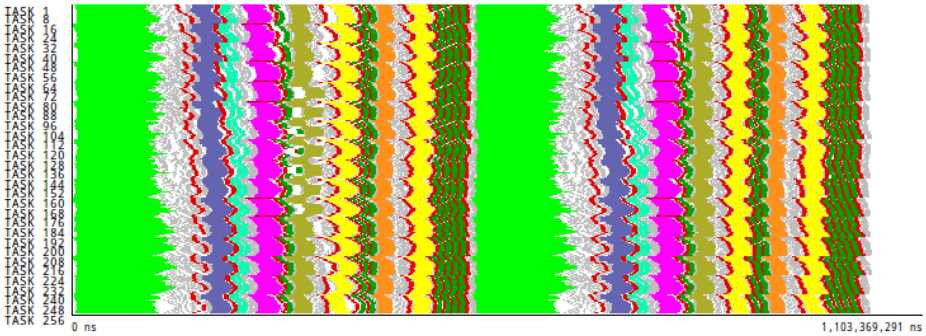


Figure 9.9.: WRF time-line distribution of clusters

observed after the region detected as Cluster 1 (light green).

The refinement tree of the structure detection, Figure 9.10, shows that except for Clusters 2, the rest of "major" clusters have been refined by absorbing noise points, so they represent SPMD without any clear sub-structures. In case of Cluster 2 (yellow), it absorved Cluster 12 (light brown) in the iteration 6. The scores obtained by Cluster 2 and Cluster 12 on iteration 5 were 92.1875% and 7.8125% respectively. This indicates that in Cluster 2 we can find a group of 20 processes (7.8125% of the 256 processes) that present an slightly different behaviour than the rest. This observation is not used in the studies we present here, but can be used for an in depth-analysis of the imbalances.

Also in the statistics in Table 9.6 and the clusters time-line we can distinguish how the different phases found have variability in terms of their average burst duration and the number of times they appear on each iteration. For example, Cluster 1 (light green) has an average burst duration around 100ms, but only appears once on each iteration, while Clusters 3 (red) and 4 (dark green) have an average burst duration between 3 and 5 ms, but appear several times on each iteration.

As we mentioned before, the load balancing of WRF is 0.898. On the other hand, observing clusters the (duration) balancing of the clusters (LB_{dur} in Table 9.6), we can detect that only Clusters 6 and 7 have values closer to the obtained by overall application, while the rest of clusters have values below 0.870. These values suggest that the small computation phases have a positive effect in the overall balance. In this case, the duration imbalances of Clusters 1 and 6 can be associated to the instructions imbalance (LB_{inst} is smaller than IPC balance), while in the rest of clusters can be associated with the IPC variability.

In terms of the sequential performance, the average IPC the whole application, 0.493 (Table 9.3), a low value. Table 9.6 shows that Clusters 1, 6 and 7, obtain an IPC slightly better than this average, while the rest of the "big" clusters obtain a lower IPC. Is interesting to note that the clusters that obtain the lowest IPCs are the Clusters 3 and 4, 0.436 and 0.444 respectively, represent those SPMD regions with shortest bursts. According to the table, we see that those regions that obtain a lower IPC also present the higher values L1 and L2, so it reflect a possible performance loss due to the way the memory is used in these phases. In any case, all the clusters obtain a performance far from the target IPC of 0.8 we consider as target, that presents an opportunity to perform sequential performance improvements in the application.

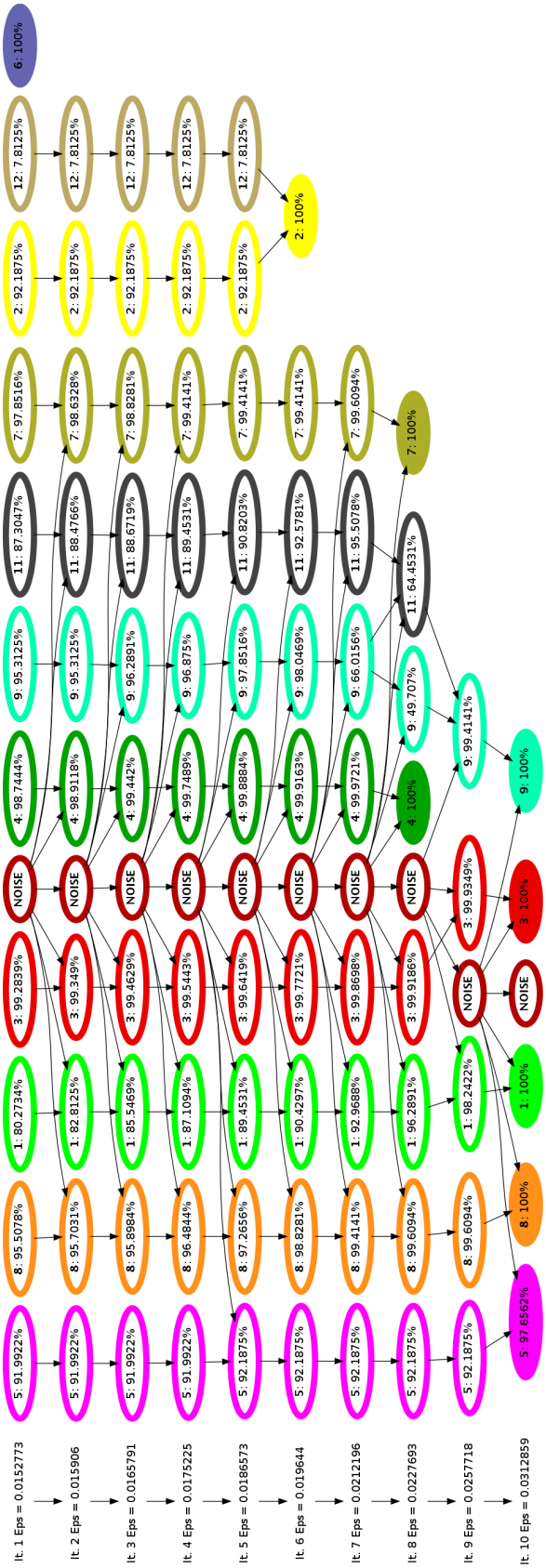


Figure 9.10.: WRF refinement tree

Metric	Cluster 1	Cluster 2	Cluster 3	Cluster 4	Cluster 5	Cluster 6	Cluster 7
Computation Application Time (%)	29.24	11.47	10.10	10.09	7.58	6.93	6.05
Avg. Duration (ms)	105.759	20.737	3.044	5.212	28.077	25.065	21.863
LB_{dur}	0.817	0.855	0.844	0.816	0.862	0.897	0.914
Avg. Completed Inst.	1.578×10^8	2.185×10^7	3.006×10^6	5.229×10^6	2.963×10^7	3.256×10^7	3.125×10^7
LB_{inst}	0.813	0.960	0.961	0.959	0.958	0.923	0.960
Avg. IPC	0.659	0.466	0.436	0.444	0.467	0.573	0.631
IPC Balance	0.958	0.891	0.897	0.865	0.868	0.938	0.882
Avg. MIPS	1492.447	1053.782	987.550	1003.254	1055.725	1298.953	1429.291
Avg. L1 misses per 10^3 inst.	5.691	18.839	19.786	26.881	15.789	7.619	3.709
Avg. L2 misses per 10^3 inst.	0.152	0.771	1.065	2.029	0.810	0.450	0.299
Avg. Memory Bandwidth (MB/s)	28.960	103.996	134.601	260.572	109.436	74.787	54.771

Table 9.6.: WRF clusters characterization

Configuration	Execution time (s)	Speedup
Nominal	1.020	–
Duration balancing (Clusters 1 to 6)	0.961	1.061
Algorithmic Improvement	0.820	1.245

Table 9.7.: WRF speedups of application improvement analyses

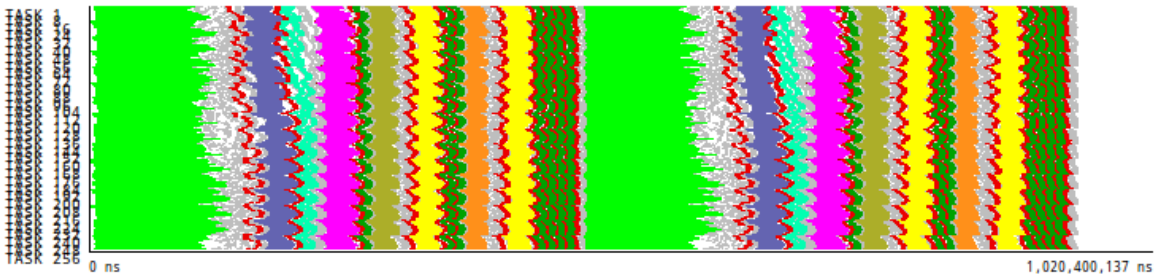
Application improvement analysis

The results of the two application improvements simulations as well as the nominal simulation are presented in Table 9.7. In this application the relative error of the nominal simulation vs. the real execution was 7.5%. In Figure 9.11 we present the time-lines containing the clusters distribution of these three simulations.

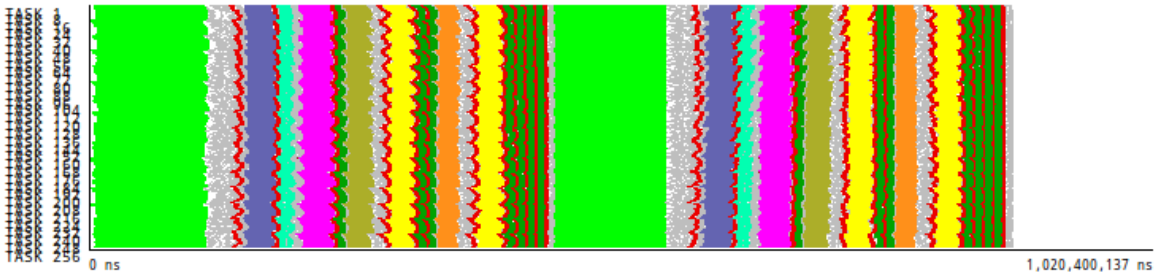
The speedup obtained when simulating a duration balancing improvement of the 7 clusters selected is 1.061. We have observed in the previous section that the duration balancing of these clusters was low, but this simulation shows that the gain obtained when balancing them is also low. This balancing implied an average reduction of 18% of the longest computation bursts contained in the clusters. As we stated in previous section, the duration balancing of the overall application is around 0.9, due to the effect of the smaller computation. The distribution of computation and communication limits total gain obtained by balancing the major clusters. We can observe this by comparing the time-lines corresponding to the nominal simulation, time-line 9.11a, and the time-line corresponding to the the balancing simulation, time-line 9.11b. Obviously, the computation regions in time-line 9.11b are more balanced, specially Cluster 1 (light green), but the communications and filtered computation region just after this cluster produces a skew in different processes. This skew affects all the computation regions and limit the total application improvement.

In terms of the sequential performance, we have seen that the average IPC of the application is 0.493 and also the average of the 7 major clusters is far from the 0.8 we set as target. For this reason, the margin of improvement we have in this scenario is higher. As a result, the speedup obtained is 1.245. Here it is interesting to note that even the duration of the different bursts have been reduced by and average of 55% (the actual reduction factor depends on the average IPC of each cluster), the overall gain is only 25%. This is caused mainly by two reasons: first, the 7 major clusters represent 81.45% of the total computation; second, the filtered bursts and the 2 clusters that have not been improved and the communications have an influence in the overall time. Comparing the time-line of this simulation 9.11c we can see how reducing the duration of these clusters produce the mentioned improvement, but also it is interesting to note that it also causes a minor increase of some communications. For example, we can observe a communications bubble that appears in the first iteration. It is visible as the wait region that starts in the intermediate processes just before the Cluster 6 (purple) region and propagates to the firsts processes, ending between the Cluster 9 (light blue) and Cluster 5 (pink) region.

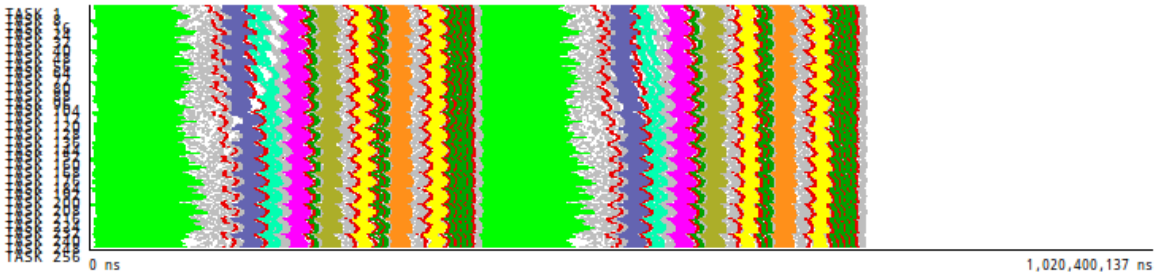
To sum up, in this application, it seems to be more interesting to put the efforts to improve the application in order to tackle the low performance obtained by the different computation regions more than their duration balance. Even the values we used to run the simulation can be considered difficult to obtain in the reality (an average performance improvement of 55% is quite high), these simulations reflect that the gain obtained applying this improvement will be higher.



(a) Clusters time-line distribution of nominal simulation



(b) Clusters time-line distribution of duration balancing simulation



(c) Clusters time-line distribution of Clusters 1 to 6 algorithmic improvement

Figure 9.11.: Clusters distribution time-line of nominal simulation, duration balancing simulation and the algorithmic improvement of WRF application

Use of new hardware analysis

The results of the experiments of porting WRF to new hardware are depicted in the surface plots of Figure 9.12. Plot 9.12a presents the results of using general purpose CPUs. Plot 9.12b presents the results of using acceleration hardware on the 7 selected clusters.

We can observe that when using general purpose CPUs, the maximum speedup reached is around 46, an acceptable value. In this scenario, we can observe that the application benefits both by using faster processors and faster networks, but it requires a trade-off between these two hardware elements. For example, to take advantage of using faster CPUs, we require using a network of more than 1GB/s. This affirmation relies on the speedups obtained when using networks of 256 and 512 MB/s, a "valley" at the right of the plot, where it seems that the performance will not increase by adding faster CPUs. On the other hand, when using CPUs of ratios 1, 2, 4 and 8, the application will not get benefited by using networks faster than 256MB/s. This observation correspond to the blue "valley" in the front of the plot. Finally, in the extreme cases, we can observe that the CPUs still a have a margin of improvement, as the slope of the CPU axis is steep, while the network bandwidth seems to be reaching a bound around the 16GB/s.

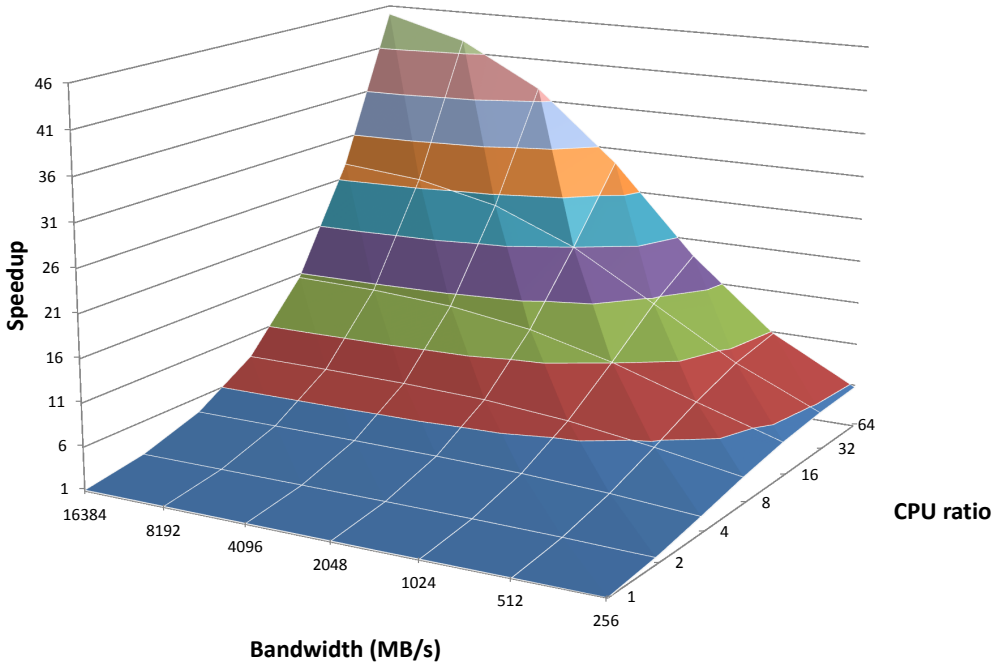
The results obtained when just accelerating the 7 majors clusters, plot 9.12b, reflect a dramatic performance loss, reaching a maximum speedup value around 3.8. This maximum speedup is 12 times smaller than the one obtained when using the general purpose CPUs. In this case, the most remarkable observation we can extract is that the application performance reaches a plateu when using accelerators 32 times faster and a network between 1 - 2GB/s. This situation reflects that WRF scalability will be very limited when using a accelerator based machine. This limitation can be very critical considering that achieving 30x acceleration of a given computation phases is feasible with current hardware and we can find 1GB/s networks in most of the supercomputers.

If we compare these two possible scenarios we can observe that we would just require a general purpose CPUs 2 times faster than the original ones and no network improvement to obtain the same performance as the maximum obtained by using 32 or 64 times faster accelerators and a network of 1GB/s.

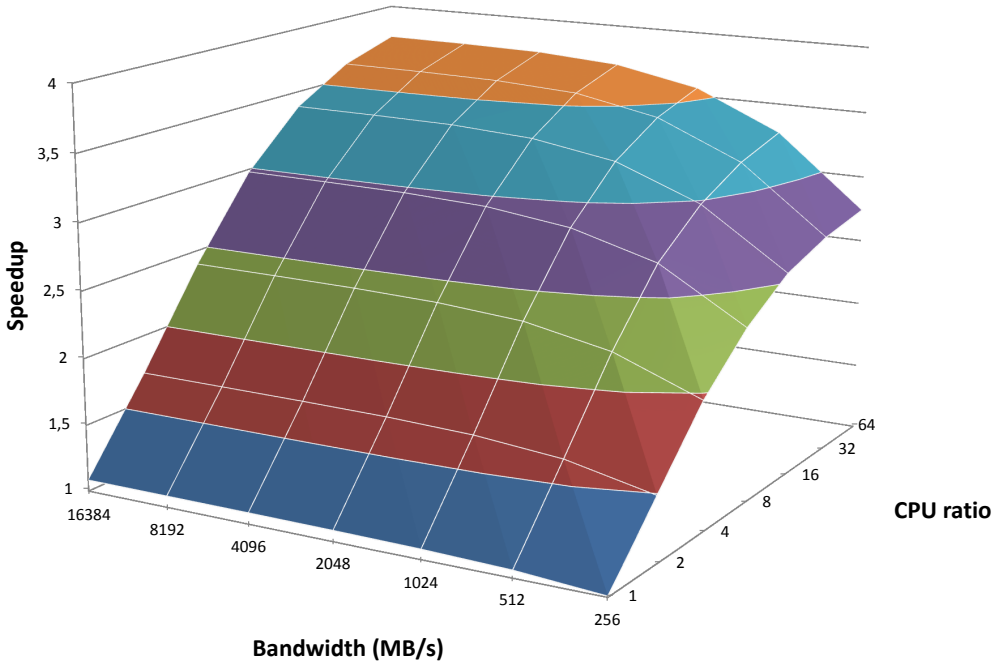
Finally, in Figure 9.13 we present the time-lines of the two extreme cases (16GB/s network and 64x CPUs/acceleration). These time-lines are at different scale.

In the general purpose CPUs scenario, time-line 9.13a, we can see that all the computation phases have been reduced proportionally compared to the real execution, time-line of Figure 9.9, but now, we can see that the communications have increase their importance (there are more white regions). We have measured it using the speedup model, and the $CommEff$ obtained when using the extreme configuration of general purpose CPUs is 0.616 when in the real execution was 0.793, also pointing the inability of this application to take advantage of a faster network, due its communication pattern.

In the acceleration hardware scenario, the time-line 9.13b shows again that those clusters that represented less time in the original execution, Cluster 8 (orange) and Cluster 9 (light blue), now have a heavy weight because they have not been "accelerated". In any case, the application is mainly driven by the filtered bursts (in grey). On the other hand, it is interesting to note that in this simulation, the communications efficiency is 0.928, a better value than the original one. This is caused by the fact that at this point, the application is mainly dominated by the computation, and reflects the observation done when looking at the surface plot 9.12b regarding to WRF will not take advantage of a fast network in this hardware configuration.

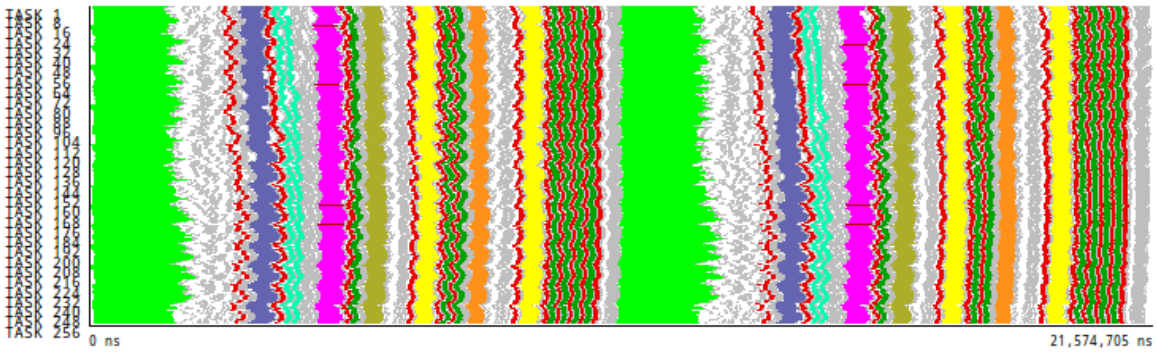


(a) Speedups using general purpose CPUs

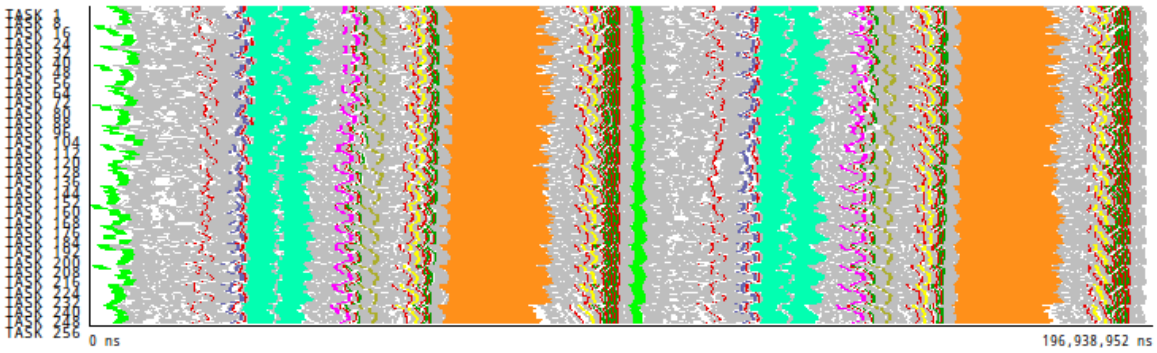


(b) Speedups porting the most consuming clusters to accelerators

Figure 9.12.: Results of the simulation when using WRF in a different hardware



(a) Clusters distribution time-line using a general purpose CPU



(b) Clusters distribution time-line using an accelerator

Figure 9.13.: WRF clusters time-lines using 16384 MB/s network bandwidth and general purpose CPU 64 faster and accelarator 64 faster

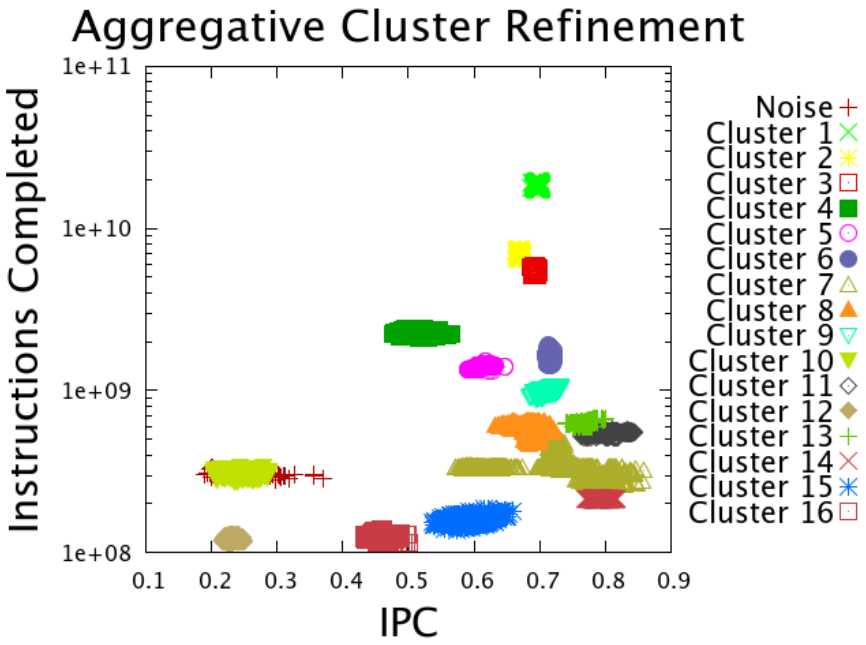


Figure 9.14.: GADGET scatter plot of discovered clusters

Metric	Cluster 1	Cluster 2	Cluster 3	Cluster 4
Computation Application Time (%)	40.54	15.92	12.17	6.87
Avg. Duration (ms)	1.166×10^{4}	4.578×10^{3}	3.501×10^{3}	1.980×10^{3}
LB_{dur}	0.940	0.948	0.950	0.927
Avg. Completed Inst.	1.844×10^{10}	6.948×10^{9}	5.497×10^{9}	2.264×10^{9}
LB_{inst}	0.943	0.951	0.952	0.939
Avg. IPC	0.698	0.670	0.693	0.505
IPC Balance	0.993	0.994	0.991	0.898
Avg. MIPS	1581.307	1517.855	1570.287	1143.466
Avg. L1 misses per 10^3 inst.	14.099	29.379	24.406	11.901
Avg. L2 misses per 10^3 inst.	0.073	0.223	0.195	1.756
Avg. Memory Bandwidth (MB/s)	14.854	43.335	39.115	257.040

Table 9.8.: GADGET clusters characterization

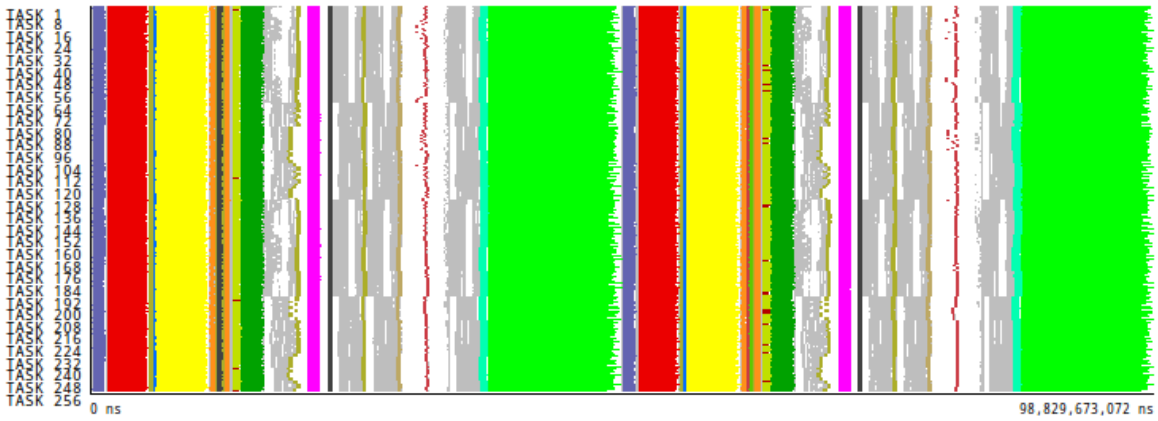


Figure 9.15.: GADGET time-line distribution of clusters

9.3.3. GADGET

Structure detection

Before examining the computational structure of GADGET it is interesting to mention that this application is the one that has the lowest parallelization efficiency in the set of the applications analysed, 0.582 (see Table 9.3). This low value is directly caused by the $CommEff$ value, 0.612, which expresses that the communications have an higher weight in the application execution time, more than in the rest of the applications analysed.

We can also see in Table 9.3 that the total number of clusters detected was 16, which represent 96.55% of the total computation time. The clusters with an aggregated time higher than 5% are 4 in this application, and represent a 75.50% of the total computation time. In Figures 9.14, 9.16, 9.15 and Table 9.8 we present the different outputs of the cluster analysis.

The refinement tree presented Figure 9.16 shows that these four major clusters appeared in the first iteration, using the smallest Eps value, so they do not contain sub-SPMD structures. In the scatter plot 9.14, these four clusters, 1 to 4, are the only ones that execute more than 2×10^9 instructions per burst. This high number of instruction executed per burst is also reflected in Table 9.8. In this table we can see that the average completed instructions per burst of these clusters is in range 2.264×10^9 and 1.844×10^{10}.

In terms of the load balance, all the clusters have values higher 0.9, a value in consonance of the overall application load balance of 0.951 (Table 9.3). We can advance that the application has a little margin of improvement by this side, so the impact of balancing the majors clusters will be minor as we will later see in the duration balancing experiment.

Regarding the sequential performance, the average IPC measured for the whole application is 0.649. Clusters 1, 2 and 3 have an average value close to 0.7, with minor variability across the different bursts (*IPC Balance* nearly 1). Cluster 4 has a lower IPC value, 0.505, and also the IPC balance is slightly lower, 0.898. This variability is observable in plot 9.14, where Cluster 4 (dark green) has a small horizontal distribution. In resume, the average IPC values obtained in these clusters suggest that the scenario of sequential improvement can be considered realistic, specially in case of Clusters 1 to 3, where the IPC is close to the target.

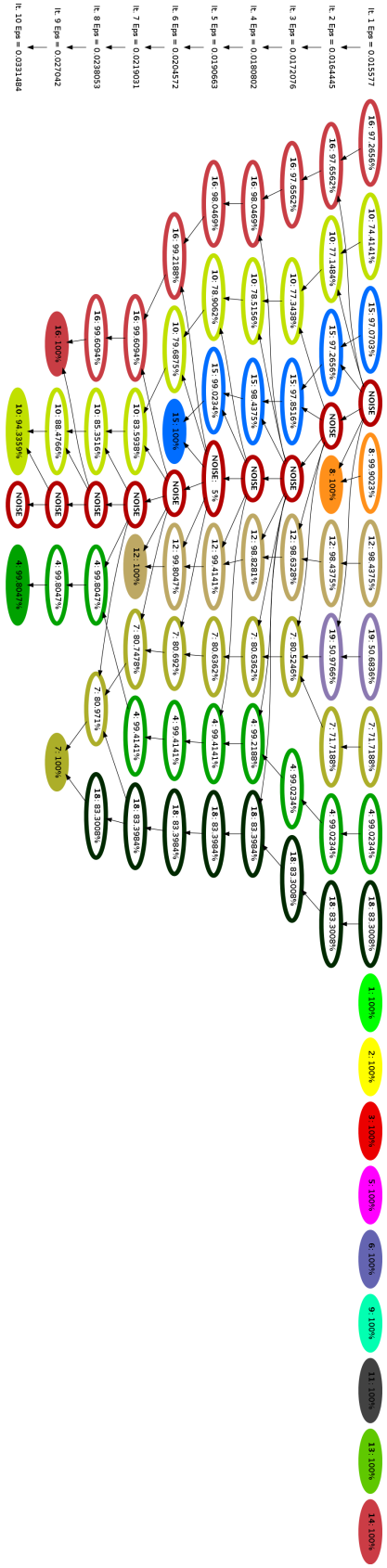


Figure 9.16.: GADGET refinement tree

Application improvement analysis

The results of the two application improvements simulations as well as the nominal simulation are presented in Table 9.9. The relative error of the nominal simulation is a little bit higher in this application, 12.260%. In Figure 9.17 we present the time-lines containing the clusters distribution of these three simulations.

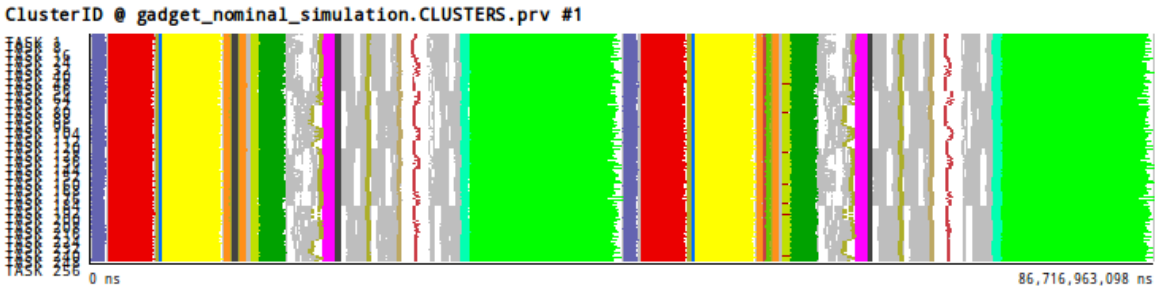
The total application load balancing is 0.951, and the load balancing of the four major clusters is always higher than 0.9. As a result, the speedup obtained in the duration balancing scenario is just 1.035. This simulation confirms that the good duration balancing values obtained offer small margin of improvement.

In the case of the sequential performance improvement scenario, we observe a speedup of 1.086, slightly higher than the obtained in balancing scenario. Time-line 9.17c presents the clusters distribution of of this simulation. In this case, we applied an average reduction around 25% to the bursts of clusters 1 to 4, according to the difference between their average IPCs and the desired IPC of 0.8. Taking into account that these clusters represent 75% of the computation of the application and the computation represents around 60% of the total application time (according to the parallelization efficiency), the minor gain observed in this scenario can be completely explained by the Amdhal's Law.

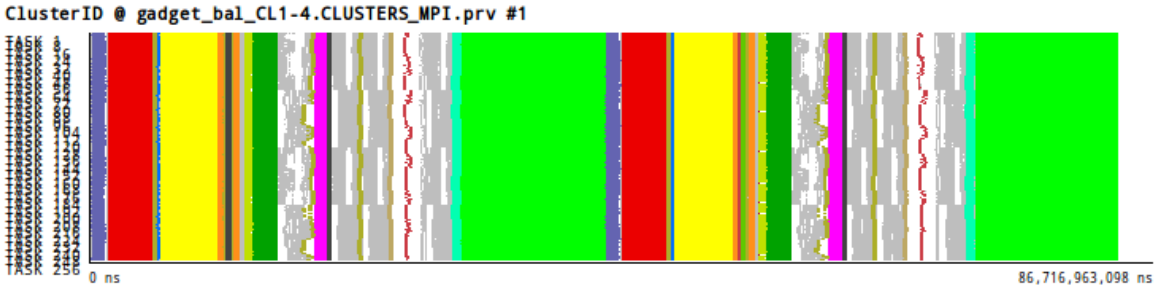
The conclusion we can extract of these two application improvements, is that in the computational part of this application have been well implemented, specially in terms of its balancing. In terms of its performance, the fact that its parallelization efficiency is low, would limit the benefits of the possible enhancements, so it would make more sense to tackle first how to improve the communications efficiency.

Configuration	Execution time (s)	Speedup
Nominal	86.717	–
Duration balancing	83.815	1.035
Algorithmic Improvement	79.885	1.086

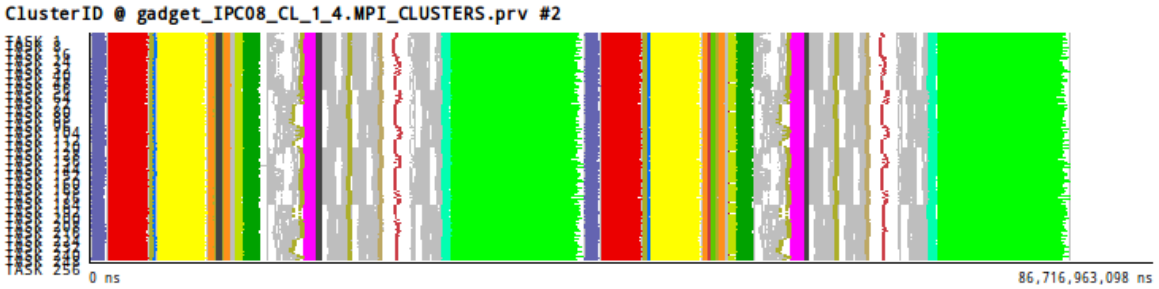
Table 9.9.: GADGET speedups of application improvement analyses



(a) Clusters time-line distribution of nominal simulation



(b) Clusters time-line distribution of duration balancing simulation



(c) Clusters time-line distribution of algorithmic improvement simulation

Figure 9.17.: Clusters distribution time-line of nominal simulation, duration balancing simulation and the algorithmic improvement of GADGET application

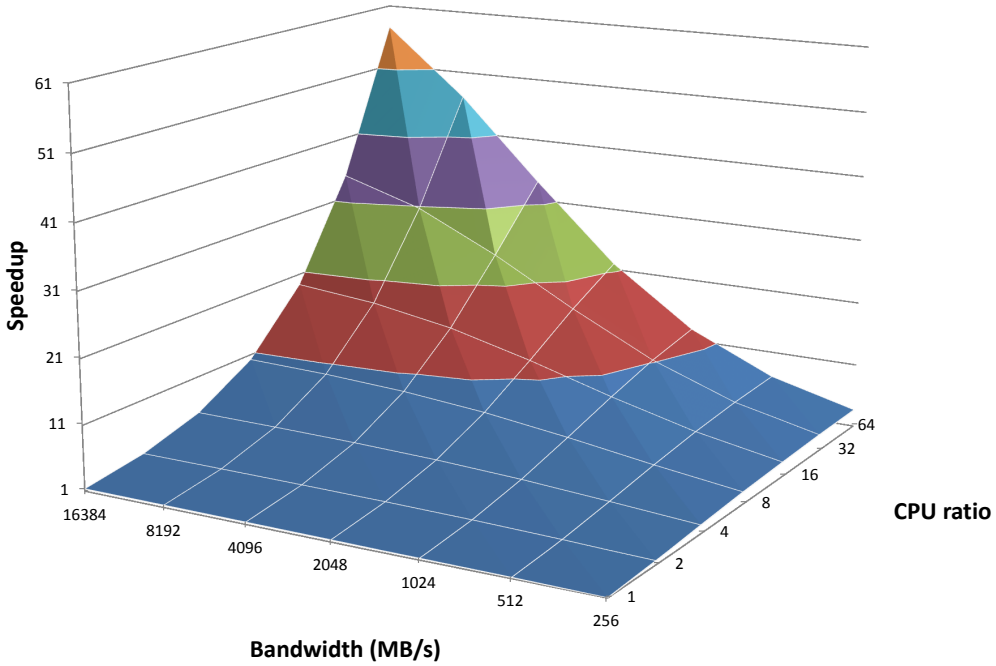
Use of new hardware analysis

Figure 9.18 contains the plots that express the simulated speedup when porting GADGET to new hardware configurations. Plot 9.18a presents the results of using general purpose CPUs and plot 9.18b presents the results of using acceleration hardware on the 4 selected clusters.

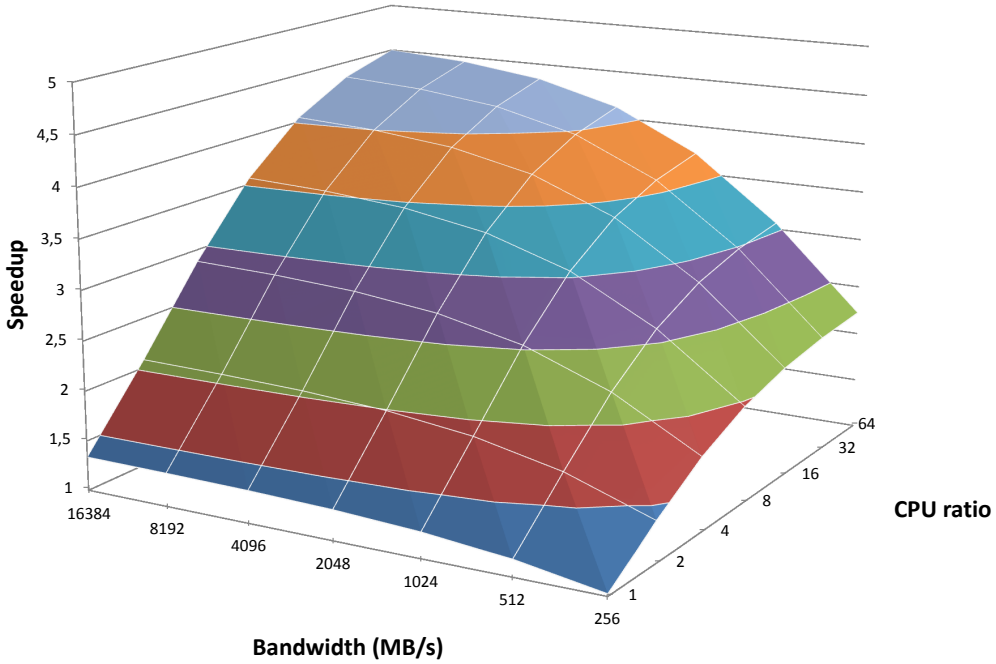
Observing the plot 9.18a, we can see how GADGET behaves when using faster general purpose CPUS. The maximum speedup observed in this plot, using a combination of CPUs 64 times faster and a network of 16 GB/s is around 61. In fact, we can observe a linear speedup when increasing the two parameters with the same proportion: for example, the speedup when using CPUs 8 times faster and network 8 times faster, 2GB/s is close to 8. Considering that we are just tuning the network bandwidth and not other network parameters, this linearity expresses that the communication part of GADGET is mainly dominated by the transference time of the communications it performs and not the number of messages transmitted or the possible serializations of small computations and communications.

On the other hand, the speedups obtained when simulating the port to a machine using accelerators results in minor speedups. In this case, the maximum speedup reached by the extreme configuration is around 4.5. Having seen that GADGET is mainly dominated by the transference time of the communications and the fact that the 4 clusters accelerated represent around the 15% of the computation time, an approximation to maximum speedup reachable in this acceleration speedup would be around 6 (in the extreme case), not so far from the actual value obtained. We can also see in this plot that the scalability of the application in this hardware configuration seems to be reaching a plateau around the extreme case, but we should simulate using larger parameter values to confirm this observation.

Finally, in Figure 9.19 we present the cluster time-lines obtained in the extreme cases in these two scenarios. As in previous applications, these two time-lines are not at the same scale. We can see for example, that the time-line obtained when using general purpose CPUs is pretty similar to the original execution time-line presented in Figure 9.15, but at a different scale. This proportional reduction reflects the previously mentioned linear scaling in this hardware configuration. On the other hand, the time-line obtained when using accelerators is totally different from the previous one, and the non-accelerated clusters and the filtered bursts now represent most of the application time.

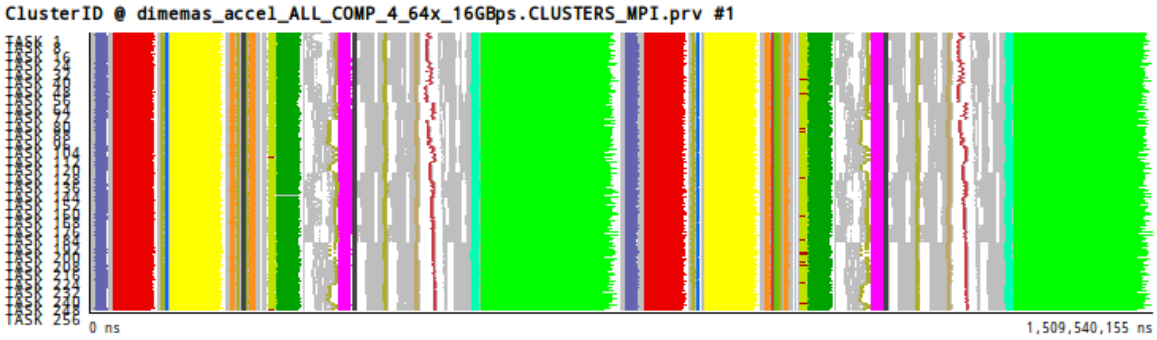


(a) Speedups using general purpose CPUs

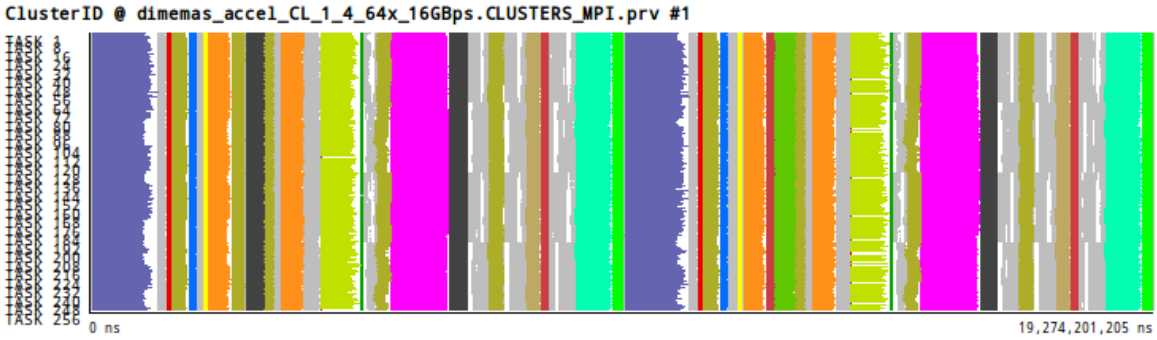


(b) Speedups porting the most consuming clusters to accelerators

Figure 9.18.: Results of the simulation when using GADGET in a different hardware



(a) Clusters distribution time-line using a general purpose CPU



(b) Clusters distribution time-line using an accelerator

Figure 9.19.: GADGET clusters time-lines using 16384 MB/s network bandwidth and general purpose CPU 64 faster and accelerator 64 faster

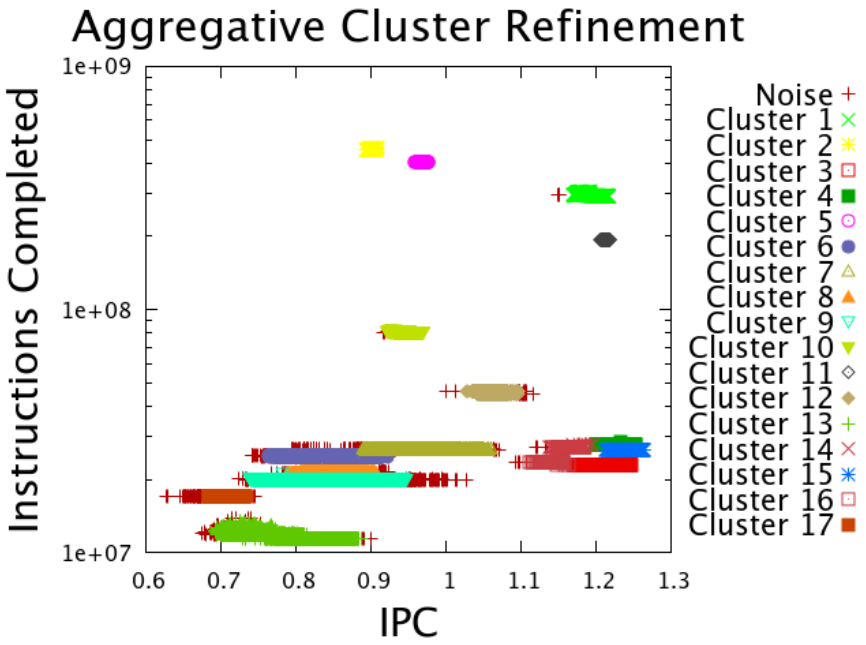


Figure 9.20.: SU3_AHiggs scatter plot of discovered clusters

Metric	Cluster 1	Cluster 2	Cluster 3	Cluster 4	Cluster 5
Application Computation Time (%)	31.48	8.05	7.43	7.14	6.61
Avg. Duration (ms)	109.235	223.198	8.250	9.915	183.294
LB_{dur}	0.993	0.988	0.989	0.991	0.990
Avg. Completed Inst.	2.957×10^{8}	4.566×10^{8}	2.284×10^{7}	2.769×10^{7}	4.024×10^{8}
LB_{inst}	0.993	0.991	1.000	0.998	0.999
Avg. IPC	1.194	0.902	1.219	1.231	0.969
IPC Balance	0.998	0.994	0.991	0.995	0.995
Avg. MIPS	2706.667	2045.860	2793.196	2195.468	1897.511
Avg. L1 misses per 10^3 inst.	1.151	3.492	0.255	0.345	3.766
Avg. L2 misses per 10^3 inst.	0.045	0.175	0.115	0.121	0.172
Avg. Memory Bandwidth (MB/s)	15.483	45.896	40.771	43.170	48.431

Table 9.10.: SU3_AHiggs clusters characterization

9.3.4. SU3_AHiggs

Structure detection

As opposite to the previous application, `SU3_AHiggs` is the application that obtain the best figures of the efficiency model presented in Table 9.3. In the reference execution using 256 tasks in the MareNostrum 2 supercomputer, this application obtained a value of 0.981 of the parallel efficiency, having a communications efficiency of 0.989 and a load balance of 0.992. SU3_AHiggs obtain this good figures by overlapping computation and communications efficiently.

In terms of the structure detection, we can also see in Table 9.3 using the Aggregative Cluster Refinement we detected up to 17 different clusters. These clusters represent 92.64% of the application computation time. On the other hand, the clusters that aggregate more that 5% of the total computation time are only 5, representing 60.70% of the computation time. In this way, a third of the total computation time is distributed along 12 clusters, which represent short computation phases. We can see the scatter plot of the resulting clusters in Figure 9.20 and the time-line of clusters distribution in Figure 9.21. In Table 9.10 we present the different statistics obtained by the 5 major clusters. In this application, we have omitted the refinement tree due its big size.

In the scatter plot, Figure 9.20, we can observe Clusters 1 (light green), 2 (yellow) and 5 (pink) appear in the top region, having the highest number of Instructions Completed, more than 2×10^8. These three clusters present a compact shape, and appear at different ranges of IPC. On the other hand, Cluster 3 (red) and 4 (dark green) appear in the bottom right region of the data space, in a region with more clusters. These two clusters execute around 2×10^7 and 3×10^7 instructions and present certain spread in terms of the IPC.

As we stated before, the load balance and average IPC of this application are very good. In the same direction, the major clusters also obtain good figures for these metrics. For example, in terms of the duration balancing, the overall application load balance is 0.992 (LB in Table 9.3). In Table 9.10 we see that the 5 "major" clusters this metric (LB_{dur}) is always above 0.991, and in the case of Cluster 3 it is nearly perfect (1.000 in the table, 0.9996 the actual measurement). These high values point that in terms of balance, there is no opportunity to improve the application.

In terms of performance, the application and the five major clusters also obtain good values. The average IPC of the overall application is 1.008, while the major clusters obtain values in the range 0.902 to 1.231. As these values represent a performance above the target we proposed for the sequential performance improvement scenario, in next section we measure a simulation where the improvement corresponds to other possible optimizations, for example a reduction in the total number of instructions of each of the major phases.

This characterization shows that SU3_AHiggs is the application that obtains the best results in terms of parallel efficiency, balancing and sequential performance (IPC) of the whole set analysed. This limits the application improvement scenarios proposed. In this case, the experiments regarding a possible porting to a new hardware will provide more interesting insight.

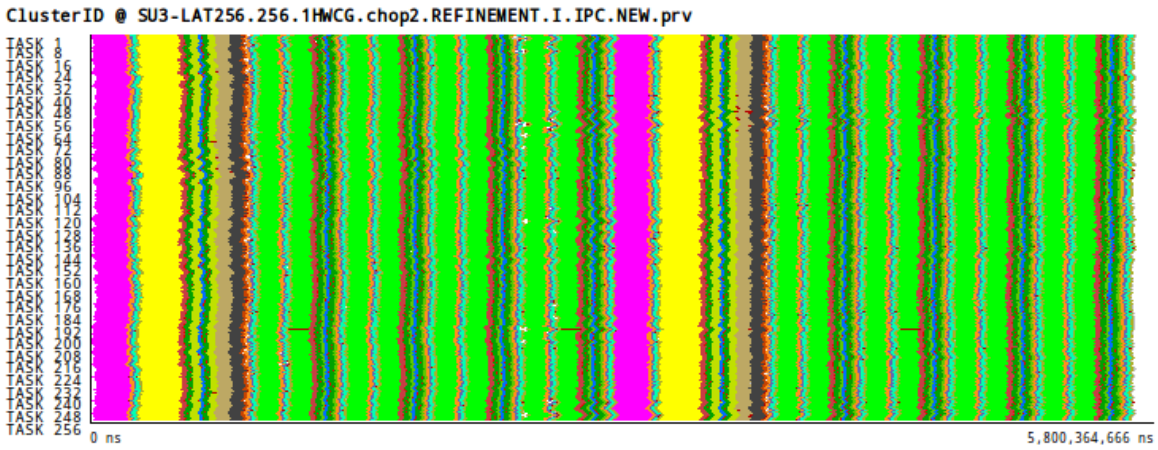


Figure 9.21.: SU3_AHiggs time-line distribution of clusters

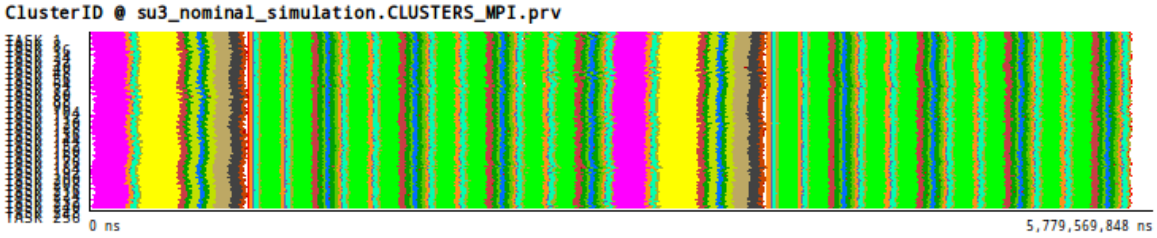
Application improvement analysis

As we have seen in the characterization of the application and the different SPMD regions found, SU3_AHiggs is an application well balanced with good performance figures. For this reason, the two application improvement scenarios will give us little information.

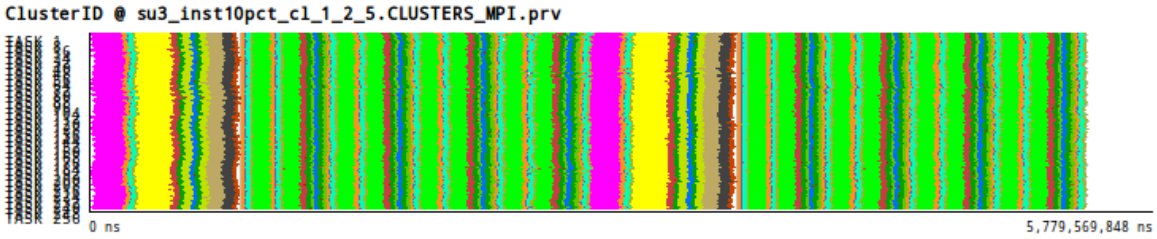
To make these analyses a little bit more illustrative, we slightly modified the simulations from the previous examples. In the case of the duration balancing scenario, we perform three different simulations: first, a duration balancing of the 5 major clusters selected; second, a balancing of all the computation (including the filtered bursts); and third, a balancing only of the filtered bursts. These three different simulations give us some indications to detect which part of the computation is more critical in terms of the duration balancing. In the case of the sequential performance improvement, instead of computing the ratio of the average IPC obtained by each cluster and 0.8, we compute a ratio based on a 10% reduction of the number of average instructions executed on these clusters whose average completed instructions is higher than 2.5×10^8: Clusters 1, 2 and 5 (see Table 9.10). In Table 9.11 we present the speedups obtained in the different simulations. In this case, the relative error of the nominal simulation vs. the real execution was 1.47%.

Configuration	Execution time (s)	Speedup
Baseline Simulation	5.780	—
Balancing all computation	5.755	1.0042
Balancing clusters 1 to 5	5.770	1.0016
Balancing filtered bursts	5.779	1.0002
10% instructions reduction Clusters 1, 2 and 5	5.540	1.0432

Table 9.11.: SU3_AHiggs speed-ups according to potential cluster improvements



(a) Clusters time-line distribution of nominal simulation



(b) Clusters time-line distribution of algorithmic improvement simulation

Figure 9.22.: Clusters distribution time-line of nominal simulation and the algorithmic improvement simulation of SU3_AHiggs application

In this table we can observe that the speedups obtained by the different duration balancing scenarios are minimum: 1.0042 when balancing all the computation part of the application, 1.0016 when balancing only the 5 major clusters and 1.0002 when balancing the filtered bursts. The measurements confirm the predicted small gain of balancing the application, but also show that just balancing the filtered bursts provide the smallest gain, which means that these small burst do not have a negative impact in the overall application. In this case, we do not present the simulation time-lines, as they do not give any interesting insight.

In terms of the sequential improvement, the speedup obtained is 1.043, a small gain. In Figure 9.22 we present the time-line of the nominal simulation and the result of this scenario, where we can see how Clusters 1, 2 and 5 have been shortened (by 10%) and the small reduction in the overall application duration. Considering that the new scenario introduced for this improvement, reducing the number of instructions of the Clusters 1, 2 and 5 could be hard to put in practice, this simulation also points that the pay off obtained by a big development effort is very low.

As a conclusion of these two experiments modelling potential application improvements, we can confirm the observations introduced previously: SU3_AHiggs is an application well implemented with a good balance and high performance that presents little opportunity to improve it.

Use of new hardware analysis

In Figure 9.23 we present the surface plots showing the speedups obtained by porting the application to a machine with faster general purpose CPUs and a faster network, plot 9.23a, and porting the application to a machine with acceleration hardware, plot 9.23b, where only

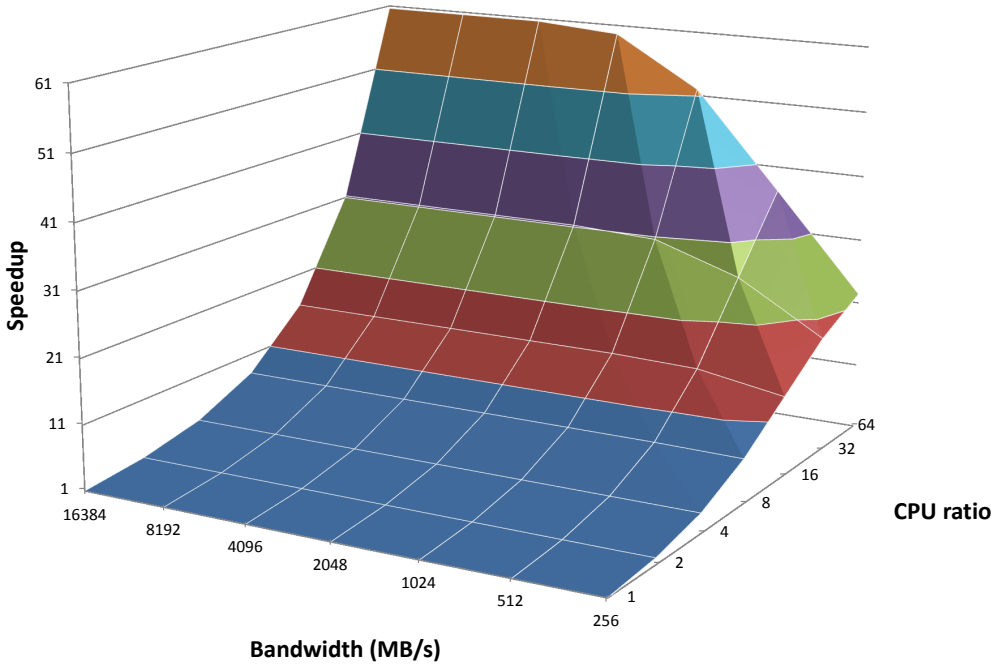
the 5 major clusters take advantage of the accelerators.

When using general purpose CPUs, plot 9.23a, we can observe that the maximum speedup reach is near 61, very close to the theoretical limit of 64. We discussed in previous applications that this limit could be reached if the network latency had no impact in the application.

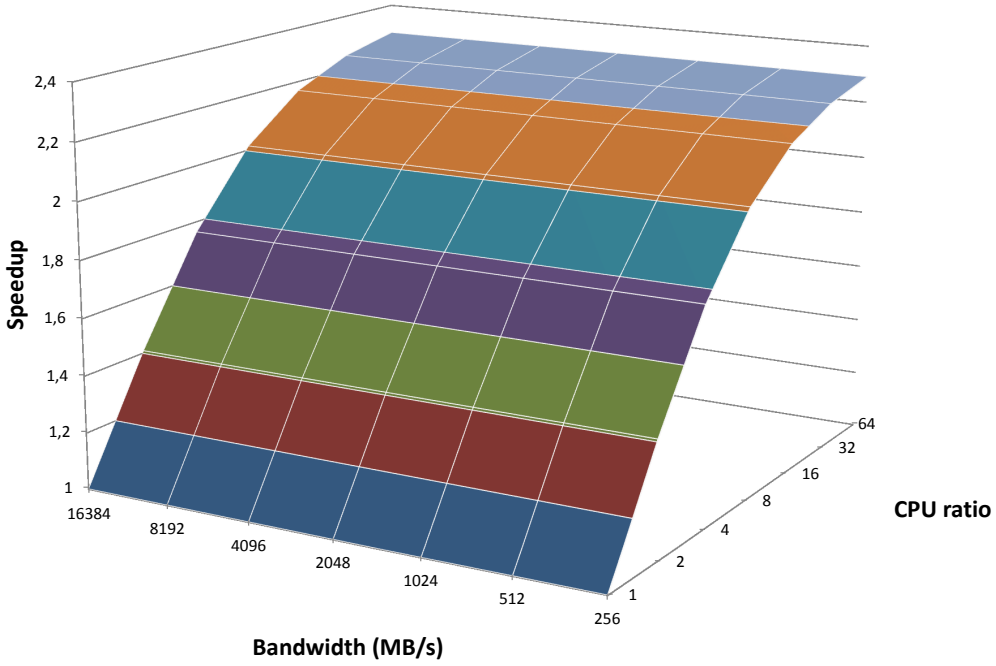
This plot shows an interesting situation regarding the scalability of the application. If we focus on the CPU ratio axis, we can see that the speedup obtained up to 16 times faster CPUs is always nearly lineal, independently of the network bandwidth. When the CPUs used are faster that 32 times the original one, the application requires some extra bandwidth to reach values near the linear speedup. This scalability simulations reflect that the way this application overlaps the communication and computation allows to reach high speedups without the need to scale the networking hardware at the same rate that the computation hardware.

In the second scenario defined, when using acceleration hardware, the speedups obtained are much lower than in the previous configuration. As can be seen in the plot 9.23b, the maximum speedup obtained by doing this port is just around 2.3. As it happens in the small CPU ratios of the previous scenario, when accelerating just the major clusters, the maximum speedup is reached independently from the network bandwidth, as the overlapping of computation and communications is able to hide the transmission times. In any case, this is a poor speedup, considering the potential effort of porting up to 5 different regions could be a hard task.

The time-lines of the extreme executions of these two scenarios are depicted in Figure 9.24. The time-lines do not share the time scale. Observing the one corresponding to the port to general purpose CPUs, 9.24b, we can see how it reflects the linear speedup measured: it is a clear re-scale of the structure observed in the real execution, Figure 9.21. Obviously, in the time-line 9.24b, corresponding to the porting to acceleration hardware, the structure reflects that the non-accelerated clusters are now the computational parts that drive the application execution. In this time-line we can also see some skew across the different tasks that affect the execution, specially in the first iteration.

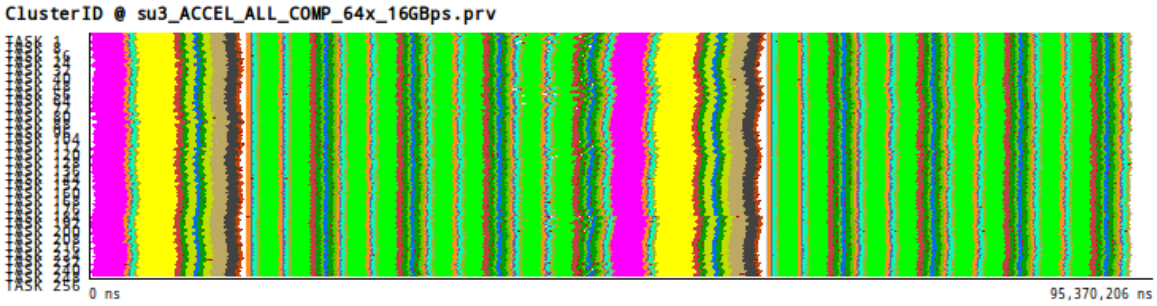


(a) Speedups using general purpose CPUs

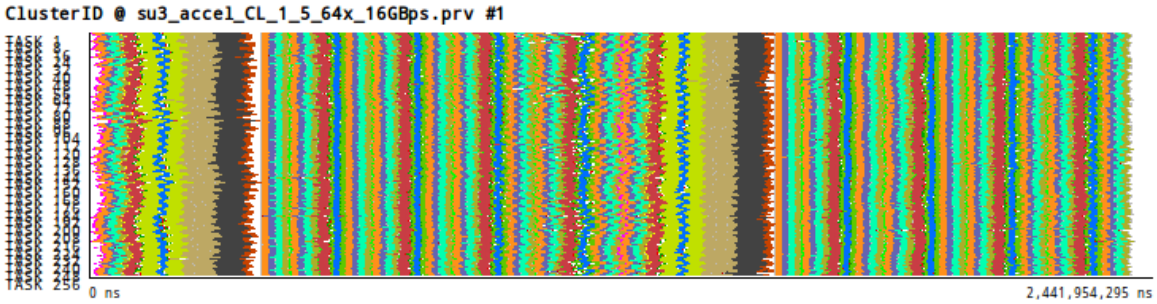


(b) Speedups porting the most consuming clusters to accelerators

Figure 9.23.: Results of the simulation when using SU3_AHiggs in a different hardware



(a) Clusters distribution time-line using a general purpose CPU



(b) Clusters distribution time-line using an accelerator

Figure 9.24.: SU3_AHiggs clusters time-lines using 16384 MB/s network bandwidth and general purpose CPU 64 faster and accelerator 64 faster

Conclusions

10. Conclusions

In this thesis we presented the research we conducted using cluster analysis algorithms and multiple sequence alignment algorithms in the parallel performance analysis scenario. As contributions to the Performance Analytics are, we proposed and validated new parallel analysis techniques that rely in these two types of algorithms. We also demonstrated the utility of the techniques introduced by using them to perform different types analyses of parallel applications.

10.1. Parallel applications computation structure detection based on cluster analysis

This first contribution, the core research of the whole thesis, explores the application of cluster analysis algorithms in the parallel performance analysis scenario. Even the use of cluster algorithms in the performance analysis is not a novelty by itself, the way we applied differs totally from the previous approaches we found in the area.

Firstly, we defined the minimum analysis granularity we work with, the *CPU burst*. A CPU burst, or simply burst, is the region in a parallel application between calls to the parallel runtime (for example, before sending a message). This granularity is intrinsically related to the application implementation, but it is orthogonal from the semantical view of the application, i.e. how it is have been programmed. We demonstrated that the CPU bursts constitute an useful granularity in the application analysis.

The cluster analysis is applied then to determine the groups of CPU bursts that behave similarly according to a given performance metrics. In this case, the metrics we used were the performance hardware counters, registers offered by processors that account low level performance characeristics, such as the instructions executed or the cycles spent to execute them. In this way, hardware counters suppose an unique piece of information to observe and measure the application performance.

Using the hardware counters as input metrics of the cluster analysis made us explore different cluster algorithms to find the more appropriate cluster algorithm. We observed that the values of these metrics do not follow a fixed distribution, so the classical *k-means-like* algorithms previously used in the performance analysis did not produce good results. As an alternative, we found that density-based cluster algorithms perfectly fulfil our requirements as this family of algorithms do not assume any structure of the data. The selected density-based cluster algorithm was DBSCAN, the most representative of this family. The DBSCAN algorithm uses two parameters, *Epsilon* (or *Eps* for short) and *MinPoints*. The resulting clusters are the different subsets of the data space with more than MinPoints individuals, where all points have at least one neighbor at distance less than Eps. The points which do

not fall in any cluster are marked as noise. Conceptually, this definition determines that the cluster of the data set are those groups "close enough" (Eps parameter) and "dense enough" (MinPoints parameter).

As a result of applying DBSCAN to the performance hardware counters data, especially the combination of Completed Instructions and IPC, we obtain a clear characterization of the computation structure of the application at detail. The resulting clusters represent the different computation regions found which are related in terms the performance hardware metrics used. These clusters, usually a small number, are characterized in terms of different performance metrics present in the input data, not just the counters used to generate thenm, and result very useful to understand how the different computation regions of an application behave.

10.2. Evaluation of the computation structure quality

Knowing that the density based cluster does not assume any model in the resulting clusters, the quantitative evaluation of the results obtained by applying DBSCAN became a hard manual task.

In order overcome this problem, we introduced the Cluster Sequence Score that quantitative evaluates the *SPMDiness* of a parallel application. The SPMDiness of an application is informally described as how well its computation structure follows the SPMD pattern.

The underlying idea to perform the quantification of the *SPMDiness* is that if in an SPMD application all tasks should be performing the same computations with different data at same time, the sequence of these computations should be the same for all tasks. Then, representing this sequence of computations as DNA/protein chains, we found that Multiple Sequence Alignment (MSA) software will be totally applicable in this context. Using an MSA algorithm to a given set of tasks sequences will produce a alignment, i.e. an disposition of the given tasks that maximize the number of equal actions that appear at on each position of the sequences. Thanks to this alignment it is easy to quantify the similarities across the tasks sequences, the desired SPMDiness quantification.

The Cluster Sequence Score we proposed relies in a computation structure based on cluster analysis. In this way, the sequences we use in the MSA algorithm are the sequence of different clusters of each task involved in the parallel application analysed. After applying the MSA algorithm, our implementation traverse the aligned sequences in parallel measuring for each position the clusters that appear simultaneously in all tasks. As a result, we obtain the *Score per cluster*, in other words, how well each cluster represents a SPMD region. A weighted combination of all the scores per cluster using the percentage of the application computation time each cluster represents produces the mentioned Cluster Sequence Score, i.e. the SPMDiness of computation structure detected.

10.3. Automatization and refinement of the structure detection

After acquiring some expertise with the structure detection technique based on DBSCAN algorithm we detected the limitations of this algorithm. First, the DBSCAN parameters could be a handicap for an non-expert user and second, and more important, for those inputs with different densities across the data space, the use of a single Eps parameter limit the ability to correctly detect the actual clusters where the dataset presents multiple densities. We proposed the Cluster Aggregative Refinement algorithm to overcome these problems.

Aggregative Cluster Refinement was inspired by the *X-Means* algorithm, which iteratively improves the quality of the classic *K-Means* algorithm, using the Bayesian Information Criterion to evaluate the resulting clusters. The proposed algorithm also takes in advantage the common points of DBSCAN and the hierarchical clustering methods. These methods basically create a dendrogram, a tree that expresses the similarities between individuals and groups of individuals. In a dendrogram the root represents the whole data set, the leaves the individuals, and the height between each intermediate node and the leaves represents the value of a given distance metric applied to the individuals it joins. In DBSCAN, using a fixed MinPoints value, the different higher an lower Eps values applied correspond to higher (close to root) or lower (close to leaves) "cuts" in a dendrogram, respectively.

Aggregative Cluster Refinement receives as inputs the data set to analyse and the maximum number of refinement iterations it can execute, N. Then it sets a value of MinPoints and generates N increasingly sorted Eps values by analysing the data distribution. On each iteration it applies DBSCAN using $MinPoints$ and the corresponding Eps to the input data set, evaluating the resulting clusters using a quality score. Those individuals that belong to clusters that pass the score are discarded in further iterations. The algorithm finishes after N iterations or when all clusters detected passed the score.

The use of multiple Eps values plus the quality score imply that final clusters will appear at different iterations of the algorithm, according to the density they have in the data space, overcoming the DBSCAN limitation.

Using the Aggregative Cluster Refinement with the CPU burst data of a parallel application, and applying the Cluster Sequence Score to evaluate the clusters on each iteration, we provide the developer/analyst a high quality SPMD computation structure detection with the added value of the (pseudo-)dendrogram that reflects the hierarchical formation of the computation regions. In addition, the algorithm just requires the data set as input, but no other parameters.

10.4. Structure detection in practice

In the rest of the thesis, we presented several uses of the computation structure detection based on cluster analysis.

10.4.1. Accurate extrapolation of performance metrics

A common problem when extracting performance data is the limit on the different metrics that can be read simultaneously. For example, when reading the performance hardware counters, the number of register that contain them are fixed in the hardware design, and also the combinations of the different counters. In order to avoid the run multiple executions of the parallel application, which are costly in terms of time and resources, it is quite normal to multiplex the counters selected during the application

We defined an extrapolation methodology where we apply the computation structure detection based on cluster analysis using the data extracted by multiplexing the performance counters groups. Using the methodology, we are able to extrapolate, for each cluster, the averages values of all the performance counters present more groups than can be read simultaneously, with minimum error. To obtain this correct averages we need to guarantee that all counters groups have a subset of counters in common, in order to apply the cluster analysis to this subset; we also need that the resulting clusters contain bursts with values for all counters groups. First requirement is solved by the fact that completed instructions and total cycles counters are present in almost all of counters defined in all processors. The second requirement is easily assumable due to the different multiplexing schemes we propose.

10.4.2. Information reduction in a multi-scale simulation

Detailed simulations of large scale message-passing parallel applications are extremely time consuming an resource intensive. We designed and validated a methodology where the computation structure detection based on cluster analysis is combined with signal processing techniques to reduce the volume of simulated data by various orders of magnitude. This reduction makes possible to perform accurate predictions full application time in reasonable time.

The information reduction process starts using a spectral analysis technique. In a *naïve* way, the spectral analysis represents a performance metric, for example the aggregated duration of all the CPU bursts at a given point of time, as time varying signal. Using the wavelet transform, it detects the regions with high-frequencies which determine the main iterative part of the application and also the iterative pattern. Selecting just one or two representative iterations, it reduces dramatically the data to analyse while being representative of the application behaviour. To this subset of the input data, we apply the cluster analysis to the CPU bursts to characterize the inner computation structure. The combination of these two analyses can be seen as an automatic detection of the computation trends in two granularity levels: main application iterations and intra-iteration computation phases.

Having this characterization, the (expensive) micro-architectural part of the simulation is dramatically reduced by carefully selecting the actual CPU bursts to simulate using the information provided by the cluster analysis. The strong statistical foundation of the cluster analysis guarantees that the use of a small set of representatives instead of all individuals adds a minimum error to the final simulation.

10.4.3. Parallel applications what-if studies

Using the fine-grain characterization of the computation regions the Aggregative Cluster Refinement algorithm provides we carried out a set of *what-if* studies to a set of four application. In these studies we measure what would the impact of applying different application improvements and how would the performance by porting the applications to different hardware. The results of these studies provides on a valuable insight about the applications behaviour, for example to take decisions on where put the programming resources to maximize the pay off in terms of performance, without the need to recodify the application, and also to anticipate the potential problems that application could suffer when using better hardware resources, without requiring the actual hardware.

10.5. Open lines for future research

The scale of current supercomputers and the demands of scientists from almost all fields to use them make the Performance Analytics a key area to research. This thesis represents our proposal into this area, focusing on the better understanding of the computation regions of parallel applications. We consider that the computation regions characterization opens the opportunity to develop a wide variety of studies, using the techniques presented to in sort of manners not contemplated in thesis and also improving the techniques themselves.

To resume, we want to focus on the four open lines that seem more interesting in short term: the scalability of the cluster analysis, the exploration of the metrics space to maximize the clustered information, a deep study of the clusters structure, and the design and validation of a diagnosis system using the structure detection.

10.5.1. Scalability of cluster analysis

As well as cluster analysis targets is to summarize the information present on a datasets, in some cases the cluster algorithms cost could be prohibitive when dealing with huge amounts of data. This is the case of the DBSCAN cluster algorithm. In this thesis we have not emphasize a formal cost analysis of both DBSCAN and the Aggregative Cluster Refinement, but we can affirm that they are costly. The foundation of these algorithms is a "epidemic" search of neighbours along the data space, an operation of order $O(n^2)$, with an average cost of $O(n \log n)$. This is extremely expensive if we consider that current supercomputers have the order of hundreds of thousands (or millions) cores, which could produced performance data at sustained rates of thousands of metrics values per second.

Our main concern with respect to this issue is that it would be critical to improve the cluster algorithm so as to guarantee that the presented techniques will be applicable in current and future scenarios of ever-growing supercomputers. To tackle this problem, our target is to parallelize the clustering algorithm itself. Having seen some other approaches in the literature, our proposal consists of distribute the analyses in a reduction network, where the leaves correspond to parallel DBSCAN executions each of them with a subset of the data. Then, the reduction consists on merge the models obtained on each of these "local clusterings", generate a final model and redistribute it to the leaves to perform a final classification.

The parallelization exposed represents and interesting starting point to research, for example to evaluate the possible ways to model the "local clustering" results (remember that DBSCAN clusters do not follow any data distribution), how merge the local models efficiently and finally how to dispose the final classified data. As a result, having a parallel implementation of DBSCAN will let us to perform analyses of bigger scales.

10.5.2. Fine-tune of the structure refinement

The Aggregative Cluster Refinement algorithm we presented relies in an accurate selection of different Eps values and the application of the Cluster Sequence Score to evaluate the resulting clusters. We demonstrated its usefulness by presenting how it is able to detect the fine-grain SPMD phases of different applications. However, we detected that in certain applications the resulting clusters could be improved.

We observed two different situations were the algorithm do not produced the desired results. The first one occurs when two clusters are close to the target score, but in the next iteration of the algorithm they got merged, producing a over-aggregation. The second one occurs when there is a cluster with high variability and some of bursts it should contain are classified as noise, even using the highest Eps values computed.

As a future work, we want to improve the Aggregative Cluster Refinement algorithm to be robust to these two situations. To do so, we require two modifications in the algorithm: to avoid the first problem, we have to adapt the evaluation of the clusters by reducing slightly the target score required and then classifying the bursts on each iteration; to avoid the second problem, we have to tune the exploration of the different Eps values.

10.5.3. Metrics space exploration

Along this thesis we have introduced different techniques to detect and evaluate the computation structure of a parallel application. Essentially, all these techniques focus on a precise characteristic of the applications analysed, and also of they computation structure: the SPMD paradigm. The Cluster Sequence Score and it use on the Aggregative Cluster Refinement has as main target the detection the application's SPMD phases at fine-grain.

To make this detection possible, the selection of the metrics used by the cluster analysis plays a crucial role, and we demonstrated that the combination of Completed Instructions and IPC counters produced a good characterization. Almost all experiments presented use these pair of metrics, showing the desired SPMD patterns, as well as the possible imbalances appearing in some regions. However, in some cases, other combinations of performance counters lead to a better detection. For example, in the analysis of WRF in chapter 9, we used Completed Instructions and Main Memory Accesses.

Considering this, we propose as a future work the exploration of which of the metrics that can be used to characterize a CPU bursts detects the best computation structure. These metrics will not only be limited to the performance counters but also any other performance metric that can be extracted during the application execution, for example the code location. The exploration itself will require the use of cluster validity indexes, such as the Cluster Sequence Score, to evaluate the quality of the clusters detected. In addition, we will require

other statistical techniques, such as the Principal Components Analysis (PCA), to guide the metrics combinations.

10.5.4. In-depth analysis of the clusters structure

Appart from determining which metrics combination, in this third proposal, we focus on the understanding of the clusters structure.

We have seen that the *refinement tree* produced by the Aggregative Cluster Refinement define a hierarchy in the points that belong to a cluster. We studied the relationship between this hierarchical formation and the target SPMD structure we want to discover.

In terms of analysis, it would be really useful to characterize each of these *sub-clusters* that appear at different levels of the refinement tree and finally conform an SPMD. The study of the possible relationships of performance counters metrics observed along this hierarchy will provide an unique insight of the internal behaviour of each SPMD phase. For example, with this information we will be able to detect not only the imbalances in these phases, but also to quantify the possible causes of them.

10.5.5. Detailed performance data extrapolation

The performance data extrapolation method we presented in chapter 7 we are able to compute the average of a wide set of more performance counters that the ones that can be read simultaneously when extracting the data by using different multiplexing schemes. These averages values are useful to obtain an initial characterization of each cluster, but as it happens in the application profiles, they can hide some variability from different occurrences of the same cluster.

A fourth proposal for a future research will consist on analysing how the different metrics multiplexed behave in the different CPU bursts used in the application execution, so as to study accurately its variability and distribution. In this way, the result of this future extrapolation method will tell more information about the extrapolated metrics, not just the average, and not hiding the variability that the different metrics can express.

In addition to the study of the extrapolated metrics distribution, it would be also interesting to add a new multiplexing schemes by using a random permutation of the performance counters groups, both in time and space. The random permutation of the groups multiplexed will be useful to avoid possible systematic errors if exists a correlation between the application computation structure and the multiplexing schemes we used.

Appendices

A. The BSC Tools Parallel Performance Analysis Suite

The BSC Tools is a suite of parallel performance analysis tools, now developed in the Performance Tools Group of the Barcelona Supercomputing Center (BSC-CNS), with more than 20 years of expertise.

The BSC Tools suite focuses on the post-mortem analysis of application traces. The big picture these tools ecosystem is presented in the Figure A.1. In this appendix we include a detailed description of the most important elements of it includes.

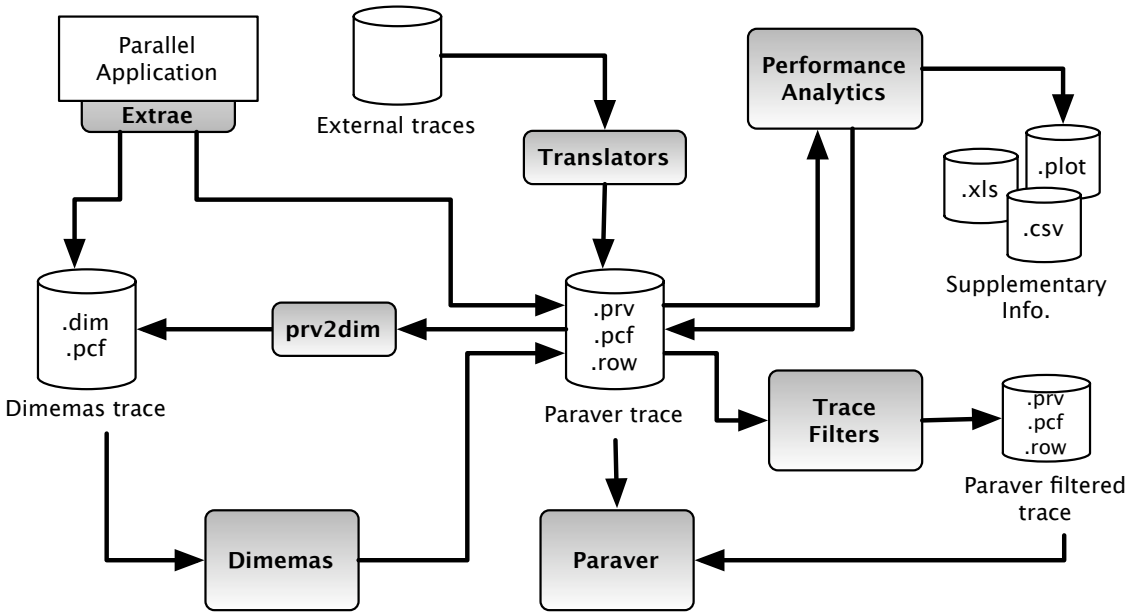


Figure A.1.: Scheme of the BSC Tools Parallel Performance Analysis Suite, depicting the different tools and their interaction

A.1. Extrae

Extrae is the package devoted to generate tracefiles which can be analyzed later by Paraver or Dimemas. Extrae is a tool that uses different interposition mechanisms to inject probes into the target application so as to gather information regarding the application performance. The Extrae instrumentation package can instrument the following parallel programming models: MPI, OpenMP, CUDA, pthread and OmpSs.

In order to specify and control of what and how to trace the application, Extrae can be configured through an XML file.

A.1.1. Interposition mechanisms

Extrae takes advantage of multiple interposition mechanisms to add monitors into the application. No matter which mechanism is being used, the target is the same, to collect performance metrics at known applications points to finally provide the performance analyst a correlation between performance and the application execution. Extrae currently uses the following interposition mechanisms:

DynInst

DynInst [27] is an instrumentation library that allows modifying the application by injecting code at specific code locations. Specfically, Extrae uses the application binary rewriting feature DynInst offers.

With the instrumentations capabilities of DynInst, Extrae can instrument different parallel programming runtimes as OpenMP (either for Intel, GNU or IBM runtimes), CUDA accelerated applications, and MPI applications. DynInst also offers Extrae the possibility to easily instrument user functions by simply listing them in a file.

Linker preload (`LD_PRELOAD`)

Most of the current operating systems allow injecting a shared library into an application before the application gets actually loaded. If the library that is being preloaded provides the same symbols as those contained in shared libraries of the application, such symbols can be wrapped in order to inject code in these calls. In Linux systems this technique is based on defining the `LD_PRELOAD` environment variable according to the interposition library we want to use.

Extrae contains substitution symbols for many parallel runtimes, as OpenMP (either Intel, GNU or IBM runtimes), pthread, CUDA accelerated applications, and MPI applications.

Additional instrumentation mechanisms

Extrae also takes the advantage of some parallel programming runtimes that have their own instrumentation (or profile) mechanisms available for performance tools. The most widely known example is the Message Passing Interface (MPI) which provides the Profile-MPI (PMPI) layer. MPI/PMPI use a linker capability to define weak and strong symbols. While the weak symbols can be overloaded, the strong symbols can't and still provide the functional part of MPI. Such mechanism is used by linking the application with a library that contains symbols that substitute the weaks symbols. In addition, this profiling interface is also used by the linker preload mechanism.

An additional instrumentation mechanism is CUPTI, which is an interface of the CUDA library that can install callback routines that are executed when relevant parts of the code have been executed in the CUDA application.

Some compilers allow instrumenting the application routines by using special compilation flags at the compilation and link phases. Applying such flag typically tells the compiler to call some predefined callbacks that can be hooked by performance tools like Extrae.

The programming model OmpSs provides instrumentation version of its compiled binaries. The instrumentation is the synergy of both BSC teams (performance tools and programming models) to facilitate and integrate the performance analysis into this successful parallel programming model. OmpSs compiler (Mercurium) can inject code into the application and the runtime (Nanos++) emits information referring its internal evolution. This results in Paraver tracefiles that not only contain information regarding the application evolution but also more specific information about the parallel programming model.

Extrae API

Finally, Extrae gives the user the possibility to manually instrument the application and emit its own events if the previous mechanisms do not fulfill the user's needs. The Extrae API is detailed in the Extrae user-guide documentation that accompanies the package.

A.1.2. Sampling mechanisms

Extrae does not only offer the possibility to instrument the application code, but also offers to use sampling mechanisms to gather performance data. While adding monitors into specific location of the application produces insight which can be easily correlated with source code, the resolution of such data is directly related with the application control flow. Adding sampling capabilities into Extrae allows providing performance information of regions of code which has not been instrumented.

Currently, Extrae sports two different sampling mechanisms. The first mechanism is the old-known signal timers, which fires the sampling handler at a specified time interval. The second sampling mechanism uses the processor performance counters to fire the sampling handler at a specified interval of performance events triggered. While the first mechanism can provide totally uncorrelated samples with the application code, the second mechanism, using the appropriate performance counters, can provide insight of the application but still presenting some correlation with the application code/performance.

A.1.3. Performance data gathered

The monitors added by Extrae gather different types of information. Depending on the monitor placement, each monitor can be taught to gather specific information. The most common information gathered is:

Timestamp When analyzing the behavior of an application, it is important to have a fine-grained timestamping mechanism (up to nanoseconds). Extrae provides a set of clock functions that are specifically implemented for different target machines in order to provide the most accurate possible timing. On systems that have daemons that inhibit the usage of these timers or that do not have a specific timer implementation, Extrae

uses advanced POSIX clocks to provide nanosecond resolution timestamps with low cost.

Performance and other counter metrics Extrae uses the PAPI and the PMAPI interfaces to collect information regarding the microprocessor performance. With the advent of the components in the PAPI software, Extrae is not only able to collect information regarding how is behaving the microprocessor only, but also allows studying multiple components of the system (disk, network, operating system, among others) and also extend the study over the microprocessor (power consumption and thermal information).

Extrae mainly collects these counter metrics at the parallel programming calls and at samples. It also allows capturing such information at the entry and exit points of the user routines instrumented.

Reference to the source code Analyzing the performance of an application requires relating the code that is responsible for such performance. This way the analyst can locate the performance bottlenecks and suggest improvements on the application code. Extrae provides information regarding the source code that was being executed (in terms of name of function, file name and line number) at specific location points like programming model calls or sampling points.

A.2. Paraver

Paraver is a flexible parallel program visualization and analysis tool based on an easy-to-use GUI. Paraver was developed responding to the basic need of having a qualitative global perception of the application behaviour by visual inspection and then to be able to focus on the detailed quantitative analysis of the problems. Paraver provides a large amount of information on the behaviour of an application. This information directly improves the decisions on whether and where to invest the programming effort to optimize an application. The result is a reduction of the development time as well as the minimization of the hardware resources required for it.

Some Paraver features are the support for

- Detailed quantitative analysis of program performance
- Concurrent comparative analysis of multiple traces
- Mixed support for message passing and shared memory (networks of SMPs)
- Flexible personalization of the semantics of the visualized information

One of the main features of Paraver is the flexibility to represent traces coming from different environments. Traces are composed of state intervals, events and communications with an associated timestamp. These three elements can be used to build traces that capture the behaviour along time of very different kinds of systems.

A.2.1. Analysis views

Paraver offers a minimal set of types of views of a trace. The philosophy behind the design was that different types of views should be supported if they provide qualitatively different analysis types of information. Frequently, visualization tools tend to offer many different views of the parallel program behaviour. Nevertheless, it is often the case that only a few of them are actually used by users. The other views are too complex, too specific or not adapted to the user needs.

Paraver differentiates two types of views:

- Time-line View: to represent the behaviour of the application along time in a way that easily conveys to the user a general understanding of the application behaviour. It also supports detailed analysis by the user such as pattern identification, causality relationships, etc.

- Histogram View: to provide quantitative measurement of the information present in the trace discarding the time component. The Histogram View provides an intuitive way of accounting the information available in the trace, offering a wide set of statistics and the possibility to easily combine different metrics. It eases the detection of correlations of the performance metrics. In addition, the Histogram View is connected to Time-line View, to observe how the information it shows is distributed along time.

Time-line View

The Time-line View is flexible enough to visually represent a large amount of information and to be the reference for the quantitative analysis. The Paraver view consists of a time diagram with one line for each represented object. The types of objects displayed by Paraver are closely related to the parallel programming model concepts and to the execution resources (processors).

In the first group, the objects considered are: application (*Ptask* in Paraver terminology), task and thread. Although Paraver is normally used to visualize a single application, it can display the concurrent execution of several applications, each of them consisting of several tasks with multiple threads.

The information in the Time-line View consists of three elements: a time dependent value for each represented object, flags that correspond to puntual events within a represented object, and communication lines that relate the displayed objects. Figure A.2 displays a basic time-line where the coloring correspond to the value of the state of each task, events are represented as small green flags and point-to-point communications are presented as yellow lines.

A Visualization Module blindly represents the values and events passed to it, without assigning to them any pre-conceived semantics. This plays a key role in the flexibility of the tool. The semantics of the displayed information (activity of a thread, cache misses, sequence of functions called,...) lies in the mind of the user. Paraver specifies a trace format but no actual semantics for the encoded values. What it offers is a set of building blocks, the Semantic Module, to process the trace before the visualization process. Depending on

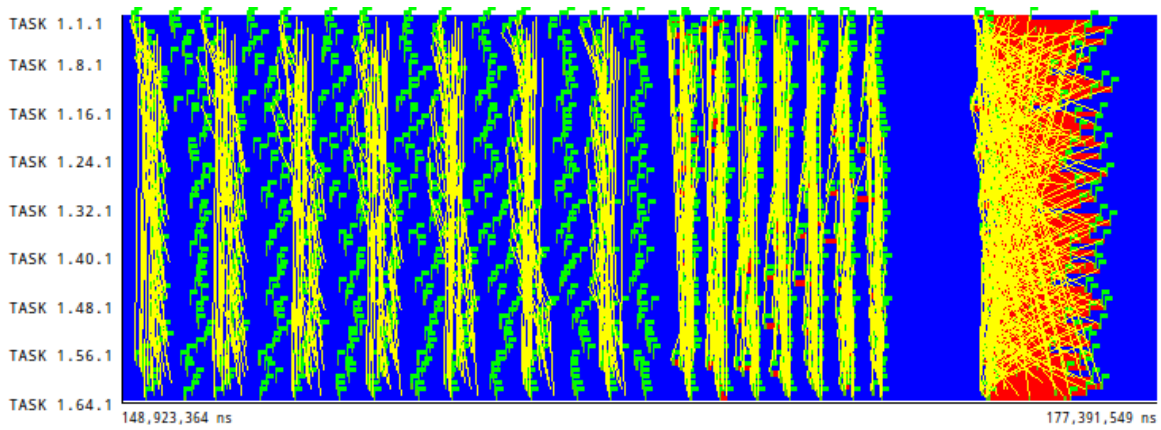


Figure A.2.: Basic Paraver time-line view

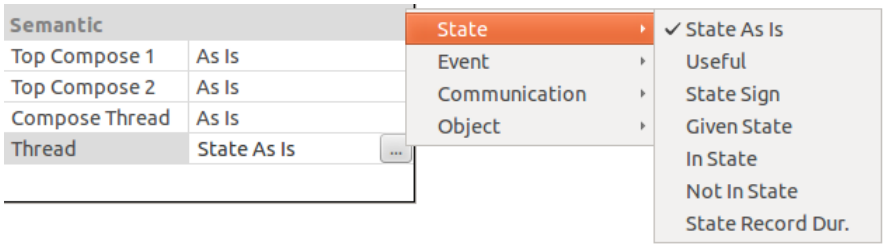
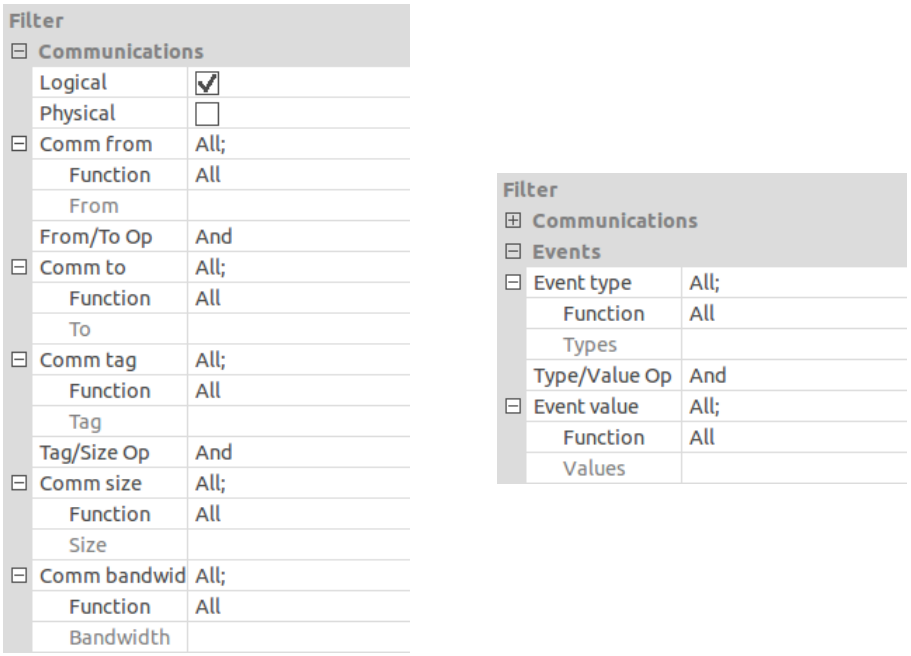


Figure A.3.: Detail of the Paraver Semantic Module showing the semantic functions regarding thread states

how you generate the trace and combine the building blocks, you can get a huge number of different semantic magnitudes.

The separation between the Visualization Module which controls how to display data and the Semantic Module which determines the value visualized offers a flexibility and expressive above than frequently encountered in other tools. Paraver Semantic Module is structured as a hierarchy of functions which are composed to compute the value passed to the visualization module, see Figure A.3. Each level of function corresponds to the hierarchical structure of the process model on which Paraver relies. For example: when displaying threads, a thread function computes from the records that describe the thread activity, the value to be passed for visualization; when displaying tasks, the thread function is applied to all the threads of the task and a task function is used to reduce those values to the one which represents the task; when displaying processors, a processor function is applied to all the threads that correspond to tasks allocated to that processor.

Many visualization tools include a filtering module to reduce the amount of displayed information. In Paraver, the Filtering Module, Figure A.4 is in front of the semantic one. The result is added flexibility in the generation of the value returned by the Semantic Module.



(a) Communications filter

(b) Events filter

Figure A.4.: Communication and Event sections of the Paraver Filtering Module

The Histogram View

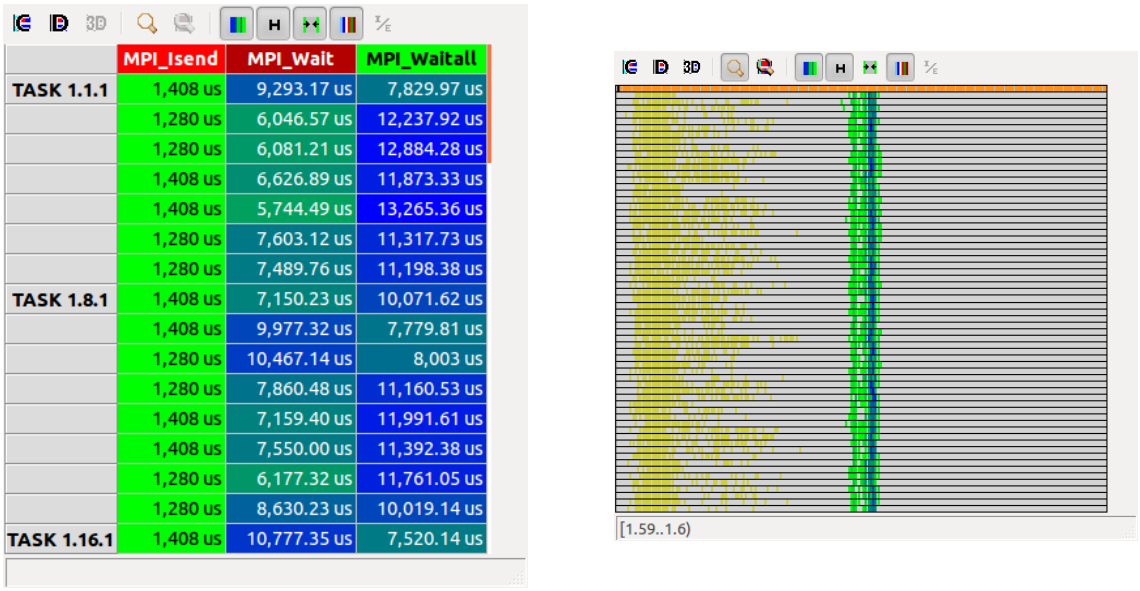
A global qualitative display of application behaviour is not sufficient to draw conclusions on where the problems are or how to improve the code. Detailed numbers are needed to sustain what otherwise are subjective impressions.

The quantitative analysis module, the Histogram View, of Paraver offers the possibility to obtain information on the user selected section of the visualized application and includes issues such as being able to measure times, count events, compute the average value of the displayed magnitude, etc.

The Histogram View functions are also applied after the Semantic Module in the same way as the Visualization Module. Again here, very simple functions (average, count, etc.) at this level combined with the power of the Semantic Module result in a large variety of supported measures.

Depending on the semantics of the information we want to visualize, the Histogram View presents two different visualizations: first, if we want to compute statistics over a discrete variable, the Histogram View will be depicted as table of the statistic we want to observe; second, if we want to compute the statistics of a continuous variable, the Histogram View will be depicted as a pure histogram.

Figure A.5 contain an example of these two different possible presentations of the information using the Histogram View. The first one, A.5a, presents a statistics table. In this example, the X axis represents the different MPI primitives (a discrete variable), the Y axis contains the different processes of the application, and each table cell contains the aggregated duration each process spent on each MPI routine. Second view, A.5b, is a pure histogram, where the Y



	MPI_Isend	MPI_Wait	MPI_Waitall
TASK 1.1.1	1,408 us	9,293.17 us	7,829.97 us
	1,280 us	6,046.57 us	12,237.92 us
	1,280 us	6,081.21 us	12,884.28 us
	1,408 us	6,626.89 us	11,873.33 us
	1,408 us	5,744.49 us	13,265.36 us
	1,280 us	7,603.12 us	11,317.73 us
	1,280 us	7,489.76 us	11,198.38 us
TASK 1.8.1	1,408 us	7,150.23 us	10,071.62 us
	1,408 us	9,977.32 us	7,779.81 us
	1,280 us	10,467.14 us	8,003 us
	1,280 us	7,860.48 us	11,160.53 us
	1,408 us	7,159.40 us	11,991.61 us
	1,408 us	7,550.00 us	11,392.38 us
	1,280 us	6,177.32 us	11,761.05 us
	1,280 us	8,630.23 us	10,019.14 us
TASK 1.16.1	1,408 us	10,777.35 us	7,520.14 us

(a) Profile-like view presentation

(b) Pure histogram data presentation

Figure A.5.: Paraver Histogram View

axis represents again the application processes and the X axis are different histogram bins of the IPC measured. In this case, the coloring represents, in a gradient increasing from green to blue, the IPC distribution obtained by each process.

A.2.2. Paraver object model

As described previously, Paraver functionality is tightly coupled with the hierarchical object model. Paraver works with two ortogonal object models:

- The process model, composed by the objects that correspond to the three levels of the most frequently programming models: application, process and thread objects.
- The resource model, that represents the physical resources where the different threads are finally executed. The resource objects are tightly connected with the cluster of SMPs where applications has been executed: processor and node.

Process Model

The Paraver process model, depicted in Figure A.6 is a superset of the most frequently used programming models. On a Paraver window, the type of process model object to be represented can be selected among:

- Set of applications (WORKLOAD)
- Application (APPL)

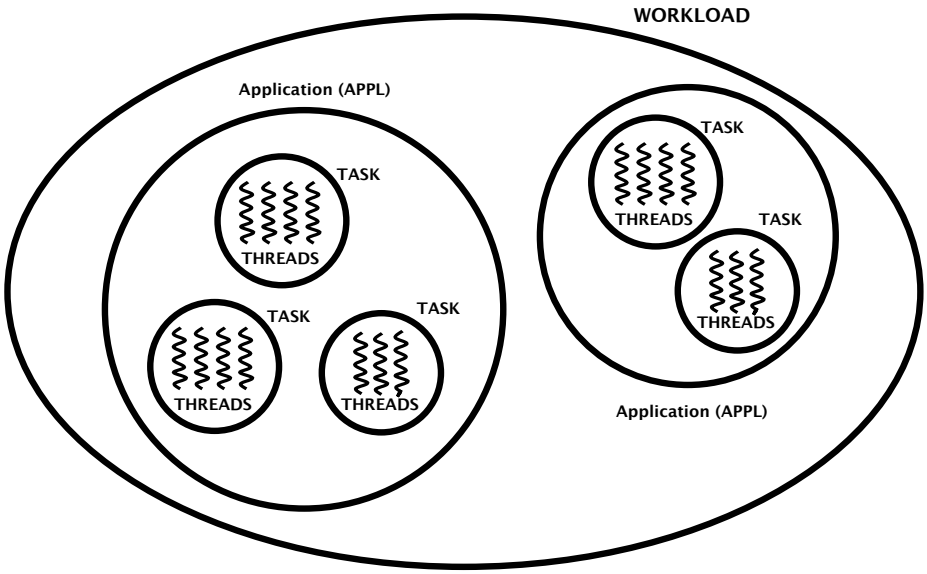


Figure A.6.: Paraver process model

- TASK
- THREAD

A parallel application (APPL level) is composed by a set of sequential or parallel processes (named as TASK in paraver process model hierarchy). The parallel processes are composed by more than one thread while the sequential processes are mapped into one THREAD. The top of the process model hierarchy is the WORKLOAD level representing a set of different applications running on the same resources.

Resource Model

The resource model represents the resources where the applications are executed. On a Paraver window, the type of resource object to be represented can be selected among:

- Cluster of nodes (SYSTEM)
- NODE (set of processors)
- Processor (CPU)

The processors (CPU) are the resources where the threads are executed. Processors are grouped in nodes (NODE). A task is mapped into a node and thus all its threads share their processors in that node. The mapping is not necessarily one to one, so it is possible to have several tasks from a single application on a given node. Tasks from different applications can also be mapped into the same node.

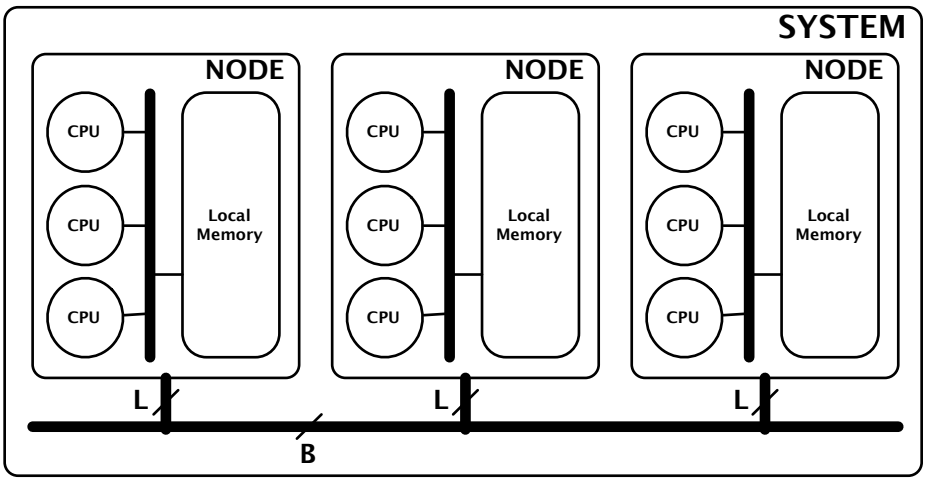


Figure A.7.: Paraver and Dimemas resource model

A.2.3. Paraver Trace

The trace file contains records which describe absolute times when events or activities take place during the execution of the parallel code. Each record represents an event or activity associated to one thread in the system. This file has the extension `prv` and is usually called the *dot prv* file. Furthermore, trace files have associated some other files to configure some aspects of the environment: the symbolic information for the numerical values of the prv file are contained in a file with the extension `pcf` and row labels are contained in a file with the extension `row`.

The Paraver trace file (the *dot prv* file) is composed by a header and a body, Figure A.8. The header describes the process and resource model objects and the body contains the ordered list of records.

Trace header

```
#Paraver (22/05/12 at 16:20):1021312_ns:2(1:1):1:2(1:1,1:2),3
c:1:1:2:1,2
c:1:1:1:1
c:1:1:1:2
```

Trace body

```
1:1:1:1:1:0:100:4
1:2:1:2:1:0:200:4
1:1:1:1:1:100:37:1
1:1:1:1:1:200:57:4
3:1:1:1:1:300:325:2:1:2:1:200:330:10:3000
2:1:1:1:1:300:670070:1
...
```

Figure A.8.: Paraver trace file structure

Trace header

The header has two different parts, a first line that defines the process and the resource model of the tracefile, Figures A.9a, A.9b and A.9c, and a set of lines defining the communicators

for each of the applications defined, Figure A.9d.

The general structure of the process and resource model is presented in Figure A.9a. It contains first a magic number (`#Paraver`), the date when the trace was obtained, and the last timestamp present on the trace (`end_time`). Next it defines the process model, including the number of nodes and its number of processors. This part is listed as `rsrcDef` in A.9a and described in A.9b. Finally, it contains total number of applications contained on the trace (`nAppl`), and the definition and mapping of each application to the resources. This part is listed as `applDef` in A.9a, and described in A.9c.

The following lines of the header contain the definition of the communicators (subsets of the Tasks that take part in a communication operation) created by each application. These records are described in A.9d. In all cases, the task identifiers and the thread identifiers are numbered from 1 to N.

For example, in the trace header present in Figure A.8 defines a system with 2 nodes and one CPU per node (`2(1:1)`). The trace contains information for a single application with two tasks, where each task has just a single thread that executed in nodes 1 and 2 respectively (`2(1:1,1:2)`. This application defined 3 communicators, one including all tasks (that could correspond to the `MPI_COMM_WORLD` if the application is MPI), and one including each task.

It is interesting to note that nodes and CPUs are identified across the trace body using values of 1 to N, being N the total number of nodes or CPUs in the system.

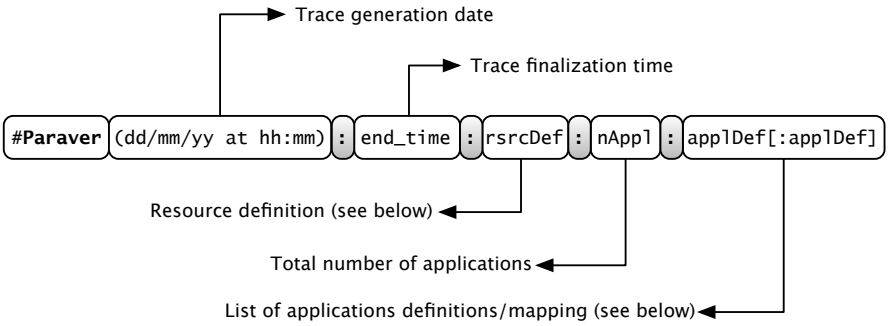
Trace body

The trace body contains list of records ordered by their timestamps. Paraver trace has three types of records: states, user events and communications.

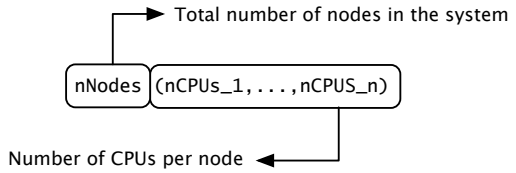
State records. State records, Figure A.10 represent intervals of actual thread status. The first field is the record type identifier (for state records, type is 1). The next fields identify the resource (`cpu` field) and the object to which the record belongs (the triad `appl:task:thread`). Remember that `cpu` is the global processor identifier (if no resource levels have been defined the processor identifier must always be zero). Beginning time and ending time of the state record also have to be specified, and finally, the state field is an integer that encodes the activity carried out by the thread. This kind of records are sorted by its `begin_time`.

Events. Event records, Figure A.11 represent punctual events. They are codified using type/value pairs. The events are often used to mark the entry and exit of routines or code blocks, hardware counter reads, etc. They can also be used to flag changes in variable values. In a single event record we can find a list of one or more type/value pairs, indicating events generated simultaneously.

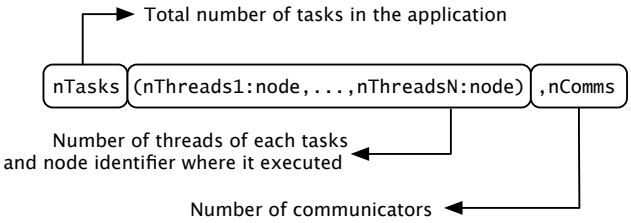
Communications. Communication records, Figure A.12, represent the logical and physical communication between the sender and the receiver in a single point to point communication. Logical communications correspond to the send/receive primitive invocations by the user program. Physical communication corresponds to the actual message transfer on the communication network. In case of using Extrae, where the value of the physical communications times are unknown, the physical communication times correspond to the logical communication times. Communication records are sorted by `lsend` (logical send) timestamp.



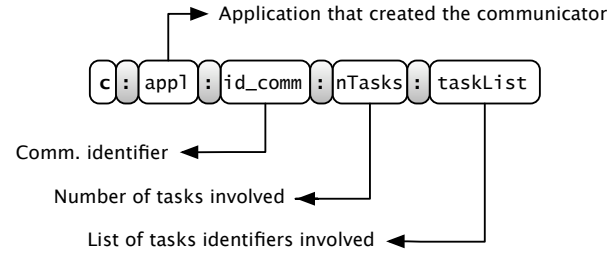
(a) Paraver trace header structure



(b) `rsrcDefinition` section of the Paraver trace header



(c) `applDefinition` section of the Paraver trace header



(d) Communicator definition record structure

Figure A.9.: Paraver trace header records definition

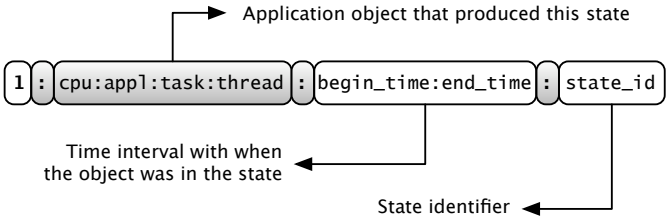


Figure A.10.: Paraver state record specification

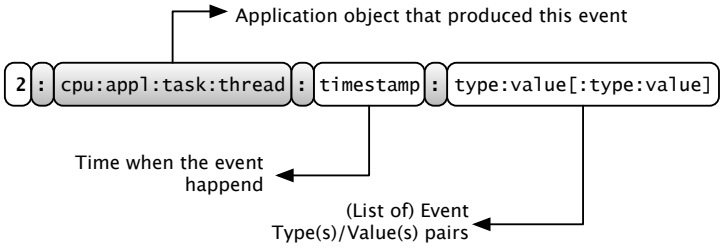


Figure A.11.: Paraver event record specification

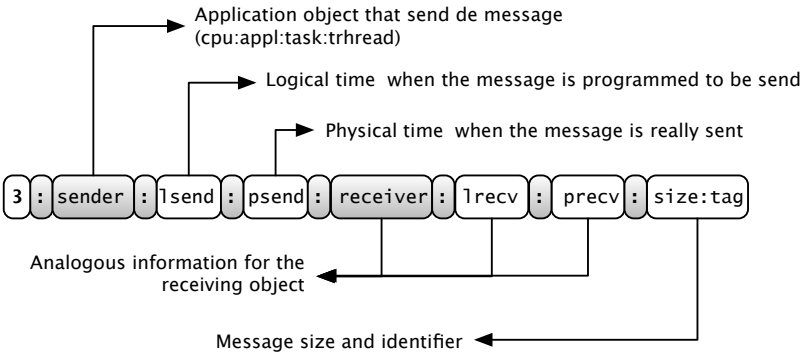


Figure A.12.: Paraver communication record specification

A.3. Dimemas

Dimemas is a performance analysis tool for message-passing programs. It enables the user to develop and tune parallel applications on a workstation, while providing an accurate prediction of their performance on the target parallel machine. The Dimemas simulator reconstructs the time behaviour of a parallel application on a machine modelled by a set of performance parameters. Thus, performance experiments can be done easily. The supported target architecture classes include networks of workstations, single and clustered SMPs, distributed memory parallel computers, and even heterogeneous systems.

Parameter	Description
System Parameters	
Number of nodes	Total number of nodes in the system
Network Parameters	
Number of buses	Total number of concurrent messages in the network
Network bandwidth	Bandwidth capacity of the network (MB/s)
Network latency	Latency of messages send by the network (μs)
Node Parameters	
Processors per node	Number of CPUs, defined per node
Processor ratio	Divisive factor of execution time[1]
Input links per node	Injection links to the network
Output links per node	Read links from the network
Memory bandwidth per node	Memory bandwidth capacity (MB/s)
Memory latency	Latency of messages sent by memory (μs)

[1] The processor ratio is applied to all CPUs in a given node

Table A.1.: Dimemas simulator parameters

A.3.1. Dimemas model

Dimemas model is the same as the Paraver resource model, presented in Figure A.7. It is composed of a network of SMP nodes. Each node has a set of processors and local memory, used for communications within the node. The different parameters that can be modelled are described in Table A.1.

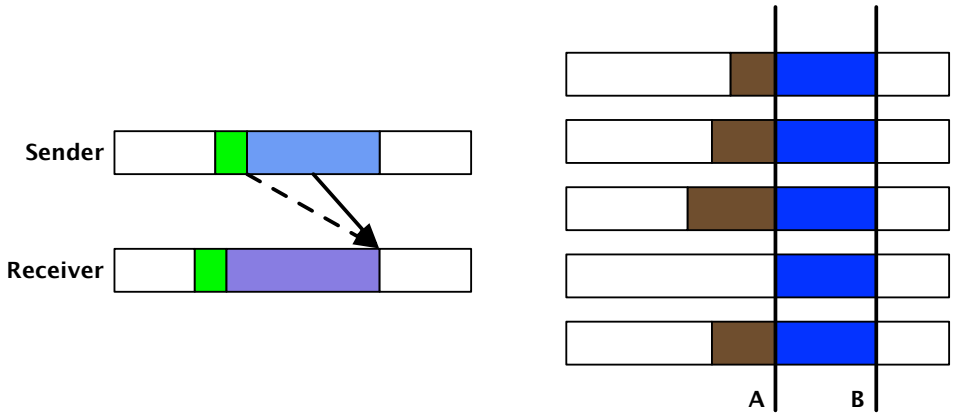
It is interesting to note that, in Figure A.7, the interconnection network is represented with two parameters: number of links from a node to the network, represented with L, and number of buses in the network, represented with B. Parameter L limits the number of messages coming in and going out for a given node, enabling connectivity analyses. Parameter B applies to the number of concurrent messages in the network, enabling network contention analyses.

Internally, the simulator models three different kind of operations: CPU consumption, point-to-point communications and collective communications. In the Dimemas trace we can also add user events to describe logical regions.

CPU consumption operations

The CPU consumption operation, or CPU burst, is simulated simply by advancing the simulation clock the duration as indicated in the input trace. This duration can be modified by the Processor ratio, a divisive factor that will be applied to all the CPU consumption opera-

tions that execute in a given processor. In addition, we can also define modules factors, also divisive facto, that will be applied to those CPU bursts inside a module. Finally, we can also substitute the duration of the CPU bursts inside a module with a fixed duration value. These modules are regions of time defined by using events pairs (start and end).



(a) Dimemas point-to-point communications diagram

(b) Dimemas collective communication diagram

Figure A.13.: Dimemas communication diagrams

Point-to-point communications

A simple Dimemas communication is depicted in Figure A.13a. In this Figure, white regions represent CPU time consumption, light blue blocking time due message not completely transmitted, light purple blocking time due message is not ready in the processor, and green stands for latency time. Two arrows represent the logical, with a dotted line, and the physical communication, with a continuous line. Logical stands for when the task sends the message and the receiver is able to read it. Physical communication stands for when the message is really passing trough the communication network, using the resources. Both can be different because of resources contention, synchronization behaviour, load balance, etc.

Point to point communications are modelled using the parameters latency and bandwidth, thus the time for a message to be delivered is computed as defined in Formula A.1. This formula is both applicable to the point-to-point transmissions messages between nodes, using the network latency and bandwidth, or transmissions intra-nodes, using the memory latency and bandwidth.

$$T = latency + \frac{message\_size}{bandwidth} \tag{A.1}$$

Collective communications

Global communications model use a different formula to compute the duration of the message transmission, and synchronization is included before the communication itself. Al-

$MODEL\_[IN\|OUT]$	$MODEL\_[IN\|OUT]\_FACTOR$	**Description**
0	0	Non existent phase
CTE	1	Constant time phase
LIN	P	Linear time phase, P = number of processors
LOG	Nsteps	Logarithmic time phase

Table A.2.: Dimemas collective communications $MODEL\_[IN|OUT]\_FACTOR$ possible values

though not all implementations of global operations require synchronization, good results suggest us to maintain this simple model. Figure A.13b shows the timing model for collective communication. In this Figure, white blocks represent application computation, brown blocks represent blocking time due the synchronization and blue blocks represent the actual communication, where message transmission time T_trans is delimited by times A and B.

Many collective operations have two phases: a first one, where some information is collected, *fan in*, and a second one, where the result is distributed, *fan out*. Thus, for each collective operation, communication time T_trans can be evaluated as defined in Formula A.2.

$$T\_trans = FAN\_IN + FAN\_OUT \tag{A.2}$$

Where FAN_IN is calculated as defined in Formula A.3.

$$FAN\_IN = \left(latency + \frac{SIZE\_IN}{bandwidth} \right) \times MODEL\_IN\_FACTOR \tag{A.3}$$

Depending on the scalability model of the fan in phase, the parameter $MODEL\_IN\_FACTOR$ can take the values detailed in Table A.2. In case of a logarithmic model, $MODEL\_IN\_FACTOR$ is evaluated as the $Nsteps$ parameter. $Nsteps$ is evaluated as follows: initially, to model a logarithmic behavior, we will have $\log_2 P$ phases, where P is the number of tasks involved. Also, the model wants to take into account network contention. In a tree-structured communication, several communications are performed in parallel in each phase. If there are more parallel communications than available buses, several steps will be required in the phase. For example, if in one phase 8 communications are going to take place and only 5 buses are available, we will need $8/5$ steps. In general we will need C/B steps for each phase, being C the number of simultaneous communications in the phase and B the number of available buses. Thus, if $steps\_i$ is the number of steps needed in phase i, $Nsteps$ can be evaluated as defined in Formula A.4.

$$Nsteps = \sum_{i=1}^{\lceil \log_2 P \rceil} steps\_i \tag{A.4}$$

Finally, for FAN_OUT phases, the same formulas are applied, changing $MODEl\_IN$ by $MODEL\_OUT$, and $SIZE\_IN$ by $SIZE\_OUT$. $SIZE\_IN$ and $SIZE\_OUT$ pos-

$SIZE\_IN/OUT$	**Description**
MAX	Maximum of the message sizes sent/received by root
MIN	Minimum of the message sizes sent/received by root
MEAN	Average of the message sizes sent and received by root
2*MAX	Twice the maximum of the message sizes sent/received by root
S+R	Sum of the size sent and received root

Table A.3.: Dimemas collective communications options for $SIZE\_IN$ and $SIZE\_OUT$

sible values are described in Table A.3.

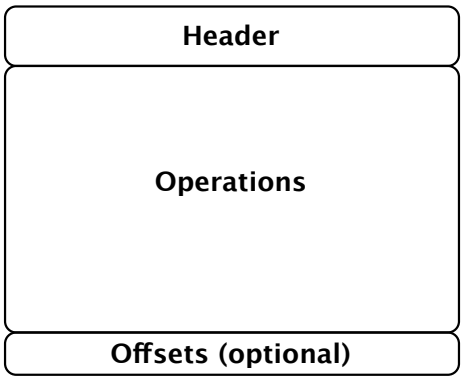
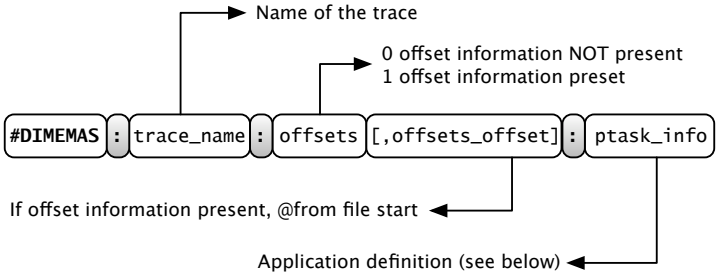


Figure A.14.: Dimemas trace file structure

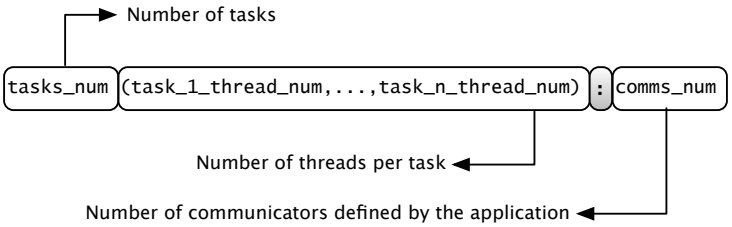
A.3.2. Dimemas trace

Dimemas trace file contains the different operations performed by a single application, that can have multiple tasks and multiple threads per task. A Dimemas trace has the extension `dim`. Physically, a Dimemas trace is very similar to a Paraver trace where records do not have any associated time-stamp. In addition, the communications, instead of being stored as a single communication record, are separated in send primitives and receive primitives. It is the task of the simulator itself to reconstruct both the execution times and also match the communications performed by the different tasks and threads using the simple model described.

The global structure of a Dimemas trace is presented in Figure A.14. It has three different parts: a header containing the application definition and the communicators definition (equivalent to Paraver trace header); a body with the different operation records and event records; and a offsets section, a dictionary pointing to the position in the file (the offset) of each task and thread to accelerate the access to the file.



(a) Initial part of the Dimemas trace header



(b) `ptask_info` field definition

Figure A.15.: First line of Dimemas trace header definition

Trace header

As in the Paraver trace file, the header has two different parts: the first line of the file that indicates if the tasks offsets are present and its location on the file plus the application description; and a set of multiple lines containing the communicators definition.

The first line of the header is described in Figure A.15. First part, A.15a, contains the magic number (`#DIMEMAS`) plus the trace name. Next field, `offsets` indicates whether the trace file contains offsets information or not. If its the case `offsets_offset` field contains the absolute position of the first tasks offset record in bytes from the beginning of the tracefile. The second part, A.15b, contains the description of the application as the total number of tasks (`tasks_num`), a list with the total number of threads per tasks and the total number of communicators described by the application.

Following to this line, we find as many communicators description records, Figure A.16, as detailed in the application description.

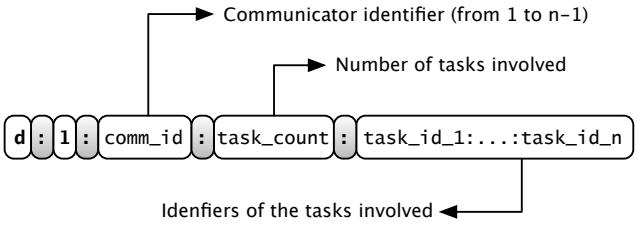


Figure A.16.: Paraver communicator definition record structure

Trace body

The trace body is a list of the operations records emitted by each thread of each task. In this case, the records are sorted by task and thread, and for a given pair task/thread, by the occurrence of the operation modelled in the execution of the application.

In the trace body we can distinguish five different types of records: CPU bursts, send operations, receive operations, collective operations (also called *global operations*) and events. The first four records correspond directly to the operations described in the Dimemas model. The event records can be used to delimit regions of the application as different modules. The CPU bursts inside a module region can be tuned by apply a divisive module factors to their duration, or substitute the duration with a constant duration to perform balancing experiments. Events are also useful to analyse a resulting simulated trace in Paraver produced by Dimemas analysis.

CPU burst. CPU burst, Figure A.17, correspond to CPU consumption operations and basically contain the object that demands the CPU consumption and the number of seconds it demands.

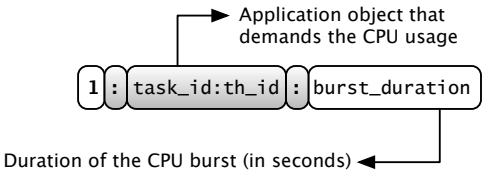


Figure A.17.: Dimemas CPU burst record definition

Send operation. Send operations records, Figure A.18, points the simulator that the object indicated, the *sender*, wants to send a message to a given partner, the *receiver*. If we are simulating a regular MPI application, the receiver `thread_id` would be marked as -1. The message has a given size in bytes as well as a communicator and a tag. The communicator indicated must have been defined on the header, and both sender and receiver must belong to it. The tag field is useful to identify the message and match the receiver operation in the partner. The `synchronism` field indicates which if the send operation will be blocking or non-blocking and the if the send protocol executed in the simulator will require a *rendezvous* with the receiver or not.

Receive operation. Receive operation records, Figure A.19, are the symmetric records to the send records, but emitted by the objects that require the reception of a message. In this case, the `recv_type` field indicates if the receive operation will be blocking or non-blocking or if it is a wait operation, block operation until the required message is effectively received, linked to a previous non-blocking receive operation.

Collective operations. Collective operation records, also named global operations, Figure A.20, are the records emitted by an application object to perform a collective communication as described in the Dimemas model. Apart from the object who emits the record, a collective operation also includes the type of collective communication, field `global_op_id`. This field indicates which of the predefined Dimemas collective communications is being modelled. The possible values are listed in Table A.4. Their implementations of the different

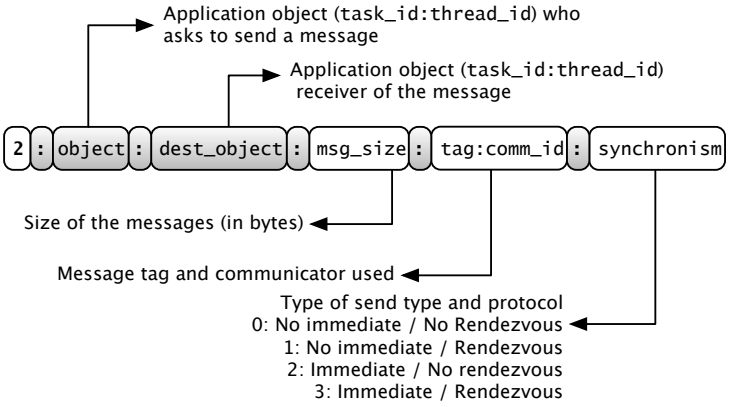


Figure A.18.: Dimemas send operation record definition

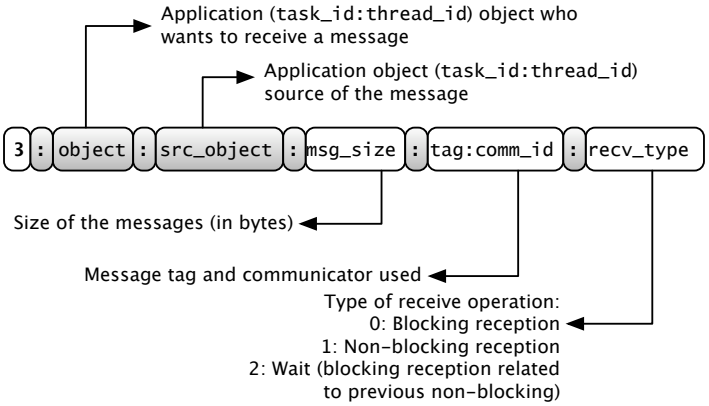


Figure A.19.: Dimemas receive operation record definition

collectives operations are inspired by the MPI collective communications listed on the table. Next fields correspond to the communicator used, `comm_id`, the object of the application that will act as root if the collective communication requires it and the total amount of bytes send and received during the communication.

Events. Event records, Figure A.21, are the equivalent records to Paraver events. In principle, the events emitted are only used in Dimemas simulator to mark regions or modules in the Paraver way, i.e. type and value different to 0 to mark and entrance to a module and the same type a value 0 to mark the exit of the module. Using the modules we can adapt the CPU factor at finer level.

Trace offsets

The trace offsets section is an optional section of to indicate where different records of tasks and threads appear in the trace body. If present (indicated in the trace header), this section contains as many offset records as tasks appear in the trace-file. The offset record is described in Figure A.22, it contains the identifier of the corresponding task, and, for each thread in this

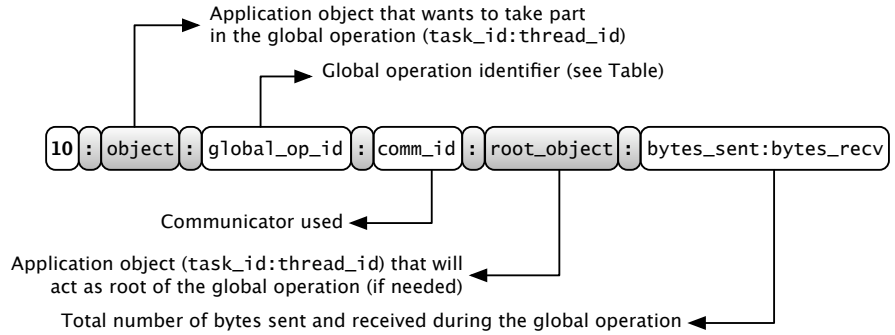


Figure A.20.: Dimemas collective operation record definition

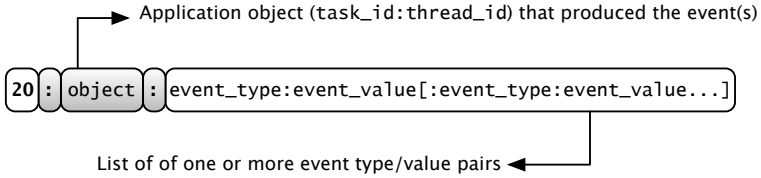


Figure A.21.: Dimemas event recod definition

task,the offset in bytes from the beginning of the trace-file where the first record of the given thread appears in the trace body. Using these offsets, the initialization of of the simulator is speedup, avoiding an expensive process of locating the initial records per tasks/threads. The trace offsets are specially interesting when using large trace-files.

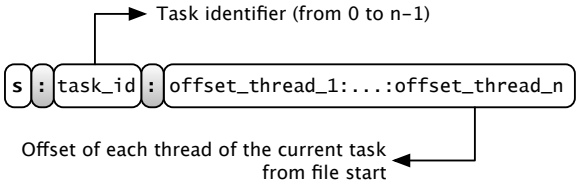


Figure A.22.: Dimemas offset record definition

A.3.3. Dimemas configuration file

The Dimemas configuration file is the file that includes the parameters values of the resource model, detailed in Table A.1, as well as the mapping between one or more application traces to the resources modelled. This configuration file has the extension `cfg`.

This file uses the Self Describing Data Format (SDDF), so on each configuration we can find the definition of the records it uses. Basically, the most important records are the ones listed in Figures A.23, A.24, A.25 and A.26.

As a summary, the `environment information` record, Figure A.23, contains the general values regarding the number of nodes in the system, the network parameters (latency, bandwidth, buses, etc.) as well as the model of the collective operation.

global_op_id value	**MPI equivalent collective communication**
0	MPI_Barrier
1	MPI_Bcast
2	MPI_Gather
3	MPI_Gatherv
4	MPI_Scatter
5	MPI_Scatterv
6	MPI_Allgather
7	MPI_Allgatherv
8	MPI_Alltoall
9	MPI_Alltoallv
10	MPI_Reduce
11	MPI_Allreduce
12	MPI_Reduce_Scatter
13	MPI_Scan

Table A.4.: Possible values of `global_op_id` field in a collective communication record of Dimemas

Next, we find as many `node information`, Figure A.24, records as nodes exist in the system to model. In this way, the cluster modelled can contain nodes with different characteristics. In the `node information` record appear the number of processors the node has; the number of input and output links to the system network; the memory parameters and the `speed ratio`, the (divisive) CPU ratio applied to the CPU bursts that will run in the present node.

`mapping information` record is the element of the configuration file where application traces are linked and mapped to the hardware described. It contains the file name of the trace, the number of tasks present on this application trace and then a list indicating on which node we want to execute each tasks.

The last record to highlight is the `modules information` record. This record contains a type/value pair, to define which event type/value pair describe a module, and which multiplicative factor (`execution_ratio`) will be applied to the CPU bursts inside the module. This factor overwrites the node speed factor while the module is active. If the factor is a negative value, it will be interpreted as a constant duration value for the CPU bursts inside the module.

```
#1:
"environment information" {
    char "machine_name"[];
    int "machine_id";
    // "instrumented_architecture" "Architecture used to instrument"
    char    "instrumented_architecture"[];
    // "number_of_nodes" "Number of nodes on virtual machine"
    int     "number_of_nodes";
    // "network_bandwidth" "Data tranfer rate between nodes in Mbytes/s"
    // "0 means instantaneous communication"
    double  "network_bandwidth";
    // "number_of_buses_on_network" "Maximun number of messages on network"
    // "0 means no limit"
    // "1 means bus contention"
    int "number_of_buses_on_network";
    // "1 Constant, 2 Lineal, 3 Logarithmic"
    int "communication_group_model";
};;
```

Figure A.23.: Dimemas system definition record structure

```
#2:
"node information" {
    int "machine_id";
    // "node_id" "Node number"
    int     "node_id";
    // "simulated_architecture" "Architecture node name"
    char    "simulated_architecture"[];
    // "number_of_processors" "Number of processors within node"
    int     "number_of_processors";
    // "number_of_input_links" "Number of input links in node"
    int     "number_of_input_links";
    // "number_of_output_links" "Number of output links in node"
    int     "number_of_output_links";
    // "startup_on_local_communication" "Communication startup"
    double  "startup_on_local_communication";
    // "startup_on_remote_communication" "Communication startup"
    double  "startup_on_remote_communication";
    // "speed_ratio_instrumented_vs_simulated" "Relative processor speed"
    double  "speed_ratio_instrumented_vs_simulated";
    // "memory_bandwidth" "Data tranfer rate into node in Mbytes/s"
    // "0 means instantaneous communication"
    double  "memory_bandwidth";
    double "external_net_startup";
};;
```

Figure A.24.: Dimemas node definition record structure

```
#3:
"mapping information" {
    // "tracefile" "Tracefile name of application"
    char    "tracefile"[];
    // "number_of_tasks" "Number of tasks in application"
    int     "number_of_tasks";
    // "mapping_tasks_to_nodes" "List of nodes in application"
    int     "mapping_tasks_to_nodes"[];
};;
```

Figure A.25.: Dimemas mapping definition record structure

```
#5:
"modules information" {
    // Module type
    int     "type";
    // Module value
    int     "value";
    // Speed ratio for this module, 0 means instantaneous execution
    double  "execution_ratio";
};;
```

Figure A.26.: Dimemas modules definition record structure

A.4. Trace manipulators and Translators

Apart from the three main tools described in this Appendix, in the BSC Tools ecosystem we find a set of minor applications useful to manipulate the application traces and make them more tractable or to translate traces from and to other analysis packages.

A.4.1. Trace manipulators

Trace manipulators apply exclusively to Paraver traces and offer a series of capabilities focused on reducing or summarizing the information present on the trace, so as to leverage the memory requirements of Paraver.

`trace_filter`. The trace filter offers two main capabilities to restrict the information present on a Paraver trace file. First, it can erase the state, event or communication records defined by user. For example, it can erase those states whose duration is smaller than a given value or the events of a given type. The second filtering option refers to the possibility to erase from the Paraver trace an application task or a set of application tasks. The trace filter application is presented both as a command-line tool and also integrated in the Paraver application.

`cutter`. As it names suggest, the trace cutter provides an interface to cut the trace. This cut can be easily by giving a time range as well as by a percentage range of the application. The resulting trace will discard the records outside the cut ranges. In addition, this trace cut can keep the original timestamps, to locate where in the original run happened the cut or the timestamps can be translated considering the initial value of the cut as the 0 time. The cut application also offers the ability of filter user-defined set of tasks. This tool is also offered both as command-line and integrated in Paraver.

`sc`. This tool name stands for *software counters* and offers a way to resume the values of a given type of events as well as the number of events of given type. The operation is simple and user just have to provide the summarization frequency, the type of event (or events) to summarize. In the output trace, the events summarized are discarded, and a new summarization events with the computed value emitted with the supplied frequency.

`comm_reducer/comm_fusion`. These two tools are the equivalent of the software counters to summarize communication records. The `comm_reducer` directly summarizes all the communications that appear every a user-defined interval. The `comm_fusion` application we find a secondary time interval that restricts the communications to be merge to those closer that this value.

`merger`. This application provides a way to merge multiple application traces into a single one. To perform this operation all traces must share the same resources description and the same application description on.

`task_shifter`. In some cases, due to imprecisions in the system clocks of different nodes where an application run, the Paraver traces present an excessive jitter across the timestamps of the different tasks. The *task shifter* application permits to manually correct the these timestamps by applying a shifting factor per task.

`stats`. The statistics application is not purely a trace manipulator, but a tool to extract statistics about the records present in a given trace, for example the number of computation

bursts states and the time they represent or the number of communications and the volume of messages transmited.

A.4.2. Trace translators

The trace translators permit the interoperation between tools inside the BSC Tools ecosystem as well as to import (and export) traces from (and to) other software packages.

`prv2dim`. The Paraver to Dimemas translator suppose the core application to interoperate between the main tools of the BSC Tools suite. Although the Extrae package is able to generate Dimemas traces, the common analysis workflow using the tools starts by obtaining a Paraver trace. First, with Extrae we obtain a full application trace that is manipulated using the filtering tools listed previously and finally analysed with Paraver. Once an interesting region is detected, we cut on the original trace and translate it using `prv2dim` to perform the desired experiments with Dimemas.

The interoperation in the reverse sense, from Dimemas to Paraver, relies in the ability of Dimemas to generate Paraver traces.

`otf2prv`. The Open Trace Format (OTF) is a trace format definition proposed by the Technische Universität Dresden, the University of Oregon and the Lawrence Livermore National Lab. It primary target is to be scalable. To do so, it defines a trace as the collection of multiple files (or *streams*) to ease the generation and the lecture. Finally it provides an optimized API to produce and read the traces. `otf2prv` is capable to translate OTF traces to Paraver format. The symmetric translator, `prv2otf`, is currently in development.

`prv2prof`. This tool provides the translation capabilities to transform the Paraver traces into profiles, summarizing the information available. This translator has been used to successfully interoperate between Paraver traces and the TAU Performance System from the University of Oregon and the HPC-Toolkit from IBM.

A.5. Performance Analytics

The Performance Analytics is the newest area added to the BSC Tools ecosystem. It is the result of the current research mainly focused on extracting insight from application traces to ease the analysis process. In Figure A.1, we depict the Performance Analytics features as a box that uses a Paraver trace as an input provides also feedback to the trace and also generates other "supplementary information". Actually, this is the initial way we operate with these new features, but they also could be categorized in other points in the workflow.

Currently, we can list up to four different Performance Analytics techniques into the BSC Tools suite: cluster analysis, spectral analysis, detailed performance evolution analysis and performance tracking. The cluster analysis tool features and its interaction with the rest of the tools is described in Appendix B. Here we introduce briefly the rest of the works.

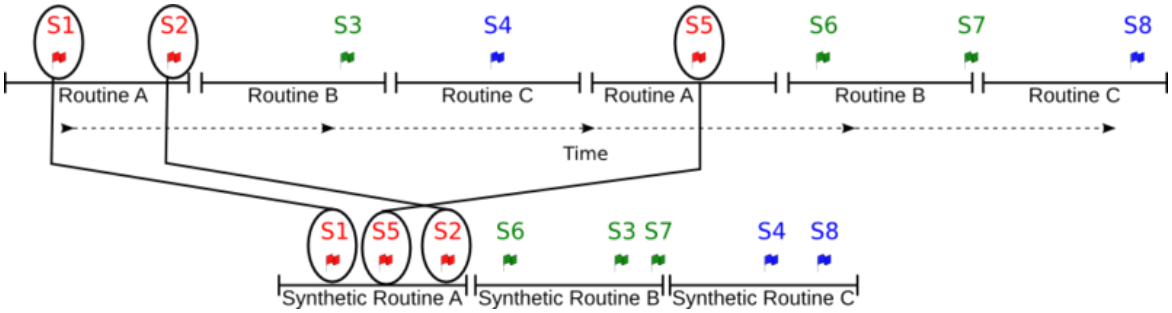


Figure A.27.: Folding mechanisms scheme. Picture obtained from [6]

A.5.1. Spectral analysis

The spectral analysis technique is described in [64]. In brief, it consists on representing the input trace as a signal, and then apply a set of signal processing techniques to detect the typical periods a scientific parallel application exhibits. In practice, the spectral analysis results in 'automatic cutter', that receives a Paraver trace of several gigabytes, a produces a trace cut that contains just one or two of those detected periods. This trace cut size is usally one or two orders of magnitude smaller than the original, but it contains a detailed view of the main computation and communication pattern the application performs.

Additionally, there is an on-line implementation of spectral analysis [128] into the Extrae package. Using this implementation, the traces produced by the tracing package are directly one or two representative periods of the application, avoiding the space requierements when generating a trace of a large-scale application.

A.5.2. Detailed performance evolution analysis

The detailed performance evolution analysis relies on combining both instrumentation and sampling. This way, it introduces a "folding" mechanism [6]. The folding provides very detailed performance information of code regions on iterative and regular applications.

Figure A.27 shows how the folding works on a trace-file that contains instrumented information on two iterations of a loop that executes three routines (A, B and C) and sampling information (shown as flags). The folding creates three synthetic routines and populates them with the sparse samples from the trace-file maintaining their relative timestamp.

The tool that implements the folding mechanism reads a Paraver trace that contains sampling and instrumentation information, applies the folding mechanism, and provides a scatter plot that characterize the time-varying behaviour of a desired metric(s) of a given computation region of the application (a subroutine, a cluster, etc.). Figure A.28 contains the time-varying evolution of five different performance counters for the most time-consuming computation region of CGPOP application, detected using the cluster analysis. Finally, this tool can also feedback the folded samples into one of the regions analysed in the original trace so as to take advantage of the Paraver analysis power.

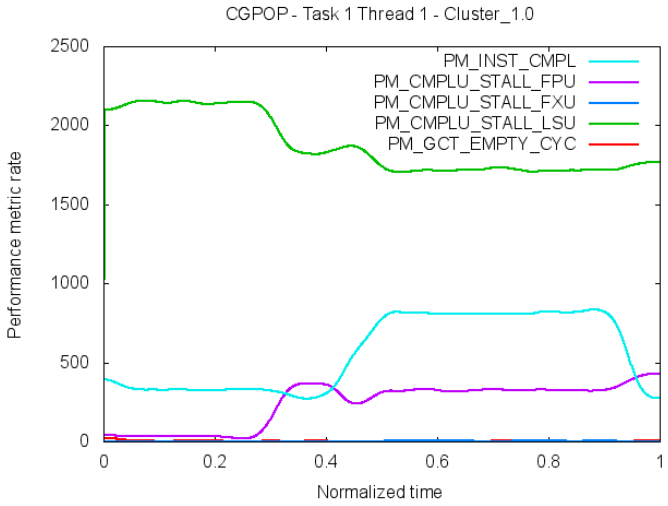


Figure A.28.: Application of the folding mechanism to multiple performance counters

A.5.3. Performance tracking

The performance tracking is a technique that relies in object tracking methods to detect how the behaviour of a parallel application evolves through different scenarios [7]. It provides a versatile approach allowing a wide scope of studies, including multiple time intervals within same experiment and multiple executions with different configurations. This enables the user to perform very diverse parametric and evolutionary studies, to see which factors (i.e. hardware, software versions or program configuration variables) have bigger impact on the resulting performance, foresee trends in future experiments and better understand how and why the behaviour of the program evolves.

This approach mimics the threefold structure of a visual tracking algorithm, including:

- Generation of a sequence of images. Each execution scenario is represented as a 2D performance space defined by a given pair of metrics (i.e. IPC and Instructions). Then, every computation of the program is depicted in this space according to their performance behaviour.

- Object recognition within each image. Clustering is used to group all computations presenting similar behaviour. In this way, we are able to automatically identify in the performance space a set of different performance trends exhibited by the most relevant code regions.

- Motion analysis of the objects across images. We analyse the whole sequence of images to see the evolution of these performance trends across scenarios.

As a result, this tool produces a sequence of scatter plots showing changes in the performance characteristics of the different code regions. This representation enables to easily

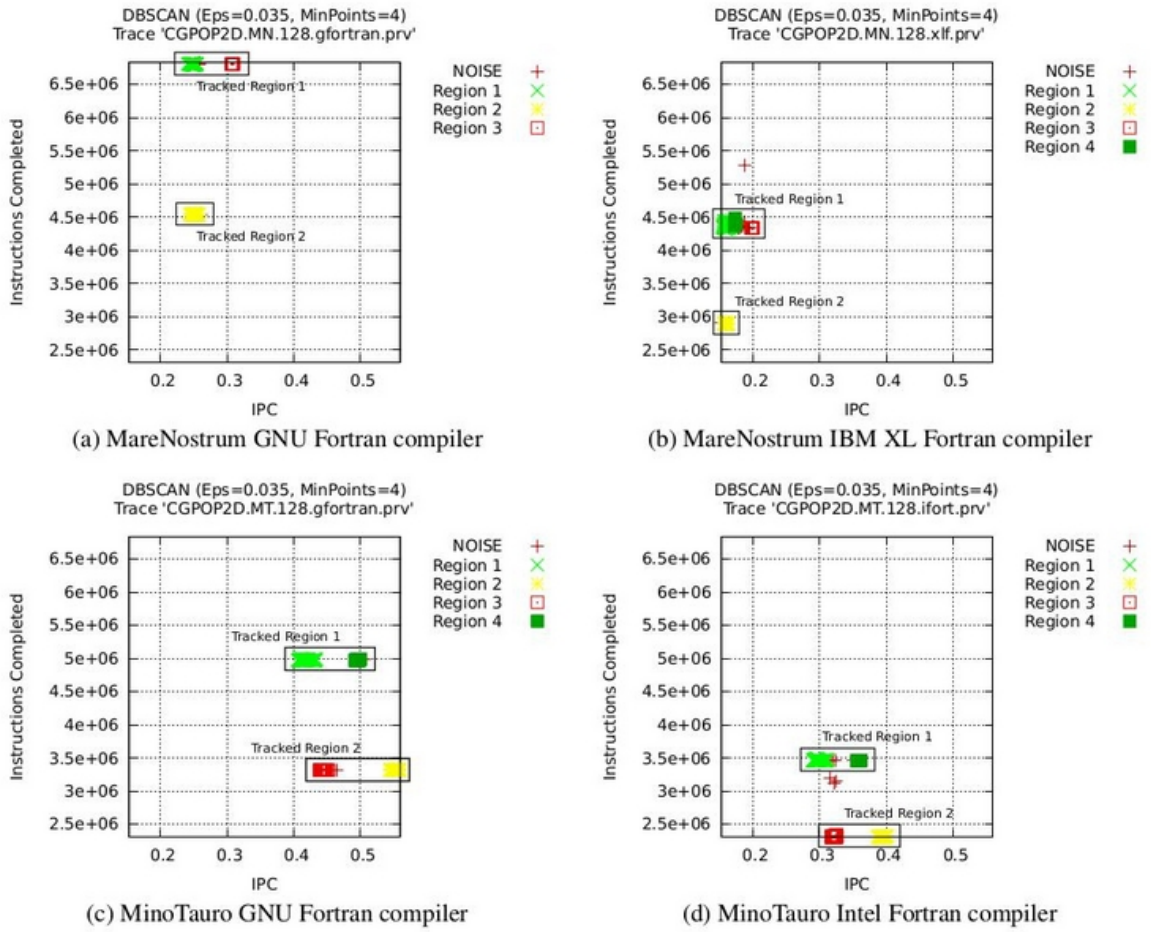


Figure A.29.: Sequence of plots showing the program structure at different scenarios. Picture obtained from [7]

identify behavioural variations and perturbations. In Figure A.29 you can see how the performance of the main code regions in CGPOP application (detected using cluster analysis) varies depending on the architecture and the compiler used.

B. The `ClusteringSuite` Software Package

All the research described on the present thesis was supported by the development of a production-class software included in the BSC Tools ecosystem. In this appendix we include the major facts of the software design as well as a detailed description of the actual tools implemented. This part can also be used as a brief manual of this software.

B.1. `ClusteringSuite` design

The `ClusteringSuite` is the piece of code that implements the techniques introduced along this thesis. Basically, it is composed by two main libraries, `libClustering` and `libTraceClustering`, and a set of binaries that invoke them to offer the different features. All this collection of software follows an object-oriented design and is implemented in C++, with limited features implemented in C.

B.1.1. Software engineering

In this section we present the a coarse-grain description of the software engineering behind the `ClusteringSuite` package. This package represents the third version of the software package that aggregates a set of features that became stable as the development of the thesis advanced.

The features of the package are divided in two main parts: first, a core cluster analysis library, `libClustering`, that includes the abstraction of the information containers and the clustering algorithms; second, the `libTraceClustering` that offers the features of extracting the required information from application traces, prepare the information to perform to use the `libClustering` and generate the different outputs. Both libraries offer a clean façade class to access the different features they implement that is used by the different binaries. In the following points we detail the classes that compose each library and the interaction among them.

libClustering

Figure B.1 contains a basic UML class model of the `libClustering`. It contains the four main classes required to perform a cluster analysis. First, a *Point*, the abstraction of n-dimensional point including basic operation such as the Euclidean distance to another Point. A set of points is aggregated over a *DataSet*, useful to manipulate data ranges or to build indexes to

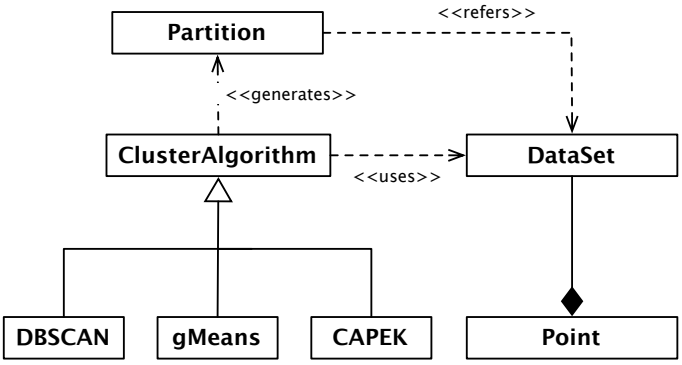


Figure B.1.: UML class model of the libClustering library

ease the access to each individual point. Next, the hierarchy where the top class is *ClusteringAlgorithm* acts as an interface to the actual implementations of multiple algorithms. The ClusteringAlgorithm objects process a DataSet and generate a *Partition*, a class that relates the Points in a DataSet to the cluster they belong. In this Figure we just depict three of the possible cluster algorithms this library offers.

libTraceClustering

libTraceClustering is the library that includes all the logic required to extract the information from an application trace (a Paraver trace or a Dimemas trace) and then process it to execute a cluster analysis using the libClustering library. As can be seen in the UML class model of this library, Figure B.2, the interaction between these two libraries relies on a hierarchy inheritance, where *TraceDataSet* and *CPUBurst* are specializations of *DataSet* and *Point* respectively defined in the libClustering library.

ClusteringDefinition class contains the information to set up all the model. First, it defines which of the cluster algorithms from the libClustering will be used (and the possible values for its parameters). Second, it defines the different *ClusteringParameter* objects. Each ClusteringParameter represents one of the dimensions that describe a Point (specialized as CPUBurst at this level) used by the ClusteringAlgorithm to discover the different clusters.

Using the ClusteringParameter's, a *DataExtractor* object will read the contents of a *Trace* filling the TraceDataSet with the CPUBursts found.

Then the particular ClusteringAlgorithm is executed and creates the corresponding *Partition* object (also part of the libClustering library). This Partition object will be used first by a *ClusteringStatistics* object to compute statistics (and also the possible extrapolations described in chapter 7) and by *PlotManager* (also defined by the ClusteringDefinition object) to generate output plots of the clusters found. Then *TraceReconstructor* will create an output trace with the same information of the original and the information (events) to identify to which cluster each CPU bursts belongs.

In case of using the Aggregative Cluster Refinement algorithm, the library behaves differently than using a regular cluster algorithm. Basically, it will make use of DBSCAN (to clarify the drawing, in Figure B.2 it is represented as it uses any ClusteringAlgorithm) on

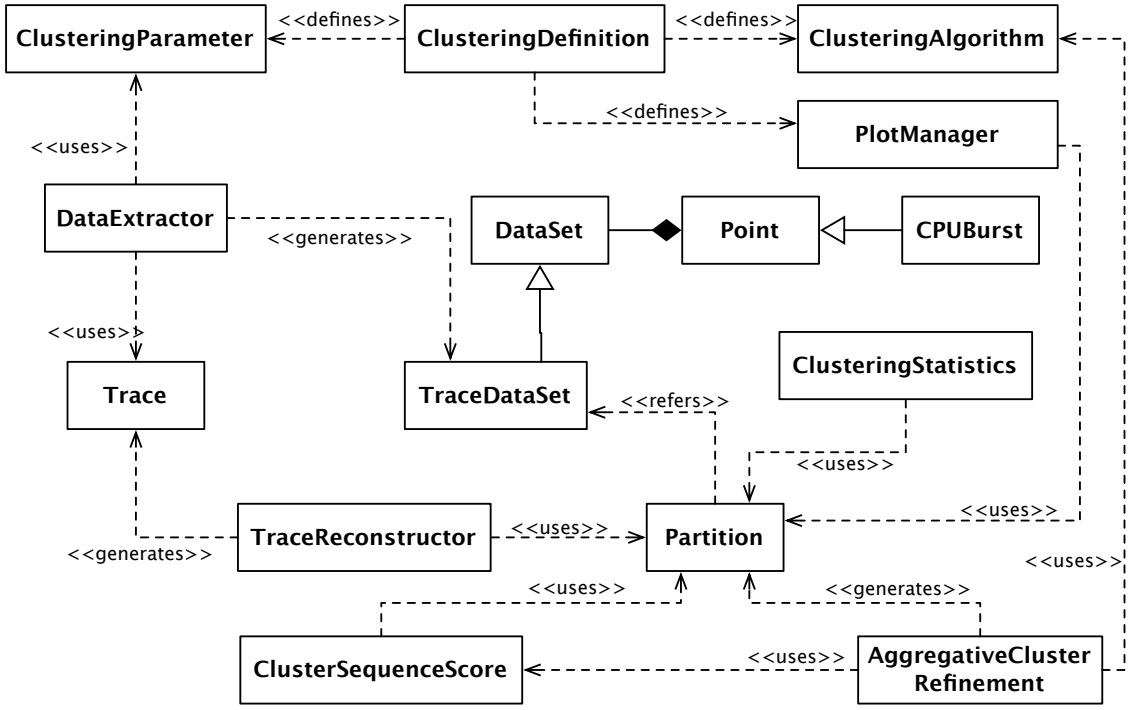


Figure B.2.: UML class model of the `libTraceClustering` library

each refinement step as well as a *ClusterSequenceScore* object to evaluate the quality of the intermediate clusters (the different Partitions). AggregativeClusterRefinement produces a final Partition of the data, but the library can keep track of intermediate ones to also produce intermediate traces and plots useful to evaluate the hierarchical generation of the final clusters.

B.1.2. Libraries and tools

The features offered by these two libraries is effectively presented to the user as a set of command-line application binaries. These binaries orchestrate the different steps of the cluster analysis to produce the desired results.

The tools offered in the `ClusteringSuite` package are `BurstClustering`, `ClusteringDataExtractor` and `DBSCANParametersApproximation`.

`BurstClustering` The main tool that includes the cluster analysis based on a application trace. The user provides an XML file (used by all three tools) to configure the analysis (a *ClusteringDefinition* class in the model) and a trace file. Using the `libTraceClustering` it processes the provided trace, run the cluster algorithm (both the ones implemented in the `libClusterig` or the Aggregative Cluster Refinement) and generates the output trace and the cluster statistics and plots.

`ClusteringDataExtractor` A tool that only offers the data extraction from the input trace and the plot generation, but not the cluster analysis. It is useful to performn prelimin-

ary observations about the data distribution so as to adapt the parameters used by the cluster analysis, for example to filter the individuals.

DBSCANParametersApproximation This tool is useful when using the DBSCAN algorithm to help the user to tune the algorithm parameters.

B.2. ClusteringSuite tools usage

This section is intended as a brief manual of regular use of the three tools included in the ClusteringSuite software[1]. As mentioned before, the tools offered in the software package use an input trace where the information is extracted. Even it could be a Paraver or Dimemas trace, it almost all cases, the input trace is a Paraver trace. The second input file these tools requires is the configuration file XML. This file is key to define which the parameters of the clustering process.

In brief, the cluster analysis process its composed by 5 steps. First four steps are required to generate the XML configuration file, while the last two are the execution of the BurstClustering tool itself and the observation and analysis of the results.

1. Selection of the clustering/extrapolation parameters.
2. Define the filters and normalization applied to the input data
3. Select the cluster algorithm and its parametrization
4. Define the output plots
5. Execute the cluster analysis
6. Observe the different outputs

The actual definition of the different records in the XML file are described in the following section (B.3), while this one include the guidelines to detect the information it will contain.

1. Select the clustering parameters

The first decision to take when performing a cluster analysis is which of the data present in the input trace will be used to describe each CPU burst, in the ClusteringSuite terminology, we call them simply the *parameters*.

Using the Paraver vocabulary, a CPU burst is expressed in a trace as a *State Record* of value 1 (*Running State*). The parameters available to characterize a CPU burst are those events that appear at the end time of the given Running State. As a Paraver event is a pair event/value, in the XML file we use the event type to indicate events we whose values will be stored in the different bursts. We can also use Running State duration (difference between end time and begin time) as a CPU burst parameter.

[1] All the guidelines presented in this section are applicable to the ClusteringSuite v2.XX

In the XML we will express those parameters that will be used by the cluster algorithm, the *clustering parameters*, and those that will be used in the extrapolation process, the *extrapolation parameters*. The parameters can be defined as single event reads (*single events*) or combinations of pair of events (*mixed events*). In case we use the CPU burst duration, it will always be used as a clustering parameter.

It could be obvious, but to define the different parameters it is essential to know first which ones we want to use and which are the event type codification present in the trace. To do that we need to go through to the Paraver Configutarion File (*.pcf* file generated by `Extrae`) and check which events appear in the trace and their event type encoding. Almost in all analyses we use the Performance Hardware Counters events, being Completed Instructions and IPC the usual metrics combinations used by the cluster algorithm.

2. Define the filters and normalizations

Once knowing which are the clustering parameters, we have to decide the possible filters we want to apply. The filters prevent the cluster algorithm of analysing CPU bursts that can bias the result or do not add any valuable information. We found two different filters: a duration filter to discard those burst whose duration is shorter than a given value, and a range filter that can be defined to each parameter and eliminates those bursts than are out of the boundaries.

To tune the duration filter we use the `stats` tools provided by the CEPBA-Tools package. Using the `-bursts_histo` parameter this tool computes a plot as the one presented in Figure B.3 for a given Paraver trace. This plot is an histogram where the x axis is the duration of the CPU bursts and quantifies both the aggregated time of the CPU bursts, the green bars, and the number of bursts, the red line. Observing this plot we can select the duration that eliminates de maximum number of bursts (red line at left of the select duration), while maintaining a high value of aggregated time (green bars at right of the selected point). For example, in the Figure B.3, a reasonable duration filter will be 10 miliseconds.

With respect to the normalizations, we provide the possibility of applying first a logarithmic normalization, useful when the parameter range is wide and can bias the results of the cluster analysis. The logarithmic normalization can be applied to each parameter independently. The second normalization is a pure range normalization to set the parameter values in range $[0, 1]$, following the formula range ($\forall a_i \in A, a_i \leftarrow (a_i - min(A))/(max(A) - min(A))$). When using the range normalization, it will be applied to each parameter used, so as to guarantee that all of them have the same weight in the analysis. If we to add more weight one of the parameters used in the cluster analysis, we can apply a multiplicative factor.

To clarify how the different normalizations and filters work, this is the order as they are applied: when a CPU burst is read, its duration is checked and then the different parameters that have range filters defined; to those bursts that pass the filters its performed the logarithmic normalization of each parameter that requires it and afterwards the range normalization. Finally, the scaling factor is applied.

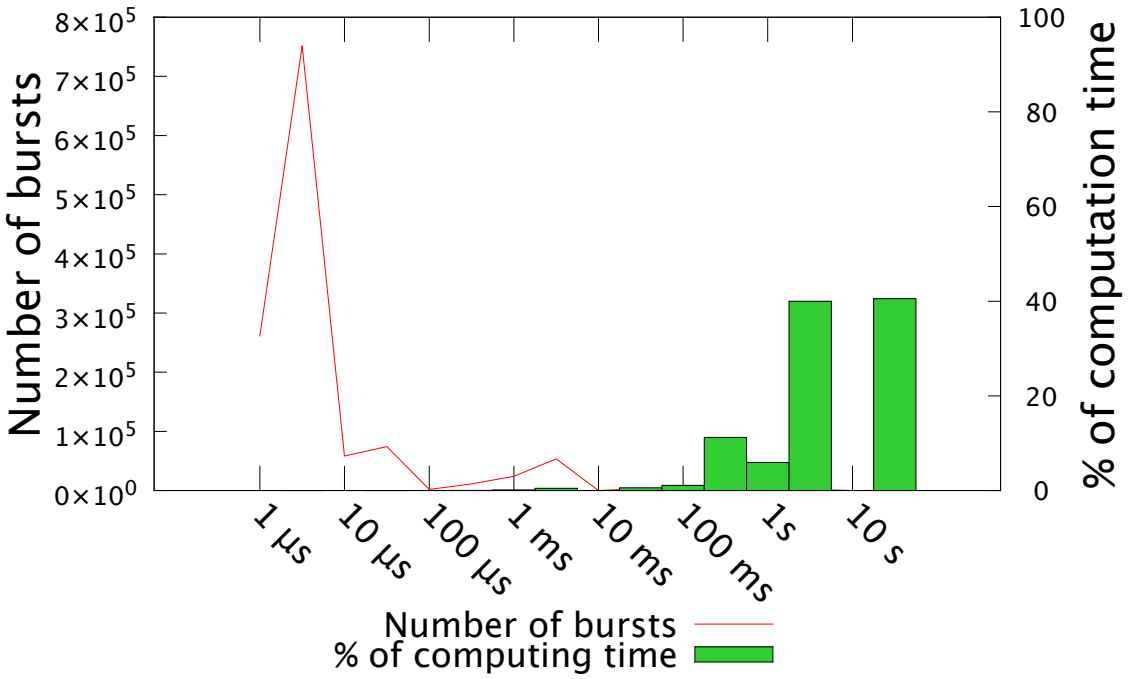


Figure B.3.: Bursts histogram produced by `stats` tool

3. Select the output plots

We can combine the parameters defined previous to generate GNUplot scripts of 2D and 3D scatter-plots. The plots can print both the normalized data or the raw data (before normalizations). The user can tune tune the ranges to print and also the axis-labels of the plots. In addition, users can let the library to produce all 2D plots obtained combining all metrics defined.

Once having the parameters, filters and plots, we can run the application `ClusteringDataExtractor` to extract the data and produce the plots described before runing the cluster algorithm. The resulting plots will show all the data available, distinguishing between the duration filtered bursts, the range filtered bursts and the ones that will take part in the cluster analyses. These plots are an useful aid to fine tune the parameter filters and normalizations.

4. Select the cluster algorithm

Even though the Aggregative Cluster Refinement and DBSCAN are the two basic algorithms offered by the `ClusteringSuite` package, there is a few more clustering algorithms offered to the user. Table B.1 contains the list of these algorithms and their parameters. It is interesting to note that the Aggregative Cluster Refinement is the only algorithm that does not require any parameter and it not have to be expressed in the XML configuration file.

For further information about the different algorithms included in the package, we point to the following papers: [78] for DBSCAN algorithm, [129] for GMEANS and [130] for CAPEK, PAM and XCLARA.

Cluster Algorithm Name	Parameters
DBSCAN	epsilon, min_points
GMEANS	critical_value, max_clusters
CAPEK[1]	k
MUSTER_PAM[1]	max_clusters
MUSTER_XCLARA[1]	max_clusters

[1] libClustering includes a common interface to this algorithms offered by the MUSTER library (http://tgamblin.github.com/muster/main.html

Table B.1.: Cluster algorithms included in the libClustering and their parameters

In case of DBSCAN we provide the application DBSCANParametersApproximation to help the paremeter selection, according to the technique described in [78].

5. Execute the cluster analysis

Once defined the different elements necessary to perform the analysis, we need to execute the BurstClustering tool. The different parameters of this tool and a short description of them are listed in Table B.2. Basically, to perform a regular analysis using the cluster algorithm defined in the XML file we need to execute the command:

```
BurstClustering -d <clustering_definition.xml> -i <input_trace> -o <output_trace>
```

The tool will process the information provided in the configuration file, extract the data from the input trace, execute the cluster algorithm and then generate the required output plots, extrapolation files and the output trace. These files will be explained in the further step.

In case we want to execute the Aggregative Cluster Refinement algorithm, the command varies slightly:

```
BurstClustering -d <clustering_def.xml> -ra[p] -i <input_trace> -o <output_trace>
```

By adding the -ra parameter, the tool discards the algorithm indicated in the clustering definition XML file and then applies this different algorithm. In case we use the parameter -rap, the tool will produce, apart from the regular outputs, the traces and plots of intermediate steps of the Aggregative Cluster Refinement algorithm.

6. BurstClustering tool outputs

The BurstClustering offers three main outputs: scatter-plots of the different metrics, a cluster statistics file (including the extrapolation) and a reconstructed Paraver trace. In addi-

Parameter	Description
Required parameters	
`-d <clustering_definition.xml>`	Clustering definition XML to be used
`-i <input_trace.prv>`	Input (Paraver) trace to be used
`-o <output_trace.prv>`	Output (Paraver) trace with cluster information added
Optional parameters	
`-h`	Show help message and exit
`-s`	Performs a silent execution, without informative messages
`-m <number_of_samples>`	Performs a cluster analysis using the number of burst indicated and classifying the rest
`-a[f]`	Generates an output file with the aligned sequences (in FASTA format when using '`-af`')
`-ra[p]`	Executes the Aggregative Cluster Refinement instead of the cluster algorithm indicated in the XML. When using '`-rap`' generates plots and outputs from intermediate steps
`-t`	Print accurate timings (in $\mu seconds$) of different algorithm parts
`-e EvtType1, EvtType2,...`	Changes the Paraver trace processing, to capture information by the events defined instead of CPU bursts

Table B.2.: `BurstClustering` tool parameters

tion, it also generates the refinement tree, when using the Aggregative Cluster Refinement. Optionally, it can produce the a file with the sequence alignment and a file containing the Cluster Sequence Score values. Here we will describe briefly all of them.

The **scatter-plots** are simply GNUplot scripts that can be load using this plotting tool. As seen in previous steps, they can be 2 or 3 dimensional combinations of different metrics used to characterize the CPU bursts. In any case, the points in the scatter plots are coloured to distinguish the different clusters found. These plots are useful to observe, qualitatively, variations in the clusters with respect to the metrics used. In Figure B.4 we show 4 different plots combining 8 different hardware counters. First plot, B.4a, show the metrics used by the cluster algorithm. In the rest of combinations we can observe that the clusters represent clear isolated clouds, with a minor exception of the plot comparing Main Memory Accesses vs. L2 Data Cache Accesses, B.4c, where Cluster 4 (in red) appear in two different clouds.

The plot scripts are named using the output trace prefix plus a trailing string expressing the combination of metrics used. They have the extension `.gnuplot`. All of them use a file ended in `.DATA.csv` that contain on each line the different parameters described in the XML file plus the cluster identifier assigned for each CPU burst analysed.

The **clustering statistics** file is a CSV file that contains the number of individuals, the aggregated duration and the average duration per CPU burst, and the average values of extrapolation parameters defined in XML, for each cluster found. This file is really useful to analysed quantitatively the behaviour of the different clusters found. The clusters statistics file is named using the prefix of the input trace, but ending in `.clusters_info.csv`.

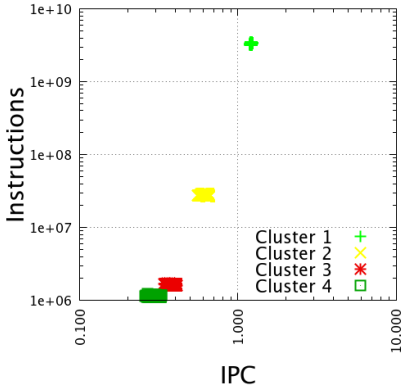
Next output that is always produced is the **output trace**. Basically, this is exactly the same input trace where all the CPU burst have been surrounded using a certain events to identify them. Thanks to these events, we can take advantage of the vast analysis power of Paraver and Dimemas to perform further analyses and correlate the clusters with all the information present on the trace. For example, we can observe the time-line distribution of the different computation regions detected. An example of Paraver time-line and its corresponding duration profile can be seen in Figure B.5. We provide a set of Paraver configuration files with pre-defined views and histograms related to cluster events.

In case we executed the Aggregative Cluster Refinement algorithm, the tool will also produce a **refinement tree** file. This file has the same prefix as the output trace and the extension `TREE.dot`. It is a text file that describes the refinement tree using the DOT language. To visualize it we require the GraphViz[2] software package. We also recommend using of the interactive tool `xdot`[3] to navigate through the refinement tree output. An example of a refinement tree can be seen in Figure B.6.

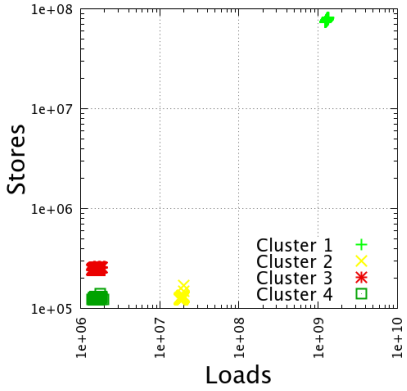
Finally, using the parameter `-a`, the tool will produce a CSV file containing the sequences obtained after applying the Cluster Sequence Score. This file, named as the output trace with the extension `.seq`, contains the sequence of the cluster identifiers (numbers) and gaps (marked as hyphens) introduced by the alignment algorithm for each task and thread present on the input trace. If use the parameter `-af`, the file will be generated in the FASTA format, transforming the first 21 clusters in an amino-acid identifier. The FASTA file can be load in

[2] `http://www.graphviz.org/`

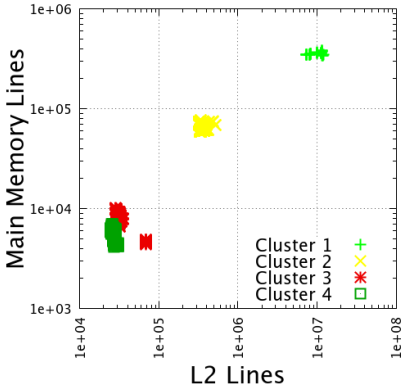
[3] `http://code.google.com/p/jrfonseca/wiki/XDot`



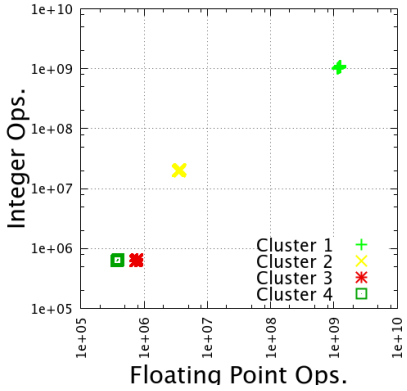
(a) Instructions vs. IPC



(b) Stores vs. Loads

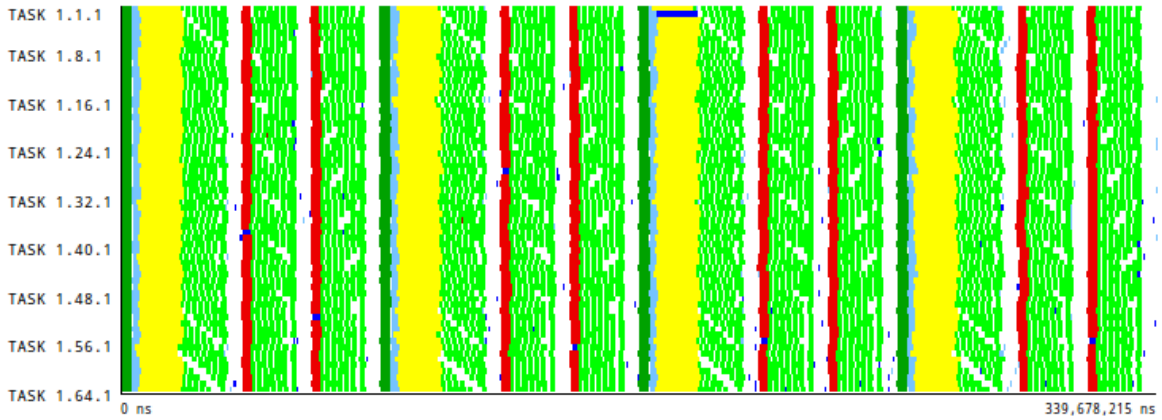


(c) Main memory accesses vs. L2 data cache acessess



(d) Integer instructions vs. Floating point instructions

Figure B.4.: Output plots produced by `BurstClustering` tool combining different metrics



(a) Time-line distribution of discovered clusters

	Cluster 1	Cluster 2	Cluster 3	Cluster 4
TASK 1.1.1	103,371,000 ns	53,163,000 ns	16,951,000 ns	11,573,000 ns
	101,279,000 ns	40,256,000 ns	19,191,000 ns	11,871,000 ns
	108,464,000 ns	54,570,000 ns	19,586,000 ns	11,895,000 ns
	108,587,000 ns	54,543,000 ns	19,852,000 ns	11,898,000 ns
	106,993,000 ns	54,539,000 ns	19,700,000 ns	11,818,000 ns
	108,652,000 ns	54,529,000 ns	19,631,000 ns	11,850,000 ns
	95,878,000 ns	54,660,000 ns	19,683,000 ns	11,892,000 ns
TASK 1.8.1	101,991,000 ns	53,945,000 ns	17,353,000 ns	11,490,000 ns
	96,053,000 ns	53,805,000 ns	20,579,000 ns	12,015,000 ns
	103,801,000 ns	53,615,000 ns	21,393,000 ns	12,348,000 ns
	109,804,000 ns	54,624,000 ns	21,986,000 ns	12,328,000 ns
	110,764,000 ns	54,577,000 ns	21,927,000 ns	12,368,000 ns
	111,561,000 ns	54,528,000 ns	21,877,000 ns	12,269,000 ns
	100,226,000 ns	54,607,000 ns	22,040,000 ns	12,320,000 ns
	96,208,000 ns	53,757,000 ns	20,143,000 ns	12,266,000 ns
TASK 1.16.1	100,976,000 ns	53,729,000 ns	19,193,000 ns	11,916,000 ns
	95,966,000 ns	53,682,000 ns	20,646,000 ns	11,920,000 ns
	104,536,000 ns	53,613,000 ns	21,416,000 ns	12,313,000 ns
	110,702,000 ns	54,448,000 ns	21,805,000 ns	12,220,000 ns
	109,171,000 ns	54,436,000 ns	22,099,000 ns	12,275,000 ns

Cluster 1

(b) Duration histogram of the clusters per application task

Figure B.5.: A Paraver time-line and profile showing information related to a cluster analysis

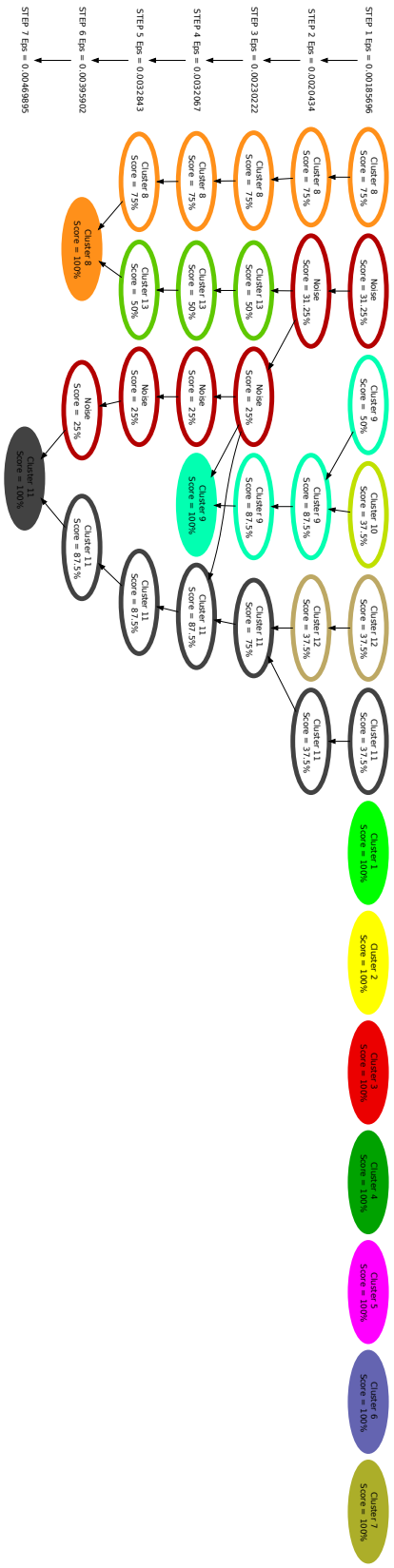


Figure B.6.: Example of a refinement tree produced by `BurstClustering` tool

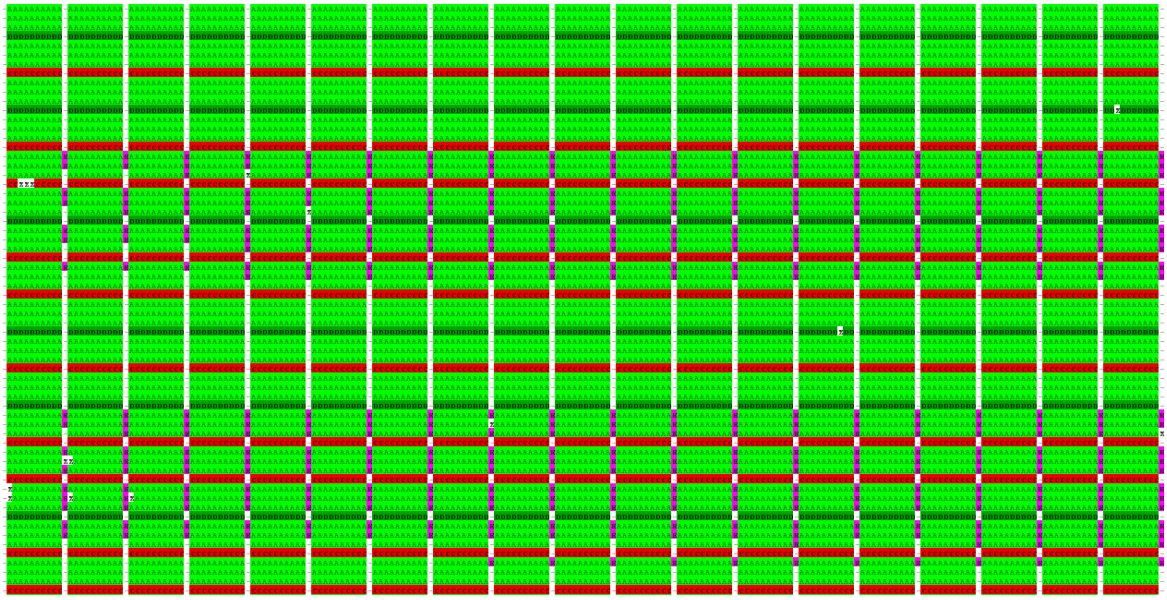


Figure B.7.: ClustalX sequence alignment window

any alignment software, such as ClustalX[4] for its visualization. In Figure B.7 we can see a ClustalX window with a set of aligned sequences.

If we use any of these two last parameters, the tool will also produce a file with the extension `SCORES.csv`, that contains the numerical results of the Cluster Sequence Score.

When using the Aggregative Cluster Refinement with the parameter `-rap` the tool will produce the plots, traces, refinement trees, sequence files and score files for each refinement step. The intermediate statistics files will not be generated and these intermediate trace files will only contain cluster events, to check the intermediate cluster distribution, but not to correlate them with other information. The intermediate files will have an inter-fix `STEPX` in their file name, to distinguish at which step (iteration) of refinement were produced.

Finally, it is interesting to note that we guarantee the colour coherence in all those outputs generated by the `BurstClustering` that use colour information to distinguish the cluster identifiers. In case of ClustalX we provide a modified version of software package with the required amino-acid colouring.

B.3. Creating the clustering definition XML

In brief, the clustering definition XML file contains the description of four elements of the clustering process: the parameters associated to each CPU burst in the trace used by cluster analysis and the extrapolation process; the filtering ranges and normalizations applied to this data; the cluster algorithm to be used; and finally, the description of the different output plots, generated as GNUplot scripts. We can see how these different parts are distributed in the XML file in Figure B.8.

[4] `http://www.clustal.org/`

```
<clustering_definition use_duration="no" apply_log="yes" normalize_data="yes"
duration_filter="10000" threshold_filter="0">

  <clustering_algorithm name="xxx">
    <!-- specific cluster algorithm parameters -->
  </clustering_algorithm>                          Cluster algorithm parametrization

  <clustering_parameters>
    <!-- 'single_event'/'mixed_events' nodes defining the CPU bursts
         parameters (dimensions) to be used by the cluster algorithm -->
  </clustering_parameters>      CPU bursts parameters to be used by cluster algorithm

  <extrapolation_parameters>
    <!-- 'single_event'/'mixed_events' nodes defining the CPU burst
         parameters (dimensions) to be used in the data extrapolation
         process -->
  </extrapolation_parameters>                  CPU bursts parameters to be extrapolated

  <output_plots all_plots="no">
    <!-- 'plot_definition' nodes defining 2D/3D plots combining the
          'clustering_parameters'/'extrapolation_parameters' --
  </output_plots>          Definition of the output plots combining the CPU parameters

</clustering_definition>                                                  End tag
```

Figure B.8.: Clustering definition XML file structure

Following the current description of the file it could be easily generated using a regular text editor or a XML editor.

Parameter selection

There are two ways to define how the parameters are read from a Paraver trace. First, the values of individual events situated at the end of the Running State, using `single_event` nodes. Second, combining the values of two different events with a basic mathematical operation, using `mixed_events` nodes.

A **`single_event`** node, see Figure B.9a, contains first two attributes: `apply_log` that indicates if a logarithmic normalization will be applied to its values; the `name` parameter is the label the will be used in the different output file. The inner node `event_type` is mandatory, to define the event type that appears in the Paraver trace. Optional nodes `range_min` and `range_max` are used to filter the CPU burst outside these boundaries. Finally, optional node `factor` is a multiplicative value so as to weight the parameter value.

A **`mixed_events`** node, see Figure B.9b, is pretty similar to the previous one, but includes two mandatory internal nodes `event_type_a` and `event_type_b`, to define the two types of events involved, and the attribute `operation` to define the mathematical operation applied to the values read. Possible operations are `+`, `-`, `*` and `/`. The operation is applied to the values of the two events defined, *before* the logarithmic normalization.

```
<single_event apply_log="yes" name="PM_INST_CMPL">
  <event_type>42001090</event_type>
  <range_min>1e6</range_min>
  <range_max>1e8</range_max>
  <factor>1.0</factor>
</single_event>
```

(a) `single_event` node structure

```
<mixed_events apply_log="yes" name="IPC" operation="/">
  <event_type_a>42001090</event_type_a>
  <event_type_b>42001008</event_type_b>
  <range_max>3</range_max>
  <factor>1.0</factor>
</mixed_events>
```

(b) `mixed_events` node structure

Figure B.9.: Nodes to define the parameters extracted from a trace

To define the CPU bursts parameters that will be used by the cluster algorithm, they have to be placed below the `clustering_parameters` node, see Figure B.8. To define those that will be used to characterize the resulting clusters (as averages in the `.clusters_info.csv` file), we have to place them below the `extrapolation_parameters` node.

If we want to use the duration of the CPU bursts as a parameter, we need to set to *yes* the attribute `use_duration` present in the root node (`clustering_definition`).

Filtering and normalization

The filtering and normalization is expressed at two points of the XML file. We have seen that the parameter definition nodes include both a range filtering and also a logarithmic normalization. The filtering information included in the extrapolation parameters is not taken into account.

The second point is the root node. In this node we find different attributes, see Figure B.8 regarding filters and normalizations. First one is `apply_log`, that indicates if logarithmic normalization will be applied to the burst duration, if used. Next one is `normalize_data`, that indicates if a final range normalization will be applied to the values of *all* parameters (independently). Next we find the `duration_filter`, expressed in μs, to discard those burst with less duration than the indicated. Finally, the `threshold_filter` is a percentage to discard all the clusters found whose aggregated duration represents less percentage of the total clusters duration than the indicated.

Output plots

Once defined the parameters used to characterize the CPU bursts, below the `output_plots` node we can define the output plots combining the different metrics.

If we set the attribute `all_plots` of this main node to *yes*, the `libTraceClustering` library will generate all possible 2D plots combining the parameters defined (clustering para-

```
<plot_definition raw_metrics="yes">
  <x_metric title="IPC" min="0.1" max="2">IPC</x_metric>
  <y_metric title="Instr. Completed" min="4e7" max="5e7">PAPI_TOT_INS</y_metric>
  <z_metric title="Memory Instructions">Memory_Instructions</z_metric>
</plot_definition>
```

Figure B.10.: `plot_definition` node of the clustering defintion XML

meters and extrapolation parameters). If we want to manually define the combinations we can use the `plot_definition` structure, see Figure B.10.

What we find first in the `plot_definition` node is the attribute `raw_metrics`. In case we applied normalization to the clustering parameters setting this attribute to "yes" indicates that the resulting plot will use the raw values of the parameters. Then we find three kind of nodes `[x|y|z]_metric`. Each of these nodes has a mandatory attribute `title` that will be used as the plot label for the corresponding axis. They have two optional attributes `max` and `min` to define the axis range. Finally, the content of each of these nodes must be the *name* attribute of any of the parameters defined previously (clustering parameter of extrapolation parameter). In case we want to use the duration, as it is defined differently from regular parameters, it has to be referenced simply using the text `Duration`.

We can combine up to three metrics to create a 3 dimensional scatter-plot, where the individuals will be distinguished in series according to the cluster identifier assigned. The same is applicable when using just two metrics (x and y). If we just define a single metric (x *metric*), the resulting plot will be a 2 dimensional plot using the cluster identifier as y axis.

Bibliography

[1] **GNU binutils Homepage**. `http://www.gnu.org/software/binutils/`.

[2] J. Mellor-Crummey, L. Adhianto, M. Fagan, M. Krentel, and N. Tallent. *HPCToolkit User's Manual*. Rice University, February 2012. For HPCToolkit 5.2.1 (Revision : 3664).

[3] N. R. Tallent, J. Mellor-Crummey, M. Franco, R. Landrum, and L. Adhianto. **Scalable Fine-grained Call Path Tracing**. In *ICS '11: Proceedings of the 25th International conference on Supercomputing*, pages 63–74, Tucson, Arizona, USA, 2011.

[4] F. Wolf and B. Mohr. **Automatic performance analysis of hybrid MPI/OpenMP applications**. *Journal of Systems Architecture*, **49**(10-11):421–439, 2003.

[5] D. G. Higgins and P. M. Sharp. **CLUSTAL: a package for performing multiple sequence alignment on a microcomputer**. *Gene*, **73**(1):237–244, 1988.

[6] H. Servat, G. Llort, J. Gimenez, and J. Labarta. **Detailed Performance Analysis Using Coarse Grain Sampling**. In *PROPER '09: Proceedings of the 2nd Workshop on Productivity and Performance*, Delft, The Netherlands, August 2009.

[7] J. G. German Llort, Harald Servat and J. Labarta. **On the usefulness of object tracking techniques in performance ana**. Technical Report UPC-DAC-RR-2013-1, Computer Architecture Department, UPC, 2013.

[8] M. Casas, R. Badia, and J. Labarta. **Automatic Analysis of Speedup of MPI Applications**. In *SC '08: Proceedings of the 22nd annual international conference on Supercomputing*, ICS '08, pages 349–358, New York, NY, USA, 2008. ACM.

[9] A. K. Jain, M. N. Murty, and P. J. Flynn. **Data clustering: a review**. *ACM Computing Surveys*, **31**:264–323, September 1999.

[10] O. Y. Nickolayev, P. C. Roth, and D. A. Reed. **Real-Time Statistical Clustering for Event Trace Reduction**. *The International Journal of Supercomputer Applications and High Performance Computing*, **11**(2):144–159, Summer 1997.

[11] D. H. Ahn and J. S. Vetter. **Scalable analysis techniques for microprocessor performance counter metrics**. In *SC '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–16, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[12] K. A. Huck and A. D. Malony. **PerfExplorer: A Performance Data Mining Framework For Large-Scale Parallel Computing**. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 41, Washington, DC, USA, 2005.

[13] R. Durbin, S. R. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press, 1998. ISBN: 9780521629713.

[14] T. Tchoukalov, C. Monson, and B. Roark. **Multiple Sequence Alignment for Morphology Induction**. In *Working Notes for the CLEF 2009 Workshop*, Corfu, Greece, September 2009.

[15] N. Shoval and M. Isaacson. **Sequence Alignment as a Method for Human Activity Analysis in Space and Time**. *Annals of the Association of American Geographers*, **97**(2):282–297, 2007.

[16] J. Gonzalez, J. Gimenez, and J. Labarta. **Automatic Detection of Parallel Applications Computation Phases**. In *IPDPS '09: Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium*, Rome, Italy, May 2009.

[17] J. Gonzalez, J. Gimenez, and J. Labarta. **Automatic Evaluation of the Computation Structure of Parallel Applications**. In *PDCAT '09: Proceedings of the 10th International Conference on Parallel and Distributed Computing, Applications and Technologies*, Hiroshima, Japan, December 2009.

[18] J. Gonzalez, J. Gimenez, and J. Labarta. **Performance Data Extrapolation in Parallel Codes**. In *ICPADS '10: Proceedings of the 16th International Conference on Parallel and Distributed Systems*, Shanghai, China, December 2010.

[19] J. Gonzalez, J. Gimenez, M. Casas, M. Moreto, A. Ramirez, J. Labarta, and M. Valero. **Simulating Whole Supercomputer Applications**. *IEEE Micro*, **31**:32–45, 2011.

[20] J. Gonzalez, K. Huck, J. Gimenez, and J. Labarta. **Automatic Refinement of Parallel Applications Structure Detection**. In *LSPP '12: Proceedings of the 2012 Workshop on Large-Scale Parallel Processing*, Shanghai, China, May 2012.

[21] **`libunwind` Homepage**. `http://www.nongnu.org/libunwind`.

[22] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. **A Portable Programming Interface for Performance Evaluation on Modern Processors**. *International Journal of High Performance Computing Applications*, **14**(3):189–204, 2000.

[23] D. Terpstra, H. Jagode, H. You, and J. Dongarra. **Collecting Performance Data with PAPI-C**. In M. S. Müller, M. M. Resch, A. Schulz, and W. E. Nagel, editors, *Tools for High Performance Computing 2009*, pages 157–173. Springer Berlin Heidelberg, 2010.

[24] A. Malony, S. Biersdorff, S. Shende, H. Jagode, S. Tomov, G. Juckeland, R. Dietrich, D. Poole, and C. Lamb. **Parallel Performance Measurement of Heterogeneous Parallel Systems with GPUs**. In *ICPP '11: Proceedings of the 40th International Conference on Parallel Processing*, pages 176 – 185, sept. 2011.

[25] M. P. I. Forum. *MPI: A Message-Passing Interface Standard*.

[26] B. Mohr, A. D. Malony, S. Shende, and F. Wolf. **Towards a Performance Tool Interface for OpenMP: An Approach Based on Directive Rewriting**. In *EWOMP '01: Proceedings of the Third Workshop on OpenMP*, 2001.

[27] B. Buck and J. K. Hollingsworth. **An API for Runtime Code Patching**. *International Journal of High Performance Computing Applications*, **14**(4):317–329, November 2000.

[28] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. **Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation**. In *PLD '05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[29] S. L. Graham, P. B. Kessler, and M. K. Mckusick. **Gprof: A call graph execution profiler**. In *SIGPLAN '82: Proceedings of the 1982 SIGPLAN symposium on Compiler construction*, pages 120–126, New York, NY, USA, 1982. ACM.

[30] **Open|Speedshop Homepage**. `http://www.openspeedshop.org/`.

[31] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. **HPCTOOLKIT: Tools for performance analysis of optimized parallel programs**. *Concurrency and Computation: Practice and Experience*, **22**(6):685–701, 2010.

[32] S. S. Shende and A. D. Malony. **The TAU Parallel Performance System**. *International Journal of High Performance Computing Applications*, **20**(2):287–311, 2006.

[33] B. Miller, M. Clark, J. Hollingsworth, S. Kierstead, S. Lim, and T. Torzewski. **IPS-2: The Second Generation of a Parallel Program Measurement System**. *IEEE Transactions on Parallel and Distributed Systems*, **1**(2):206–217, 1990.

[34] M. Geimer, F. Wolf, B. J. N. Wylie, E. Ábrahám, D. Becker, and B. Mohr. **The Scalasca performance toolset architecture**. *Concurrency and Computation: Practice and Experience*, **22**(6):702–719, April 2010.

[35] Z. Szebenyi, F. Wolf, and B. J. N. Wylie. **Space-Efficient Time-Series Call-Path Profiling of Parallel Applications**. In *SC '09: Proceedings of the 2009 ACM/IEEE Conference on Supercomputing*, Portland, Oregon, USA, November 2009.

[36] CEPBA-Tools. *Paraver. Parallel Program Visualization and Analysis tool Version 3.0. TRACEFILE DESCRIPTION*. Barcelona Supercomputing Center, June 2001.

[37] **Score-P Project Homepage**. `http://www.score-p.org`.

[38] H. Brunst and B. Mohr. **Performance Analysis of Large-Scale OpenMP and Hybrid MPI/OpenMP Applications with VampirNG**. In *IWOMP 2005: First International Workshop on OpenMP*, 2005.

[39] D. Eschweiler, M. Wagner, M. Geimer, A. Knüpfer, W. E. Nagel, and F. Wolf. *Open Trace Format 2: The Next Generation of Scalable Trace Formats and Support Libraries*, **22** of *Advances in Parallel Computing. Applications, Tools and Techniques on the Road to Exascale Computing*. IOS Press, 2012. ISBN 978-1-61499-040-6.

[40] A. Knüpfer, R. Brendel, H. Brunst, H. Mix, and W. Nagel. **Introducing the Open Trace Format (OTF)**. In *ICCS '06: Proceedings of the 6th International Conference con Computational Science*, pages 526–533, Reading, Uniterd Kingdom, May 2006.

[41] M. Wagner, A. Knüpfer, and W. E. Nagel. **Enhanced Encoding Techniques for the Open Trace Format 2**. In *ICCS '12: Proceedings of the 12th International Conference on Computational Science*, 2012.

[42] M. Itzkowitz, B. J. N. Wylie, C. Aoki, and N. Kosche. **Memory Profiling using Hardware Counters**. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, pages 17–, Phoenix, Arizona, USA, November 2003.

[43] G. Lakner, I.-H. Chung, D. G. Cong, S. Fadden, N. Goracke, D. Klepacki, J. Lien, C. Pospiech, S. R. Seelam, and H.-F. Wen. *IBM System Blue Gene Solution: Performance Analysis Tools*. IBM, September 2008.

[44] L. Adhianto, J. Mellor-Crummey, and N. R. Tallent. **Effectively Presenting Call Path Profiles of Application Performance**. In *PSTI '10: Proceedings of the First International Workshop on Parallel Software Tools and Tool Infrastructures*, pages 179–188, Washington, DC, USA, 2010.

[45] J. Mellor-Crummey, R. J. Fowler, G. Marin, and N. Tallent. **HPCVIEW: A Tool for Top-down Analysis of Node Performance**. *Journal of Supercomputing*, **23**(1):81–104, August 2002.

[46] R. Bell, A. Malony, and S. Shende. **Paraprof: A Portable, Extensible, and Scalable Tool for Parallel Performance Profile Analysis**. In *Euro-Par 2003: Proceedings of the 9th International European Conference on Parallel and Distributed Computing*, 2003.

[47] K. A. Huck, A. D. Malony, and A. Morris. **Design and Implementation of a Parallel Performance Data Management Framework**. In *ICPP '05: Proceedings of the 34th International Conference on Parallel Processing*, ICPP '05, pages 473–482, Washington, DC, USA, 2005. IEEE Computer Society.

[48] M. Geimer, B. Kuhlmann, F. Pulatova, F. Wolf, and B. J. N. Wylie. **Scalable Collation and Presentation of Call-Path Profile Data with CUBE**. In *ParCo '07: Proceedings of the Conference on Parallel Computing*, pages 645–652, Aachen/Juelich, Germany, September 2007. *Minisymposium Scalability and Usability of HPC Programming Tools*.

[49] J. M. Bull. **A hierarchical classification of overheads in parallel programs**. In *Proceedings of the First IFIP TC10 International Workshop on Software Engineering for Parallel and Distributed Systems*, pages 208–219, London, UK, UK, March 1996. Chapman & Hall, Ltd.

[50] F. Wolf and B. Mohr. **Automatic Performance Analysis of MPI Applications Based on Event Traces**. In *Euro-Par 2000: Proceedings of the 6th International European Conference on Parallel and Distributed Computing*, pages 123–132, London, UK, 2000. Springer-Verlag.

[51] F. Wolf, B. Mohr, J. Dongarra, and S. Moore. **Efficient Pattern Search in Large Traces Through Successive Refinement**. In D. L. Marco Danelutto, Marco Vanneschi, editor, *Euro-Par 2004: Proceedings of the 10th International European Conference on Parallel and Distributed Computing*, page 47. Springer-Verlag GmbH, 2004.

[52] M. Geimer, F. Wolf, B. J. N. Wylie, and B. Mohr. **Scalable parallel trace-based performance analysis**. In *EuroPVM/MPI '06: Proceedings of the 13th European PVM/MPI User's Group conference on Recent advances in parallel virtual machine and message passing interface*, EuroPVM/MPI'06, pages 303–312, Berlin, Heidelberg, 2006. Springer-Verlag.

[53] D. Böhme, M. Geimer, F. Wolf, and L. Arnold. **Identifying the root causes of wait states in large-scale parallel applications**. In *ICPP '10: Proceedings of the 39th International Conference on Parallel Processing*, pages 90–100, San Diego, CA, USA, September 2010. IEEE Computer Society.

[54] A. Espinosa, T. Margalef, and E. Luque. **Automatic Performance Evaluation of Parallel Programs**. In *PDP '98: Proceedings of the Sixth Euromicro Workshop on Parallel and Distributed Processing*, pages 43–49, Los Alamitos, CA, USA, 1998. IEEE Computer Society.

[55] J. Jorba, T. Margalef, and E. Luque. **Performance Analysis of Parallel Applications with KappaPI 2**. In *ParCo '05: Proceedings of the Parallel Computing Symposium*, Malaga, Spain, September 2005.

[56] T. Fahringer, M. Gerndt, G. Riley, and J. Täff. **Knowledge Specification for Automatic Performance Analysis**. Technical report, APART Technical Report, 2001.

[57] H. L. Truong and T. Fahringer. **SCALEA: A Performance Analysis Tool for Distributed and Parallel Programs**. In *Euro-Par 2002: Proceedings of the 8th International European Conference on Parallel and Distributed Computing*, pages 75–85, London, UK, 2002. Springer-Verlag.

[58] C. Seragiotto, T. Fahringer, M. Geissler, and H. Madsen, G.and Moritsch. **On using Aksum for semi-automatically searching of performance problems in parallel and distributed programs**. In *Euromicro 2003: Proceedings of the 11th Conference on Parallel, Distributed and Network-Based Processing.*, pages 385–392, 2003.

[59] T. Fahringer and C. Seragiotto, Jr. *Aksum: a performance analysis tool for parallel and distributed applications*. Kluwer Academic Publishers, Norwell, MA, USA, 2004.

[60] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall. **The Paradyn Parallel Performance Measurement Tool**. *Computer*, **28**(11):37–46, 1995.

[61] P. C. Roth and B. P. Miller. **On-line automated performance diagnosis on thousands of processes**. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 69–80, New York, NY, USA, 2006. ACM Press.

[62] M. Gerndt, A. Schmidt, M. Schulz, and R. Wismuller. **Performance Analysis for Teraflop Computers: a Distributed Automatic Approach**. In *Euromicro 2002: Proceedings of the 10th Conference on Parallel, Distributed and Network-Based Processing.*, pages 23–30, 2002.

[63] M. Gerndt, K. Fürlinger, and E. Kereku. **Periscope: Advanced Techniques for Performance Analysis**. In *ParCo '05: Proceedings of the Parallel Computing Symposium*, Malaga, Spain, September 2005.

[64] M. Casas, R. Badia, and J. Labarta. **Automatic Structure Extraction from MPI Applications Tracefiles**. In *Euro-Par 2007: Proceedings of the 13th International European Conference on Parallel and Distributed Computing*, **4641** of *Lecture Notes in Computer Science*, pages 3–12, 2007.

[65] F. Freitag, J. Corbalan, and J. Labarta. **A Dynamic Periodicity Detector: Application to Speedup Computation**. In *IPDPS '01: Proceedings of the 15th IEEE International Parallel and Distributed Processing Symposium*, **01**, Los Alamitos, CA, USA, 2001. IEEE Computer Society.

[66] A. Knüpfer, B. Voigt, W. E. Nagel, and H. Mix. **Visualization of Repetitive Patterns in Event Traces**. In *PARA'06: Workshop on State of the Art in Scientific and Parallel Computing*, 2006.

[67] A. Knüpfer and W. E. Nagel. **New Algorithms for Performance Trace Analysis Based on Compressed Complete Call Graphs**. In *ICCS '05: Proceedings of the 5th International Conference on Computational Science*, Lecture Notes in Computer Science, pages 116–123, 2005.

[68] P. C. Roth. *ETRUSCA: Event Trace Reduction Using Statistical Data Clustering Analysis*. Master's thesis, University of Illinois at Urbana-Champaign, 1996.

[69] H. Abdi and L. J. Williams. **Principal Component Analysis**. *Wiley Interdisciplinary Reviews: Computational Statistics*, **2**(4):433–459, 2010.

[70] R. Gorsuch. *Factor analysis*. L. Erlbaum Associates, 1983.

[71] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. **Automatically characterizing large scale program behavior**. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 45–57, New York, NY, USA, 2002. ACM Press.

[72] P. Berkhin. **Survey Of Clustering Data Mining Techniques**. Technical report, Accrue Software, San Jose, CA, 2002.

[73] R. Xu and D. W. II. **Survey of clustering algorithms**. *IEEE Transactions on Neural Networks*, **16**:645–678, 2005.

[74] J. Hartigan and M. Wong. **Algorithm AS 136: A K-Means Clustering Algorithm**. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, **28**:100–108, 1979.

[75] D. Pelleg and A. W. Moore. **X-means: Extending K-means with Efficient Estimation of the Number of Clusters**. In *ICML '00: Proceedings of the Seventeenth International Conference on Machine Learning*, pages 727–734, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[76] L. Kaufman and P. J. Rousseeuw. *Finding Groups in Data*, chapter Partitioning Around Medoids (Program PAM), pages 68–125. John Wiley & Sons, Inc., 1990.

[77] J. Joe H. Ward. **Hierarchical Grouping to Optimize an Objective Function**. *Journal of the American Statistical Association*, **58**(301):236–244, March 1963.

[78] M. Ester, H. P. Kriegel, J. Sander, and X. Xu. **A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise**. In E. Simoudis, J. Han, and U. Fayyad, editors, *KDD96: Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, pages 226–231, Portland, Oregon, 1996. AAAI Press.

[79] C. Notredame. **Recent progress in multiple sequence alignment: a survey**. *Pharmacogenomics*, **3**(1):131–144, January 2002.

[80] S. B. Needleman and C. D. Wunsch. **A general method applicable to the search for similarities in the amino acid sequence of two proteins**. *Journal of Molecular Biology*, **48**(3):443–453, March 1970.

[81] T. F. Smith and M. S. Waterman. **Identification of common molecular subsequences**. *Journal of Molecular Biology*, **147**(1):195–197, March 1981.

[82] R. Bellman. **The theory of dynamic programming**. *Bulletin of the American Mathematical Society*, (60):503–515, 1954.

[83] D. G. Higgins, A. J. Bleasby, and R. Fuchs. **CLUSTAL V: improved software for multiple sequence alignment**. *Computer applications in the biosciences : CABIOS*, **8**(2):189–191, 1992.

[84] F. Jeanmougin. **Multiple sequence alignment with Clustal X**. *Trends in Biochemical Sciences*, **23**(10):403–405, 1998.

[85] R. Chenna, H. Sugawara, T. Koike, R. Lopez, T. J. Gibson, D. G. Higgins, and J. D. Thompson. **Multiple sequence alignment with the Clustal series of programs**. *Nucleic Acids Research*, **31**(13):3497–3500, 2003.

[86] M. A. Larkin, G. Blackshields, N. P. Brown, R. Chenna, P. A. McGettigan, H. McWilliam, F. Valentin, I. M. Wallace, A. Wilm, R. Lopez, and et al. **Clustal W and Clustal X version 2.0**. *Bioinformatics*, **23**(21):2947–2948, 2007.

[87] C. Notredame, D. G. Higgins, and J. Heringa. **T-Coffee: A Novel Method for Fast and Accurate Multiple Sequence Alignment**. *Journal of Molecular Biology*, **302**(1):205–217, September 2000.

[88] T. Lassmann and E. Sonnhammer. **Kalign - an accurate and fast multiple sequence alignment algorithm**. *BMC Bioinformatics*, **6**(1):298, 2005.

[89] T. Lassmann, O. Frings, and E. L. L. Sonnhammer. **Kalign2: high-performance multiple alignment of protein and nucleotide sequences allowing external features**. *Nucleic Acids Research*, **37**(3):858–865, 2009.

[90] N. Shu and A. Elofsson. **KalignP: Improved multiple sequence alignments using position specific gap penalties in Kalign2**. *Bioinformatics*, **27**(12):1702–1703, 2011.

[91] R. C. Edgar. **MUSCLE: multiple sequence alignment with high accuracy and high throughput**. *Nucleic Acids Research*, **32**(5):1792–1797, 2004.

[92] R. Edgar. **MUSCLE: a multiple sequence alignment method with reduced time and space complexity**. *BMC Bioinformatics*, **5**(1):113, 2004.

[93] A. Joshi, A. Phansalkar, L. Eeckhout, and L. K. John. **Measuring Benchmark Similarity Using Inherent Program Characteristics**. *IEEE Transactions on Computers*, **55**(6):769–782, 2006.

[94] J. Hutter and A. Curioni. **Car-Parrinello Molecular Dynamics on Massively Parallel Computers**. *ChemPhysChem*, **6**:1788–1793, 2005.

[95] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R.Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. **The NAS Parallel Benchmarks**. Technical Report RNR-94-007, NASA Advanced Supercomputing (NAS) Division, 1994.

[96] P.-F. Lavallée, G. C. de Verdière, P. Wautelet, D. Lecas, and J.-M. Dupays. **Porting and Optimizing HYDRO to new Platforms**. Whitepaper, PRACE, December 2012.

[97] R. Teyssier. **Cosmological hydrodynamics with adaptive mesh refinement**. *Astronomy and Astrophysics*, **385**(1):337–364, 2002.

[98] N. Anquetil, C. Fourrier, and T. C. Lethbridge. **Experiments with Clustering as a Software Remodularization Method**. In *WCRE '99: Proceedings of the Sixth Working Conference on Reverse Engineering*, page 235, Washington, DC, USA, 1999. IEEE Computer Society.

[99] P. Tonella, F. Ricca, E. Pianta, and C. Girardi. **Evaluation methods for Web application clustering**. In *Proceedings of the 5th IEEE International Workshop on Web Site Evolution*, pages 33–40, 2003.

[100] Y. Liu, Z. Li, H. Xiong, X. Gao, and J. Wu. **Understanding of Internal Clustering Validation Measures**. In *ICDM '10: Proceedings of the 2010 IEEE International Conference on Data Mining*, pages 911–916, Washington, DC, USA, 2010. IEEE Computer Society.

[101] M. Halkidi and M. Vazirgiannis. **A density-based cluster validity approach using multi-representatives**. *Pattern Recognition Letters*, **29**:773–786, April 2008.

[102] C. B. Do and K. Katoh. **Protein Multiple Sequence Alignment**. In J. D. Thompson, M. Ueffing, and C. Schaeffer-Reiss, editors, *Functional Proteomics*, **484** of *Methods in Molecular Biology*, pages 379–413. Humana Press, 2008.

[103] G. Tóth. **Versatile Advection Code**. In *HPCN Europe '97: Proceedings of the International Conference and Exhibition on High-Performance Computing and Networking*, pages 253–262, London, UK, 1997. Springer-Verlag.

[104] B. Sprunt. **The Basics of Performance-Monitoring Hardware**. *IEEE Micro*, **22**(4):64–71, July 2002.

[105] J. M. May. **MPX: Software for Multiplexing Hardware Performance Counters in Multithreaded Programs**. In *IPDPS '01: Proceedings of the 15th International Parallel & Distributed Processing Symposium*, page 22, Washington, DC, USA, 2001. IEEE Computer Society.

[106] Q. Liang. **Performance Monitor Counter data analysis using Counter Analyzer**. `http://www.ibm.com/developerworks/aix/library/au-counteranalyzer/index.html`.

[107] R. Azimi, M. Stumm, and R. W. Wisniewski. **Online Performance Analysis by Statistical Sampling of Microprocessor Performance Counters**. In *ICS '05: Proceedings of the 19th International Conference on Supercomputing*, pages 101–110, Cambridge, Massachusetts, USA, 2005.

[108] W. Mathur and J. Cook. **Improved Estimation for Software Multiplexing of Performance Counters**. In *MASCOTS 2005: Proceedings of the 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 23–34, Washington, DC, USA, 2005. IEEE Computer Society.

[109] D. Vianney, A. Mericas, B. Maron, T. Chen, S. Kunkel, and B. Olszewski. **CPI analysis on POWER5, Part 1: Tools for measuring performance**. `http://www-128.ibm.com/developerworks/library/pa-cpipower1`.

[110] D. Vianney, A. Mericas, B. Maron, T. Chen, S. Kunkel, and B. Olszewski. **CPI analysis on POWER5, Part 2: Introducing the CPI breakdown model**. `http://www-128.ibm.com/developerworks/library/pa-cpipower2`.

[111] M. Rosenblum, S. A. Herrod, E. Witchel, and A. Gupta. **Complete computer system simulation: The SimOS approach**. *IEEE Parallel and Distributed Technology*, **3**:34–43, 1995.

[112] F. Bellard. **QEMU, a Fast and Portable Dynamic Translator**. In *USENIX'05: Proceedings of the annual conference on 2005 USENIX Annual Technical Conference*, pages 41–46, 2005.

[113] R. Bedichek. **SimNow: Fast Platform Simulation Purely in Software**. In *Hot Chips 16: Proceedings of the 16th Hot Chips Symposium on High Performance Chips*, Aug 2004.

[114] E. Perelman, G. Hamerly, M. Van Biesbrouck, T. Sherwood, and B. Calder. **Using SimPoint for accurate and efficient simulation**. In *SIGMETRICS 2003: Proceedings of the 2003 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, SIGMETRICS '03, pages 318–319, New York, NY, USA, 2003. ACM.

[115] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. **SMARTS: accelerating microarchitecture simulation via rigorous statistical sampling**. In *ISCA '03: Proceedings of the 30th annual international symposium on Computer architecture*, ISCA '03, pages 84–97, New York, NY, USA, 2003. ACM.

[116] E. A. Brewer, C. N. Dellarocas, A. Colbrook, and W. E. Weihl. **PROTEUS: a high-performance parallel-architecture simulator**. In *SIGMETRICS '92/PERFORMANCE '92: Proceedings of the 1992 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, SIGMETRICS '92/PERFORMANCE '92, pages 247–248, New York, NY, USA, 1992. ACM.

[117] J. E. Miller, H. Kasture, G. Kurian, C. G. III, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal. **Graphite: A Distributed Parallel Simulator for Multicores**. In *HPCA '10: Proceedings of the 16th IEEE International Symposium on High-Performance Computer Architecture*, 2010.

[118] E. S. Chung, E. Nurvitadhi, J. C. Hoe, B. Falsafi, and K. Mai. **A complexity-effective architecture for accelerating full-system multiprocessor simulations using FPGAs**. In *SIGDA '08: Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays*, FPGA '08, pages 77–86, New York, NY, USA, 2008. ACM.

[119] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. **Simics: A full system simulation platform**. *Computer*, **35**(2):50–58, Feb 2002.

[120] E. Argollo, A. Falcón, P. Faraboschi, M. Monchiero, and D. Ortega. **COTSon: infrastructure for full system simulation**. *SIGOPS Oper. Syst. Rev.*, **43**(1):52–61, 2009.

[121] L. Carrington, A. Snavely, X. Gao, and N. Wolter. **A Performance Prediction Framework for Scientific Applications**. In *ICCS '03: Proceedings of the 3rd International Conference on Computational Science*, 2003.

[122] E. A. León, R. Riesen, A. B. Maccabe, and P. G. Bridges. **Instruction-level simulation of a cluster at scale**. In *SC '09: Proceedings of the 2009 ACM/IEEE Conference on Supercomputing*, pages 1–12, Portland, Oregon, USA, November 2009.

[123] G. Zheng, G. Gupta, E. Bohm, I. Dooley, and L. V. Kale. **Simulating Large Scale Parallel Applications Using Statistical Models for Sequential Execution Blocks**. In *ICPADS '10: Proceedings of the 2010 IEEE 16th International Conference on Parallel and Distributed Systems*, ICPADS '10, pages 221–228, Washington, DC, USA, 2010. IEEE Computer Society.

[124] M. Winkel, R. Speck, H. Hübner, L. Arnold, R. Krause, and P. Gibbon. % bf A massively parallel, multi-disciplinary Barnes–Hut tree code for extreme-scale N-body simulations. *Computer Physics Communications*, **183**(4):880–889, 2012.

[125] J. Michalakes, J. Dudhia, D. Gill, T. Henderson, J. Klemp, W. Skamarock, and W. Wang. **The Weather Reseach and Forecast Model: Software Architecture and Performance**. In *Proceedings of the 11th ECMWF Workshop on the Use of High Performance Computing In Meteorology*, Reading, UK, 25 - 29 October 2004 2004.

[126] V. Springel. **The cosmological simulation code GADGET-2**. *Monthly Notices of the Royal Astronomical Society*, **364**(4):1105–1134, 2005.

[127] K. Kajantie, M. Laine, K. Rummukainen, and M. Shaposhnikov. % bf 3D SU(N) + adjoint Higgs theory and finite-temperature QCD. *Nuclear Physics B*, **503**(1–2):357–384, 1997.

[128] G. Llort, M. Casas, H. Servat, K. Huck, J. Gimenez, and J. Labarta. **Trace Spectral Analysis toward Dynamic Levels of Detail**. In *ICPADS '11: Proceedings of the 2011 IEEE 17th International Conference on Parallel and Distributed Systems*, ICPADS '11, pages 332–339, Washington, DC, USA, 2011. IEEE Computer Society.

[129] A. G. Foina, R. M. Badia, and J. R. Fernandez. **G-Means Improved for Cell BE Environment**. In *Facing the Multicore-Challenge*, pages 54–65, 2010.

[130] T. Gamblin, B. R. de Supinski, M. Schulz, R. Fowler, and D. A. Reed. **Clustering Performance Data Efficiently at Massive Scales**. In *ICS '10: Proceedings of the 24th International Conference on Supercomputing*, pages 243–252, Tsukuba, Japan, 2010. ACM.