Intelligent instrumentation techniques to improve the traces information-volume ratio

Author: Germán M. Llort Sánchez

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, 2015

Abstract

With ever more powerful machines being constantly deployed, it is crucial to manage the computational resources efficiently. This is important both from the point of view of the individual user, who expects fast results; and the supercomputing center hosting the whole infrastructure, that is interested in maximizing its overall productivity. Nevertheless, the real sustained performance achieved by the applications can be significantly lower than the theoretical peak performance of the machines. A key factor to bridge this performance gap is to understand how parallel computers behave.

Performance analysis tools are essential not only to understand the behavior of parallel applications, but to identify why performance expectations might not have been met, serving as guidelines to improve the inefficiencies that caused poor performance, and driving both software and hardware optimizations. However, detailed analysis of the behavior of a parallel application requires to process a large amount of data that also grows extremely fast.

Current large scale systems already comprise hundreds of thousands of cores, and upcoming exascale systems are expected to assemble more than a million processing elements. With such number of hardware components, the traditional analysis methodologies consisting in blindly collecting as much data as possible and then performing exhaustive lookups are no longer applicable, because the volume of performance data generated becomes absolutely unmanageable to store, process and analyze. The evolution of the tools suggests that more complex approaches are needed, incorporating intelligence to perform competently the challenging and important task of detailed analysis.

In this thesis, we address the problem of scalability of performance analysis tools in large scale systems. In such scenarios, in-depth understanding of the interactions between all the system components is more compelling than ever for an effective use of the parallel resources.

To this end, our work includes a thorough review of techniques that have been successfully applied to aid in the task of *Big Data Analytics* in fields like machine learning, data mining, signal processing and computer vision. We have leveraged these techniques to improve the analysis of large-scale parallel applications by automatically uncovering repetitive patterns, finding data correlations, detecting performance trends and further useful analysis information. Combinining their use, we have minimized the volume of performance data captured from an execution, while maximizing the benefit and insight gained from this data, and have proposed new and more effective methodologies for single and multi-experiment performance analysis.

Acknowledgements

This work was partially supported by Ministerio de Economía y Competitividad, under contracts Computación de Altas Prestaciones V (TIN2007-60625) and VI (TIN2012-34557); Departament d'Innovació, Universitats i Empresa de la Generalitat de Catalunya, under project MPEXPAR: Models de Programació i Entorns d'Execució Parał.lels (2009-SGR-980 and 2014-SGR-1051); the BSC-CNS Severo Ochoa program (SEV-2011-00067); the Intel-BSC Exascale Laboratory project; and the HOPSA: Holistic Performance System Analysis project.

Contents

List of Figures

List of Tables

Part I.

Introduction and Related Work

Chapter 1

Introduction

In this thesis we present a set of novel techniques to enhance the use of detailed performance analysis in large-scale supercomputers. The work specializes in parallel applications that run on high-performance computing (HPC) systems. This chapter presents the motivation for this research and introduces the challenges faced by using traditional detailed analysis techniques as we move into and then beyond the petascale era. Lastly, this chapter details the list of contributions made by this research to the literature, as well as the thesis' organization.

1.1. Motivation

Supercomputing has come to be a rightful cornerstone of science. The list and extent of the achievements made possible through the use of supercomputers is extraordinary: Today, we are able to accurately forecast weather conditions months in advance and predict the exact place and time of natural disasters. Advances in supercomputers have allowed extensive use of computational fluid dynamics in aerospace research, resulting in more accurate and faster air flow calculations and better vehicle designs. They have helped to speed up the drug discovery process, identifying molecular structures that have the highest potential to serve as the basis for new medications, and virtually test their effects without direct delivery to humans or animals. Supercomputing drives technological innovation, and while scientifics struggle to find answers to the big questions: "How did life begin?", "How does the brain work?", or "Will we cure cancer?"; supercomputers play a fundamental role in their research; computing realistic simulations of the formation of the Universe like in the *Evolution and Assembly of Galaxies and their Environments* (EAGLE) project [1], clearing the road for systems with brain-like intelligence as in the *Human Brain Project* [2], or detecting genetic changes responsible for the onset and progression of tumors such as the *Somatic Mutations Finder* (SMuFin) project [3]. And every scientific success draws more experts from chemistry, physics, astronomy, engineering and every other field into high performance computing, with more difficult problems to solve.

The ever-growing demand for yet more computing power, faster simulations and more

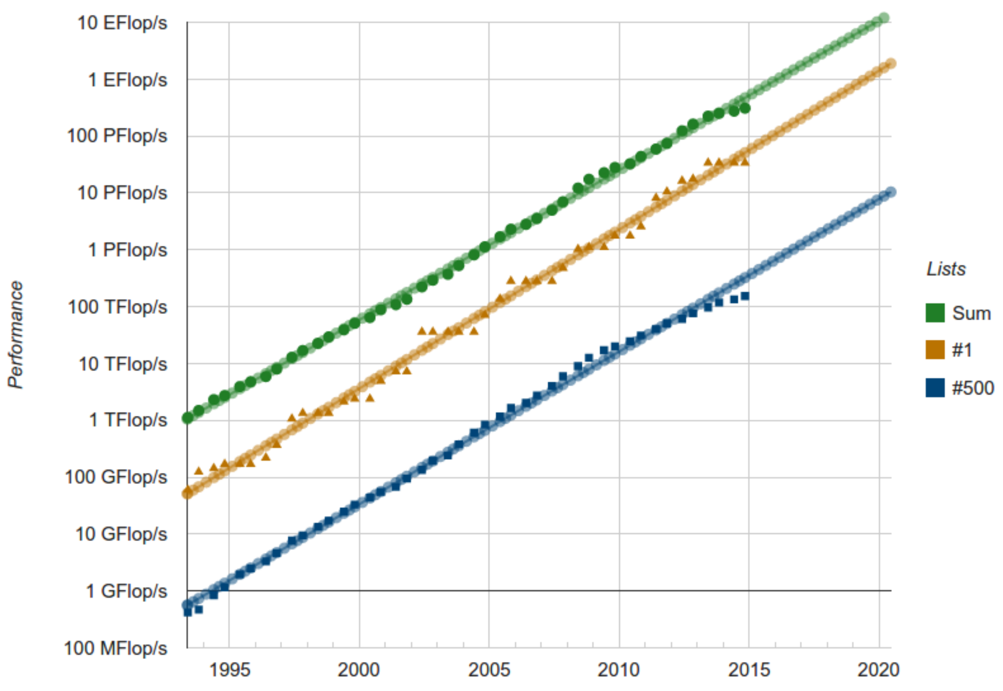


Figure 1.1.: Past and projected performance development of supercomputers as of November/2014 in the TOP500 list.

precise results has pushed supercomputers to evolve unstoppingly. Figure 1.1 shows an exponential growth of supercomputing power recorded by the TOP500 [4], a list of the 500 most powerful computer systems ranked by their performance on the LINPACK benchmark [5]. Larger and more intrincate infrastructures are being deployed constantly, enabling the study of bigger and more complex problems. But with each generation of supercomputers having a more sophisticated design, one of today's biggest challenges is to make efficient use of the hardware. For decades, computer processors benefited from faster clock speeds, but since 2005 they have mostly plateaued at less than 4GHz due to power consumption and heat dissipation constraints [6]. This has led chip designers to fit multiple processing cores into a single chip, an approach that requires software to be broken down into subtasks that run in parallel.

Parallel computing is a staple of supercomputing, but it is still difficult and it is getting more complicated. On one hand, modern high-end machines are composed of many independent systems linked with high-speed networks; each of those systems has multicore processors and, often, graphics-chip accelerators that add an entirely new level of parallel computation ability. On the other hand, the broadening of supercomputing accessilibity to other areas of science has scattered the responsibility for software development across experts of all other disciplines, who might not necessarily be experts in supercomputers, and in turn, might not be familiar with all the supercomputer's architecture details and all the issues related to parallel machines which have to be addressed. The experts in supercomputers, though, might not be necessary knowledgeable about the foundations and theory behind the problem to solve, and so it is becoming more common and important to have interdisciplinary teams working together in order to develop an efficient application. The upshot, supercomputers are not trivial to program, and their users need effortless, intuitive and powerful tools to ease the development.

These reasons have raised debugging and performance analysis tools to new heights of importance. Developers are more than ever in need of effective tools and methodologies to get detailed measurements to understand how their applications behave and what can be improved in order to achieve the best possible performance. When hundreds, thousands or even tens of thousands of processors are interacting, even small effects that cause poor performance of one or a few processes can lead to a large negative impact, in the form of load imbalance, network congestion or chain delays. All these factors keep an application from exploiting the system fully, and stepping into the next exascale supercomputing wave will worsen the situation as the number of cores increase.

But the scale of current supercomputers already poses a problem for traditional performance analysis techniques. Over the last few years, this field has witnessed a significant change of paradigm, moving from "the more, the better" idea, to somewhat the opposite. In the past, execution environments usually involved some tens of processes, and eventually no more than a few hundred. In those scenarios, the main focus was to retrieve as much information as possible to do a better analysis, and there were not strong concerns about the size of the data. Moving on to today's large scale runs, the amount of information that can be generated from a single core in an everyday simulation, that may run for hours while producing data every few nanoseconds, can already be daunting. And even in the case that the data gathered from a single process was small, the aggregation of thousands or more processes can easily result in terabytes or higher orders of magnitude of data. At this point, even the simpler task of storing the data becomes a substantial problem. But storage space is not the major limitation, as the price of hard disks is continuously dropping. The time and the effort that is required to manipulate such a large data volume increases dramatically, and sifting through all the data to find relevant information becomes a tedious and expensive task. The effective use of the data in a reasonable time frame for actionable decision-making is definitely the most important problem to solve.

According to the strategy adopted to collect and process the data, we can discern two main schools of performance analysis tools: profiling and tracing. On one hand, profiling tools measure the behavior of a program as it runs. The output is essentially a summary of cumulative first order statistics. The typical information provided would include the frequency and average duration of function calls, and also call-chains. As the summarization is often related to the source code parts where the events are observed, the size of the measured data is linear to the code size of the program.

On the other hand, tracing tools store a raw sequence of time-stamped events into files, usually for post-mortem analysis. Performance bottlenecks in parallel programs (e.g. waiting for a message or synchronization issues) often depend on the temporal relationship between events, thus requiring the full trace to get a complete understanding of the problem. However, a complete timetabled sequence of events of a parallel application usually results in a huge file which becomes difficult, sometimes impossible, to handle with most analysis tools. To reduce the size of the trace, a common approach consists in carefully controlling which information is recorded during the tracing process. For example, enabling the tracing in the interesting parts of the application and disabling otherwise; or limiting the type and number of events registered (i.e. function entries and exits, performance counters metrics, call stack information, parallel run-time activity, etc.). But this process is mostly cumbersome and requires prior knowledge of the source code of the application. Moreover, some parts of the trace are often perturbed because of the overhead of storing the data into disk, and so it is important to identify representative regions of the execution to conduct an effective analysis.

For these reasons, several authors in the literature take stance in favor of profile-based tools due to the difficulties that trace-based alternatives entail. Nevertheless, tracing tools have several advantages over profilers. Trace-based techniques allow a very detailed study of the small variations across processes and over time. These microscopic effects are critical to analyze because they are often the root-cause of a global performance issue, but profilers cloak this type of information by making averages and summarizations. Rather than relying on a predefined set of metrics averaged for the whole run, tracing tools provide all the information to conduct the analysis to specific metrics as it progresses, enabling the discovery of new unexpected patterns of inefficient behavior, and keep pulling the cord until the source of the problem is located. Therefore, there is a need to develop techniques that allow us to conveniently handle large event traces.

The purpose of this thesis is to investigate how far we can go in the use of trace-based approaches to analyze with detail the performance of large scale parallel computers. This thesis defends that even for large numbers of processors, trace-based approaches can still be applied, offering the advantages of supporting a detailed and flexible analysis. Certainly, blind tracing of large scale systems is unmanageable, thus it is necessary to explore the direction of intelligent selection of the traced information.

1.2. Objective

The main goal of this thesis is to address the scalability problem of trace-based performance analysis tools by developing new techniques and methodologies to produce and process *smart traces*. The target of a *smart trace* is to to find an appropriate trade-off between the amount of information stored from the execution of a parallel application and its relevance for the analysis.

While it would stand to reason that the more information is collected, the more precise is the analysis likely to be, this is not necessarily true. Trace data is usually repetitive due to the iterative algorithmic schemes of the applications, and not every performance measurement is pertinent to describe the observed application behavior.

Irrelevant and redundant data do not contribute to improve the overall quality of the analysis. Discarding those pieces of information aims at reducing the size of the traces, as well as the time required to perform the analysis and deliver precise results to the user. To this aim, we have developed an autonomous tracing library able to intelligently select at run-time the most useful information for its further analysis, while keeping the total trace data volume at a reasonable size.

In pursuit of this objective, we have covered the following key aspects:

- **Study, design and implement mechanisms that allow to control the size of the resulting trace.**

Many approaches have been proposed in the literature aiming at reducing the trace size. These techniques can be classified depending on which phase they are applied: during the execution, at post-processing, or in the visualization/analysis stage. While there is room for improvement in each phase, this thesis focuses on developing mechanisms applied online, during the program execution.

- **Span a wide range of granularities.**

An effective analysis of a large amount of information requires a top-down approach that goes from first getting general understanding of the program behavior to delving into the small details. Performance analysis tools should present a very high level view for the whole run and then allow to focus on any small time interval and subset of processes to inspect every microscopic phenomena that may have a significant global impact. To this end, the tracing mechanisms must be able to collect performance information at different levels of detail (from very general to very specific data), as the execution progress dictates.

- **Adaptive tracing behavior.**

The behavior of an application may change as the execution progresses. The decision of which information is useful or not for the analysis is tightly correlated with the current status of the program. In order to select the information that describes most accurately the execution at a given time, it is necessary to detect changes in the program behavior and react accordingly.

- **Minimal user intervention.**

With current instrumentation mechanisms, the users can freely enable or disable tracing features to select which type of information will be either stored or discarded (i.e. limit the trace file size, select specific metrics, ignore certain events, etc.). However, there is no guarantee that the collected information will describe the behavior of the program precisely enough in order to identify the inefficiencies that caused a poor performance. This may end up in having a fairly large trace where the actual performance bottlenecks have been masked. The objective is to let the instrumentation mechanism automatically determine which information is relevant and which techniques to apply to control the volume of collected data as the program runs, without requiring further user supervision.

- **Demonstrate the quality of the resulting traces.**

In the end, we want to obtain smaller execution traces, yet containing all the useful information for the analysis. Irrelevant data can distort the trace quality, making the analysis slower and more difficult. Our objective is to demonstrate that discarding this information enables to a better comprehension of the behavior of a parallel application, and the analysis task improves in terms of precision and simplicity.

1.3. Contributions

A significant part of our work has mainly consisted in developing new analysis mechanisms to inspect the traced data in real-time, to take intelligent decisions on whether to store, transform or discard it. The result is a *smart trace*: the minimum amount of information that best describes the application behavior. From there, we have defined new techniques and methodologies to improve the analysis of the captured data from several angles, all aimed at extracting useful and comprehensive insight about the program behavior.

In this work, we present the following original contributions:

- **The introduction of a framework to perform on-line analysis of extreme scale parallel applications.**

We have developed a tracing framework with autonomous capabilites to control when, what and how performance data generated during the execution of parallel application is stored. The intelligence of this system lays in the automatic analyses that are automatically conducted, currently based on machine learning and signal processing techniques. Furthermore, the system presents a modular construction that enables to be extended with new plug-ins of analysis. The proposed implementation follows an extremely scalable design that has been proven to handle very large executions of tens of thousands of cores.

- **A method to determine at run-time relevant regions of the execution.**

The method proposed leverages machine learning (cluster analysis) techniques to automatically detect performance trends in the computing phases of the applications. The analysis on the performance behavior of the computations of the program enables us to identify which are the most relevant phases of the code, and focus the tracing process to obtain information from the most interesting regions.

- **The design and validation of a new massively parallel algorithm to improve the scalability of on-line cluster-based analysis.**

Cluster analysis of large volumes of performance data is unsuitable for on-line application due to the expensive computational cost. We have designed a new hierarchical distributed algorithm to scale cluster-based analysis to a large number of cores. One of the main advantages of our approach is that it does not require an explicit pre-process of the data. Each parallel process is able to analyze their own self-taken performance measurements, eliminating the need to redistribute data between them, and thus making this solution very convenient to be applied on-line. Moreover, our approach increases by several orders of magnitude the volume of data that can be analyzed in near real-time, improving the quality and precision of the analysis results.

- **The dynamic generation of complementary performance reports to enrich the analysis.**

We introduce a new representation for the clustering results based on the idea of heat maps, where the temperature indicates how many parallel processes support the different areas of the clustering space. The temperature of the clusters enables quick identification of the most frequent performance behaviors among parallel tasks, and easy characterization of the source of variabilities behind very disperse clusters. This development also gives support to metrics extrapolation techniques, that enable to characterize the clusters with many more metrics than those that can be gathered in a single run, as well as building break-down models based on them such as the architecture impact and the instructions mix stack charts. Performance reports are produced periodically, providing extra insight about the structure of the application at different time intervals and its evolution over time.

- **A method to generate multi-detail traces.**

A vast majority of high-performance computing applications implement algorithms, procedures and numerical methods that are iterative in nature. The proposed method applies signal processing techniques to exploit this characteristic, enabling to dynamically identify repetitive patterns in the program, select representatives to trace in detail, and summarize the data for the repeating phases. This method results in a compact trace that fully covers the whole execution, with selected intervals traced at very fine-grain detail, and other regions where the data is summarized, filtered out, or aggregated at different levels of detail.

- **The definition of a new methodology to improve comparative and multi-experiment analysis.**

Understanding the possible changes in behavior that an application can undergo over time requires to integrate observations from multiple execution intervals. Analogously, understanding the impact of different settings on the application performance often requires to evaluate the results of multiple experiments. If the amount of performance data gathered from a single execution region can already be large, which makes it more difficult to understand, the problem intensifies if we have to contrast uncorrelated information from several regions or experiments. The proposed methodology employs object tracking techniques to provide an intuitive and comprehensive way to present and compare the performance of different execution scenarios. This approach enables the analyst to study the impact of virtually any configuration on the application performance without prior knowledge of the program; compare and correlate performance data between experiments; determine the best setup to meet specific performance requirements; follow the evolution of the program over time; and ultimately helps to gain better understanding of the application behavior, much beyond what can be learned from a single experiment.

- **A method to perform all-in-one series of experiments.**

Running multiple experiments is often necessary for an effective analysis, but this might be expensive in terms of time, computational resources and the amount of information produced. This method relies on active measurement techniques to introduce controlled interferences into the application execution to simulate different execution conditions, enabling to test multiple scenarios under the same program run. In this way, just one program execution is enough to extract all the necessary samples of performance data that would have otherwise required many experiments. The potential of this technique is not limited to measure the program behavior, but enables to provide dynamic feed-back to the parallel run-time to optimize the resources and improve the program performance.

- **Validation of the quality of the smart traces.**

We present multiple studies of proxy applications, benchmarks and real production codes that demostrate that it is possible to greatly reduce the amount of traced data, in favor of a more directed and efficient analysis, focused on the zones of real interest, providing useful insight and concise recommendations to improve the application development, and performed in a timely manner. In some cases of study, the knowledge gained has materialized in technical reports that were willingly forwarded to the program's developers to contribute to improving the application's efficiency.

1.4. Thesis organization

This document is structured in four main parts.

Part I introduces the work by defining the motivation, objectives and contributions in Chapter 1, and covering background knowledge and the previous work in the field of performance analysis that has been explored to improve the scalability of trace-based tools in Chapter 2.

Part II focuses on the development of new techniques to produce smart traces that obtain the maximum relevant information in the minimum volume of data possible for a single execution. Chapter 3 introduces the on-line analysis framework that we propose to conduct automatic analyses on the performance data as it is being collected, in order to intelligently select which data is more relevant. In Chapter 4, we present a clustering-based analysis technique that is able to identify at run-time relevant phases of the execution and the most important computations of the application, an information that we use to automatically reduce the amount of traced data. In Chapter 5, we present a technique based on spectral analysis of signals to detect periodic behavior in the program execution. This information enables us to further reduce the amount of traced data, as well as obtaining a full characterization of the whole execution at multiple levels of detail.

Part III focuses on the development of new techniques and methodologies to improve several facets of the analysis task. In Chapter 6, we present a new parallel implementation of a density-based clustering algorithm that enables to perform this kind of analysis at large scale, aiming at improving the scalability and the quality of the analysis. Chapter 7 presents a novel technique that leverages object tracking concepts to compare and contrast performance observations that may belong to multiple execution intervals or different experiments, easing the conduction of comparative analyses and preventing the data explosion problem of processing high amounts of data from several sources. In Chapter 8, we extend this technique to simulate different experiments in just a single run, minimizing the costs in time and resources. Moreover, this preliminary work opens the possibility of providing dynamic feed-back to the parallel run-time, taking advantage of the previous analysis techniques to give hints on how to tune the execution settings to improve the program performance.

Lastly, Part IV compiles the contributions of our work, and discusses further research and development opportunities in Chapter 9.

Additionally, Appendix A presents the user guide of the proposed on-line analysis framework. Appendix B describes a middleware software that we have produced to facilitate the inclusion of new automatic analyses to run on top of the on-line framework. Finally, Appendix C presents the user guide for the tracking-based analysis tool developed on top of the work presented in Chapter 7.

1.5. Publications

[7] G. Llort, J. Gonzalez, H. Servat, J. Gimenez, and J. Labarta. **On-line detection of large-scale parallel application's structure**. In *24th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010, Atlanta, Georgia, USA*, pages 1–10, 2010

[8] G. Llort, M. Casas, H. Servat, K. Huck, J. Gimenez, and J. Labarta. **Trace Spectral Analysis Towards Dynamic Levels of Detail**. In *17th IEEE International Conference on Parallel and Distributed Systems, ICPADS 2011, Tainan, Taiwan*, pages 332 – 339, 2011

[9] G. Llort, H. Servat, J. González, J. Giménez, and J. Labarta. **On the Usefulness of Object Tracking Techniques in Performance Analysis**. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 29:1–29:11, New York, NY, USA, 2013. ACM

[10] G. Llort, H. Servat, J. Gonzalez, J. Gimenez, and J. Labarta. **Studying Performance Changes with Tracking Analysis**. In *Tools for High Performance Computing 2014*, pages 175–209. Springer International Publishing, 2015

[11] G. Llort, J. González, H. Servat, J. Gimenez, and J. Labarta. **Distributed tree-based implementation of DBSCAN cluster algorithm for on-line performance analysis**. Technical report, Polytechnic University of Catalonia - BarcelonaTech, 2015

[12] G. Llort, M. Casas, M. Garcia, V. Lopez, H. Servat, J. Gimenez, and J. Labarta. **On-line Active Measurement to Perform All-In-One Series of Experiments**. Technical report, Polytechnic University of Catalonia - BarcelonaTech, 2015

Chapter 2

Background and Context

In this chapter we describe several generalities and basic concepts about parallel programming and performance analysis that are widely used through this document. We present a taxonomy of the state-of-the-art performance tools and the challenges they face towards the analysis of large-scale applications. Then we discuss the different directions that trace-based tools have explored to deal with the scalability problem. More and more sophisticated solutions are being applied to deal with this issue, and the more advanced ones have come to be known as Performance Analytics, a term that refers to the application of techniques from the fields of statistics, machine learning, signal processing or image recognition; to automatically extract useful insight from the data without requiring high expertise to manipulate the large amount of information. Finally, we explain how the main goals of this thesis align with the current state-of-the-art and put the presented work in context.

2.1. Introduction to parallel programming

With the onset of multicore processors and the more complex computer architectures that evolved from there, parallel programming has become increasingly important. Simply put, parallel programming is a computer programming technique for writing programs which can make simultaneous use of multiple computing resources to solve a problem. These computing resources are typically a single machine with multiple processors, and an arbitrary number of such machines connected by a network. From a hardware point of view, the architecture of these machines can be generally classified into two main categories: shared memory and distributed memory systems [13].

Shared memory systems refer to a multiprocessing design where several processors access all memory as a single global address space. Multiple processors can operate independently but share the same memory resources, so any change in a memory location caused by one processor is visible to all others. Distributed memory systems refer to a multiple-processor computer system in which each processor has its own private memory. When a process needs to access data that is stored in another processor they need to communicate, and so this architecture requires a communication network to connect inter-processor memory.

The largest and fastest parallel computers in the world today employ a hybrid distributed-shared memory architecture. Multiple shared memory nodes (possibly equipped with accelerators) are networked together to make larger clusters. This design combines the advantages of both architectures, which essentially results in an increased system scalability. For this reason, current trends [4] seem to indicate that this type of hybrid memory architecture will continue to prevail in future Exascale systems. However, the hybrid architectures come with a major disadvantage: an increased complexity to program, synchronize and communicate all the system's components.

Parallel programming models appeared as an abstraction of the hardware design to facilitate to program parallel computers. Generally speaking, any programming model could be applicable despite the underlying architecture, but there are combinations of hardware and software that are naturally more suited to work together. For distributed architectures, the most widely used programming paradigm is, by far, the message-passing model. The central idea of the message-passing paradigm is to have multiple autonomous processes that can be running on different computers and have the ability to interact exchanging messages. Message passing may be implemented by various mechanisms, like Remote Procedure Call (RPC) [14], Common Object Request Broker Architecture (CORBA) [15], Java Remote Method Invocation (Java RMI) [16] or D-BUS [17], but the de facto standard is the Message-Passing Interface (MPI) [18].

The MPI standard defines a set of routines to support various parallel operations such as point-to-point communication, collective communications and parallel I/O operations. Unlike other programming paradigms that allow to parallelize a code semi-automatically, writing an MPI program requires to explicitly insert calls to the MPI API into the code, program how the processes are going to interact, and consider the semantics of which parts of the code will be executed by the different processes. While the development of MPI codes requires more work than other paradigms, it also enables to develop more realiable and scalable code since the programmer has total control of all aspects of the program. Well-written MPI codes can scale to thousands and more processes, and the scientific community agrees that MPI will continue to play a major role in exascale systems [19]. It is essential to be able to identify, understand and overcome the performance problems happening at large scale, and for this reason, the work developed in this thesis focuses on the performance analysis of MPI applications.

In any case, MPI is often used along with other programming models to take full advantage of the common hybrid architectures. A typical scheme is to combine MPI to communicate between nodes with a shared-memory programming model to exploit the multi-core resources inside each single node, such as OpenMP [20]. OpenMP is a set of compiler directives and callable runtime library routines that enables to run certain parts of the code in parallel without explicitly managing (creating, destroying, assigning) threads. Basically, OpenMP adds thread-level concurrency abstracting the developer from the complexity of managing threads, splitting some program tasks into parts and handing off each of these parts to different threads automatically. Important extensions to the OpenMP standard are being developed, such as OmpSs [21]. OmpSs is an effort to extend OpenMP with new features like asynchronous parallelism and heterogeneity (the ability to manage accelerator devices like GPUs). It also incorporates the concept of task-based parallelism, allowing to express data-dependencies between the program's tasks. Task dependencies are used to decide how the program will run in parallel. Given a set of interdependent tasks and a set of available threads, each thread repeatedly selects a task with no unsatisfied dependencies and executes it.

These programming paradigms are also widely used today, and while our work mainly targets MPI parallelism and its variants, it also supports the combination of other programming models as the ones stated.

2.2. The parallel performance analysis field

The raise in parallel programming brought new questions to the fore. Am I using the parallel resources efficiently? Which of the many individual components may be causing bottlenecks for my execution? How can I further improve my program? The role of performance analysis tools is to answer all of these questions.

Performance tools comprehend several techniques and methodologies that assist the analyst in understanding the behavior of parallel applications, measuring different performance parameters, detecting hotspots in the program, identifying the root source of performance inefficiencies, and driving the task of code optimization. In this section, we give a general overview of the existing tools and analysis strategies, classifying them according several fundamental aspects: the type of data used to characterize the performance, and how the data is acquired, stored and processed.

2.2.1. Definition of performance data

When we talk about performance data, we actually refer to metrics that measure how well a parallel program behaves. The simplest of these metrics would be the total time to solution, and basically, this is first and foremost metric that the user would want to improve. However, understanding why a program is underperforming often requires to inspect other metrics that explain in detail how specific sub-parts of the program behave.

A first approach to determine how the different parts of the program behave consists in measuring the time invested in the different subroutines, loops or other program phases. To take these measurements we can make use of the timing mechanisms provided by the operating system, for example, the programmable clock interrupts. This information is useful to detect phases that might be abnormally long. Going one step further, the OS can also report metrics regarding the state of the system's resources. For example, these may come from the `getrusage()` or the `mallinfo()` system calls, which return resource usage statistics for the calling thread regarding CPU time used; and page faults, context switches and memory usage statistics; respectively.

Further performance data can be obtained directly from the microprocessor. Hardware performance counters are available on most modern microprocessors, and they are usually accessible through interfaces like PAPI [22] or PMAPI [23]. These counters exist as a small set of registers that count events and occurrences of specific signals related to the processor's function, such as the number of instructions executed, cycles elapsed, memory cache misses or branch misspredictions. This type of information facilitates to correlate the efficiency of the program with the underlying architecture. More recently, due to the need to better understand new hardware, we can also find performance counters for hardware components other than the CPU, like network performance counters for the Myrinet or the Infiniband networks [24]; or the Nvidia CUDA GPU's [25].

The metrics regarding the parallel runtime are varied and their accessibility usually depends on the programming model implementation. For example, in MPI message-passing applications it is normal to measure the number of messages sent or received, the size of the messages, the number of collective operations, etc. To this end, MPI provides the MPI profiling interface (PMPI) [26] to intercept the MPI calls and extract these values. While not all programming models make their internal performance metrics publicly available, their trend is to become more open and facilitate these measurements, as in the OMPT standard effort [27].

And for all these metrics to be most useful, it is necessary to be able to attribute the observed performance to the application source code. Location information can be obtained via the Program Counter (PC) of the CPU, or the call path, that includes the list of all active subroutines which can be accessed using stack unwinding tools such as libunwind [28], StackwalkerAPI [29], or the system-dependent stack trace methods like Linux `backtrace()`.

Beyond this list, there are a variety of metrics to evaluate specific performance elements like I/O performance [30], energy consumption [31], etc. It is not the aim of this section to present every performance metric available, but those commonly used by most performance tools.

2.2.2. Data collection techniques

There are two main techniques to extract performance data from a parallel application: sampling and code instrumentation. The difference between them is that the first method periodically takes statistical data to give you an overview of where your application is spending the most time, while the latter integrates into the code, therefore being able to deliver precise measurements of the number of times a method has been called, the time it needed to perform, and more than stastistical estimates.

More specifically, sampling techniques consist in taking random measurements at periodic time intervals or when an event is triggered by an alarm. The interrupts generated by a sampling mechanism generally add negligible overhead to the total execution time, because the perturbation is introduced only in the actual measurement, which can be easily controlled by reducing the sampling frequency. Low sampling frequencies (e.g. 100 Hz or below) are less disruptive, but also the statistical information collected becomes less accurate.

On the other hand, code instrumentation is generally more disruptive, but allows to record all the events regarding the program status. Instrumentation consists in injecting monitoring probes into different points of the program. The typical points to instrument are the entries and exits of the routines, before and after calls to external libraries, or other relevant phases such as loops. Whenever the execution flow goes through any of these points, the monitors trigger and take their measurements. In this case, one has to be aware not to instrument very fine-grain routines that are called very often (e.g. a simple `printf`) to avoid introducing large perturbation.

There are many mechanisms to perform code instrumentation that basically fall into two categories: source code and binary instrumentation. Source instrumentation comprehends the basic approach of manually modifying the source code to insert the monitors, as well as compiler-based techniques to automatically insert the monitors at the level of intermediate language representation like gprof [32], and code-to-code compilers that return instrumented source code like OPARI [33, 34]. Binary instrumentation can be achieved through binary rewriting tools like Dyninst [35] or SIGMA [36], library interposition through static linking or dynamic preloading, and dynamic binary memory image modification at load time using tools like Dyninst [35] or PIN [37].

Overall, sampling techniques are generally considered suitable to perform first-line analyses to discover the relative percentage of time spent in frequently-called methods, while code instrumentation is considered useful to discover high-level details of the interesting methods, enabling an in-depth analysis. Both approaches entail advantages, yet they share one major disadvantage that regards to the execution coverage. Deciding an appropriate sampling frequency might not be trivial, and a bad choice may lead to coupling the sampling interrupts with the application periodic behavior, which in the end may result in having a non-representative sample space because all samples were taken from the same points of the program. Instrumentation, on the other hand, is driven by the location of the monitors. If a monitor is never installed in a particular point of the execution, then instrumentation collects no data at that point.

A recent approach that is gaining importance is to combine instrumentation and sampling to offer the benefits of better execution coverage with reduced perturbation, such as the techniques presented in [38, 39] that employ mixed information.

2.2.3. Approaches to store the data

Regardless of which is the mechanism employed to collect performance data, there are basically two strategies to store it. The first strategy consists in summarizing the data, an approach that gives rise to *profiling* tools. The second approach is to store the data raw for later processing, which opens the way for *tracing* tools.

Profiles

An application profile is a summary of first-order statistics that concisely state which parts of the program consume most of the CPU cycles. Profilers can produce several different output styles, and some examples follow. *Flat profiles* show how much time was spent executing each program routine, and how many times that function was called. *Call graph profiles* show which functions called which others, and how much time each function used when its subroutine calls are included. *Line-by-line profiles* report statistics for each invididual source code line. And *annotated source profiles* display source code labeled with execution counts. The fundamental characteristic, which is common to all types of profiles is that the temporal information regarding when the data was collected is lost in the summarization process, and so they report averages for the whole execution.

`gprof` [32] is the GNU profiling tool. It is considered the de facto standard for profilers, and has appeared on the list of the 50 most influential PLDI papers of all time [40]. It provides information about the number of times each function in the code was called, the length of time that was spent within that function, and also information about the call-graph structure of the code. `gprof` operates as a hybrid instrumenting and sampling profiler. It requires code to be specifically compiled with profiling support to gather caller-function data, and also relies on POSIX timers and signalling mechanisms to take time measurement samples.

HPCToolkit [41] and Open SpeedShop [42] are a collection of tools that rely on statistical sampling techniques to generate profiles of parallel applications, taking into account the performance results from all processes or threads in the parallel execution. Moreover, they provide the ability to profile not only the program subroutines, but also basic blocks of code, or even invididual source code lines.

Periscope [43] is an online profiler that automatically evaluates performance properties and tests hypotheses about typical performance problems, which are reported to the user if detected.

mpiP [44] is a lightweight profiling library that collects statistical information about MPI functions in message-passing applications. Unlike other tools that focus on the user code, this tool focuses on reporting metrics regarding the parallel run-time, like the percent of a task's time attributed to MPI calls, where each MPI call is made within the program, and callsite statistics.

The TAU Performance System [45] is an extensive profiling tool-set that relies on instrumentation mechanisms to capture additional information from the parallel run-time (e.g. MPI and OpenMP) and performance hardware counters. To overcome one of the main limitations of profilers regarding to not storing any temporal information, TAU introduced the concept of *phase profiles*. These are partial profiles extracted at different phases (i.e. time intervals) of the execution that enable to study performance variabilities that may occur over time.

Scalasca [46, 47] is a software tool that also offers the collection of regular profiles with metrics from MPI, OpenMP and hardware counters through instrumentation. Scalasca introduced the *time-series call-path profiles* [48], a call-graph oriented version of the phase profiles mentioned before, that enable to study the performance behavior variations as the execution progresses.

Event traces

Event traces record the entire dynamic execution behavior of a program up to any required level of detail. They store each occurrence of specified events, such as the entry and exit to subroutines, the invocations to the parallel run-time, the performance hardware counters values for a given point in time, or call-path information to attribute the observed performance to points in the source code. Unlike profilers, event traces capture the temporal and spatial relationship between individual events, and so they allow application developers to study the time-varying behavior of the application and the performance variations that may occur across processes. Although event traces offer highly-detailed information regarding the program performance, they typically result in very large amounts of data.

Relevant examples of trace-based performance tools are Scalasca [46, 47], a tool to perform post-mortem analysis of event traces and automatically detect performance critical situations; TAU [45], which is capable of gathering performance information through instrumentation of functions, methods, basic blocks, and statements; or Vampir [49], an interactive event trace visualization software which allows to analyze parallel program runs with various graphical representations in a post-mortem fashion. These tools used to implement their own instrumentation mechanisms and native formats to store the data, but they recently adopted a common infrastructure for event trace recording under the joint Score-P initiative [50, 51], which adopts the Open Trace Format version 2 (OTF2) [52] for data storage.

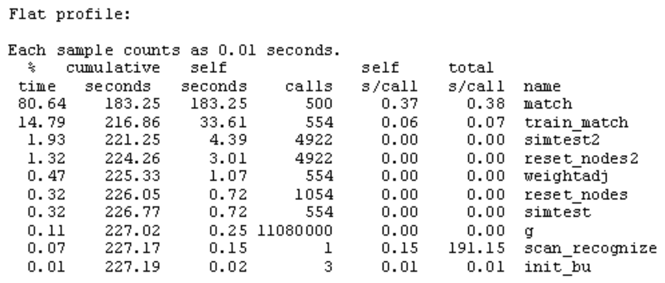
The BSC Tools [53] is a suite of parallel performance analysis tools also based on traces. This ecosystem revolves around the Paraver trace format [54], a structured text file that stores a list of timestamped key-value events without any semantics. This distinguishing feature enables a flexible data browsing that does not rely on hardwired metrics, but allows to program new metrics to give different meanings to the data as the analysis progresses.

While trace-based tools mainly rely on instrumentation techniques to capture the complete stream of events from the execution, it is also possible to generate traces from sampling data. The Oracle Solaris Studio [55] comprises a set of tools for analyzing the application performance using tracing and sampling mechanisms to collect information regarding the user routines. HPCToolkit has incorporated sampled call-stack data into their traces that can be visualized with the `hpctraceviewer` [56] tool. The BSC Tools also added sampled performance hardware counters and call-stack data into their traces to be able to report the instantenous performance between instrumentation points [38].

2.2.4. Analysis of performance data

Most performance tools analyze the data post-mortem, i.e. after the execution has finished. The type of analyses that can be conducted very much depends on the choices taken prior to the execution regarding what data to collect during the run. Also, the way the information is presented determines the type of observations and hypotheses that the user will be able to extract about the program performance.

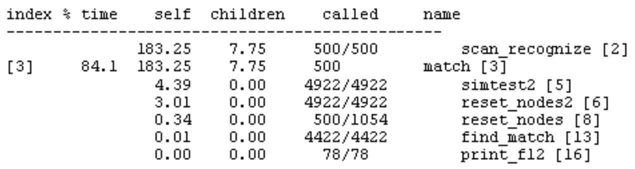
While the possible presentations of the results are many and very varied, there are two predominant choices that comprehend most of the tools. Profiling data, which typically consists of summary statistics for different parts of the code, is usually presented using *tabular* representations. Otherwise, tracing data, which exhibits the time evolution of the program and the relationships between processes and events, is usually presented using *timeline* representations.



```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 80.64    183.25   183.25      500     0.37     0.38  match
 14.79    216.86    33.61      554     0.06     0.07  train_match
  1.93    221.25     4.39     4922     0.00     0.00  simtest2
  1.32    224.26     3.01     4922     0.00     0.00  reset_nodes2
  0.47    225.33     1.07      554     0.00     0.00  weightadj
  0.32    226.05     0.72     1054     0.00     0.00  reset_nodes
  0.32    226.77     0.72      554     0.00     0.00  simtest
  0.11    227.02     0.25 11080000     0.00     0.00  g
  0.07    227.17     0.15        1     0.15   191.15  scan_recognize
  0.01    227.19     0.02        3     0.01     0.01  init_bu
```

Figure 2.1.: GNU gprof's flat-profile showing a list of the most time-consuming routines



```
index % time    self  children    called     name
-----------------------------------------------
               183.25    7.75     500/500        scan_recognize [2]
[3]     84.1  183.25    7.75     500         match [3]
                 4.39    0.00    4922/4922       simtest2 [5]
                 3.01    0.00    4922/4922       reset_nodes2 [6]
                 0.34    0.00     500/1054       reset_nodes [8]
                 0.01    0.00    4422/4422       find_match [13]
                 0.00    0.00      78/78         print_fl2 [16]
```

Figure 2.2.: GNU gprof's call-graph showing caller-callee relationships for function match

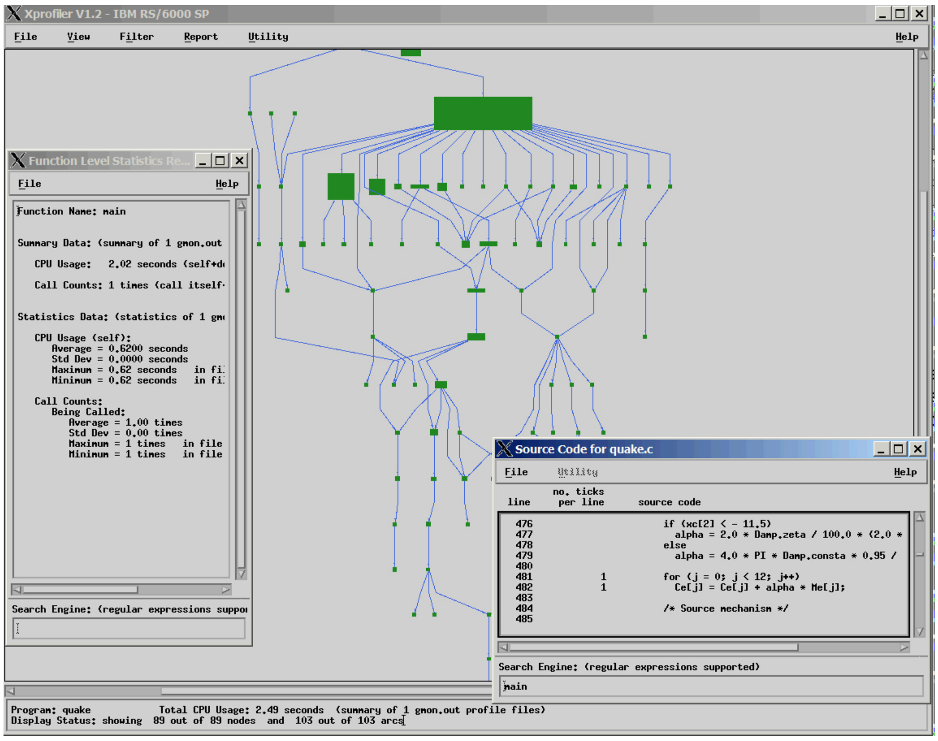
Presentation of profiles

In general, profiles are presented as tables. The different code lines, routines or phases of the program are separated by rows, and the different metrics measured are displayed by columns. This simple arrangement enables to focus, for example, on a particular routine of the program and get a quick view of all the metrics measured for it, or otherwise, focus on a given metric and very quickly compare the differences between all parts of the program with respect to that metric. This kind of quick interpretations make profilers a good choice for first-line analyses to get an initial overview of the program performance.

Figure 2.1 shows a flat profile produced by `gprof`. In this case, the output is a list of program subroutines sorted by the percentage of execution time that they represent. Columns display several metrics for each subroutine such as the average duration per call excluding the time spent in second-level routines (exclusive time), including the time spent in second-level routines (inclusive time), and the number of invocations.

Figure 2.2 shows a call-graph profile also produced by `gprof`. The call-graph provides metrics regarding the routine invocations and displays caller-callee relationships. Tools like XProfiler [57] can interpret `gprof`'s results and construct a graphical display of the functions within an application, as shown in Figure 2.3. The nodes represent functions and arcs represent function calls. The relative width and height of nodes show the execution time of a function including and excluding the execution time of its descendants. This type of representation helps users identify the functions that are the most CPU-intensive and facilitates to find the application's performance-critical areas.

Flat profiles can also be displayed graphically. TAU's profiler ParaProf [58] uses simple bar charts linking the metrics with the source code. Figure 2.4 shows the most time consuming routines, including the calls to the MPI parallel run-time. In addition, ParaProf can display comparisons of profiles from different executions of the same application. To this


Xprofiler V1.2 - IBM RS/6000 SP
File View Filter Report Utility Help
Function Level Statistics Re...
File Help
Search Engine: (regular expressions suppo
Source Code for quake.c
File Utility Help
Search Engine: (regular expressions supported)
main

```
Function Name: main


Summary Data: (summary of 1 gmon.out

   CPU Usage:   2.02 seconds (self+de

   Call Counts: 1 times (call itself·


Statistics Data: (statistics of 1 gmo

   CPU Usage (self):
      Average = 0.6200 seconds
      Std Dev = 0.0000 seconds
      Maximum = 0.62 seconds   in fi
      Minimum = 0.62 seconds   in fi

   Call Counts:
      Being Called:
         Average = 1.00 times
         Std Dev = 0.00 times
         Maximum = 1 times   in file
         Minimum = 1 times   in file
```

```
         no. ticks
 line    per line     source code

  476                     if (xc[2] < - 11.5)
  477                       alpha = 2.0 * Damp.zeta / 100.0 * (2.0 *
  478                     else
  479                       alpha = 4.0 * PI * Damp.consta * 0.95 /
  480
  481          1          for (j = 0; j < 12; j++)
  482          1            Ce[j] = Ce[j] + alpha * Me[j];
  483
  484                     /* Source mechanism */
  485
```

```
Program: quake          Total CPU Usage: 2.49 seconds  (summary of 1 gmon.out profile files)
Display Status: showing  89 out of 89 nodes  and  103 out of 103 arcs
```

Figure 2.3.: IBM XProfiler showing a call-graph representation

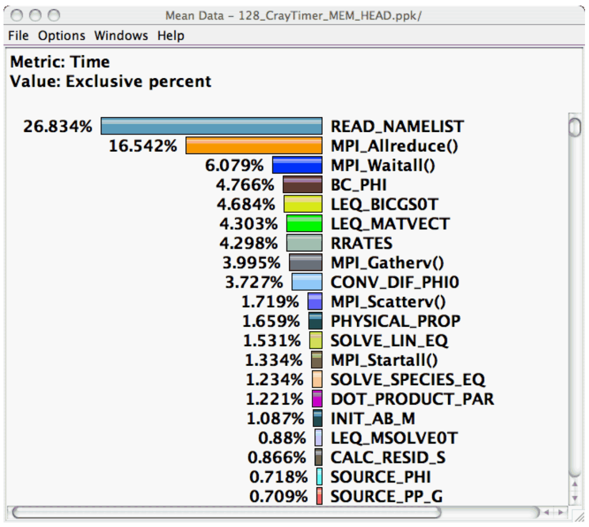


Figure 2.4.: TAU ParaProf showing a list of the most time-consuming routines

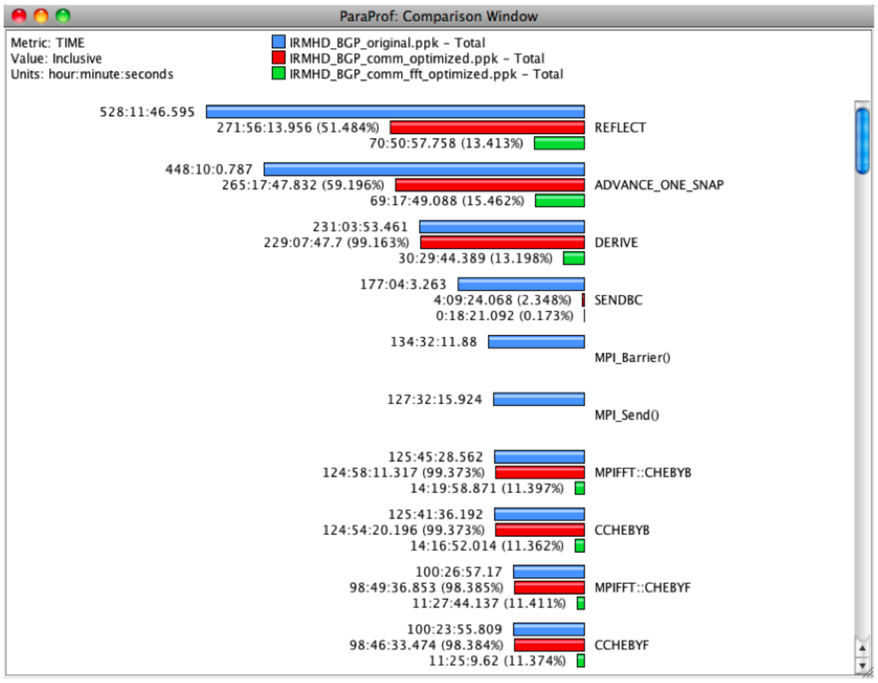


Figure 2.5.: TAU ParaProf showing a comparison of the most time-consuming routines in three versions of the program with different optimizations

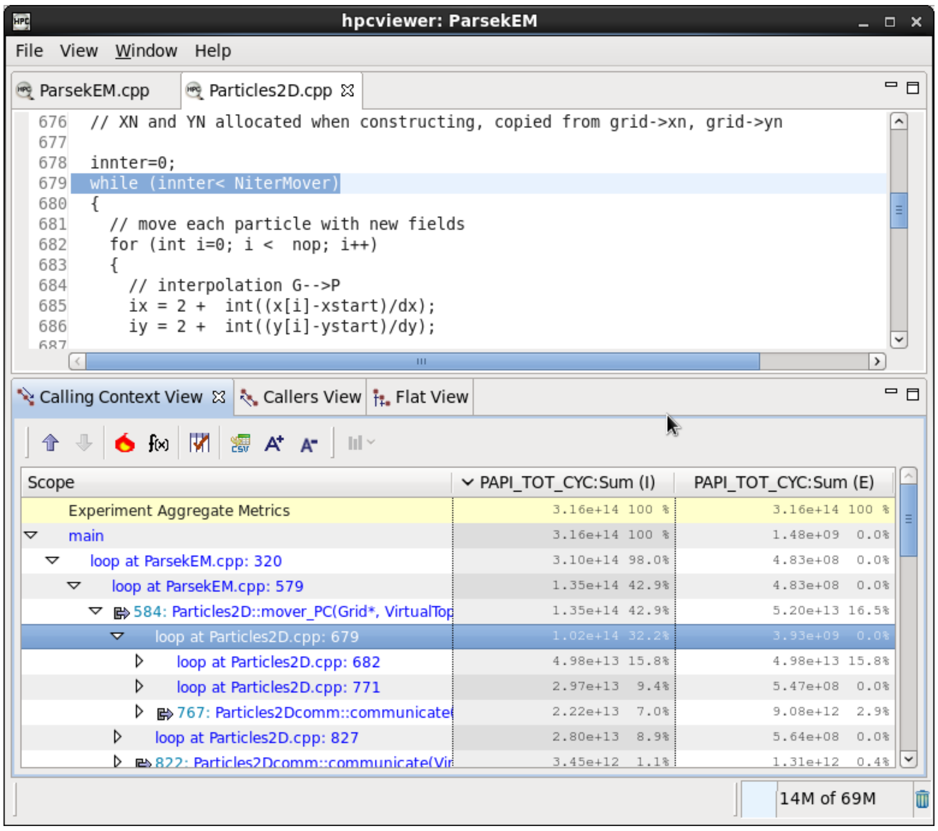
end, ParaProf relies on PerfDMF [59], a data management framework to store and retrieve profiling data from a multi-experiment database. Figure 2.5 shows a comparison of the most time consuming routines with different optimizations.

HPCToolkit [41] offers similar functionalities as the previous tools, supporting interactive visualization of call-path and flat profiles, textual summaries, and correlations of the metrics with the source code structure. Figure 2.6 shows the main interface of the profile visualization tool `hpcviewer`, displaying performance hardware counters averages associated to the calling context.

CUBE [60], which is the performance report explorer for Scalasca [46, 47], is a generic tool for displaying a performance space consisting of three dimensions: performance metric, call-path and system resource, as shown in Figure 2.7. This three-part categorization aims at answering three main questions: “which performance problem?”, “where in the program?”, and ”where in the system?”. Each dimension can be represented as a tree, where non-leaf nodes of the tree can be collapsed or expanded to achieve the desired level of granularity, and the coloring helps the user to quickly evaluate the severity of the metric being inspected.

Presentation of event traces

In contrast to profiles, application traces preserve the dynamic application behavior. In order to enable the study of the time relationships between the discrete events recorded and the resources used, trace-based tools commonly employ a representation paradigm

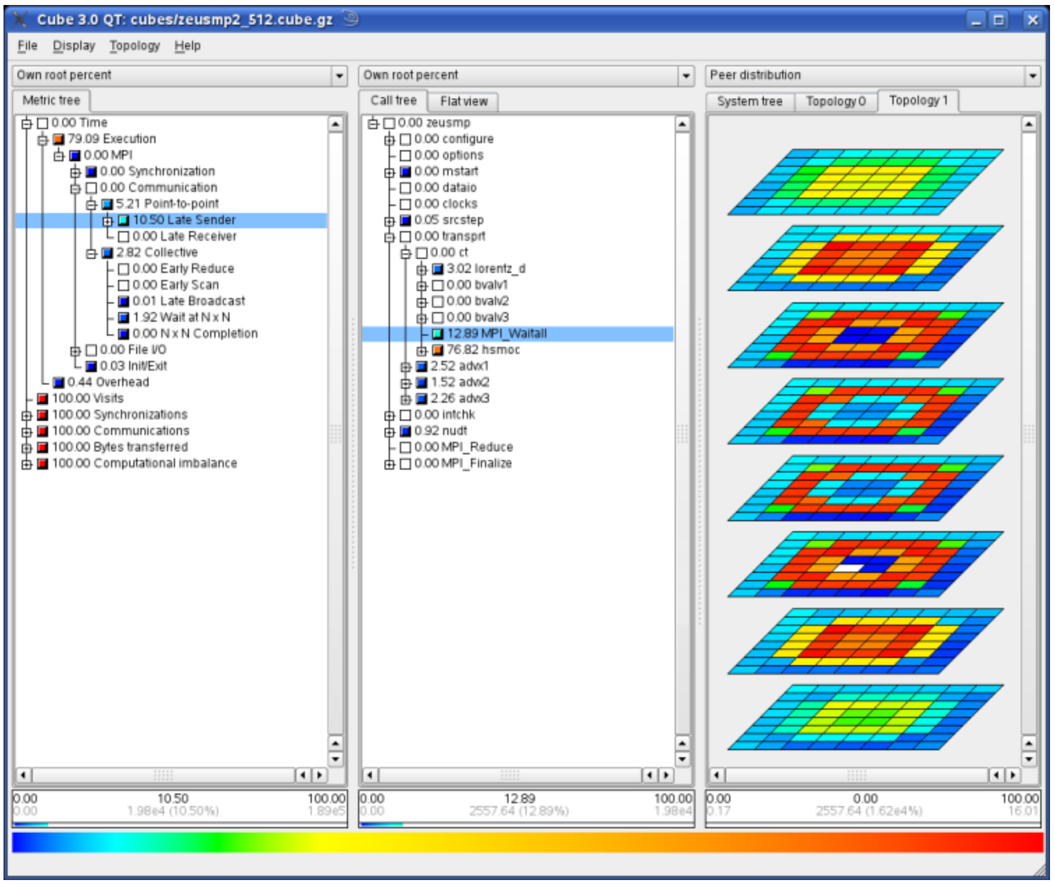


```
// XN and YN allocated when constructing, copied from grid->xn, grid->yn

innter=0;
while (innter< NiterMover)
{
  // move each particle with new fields
  for (int i=0; i <  nop; i++)
  {
    // interpolation G-->P
    ix = 2 +  int((x[i]-xstart)/dx);
    iy = 2 +  int((y[i]-ystart)/dy);
```

Scope	PAPI_TOT_CYC:Sum (I)	PAPI_TOT_CYC:Sum (E)
Experiment Aggregate Metrics	3.16e+14 100 %	3.16e+14 100 %
main	3.16e+14 100 %	1.48e+09 0.0%
loop at ParsekEM.cpp: 320	3.10e+14 98.0%	4.83e+08 0.0%
loop at ParsekEM.cpp: 579	1.35e+14 42.9%	4.83e+08 0.0%
584: Particles2D::mover_PC(Grid*, VirtualTop	1.35e+14 42.9%	5.20e+13 16.5%
loop at Particles2D.cpp: 679	1.02e+14 32.2%	3.93e+09 0.0%
loop at Particles2D.cpp: 682	4.98e+13 15.8%	4.98e+13 15.8%
loop at Particles2D.cpp: 771	2.97e+13 9.4%	5.47e+08 0.0%
767: Particles2Dcomm::communicate(	2.22e+13 7.0%	9.08e+12 2.9%
loop at Particles2D.cpp: 827	2.80e+13 8.9%	5.64e+08 0.0%
822: Particles2Dcomm::communicate(Vir	3.45e+12 1.1%	1.31e+12 0.4%

Figure 2.6.: HPCToolkit hpcviewer profile interface showing call-graph context with related hardware counter metrics



```
0.00 Time
  79.09 Execution
    0.00 MPI
      0.00 Synchronization
      0.00 Communication
        5.21 Point-to-point
          10.50 Late Sender
          0.00 Late Receiver
        2.82 Collective
          0.00 Early Reduce
          0.00 Early Scan
          0.01 Late Broadcast
          1.92 Wait at N x N
          0.00 N x N Completion
      0.00 File I/O
      0.03 Init/Exit
  0.44 Overhead
100.00 Visits
100.00 Synchronizations
100.00 Communications
100.00 Bytes transferred
100.00 Computational imbalance
```

```
0.00 zeusmp
  0.00 configure
  0.00 options
  0.00 mstart
  0.00 dataio
  0.00 clocks
  0.05 srcstep
  0.00 transprt
    0.00 ct
      3.02 lorentz_d
      0.00 bvalv1
      0.00 bvalv2
      0.00 bvalv3
      12.89 MPI_Waitall
      76.82 hsmoc
    2.52 advx1
    1.52 advx2
    2.26 advx3
  0.00 intchk
  0.92 nudt
  0.00 MPI_Reduce
  0.00 MPI_Finalize
```

```
0.00        10.50                100.00
0.00        1.98e4 (10.50%)      1.89e5

0.00        12.89                100.00
0.00        2557.64 (12.89%)     1.98e4

0.00        0.00                 100.00
0.17        2557.64 (1.62e4%)    16.01
```

Figure 2.7.: CUBE showing known performance problems (left) associated to the code (middle) and the system (right)

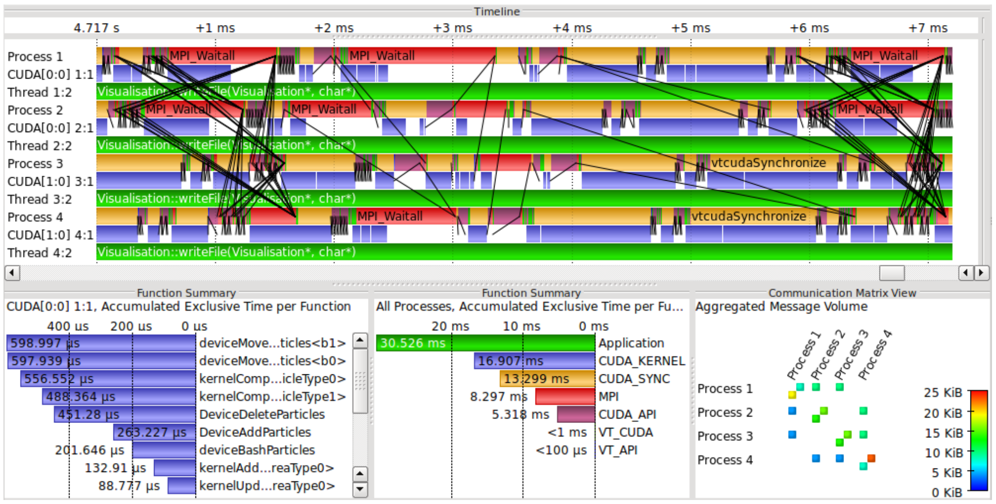


Figure 2.8.: Vampir Trace Analyzer showing a time-line view (top) and summary statistics (bottom)

consisting of a timeline. Timelines display a list of events in chronological order, typically showing a long bar labelled with dates, and events are depicted along the time bar where they happened. In parallel performance analysis, the timeline is usually represented as a 2-D matrix, where the secondary axis shows all the parallel resources involved in the execution. In this way, each event can be related not only to when it happened, but also where. This type of representation helps to visualize time lapses between events, durations, and the simultaneity or overlap of spans and events between parallel processes.

Figure 2.8 shows an example of the Vampir [61] master timeline. The X-axis represents time in milliseconds, the Y-axis represents processes, threads and accelerator resources of the parallel application, and the colors represent the routines that where executed at a given point in time for each thread. The black lines represent point-to-point communications (i.e. MPI messages) between the different processes. The bottom half displays three panes containing profiling statistics. Obvious as it may seem, profile-like statistics can also be computed from the trace data, and most trace-based tools also provide this type of summarized statistics in one way or another.

HPCToolkit's `hpctraceviewer` employs a very similar representation, as shown in Figure 2.9. `hpctraceviewer` comprises three parts: The *trace view* (top) is similar to the conventional process/time timeline, showing time on the horizontal axis and processes (or threads) on the vertical axis, with one key difference. To show call path hierarchy, the view is actually a user-controllable slice of the process/time/call-path space. Given a call path depth, the view shows the color of the currently active procedure at a given time and process rank. For the selected process, the *depth view* shows a call-path along the vertical axis for each virtual time along the horizontal axis. And the *summary view* shows for the whole time range displayed, the proportion of each subroutine in a certain time.

The Paraver trace timeline [53] is shown in Figure 2.10. Similarly to the previous tools, this display represents the activity of the application over time on the X-axis, and across processes over the Y-axis. The top timeline (2.10a) shows the program calls to the MPI run-

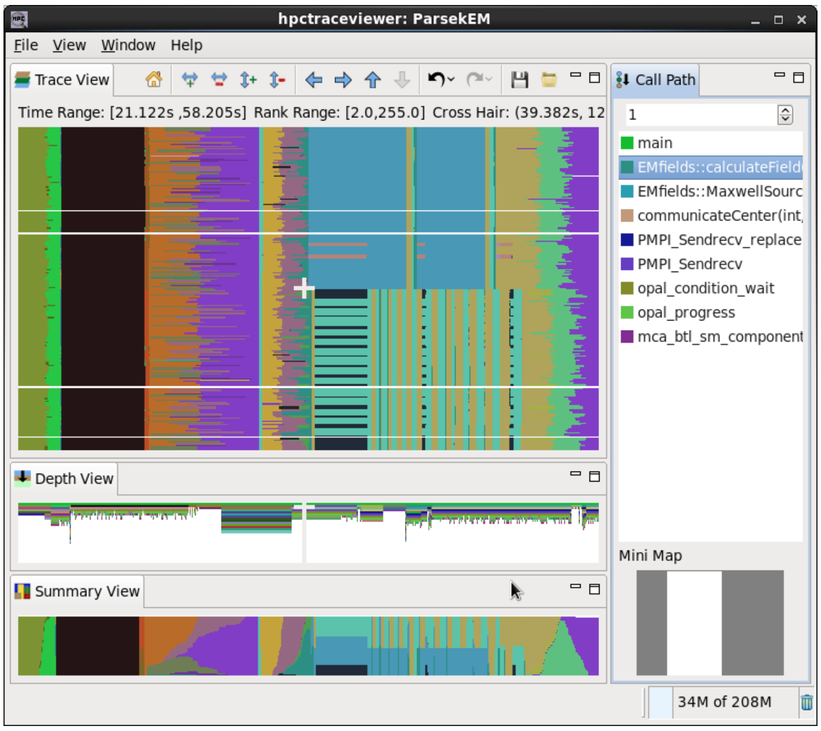
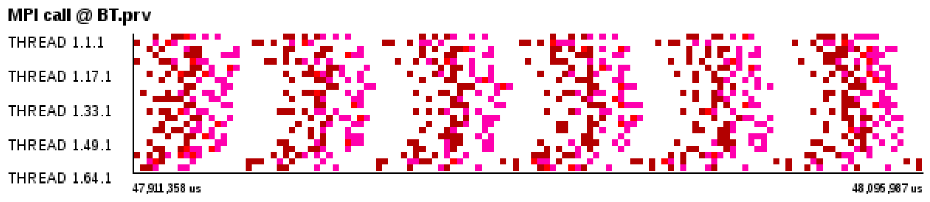


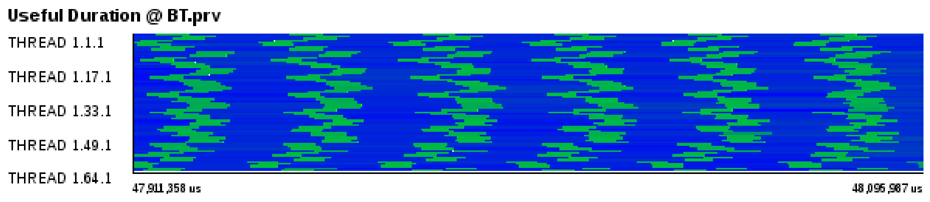
Figure 2.9.: HPCToolkit hpctraceviewer showing a detailed time-line view (top), call-graph over time (middle), and summary time-line (bottom)

time, where each different routine is identified with its own color. The main characteristic of Paraver is that this tool is semantic free. Instead of providing pre-computed metrics, it can be configured to compute any metric with the data available in the trace. This approach enables to evaluate the program's performance from many different perspectives. For example, the middle timeline (2.10b) shows the duration of the computations for the same execution phase. The colors here are a gradient that goes from green (short computations) to blue (long computations), giving a complementary view to the parallel run-time activity. Moreover, the metrics displayed can be complex derivatives of simpler metrics. The bottom timeline (2.10c) shows the achieved Instructions per Cycle (IPC), computed from the Instructions Executed and Elapsed Cycles hardware counters stored in the trace. Derived metrics can be arbitrarily combined to produce even more specific metrics on demand.

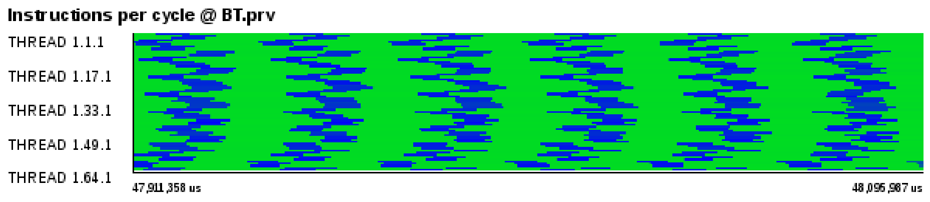
Just by visual comparison of the different timelines it is possible to identify correlations between different metrics. In this case, one can easily see that when the computations are short, the IPC achieved was high, and vice-versa. Paraver also offers summary views that allow to freely combine different metrics to facilitate to detect possible correlations. Figure 2.11a shows a histogram of the durations of the computations. The X-axis represents bins of durations, the Y-axis represents processes, and cells with values indicate that there are computations in the program of that given duration. The color in this case represents the average IPC for those computations. This type of representation allows to find, for example, if there is a direct correlation of higher IPC's with faster computing times, which is not the case in this example, because there are long computations (right part of the his-



(a) MPI calls timeline. Each color represents a different MPI routine.



(b) Computation duration timeline. The color gradient from green to blue shows increasingly longer computations.



(c) Instructions per Cycle. The color gradient from green to blue shows increasingly higher IPC.

Figure 2.10.: Paraver timeline view

togram) with higher IPC (blue) than shorter computations (left part of the histogram) with lower IPC (green). The histogram view can represent any other metric as well. For example, Figure 2.11b represents processes in both axes, and the colors indicate the number of MPI messages exchanged between every pair of processes. Also, Paraver is able to compute flat-profiles from the trace data for a wide range of statistics (e.g. time, occurrences, average value, standard deviation, etc.), as seen in Figure 2.12, that shows time statistics for the MPI calls.

2.2.5. Which performance tool is better?

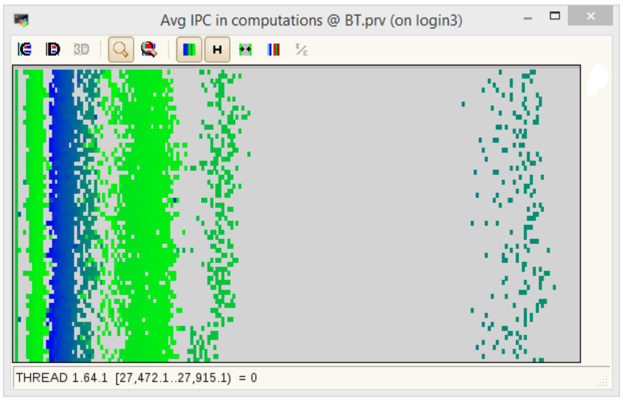
The question of which performance tool is better is often asked, but can not be answered clearly and some consider it argueably irrelevant. Most of the tools mentioned in the previous section admit several modes of operation, use similar representations, share the type of data they gather, and overlap many functionalities, but also complement each other with very specific features and slightly different standpoints. There is no best tool, but there is the most appropriate tool depending on the objective and the particular situation.

Generally speaking, profilers are a useful approach to perform first-line analyses to obtain general observations and information regarding the major hotspots in the program. Tracing tools provide much more information enabling to delve into the details and perform more in-depth analysis. As we have seen, both approaches have supporting and opposing arguments, which we could summarize in the following aspects: Profilers are generally easier to use, but the summarizations may hide relevant performance issues. On the other hand, tracing tools capture all the details, but the amount of data easily grows so large, that it becomes difficult to process and analyze it.

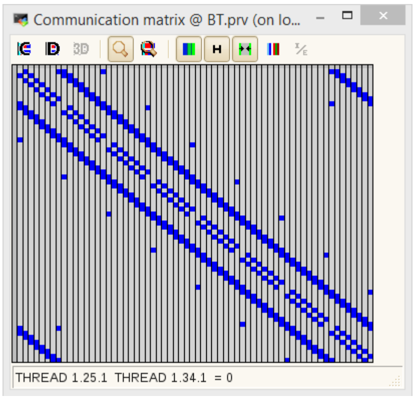
The large number of cores available to applications in future extreme-scale systems will pose a challenge to all performance analysis tools, but the size of today's large-scale systems already constitutes a problem, which is particularly pressing for tracing tools. In the November 2014 list of the Top500 supercomputers [4], even the smallest system already comprises several thousands of cores; and the average core-count is over 45,000. So tools must be able to handle at least tens of thousands of cores, but traditional tracing techniques are no longer applicable at such scales, because the amount of data generated becomes absolutely unmanageable.

It is important to be able to use tracing tools in the largest-scale because they will enable to see in detail the new types of problems that arise due to the interaction of so many components, understand how they interrelate, identify the causes and their possible solutions, model the patterns of inefficient behavior into expert systems for automatic analyses, and aid in the design of new programming models and future architectures to overcome these deficiencies.

In the following sections, we make a summary of the techniques that have been explored to deal with the traces' scalability problem and contextualize our contribution to the field.

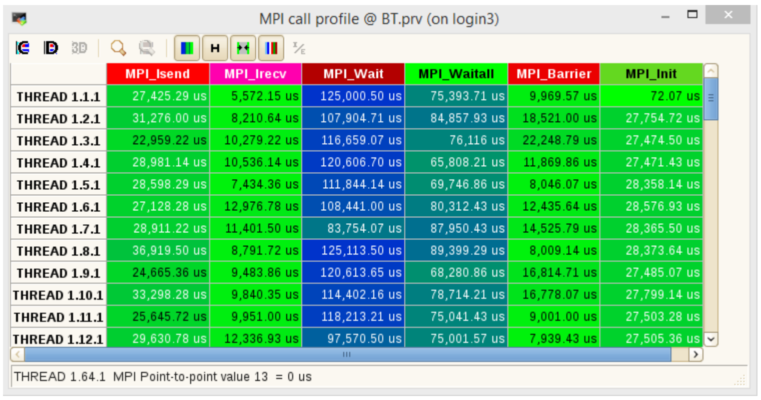

Avg IPC in computations @ BT.prv (on login3)
THREAD 1.64.1 [27,472.1..27,915.1) = 0

(a) Average IPC per computation. Data columns from left to right show increasingly longer computations. The color gradient from green to blue shows increasingly higher IPC.


Communication matrix @ BT.prv (on lo...
THREAD 1.25.1 THREAD 1.34.1 = 0

(b) Communication matrix showing the number of messages sent between every pair of peers

Figure 2.11.: Paraver histogram view



	MPI_Isend	MPI_Irecv	MPI_Wait	MPI_Waitall	MPI_Barrier	MPI_Init
THREAD 1.1.1	27,425.29 us	5,572.15 us	125,000.50 us	75,393.71 us	9,969.57 us	72.07 us
THREAD 1.2.1	31,276.00 us	8,210.64 us	107,904.71 us	84,857.93 us	18,521.00 us	27,754.72 us
THREAD 1.3.1	22,959.22 us	10,279.22 us	116,659.07 us	76,116 us	22,248.79 us	27,474.50 us
THREAD 1.4.1	28,981.14 us	10,536.14 us	120,606.70 us	65,808.21 us	11,869.86 us	27,471.43 us
THREAD 1.5.1	28,598.29 us	7,434.36 us	111,844.14 us	69,746.86 us	8,046.07 us	28,358.14 us
THREAD 1.6.1	27,128.28 us	12,976.78 us	108,441.00 us	80,312.43 us	12,435.64 us	28,576.93 us
THREAD 1.7.1	28,911.22 us	11,401.50 us	83,754.07 us	87,950.43 us	14,525.79 us	28,365.50 us
THREAD 1.8.1	36,919.50 us	8,791.72 us	125,113.50 us	89,399.29 us	8,009.14 us	28,373.64 us
THREAD 1.9.1	24,665.36 us	9,483.86 us	120,613.65 us	68,280.86 us	16,814.71 us	27,485.07 us
THREAD 1.10.1	33,298.28 us	9,840.35 us	114,402.16 us	78,714.21 us	16,778.07 us	27,799.14 us
THREAD 1.11.1	25,645.72 us	9,951.00 us	118,213.21 us	75,041.43 us	9,001.00 us	27,503.28 us
THREAD 1.12.1	29,630.78 us	12,336.93 us	97,570.50 us	75,001.57 us	7,939.43 us	27,505.36 us

THREAD 1.64.1 MPI Point-to-point value 13 = 0 us

Figure 2.12.: Paraver profile view showing time statistics for the MPI calls

2.3. The trace scalability problem

Trace-based performance tools face a fundamental problem when long execution runs and thousands of processes are involved. First, just saving and handling the trace may be unfeasible due to storage limitations. Second, the vast amount of data that has to be analyzed dramatically plummets down the responsiveness of the analysis tools. The amount of time required to perform the analysis easily becomes daunting, having a negative impact on the analyst's interest and the interaction with the tool.

Many different trace-based performance tools exist, and most of them have in common a good awareness of the traces scalability problem. Motivated by this concern, several projects have got off the ground. Although their common goal is to tackle this problem, different directions are being explored:

- Selective acquisition of performance data; techniques to filter the traced data, picking out specific performance information.
- Trace structures and compression mechanisms; new trace definitions and efficient compression schemes are proposed to reduce the size of traces.
- Distributed parallel architectures; parallelizing the tools themselves is an alternative to speed-up their execution and let them use more resources than available in a single node.
- Expert systems; methods to automatically analyze and draw conclusions from large amounts of data.
- Cluster analysis; application of techniques to simplify the description of complex multi-dimensional data by identifying a few number of invididual elements that represent the whole set of data.
- Pattern recognition; application of techniques to detect redundant data that does not provide additional information and thus can be discarded.

These are the most important topics being explored in the context of how to deal with the trace-based performance tools scalability problem. Existing performance tools draw multiple techniques from these fields and apply them at different phases of the performance analysis process. Furthermore, interaction between tools is very usual. In this way, most performance tools fall under more than one of these categories. However, we are going to discuss these areas in more detail and point up the most representative research works on each of these directions.

2.3.1. Selective acquisition of performance data

The Paradyn Parallel Performance Tools project [62] opened the direction of intelligent selection of performance data leveraging a technique called dynamic instrumentation [63].

This technique permits to attach to a running program, create a bit of code and insert it into the program. The program is able to continue executing and doesn't need to be re-compiled, re-linked or even re-started. The next time the program executes the block of code that has been modified, the new code is executed in addition to the original one. Dynamic binary instrumentation technology is available to researchers via the Dyninst API [64]. This approach extends the capabilities of other post-compiler instrumentation tools such as ATOM [65] or SiGMA [36, 66], that only permit code to be inserted into a binary before it starts executing.

Run-time code changes allow instrumentation to be inserted and removed whenever desired. This feature lays the foundations for intelligent selection of performance data. The idea consists in performing the analysis as the program runs, and deciding whether the performance data that is being collected is interesting or not. Depending on the performance values observed, the instrumentation can be changed dynamically in order to collect more relevant performance metrics.

Paradyn [62] is a performance tool which relies on run-time performance analysis, rather than recording a complete trace of the whole execution. The user can select which performance metrics they wish to view. Additionally, it also provides a bottleneck search algorithm named the Performance Consultant (PC), that automatically searches for a set of known performance problems. This algorithm uses the W^3 model to guide the bottleneck search, which attempts to answer why, where and when the application is performing poorly. This is an iterative process of formulating hypotheses and refining them based on the performance data being collected. For example, if a routine performing synchronization operations is taking up a large amount of time, the hypothesis of ExcessiveSyncTime will be formulated, and instrumentation will be dynamically inserted into that routine in order to collect more specific data that helps to find the specific causes or to reject the initial hypothesis. The PC periodically gathers performance data from every process and decisions about how does the bottleneck search progress are taken centrally.

OPAL [67] uses hypothesis, refinements and proof rules similar to Paradyn, except that refinements cause new executions of a program to be started with new data to be collected via selective tracing.

DynTG [68] shares the same idea of on-line analysis, except that it does not include automatic selection of the performance data. It includes a source browser where users can instrument their program as it runs by clicking on source code lines in the browser. The performance data gathered is presented in real-time. This enables users to monitor their application progress and interactively adapt the instrumentation based on their own observations.

TAU [45] comprises a set of static and dynamic tools that form an integrated analysis environment for parallel applications. It includes the tau_reduce tool, which allows to create a list of user-defined rules for excluding events that result in a lot of run-time overhead and do not offer much information (i.e., functions that do a small amount of work and are called many times).

A more specific work presented in [69] illustrates the use of partial data traces for successfully detecting memory performance bottlenecks. Continuing this work, the authors propose a method based on Dyninst to extract partial data traces from running applications by observing their memory behavior [70]. The instrumentation is placed at memory accesses to capture the data references issued by the program. An on-line algorithm is used to select the most critical memory accesses. Once a specified number of those events have been logged or a time threshold has been reached, the instrumentation is removed.

The BSC Tools [53] comprise a set of post-mortem utilities to post-process Paraver traces. These mechanisms allow to cut, summarize, translate or accumulate the performance data in order to produce smaller traces with a partial view of the overall application behavior.

2.3.2. Trace structures and compression mechanisms

Several proposals address the traces scalability problem from the data structure point of view. Their objective consists in reducing the traces size, and at the same time speed-up the access to the data either to display it or to compute statistics. To this end, structures to store the trace data both in disk and memory, as well as adequate algorithms to traverse through the data have been designed.

Continuing the work on data compression of trace files [71], the Compressed Complete Call Graphs (cCCGs) [72] is a recent compressible memory data structure for event traces. Its primary intention is to significantly reduce the memory requirements in the analysis of huge traces. In the traditional scheme, events are stored sequentially in the trace sorted by time-stamp. Although this is a simple and fairly effective data structure, it offers no way to exploit the inherent redundancy of trace data. A cCCG of a program consists of a call tree, that represents the run-time call stack, for every process. All properties of a function call (i.e., the call duration, performance counter values, etc.) are annotated in the node representing that particular invocation in the tree. The data compression is achieved by collapsing nodes of the tree with the same (or very similar) properties. The decision of collapsing two nodes with not exactly the same but similar properties depends on whether lossy compression is allowed or not. This development has been integrated into Vampir-NG [61, 73].

Open Trace Format (OTF) [74] appeared as a new trace definition for use with large-scale parallel platforms. The main objective addressed is to provide efficient sequential and parallel access to trace data. To this end, it provides an API that enables to read and write trace files in parallel. This proposal was incorporated into TAU [45], which can convert their own trace format into OTF with the tau2otf tool; into Vampir-NG [61, 73], which can load OTF traces; and into the BSC Tools, which included a translator to convert OTF traces into Paraver format [75].

Recently, as a part of the Score-P initiative [50, 51], the analysis toolkits Scalasca, Vampir, Periscope and TAU decided to adopt the Open Trace Format version 2 (OTF2) [52]. 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 [76], and the access API uses techniques to reduce the memory footprint.

Scalatrace [77] provides online trace compression of MPI communication trace-files at two different levels: intra-node and inter-node. Intra-node compression is achieved by describing loops with regular section analyses (RSA) [78] and compressing call-path information. On the other hand, Scalatrace compresses at the inter-node level by combining the information from the involved processes into a single one for the whole application.

Jumpshot [79] is a Java-based visualization tool that relies on time-line representations for post-mortem analysis. Jumpshot's SLOG trace format [80] is a graphics-oriented data representation. It is more computationally expensive to create than traditional timestamped list of events but allows Jumpshot to more efficiently display trace data. It is based on describing composite graphical objects rather than individual events, and is capable of handling traces in the gigabyte range.

Intel Trace Collector and Trace Analyzer are Intel's updated versions of Pallas' Vampir tools [49]. They define the Structured Trace Format (STF) to store trace data. Files can be written in parallel, thus generating trace files faster and includes some indexing techniques that allow random access to different portions of the same trace independently, speeding up the process of loading large amounts of data.

DeWiz [81, 82] proposes to represent an event trace as a directed event graph. The vertices represent the events observed during the program execution, like for example send or receive events in message passing programs, and read or write memory accesses on shared memory programs. The edges represent the relation between the events, that are connected using the happened-before relation. The objective of such representation is tailored towards the analysis tasks, which can then be performed as a set of graph filtering and transformation operations. Similarly, PARADIS [83] also utilizes the event graph model to represent a program run for all further analysis tasks.

2.3.3. Distributed parallel architectures

Parallelizing the tools themselves is an alternative to speed-up their execution and allow them to access more resources than available in a single node.

Vampir Next Generation (Vampir-NG) [61, 73] explores the direction of distributed data processing. They propose a distributed software architecture for parallel program analysis. This design consists of a parallel analysis server running on a segment of a parallel production environment, and a visualization client running on a remote graphics workstation. Both components interact through a socket based network connection. This architecture allows the analysis server to be close to the location of the trace data. The server uses a master/workers approach. Workers are responsible for storage and analysis of a part of the overall trace data. The master decides how to distribute analysis requests among the workers and merges the results from multiple workers into a single response that is sent to the visualization client.

Paradyn's automatic performance bottleneck search has been recently extended [84]. The new strategy uses distributed, autonomous agents that monitor each running process, searching for a predefined set of performance problems, and taking local decisions about how does the search progress. A very similar architecture for on-line performance analysis is proposed with Periscope [43], an automatic analysis tool that consists of several analysis agents that search autonomously for inefficiencies in a subset of the application processes.

A parallel extension of KOJAK [85, 86] is the Scalasca project [46]. Rather than sequentially analyze a single global trace file, Scalasca analyzes separate local trace files in parallel.

DeWiz's [81, 82] architecture also has a modular design which enables distributed data processing. The different modules that perform analysis tasks are placed on arbitrary computing nodes. A dedicated module, the DeWiz Sentinel is used to coordinate interconnection between them.

In order to develop distributed architectures using the master/worker scheme like the tools mentioned above, it is important to efficiently manage the flow of data between both components. The Multicast/Reduction Network (MRNet) [87] is a software tailored towards building scalable parallel performance tools. It uses a tree of processes to communicate between the master and workers. These internal processes are used to synchronize and aggregate data flowing across the network. Using filters, these processes can compute averages, sums, and other more complex reductions on tool data. Tools like TAU [88], Paradyn [84] and STAT [89] have already incorporated this kind of technology into their software.

2.3.4. Expert systems

While the approaches above try to control the volume of data to keep it at a reasonable size, other works focus their efforts on providing efficient mechanisms to analyze huge traces. Manually sifting through large amounts of trace data and trying to identify specific performance problems can be daunting. Instead, automatic analysis mechanisms are proposed.

The automatic analysis is often performed post-mortem. This approach has the advantage of being able to consider all detailed information gathered during the application execution and the analysis phase does not introduce any overhead in the run.

The majority of automatic analysis methods adopt a knowledge-based system solution to the problem of identifying performance problems in existing programs. These tools contain a knowledge base of known performance problems which are matched against complete trace files. Once bottlenecks are detected, some tools also suggest a few strategies on how to fix the performance problem based on information contained in the knowledge base.

When applying knowledge-based expert systems, it is often very useful to let the user add their own entries into the knowledge base of performance problems. While some tools use fixed sets of bottlenecks, more often than not the core set of rules can be extended by the user.

KOJAK's [85, 86] main feature is EXPERT [90, 91], an automatic trace analyzer that attempts to detect specific bottlenecks such as inefficient use of the programming model and low CPU and memory performance. The set of rules that model inefficient behavior of the program can be defined by the user using the EARL language [92]. Every metric is computed for every thread or task and every part of the code. The results are presented on the CUBE viewer [60], which displays the performance problem - task - code part resulting combinations, making easy to correlate where do performance problems actually happen.

Kappa-PI [93] is a knowledge-based system similar to EXPERT in terms of applying rules that model inefficient behavior to detect performance bottlenecks. The main difference is that Kappa-PI checks the source code and makes suggestions to the user on how to fix each bottleneck and improve their application. One of the main disadvantages of this tool is that the performance knowledge is directly coded in the kernel of the tool. Keeping in mind the same goal of providing useful hints to the application developer a new version of the tool, Kappa-PI 2 [94], was developed. It is based in the specification of performance knowledge as input data for the tool using the APART Specification Language (ASL) [95], which is a standard grammar for specifying performance problems.

ASKALON [96] is a set of tools for cluster and grid computing. Among other tools, it comprises SCALEA [97] and AKSUM [98]. SCALEA provides instrumentation mechanisms that allow to select code regions (i.e. loops or procedures) for which specific performance metrics should be obtained. AKSUM is an automatic analysis mechanism that tries to discover bottlenecks using a base of known performance problems. The set of rules can be specified with JavaPSL, a Java implementation based on the standard ASL mentioned above. Beyond single-experiment analysis, it also supports multiple experiment performance analysis, that allows to compare the performance outcome of several experiments.

2.3.5. Performance modeling

A different approach within the framework of automatic analysis is based on data modeling techniques. Performance modeling is often used to make comprehensible interpretations of the tracing data, identify the relevant parts of the program, pinpoint their bottlenecks, and generate accurate performance predictions; while avoiding the data storage overheads.

A significant research effort has been invested in the development of approaches for performance modeling and prediction of parallel and distributed systems [99]. To this extent, Hoefler et al. [100] popularized performance modeling by defining a manual six-step process to create interpretable application performance models that could be presented to the user so that they can estimate how the program's performance changes with the change in values of the input's parameters. Calotoiu et al. [101] extend this work, developing an empirical scalability model to predict the scaling behavior of applications beyond the executed configurations.

PEMOGEN [102] is a modeling framework that complements runtime profiling mechanisms with online performance modeling, a method that generates performance models while the application is running. This allows to produce accurate predictions based on trace data, while greatly reducing the storage overhead. PEMOGEN relies on LASSO, a statistical learning method to automatically learn performance models during execution for the most relevant kernels (i.e. loops and functions), considering their critical parameters (i.e. sizes of dimensions or number of iterations), and generates models for multiple target metrics (i.e. hardware performance counters, the number of messages communicated, the size of the messages and the execution time). Overall, PEMOGEN assesses the scaling and potential bottlenecks with regards to any input parameter and the number of processes of a parallel application.

Morajko et al. [103] also explore the direction of on-the-fly trace analysis through performance modeling. They aim at reflecting the application behavior by modeling execution flows through high-level program structures, such as loops and communication operations, and to characterize them with visual statistical execution profiles. They define the Task Activity Graph (TAG) as a directed graph that abstracts the communication and computation activities of a single process in message-passing applications. The model provides a high-level view of the execution and enables the easy detection of performance bottlenecks and their location in each task. By merging individual TAGs, they provide a global application view, which provides the opportunity to analyze the whole application while it runs and develop tools for root-cause problem diagnosis. In this same direction, MATE [104] relies on online performance modeling techniques to perform dynamic and automatic tuning of MPI parallel applications, by conducting message aggregation, workload balancing and PVM library tuning according to the conditions of the system.

COMPASS [105] is a framework for automated performance model generation and prediction. COMPASS generates a structured, parametrized performance model in the Aspen performance modeling language [106] by using automated static analysis of the target application. This performance model can then be used for a variety of purposes, including predicting performance of the target application on a range of current and future architectures, as well as predicting runtimes under different application and system configurations.

A recent work that is being deployed by Rosas et. al [107] suggests the decomposition of the application's performance in fundamental factors such as load balance, transfer and serialization to describe the application behavior. This scalability empirical model infers the expected performance based on the known limitations of the code on a given architecture, using an Amdalh's law-based fitting. Their approach is applied post-mortem to easily extract high-level metrics from a set of heavy execution traces and thus further automate the performance understanding process.

2.3.6. Cluster analysis

A different trend on automatic analysis is the application of statistical methods to perform clustering in event traces. Cluster analysis is the task of grouping a set of objects so that objects in the same group (cluster) are more similar to each other than to those in other groups, with respect to one or several characteristics. This technique has long been used for exploratory data mining, statistical data analysis, unsupervised machine learning or image analysis. The knowledge of the clusters enables a very concise description of a large set of complex multi-dimensional data, just by replacing the description of each individual element of the cluster by a single representative of the group.

Several works have applied clustering techniques to find representative processes in a parallel application. Nickolayev et al. [108], propose the application of the K-means clustering algorithm in an on-line analysis to determine the similarities among processors involved in a parallel execution. The authors use coarse-grain granularity metrics such as processor idle or running times to describe the behavior of each individual processor. The work was developed as part of the Pablo Performance Environment [109], 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.

Ahn et al. [110] developed a statistical analysis of event traces containing processor performance hardware counters that characterize application subroutines. In this work, hierarchical and K-means clustering are used to determine the similarity across 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 on the problem decomposition. In addition, the authors apply multivariate statistical methods, Principal Component Analysis (PCA) [111] and Factor Analysis [112], 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.

PerfExplorer [113] offers similar features 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 an experiment database, that includes a wide variety of metrics, ranging from high level idle or running times to low level processor hardware counters. Similarly, the target of the clustering analysis is to find processes (or threads) with similar behavior. The major contribution of this work is the correlation of the groups found with the profiling information available, providing the analyst a clear characterization of the behavior of the clusters.

González et al. [114] explore a different direction. Instead of looking for similar processes or threads, they detect which computing phases of the program present similar performance behavior. This approach does not aim at reducing the size of the data by selecting representatives, but to expose the fine-grain structure of the main performance trends of the program. They propose to apply the density-based cluster algorithm DBSCAN [115] to the program's computations, which are stored in an event trace characterized by their duration and several performance hardware counter metrics such as the Instructions Executed and the Instructions per Cycle. As a result of the analysis, all the computations in the trace are enriched with the information of to which cluster they belong to. This information enables to represent in a timeline the distribution of clusters over time, expose SPMD or sub-SPMD program structures, identify multi-modal behavior, and study in detail the time-varying performance variabilites that may happen across processes and over time.

2.3.7. Pattern recognition

Computation-intensive scientific applications usually present iterative algorithms being repeated over and over. In many cases, the program behavior does not vary between iterations. It stands to reason that significant performance problems that may exist will be noticeable repeatedly. Therefore, saving detailed performance data for every iteration does not necessarily provide much more useful information than tracing just a few iterations. Redundant performance data can be safely discarded, sharply decreasing the trace data volume while precision of the analysis is not lost.

In this direction, the Dynamic Periodicity Detector (DPD) [116] detects the periodic structure of an OpenMP application as it runs, and emits to the trace file only a few periods of such pattern. The DPD uses the stream of OpenMP parallel function identifiers that are being executed to identify whether periodicity exists in the current data stream. Once enough periods of the current pattern have been traced, the tracing facility stops recording events, but keeps on feeding the DPD with the stream of data until a change in the program behavior is detected, and trace data starts to be collected again.

An analogous approach [117] detects iterative communication patterns in message passing (MPI) applications. A combination of parameters of the MPI calls (e.g. memory addresses of the accessed buffers) is used to detect periodicity. This work aimed at dynamically balancing the assignment of resources (processors) to processes.

A post-mortem approach described in [118] derives signals from the data contained in the trace file, calculating metrics that summarize the behavior of the application from a particular point of view. Then, signal processing techniques are applied in order to discover execution phases in the temporal evolution of the application. This system allows to generate smaller sub-traces which contain only a few repetitions of the periodic patterns detected. Additionally, it reports a global view of the structure of the program in terms of which were the repetitive execution phases.

A. Knüpfer et al. [119], tackle this problem at the visualization phase. Their position is that, although performance tools can successfully cope with ever growing trace sizes, human perception is unable to scale-up with the increasing amount of data. Their objective is to display less data but more information within the typical time-line views. To this end, they collapse all repetition patterns of function calls with a single marked box. These boxes indicate a region with repetitive behavior, and inner details are not shown. Since this display hides too much information, patterns can be decomposed interactively, replacing every pattern with its direct sub-patterns. This process can be repeated until there are no more boxes and the fully decomposed view is identical to the original one.

2.4. Thesis work in context

The algorithms, techniques and methodologies developed in this thesis have been implemented into the BSC Tools [53] ecosystem. In this section, we describe the starting point of these tools and put the work developed in context with respect to the state-of-the-art presented before.

2.4.1. Preliminary work

The two main lines of previous work in the BSC Tools towards improving the scalability of tracing tools were to provide user-driven mechanisms to select the data being collected and to develop a lightweight tracing process for summarizing the data of large runs [120].

The mechanisms for data selection included rudimentary techniques to control the range of instrumented processes, stop the tracing after a certain amount of data had been collected, and pause or resume the tracing at different phases of the program. Since they determine which information is collected, they have direct impact on reducing the size of the final trace. These mechanisms were useful, but they had to be explicitly activated by hand before starting the application, and thus required prior knowledge of the program to decide what to select.

The lightweight tracing approach considered to collect information only about phases of computation that were longer than a given threshold, focusing the analysis on the parts of the program that represented more execution time. For these regions, the tool provided summaries of performance hardware counters and several first-order statistics. The experience acquired from this work was that it is possible to obtain interesting information without requiring to generate the whole detailed event trace.

2.4.2. Proposed work

This thesis continues this previous work along two main directions. First, instead of relying on user-driven mechanisms that have to be manually activated to reduce the size of the traces, we have proposed to apply intelligence to the tracing process to automatically focus on what is relevant for understanding the application behavior. To this end, we devised a new tracing infrastructure that is capable of examining the application behavior on-line, and applying performance analytics techniques to select the most relevant performance information to be stored in the trace, drastically reducing the volume of data being collected.

Our proposal shares common aspects with Paradyn [84] and Periscope [43]. Similarly, we propose an on-line analysis system that inspects the performance data being collected. However, two fundamental issues differentiate both proposals. First, their objective is to point up common performance bottlenecks in the program. Instead, we want to obtain a complete depiction of the program run that helps to fully understand its behavior. Second, they rely on summarized profiling snapshots to compute the analysis. On the contrary, we base the decisions about which information has to be stored on analyzing the detailed stream of events being captured, which supports the study of variance in time and space.

Automatic trace analysis systems such as EXPERT [90, 91], AKSUM [98] or Kappa-PI [94] mainly differ with Paradyn [84] and Periscope [43] in the phase the analysis is applied. Their post-mortem approach allows a thorough search for performance bottlenecks over the whole detailed event trace, rather than relying on summarized statistics. Nevertheless, all of them focus on identifying previously known performance problems, limiting the purpose of the analysis to a very specific objective.

In a very similar way to rule-based systems, performance modeling approaches like PEMOGEN [102] also aim at interpreting the data to gain automatic insight about the program's behavior and bottlenecks. Their approach allows to make useful predictions beyond the experiment space, yet the accuracy of the results will depend on a wise choice of the parameters to model and a good inference of their impact on the program's performance. Making the appropriate choices is particularly decisive when these solutions are applied on-line and the data used to build the model is no longer kept, because if it is later verified that the model does not fit well, it is then necessary to repeat the execution and rebuild the model, which may be expensive. In this regard, the ability of the domain expert to define the model's critical parameters, either manually or automatically, is crucial.

Arguably the major drawback of expert systems and modeling approaches is that the ability to identify performance problems is strictly limited by the rules defined in the knowledge base and the selection of parameters to model. Rather than trying to synthesize the data into higher-level conclusions automatically or into specific target metrics, our approach aims to keep a minimal recorded instance of a run, which permits the analyst to mature their understanding about the causes of the observed performance, formulate their own hypotheses and design specific metrics to corroborate them. This process can lead to discover new and unexpected patterns that model an inefficient behavior, that could then be used to refine the knowledge base and models of the automatic analysis tools.

Aside from the strategies described above, there are no other proposals in the literature sharing the idea of intelligently selecting performance data of high interest in detailed event traces to deal with the scalability problem of trace-based performance tools. This methodology raised a significant challenge that we have met: to have the ability to automatically determine the minimal subset of relevant performance information that fully describes the overall performance behavior, when not even the user knows what to expect and where to look at when facing the analysis of a new application. These achievements are described through Chapters 3 to 5, where we show how cluster analysis and pattern recognition techniques can be leveraged to this purpose.

With the volume of data generated successfully controlled, the second part of our work focuses on improving several important aspects of the task of analysis. In the first place, the application of on-line analysis techniques imposes certain new restrictions. First and foremost, for the analysis to take place at runtime it is important to deliver near real-time results, and so the efficiency of the mechanisms used to process large volumes of data can limit the scalability of the analysis.

In our work, we have taken as a starting point for the on-line analysis the clustering strategy presented in [114], due to its ability to provide very fine-grain details about the program structure. After evaluating the efficiency of this method, which is computationally expensive, it became clear the necessity to design a new distributed strategy to parallelize the cluster analysis in order not to sacrifice quality in expense of fast results in large scale experiments. Other works have also followed the approach of parallelizing their clustering algorithms [121, 122, 123, 124, 125], yet our approach offers the main advantages of eliminating the need for an expensive pre-processing phase to prepare the data, it is aimed for general purpose processors, and proposes a work distribution scheme that is generic enough to parallelize different families of clustering algorithms. The achievements regarding the increase of the scalability and the quality of fine-grain cluster analysis to be applied on-line are discussed in Chapter 6.

In the second place, an effective analysis of a parallel application often requires to understand how the behavior of the program changes over time, or over different experiments with different settings that may impact on the performance. Comparing large amounts of performance data that may come from different sources can be a difficult task. This thesis proposes a new methodology to compare and contrast the large amount of performance observations generated from multiple execution scenarios, providing useful and intuitive insights about how the program behavior varies under different circumstances.

Comparative and multi-experiment analysis has been approached by several performance analysis tools. Scalasca [126] includes a tool called performance algebra that can be used to merge, subtract, and average the data from different experiments and view the results as a single derived experiment. PerfExplorer [113] supports data mining analyses on multi-experiment parallel performance profiles. Its capabilities include general statistical analysis of performance data, dimension reduction, clustering and correlation of performance data, and multi-experiment data query and management. TAU [45] incorporates the concept of phase profiling for the study of the evolution within a single experiment. This is an approach to profiling that measures performance relative to a phase of execution, having its entry and exit marked by the user. HPCToolkit [41] merges profile data from multiple performance experiments into a database file and perform various statistical and comparative analyses.

The fundamental difference that distinguishes our approach is that we do not merely report the outcome of different experiments together. We automatically determine the regions of interest and track their evolution along multiple executions. To this end, we translate performance data from different execution scenarios into a sequence of images, detect structure in each image and automatically correlate them. These achievements are described in Chapter 7.

Lastly, an important limitation for the analysis is to gain access to all the computing resources that are necessary to perform all the experiments and generating all the data, which is often very time- and cost-consuming. Aiming at improving the productivity of the analysis and the effective use of the resources, we propose a novel strategy based on active monitoring techniques to inject controlled interferences into the program. This approach enables us to obtain from a single execution all the performance measurements that describe the behavior of the program under different conditions, an information that would require multiple executions to get otherwise, saving on time, energy and resources.

In this direction, we find tools like AutoTune [127], a plugin-driven framework that automatically runs the program one or more times to try different configurations of compiler flags, energy efficiency parameters and different execution patterns, and returns recommendations to tune the code. Similarly, Active Harmony [128] and MATE [104] are automated runtime tuning systems that put emphasis on dynamically adapting to changing resource capacities and application requirements. The technique that we have proposed shares the idea of testing different settings during the execution but with a different goal: to extract all the detailed measurements necessary to conduct an in-depth analysis of the data, instead of reporting general and predefined recommendations. Moreover, our approach enables to simulate on-line both different hardware and runtime settings, requiring only a single program run. These achievements are described in Chapter 8.

Part II.

New Techniques to Produce Smart Traces

Chapter 3

The on-line analysis framework

The very first handicap for trace-based performance tools when it comes to analyze large-scale parallel applications is to store all the performance data that is produced during the execution. And even if you have enough storage available, the amount of time required to process the sheer volume of data can be extremely high.

Indiscriminate tracing is no longer a feasible option, and so it is necessary to add intelligence into the tracing process and explore on-line techniques so that you directly obtain the most relevant information regarding the program behavior, getting rid of the tedious task of handling large files.

In this chapter we present an on-line analysis infrastructure that enables to attach to a parallel application, extract performance measurements, analyze the data at run-time and take intelligent decisions on which information is relevant for the analysis and what can be discarded.

3.1. Background and motivation

Nowadays it is not surprising to find applications running in real production systems for hours and using thousands of processes. And the scale of these applications keeps increasing everyday with the advent of Exascale computing. It is clear that a blind tracing approach in these scenarios would produce huge data volumes that are absolutely unmanageable.

Despite the drawbacks that entails using traces, it is important to use them because they provide far more information for a very detailed analysis. In the traces it is possible to see small effects that may have a large global impact in the performance of the application. In particular, they enable to study spatial and temporal variabilities, which are the usual triggers for load imbalance problems, and in turn one of the major bottlenecks in high performance computing.

The first problem with trace-based tools is to store the data that is produced. Even if

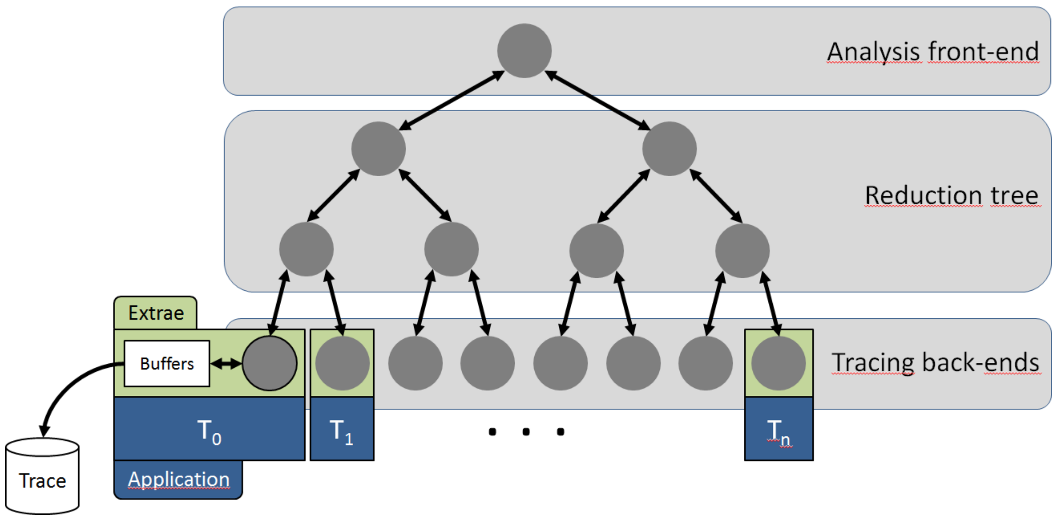


Figure 3.1.: On-line analysis framework system architecture

the amount of performance measurements gathered by a single process is small, traces rapidly become unmanageable when merging together the information collected from all processes. But not all the data that is recorded in a trace might be essential to understand the application behavior. For example, the iterative nature of most scientific applications and their common SPMD organizations results in huge amounts of redundant information. An efficient analysis of such large volumes of data requires to direct the analyst's attention towards what is more meaningful. Discarding irrelevant data aims at reducing both the size of the traces, and the time required to perform the analysis and deliver results.

To date, the usual approach to deal with large traces has typically consisted in filtering the information: one would first obtain a whole trace, then do summarizations to get a broad view of the application behavior, and finally focus on several small, representative regions to inspect all the details. And this kind of post-process has been typically applied post-morten, and manually driven by the expert. The next obvious step is to automatize this process to be done on-line, so that one directly obtains the most relevant information from the execution, getting rid of the cumbersome task of storing, processing and manipulating large trace files. To this end, we have designed an analysis framework that incorporates intelligence to the tracing process, enabling to apply diverse techniques of analysis to inspect and filter the data automatically and at run-time, towards selecting the minimum amount of information that best describes the application behavior.

3.2. System architecture

The architecture of the on-line analysis framework is structured in three main components, as depicted in Figure 3.1: the tracing back-ends, the reduction tree, and the analysis front-end.

The tracing back-ends are responsible for attaching to the parallel application and collecting performance measurements from the execution, and they provide mechanisms to filter, manipulate and store the data into a trace file.

The reduction tree provides a scalable mechanism to reduce and summarize the information that is collected from every independent process in the parallel execution. The tree emulates the structure of a multi-layer perceptron neural network, where each node applies an activation function that combines a set of inputs to produce a single output. These functions can compute complex data aggregations so as to reduce the volume of data that flows through the network to keep the system scalable.

The analysis front-end performs the role of a global orchestrator. It is a centralized process that coordinates the interaction between these components, and enables to aggregate data coming from all processes of the parallel application to conduct global analyses having an overall view of the status of the program. Based on this information, it can provide dynamic feedback to the tracing back-ends about which information is more interesting to record, and take intelligent decisions regarding:

- When to act; decide when it is necessary to activate the tracing back-ends. For example, when do they have to start collecting data, and how often the captured data has to be analyzed.

- Which actions to take; select which of the mechanisms that are available in the tracing back-ends to filter and manipulate the trace data have to be used at a given time in order to focus on the more relevant information. For example, switch the level of granularity of the recorded data, change the metrics that are being inspected, or decide the type of analysis that has to be performed.

A significant part of the thesis has then consisted in designing different protocols of analysis to automatically decide when and what to do, based on how the application is behaving. The modular construction of the on-line analysis framework enables to implement these protocols as plug-ins that can be loaded alone or in combination into the different components of the system. Depending on where are the plug-ins connected we can meet different analysis needs:

- Local analysis; protocols running on the back-ends can analyze their own data straight, as they have direct access to the trace data that is being captured from the execution. Local analyses are interesting because they diminish the volume of data that is transferred through the network, and enables the back-ends to take autonomous decisions based on the behavior of each single process.

- Distributed analysis and summarizations; protocols running on the intermediate nodes of the reduction tree can take full advantage of the resources and their tree-like organization to run hierarchical analyses and perform complex data reductions to keep the system scalable.

- Global analysis; protocols running on the analysis front-end can analyze an aggregate of the data produced from all back-ends. Global analyses are interesting because they enable to find a consensus on the state of the whole program and take coordinated actions involving all processes.

Through Chapters 4 to 5 we will discuss the protocols that we have designed acting on the different tree components to drive the exploration of the performance data and intelligently select the most relevant information to understand how the application behaves, while keeping the volume of data at a reasonable size. The following sections describe the workflow of interaction between the analysis framework components.

3.3. Components interaction

This section gives details on the implementation of each component in the system and how they interact. In summary, we have combined two pieces of software: a tracing facility and an overlay network; extending their standalone functionalities to develop an on-line system that inspects the trace data on-the-fly and takes intelligent decisions dynamically.

The basic interaction between these tools is as follows: The user application is intrumented with a tracing tool that automatically intercepts the entries and exits points of the calls to most parallel runtimes. At these points, it takes automatic measurements for different hardware components describing what is the performance behavior of the different parts of the program.

Periodically, the captured data is sent through the overlay network and reduced across the intermediate nodes of the tree upon its way to the root, so as to preserve the system's scalability. The front-end analyzes the data that comes from all parallel processes, and sends feedback to the tracing back-ends instructing them on which is the most relevant information to focus.

In order not to introduce a continous overhead on the program execution, the data is not analyzed in streaming, but periodically. When the system decides to perform the next phase of analysis, the front-end broadcasts a message to activate all back-ends, and this starts the process of data extraction and analysis. This process is repeated over time, interspersing intervals of data collection without any interference, with steps of analysis of the collected data, at the discretion of the analysis front-end.

3.3.1. The tracing back-ends

Performance data from the running application is collected through Extrae [53], a tracing tool that automatically intercepts the entries and exits points of the calls to most parallel runtimes. At these points, it relies on low-level libraries (e.g. PAPI [22]) and system services (e.g. resource usage or memory allocation information) to take performance measurements for the different hardware components, describing what is the performance behavior of the different parts of the program.

In particular, our work has focused on the analysis of message-passing applications (MPI), optionally including shared memory programming (MPI + OpenMP). We have focused on these two runtimes because they are the most widely used to exploit parallelism in HPC applications, but our solution can be likewise applied to other programming paradigms. The typical information gathered for MPI applications consists in hardware counters (i.e. elapsed cycles, instructions executed, cache misses, etc.), references to the source code (i.e. call-stack information) and punctual events (i.e. entry or exit from a function). A basic tracing tool stores all these measurements as a time-stamped sequence of events into per-thread memory buffers, and eventually writes the data to disk when the buffers are full to reduce the overhead of instrumentation.

In our on-line implementation, the memory buffers operate instead in a circular mode. This is to say, once they have been filled, every new event overwrites the oldest. In this way, the system has always available to analyze the data that refers to the most recent activity of the application, and when the analysis has concluded, a subset of interesting information is written into disk and the buffers are cleared so that the next analysis will only consider the data generated from that point on.

In order to access the information that the tracing mechanism is recording during the program execution, an auxiliary thread is created when the traced program starts executing. More precisely, when the user program calls *MPI_Init* to initialize the MPI run-time environment, the tracing tool intercepts the execution and spawns a new POSIX thread to act as the tracing back-end. By default, this second thread remains latent while the program runs and is activated on demand of the analysis front-end. Being a thread of the traced application has the major advantage of having direct access to the memory buffers that store the trace data. Once active, the tracing back-ends are responsible for stopping and resuming the application (using signals or locks); extracting, filtering and manipulating data from the tracing buffers; sending data through the overlay network; performing local analyses and writing to disk. The actions to perform are determined by the front-end, which synchronizes all back-ends to come into play at the same time broadcasting messages through the overlay network.

3.3.2. The reduction tree

While the structure of the analysis framework is conceptually simple, creating and connecting the internal processes of the overlay network is complicated by interactions with the various job scheduling systems. To facilitate the process of creating the network, we rely on the Multicast/Reduction Network (MRNet) utility [87].

MRNet is a customizable, high-throughput communication software system for parallel tools and applications with a master/slave architecture. MRNet reduces the cost of these tools' activities by incorporating a tree-based overlay network (TBON) of processes between the tool's front-end and back-ends. MRNet uses the TBON to distribute tool communication and computation activities, reducing analysis time and keeping tool front-end loads manageable. MRNet-based tools send data between front-end and back-ends on logical flows of data called streams. The internal TBON processes use filters to synchronize and aggregate data sent to the tool's front-end. Using filters to manipulate data in parallel as it passes through the network, MRNet enables to efficiently compute averages, sums, and other more complex aggregations on back-end data. The organization of an MRNet-based application is perfectly suited for the structure of the analysis framework described: the analysis front-end corresponds to the root node in the TBON, the intermediate processes of the reduction tree run the MRNet filters, and the tracing back-ends play the role of the leaf nodes.

In the simplest environments, MRNet supports to start the processes directly using facilities like rsh or ssh. However, it is common in production environments to submit all requests to a job management system. In this case, MRNet is constrained by the operations provided by the job manager. For example, MPI implementations like OpenMPI [129], Intel MPI [130] or IBM POE [131] use environment variables to pass information, such as the process' rank within the application's global MPI communicator, to the MPI run-time library in each application process. In cases like this, MRNet alone cannot provide back-end processes with the environment necessary to start MPI application processes, and so it has to rely on the specific process management tools like aprun, mpiexec, mpirun or srun to start the back-ends. This mode of operation accommodates tools that require their back-ends to create, monitor, and control other processes, which is our case with the tracing back-ends attaching to the user parallel application.

To support this mode, MRNet has to start all processes except the back-ends recursively as in the normal instantiation mode using rsh or ssh. MRNet then waits for the tool back-ends to be started by the process management system to ensure they have the environment needed to create application processes successfully. To allow back-ends to connect to the MRNet network, information such as process host names and connection port numbers must be provided to the back-ends. This information can be provided via the environment, using shared filesystems or other information services as available on the target system. To collect the necessary information, the front-end can use the MRNet API methods for discovering the network topology details. This mode of process instantiation is referred to as *back-end attach mode.*

The resources needed to create the network can be either additional or shared with those used by the parallel application itself. In our case, since the overlay network is inactive while the application is running and vice-versa, we chose to share the resources because there is no actual penalty in the application performance and we reduce the amount of resources that are needed to run, yet using extra resources is also a viable option.

Once all back-ends have connected to the network, all the initializations are complete and both the tracing subsystem and the overlay network are ready to be used. Then all processes in the network stall in a blocking receive operation waiting for messages from their respective parent in the TBON. Periodically, the front-end node broadcasts a control message that gets propagated through the network awakening them in cascade. When the back-ends are activated with an analysis request, they pause the application and transfer all performance data (relative to the analysis that is going to be computed) that is currently stored in the buffers of the tracing tool, and then resume the application.

The data extracted from the tracing back-ends can be filtered in the intermediate nodes of the tree using the filters functionality offered by MRNet. Moreover, we have used this functionality to develop complex aggregations, taking advantage of the tree-like structure of the network to implement distributed hierarchical analysis algorithms, exploiting the full power of all the resources devoted to this structure. More details about complex aggregations are given through Chapters 4 to 6.

3.3.3. The analysis front-end

The analysis front-end orchestrates all the operations over the on-line framework. On one hand, it is responsible for deciding when to activate the back-ends to extract performance data and when to start the next phase of analysis.

The default interaction consists in triggering each new phase of analysis periodically every certain seconds, but more sophisticated interactions are also possible. For example, the system can automatically adjust the frequency of the analyses based on the speed at which the application is producing data so as not to exceed a set maximum data load in each analysis phase; based on the results of previous analysis steps (e.g. decide that it needs more data to reach a conclusion); or even it can ask the user interactively.

On the other hand, the front-end is a central point that receives data from all the back-ends. In this way, the front-end gets a global view of the status of all application's processes and how they are performing. Analyzing the aggregated information, it can take global decisions and provide dynamic feedback to the back-ends, instructing the tracing tool in what is the most interesting information to record. Through the following chapters we will show three protocols of analysis that take advantage of having such a global view of the application behavior to detect structure and periodicity in the data, leveraging techniques and methods from the fields of machine learning, data mining and signal processing.

3.4. Closing remarks

In this chapter we have presented an on-line analysis framework that combines a tracing system and an overlay network to automatically extract performance data from the execution of a parallel application, analyze it on-the-fly, and provides mechanisms to dynamically filter, summarize and manipulate the data to intelligently select which is the most interesting information to record.

The intelligence of the system lies in the analyses that are performed on the data and the conclusions that can be extracted from them. The following chapters show the different types of analyses that we have designed and how we use the results to discriminate between meaningful and irrelevant data, in order to meet the objective of generating minimal application traces that best describe the application behavior.

The modular construction of the system enables to easily extend its functionalities by connecting new plug-ins of analysis to the different components of the framework. In order to ease the development of new plug-ins, we have developed a utility middleware named *Synapse* that helps to develop MRNet-based applications that are structured in a tree-like topology, hiding all the difficulties that are inherent to creating, connecting and communicating the processes, and allowing the developer to focus on defining the actions that the different components of the tree (the root, intermediate nodes and leaves) will perform. More details on Synapse can be found in Annex B. Also, in Annex A the reader can find the user guide for the on-line analysis framework.

Chapter 4

On-line cluster analysis to find the program's most relevant execution regions

Cluster analysis is one of the most used explorative data mining techniques used for classification of data. The data elements are partitioned into groups called clusters that represent collections of data elements that are proximate based on a distance or dissimilarity function. For example, cluster analysis has been used to group spatial location prone to earthquakes or group related documents for browsing.

This type of analysis is particularly well suited to be applied on performance data. Due to the strong iterative nature of HPC applications and their frequent SPMD organizations, the performance achieved by the different processes and through the different iterations tends to repeat. By identifying similarities in the performance metrics that are being mesured during the program run, it is possible to detect performance trends and analogous processors. We can then focus on selecting representative samples for the different behaviors detected, describe these in all detail, and save us from capturing all the remaining redundant data.

In this chapter we present a cluster analysis protocol built on top of the on-line framework, targeted towards exposing the fine grain structure of the main computing regions of the program, for which we automatically generate a small, yet very detailed trace.

4.1. Background and motivation

When the amount of performance data collected from a parallel execution has grown large, finding ways to summarize the information becomes a cornerstone necessity. A common technique that has been widely used to deal with the summarization of large volumes of data is statistical profiling. 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 main weakness of profiles is that they mask the time- and space-varying behaviors. For example, the execution of a given subroutine may exhibit different times to solution depending on the parameters passed, the phase of the application when it is invoked, or the processor that is running. A profile just averages all these different occurrences and will not make any further distinction.

Cluster analysis is a better approach to overcome the intrinsic problem of loss of variability in the summarization process. As defined in [132], cluster analysis is the "unsupervised classification of patterns (observations, data items or feature vectors) into groups (clusters)". Since this process is a classification and not an aggregation of the information, the different trends in the data are not masked.

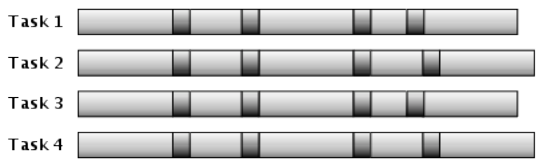
A common use for cluster analysis is to reduce the amount of information generated, taking into advantage the repetitive patterns of parallel applications. In works such as [108, 133, 113], the authors exploited the structure of the Single Program Multiple Data (SPMD) paradigm that the vast majority parallel applications follow. In a SPMD application it is expected that all processes involved perform the same sequence of computations simultaneously. In this context, cluster analysis has been demonstrated to be effective to group those processes that behave similarly. Selecting a representative process per cluster, the authors easily reduce the amount of performance data.

In [114], the authors were pursuing a different goal: to determine the computational structure of the application. Instead of grouping the processes that behave similarly, they focus on identifying similarities in the performance achieved in the computing regions executed by the processes, i.e. the phases of serial code between communication primitives or calls to the parallel run-time. Following this approach, they obtain a small number of clusters that expose the behavior of the program's computations, and provide the user with useful insight about the main performance trends.

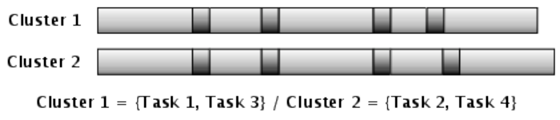
Figure 4.1 compares these two different approaches. Figure 4.1a represents the different processes in a parallel application, where we have distinguished the computations (light grey) and communications (dark grey). Figure 4.1b represents the results of the approaches presented in [108, 133, 113]: considering for example the duration of the processes as the similarity metric, they detect two different clusters grouping two processes each. Figure 4.1c represents the approach followed in [114], where they group the similar computing regions that appear in all processes, obtaining three different clusters according to their durations.

The latter approach maintains very fine-grain details about variabilities that may ocurr over processes and through time, enabling later in-depth analyses that can detect microscopic effects that may have a large global impact in the program performance. However, generating the clusters might be computationally very expensive. A post-mortem sequential approach that has to assess all the data captured from all the processes in a full parallel execution might result in millions of data points to process and extremely slow response times due to the clustering algorithms complexity.

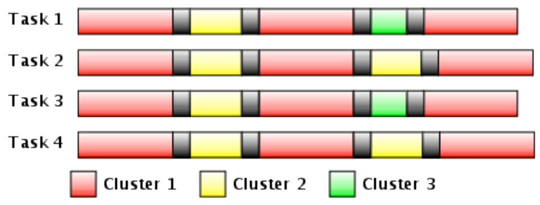
In order to support the computations' structure analysis of large-scale applications and their massive volumes of data, it is necessary to apply these techniques at run-time to intelligently focus on what is more relevant for understanding the application behavior, and thus notably reduce the amount of data generated. In this chapter, we introduce a novel alternative to tackle the trace scalability problem in the direction of intelligent selection of



(a) Original sequence of computations/communications to analyze



(b) Cluster analysis to detect processes with similar behavior



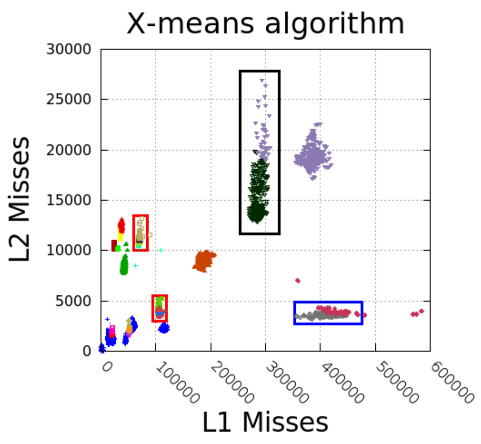
(c) Cluster analysis to detect computing regions with similar behavior

Figure 4.1.: Comparison of clustering analysis targets

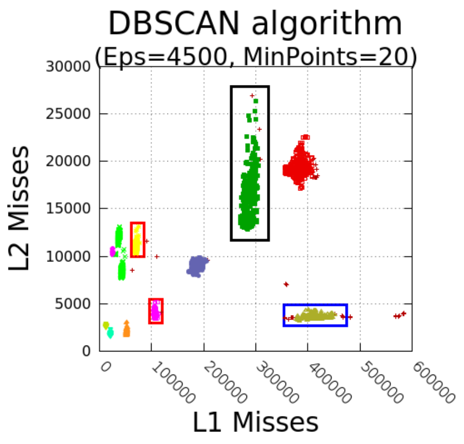
the traced information. We have designed a protocol of analysis that runs on top of the online framework presented in the previous chapter, to perform automatic cluster analysis on the performance data that is being generated during the run to accurately identify the application's structure and report its evolution over time. We are able to detect at runtime a fairly small region that faithfully represents its overall behavior. Performance data is stored in the trace with a wealth of detail for this particular region only, keeping it at a manageable size. The combination of periodic reports and a representative trace offers the analyst a general understanding understanding of the structure of the application, and allows easy identification of potential imbalances and their possible causes, while still providing all the advantages of a detailed trace-based analysis.

4.2. Computation structure detection based on DBSCAN

The target of the technique presented in [114] was to analyze the computational behavior of the CPU bursts executed by a parallel application. A CPU burst is the region in a parallel application between an exit from the parallel runtime and the following entry. Each CPU burst is characterized with a vector of performance hardware counters, erasing the time and space (process) components. Typically, the hardware counters used for this character-

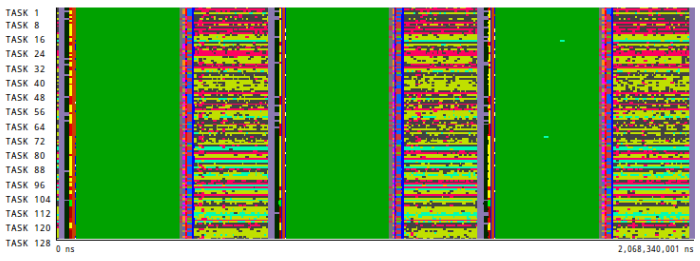


(a) X-means cluster algorithm results

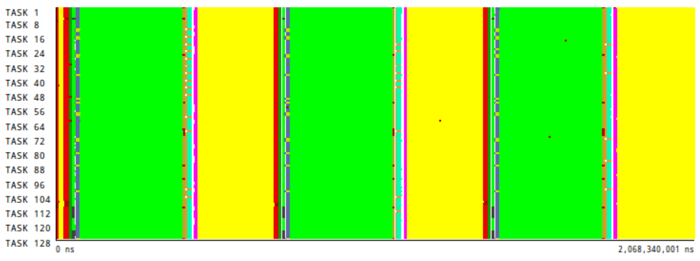


(b) DBSCAN cluster algorithm results

Figure 4.2.: X-means and DBSCAN algorithms comparison for 3 iterations of CPMD. Boxes highlight clouds of points with strong vertical or horizontal components where X-means splits the group and DBSCAN does not.



(a) X-means clusters time-line distribution



(b) DBSCAN clusters time-line distribution

Figure 4.3.: Comparison of X-means and DBSCAN clusters distribution over time for 3 iterations of CPMD

ization are Instructions Completed and IPC, focusing on a general performance view of the application. This combination enables to detect regions of the code with the same computational complexity (Instructions Completed), as well as to differentiate regions with the same complexity but different performance (IPC). Grouping the CPU bursts according to their similarity on these metrics results in clusters that represent the main behavioral trends of the program.

The choice of DBSCAN responds to the fact that performance counters data can have an arbitrary shape. Figure 4.2 compares the resulting clusters when applying two different clustering algorithms to the performance data from three iterations of an execution of CPMD [134]. Algorithms such as K-means or X-means, that are of common use in the field, are not able to detect the most suitable clusters due to certain assumptions about the data distribution that are no longer true for performance data. X-means (left plot) tends to partition the long ellipses, while DBSCAN (right plot) merges the disperse clouds into a single cluster. In terms of the quality of the computation structure detected, the data partition generated by DBSCAN is better. This can be seen in the trace time-lines in Figure 4.3. In these time-lines, the X-axis represents the time, the Y-axis represents the 128 processes involved in the application execution, and the color identifies the cluster of each CPU burst. In the top time-line, the clusters detected by X-means show high variability between processes, while when applying DBSCAN, the bottom time-line now shows a very clear SPMD structure, with all processes performing the same type of computation simultaneously.

DBSCAN [115] is the first and most cited density-based clustering algorithm designed to discover non-linearly separable clusters of arbitrary shape. DBSCAN requires two parameters: a distance (Eps) and the minimum number of points that form a cluster (MinPts). The basic idea is to explore the Eps-neighborhood of each point, and if it comprises more than MinPts a cluster is formed. For each point that belongs to the cluster, its Eps-neighborhood will also be added as long as its density is high enough. The cost of the algorithm is bounded above by $O(n^2)$, although the average case cost improves to $O(n\, log\, n)$ when the implementation includes a spatial index to optimize the searches in the data space.

When applied to the analysis of performance data, the scalability of the algorithm becomes a serious limitation. With tens and even hundreds of thousands of cores generating data every few nanoseconds, the amount of data to cluster grows extremely large, and so the clustering times quickly turn prohibitive, specially when the results are required in real-time.

4.3. The on-line clustering protocol

The on-line clustering protocol is built on top of the analysis framework introduced in Chapter 3. Figure 4.4 shows how this protocol integrates with the system.

The tracing back-ends are configured to capture performance metrics to characterize the computing regions of the application. Periodically, this data is aggregated in the analysis front-end through the reduction tree. Then, the computations structure is analyzed

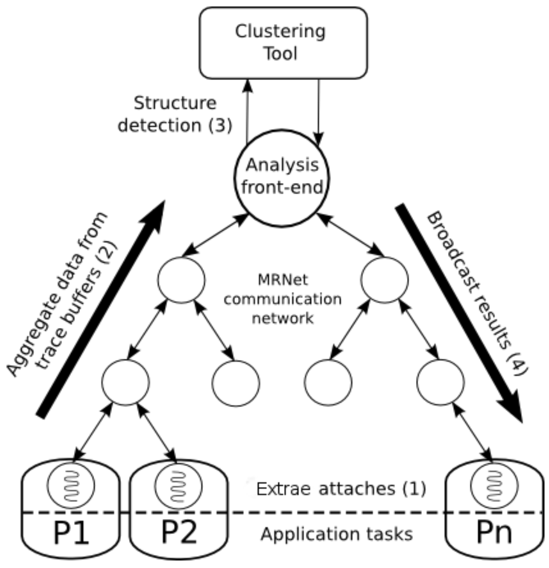


Figure 4.4.: On-line clustering analysis protocol workflow

centrally using a basic DBSCAN sequential algorithm.

This process of analysis enables us to determine at runtime a relevant region of the execution that illustrates the overall behavior of the application, and produces several performance reports, plus a full detailed yet small trace for this time interval.

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

4.3.1. Global cluster analysis of the program's computations

The analysis conducted consists of a computation structure detection using a density-based clustering algorithm [114]. The main purpose of this mechanism is to detect computing regions (i.e. CPU bursts) with similar behavior and, eventually, to identify phases of the application.

Every CPU burst is defined by its duration and a set of performance metrics read at the beginning and end of the region. The clustering algorithm can use an arbitrary number of these available metrics to characterize the application. Amongst all traced metrics, we select a combination of *IPC* and *instructions completed* to execute the clustering analysis with. These two metrics are useful to bring insight into the overall performance of the application. Trends in the *instructions completed* counter reflect regions with different computational complexity. In combination with *IPC*, it is possible to differentiate between regions with the same complexity but different performance.

As a result of the analysis we obtain a fine-grain characterization of the computing regions of the application, grouped by performance trends. The tool presents a numerical report with the average values for the different metrics that were used, and a scatter plot depicting the general structure of the application. Figure 4.5a shows an example of the structure of GROMACS [135]. It can be seen that the first three clusters (they are sorted according to the highest percentage of execution time that each cluster represents) exhibit different distributions. While cluster 2 is rather compact, clusters 1 and 3 present a significant dispersion in IPC and instructions respectively, and they all represent different computation regions that achieve divergent performance.

The analysis is carried out centrally at the analysis front-end of the framework. In order to accelerate the process and keep the system scalable, it is advisable to apply some reductions on the amount of data to cluster. While data is being retrieved from the tracing buffers and sent through the reduction tree, those CPU bursts whose duration is negligible are directly discarded. In this way, the clustering tool needs not to process irrelevant data, and the communication network is not flooded with useless traffic. Experiments in [114] show that the 99% of the total computation time is covered by only the 20% of all CPU bursts.

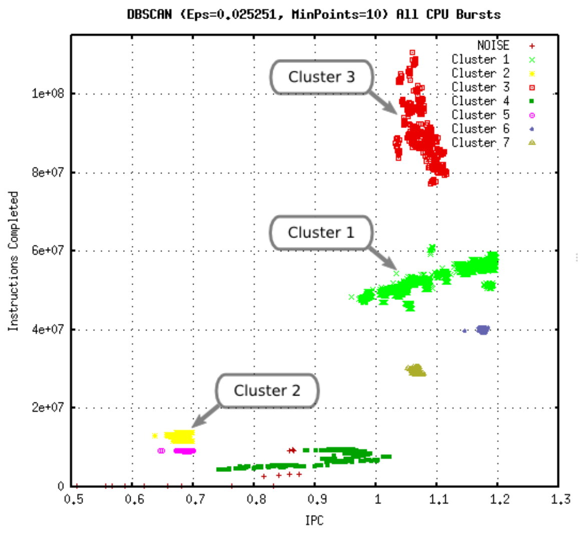
Even after having filtered all meaningless data, the cost of the clustering process grows more than linearly with the input size. In the applications tested (GROMACS, SPECFEM3D, NAS BT, etc.) more than 50,000 bursts can be generated in roughly 30 seconds on average, which can take up to 10 minutes to analyze. Among those bursts that make the cut, only a small subset is actually clustered to speed up the process. Taking as starting point the results of the clustering analysis with the selected training set, the remaining bursts are *classified* to their closest cluster using a nearest neighbor search. Different strategies have been tested to select a representative set:

- *Sampling across time.* Select a small percentage (10-20%) of CPU bursts of every process, randomly taken across the whole time range.

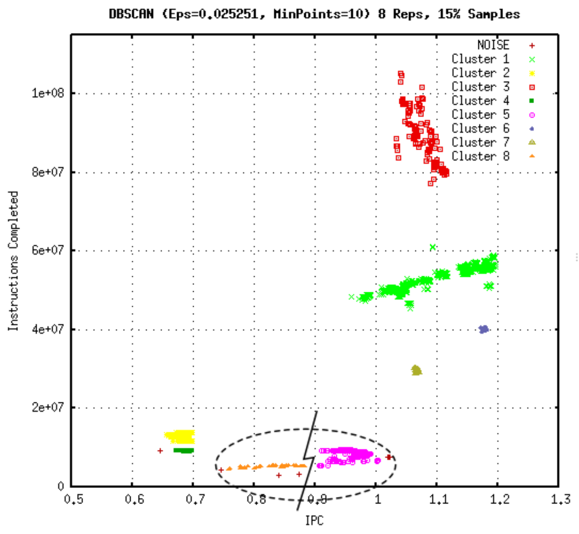
- *Sampling across space.* Select all CPU bursts from a few random processes.

In this way, a reduction of the input size to a few thousands of samples drops the analysis time to 5-10 seconds. Obviously the less samples are taken, the quickest but less precise results are produced.

Figure 4.6 shows the effect of different sampling strategies for the clustering in the resulting trace time-lines for the GROMACS application. The column on the left shows the results of taking samples across space from different number of processors. Figure 4.6a corresponds to an experiment where all the data from all processes was clustered, so this result can be considered as the reference. In Figures 4.6c and 4.6e the number of samples is reduced, clustering all the data from 32 and 16 random tasks only, respectively. The column on the right shows the results of taking samples across time. Figure 4.6b represents the results of selecting 25% random data points over time from each process. In this case, the results are identical to the reference case 4.6a. Figures 4.6d and 4.6f correspond to selecting 15% and 10% of random data points over time, respectively. As the reader can see, when reducing the number of samples, either across space or time, the results become more distorted compared to the reference case.

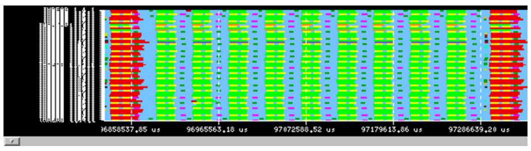


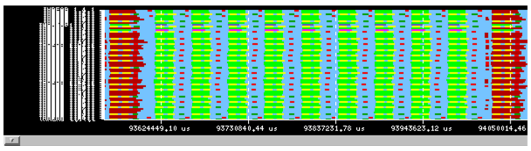
(a) Clustering all computing bursts

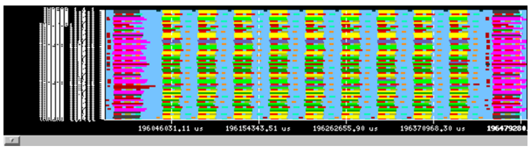


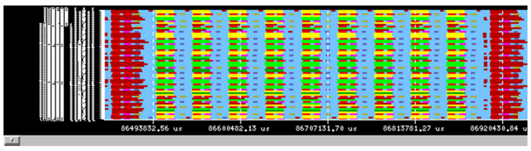
(b) Clustering a small subset of bursts and classifying the rest with a nearest-neighbor algorithm

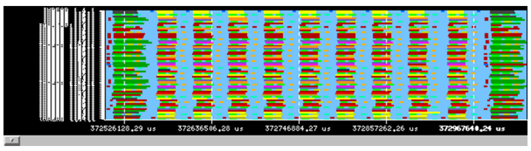
Figure 4.5.: Comparison of clustering the whole data set vs. a small training set for GROMACS

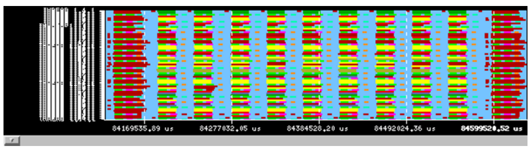

(a) All data from all processes


(b) 25% of random samples from all processes


(c) All data from 32 random processes


(d) 15% of random samples from all processes


(e) All data from 16 random processes


(f) 10% of random samples from all processes

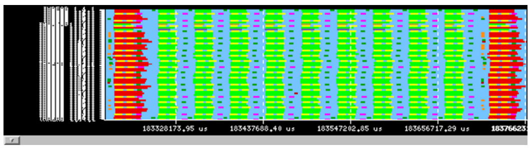

(g) Combining space sampling (all data from 8 random processes) and time sampling (15% of random samples from the remaining processes)

Figure 4.6.: Visual comparison of time vs. space sampling strategies for GROMACS

However, this does not imply a bad characterization of the application at all, but just a different one. Typically, a single large cluster can split into two or more (or even disappear) because samples in the middle were not taken, and subclusters were not close enough to merge. This is still a valid clustering that shows structure, though with different finesse. Depending on the application (e.g., if it is purely SPMD or not), one approach might be more convenient than the other, and their suitability responds more to the expert's objective than to any strict rule. In our experiments, the combination of both methods (i.e. include all bursts from a few processes and sample the rest) has proven to work fairly well for all situations, as it provides a better chance to capture both time and space variations.

Figure 4.5b corresponds to the same analysis than the one shown in Figure 4.5a. Instead of having all data clustered, we took all bursts from just 8 processes (out of 64), and sampled the 15% of bursts from the rest. In comparison, 75% less data was processed (2,500 bursts down from 10,000) and the analysis finished 20 times faster (6 seconds down from 2 minutes). Although the resulting clusters are less populated, and the former cluster 4 was split into two subclusters, the same overall structure can be identified.

The sampling percentage and number of representative processes might vary between applications. In our experiments, these specific values lowered the number of samples to just a few thousands, keeping the clustering analysis responsive enough to be computed on-line, while maintaining the quality of the results. An easy way to automatically adjust these parameters is to limit the data to be clustered to a known maximum value and reduce the number of samples proportionally.

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

4.3.2. Monitoring the application evolution

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

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

The application is considered *stable* when several clusterings in a row are equivalent. This is to say, the clusters found in the current analysis have similar shape, size and pos-

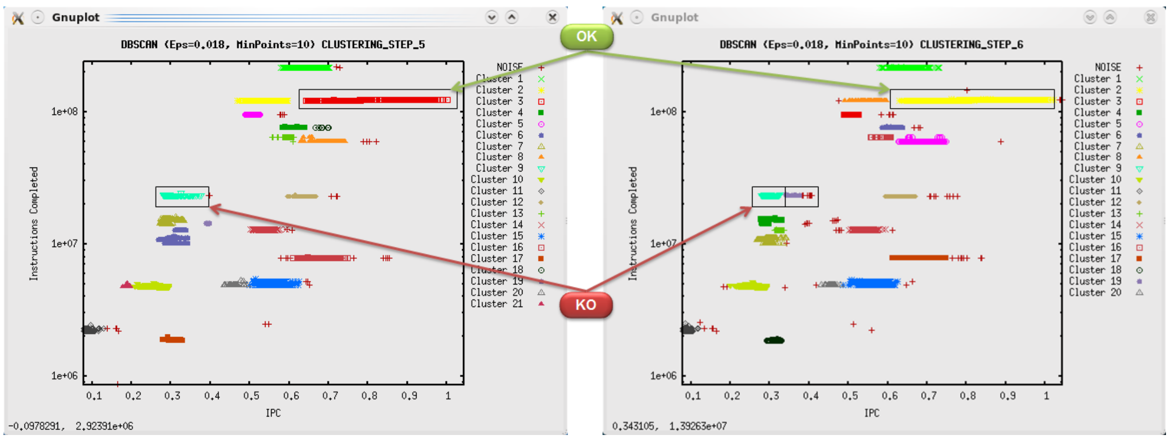


Figure 4.7.: Stability heuristic to compare whether two clusterings are equivalent. In 2D this can be seen as inscribing the clusters inside a rectangle and see if overlapping area is high.

ition in space than those detected in previous steps. Basing the precise numbers on previous tests experience, for every cluster we check that the values of the extremes of each dimension are within a $\pm 5\%$ margin of variance compared to the previous clusterings. In the case of using just two dimensions (*instructions* and *IPC* in our case studies), this can be interpreted as inscribing every cluster into a rectangle and matching those with high overlapping area, as shown in Figure 4.7. Two clusterings are considered equivalent if the matching clusters represent at least the 85% of the total computation time. In case the application would keep on changing behavior and the stability criterion could not be met, the requisites are gradually lowered to attempt to find the best possible region. This would also relax any inaccuracy that could be introduced due to the sampling and classification process where, as explained before, a single cluster can intermittently split depending on the selected samples. The inability to find a stable region can also be the consequence of too small trace buffers which can not hold an entire representative region. A possible alternative is to dynamically increase their size and check whether the results improve.

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

Figure 4.8 shows this heuristic applied to follow the evolution of the SPECMPI benchmark MILC [136]. Each clustering plot corresponds to the analysis of the data of subsequent time intervals of the application. As the reader can see, the structure of the application performance keeps on changing during the initial phases, while the initializations are still taking part, until the program enters the main computing loop (step 5 and beyond).

Once a stable region has been detected, clustering results are transferred back to the back-end threads, and every CPU burst is labeled with the cluster to whom it belongs to.

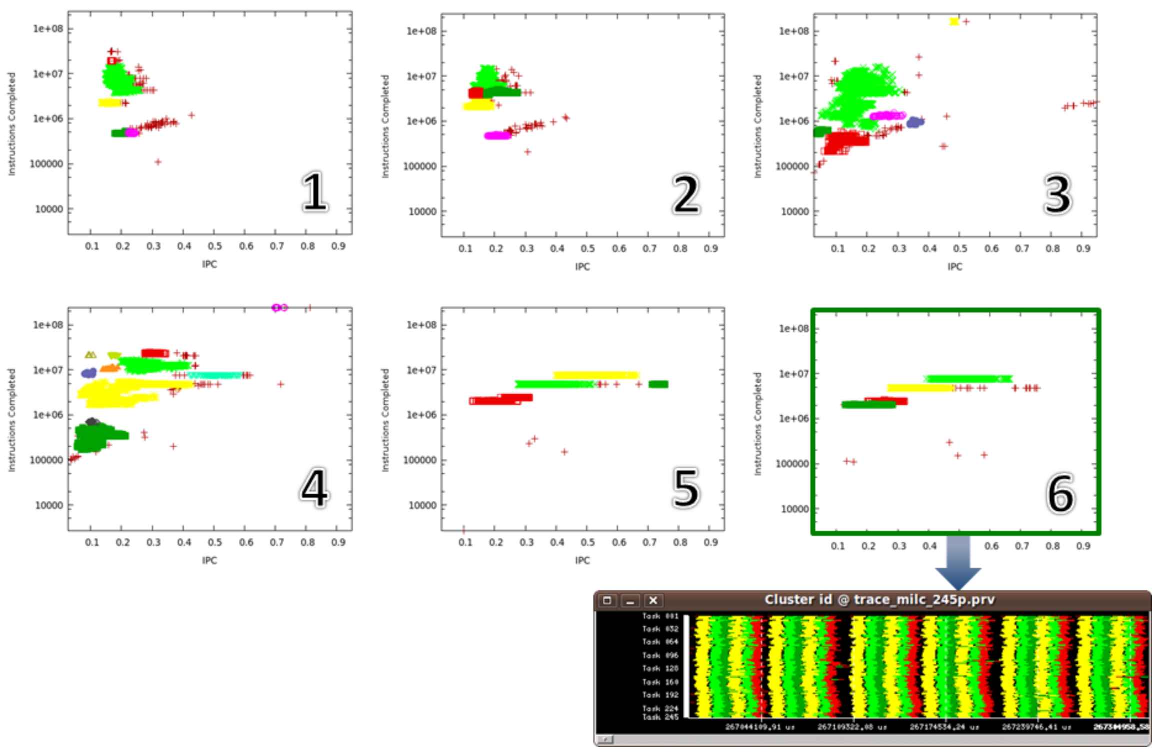


Figure 4.8.: Monitoring the evolution of MILC until a stable region is detected. The scatter plots show the structure of the computations detected for subsequent execution intervals.

Along with the clusters distribution, all performance data within the same time interval is flushed from the tracing buffers in order to produce a detailed trace of that region. In this example, the trace generated comprises about 6 full iterations with all the details, including MPI calls, communications, hardware counters, callstack information, plus the distribution of clusters over time as depicted in the time-line in Figure 4.8.

Structural information incorporated into the trace helps the user to identify those computing phases that they may expect of their application and to correlate them with the code. In addition, we obtain periodic reports that allow to understand the evolution of the application's structure until then. At this moment, the system does not necessarily have to stop, but can work the other way around. Taking the stable region as representative, subsequent clusterings can be compared looking for significant differences in structure. This might be a reflection of the application entering a different computation phase or undergoing significant perturbations, and detailed snapshots of these regions can be stored along with the trace.

	SPECFEM3D	Gromacs	MILC	Zeus-MP	Leslie3D
Number of tasks	64	64	245	256	512
Requested trace size	100 MB	200 MB	200 MB	350 MB	600 MB
Full run time	58 m	8 m	5.5 m	10 m	56 m
Full trace size	3 GB	20 GB	5.5 GB	22 GB	82 GB
Analysis steps	6	8	7	4	8
Clustering time / step	0.6 s	4.5 s	5.85 s	1 s	60 s
Classification time	1 s	1 s	1 s	1 s	3 s
Time to get results	15 m	2.5 m	3 m	3 m	12 m

Table 4.1.: On-line clustering experiments

4.4. Experimental validation

The aim of this section is to demonstrate the ability of the on-line cluster analysis to automatically identify the structure of an application as it runs, obtaining a representative trace along with performance reports showing its evolution, and how this all contributes to an improvement of the analysis methodology of parallel applications.

To this end, a variety of real applications and benchmarks were run on the Marenostrum supercomputer, a cluster comprising 10,240 IBM Power PC 970MP processors at 2.3 GHz (2560 JS21 blades) interconnected by a Myrinet network. Table 4.1 compiles the main configuration parameters and results of the experiments executed. The meaning of the main fields and some interesting quick facts are detailed below:

- The user specifies the final trace size, usually according to the amount of data the trace visualization tool can conveniently handle. We attained reductions in size up to two orders of magnitude while still obtaining a relevant trace comprising a few iterations of the execution. As the number of processes grows large, it begins to be necessary to increase the final trace size so as to capture a time interval that is meaningful enough. However, this progression is much slower than in the full trace size for a whole run.

- The frequency at which the analysis steps trigger is determined by the application itself. For instance, GROMACS generates an average of 2,500Mb/min of trace data. At such production rate, the requested amount of data for the resulting trace is produced every 5 seconds, so subsequent analysis are computed at that incidence rate.

- In most cases, a representative time interval is found after a few analysis steps. Intermediate reports are produced at every step, and a first trace right when the application is considered stable, which happens much before it concludes. The listed total time to get results includes initializations, the aggregated analysis times and the application computation itself.

- As explained in Section 3.3 only a small subset of data is actually clustered, while the rest is classified in order to keep the analysis times short. In most experiments the same number of samples were randomly taken to be clustered (approximately a maximum of 8,000 samples) but noteworthy variances in the analysis time can be observed between applications. This is due to the fact that the analysis cost does not only depend on the volume of data being processed, but also on the data itself and the input parameters of the analysis algorithm. Further details on this are discussed in [114].

The following sections present a more in-depth analysis of three of these applications to demonstrate how the proposed technique contributes for an easier analysis methodology and a better understanding of the application, and how the presented information can be used to detect interesting performance issues.

4.4.1. GROMACS analysis

GROMACS is an engine to perform molecular dynamics simulations and energy minimization. These are two of the many techniques that belong to the realm of computational chemistry and molecular modeling [135]. In our experiments, the application ran with 64 MPI tasks and we obtained a trace comprising just 10 full iterations.

Figure 4.9 shows the structure of the application at the different analysis steps. As the reader can see, it presents minor variations over time, with the exception of the Cluster 1 (green) that sometimes splits into two (steps 2 and 5). This can happen because of small variations of the application, or because the data points in the middle that make the sub-clusters merge are not very frequent and might not get sampled. Anyway, the application stays very stable in the following steps, as shown in the bottom left chart that displays the total percentage of time covered by the clusters that were considered equivalent by the stability heuristic. After two consecutive hits of 100% similarity, the region analyzed in step 8 is selected as representative. Even if the green cluster had continued splitting intermittently, since the comparison heuristic is adaptative and lowers the requisites as time passes, we would have come to the same solution anyway.

The scatter plot in Figure 4.10 shows a detailed view of the the structure of the different computing regions for the traced interval in terms of IPC and instructions. Coinciding with the different phases of the application, three predominant zones with different behaviors can be easily identified.

Both clusters 1 and 2 correspond to the heaviest computing phases, with higher IPC and number of instructions completed. A third group of less-populated clusters represent the smaller computing zones, with less than 20 million of instructions executed and lower performance.

While cluster 1 stretches well-balanced (the more instructions, the more IPC), cluster 2 presents a significant fluctuation in the number of instructions but constant IPC, which is indicative of a potential load imbalance. This will be an actual imbalance if those variances occur simultaneously in different processes. Further verification will require to check the distribution of clusters over time, that can be seen within the trace time-line.

A closer look to the region of clusters with lower performance (see Figure 4.11) reveals both instructions and IPC variances between different clusters. Having the plot correlated

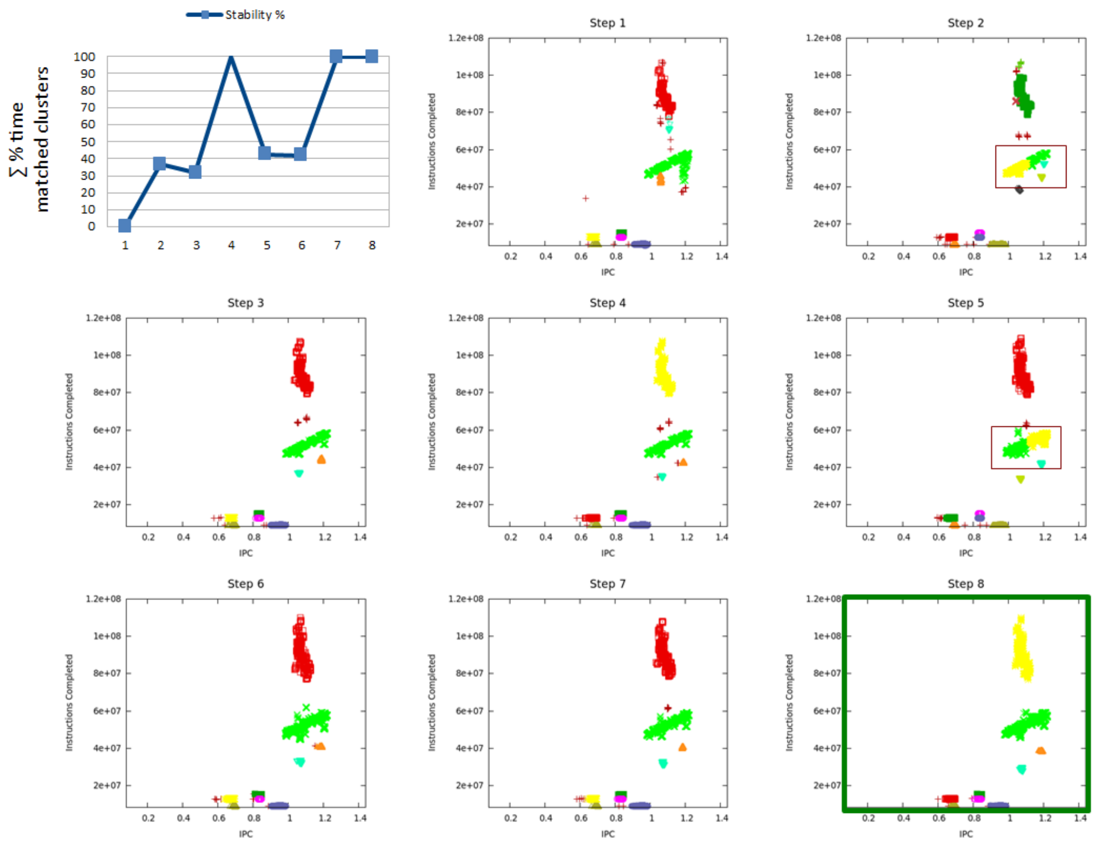


Figure 4.9.: Monitoring the evolution of GROMACS until a stable region is detected. The scatter plots show the structure of the computations detected for subsequent execution intervals. Top left chart shows the aggregate time covered by the matched clusters when applying the stability heuristic to every pair of images. The application is considered stable at step 8, which is selected as representative.

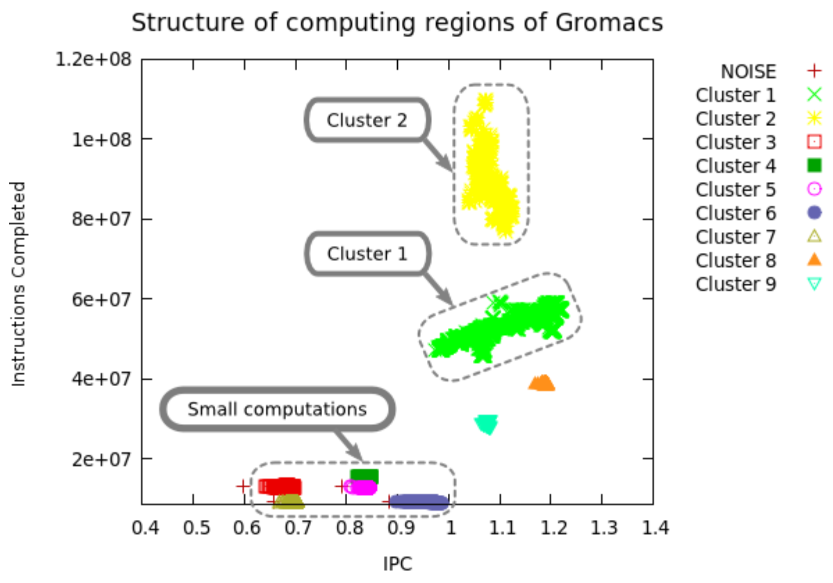


Figure 4.10.: Structure of the main computing regions of GROMACS. Two main compute-intensive phases are represented by clusters 1 and 2. Clusters 3 to 7 group smaller computations.

with the time-line, we can verify a non-SPMD behavior where different computations with variable performance are taking place simultaneously. Computing zones with constant instructions but variance in IPC and vice-versa are interspersed, which results in a duration variance, thus a real load imbalance.

Moreover, these small computations are being performed by a subset of the processes. Complementary processes (not depicted) perform those heavy computations that fall into clusters 1 and 2 only. These defines two disjoint types of processes that match the performance behaviors initially observed in the scatter plot, which correspond to the two types of job classes (PME and non-PME processes) in which the application is divided. Furthermore, every computing zone can be easily correlated with the source code, as the trace contains their associated call stack information. In this way, a precise recommendation to study the load-balancing characteristics of these particular regions of code can be made to the user.

4.4.2. Zeus-MP analysis

ZEUS-MP [137] is a computational fluid dynamics code for the simulation of astrophysical phenomena. It is included as part of the SPEC MPI2007 benchmark suite [136] for measuring and comparing MPI-parallel, floating point, compute intensive performance across a wide range of systems. In our experiments, we ran Zeus-MP with 256 MPI tasks, and obtained a trace comprising 4 representative iterations. The analysis completed in just 4 steps, meaning that the application behavior is very stationary.

The scatter plot in Figure 4.13a shows the structure of the different computing regions for the traced interval. Most clusters can be seen to spread horizontally, which typically

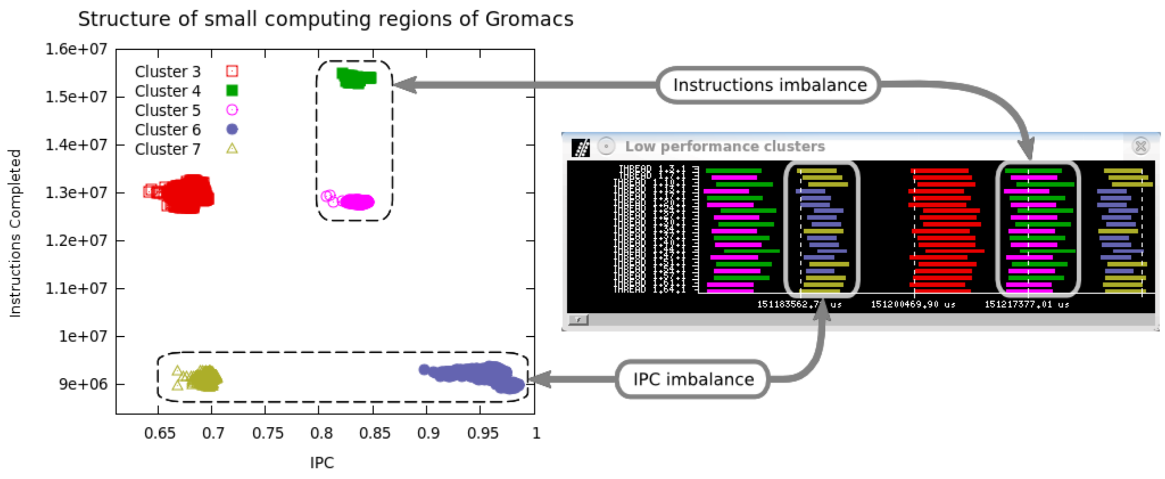


Figure 4.11.: Detailed view of clusters of GROMACS with lower performance and their distribution over time. Clusters with same IPC and different instructions and vice-versa denote imbalance problems.

means the computing regions are well balanced in terms of amount of work (they all execute the same number of instructions), but there are fluctuations of IPC which may result in load imbalances depending on how they are distributed over time, as we saw in the previous example.

The clusters information included in the trace helps the analyst not only to identify where do these imbalances happen, but to decide whether they are significant and quantify their impact. Focusing on the most compute-intensive clusters, the graph in Figure 4.13b correlates the variability between computation durations and their achieved IPC within each cluster. As one could expect, decreases in IPC that range from 20 to 30%, incur proportional duration increases.

The repercussion of these variances can be observed in the time-line in Figure 4.12, which shows the distribution of clusters over a single full iteration. While at the beginning of the iteration all processes start computing at the same time, the fastest processes keep on outrunning the slowest as the iteration progresses, making the processes more and more desynchronized. Point to point communications between computing phases hold the accumulation of delays back slightly as they keep partners synchronized, but it is not until the end of the iteration that a global collective operation absorbs the whole imbalance. A different work distribution could be considered in favor of a better balance of the varied computation costs, rather than the raw amount of work.

4.4.3. SPECFEM3D analysis

The software package SPECFEM3D [138] simulates seismic wave propagation in sedimentary basin. The solver is completely general and can be used to simulate seismic wave propagation on regional and local scales.

This case study focuses on showing the utility of the reports that are generated in ad-

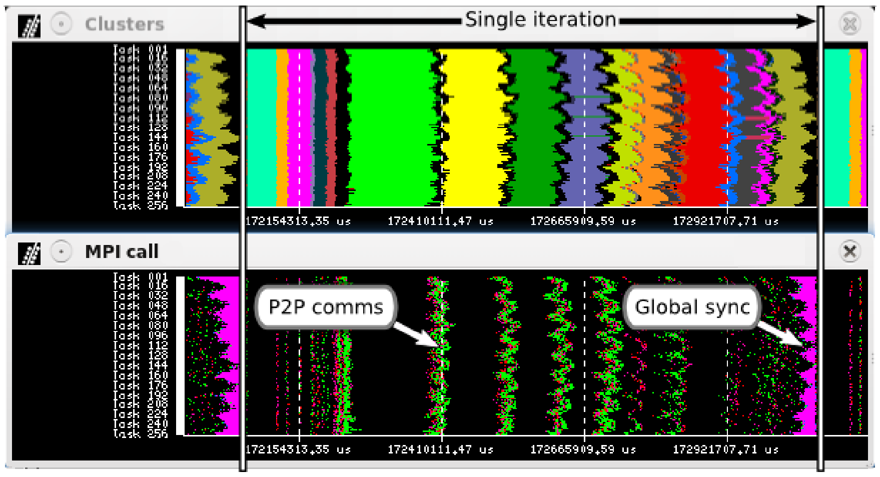


Figure 4.12.: Distribution of clusters over 1 iteration of Zeus-MP (top) and complementary MPI communications (bottom)

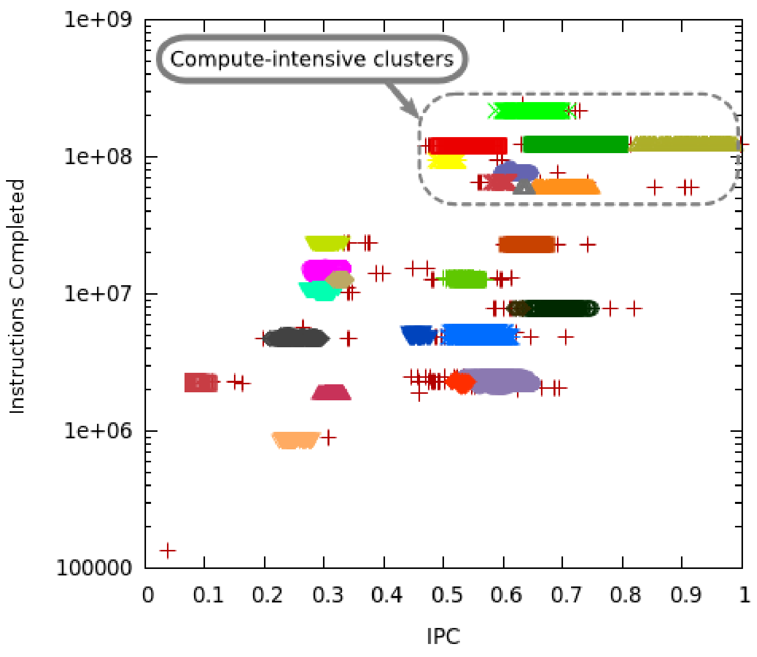
dition to the trace. Even though these are presented as complementary material to the trace, they are useful tools on their own to draw quick conclusions and get a better general understanding of the application without getting into the trace details. Going one step beyond, they can provide the expert with the initial clues to decide on which direction to move the analysis toward.

A first look to the scatter plot in Figure 4.14 provides a quick and very simple characterization of the algorithm structure, clearly showing three well-differentiated computing trends. Generally, the analyst's interest will lie in the performance of the most compute-intensive regions of the code, since it is where the useful work is done. Clusters 1 and 2 at the top right corner represent the two main computing phases of the application. Both issue a large number of instructions (more than a billion) and achieve high performance. However, they also present fluctuations in IPC, which are even more remarkable in cluster 2.

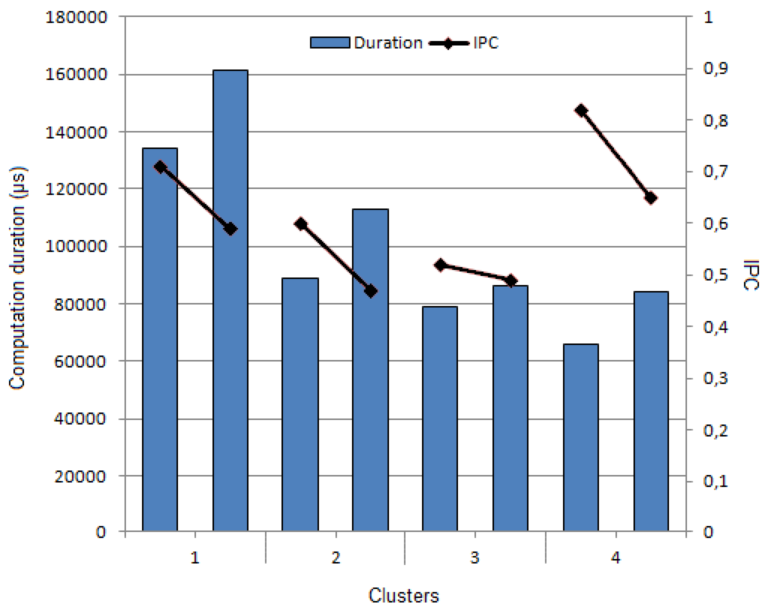
Along with the plot and the trace itself, the analysis produces a summary report listing average values for a wide range of relevant performance metrics for every cluster. Table 4.15a shows a small excerpt of this list. This information can be used to characterize the different clusters, and can be represented more visually through the Cycles per Instruction breakdown model [139], as shown in Figure 4.15b.

The CPI stack depicts the percentage of cycles in which the processor is completing work or stalled. The stall component is decomposed into the possible causes (i.e. empty pipeline; waiting for memory, arithmetic or floating point units). In this particular example, it is interesting to point out that more than 50% of the stall cycles in both clusters 1 and 2 are due to memory accesses (LSU stall cycles).

Whether the high percentage of inefficient memory accesses justifies the fluctuations in IPC that we observed in the scatter plot can be answered by delving into the trace. A more detailed analysis correlating the duration of computations against L2 data cache misses shows a proportionally increasing relation between both variables, highlighting a possible



(a) Most compute-intensive clusters present large IPC fluctuations



(b) Correlation of computing durations and IPC per cluster

Figure 4.13.: Structure of the main computing regions of Zeus-MP

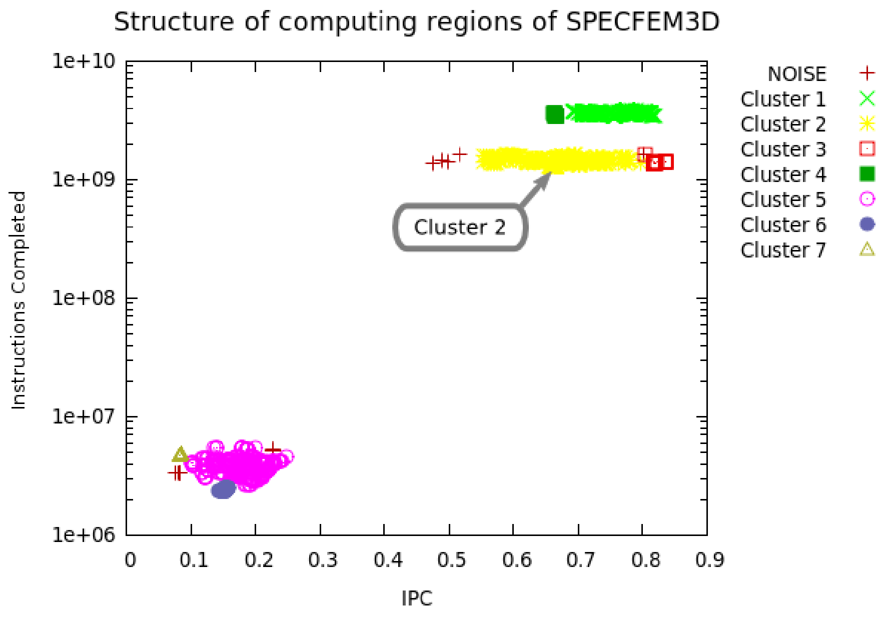


Figure 4.14.: Structure of the main computing regions of SPECFEM3D. Three main performance trends stand out.

improvement in the data access pattern.

4.4.4. Quality assessment of the clustering results

In previous sections we have shown the on-line clustering protocol applied to the analysis of different applications and how the resulting reports can be used to gain understanding of their structure and achieved performance. This approach provides greater analysis flexibility than a profiler, while sharply reducing the amount of data compared to a full execution trace. However, such data reduction goes through an inevitable data loss, where punctual variations between iterations might get lost.

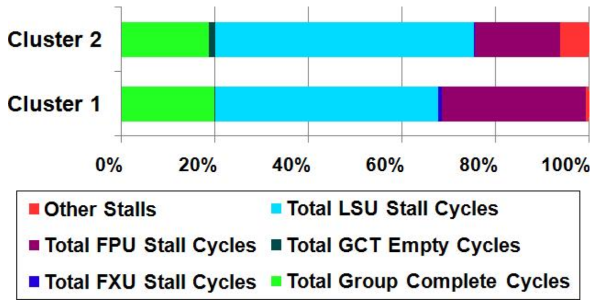
While not spurning the interest of these details, it becomes even more important to find a time interval that is illustrative of the application's iterative behavior. Any performance problem detected in this region is likely to be replicated thorough the whole execution, and this strong is the impact of an optimization as well.

In order to ensure that the traced region does not miss relevant areas and to evaluate how representative of the overall execution it is, we have compared our results to the performance data gathered by an external tool from a full run. To this end, we have used a profiler to obtain several metrics of the most relevant user functions. The same metrics were also computed for the traced interval that is automatically selected by our mechanism. In this way, we have verified that the values reported by both approaches remain practically the same. For the comparison, we took into consideration the percentage of time spent in each function, and the average values of instructions completed and elapsed cycles per call.

Table 4.2 compares the measurements obtained both from the complete and selected

	Cluster 1	Cluster 2
% Duration	70.23%	26.02%
Avg. Burst Duration (μs)	2,142.66	947.04
IPC	0.75	0.66
MIPS	1,702.70	1,501.75
MFLOPS	1,421.36	555.95
Memory BW (MB/s)	260.22	203.84
Memory instr/s	1,324.80	1,740.67

(a) Detailed list of metrics



(b) CPI stack report

Figure 4.15.: Performance reports per cluster of SPECFEM3D

time intervals of GROMACS application. Columns 1-3 show the values for these metrics as reported by TAU profiler [45] for the whole run of the application. Columns 4-6 show the same metrics computed for the representative traced region. As noted, differences are marginal (percentage differences are lower than 4% and under 1% in most cases) and can be due to small variances between iterations, executions, or even to the different overheads introduced by the respective measuring tools. Showing all user functions the same overall behavior is a simple indicative of the traced region as a fair representative for the whole execution.

4.5. Related work

The Paradyn Parallel Performance Tools project [62, 84] paved the way for intelligent selection of performance data. It provides a profile-based on-line bottleneck search algorithm that automatically looks for a set of known performance problems. This is an iterative process of formulating hypotheses and inspecting related performance data in an attempt to confirm or reject them. Periscope [43] proposes a very similar on-line distributed analysis, having agents autonomously searching for performance problems of a small subset of processes. Similarly, OPAL [67] used hypothesis and proof rules, except that refinements caused new executions to be started with new data to be collected via selective tracing. DynTG [68] does not include automatic selection of performance data. Instead,

	Complete execution			Traced interval			Difference %		
Function	% Time	Kinstr	Kcycles	% Time	Kinstr	Kcycles	% Time	Kinstr	Kcycles
do_nonbonded	23.72%	24,709	22,349	23.94%	24,700	22,533	0.22%	0.04%	0.82%
solve_pme	10.47%	6,795	9,913	10.52%	6,776	9,898	0.05%	0.28%	0.15%
gather_f_bsplines	5.69%	5,286	5,387	5.64%	5,248	5,302	0.05%	0.72%	1.58%
dd_pmeredist_x_q	5.38%	4,063	5,044	5.71%	3,911	4,943	0.33%	3.74%	2.00%
make_bsplines	5.16%	3,521	4,879	5.06%	3,476	4,759	0.10%	1.28%	2.46%
nsgrid_core	4.07%	44,608	38,321	4.18%	44,579	39,771	0.11%	0.07%	3.78%
spread_q_bsplines	4.07%	4184	3,812	3.99%	4,147	3,736	0.08%	0.88%	1.99%
ewald_LRcorrection	3.42%	2,383	3,216	3.46%	2,369	3,246	0.04%	0.59%	0.93%
csettle	1.40%	454	662	1.42%	451	656	0.02%	0.66%	0.91%
calc_idx	1.04%	798	989	1.04%	806	974	0.00%	1.00%	1.52%

Table 4.2.: Comparison of the performance measurements between the whole execution and the automatically traced interval of GROMACS.

the user interactively instruments the program as it runs by clicking on source code lines in a browser, based on their own observations of the results presented for the selected zones.

Automatic trace analysis systems such as Scalasca [46], AKSUM [98] or Kappa-PI [93] mainly differ in the phase the analysis is applied. Their post-mortem approach allows a thorough search for known performance bottlenecks over the whole detailed event trace. Nevertheless, their ability to identify performance issues is limited by the knowledge bases themselves, and the more extensive they are, the more time the analysis is going to require.

The application of clustering techniques in real-time analysis has been approached by Nickolayev et al. [108]. In their work, K-means statistical clustering is taken in the Pablo Performance Analysis Environment [109] to reduce the volume of captured data by identifying and recording events only from representative processors. Instead we aim to detect the internal structure of the application, basing on a density-based clustering strategy proposed by Gonzalez et al. in [114].

Overall, our approach shares common aspects with the tools above. Similarly, we propose to inspect and filter the performance data automatically and at run-time. However, two fundamental poses differ in our contribution. First, we do not rely on summarized profiling snapshots to compute the analysis. On the contrary, we analyze the detailed stream of traced events, where small variances that may have a significant impact are not masked. Second, our objective is not to produce a summary report of common performance bottlenecks, but a detailed description of the application structure along with a representative yet small trace. While the structure reports are meant to provide the analyst with a general understanding of the application, having a recorded trace enables them to formulate their own hypotheses about the causes of the observed performance and design specific metrics to corroborate them, which may lead to the discovery of new and unexpected patterns that model an inefficient behavior.

4.6. Closing remarks

In this chapter we have presented an analysis protocol running on top of the on-line analysis framework that leverages clustering techniques to automatically detect the performance trends that the computing regions of a parallel application exhibit during the execution. This information enables us to minimize the amount of data emitted to the trace while maximizing the amount of relevant information presented to the analyst. This contribution tackles the trace scalability problem and allows tracing-based solutions to be used even in large-scale scenarios. Furthermore, our on-line solution overcomes the limitations of post-mortem analysis tools regarding the manipulation of large volumes of data and the inevitable data loss that results from the necessary summarization processes.

Performance measurements collected during the execution are periodically analyzed using density-based clustering in order to detect the application's structure at run-time. As a result, we produce a compact trace of a representative region of the whole execution, preserving very fine-grain details about time and space variabilities that are important to be analyzed. In addition, periodic performance reports and trend plots are produced, providing extra insight about the structure of the application at the respective time intervals and its evolution over time.

Unlike other approaches, the target of our on-line analysis is not to find and filter periods for a mere data compression. Instead, we aim at identifying the application structure as it runs, and decide which are the most interesting regions to present to the analyst, who can then conduct the analysis not being limited by predefined and potentially incomplete knowledge-based rules. In this regard, automatic analysis tools like expert systems and performance modeling frameworks could immediately benefit from the structural information that we discover in several ways. On one hand, such automatic analysis tools could still do their job maintaining the same levels of accuracy but much more efficiently if they are presented with a subset of the whole data that is equally representative of the potential performance problems. On the other hand, the insight extracted about the program's structure could be used to enrich the set of rules of a knowledge-based system for a more precise diagnosis. Furthermore, it would be possible to build new performance models based on the clusters information to better undestand and predict changes in the program's structure.

Our work opened several new research horizons that we have explored. First, although the work presented relies on clustering, the underlying infrastructure enables the application of other analysis techniques towards the intelligent selection of performance data. A downside of the clustering approach is that there is not precise control over which part of the code will correspond to the traced region. In Chapter 5, we discuss on how to precisely identify the program's iterations to delimit more accurately the traced regions, using signal processing techniques.

Second, it is interesting to redesign the density-based clustering algorithm DBSCAN to follow a hierarchical approach that would make the most of the reduction network, by running distributedly over all the intermediate processes of the tree so as to analyze the data incrementally and to keep the system more scalable. The details of a hierarchical parallelization for DBSCAN are discussed in Chapter 6.

Finally, Chapter 7 develops the idea of comparing clusterings from different time intervals to perform an automatic analysis of the evolution of the application over time, and further extends it to present a technique that enables to perform very versatile multi-experiment analyses.

Chapter 5

On-line spectral analysis to generate multi-detail traces

Computation-intensive applications are customarily characterized by presenting an iterative dataflow. In most cases, the program behavior will not change over time, and so any significant performance problem that occurs over the course of a single iteration will also appear every single other iteration. Therefore, storing performance data for every iteration of the program does not necessarily provide much more useful information than tracing just a few selected time steps. Identifying the iterative pattern of the application aims at discarding redundant data while maintaining the precision of the analysis.

Determining which iterations are interesting to trace is not a trivial task. The user could easily annonate the main loops in their code, but that would be inpractical for the analysis as it requires source code modifications and recompiling. Static code analysis techniques can be used to instrument the loops automatically, but the decision of which ones are relevant has to be taken without any knowledge about the performance behavior exhibited during the loop, and so these mechanisms alone lack of criteria. In this chapter we present a technique based on spectral analysis of signals to dynamically discover the iterative pattern of an application considering its performance characteristics.

The information about the periodic phases of the program enables the on-line analysis framework not only to intelligently select representative periods to trace, but also to decide the granularity of the information gathered for each region. As a result, the execution is completeley characterized at different levels of detail, reducing the amount of data collected while maximizing the amount of useful information presented for the analysis.

5.1. Background and motivation

The problem of trace scalability has been tackled before from several angles: efficient trace formats and compression schemes, automatic methodologies to analyze large volumes of data, parallel tools and distributed processing. A different approach to the problem is to exploit the highly iterative nature of HPC applications and their frequent SPMD organization. In this sense, performance measurements can be repetitive, and a periodicity analysis is going to detect loop-phases and expose the application's iterative structure.

Ideally, a tracing tool should be able to automatically discard all irrelevant measurements, focusing the analysis just on the essential information and filtering out the redundant data. In Chapter 4, we introduced the use of clustering techniques to identify the structure of the computing regions and monitor the evolution of the application to see changes in its behavior. The analysis was repeated periodically and the results compared. If the resulting clusters were similar, that was indicative of the application being in a repetitive phase, and then just one of these phases was traced as a representative of the execution.

While this method has proved useful to reduce the amount of data emitted to the trace, it lacks precision when it comes to delimiting the boundaries of the traced region since the exact start and ending points of every iteration inside the loops of the program is not known. At best, this rough approach may result in tracing more iterations than what is actually needed for an effective analysis, if not the very opposite.

Spectral analysis techniques such as Fourier transform and Wavelet analysis have been previously applied post-mortem for pattern recognition in event traces [140]. The analysis of signals to detect phases in the application is advantageous for several reasons. First, the time-stamped sequence of performance events that is contained in a trace can be easily and naturally represented as a function of time. Second, a signal can reflect very accurately all the high-detail variabilities across time and space (processors) that are important for the analysis. Third, it is possible to isolate a given performance metric and perform the analysis based only on it, and therefore, to determine the impact of this parameter over the whole execution. Finally, the algorithms based on signal processing theory have low computational complexity, they rest on a very solid mathematical foundation, and they are able to provide relevant information automatically.

In this chapter we show how these techniques can be adapted to run in real time, enabling us to identify the different phases and periods of the application while it is being executed. This knowledge incorporated into the on-line analysis framework provides the opportunity to automatically select an exact number of representative iterations for every different pattern, and dynamically decide whether to keep, discard or aggregate the information for the repetitive phases at different levels of detail. As a result, a minimum trace is produced that maximizes the amount of relevant information presented to the analyst and fully characterizes the whole execution, notably simplifying the subsequent analysis in terms of the amount of data that has to be processed.

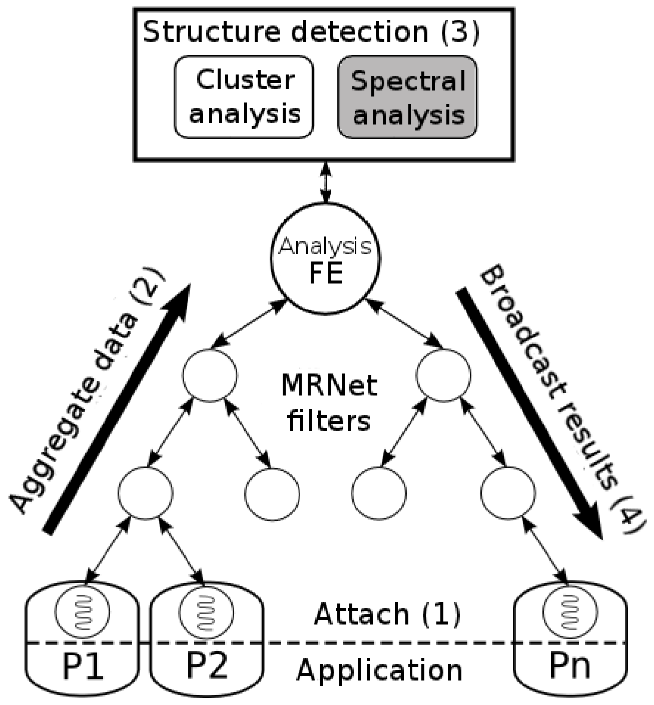


Figure 5.1.: On-line spectral analysis protocol workflow

5.2. The on-line spectral analysis protocol

The on-line spectral analysis protocol is built on top of the analysis framework introduced in Chapter 3. Figure 5.1 shows how this protocol integrates with the system. The objective of this analysis is to identify and differentiate every different periodic behavior of the application. This addition brings to the fray the ability to precisely control what information is relevant to be traced and what is subject to be discarded.

5.2.1. Generation of a signal from performance data

In order to apply signal processing techniques to detect periodic phases in the application, it is necessary to characterize the execution with signals. The first step then consists in deriving a signal from the trace data that is being collected by the tracing back-ends. We convert the stream of time-stamped performance measurements into a time-discrete function described by triplets of time, duration and value of the metric used to compute the signal, representing the time evolution of the given metric during the execution of the application.

Any of the performance measurements available in the trace can be used to build the signal. In this way, we can select metrics that focus on the execution phases (e.g. the number of processes that are computing simultaneously), metrics that focus on the performance characteristics (e.g. the instructions executed per cycle), or metrics that put more emphasis on the communications (e.g. number of message-passing calls or number of in-flight communications).

An interesting metric to analyze is any that reflects the periodic structure of the application, and depending on the case, some metrics might show phases more clearly than others. The selected metric does not necessarily have to distinguish code regions unequivocally, but to represent a marked sequence of repetitions where to find a clear periodicity. Among the possible options, we usually select the duration of the *CPU bursts*, defined as the duration of the computing regions between consecutive calls to the parallel run-time.

This metric has the major advantage of being applicable regardless of the parallel programming paradigm the application uses. And more importantly, it shows a very strong pattern despite small variations or fluctuations the application might undergo, as exemplified in Figure 5.2 for the NAS BT benchmark [141].

5.2.2. Signals integration

One such signal is independently generated per every application task, using its local performance data, and therefore represents the periodic behavior of a single task each. In order to analyze the application globally, we integrate all individual signals over the reduction tree. These are sent from the back-ends of the tree to the front-end, and they are added up in the intermediate nodes as they go up in the tree. The aggregation simply consists of increasing the value of the resulting signal whenever any of the summed signals is non-zero, as shown in Figure 5.3a. When the application model is SPMD, all task's signals will approximately show the same amplitude in the same time intervals, thus the summed signal will present the same shape with amplitudes scaled by the number of tasks. This effect strengthens the periodic behavior of the signal and facilitates the later periodicity analysis.

If all tasks and node clocks were perfectly synchronized, all variations in the signals amplitudes would occur in the same exact timestamp, and the total number of different transitions to describe the summed signal would remain constant. But in reality, small clock skews and performance variations between processors make the tasks misaligned, and so the aggregated signal to grow linearly to the number of tasks, as shown in Figure 5.3b. In this case, the number of transitions that are needed to express the summed signal more than doubled, and the result does not better reflect the state of the application.

In order to keep the summed signal scalable, we added a second filter operation in the

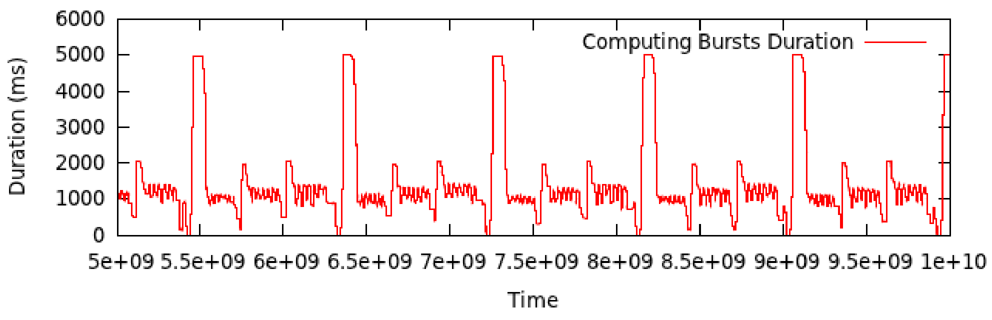
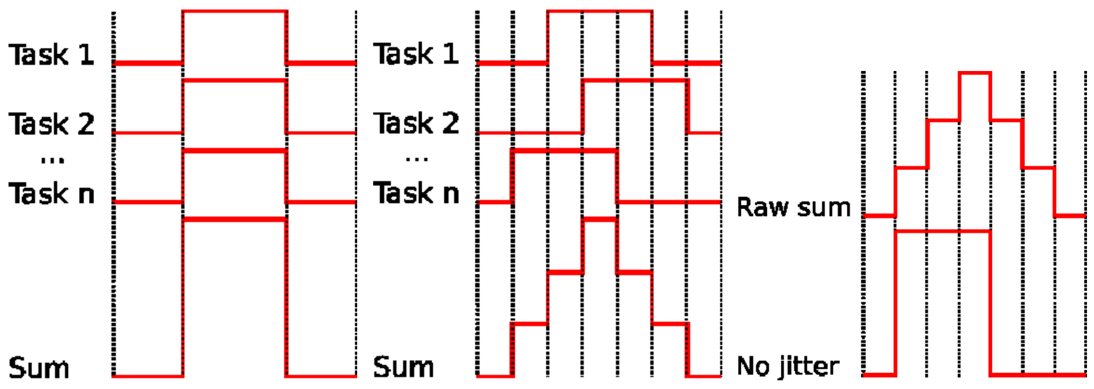


Figure 5.2.: *CPU burst duration* signal of the NAS BT benchmark



(a) Signals aligned (b) Signals misaligned (c) Jitter reduction

Figure 5.3.: Signals integration process

intermediate nodes to clean the jitter. Consecutive rising or falling edges as shown in Figure 5.3c (top) are accumulated in a single flank transition, resulting in the simplified signal 5.3c (bottom).

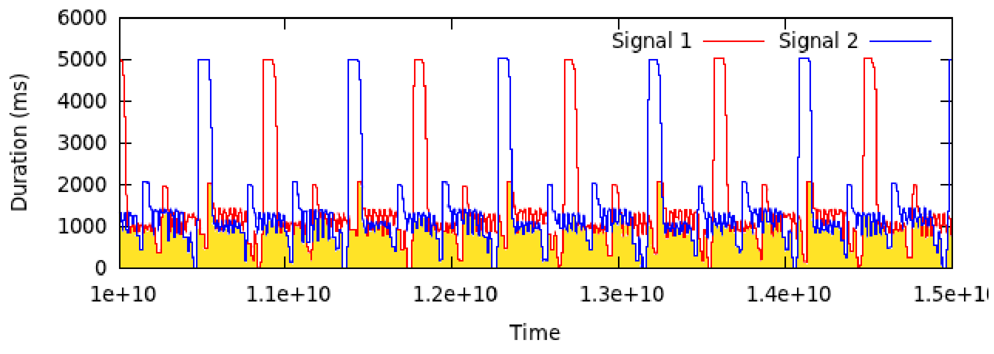
The combination of both filters sharply reduces the size of the integrated signal, keeping the system scalable in terms of memory consumption and required computation for the analysis to be computed at the front-end, which receives a single signal to analyze that expresses the global periodic behavior of the whole application.

5.2.3. Spectral analysis of signals

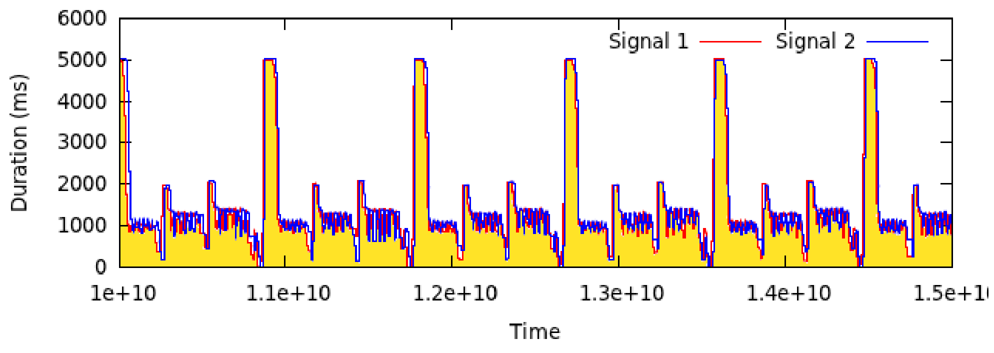
The spectral analysis mechanism that we use [140] consists of four main signal processing techniques to detect periodicity in the signal. First, morphological filters derived from the theory of Mathematical morphology are applied to clean up perturbed regions of the trace (e.g. zones of disk I/O). Second, Discrete Wavelet (DWT) and Fast-Fourier (FFT) transforms are used to identify high-frequency regions in the signal, which are directly related to the execution of the internal loops of the application's source code, and allows to separate regions according to their frequency behavior, i.e. a region with a small iteration which is repeated many times is separated from another region with no periodic behavior. Then, autocorrelation analysis provides a way to detect the period within the regions with strong periodic behavior. And finally, cross-correlation enables to detect which iterations are the most representative of the whole periodic region.

If the execution has the typical structure of HPC applications, the periodic phase will be detected because it has a strong high frequency behavior. Besides, if the execution contains multiple periodic phases, the analysis will also detect them because DWT is able to separate the periodic phases characterized with different frequencies. Assuming that each periodic region detected comprises an iterative pattern where there are no significant differences between the repetitions of the pattern, it is possible to select just a few of them as representatives.

At the end of the spectral analysis, the mechanism reports the time bounds for each



(a) Intersecting area is low when signals are decoupled



(b) Maximum intersecting area when signals are coupled

Figure 5.4.: Cross-correlation of two signals

periodic region detected, the average length of the period, and thus the total number of iterations within the periodic region. Additionally, the user can specify a number of consecutive iterations, and the tool recommends those where the period is more marked and presents less fluctuations.

This information is then used to monitor the evolution of the application over time. At fixed time intervals, or whenever a given volume of new performance data is produced (orchestrated by the analysis front-end of the on-line framework), a new step of analysis triggers. Subsequent analyses produce a sequence of signals, whose shape (length, frequency, amplitude) can be compared to see changes in the application structure. For this comparison, we perform a cross-correlation of the signals in pairs, which is a measure of similarity of two waveforms as a function of a time-lag applied to one of them. This can be seen as sliding one function over the other, to find the point where the area under both functions is maximum, as shown in Figure 5.4.

The result of the cross-correlation is expressed as a percentage of similarity. When this value is greater than a given threshold, both signals are considered to be equivalent, meaning the periodic behavior remains the same. If the signals differ, then the application is either under significant perturbations or entered a different computing phase, and the value for the cross-correlation will drop. Contrasting empirical experiments, we have

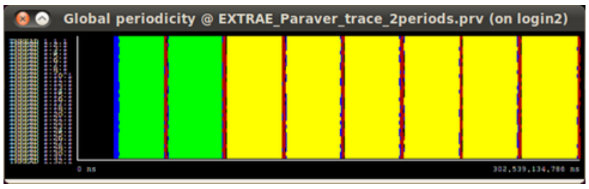

Global periodicity @ EXTRAE_Paraver_trace_2periods.prv (on login2)
0 ns
302,539,134,786 ns

Figure 5.5.: Global periodicity view in the full execution of PFloTran with 32 MPI tasks

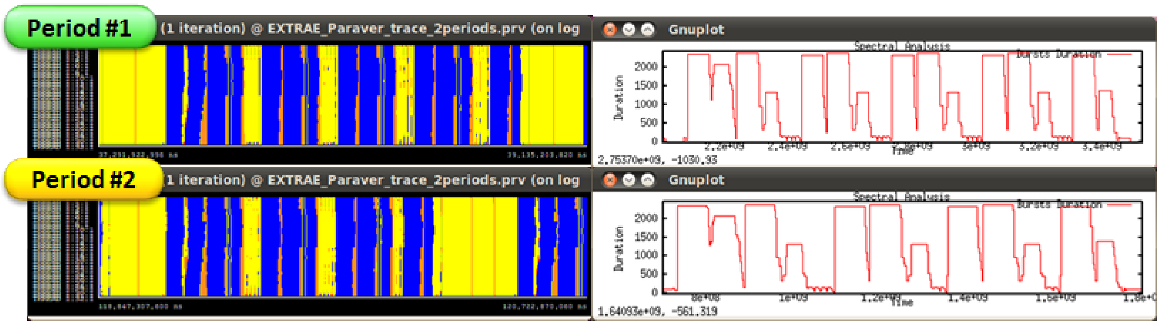


Figure 5.6.: Detail of the two different periodic behaviors observed over time during the execution of PFloTran

determined that an appropriate similarity threshold for the cross-correlation ranges from 90 to 100% for synthetic benchmarks, which are usually very repetitive; 80% is a reasonable value for most normal cases; lower than 70% denotes noticeable differences; and very significant differences under 50%.

The sensitivity to execution variations can be tuned with this parameter, so the higher the threshold, the more likely two periods are detected to be different due to small fluctuations, and the more information is finally traced. Other parameters that control the amount of traced information are: the maximum number of periods to trace, the minimum times a period has to be seen before it is traced, and the maximum number of iterations to trace per period.

Having prior knowledge of the application (i.e. there is certainty that the application presents just one type of periodic behavior), these parameters can be easily adjusted so that the analysis produces results before the execution completes. Otherwise, the system will monitor the whole run, periodically repeating the analysis across the execution. For every new pattern of periodic behavior that is detected, it will then trace in detail an exact number of representative iterations. Generally, a minimum of 2 or 3 full iterations is enough for an effective analysis, but as much extra data as specified can be included beyond that while keeping the total trace size under control. If the observed application behavior remains unchanged, the system will not trace any more samples, but can be configured to summarize or completely discard the surplus data.

Figure 5.5 shows an example of the on-line spectral analysis protocol applied to the full run of PFloTran using 32 MPI tasks. The analysis was performed every 30 seconds, introducing an overhead of 1.5 seconds on average. As a result, the system detected two different periodic behaviors, shown as the green and yellow phases in the timeline. The size of the resulting trace, that comprises a sample of a few iterations per each of these two periods, is 275 Mb, down from 8.5 Gb for the full trace. This application executes 100 iterations of the same code, so we could have expected to find a single type of periodic behavior. The fact that two different periods are detected means that the behavior of the application changed over time.

What the analysis found is that the length of the period decreases from 1,600 ms (green phase) to 1,300 ms (yellow phase). The reason for this is not because of performance perturbations, but because the execution flow of the program changes. Figure 5.6 shows how the main iterative phase of the program is comprised of 4 sub-phases during the green period, while it is only 3 sub-phases during the yellow period. This happens because the algorithm is a solver that can converge to a stable solution faster or slower. These differences change the shape of the signals depicted on the right, decreasing in turn the value for the cross-correlation below the similarity threshold, and thus both phases are considered different and a sample for each one gets automatically traced in detail.

5.3. Multi-detail levels of trace data

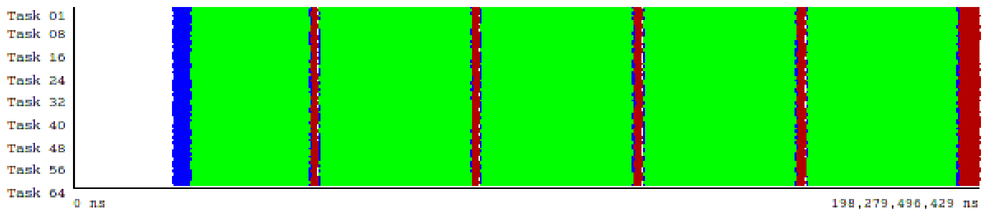
In the previous section we have described the ability of the system to detect patterns of periodic behavior and automatically select a few representative iterations to be traced in detail. In this way, we avoid storing redundant data and keep the trace manageable. But this does not necessarily mean that we can not have any information for the rest of the execution, other than these selected iterations.

The tracing back-ends are able to trace information under two main modes of operation: *full detail* streams of events (all function calls, hardware counters, communications, call stack, etc.), and *summarized traces* [120]. The latter focuses on providing information only for the most intensive computing regions, so that the main structure of the application can still be analyzed in detail. The most negligible but frequent computations that make the trace grow large, along with general MPI statistics (time elapsed, bytes transferred and number of calls), are accumulated using software counters. The minimum computing burst length to consider determines the granularity of the summarized trace, which is a trade-off between size and usefulness of the information presented to the analyst, enabling a good characterization of the execution at a reasonable size.

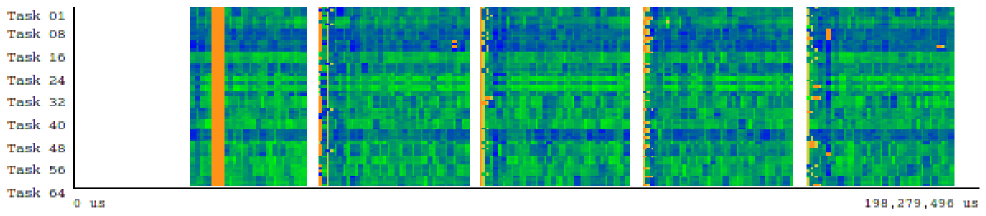
The strategy we follow is to combine both tracing modes and different levels of granularity so that a few representative iterations are traced in detail, and the remaining repetitive behaviors and non-periodic zones get partially summarized. In this way, time and space variability can still be studied across the whole execution while avoiding redundant data.

Figure 5.7 shows an example of this protocol applied to a full 3.5 minutes run of the NAS BT benchmark with 64 processes. The trace timelines represent the state of every process (Y-axis) over time (X-axis) with respect to different metrics.

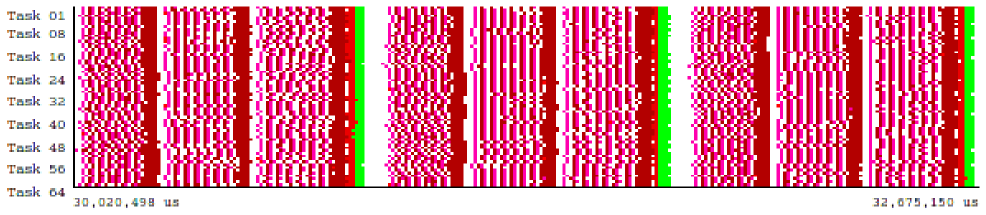
Figure 5.7a shows the different periods that were detected for a global view of the iterative structure of the application. Colors represent every different periodic behavior of the application: blue for the non-periodic region (occurs at the very beginning, due to initializations); red for the analysis time, which was computed every 30 seconds and took



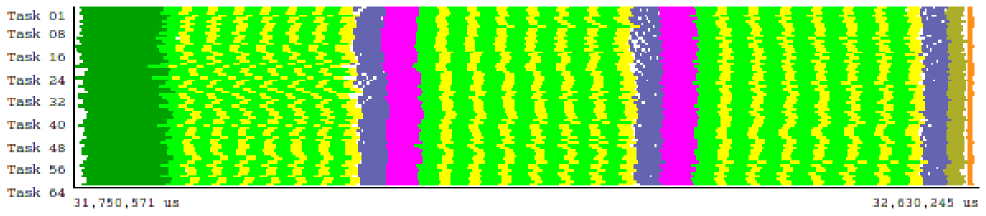
(a) Global periodic view for the whole run



(b) Phase profiler of *Instructions per cycle* for all periodic regions



(c) Sequence of MPI calls over three detailed iterations



(d) Internal computing bursts structure of a single iteration

Figure 5.7.: Increasing levels of detail stored in a single trace of the NAS BT benchmark

1.5s on average to complete; and green for the regions analyzed. Every analysis region comprises 40 iterations of the application, which executes a total of 200. The fact that all these regions are green means that the application presents a single periodic behavior that repeats across the whole execution.

The very first time this behavior is seen, the system selects a few representative iterations to be traced in detail. That includes function calls, hardware counters, call stack references, and inter-process communications, as shown in Figure 5.7c with the detailed sequence of MPI calls over three complete iterations of the application's main loop. In this way, every different pattern of behavior has a compact, but complete representation in the trace.

Then, for the remaining iterations within a periodic region that will not be traced in detail, we accumulate software counters for MPI statistics and performance counters, providing per-step phase profiler information. Figure 5.7b shows the average *instructions per cycle* per iteration of NAS BT over the entire run. In this case, the gradient from green to blue represents increasingly higher IPC values. Just having a quick look at it, one can easily identify a subset of processes that are systematically achieving higher performance.

Combining spectral analysis with clustering, we can also incorporate into the trace information about the structure of every computing burst for a given time region, which provides high details about the internal computing structure of the application. Figure 5.7d shows the distribution of the different computing clusters over a single iteration of the application, where three main sub-phases can be quickly identified.

And for the non-periodic regions, we can optionally store a summarized trace that filters out the less relevant computations. Including detailed tracing for these regions is usually not an option, as the final trace size can grow large and its utility is debatable. Generally speaking, a highly repetitive behavior is more interesting than an isolated effect, as any performance issue detected and optimized there will have a strong impact all over the execution. However, this does not imply that non-periodic effects are all irrelevant, so rather than discarding all this data, we present it partially summarized.

All these types of information, which range from the most general view of the application's iterative structure to the most specific details in an iteration, can be combined in a single trace that virtually has the same information that is comprised in a full trace, but at a minimum size. The automatically generated trace used in this example occupies 20Mb only, down from 1.5Gb for the full trace.

5.4. Experimental validation

The automatic analysis system presented has been tested with a variety of real applications and benchmarks in the Cray XT5 supercomputer *Jaguar*. The XT5 partition contains 18,688 compute nodes with dual hex-core AMD Opteron 2435 (Istanbul) processors running at 2.6GHz, for a total of 224,256 processing cores. The objective of these experiments is threefold:

- Validate that the system successfully identifies the iterative behavior of an application, and it is responsive to small fluctuations that might be present in the execution.

- Show that the automatic traces are much smaller compared to the full runs, and they can be effectively used to detect interesting performance issues.
- Evaluate the scalability of the system up to tens of thousands of cores.

The following sections present an in-depth analysis on two real applications to demonstrate how the proposed analysis contributes to obtaining manageable volumes of information that are useful for the analyst to understand how the application behaves.

5.4.1. Quality assessment of the resulting traces

As shown in the following examples, the application of this protocol results in automatically generated traces that attain size reductions of several orders of magnitude, sharply shrinking the amount of data compared to a full execution trace. However, such data reduction goes through an inevitable data loss, where isolated effects might get lost.

One question that arises then is whether a compact trace for a short time interval is a fair representative for a full trace, and if the analysis of the first would lead to the same conclusions as the use of the latter. In order to respond to this matter, we evaluate the trace quality loss in terms of the variability that is going to be lost because of focusing on just a few iterations. The objective is to ensure that the traced region does not miss relevant areas and represents well the overall execution. We do so by comparing our results with the performance data gathered from a full run. In particular, we have used a profiler to obtain several metrics of the functions of the program. The same metrics were computed for the trace interval that is automatically selected by our mechanism. In this way, we have verified that the values reported in both cases remain practically the same.

Table 5.1 compares the percentage of time spent in the most consuming MPI functions, both for the complete and selected time intervals of PEPC and PFloTran applications. As noted, differences are marginal (percentage differences are lower than 3%) with trace reductions greater than 90%. Overall, the same structure can be identified, so in all cases the trace segment proved to be a good representative for the rest of the execution.

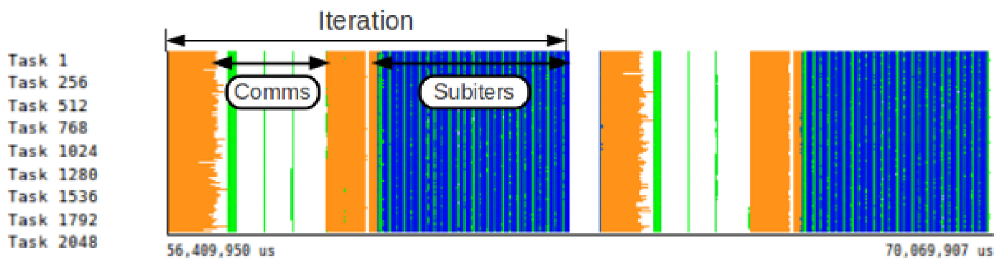
5.4.2. PEPC analysis

PEPC 1.0 [142] is a parallel tree-code for rapid computation of long-range Coulomb forces for large ensembles of charged particles. The execution was run with 2,048 processors on Jaguar. The analysis was triggered at a fixed time rate of 3 minutes, and took 4.5 seconds on average to complete. A full trace has an approximate size of 0.5 Tb for 30 minutes of execution. Instead, the tool produces an automatic trace that represents a single periodic behavior with two iterations traced in detail in roughly 5 Gb. Some tools may still present issues when handling such amount of data, but the important fact to remark is that the trace generated is 99% smaller.

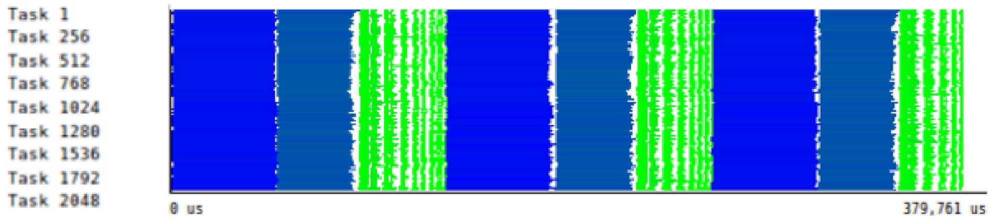
Figure 5.8a shows the structure for these two iterations, which alternates a sequence of computing and global communication phases. The computing phase is in turn divided into an iterative pattern that comprises two heavy computations followed by several short

	PEPC (1,024 tasks)				PFloTran (128 tasks)		
	Full trace	Smart trace	Diff %		Full trace	Smart trace	Diff %
Trace size	7.0 GB	700 MB	90.3%	Trace size	1.3 GB	93 MB	93.01%
Functions profile				Functions profile			
Computing	59.10%	61.54%	2.44%	Computing	9.90%	7.55%	2.35%
MPI_Alltoallv	14.71%	14.33%	0.38%	MPI_Waitany	26.69%	28.24%	1.55%
MPI_Alltoall	13.08%	11.48%	1.60%	MPI_Allreduce	24.65%	25.12%	0.47%
MPI_Allgather	9.06%	6.39%	2.67%	MPI_Start	23.09%	24.88%	1.79%
MPI_Barrier	2.48%	4.91%	2.43%	MPI_Startall	4.95%	5.33%	0.38%
MPI_Allgatherv	0.83%	0.88%	0.05%	MPI_Waitall	4.28%	4.59%	0.31%

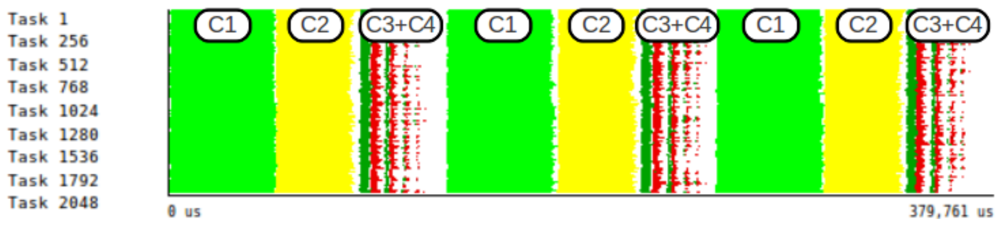
Table 5.1.: Comparison of a full vs. automatic trace



(a) The main iteration is comprised by two different phases



(b) Three sub-iterations inside the computing phase



(c) Distribution of clusters over the sub-iterations

Figure 5.8.: Analysis views for PEPC (2,048 tasks)

bursts, as can be seen in more detail in Figure 5.8b. In both timelines, the color gradient from green to blue represents increasingly larger computations.

We have clustered the computing bursts on this sample of three sub-iterations by their number of executed instructions and the achieved IPC. This results in all computations with similar characteristics to be grouped together, as shown in Figure 5.9. This scatter plot is to be interpreted as having four different types of computations, where clusters 1 and 2 happen to execute a high number of instructions and are the most time-consuming, which makes them the best candidates to focus the analysis on. The distribution of clusters over the timeline can be seen in Figure 5.8c. Clusters 1 and 2 correspond to the heavy computations on the sub-iteration, while clusters 3 and 4 correspond to the short computations between MPI calls.

A histogram of the computation durations correlated with the number of executed instructions is shown in Figure 5.10. This is read as the classic histogram rotated, where the processes are on the Y-axis and the values on the X-axis, with higher values on the right-most part of the histogram. Any structure other than a vertical line is reflecting dispersion, which is specially remarkable in cluster 2. The lower half of the histogram is shifted to the right, meaning the computations for the lower half of the processors are taking a 2% longer. The color gradient from green to blue represents an increasing number

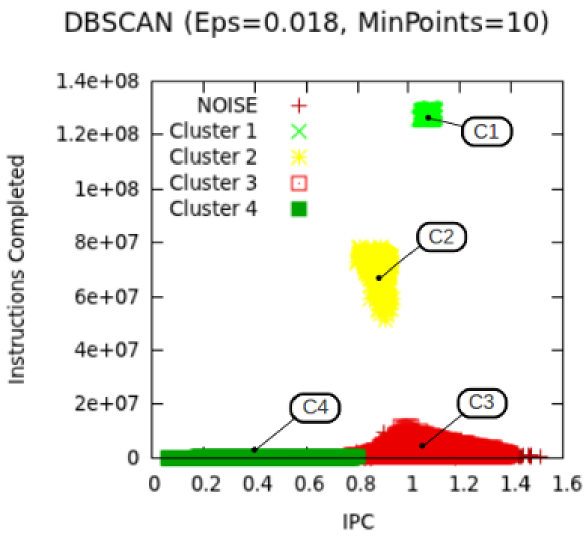


Figure 5.9.: Structure of the main computing regions of PEPC

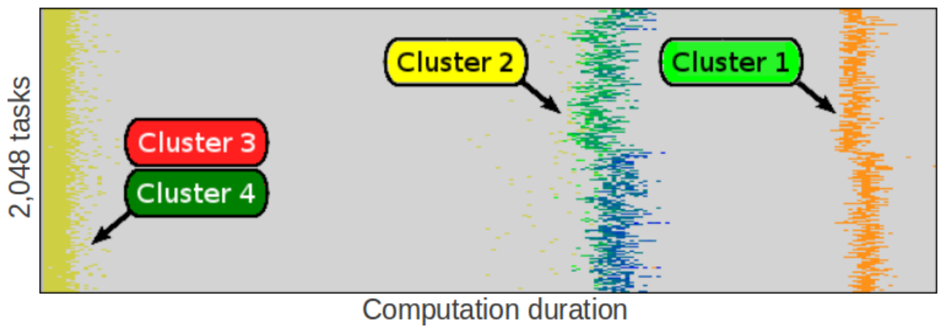
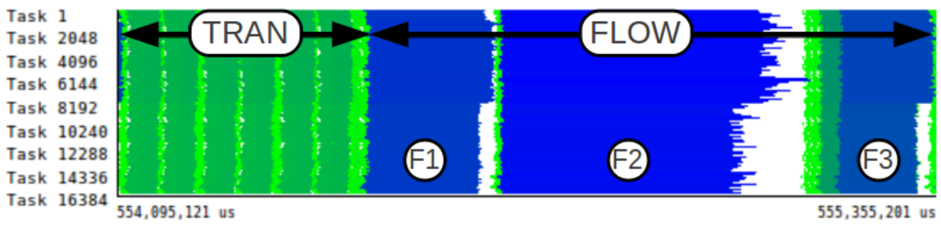


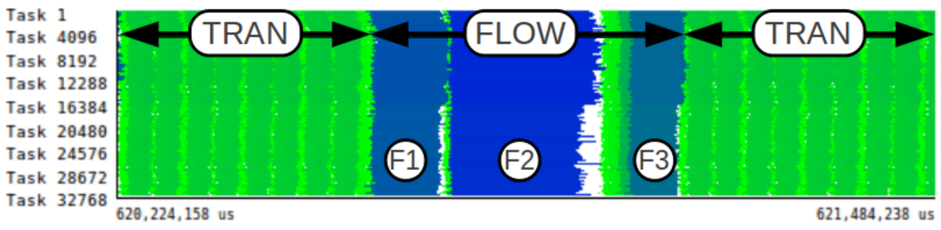
Figure 5.10.: Histogram correlating the duration of the computations and the executed instructions in PEPC. Longer computations (the more to the right part of the histogram) execute more instructions (darker gradient colors from green to blue, being orange the maximum value).

of executed instructions, being higher for these same processes. This reflects work imbalance in the application, that could also be inferred from the significant dispersion in the instruction axis in Figure 5.9.

The corresponding source code is the function *tree_walk*, between lines 160 and 378. The code within this region classifies a list of particles whether their interaction has to be computed or deferred, and the code flow is likely to differ depending on the process, being some case statements more compute intensive than the rest. This effect eventually causes communication delays in the next program phase. The imbalance in computations that belong to cluster 2 are absorbed in MPI_Alltoall collective operations that take place immediately after, where the late-arriving processes provoke a global stall. The impact of this imbalance can be measured by Paraver configurations as defined by Casas et al. in [143]. The loss in the *load balance efficiency* can be quantified up to a 16% of the total computing time for this region. So if the user gets to balance the dispersion of instructions,



(a) Time interval for a single iteration running with 16,384 tasks



(b) Same time interval doubling the number of tasks to 32,768

Figure 5.11.: Structure of a single iteration of PFloTran

these are the improvements they are expected to achieve.

All in all, while the spectral analysis of signals delimits which are the interesting regions to be traced, clustering analysis is used to characterize their internal structure. The combination of both techniques attains full coverage for the execution, capturing meaningful information for every relevant region to help the analyst getting complete understanding of the application behavior.

5.4.3. PFloTran analysis

PFloTran [144] is built on top of PETSc for modeling multi-phase, multi-component subsurface flow and reactive transport using massively parallel computers. In this experiment, we ran two large executions on Jaguar with 16,384 and 32,768 cores, with the objective of studying how well the application scales. The full traces for these runs approximately size 28 and 78 Tb, while the automatic traces resulted in 2.6 and 7.3 Gb respectively, comprising three iterations traced in full detail each. In both cases, the extra analysis time did not exceed the 1% of the total execution time.

PFloTran performs an iteration-based simulation with two main stages: flow and transport. Figure 5.11a shows the execution timeline for a single iteration of the 16k cores test case, where both phases intersperse. The flow stage comprises three intensive computations separated by large global communications where data is exchanged between all processes. The transport phase comprises, in turn, iterative short computations also separated by global synchronizations. If the application scales well, when doubling the number of processes we expect it to run twice as fast, and so in the same time interval that a single iteration took to execute, it now should execute two. Figure 5.11b shows the same time scale for the 32k cores experiment. In this interval, it now gets to execute one and a half iteration only, which is indicative of some phases that are not linearly scaling. Visually, it is easy to see that flow gets significantly reduced, while transport lasts the same. However,

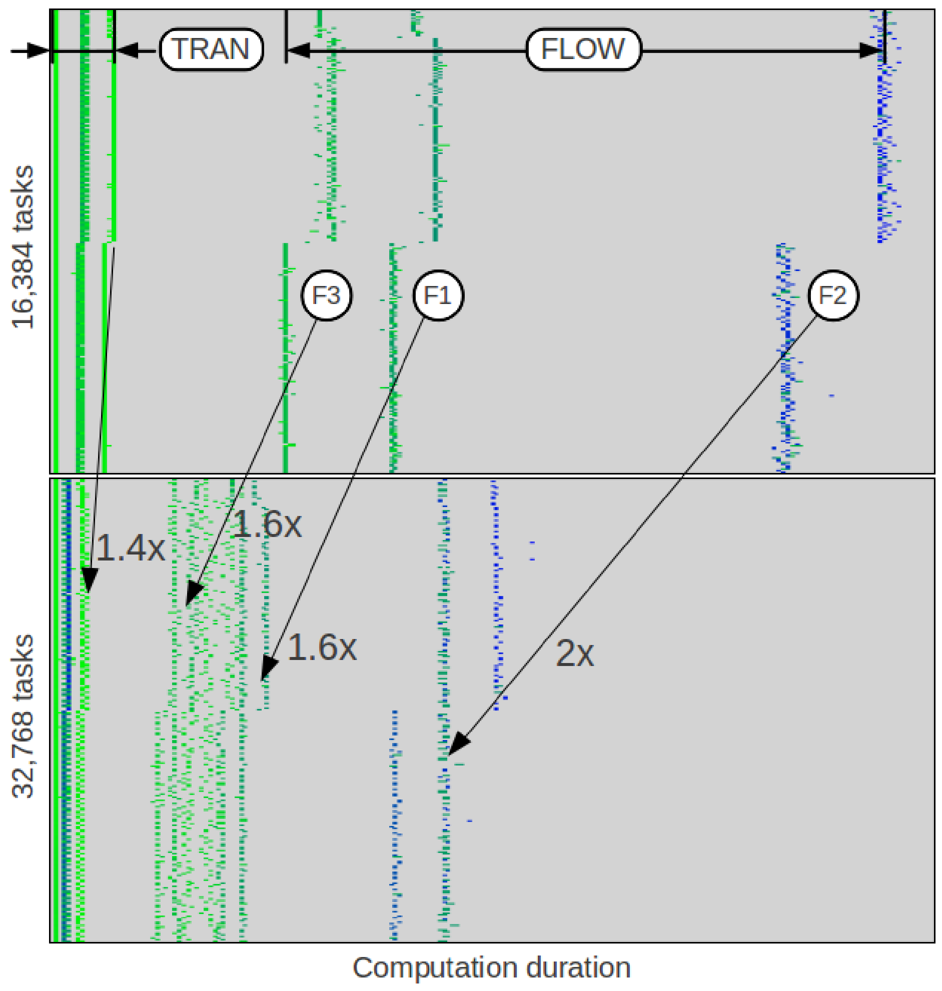


Figure 5.12.: Histograms of the duration of the computations in PFloTran when using 16,384 (top) and 32,768 (bottom) MPI tasks. The rate at which data moves to the left indicates how well the different parts of the program scale.

it is not that the computations in this stage do not scale, but it is now performing more sub-iterations, which is dependant on how quick the solver converges.

For a detailed study of the scalability of the different stages we can compare the histograms for the computations durations, as shown in Figure 5.12. Here we can easily see that all phases scale, but at a different rate. For the three sub-phases (named F1, F2 and F3) that comprise the flow stage, F2 scales linearly being reduced by a 50%, while F1 and F3 achieve a smaller 1.6 speed-up. The transport stage scores the worst improvement, with a 1.4 speed-up only. Also, it is easy to notice that the dispersion between processors increases in the largest execution, with new trends of behavior (more columns) appearing for the different stages. The fact that the second half of processors are all skewed to the left reflects a load-balance problem, as multiple computations with variable performance are taking place simultaneously.

Starting with the region that performs the worst, we can then correlate the duration

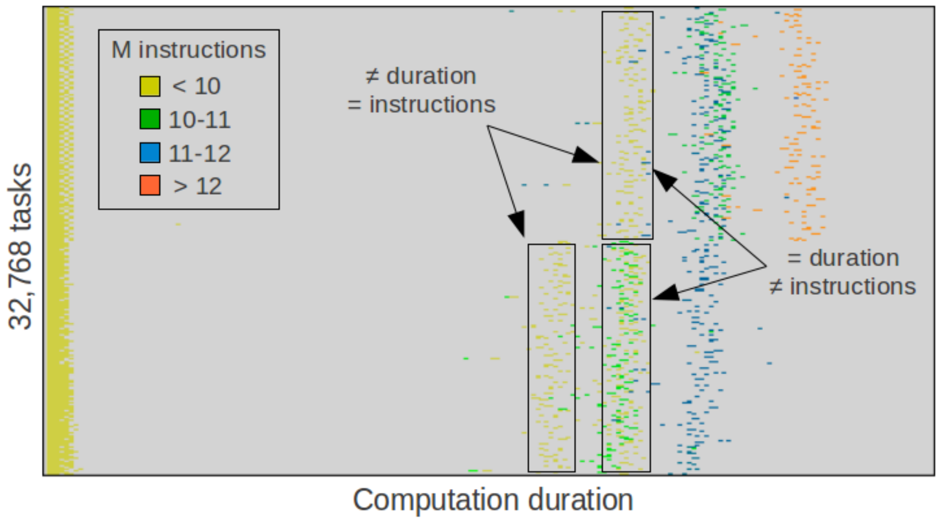


Figure 5.13.: Histogram correlating the duration of the computations (columns) and the executed instructions (color gradient) in PFloTran showing imbalances between processes (rows)

of the computations with the number of instructions executed (see Figure 5.13), so as to check whether the reason for the bad scaling is due to problems in the work distribution. The more to the right a point appears in the histogram, the longer is the duration of the computation that represents. The color gradient from green to blue indicates an increasing number of executed instructions per computation. There is a direct relation between both variables showing that the longer a computation is taking, the higher number of instructions is executing. Furthermore, it so happens that there are computations that last the same but are executing different numbers of instructions, which is a sign of work imbalance. Contrariwise, we can also observe computations performing the same number of instructions, but taking variable times to complete, which also evinces IPC imbalance.

The flow phases F1 and F3 also present work distribution issues. The average number of instructions executed per computation in these regions is 368M, which gets reduced by a 37.5% to 230M, rather than the expected half. A final recommendation that could be given to the user is to study the load-balancing characteristics of these regions of the code.

Applying our analysis mechanism, we renounce a full trace for a meaningful representative, where we are still able to quickly identify microscopic effects that have a large global impact. And when these are optimized, not only the single iteration where they were seen will improve, but all along the same periodic region. So this leads to the remarkable conclusion that there is much room for performance improvement from the analysis of a very compact trace.

5.5. Related work

The application of periodicity detection techniques in real-time analysis was explored by Freitag et al. [145]. The Dynamic Periodicity Detector uses the stream of OpenMP parallel functions that are being executed to identify periodic patterns of behavior in the current data stream. Once enough periods have been recorded, the tracing facility stops collecting events, but keeps on feeding the DPD with the stream of data until a change in the program behavior is detected, and trace data starts to be collected again.

Also based on the detection of repetitive sequences we find ScalaTrace [77]. This tool is targeted at generating scalable MPI event traces by introducing compression techniques that are capable of extracting the application's communication structure. Repetitive sequences of MPI events are merged into a single entity called Regular Section Descriptor, which enable the representation of traces with near constant size.

Mohror et al [146] investigate pattern-based methods for reducing traces. They present an approach for trace segmentation, that consists in collapsing sequences of segments (i.e. loop iterations) with similar behavior and retaining a representative. To decide the similarity of segments, the authors evaluate several methods based on distances and iterations.

While the previous approaches are based on a sequence of events, ours is based on signals that are not bound to the parallel programming model and do not require a repetitive sequence of specific events. Instead, periodic behavior is detected from any performance metric, and thus it is applicable in a wider range of scenarios. Also, we perform a global analysis of a selected signal that characterizes the periodicity of the application, so as to determine a trace interval to study time and space variations at the application level.

A. Knüpfer et al [147] apply periodicity detection at the visualization phase. Their objective is to display less data but more information within the typical time-line views. To this end, they collapse all repetition patterns of function calls, and since that hides too much information, the patterns can be decomposed interactively to show the inner details. Our approach brings the same concept of reducing the amount of information presented to the analyst to the data collection phase, because performance tools can no longer cope with the ever-growing amounts of gathered data.

The Paradyn Parallel Performance Tools project [62, 84] paved the way for intelligent selection of performance data. It provides a profile-based on-line bottleneck search algorithm that automatically looks for a set of known performance problems. This is an iterative process of formulating hypotheses and inspecting related performance data in an attempt to confirm or reject them. Periscope [43] proposes a very similar on-line distributed analysis, having agents autonomously searching for performance problems of a small subset of processes. The common aspect that our approach shares with these tools is that we similarly inspect and filter out the data that is being collected. However, we perform the analysis over the detailed stream of events rather than summarized profiles. Also, our objective is not to report common performance bottlenecks but a detailed trace, whose analysis may lead to the discovery of new and unexpected patterns that model an inefficient behavior.

Our work on intelligent selection of the traced information was first approached using cluster analysis [7]. The objective was to characterize the structure of the computing regions according to similarities in hardware counters values. The analysis was repeated periodically and the results were compared. If the resulting clusters happened to be the same, that was indicative of the application being in an iterative phase, and then that region was selected and traced as representative of the execution. This mechanism has poor control over the traced period, while with the approach presented here we gain precision as we can control the exact number of traced periods.

5.6. Closing remarks

In this chapter we have presented an analysis protocol running on top of the on-line analysis framework that leverages signal processing techniques to identify the periodic structure of a parallel application. Then for each different pattern of behavior detected we automatically select a small representative to be emitted into the trace. Trace data is stored at different levels of granularity, ranging from a general view of the periodic structure of the application, to every single detail for an specific time interval. This brings the concept of reducing the amount of information presented to the analyst to the data collection phase, because performance tools can no longer cope with the ever-growing amounts of data.

For every period, a few representative iterations are traced in full detail. For the remaining iterations, we store accumulated software counters (i.e. MPI statistics, performance counters), providing the information equivalent to a phase-profiler for the whole run. In combination with clustering (refer to Chapter 4), we can also incorporate into the trace the internal structure of every computing burst, and with it, the full power of cluster-based analysis that we have previously discussed. And for the non-periodic regions, rather than discarding all data we optionally store a summarized trace that only contains information for the most significant computations. All together, this provides both general and detailed information that is useful for the analyst to get complete understanding of the application behavior.

The resulting trace fully describes the execution, and enables the study in detail of time and space variances across the whole run, while avoiding redundant data. Since it comprises small samples of the application iterative behavior, we achieve size reductions of several orders of magnitude compared to a full trace. In general, the analysis of such a compact trace will successfully lead to important performance improvements, as any persistent performance flaw that is solved for a single iteration, will have a positive impact throughout the execution.

This approach can be seen as a first step in the analysis methodology, obtaining the minimum amount of information that is relevant to the analysis. Other analysis mechanisms could then be applied, not having to deal with large amounts of data and all the difficulties that are inherently associated.

We have demonstrated the usefulness of this approach to extend the traces scalability up to tens of thousands of cores and beyond. But as we keep moving forward to larger-scale scenarios and the number of cores increase, we will eventually need additional reductions. Knowing the structure of the application (where the iterations are, their size, etc.), we can predict the size of the resulting trace before writing it, and take automatic decisions on whether to keep the data in detail, discard or further summarize it, towards the maximum trace precision that is still manageable.

Part III.

New Techniques to Enhance the Analysis Task

Chapter 6

A massively parallel version of the DBSCAN cluster algorithm to improve the scalability of on-line analysis

We have presented density-based cluster analysis as a useful technique for the performance analysis of parallel applications. By identifying similarities in the computing bursts of the application, it is possible to expose their fine-grain structure and detect relevant regions of the execution, and characterize their performance behavior to detect imbalances and other performance problems. Nevertheless, clustering algorithms are often computationally expensive, and since their cost heavily depends on the size of the input, they scale very poorly with large data sets.

In this chapter we present a distributed scheme to parallelize clustering algorithms. We apply this technique to parallelize one of the most common clustering algorithms in scientific literature, the density-based spatial clustering of applications with noise (DBSCAN). In the context of performance analysis, we have chosen to relax some of the axioms that determine the cluster formation in DBSCAN on account of improving the quality of the results due to certain characteristics of performance data. For this reason, our implementation can not be assumed to be compliant with the original DBSCAN formulation, but in the described scenario produces oustanding results being able to scale up to tens of thousands of cores and analyze data sets three orders of magnitude bigger than the sequential DBSCAN in equivalent execution time. Moreover, our approach is general enough to be applicable to other cluster algorithms families that can also take advantage of a distributed structure to easily parallelize their computation.

This work extends previous research presented in Chapter 4 in two main directions. First, we increase the scalability of the clustering process to be applied on-line for the analysis of large scale applications. And second, by being able to process larger data sets in less time, the quality and precision of the analysis improves.

6.1. Background and motivation

With analysis tools extracting performance measurements from a parallel application every few milliseconds, the data generation rate for a large-scale execution quickly grows to millions of events per second. Sifting through huge amounts of data to uncover relevant performance issues makes the task of analysis for finding bottlenecks and potential optimizations not a trivial pursuit.

Clustering has become an important technique for the analysis of high-performance computing applications. In brief, clustering is the task of assigning data into groups according to a similarity measure in order to reduce the volume of data by selecting representative observations that serve as prototypes of the clusters. In [108, 110, 148] the authors applied K-means to find similar behavior among all the processes involved in a parallel execution.

These approaches are very effective in reducing the sheer amount of data, but at the expense of variability. Time and space variabilities are important in performance analysis so as to detect microscopic effects that may have a large global impact, which are often masked out when looking just at a few representatives. In [114], the authors follow a different approach. Instead of searching for representative processes, they apply DBSCAN [115] to cluster all the CPU bursts in the execution (i.e. the computing regions between MPI or OpenMP synchronization primitives) with respect to hardware counters metrics. In this way, all the details on how the variability distributes over time and processes are kept, enabling a precise detection of the computations' structure. This information was demonstrated to be useful to better understand the application behavior and serves as a starting point to direct a deeper analysis.

However, DBSCAN shows scalability issues basically motivated by its algorithmic cost, which is quadratic in the worst case. In Chapter 4, we further developed the previous method to build an on-line analysis tool and overcame the scalability limits using a sample-based strategy. There, each parallel process captures its own performance measurements, samples a subset of its data, and transfers its samples to a single node where the clustering is performed centrally with an aggregate of the samples from all tasks. Then, the rest of the data is classified in parallel using a nearest-neighbor algorithm. Withal, when moving to the large-scale the number of samples increases very fast, resulting in slow response times that are not reasonable in real-time; and further reducing the number of samples would degrade the resulting clusters up to the point where they are bad prototypes of the overall data and the amount of noise grows large.

This contribution extends the work presented in previous Chapter 4 by introducing a new distributed implementation of the DBSCAN cluster algorithm. Our objectives are fourfold: first, to design a general algorithm that can be applied to any data source, even though our study is tailored to the on-line analysis of performance data; second, to further improve the scalability of the DBSCAN algorithm to support large-scale experiments; third, to support both post-mortem analysis from prerecorded trace data, and on-line analysis where every process clusters its own performance data; and fourth, not to perform an explicit preprocessing phase to make optimal data partitions and redistributions.

To this end, we present a tree-based algorithm that takes advantage of the inherent data distribution of a parallel application to perform local clusterings with partial data and hierarchilly combine the results. Having no data redistribution phase, the main difficulty of the algorithm becomes to expand the search for points that belong to the cluster across the data stored in all other processes. In order to solve this algorithmic problem efficiently, we transform the domain of the problem from checking the Euclidean distances between pairs of points into modelling the partial clusters as simple geometries, and checking for their collisions in the clustering space. This approach approximates very well the original DBSCAN algorithm with marginal differences below 2%.

6.2. Distributed scheme to parallelize clustering algorithms

This section describes in detail our parallel implementation of DBSCAN, taking advantage of the on-line analysis infrastructure presented in Chapter 3. Figure 6.1 shows the overall structure of the algorithm.

The process starts with a local clustering performed in the back-end nodes of the tree, so that each one will process the data extracted from one or more application's tasks (steps 1 and 2). Local clustering results are then aggregated in the intermediate nodes of the tree (3). In order not to flood the network with data, the points that shape the clusters are not transmitted per se, but they are simplified into a minimal model that represents the cluster using convex hulls [149]. The partial models are combined upon their way to the front-end, who ends up receiving a single global representation of the state of all clusters (4). Then, the global clustering model is broadcasted back to the back-ends (5), and their local data is independently classified with a nearest-neighbor algorithm (6), using the global model as a reference. In this way, we achieve a highly scalable alternative to centralize and cluster the whole data set together.

The following Sections describe the phases above in more detail.

6.2.1. Data partitions

In the performance analysis scenario, where every process of the application is producing performance measurements, there is an intrinsic distribution of the data right from the start. Each process could cluster its own data removing the need to create partitions and redistribute it. How the data is distributed has a significant impact in the performance of a parallel DBSCAN implementation because of the costs associated with looking for candidate points to be part of a cluster when they are stored in a different process. For this reason, algorithms in the literature have paid careful attention to how to better balance the work among the processes. But finding an optimal partition is not a trivial task, and so a common approach is to simply distribute the whole data set equally [124]. When the data has a physical interpretation (e.g. geographical data), it is desirable to make specialized partitions to keep the dense areas close, minimizing the messages between processes [125].

Nevertheless, parallel applications broadly feature several characteristics that enable us to make certain assumptions about the data distribution. Typically, these programs present

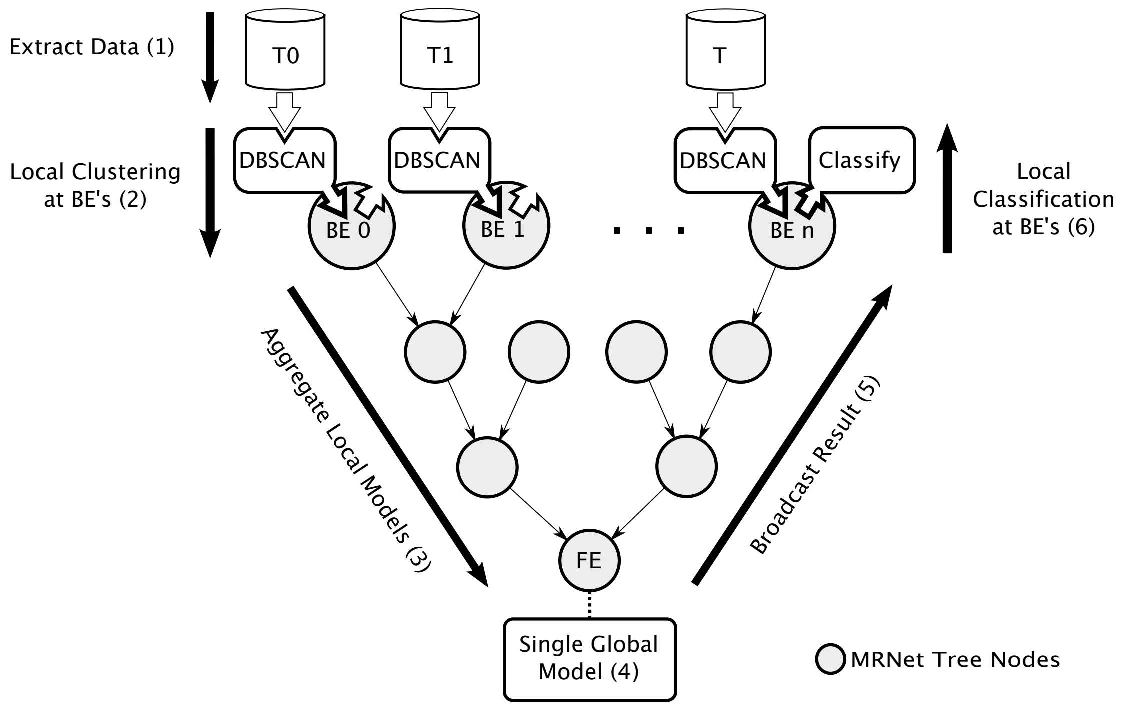


Figure 6.1.: Distributed tree-based DBSCAN cluster algorithm design

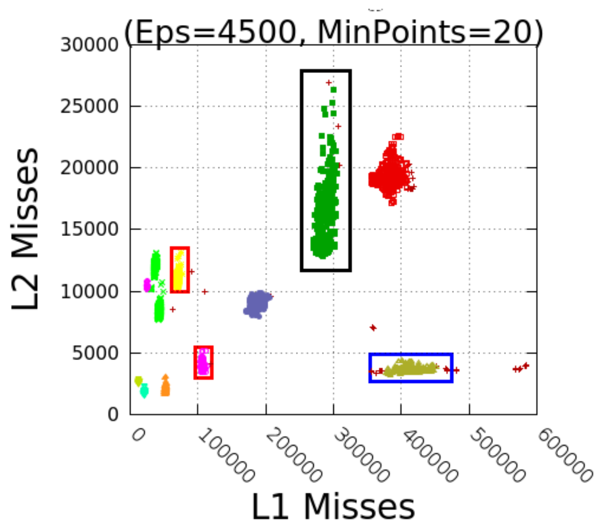


Figure 6.2.: DBSCAN of complete data from all 128 tasks of CPMD

strong iterative patterns and marked SPMD models [150]. This is to say, if a process reaches a certain degree of performance for a given region of the code, it will generally achieve very similar performance whenever it is executed again. On the other hand, different regions of code may exhibit very different performance. But whatever is the shape of the data for one process, it will be very similar for the others if the application is SPMD. Representing the performance measurements into the space would ideally result in very compact and clearly separated clouds of points, as shown in Figure 6.3. The scatter plots show the result of applying DBSCAN to the performance data from tasks number 1, 64 and 128 of CPMD. It is easy to see that the structure is very similar for the different tasks. Figure 6.2 shows the resulting clustering when the data from all tasks is aggregated. Now some clusters stretch vertically (e.g. black and red boxes) and horizontally (e.g. blue box), denoting performance variabilities due to work and time imbalances between the different processes of the application, respectively.

Since all processes participate to the global clustering with a partial subset of the data, an optimal redistribution that would keep the dense areas together would then require huge data movements between all processes, incurring in a cost possibly higher than the clustering algorithm itself. This is the reason behind the decision of designing an algorithm without an explicit partitioning phase.

6.2.2. Local clustering

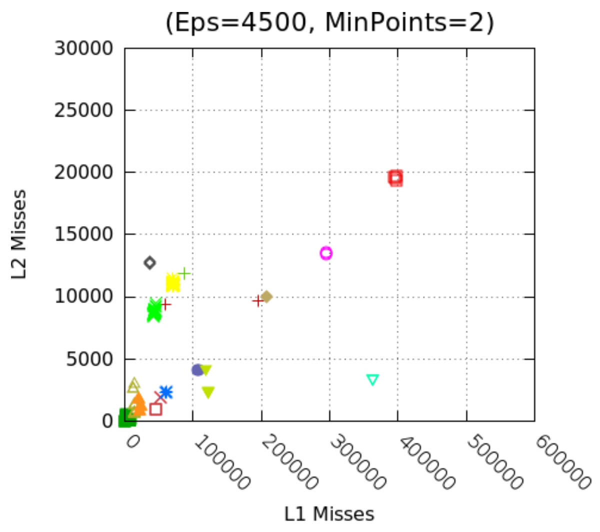
Each back-end node in the tree processes the performance data produced by one or more tasks from the parallel application, and runs a regular sequential DBSCAN with its own data subset. In particular, we assign to each back-end the same number of application's tasks, under the assumption that most codes are SPMD and so the volume of data per back-end will be well-balanced. For non-SPMD codes this could lead to a certain work imbalance, yet these are seldom seen and the algorithm is still applicable. For the maximum degree of distribution, as many back-end processes as application tasks are created, so that each task's data is independently clustered.

The resulting local clusterings represent the computational structure of one or a few tasks each, and these partial results have to be combined for a global view of the application structure.

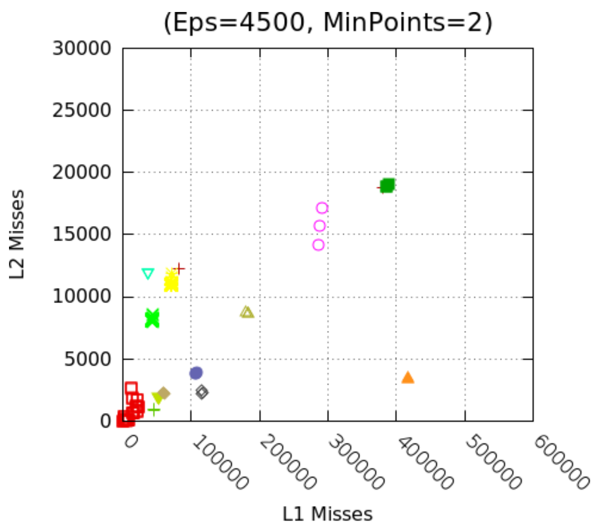
6.2.3. Global clustering

Combining the partial results requires to define an efficient representation of the clustering, so as not to flood the network with an excess of data. In this sense, putting together all clusters points would scale very poorly. Instead, we define a simple geometrical model that minimally describes the clusters using convex hulls, as implemented in the Computational Geometry Algorithms Library [151].

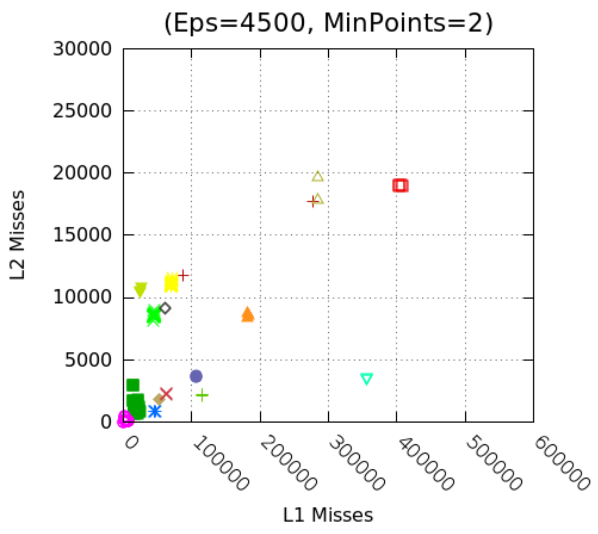
The convex hull for a set of points P is the minimal convex polygon with vertices in P containing all points in the set. This can be seen as an elastic band stretched open to encompass the given cluster; when released, it will assume the shape of the convex hull, as shown in Figure 6.4. The number of points required to represent the convex hull is orders



(a) DBSCAN on Task 1 data



(b) DBSCAN on Task 64 data



(c) DBSCAN on Task 128 data

Figure 6.3.: DBSCAN of partial data from tasks 1, 64 and 128 of CPMD

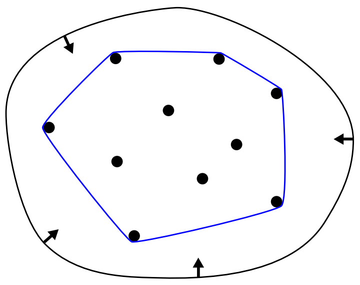


Figure 6.4.: Convex hull of a set of points

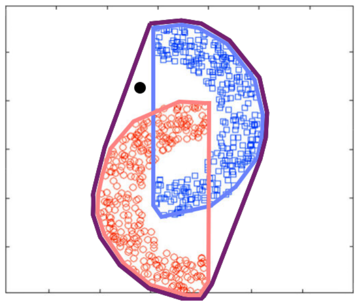


Figure 6.5.: Intersection of convex hull cluster models

of magnitude smaller than the number of points for the whole cluster. Using this representation, we can easily transfer the local clusterings through the network and aggregate the results directly operating with their models.

The process of aggregation is done at the intermediate nodes of the reduction tree as data goes up to the front-end. At every node, all convex hulls received from the children nodes are combined into a new partial model that describes the structure of that sub-tree, which is then retransmitted to the upper tree levels. Whenever two convex hulls intersect, or any pair of their vertices are at distance lower than *Eps* (see DBSCAN parameters in Section [115]), these are merged into a new convex hull that embraces both. Figure 6.5 shows an example of the merging phase. An intermediate node receives the models that describe the red and blue clusters from different back-ends. Since the polygons intersect, a new convex hull that envelopes both is created (purple), and only the newly created model will be passed to the following node.

Those polygons that do not intersect with any other are just bypassed to the next levels of the tree. In this way, data from different tasks with the same type of performance structure is combined into a single cluster representation, keeping the system scalable in terms of the amount of information to handle. The previous example also serves to illustrate why our implementation is an approximation to the original DBSCAN algorithm. If the data presents concave shapes as in Figure 6.5, modelling the cluster with a convex hull makes the area of the cluster bigger. Any points falling far from dense areas but inside the polygon area, like the black dot in the example, would be taken as part of the cluster, while DBSCAN could detect them as noise or even as a new cluster if they meet the necessary density and distance conditions. This limitation imposed by the use of convex hulls can be easily circumvented by modelling the clusters with a different geometry, for example, medial axis [152], Voronoi diagrams [153] or straight skeletons [154]. However, we had no need to follow this direction because we learnt from empirical experience that performance data from parallel applications hardly, if ever, presents these problematic shapes.

Also, merging two convex hulls just because they intersect does not take into account the density of points around the intersection area. This might lead to merge clusters that the original DBSCAN would not expand because the density of a given point inside the convex hull is not high enough. To deal with this issue, our model representation also stores the neighborhood density around the vertices of the convex hull. When two polygons intersect, our algorithm assumes that the density in the intersection area is higher than *MinPts*, and we always merge the clusters. However, in the case where two hulls do not intersect, but any of their vertices are under *Eps* distance, we check for the average density of the vertices to be above *MinPts*, assuming a homogeneous distribution of points around the vertices. While this assumption is not necessarily true, we will later see in the validation Section 6.3 that this relaxation generally leads to better results in our context of performance analysis.

6.2.4. Local classification

When the merged clusters models reach the front-end node of the network, they have been reduced to a global model that represents the structure of all tasks of the application. This information is then broadcasted back to the leaves, so that every local task is able to see the global clustering state of the application.

Now local data can be independently classified using the global clustering model as a reference. The process of classification turns to be very easy and non-demaning in terms of computing power, as for every point we only have to check whether they fall inside of any of the polygons formed by the global convex hulls. In the end, the result is equivalent to have centralized all data and performed a single cluster analysis, but with a scalable approach that overcomes the limitations imposed by this type of analysis, where the time and memory required grows exponentially to the input data.

6.2.5. Noise management

One of the parameters that determines the quality of the resulting clustering in DBSCAN is the minimum number of points (MinPts) required to form a cluster. But when the data is distributed among many back-ends, the density of the data in each of the local clusterings becomes proportionally lower to the number of processes. Because of this, clusters that would appear when all the data is centralized, might not form a cluster in the local clusterings because the density is too low and detected as noise. To deal with the noise, our algorithm performs two operations.

First, the value for MinPts is dynamically adjusted depending on the level of the reduction tree. At the bottom level, where the local clusterings are performed, MinPts is fixed to a minimum value of 3, because this is the minimum number of vertices needed to build the convex hull model of a cluster. In the intermediate levels of the tree, MinPts is updated to be the value set by the user divided by the number of sibling nodes in the current tree depth. This is made under the assumption that if the application is SPMD, all processes contribute to each of the clusters with at least one point, and so each SPMD cluster will have at least as many points as the number of processes. In this way, the value of MinPts increases as we move to upper levels in the tree. At the root-level, MinPts is fixed to be the value set by the user.

Second, after running the local DBSCAN in each of the processes, the remaining points that were detected as noise are also aggregated through the reduction tree, and we repeat the clustering step with the noise points from all the children only. If new clusters appear from clustering the noise, we also build their convex hull models and combine them with the rest of the models. This process is streamlined: while a given level of the tree is clustering the aggregated noise data from their children, the level below is building the clusters models and merging them, so as to keep all nodes in the tree busy. By the time the data arrives to the root level of the tree, the points that are still considered noise are those that the basic implementation of DBSCAN would also consider as noise with all the data centralized.

6.2.6. Time complexity analysis of the algorithm

The basic DBSCAN visits all the data points, possibly multiple times as they can be candidates to different clusters. The time complexity of the algorithm is mostly dominated by the cost of the neighborhood searches, which are executed exactly once for each data point. The general complexity of the neighborhood search is $O(n^2)$ (i.e. for each point, find all others that are close enough). If an indexing structure that executes the neighborhood query in logarithmic time is used (e.g. an R-tree), the overall complexity of the algorithm is reduced to $O(n\ log\ n)$.

In our parallel implementation, the DBSCAN algorithm is split in 3 phases: the local clustering phase, the merge phase, and the local classification phase. Being n the number of data points, P the number of parallel processes, h the height of the reduction tree, f the fan-in of the tree, and c_{max} the maximum number of clusters detected by a single process in the local clustering phase, the asymptotic complexity of the algorithm has three addends:

$$O\left(\frac{n^2}{P}\right) + O\left(\sum_{h=1}^{\lceil log_f P \rceil} (c_{max} \cdot f^{(h-1)})^f\right) + O\left(\frac{n\ log\ n}{P}\right) \tag{6.1}$$

The first addend corresponds to the cost of the local clustering phase. Since each back-end runs a basic DBSCAN with a subset of the data, the complexity is the same of the DBSCAN algorithm (quadratic) divided by the number of parallel processes.

The second addend corresponds to the cost of the merge phase, which includes the cost of intersecting all children clusters at each node of the tree. This is an exponential operation, but being the number of clusters much smaller that the total volume of points (i.e. $c_{max} << n$), the cost of this operation will be exponentially smaller than clustering all the data points. Worst-case scenario, the local clusters never intersect, and so the next level of the tree will have an increasing number of clusters to intersect, bounded by $(c_{max} \cdot f^{(h-1)})^f$.

The third addend corresponds to the classification phase, where each back-end runs a k-nearest neighbor classification algorithm on their local subset of points, so the complexity is the same as the classification algorithm (quasilinear) divided by the number of processes.

In the average case, where the local DBSCAN uses an indexing structure, the cost of the local clustering phase gets reduced to quasilinear. In the merging phase, if the clusters intersect, each level of the tree processes a constant amount of clusters. Then, the average cost of the algorithm can be reduced to:

$$O\left(\frac{n\ log\ n}{P}\right) + O\left(\lceil log_f P\rceil \cdot c^f_{max}\right) + O\left(\frac{n\ log\ n}{P}\right) \tag{6.2}$$

6.3. Experimental validation

In this section we measure the efficiency and validate the results of our tree-based parallel implementation of the DBSCAN algorithm, contrasting them against a basic sequential implementation, and a sampling-based alternative. We compare the results from the three implementations and discuss on their accuracy.

The experiments were performed on Blue Waters, a Cray XE6/XK7 system consisting of more than 22,500 XE6 compute nodes (each containing two AMD Interlagos processors) augmented by more than 4200 XK7 compute nodes (each containing one AMD Interlagos processor and one NVIDIA GK110 Kepler accelerator) in a single Gemini interconnection fabric [155].

6.3.1. Topological analysis

As discussed in Section 6.2.6, the time complexity of the algorithm depends on three components: the cost of the local clustering phase, the cost of the merge phase, and the cost of the classification phase.

The cost the clustering and the classification phases heavily depends on the number of back-ends used, and so this setting will directly impact the efficiency of these two phases. Additionally, the cost of the merge phase also depends on the number intermediate processes used to build the reduction tree, as well as the shape of this tree. The topology of the tree is determined by the number of back-ends and a fan-in. The fan-in specifies how many lower-level nodes a higher-level node serves. A high fan-in requires less extra computing resources and results in flat trees; whereas a low fan-in results in tall trees with more nodes that can operate in parallel, at the cost of further resources.

In this Section we study the impact of both parameters -the number of back-ends and the tree fan-in- in the algorithm performance, and draw some conclusions on how to apply the most convenient settings. To this end, we cluster the performance data extracted from a run of the Nekbone mini-app using 8192 cores. Nekbone is a Thermal Hydraulics proxy application [156] that allows users to study the computationally intense linear solvers that account for a large percentage of the more intricate Nek5000 software. In this experiment we try different number of back-ends to cluster the data, doubling them from 2 to 1024, to see the impact on the clustering and classification phases, which are directly influenced by this parameter. Then, for each fixed setting of back-ends we try different fan-ins, from 2 to the number of back-ends, to see what is the effect of having a flatter or taller tree in the merge phase. All the possible combinations of these two settings result in 55 different experiments, and for each one, we plot the computing time required to complete each of the three phases of the algorithm in Figure 6.6. The major divisions in the X-axis correspond to the number of back-ends, the minor divisions to the selected fan-in, and the Y-axis shows time in microseconds. Please note the scale in the Y-axis is logarithmic to improve the visualization of the graph.

The blue series represents the computing time for the clustering phase. With every increase of the number of back-ends, the clustering time drops. This is an expected result, because DBSCAN complexity depends on the input size. Given that we are increasing the number of processes exponentially, the number of points per back-end decreases also exponentially, and so does the computing time. With a fixed number of back-ends, the blue line remains constant, meaning that the fan-in does not impact this phase of the algorithm.

Analogously, the green series represents the computing time for the classification phase. Increasing the number of back-ends reduces classification costs, as the amount of data to classify gets distributed among the back-ends. In absolute values, the classification time is always very small compared to the other two phases.

The red series represents the computing time for the merge phase. The overall progression of this phase exhibits a slow increase, meaning that this part of the algorithm becomes more expensive when there are more processes involved. At small core counts, the merge cost is orders of magnitude cheaper that the clustering cost. As we increase the number of cores, the clustering cost drops very quickly, while the merge cost keeps increasing slowly. When using 256 back-ends, the clustering and the merge times cross. Beyond this point, the clustering time becomes so fast (below 1 second) that is almost negligible, and adding more processes does not really improve the results, because the total time becomes bounded by the merge time.

This effect can be seen more clearly in Figure 6.7. Graph 6.7a shows the total time of the algorithm (aggregated sum of the clustering, merge and classification times) and the total amount of resources used, with respect to the different settings for the number of back-ends and fan-in. As the reader can see, the total time keeps decreasing until the 256-2 case. Beyond this point, the 512-2 and 1024-2 configurations are still better (about 50% faster), but they require 2 and 4 times more resources for a very slight overall improvement of about 1 second in absolute numbers, so the trade-off between performance and resources becomes very unbalanced and all the extra resources are probably not worth it. Figure 6.7b illustrates how the efficiency of the algorithm (achieved speedup divided by the ideal) decreases in the right part of the plot, when the highest core counts are used.

The point where the clustering and merge lines cross is basically determined by the size of the data and the number of clusters detected. When the number of points per back-end is around one thousand, the clustering response time becomes practically immediate. However, the number of clusters detected does not vary with the number of back-ends. In our scenario, applying clustering to the performance data of a parallel application typically results in less than 10 relevant clusters detected. When the back-ends process so few points, the average merge time for such amount of clusters starts to exceed the clustering time. The

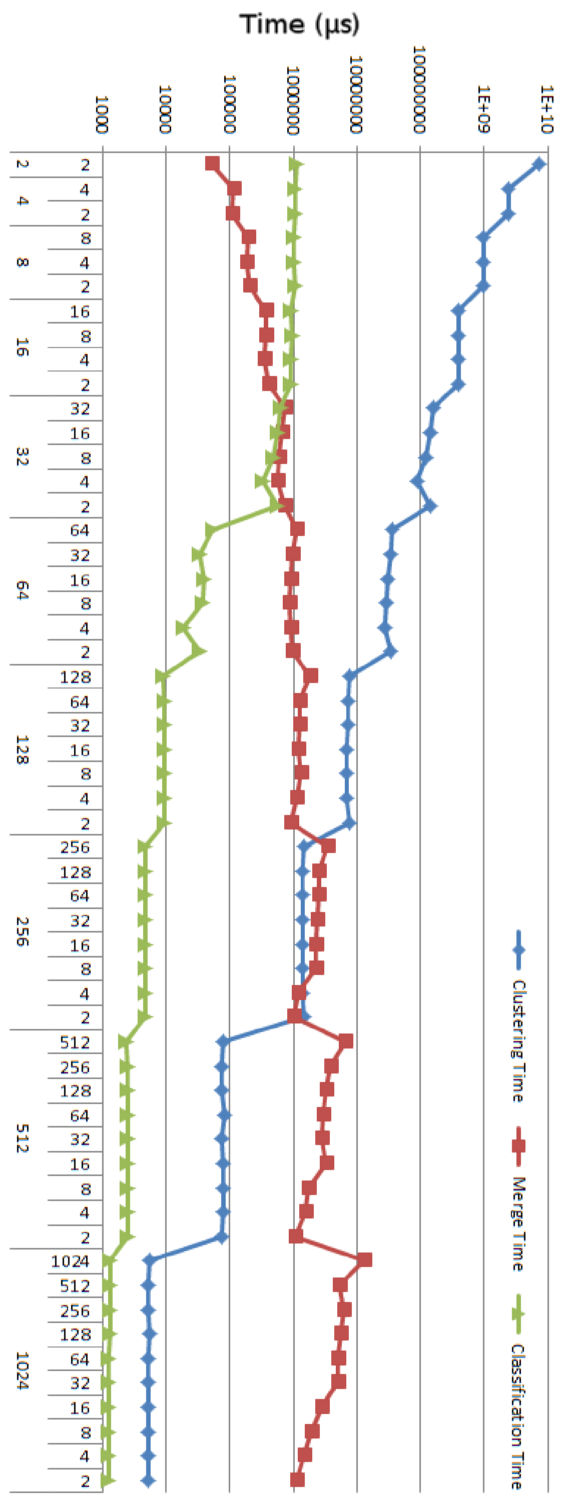


Figure 6.6.: Impact of the tree topology on the parallel DBSCAN algorithm performance

major gain from our approach is obtained in the exponential reduction of the cost of the clustering and classification phases by distributing the data, until they become negligible. Beyond this point, adding more processes does not improve much the net performance, and so the recommended setting for the number of back-ends is the one that keeps the amount of data per back-end around a few thousand points.

Regarding the fan-in, we can observe that the merge time (red line) decreases with lower fan-in values for any given setting of back-ends. A large fan-in means that the reduction tree is very flat. For example, when the fan-in is equal to the number of back-ends, all processes are connected to a single root node. Then, the root node is the only responsible for merging the clusters from all back-ends, and this process easily becomes saturated. This is the reason why the merge time is higher when the fan-in is large. Moreover, this setting is usually impractical, because a single process receives so many connections that the network becomes saturated. On the other end, when the fan-in is small (i.e. 2), the reduction tree is binary and very tall, and there are many intermediate nodes that can operate in parallel. As we can see in the graph, this is the point where the merge cost is lower, so using more processes compensate the cost of having more tree levels. However, a fan-in of 2 doubles the amount of resources necessary to build the tree. In this case, the recommendation for the fan-in is to be as small as possible, as long as the resources permit.

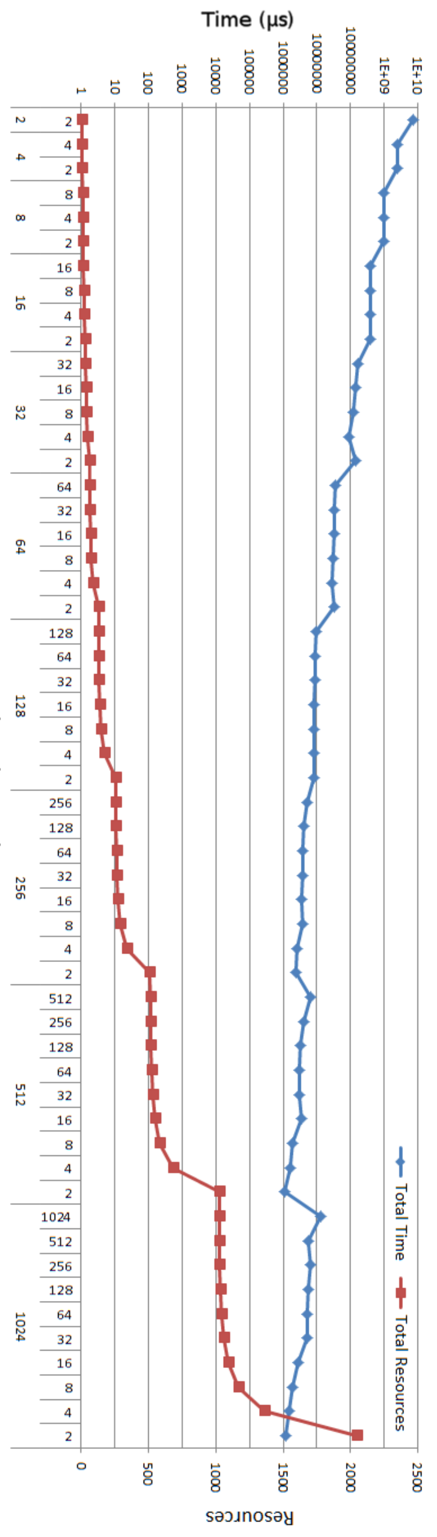
6.3.2. Speed-up analysis

Following on the Nekbone experiment, the sequential version of DBSCAN applied to this data set took 5.38 hours to complete the analysis.

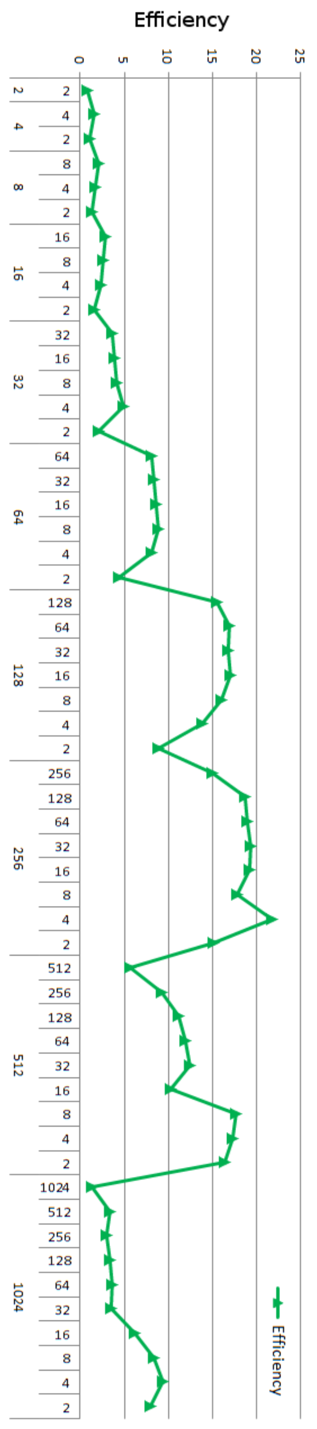
Figure 6.8 shows the speedup of our algorithm for the different configurations with respect to the sequential run. The X-axis now shows the total number of processes used (computed from the number of back-ends and the selected fan-in). The Y-axis shows speed up, and again, is displayed in logarithmic scale for better readability. Each series corresponds to the experiments done with a fixed number of back-ends but variable fan-in, where each point in the series has a decreasing fan-in, starting at the number of back-ends down to 2, from left to right. For example, the right-most series (purple), corresponds to all the experiments done with 1024 back-ends, and the first point in the series has a fan-in of 1024, while the last point has a fan-in of 2.

The black lines on the bottom represent the ideal speedup (this is linear to the number of processes, but the scale is logarithmic). All our experiments are above the ideal, which means that we are able to achieve super-linear speedup in all cases, approximately of one order of magnitude higher, even when we surpass the optimal number of back-ends (256 in this example, as we discussed in the previous section). Each of the series exhibits a mainly upward tendency from left to right, meaning that smaller fan-ins achieve higher speedups.

The highest speedup achieved is met in the 512-2 case (this configuration makes use of 1023 processes in total, 512 back-ends and a fan-in of 2), and the net speedup in this case goes up to 16800x with respect to the sequential run, which is about 16 times faster than the ideal speedup. The average speedup obtained considering all 55 experiments is around 3650x.



(a) Total execution time vs. total resources used



(b) Efficiency of the parallel DBSCAN algorithm

Figure 6.7.: Cost analysis of the parallel DBSCAN algorithm considering different resources

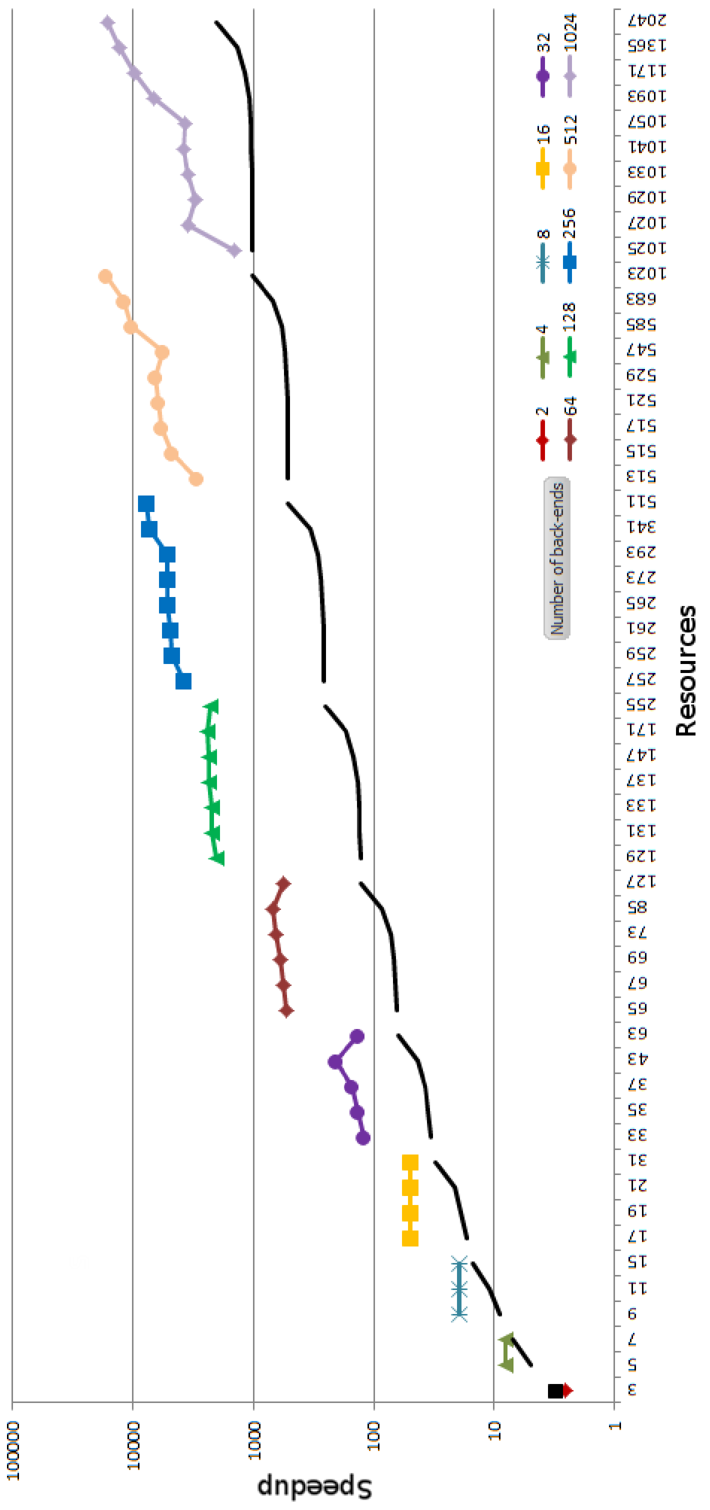


Figure 6.8.: Speed-up analysis of the parallel DBSCAN algorithm for a fixed problem size (strong scaling)

Next we evaluate the performance of our tree-based parallel implementation of the DBSCAN algorithm, constrasting it against a basic sequential implementation, and a sampling-based alternative. Then, we validate the results from the three implementations and discuss on their accuracy.

6.3.3. Performance evaluation

To evaluate the performance of the algorithm, we have selected a variety of benchmarks and real production codes to generate input data sets of different sizes and variable point distribution, and clustered the data using a basic sequential DBSCAN algorithm, a sampling-based algorithm, and the proposed parallel implementation. The sampling-based experiment applies the same sequential algorithm, but reduces the data to cluster by selecting a 25% of samples uniformly distributed across all tasks of the application being analyzed and over time. This setting was empirically proven valid in [7]. To perform the parallel experiments, we follow the criteria described in previous sections to select the number of back-ends and the tree fan-in: as many back-ends so that each one processes a few thousand points approximately, and a rather small fan-in. The first parameter will depend on the input size of each experiment, and the second, we have fixed it to 16 in order to balance the trade-off between good performance and low amount of resources in the larger-scale tests.

Table 6.1 shows the time in seconds to cluster the performance data of 9 different MPI applications. *Processes* indicates the amount of MPI tasks used to run the application. *Points* counts the total volume of data to cluster. *Eps* and *MinPts* are the DBSCAN parameters used. T_{seq}, T_{samp} and T_{par} show the clustering times for the sequential, sampling-based and parallel algorithms, respectively. In some cases, the sequential and sampling-based algorithms were not able to finish under the maximum allocation time of the system, which is set to 24 hours. This is marked with a *greater than* sign. T_{par} displays the total clustering time for the parallel algorithm, which is the aggregate of its three internal phases (clustering, merge and classification). *Back-ends* indicates the number of processes used in our distributed algorithm to cluster the data in parallel (i. e. the processes in the last level of the reduction tree). As explained before, the number of back-ends is adjusted based on the number of input points, so that each process clusters a small number of points (always below 10K per process). Finally, S_{samp} and S_{par} show the speedup for the sampling-based and the parallel algorithms with respect to the sequential time.

As the reader can see, the achieved speedup for the sampling-based algorithm (S_{samp}) is several times faster (up to 80x) than the sequential algorithm, which is expected as in this case there is a direct input data reduction. The speedup obtained with the parallel algorithm is in turn up to two or three orders of magnitude higher, more than 8,250 times faster than the sequential version in the CPMD case. In all cases, the parallel speedup is super-lineal, which suggest that our algorithm performs very well if the problem size is sufficiently large and the data is distributed among many back-ends.

Table 6.2 shows the clustering times for large scale tests. Here, we used as many back-ends to cluster the data as processes were used to run the application to analyze. In these cases, it was not feasable to compare with the sequential and sampling-based algorithms,

Application	Processes	Points	Eps	MinPts	T_{seq}	T_{samp}	S_{samp}	Back-ends	T_{par}	S_{par}
CPMD	256	244,913	0.018	64	>86,400	1,713.80	>50.41	256	10.47	>8,254.39
GTC	32	252,885	0.025	4	32,253.26	609.66	52.90	32	98.02	329.03
HACC	1,024	275,539	0.025	4	>86,400	15,841.64	>5.45	256	17.44	>4,953.73
LINPACK	16,384	360,390	0.01	1024	69,446.70	837.89	82.88	256	27.45	2,529.17
NEKBONE	8,192	204,800	0.025	4	19,363.61	586.06	33.04	1,024	2.85	6,786.52
PEPC	4,096	1,118,210	0.01	256	>86,400	>86,400	-	256	78.35	>1,102.70
PEPC	8,192	81,792	0.01	128	3,529.28	43.76	80.64	64	8.58	411.22
PFLOTRAN	16,384	868,368	0.01	1024	>86,400	19,365.35	>4.46	1,024	32.75	>2,638.31
SAMRAI	512	194,920	0.01	64	21,333.80	387.25	55.09	256	7.85	2,718.21

Table 6.1.: Performance evaluation of the proposed parallel DBSCAN algorithm compared to a sequential and a sampling-based version

Application	Processes	Points	Eps	MinPts	Time (s)
LINPACK	16,384	360,390	0.01	1,024	24.26
GROMACS	512	1,069,056	0.01	32	22.18
PEPC	4,096	1,118,210	0.01	256	13.02
PEPC	8,192	7,143,424	0.01	128	10.49
NEKBONE	8,192	2,064,384	0.025	4	6.97
SAMRAI	512	25,528,320	0.01	64	5,601.93
ZEUS	512	39,728,640	0.01	64	1408.61

Table 6.2.: Parallel DBSCAN large-scale experiments

because the input sizes are too large to finish the executions under 24 hours. The objective of these experiments is threefold: First, to show that the proposed parallel algorithm is able to process millions of points with low response times. For example, the ZEUS case processes up to 40 M points under 25 minutes. Second, to demonstrate that the algorithm can scale up to a large number of cores. For instance, the LINPACK case was run using 16,384 back-ends (17,477 cores in total considering the extra resources to build the reduction tree with a fan-in of 16), and the result was produced in less than 25 seconds, which doubles the highest scale achieved prior to our work by [124] and [125]. And lastly, to prove this approach useful to be applied in an on-line scenario.

As we have mentioned before, one of our targets is to apply cluster analysis on the performance data of a parallel application at run-time. Using as many back-ends as application processes, each process clusters its own data. This eliminates the need for an expensive phase of data redistribution, and enables us to scale to very large core counts. The unavailability of more resources has prevented us from experimenting with larger number of processes. However, the hierarchical distributed scheme presented here has been similarly applied in other works, which have proven to scale successfully to extremely large core counts (above 200K) [89].

6.3.4. Quality assessment of the clustering results

Due to the expensive costs of the regular sequential implementation, it is not possible to do a fair comparison using cases with large volumes of data. Instead, the validation process was performed with a subset of data extracted from a trace, reduced to a volume that is manageable for the sequential version, so that all three algorithms run with the same exact input. In order to compare the clustering results of the three algorithms, we will be using two quantitative measures: the *Mirkin Distance* and the *Sequence Score*.

Given two clusterings on the same data, the Mirkin Distance [157, 158] measures the number of pairs of points that are in the same cluster in the first clustering, but in different clusters in the other. Basically this is a percentage that tells you how many points were accidentally put in separate clusters, which is directly interpretable like a percent error. Regarding the noise points, they are considered to be their own cluster. If two points are both noise in the first clustering, they should still be noise in the second, and so it is just another class of points as far as the Mirkin Distance is concerned. This metric provides a simple quantitative statement to determine how different the clustering implementations are.

The Sequence Score [159] is applicable as long as the data can be represented in the form of a temporal sequence. This is exactly our case, since the data corresponds to sequences of timestamped performance measurements per process. Given this condition, the Sequence Score employs a Multiple Sequence Alignment (MSA) algorithm to align the data sequences, and calculates the percentage of points that belong to the same cluster that happen simultaneously in all of the sequences. In our particular case, this gives a quantitative measurement of the degree of SPMDiness of the application. If two points that belong to the same cluster are accidentally separated, that would reduce the value of the Sequence Score, and so it can be also interpreted as a metric to compare the similarity of the structures detected in the different implementations.

Table 6.3 shows the evaluation of these two metrics for the three clustering algorithms, for 11 different applications. Mk_{samp} and Mk_{par} show the Mirkin error metric for the sampling-based and parallel algorithms. A value of 0 indicates that the results are exactly the same compared to the sequential DBSCAN algorithm. As the reader can see, the Mirkin metric stays under 5% of error for the sampling-based algorithm (Mk_{samp}), and slightly improves in the parallel algorithm reducing the error below 2%. In the sampling-based algorithm the error is introduced because the amount of data clustered has also been reduced. If the selected samples were not good representatives, the algorithm might fail to form or expand a cluster. In the parallel version, the error is introduced because we chose to relax some of the DBSCAN axioms, as we explained in Section 6.2.3. However, our approach does not only improve the quality of the results with respect to a sampling-based alternative, but also has a positive effect that can be studied with the Sequence Score metric.

Columns Ss_{seq}, Ss_{samp} and Ss_{par} show the Sequence Score metric for the sequential, sampling-based and parallel algorithms, respectively. A value of 100% means that the application exhibits a perfect SPMD pattern. Looking at the sampling-based results (Ss_{samp}), it is easy to see that the values for this metric are slightly worse in most cases compared to the baseline sequential algorithm (Ss_{seq}). This is due to the fact that the sampling-based algorithm tends to produce slightly different results around the clusters borders, because missing a few samples in these areas might reduce the density enough so as to prevent the cluster to continue expanding. The interpretation on the data would be that these border points correspond to performance outliers. If we discard these outliers but the code is SPMD, the Sequence Score metric decreases. However, in the SAMRAI case the Ss_{samp} error increases by 9%, and in the GROMACS case the error increases up to 23%. This is because these two codes present non-SPMD patterns. SAMRAI employs adaptive mesh refinement methods to change the accuracy of the solution in certain regions. GROMACS performs two major compute tasks in different sets of processes simultaneously: non-bonded forces on some processors, and Particle Mesh Ewald calculations on the others. Skipping samples of non-SPMD patterns increases the chance to miss relevant points, which leads to worse clustering results.

Nevertheless, the Sequence Score for the paralell algorithm (Ss_{par}) presents differences

Application	Processes	Points	Eps	MinPts	Back-ends	Mk_{samp}	Mk_{par}	Ss_{seq}	Ss_{samp}	Ss_{par}
AMG2013	384	16,896	0.025	8	16	1.87%	1.78%	97.66%	94.89%	95.95%
AVBP	960	19,200	0.03	4	16	0.06%	0%	99.92%	99.64%	99.92%
CGPOP	128	1,280	0.035	4	2	0.01%	0%	68.61%	68.62%	68.61%
GADGET	64	1,877	0.02	10	2	0.12%	0%	92.65%	92.43%	92.65%
GROMACS	512	37,504	0.01	32	32	4.05%	0.47%	52.15%	29.95%	51.70%
GTC	32	10,752	0.025	4	32	0.05%	0.08%	93.15%	93.09%	93.10%
IPIC3D	512	4,800	0.025	4	4	0%	0%	100%	100%	100%
MRGENESIS	8	408	0.06	13	2	0.08%	0.06%	99.78%	99.61%	99.89%
NEKBONE	8,192	204,800	0.025	4	1,024	0%	0%	100%	100%	100%
PEPC	8,192	81,792	0.01	128	64	0.60%	1.34%	63.83%	57.03%	65.30%
SAMRAI	512	194,920	0.01	64	256	0.3%	0.03%	48.26%	39.93%	45.13%

Table 6.3.: Quality assessment of the parallel DBSCAN clustering results compared to a sequential and a sampling-based version

under 1% in most cases and always below 4%, and even in some experiments (e.g. MRGENESIS and PEPC), the Sequence Score slightly improves, which means that relaxing the DBSCAN conditions with a variable MinPts parameter and a more permissive criterion to consider whether a point is part of the cluster, generally leads to a slightly better detection of SPMD structure.

The sampling-based implementation shows higher errors in all cases, and they obviously keep increasing when the training set is further reduced. Despite the resulting clusterings are not exactly equal, we do not necessarily care that we get to the same exact results of the regular DBSCAN, but to get extremely close. In our context of application, the clustering itself is mostly useful for qualitative analysis, and if the clusters are mostly the same, the marginal differences are perfectly valid for our purposes.

6.4. Performance models based on cluster analysis

Cluster-based structure detection adds value to the performance analysis of parallel applications because the whole information available is resumed into a small subset of clusters that characterize the main computing trends of the program. In this section, we show two possible ways to look at the resulting data to get useful analysis information. We present two types of studies based on defining simple performance models to extract quick conclusions from large volumes of clustering data.

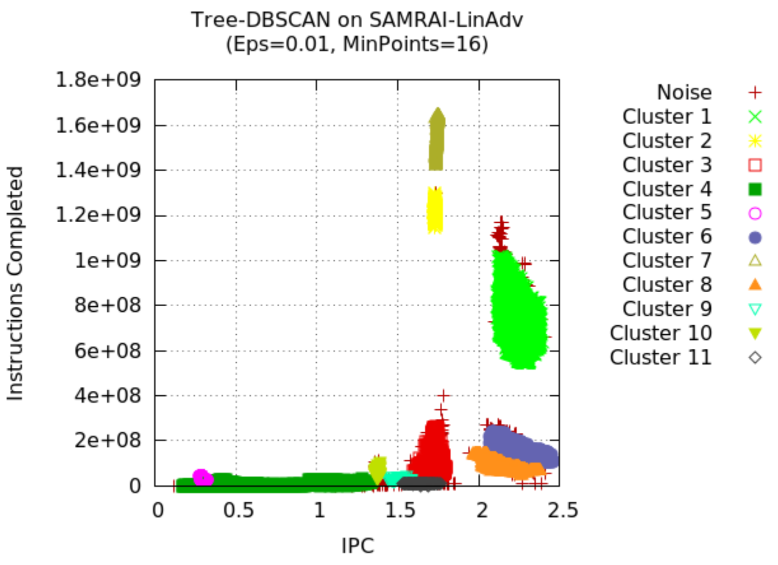
To this end, we studied a linear advection (LinAdv) benchmark that uses a popular SAMR library called SAMRAI [160]. This code is interesting because it employs adaptive mesh refinement (AMR) methods to change the accuracy of the solution in certain regions dynamically, and so we expect from it to exhibit variable computations and disperse clusters. The experiment ran on MareNostrum III, a cluster comprising 3,028 nodes, each containing 2 Intel SandyBridge-EP E5-2670 8-Core at 2.6 GHz with 32 GB of RAM. The application used 512 MPI tasks, which generated a total data volume of 194,920 points to cluster, processed in 13.85 seconds.

The results of the clustering are shown in Figure 6.9a. As expected, the application presents different computing behaviors with divergent trends. For instance, clusters 2 (yellow) and 7 (brown) on the top clearly stretch vertically, denoting instructions variability. Clusters 4 (dark green), 6 (purple) and 8 (orange) elongate horizontally instead, indicating performance variabilities. While some clusters are very compact, clusters 1 (light green) and 3 (red) widen in both axes covering a large range.

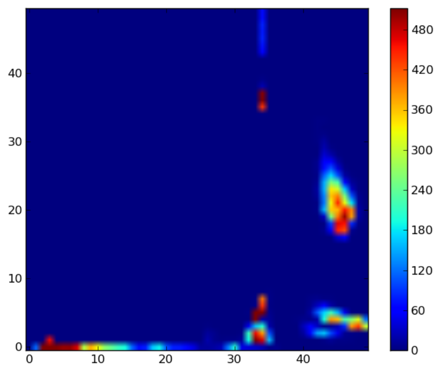
Figure 6.9d shows the distribution of the clusters over 3 iterations of the main loop of the application. We were able to detect a very clear SPMD structure of the program, as all processes are executing the same type of computation simultaneously.

6.4.1. Heat map model to characterize cluster variability

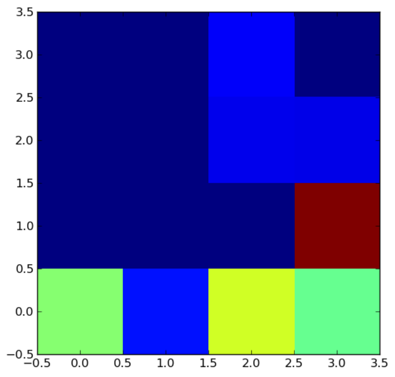
When performance measurements of a computing burst are plotted in a scattergraph, time and space components are ignored. This is to say, you can no longer know which parallel task has produced which points, or when.



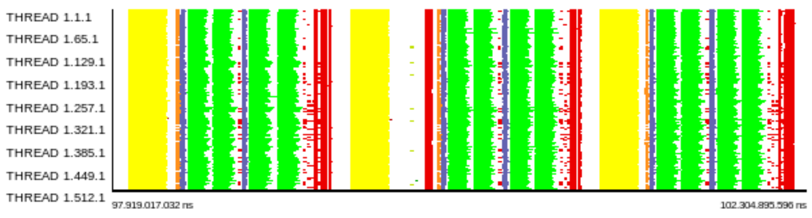
(a) Computations structure



(b) Fine-grain temperature



(c) Coarse-grain temperature



(d) Distribution of clusters over 3 iterations of LinAdv

Figure 6.9.: Clustering results for the SAMRAI LinAdv benchmark

The time component (when the computation happened) and the space component (in which process) are relevant because it depends on how the data distributes over time and space that we can attribute performance variabilities to problems of imbalance, arguably one of the main factors that limit scalability. For example, when all processes execute the same computing phase at the same time, but with variable performance, this indicates a problem of imbalances between processes. Or if a given process achieves different performance for the same computing phase over different iterations, this indicates a problem of imbalances over time.

The shape of the clusters in the scattergraph gives us hints about potential imbalance problems in those clusters that appear elongated because they present high variability. But this information alone does not allow us to be more precise until we correlate the scattergraph with the trace timeline that preserves the temporal and spatial information.

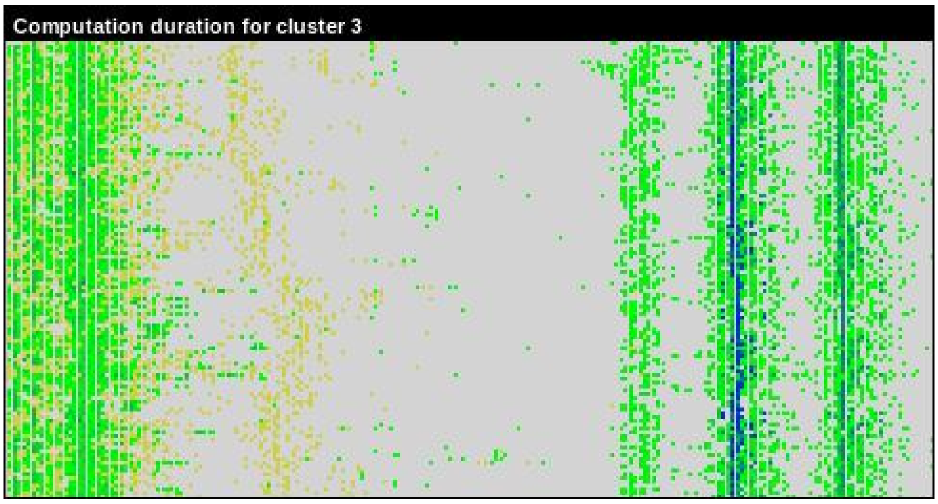
To partially recover the information that we lose when building a classical scattergraph from trace data, we propose a new representation where the color of the points does no longer represent the cluster to which they belong. Instead, the color indicates how many processes contribute with data to each region of the space that is part of the cluster. To do so, we discretize the space into small equi-sized bins (a grid of 50x50 by default), and each back-end annotates whether they have points in each of the bins. This information is then aggregated over the reduction tree, and so the front-end node receives a summation of how many processes have data in each area of the space. This can be seen as the heat map depicted in Figure 6.9b, where the temperature indicates the number of processes with data in the area. Then for each cluster, we can compute its average temperature (the average temperature of all the bins that are part of the cluster). The resolution of the heat map can be reduced, as shown in Figure 6.9c, enabling a quick detection of where the most frequent type of computations of the program fall in the clustering space. In our example, most frequent computations are located at the bottom-right corner, meaning that most processes usually report low instructions and high IPC.

This information is also useful to determine the source of the variabilities in wide, disperse clusters. If the temperature of the cluster is low, the different processes that contribute to the cluster achieve different performance for the same computations (e.g. some processes perform their computations faster than others), and thus this is a reflection of behavioral differences between processes, which in the case of a SPMD code will be indicative of spatial imbalance problems.

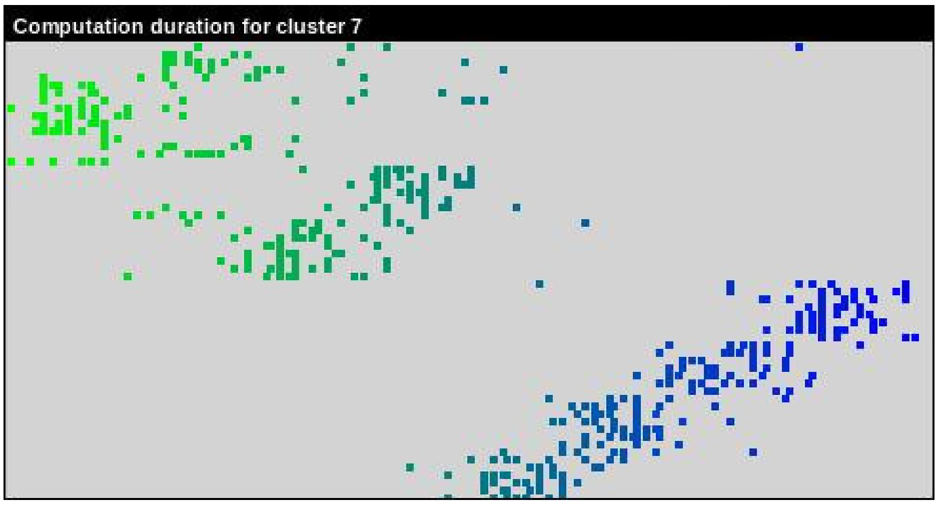
However, if the temperature of the cluster is high, all processes perform all types of computations (e.g. all processes have instances of fast and slow computations), so the dispersion in the cluster can be explained because there are also behavioral differences at different points of the execution, and thus this is indicative of problems of imbalances over time.

Following with the LinAdv experiment, Table 6.4 shows the average temperature for each cluster. The temperature is calculated as mentioned above, averaging the temperature of all the bins of the discretized space that belong to the cluster.

Cluster 3 which appears in Figure 6.9a as a disperse cluster in both axes, has a very high temperature of 91.78%. Figure 6.10a shows the histogram of the duration of the computations that belong to this cluster. The rows represent the parallel tasks, and the columns are



(a) Cluster 3. The three vertical trends on the right-most part of the histogram indicate that all processes execute three modes of computations with increasing duration.



(b) Cluster 7. The disperse points falling into different columns indicate that every process executes computations with different duration

Figure 6.10.: Histogram of computations duration for two clusters of LinAdv showing different patterns of imbalance

Cluster	1	2	3	4	5	6	7	8	9	10	11
Avg. Temperature	69.22%	99.01%	93.46%	91.78%	90.18%	61.18%	14.37%	19.47%	30.37%	3.12%	12.96%

Table 6.4.: Average temperature per cluster in SAMRAI LinAdv experiment

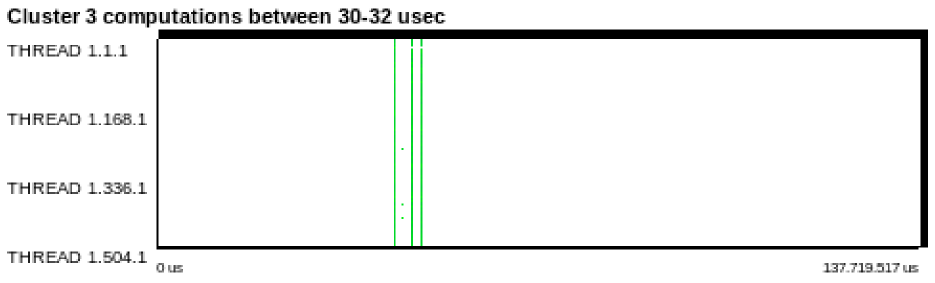
increasingly higher intervals of durations. In this way, points at the right-hand side represent the longest computations, and the colors from green to blue indicate which trend is more frequent. On the right side of the histogram there are 3 clear vertical trends, meaning that all processes execute phases with short, medium and large computations, respectively. Figure 6.11a shows the timeline for the whole execution where the short computations that belong to cluster 3 happen. As the reader can see, they occurr in all processes simultaneously, but at different moments in time. We have the same situation for the medium and large computations, as shown in Figures 6.11b and 6.11c, respectively. All these computations are interspersed in time, revealing a problem of performance variabilities occurring over time, as the temperature measurement had predicted.

Cluster 7 is located on top of the instructions axis in Figure 6.9a, so these are the computations that perform more work in the execution, and shows significant dispersion in this axis. The temperature for this cluster is very low (14%). As stated before, low temperatures indicate variability between processes. This can be verified looking at the histogram of the computations durations in Figure 6.10b. Each process (rows) has data in a different column of the histogram, meaning that some perform faster and some slower, but they are all different.

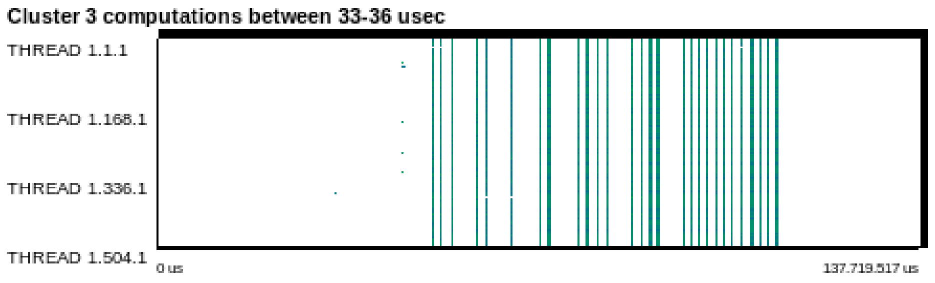
Since the computing regions are linked to the source code through callstack information gathered during the execution, it is possible to relate the clusters with points in the code. For these two, Cluster 3 relates to the regridding procedure of each AMR hierarchy level at routine `regridFinerLevel`, and Cluster 7 corresponds to the initialization phase of the AMR hierarchy at `initializeHierarchy`. The cluster temperature measurement helped us to characterize the source of variability behind clusters with high dispersion, without having to look into the trace to study the temporal and spatial distribution of the data.

6.4.2. Performance-breakdown models based on hardware counters extrapolation

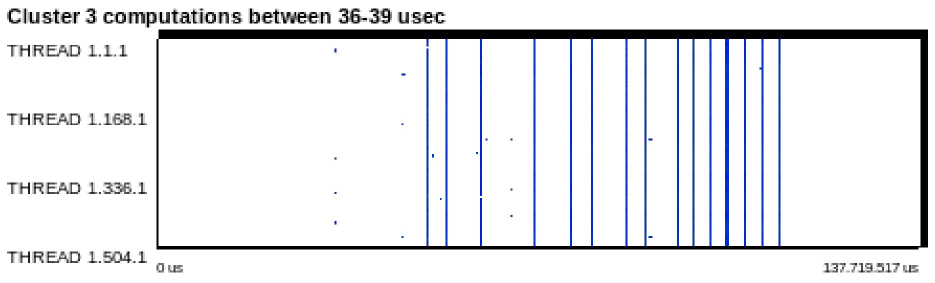
There are some limitations to retrieve hardware counters information from the microprocessor: the number of available registers usually range from 4 to 8, and many combinations of counters are not allowed. In order to measure more counters beyond these limitations one would normally have to repeat the execution several times, defining different sets of counters to gather all the information. The performance counters extrapolation technique [161] consists in multiplexing the set of hardware counters that is being read at run-time,



(a) Trend of short computations performed by all processes



(b) Trend of medium computations performed by all processes



(c) Trend of long computations performed by all processes

Figure 6.11.: Timelines showing where the short (up), medium and long (bottom) computations that belong to Cluster 3 of LinAdv occur over time. The vertical lines indicate that all processes execute the same type of computation simultaneously.

and then using cluster analysis to characterize the computations with more performance counters than the ones that can be read simultaneously with minimum error and just a single execution of the application.

More specifically, the set of hardware counters is changed periodically so that there is enough coverage for all metrics in the entire run. In this way, different computing phases will measure different counters. As long as there is a subset of counters that is common in all sets, those will be present in all the measurements, and they can be used to cluster the data. Once the groups have been detected we can combine the data from the different bursts and compute the average value of all counters present in all groups for all the clusters detected. Consequently, we obtain a complete characterization of all clusters.

Since the clustering process is performed locally in each of the back-ends with partial data, the counters extrapolation process applied locally results in incomplete data. Our implementation extends the cited technique so that the local metrics are combined across the reduction tree, where each node computes the average value for all the metrics seen in their children nodes. As a result, the front-end node obtains a profile of all metrics for all clusters, taking into account the data from all tasks of the parallel application.

Using this technique we are able to retrieve from a single run enough hardware counters measurements to define break-down models to describe the behavior of the different computing regions of the application. Following on the previous example, Figure 6.12a shows the architecture impact model for the LinAdv benchmark that measures the percentage of cache misses and branch mispredictions over the total number of instructions executed. It is interesting to note that clusters 5 and 6 present a significantly higher percentage of cache misses compared to the rest. These two clusters correspond to the creation of the input database and the parsing of the input data during the initialization phase, and time integrator methods invoked from the main loop of the benchmark, respectively. Figure 6.12b shows the instructions mix break-down, classifying the instructions executed into different typologies. For the previous two clusters, they have balanced percentages of loads, stores and branches, while others such as clusters 1, 2, 7 and 12 are executing loads for the most part.

6.5. Related work

Regarding the problems of DBSCAN to handle very large data sets in the context of performance analysis and others, two opposed strategies have been explored to improve the scalability of the algorithm: scaling down the data and scaling up the algorithm with different parallelization schemes.

In the first direction, SDBSCAN [162] was the first sampling-based alternative. This approach incorporates sampling techniques to randomly select samples from the Eps-neighborhood to perform the exploration phase of the algorithm. In [7] the authors apply this idea in an on-line scenario, selecting representative samples from different processes as a training set for the clustering phase, and then classify the remaining data following a nearest-neighbor criterion. The sampling approach can reduce the clustering time while preserving the quality of the results in many cases. But when the data volume grows very large and the samples subset represents a very small fraction of the total data, the correctness of the resulting clustering quickly decreases. This effect is particularly dangerous in performance analysis, where the data often presents variability that is important to capture so as to detect zones of imbalance, and a poor selection of the samples easily mistakes variability for noise.

On the other hand, several parallelization strategies for DBSCAN have been presented. PDBSCAN [121] used a distributed spatial index structure in which the data is spread among multiple nodes. When a given node needs to query data that is stored in a different one, they communicate through messages to retrieve the data. This approach showed near-linear speedup up to 8 nodes, but beyond the amount of messages quickly grew superlinearly restraining the scalability.

MR-DBSCAN [122] and DBSCAN-MR [123] are implementations based on the MapReduce paradigm. In both cases, the data needs to be preprocessed into balanced partitions so as to contend with load imbalance issues caused by variable-density regions. Neither of both showed near-linear speedup nor the scalability of the algorithm beyond 12 nodes.

PDSDBSCAN [124] applies graph-algorithmic concepts to define the disjoint-set data structure to break the access sequentialitity of DBSCAN, and a tree-based approach to construct the clusters. This approach outperforms previous master-slave strategies by yielding a better-balanced workload and hence result in higher parallel efficiency, scoring speedups up to 5765 using 8192 cores on distributed memory architecture, but degrades beyond this point due to the large number of messages required to manipulate the distributed data structure.

Mr. Scan [125] combines a tree-based distribution network with hardware accelerators. this hybrid implementation clustered 6.5 billion points on 8192 GPU cores, which is the largest run of DBSCAN by point and core count that we are aware of. Mr. Scan also starts with a preprocess phase in which they apply an optimized partitioning algorithm to divide the space into high-density areas, allowing for points in these regions to be marked as members of a cluster without incurring the cost of expanding each point individually. Speedup decreased beyond 2048 nodes because the computation is limited by the slowest cluster process that is executing a partition that can not be subdivided any further.

Our version of DBSCAN shares some design aspects with the different works stated above, and its distinguishing key features are: First, we follow a tree-based structure where the complete data set is splitted into smaller, local clusterings performed at the leaves of the tree, and the partial results are combined hierarchilly. Second, we do not require any kind of data preprocess for optimal partitioning: our data distribution implies that all local clusterings process a subset of points that are scattered across the whole data space. Third, we do not rely on specific hardware accelerators, but aim at general purpose processors. Fourth, we simplify the expensive problem of distance searches in the Eps-neighborhood by modelling the clusters into simple geometries, and detecting collisions between geometries. Lastly, this approach does not guarantee that all axioms in the original definition of DBSCAN are preserved, hence the resulting clustering is an approximation, that we have tested empirically to present dissimilarities below 2%.

6.6. Closing remarks

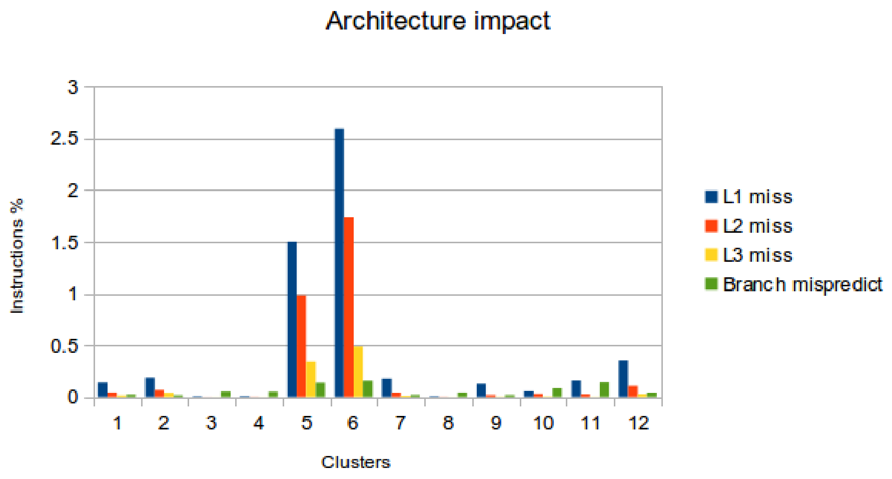
When the amount of performance data grows large, aggregating all the data and doing a single global analysis is not an option, as the density-based clustering cost increases quadratically with the input size.

In this chapter we have presented a parallelization strategy for the DBSCAN algorithm structured as a hierarchical tree, where the leaves execute a basic DBSCAN with local data, the resulting clusters are modelled with convex hulls, and these are intersected in parallel in the intermediate nodes of the tree to obtain a global clustering model. Finally, the local data of each process is classified using a nearest neighbor algorithm using the global model as reference. Our approach is general enough to be applicable by other cluster algorithms families that can also take advantage of a distributed structure to easily parallelize their computation.

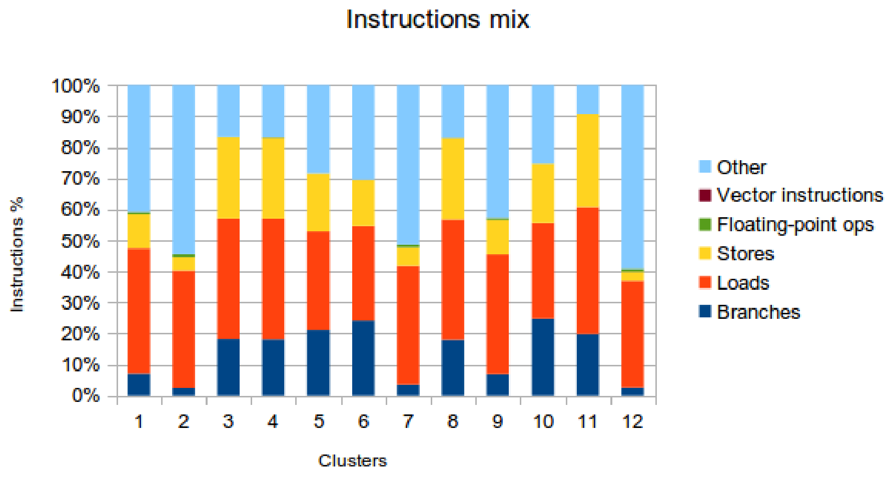
Our implementation does not require an explicit data redistribution phase to keep dense areas close. It is actually designed to work efficiently when the dense areas are highly distributed, which is the situation of performance data from parallel applications. In the experiments, we scaled comfortably to 16,384 cores and clustered millions of points in few seconds. The clustering quality was verified with two different metrics: the Mirkin distance and the Sequence Score, and the results obtained approximate those of the original DBSCAN with marginal differences below 2%.

Considering its design, this algorithm will scale very well to larger core counts, given that the local clusterings are performed with low volumes of points. When applied to performance data from parallel applications, the resulting clusters are usually in low counts (rarely over 10 to 15 clusters), and so the reduction phase is performed with a steady and small number of convex hulls per child. Withal, there is room for improvement in both phases. The local clustering could use an implementation other than the basic DBSCAN to support larger local data volumes, and even a second level of parallellism could be used for the local clustering. In the reduction phase, the current implementation tries to intersect all convex hulls, increasing the cost exponentially, which is only reasonable because the number of convex hulls is small. However, very much like many implementations of this algorithm do, it would be possible to include a spatial index to reduce the number of intersections and lower the complexity to logarithmic cost.

In addition to the new DBSCAN implementation, we have presented two analysis features to demonstrate the utility of density-based clustering in performance analysis. First, we introduced a new scatter-plot representation for the clusters based on the idea of heat maps, where the temperature indicates how many parallel processes support the different areas of the clustering space. The average temperature of the clusters can be used to characterize the type of imbalances that may be behind very disperse clusters. Second, we extended an extrapolation mechanism for hardware counters to compute global statistics in parallel, enabling us to characterize the clusters with many more metrics than those that can be gathered in a single run, as well as building break-down models based on them such as the architecture impact and the instructions mix stack charts. These high-level representations of the clusters performance provide quick and easy to understand information and useful insight about the application behavior.



(a) Architecture impact model



(b) Instructions mix model

Figure 6.12.: Performance models based on hardware counters metrics for SAMRAI

Chapter 7

Object tracking techniques to improve comparative and multi-experiment analysis

Understanding the possible changes in behavior that an application can undergo over time requires to integrate observations from multiple execution intervals. Furthermore, scientific applications can have so many variables, possible usage scenarios and target architectures, that a single experiment is often not enough for an effective analysis that gets sound understanding of its performance behavior. Different software and hardware settings may have a strong impact on the results, but trying and measuring in detail even just a few possible combinations to decide which configuration is better rapidly floods the user with excessive amounts of information to compare.

In this chapter we introduce a novel methodology for performance analysis based on object tracking techniques. The most compute-intensive parts of the program are automatically identified via cluster analysis, and then we track the evolution of these regions across different execution intervals and multiple experiments to see how the behavior of the program changes with respect to the selected settings and over time.

This approach addresses an important problem in HPC performance analysis, where the volume of data that can be collected expands rapidly in a potentially high dimensional space of performance metrics, and we are able to manage this complexity and identify coarse properties that change when parameters are varied to target tuning and more detailed performance studies.

7.1. Background and motivation

The execution of a scientific code is dependent on a variety of parameters that may have a strong impact on its performance. Some examples are the size of the input problem, the number of processes running in parallel, the physical mapping and sharing of the resources or the parallel programming model used, as well as many other settings. Anticipating the impact of different configurations on the achieved performance, work balancing or memory usage of the program is far from trivial and not seldom leads to discover unexpected issues.

Analyzing these effects is important not only to get better understanding of the program behavior, but also to foresee improving or degrading trends in the different parts of the code, identify the main limiting factors, and in the end, to help the users making the right decisions to tune the application to achieve the most performace outcome. To this end, it is necessary to have tools to easily compare different experiments and correlate observations between them.

In order to deal with the difficulties inherent to running, measuring and comparing multiple experiments, we have designed a tool to conduct truly diverse parametric and evolutionary studies, enabling to correlate performance information either from multiple runs with different configurations, or different time intervals within the same experiment. Our approach focuses on the computational behavior of the most relevant code regions and shows their evolution with respect to several performance metrics to explain which factors lead the different parts of the code to improve or degrade. In this context, the use of object tracking techniques has revealed to be a natural and intuitive way to detect the performance changes sustained by each part of the code automatically, and represent the information in a very visual manner.

While previous approaches for comparing experiments or phases [113, 45, 126] have been proposed, our work goes one step further and presents a novel technique that does not rely on preselected metrics and profile data for static code phases, such as routines, loops or user-defined sections. One problem of summarizing the data at these levels is that one same section of code can exhibit behavior variations, thus making averages will hide divergent performance trends. Our position is that it is necessary not to consider averages, but every independent instance to detect fine-grain structure and capture multi-modal variability.

7.2. Object tracking for performance analysis

Tracking techniques have been traditionally used to follow moving objects in an image or video sequence. Practical examples include augmented reality, medical imaging, surveillance or traffic control. A first step to these problems is to delimit the objects of interest within the scene depicted in the image. Therefore, object recognition algorithms (e.g. image segmentation and edge detection) will look for appearance characteristics and distinguishing features (e.g. color, direction or shape) that identify them. Then, consecutive frames in the sequence are compared to find correspondences between the objects and their displacements.

Analogously, we will represent different executions as images, each one picturing the program behavior for a given configuration, and arrange them as a sequence of images that expresses the evolution of the application behavior across experiments. Code regions are drawn in the images as independent trackable objects, in a space whose dimensions are not the actual physical dimensions of height, length and breadth, but performance metrics that describe how these regions behave. Movements in the performance space across the images highlight changes in the application performance, modeled into metrics that evaluate the performance trends of the different regions of code.

This approach is useful to discover valuable performance insights about the application response to different configurations, enabling the analyst to draw quick conclusions on the key factors limiting performance, direct the optimization effort and easily determine the best setup to maximize a certain performance requirement. Throughout this chapter, we will be showing how this method applies to very diverse cases of analysis to get better understanding of the impact of different architectures, input problems, workloads, memory and resource sharing schemes, and levels of scalability on several parallel programs.

7.2.1. Application structure characterization

Analysis tools usually choose to display performance data to the user in the form of profiles at the level of syntactic program structures (i.e. subroutines, loops, or user-defined sections). This has the advantage of being a very natural and understandable representation, but also carries some drawbacks along. Prior knowledge of the application may be required to determine which functions are relevant, so as to skip too fine-grain routines that would perturb the execution due to the instrumentation overhead. When no automatic interposition mechanisms are available [35], access to the sources and manual modifications are needed to inject measurement probes in these points of interest. Moreover, considering a whole routine as a single unit of behavior can be deceitful, because different invocations may behave differently, depending on the parameters and conditional phases leading to distinct code flows with divergent performance. In these cases, a global average may convey the wrong idea of a reasonable overall behavior, while specific sub-phases may be reporting low performance and their optimization could lead to significant improvements, as proven in [38, 163].

A different granularity to characterize the application performance is the computing regions (i.e. CPU bursts). These are defined as the sequential computations between calls to the MPI or OpenMP runtime. Delimiting these regions only requires library interposition to instrument the parallel programming API, thus there is no need for user intervention nor access to the sources. Each CPU burst is characterized by its duration, call stack references that point to the corresponding source code, and a vector of hardware counters metrics describing how it performed. Considering every CPU burst rather than simple averages, we can detect variabilities across processes and time, exposing a fine-level characterization of every code region and the nature of their inefficiencies.

This approach is less attached to the structure of the source code, but focuses on the performance properties of the actual computations. In [114], the authors prove that this granularity is useful for the analysis of parallel programs, as it reflects an intermediate point of view between very low level characterizations (i.e. basic blocks or instruction-level simulators) and higher abstractions (i.e. functions, loops or user-defined sections). Regardless of our implementation, which selects CPU bursts as the target granularity, the

	IPC	Instructions	L1 miss
x_solve	2.16	43.04 M	295.92 K
y_solve	2.16	43.83 M	323.07 K
z_solve	2.17	46.22 M	55.63 K

Table 7.1.: Profile of user functions for the NAS BT-MZ benchmark

technique presented would as well be applicable using other abstractions.

7.2.2. Generation of tracking images

In computer vision, one or more particular objects (e.g. humans, cells or cars) are first identified within a frame (a single picture in a series of images) and then tracked as they move through a sequence of frames. Likewise, we are going to identify the computing regions of interest and keep track on how their performance evolves along multiple experiments. To this end, we first need to represent the performance measurements observed in each experiment graphically, or in other words, to capture our sequence of frames. This process consists in selecting any pair of metrics to draw a two-dimensional space where we express the behavior of every individual CPU burst with a point in the plane. Typically, we select *Instructions per Cycle (IPC)* and *Instructions Completed*, which are useful to bring insight into the overall performance: trends in *Instructions Completed* indicate regions with different workloads, while *IPC* measures how fast the work is done. Anyhow, this process can be applied to any arbitrary combination of metrics that may be used to describe the CPU bursts (e.g. cache misses, floating-point operations or power consumption) to support even more precise multi-dimensional characterizations of the data.

With the images generated, the next step is to identify the objects of interest within them. Due to the highly iterative nature of HPC applications, many computations will be very alike in terms of the performance they achieve. In the image, this translates as clouds of points that are close in the space, which can be grouped into a single entity according to their similitude. Therefore, we apply the cluster analysis technique presented in [114, 161], that uses density-based clustering in order to group similar CPU bursts with respect to the metrics selected.

The result of this process is a scatter-plot representation of the performance space, where the axes correspond to the metrics used to cluster the data, and all CPU bursts that are similar with respect to these metrics get grouped into the same object. Clusters are then intrinsically connected to the source code regions of their belonging CPU bursts, and both terms will be indistinctly used for clarity, but this connection is not necessarily unambiguous: a single region presenting bimodal behavior will result in two distinct clusters, while two different regions with similar behavior will conform the same cluster. So in essence what each cluster represents is a behavioral trend, independently of the code region that exhibits it.

One question that may arise about the benefits of using these performance images is to what extent they are better than just a straightforward profile. To dispel the doubt,

	IPC	Instructions	L1 miss	% Time
Region 1	2.21	19.15 M	56.45 K	36.95%
Region 2	2.13	53.46 M	266.33 K	12.28%
Region 3	2.16	42.36 M	194.32 K	12.08%
Region 4	2.12	65.79 M	363.01 K	11.43%
Region 5	2.18	33.87 M	133.19 K	11.42%
Region 6	2.11	83.27 M	494.41 K	9.68%
Region 7	2.05	101.61 M	949.55 K	4.01%
Region 8	2.10	109.46 M	115.66 K	2.13%

Table 7.2.: Profile of clusters for the NAS BT-MZ benchmark

we have selected as example the BT-MZ benchmark [141], a solver for block tri-diagonal systems that performs computations of uneven size. Table 7.1 shows the average IPC, instructions and L1 misses scored by three of the main functions, measurements obtained by instrumenting the routines at their start and end points. From these numbers, we can easily infer that all three routines present a similar behavior, with the same amount of work (Instructions) executed at the same speed (IPC), yet they show different memory efficiency with lower L1 cache misses in the Z-direction, certainly due to the data access pattern. One could expect this result, as these functions perform the same kind of computation over different axes.

Figure 7.1 shows the performance image generated for these functions, with each dot in the plot being a single instance of invocation, and grouped in clusters with respect to the IPC achieved and the number of instructions executed. A function-agnostic view of the data brings new insights about the application structure: The three functions show eight different computational behaviors with increasing amounts of work and decreasing speed. Computations with high amount of work but low performance are interesting to study, as well as those with the same amount of work at different speeds, or vice-versa, as these indicate potential load-imbalances. All eight behaviors are exhibited by all three functions, which still conveys the idea that these functions are similar, but exposes their inner variability as they behave more or less optimal depending on the size of the workload.

Table 7.2 shows the same statistics for the clusters, and now you can easily see a large dynamic range in the metrics. Most significantly, a standard deviation of 30 M of instructions reveals a large work imbalance between clusters, which was masked in a traditional function-based profile. Column *% Time* makes clear that these computational behaviors cover an important fraction of the total execution time, and thus the importance of being aware of these variabilities. This example highlights the importance of focusing on the dynamic behavior of the regions rather than static code structures to guarantee that we detect performance variabilities and direct the analysis towards the zones of real interest.

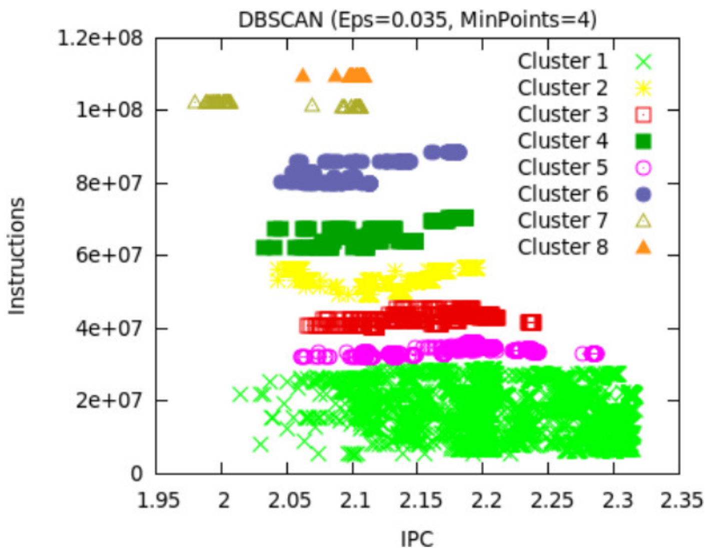


Figure 7.1.: Clusters for three main functions of the NAS BT-MZ Class B (4 tasks) benchmark

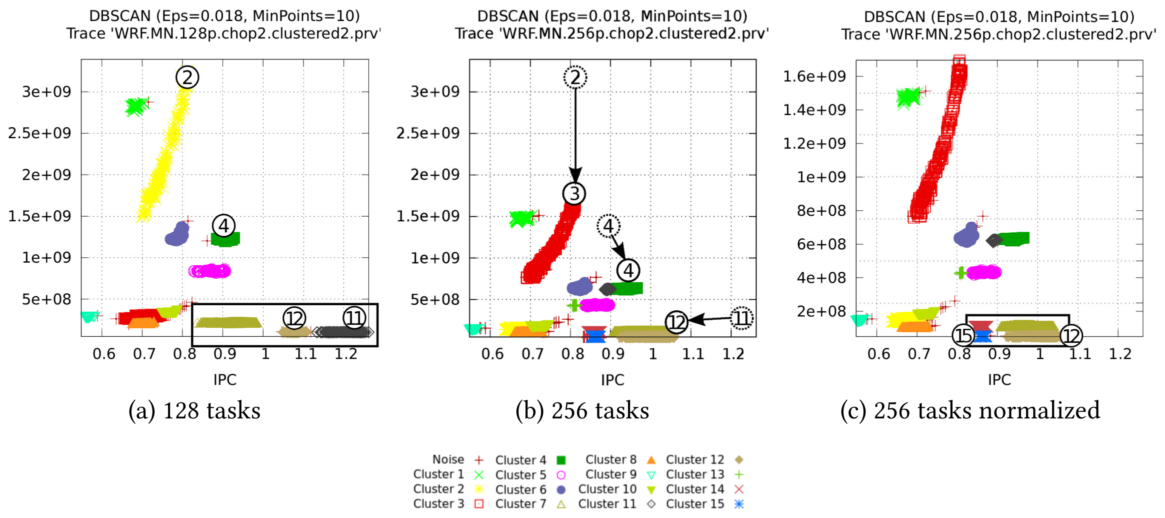


Figure 7.2.: Illustration of the difficulty of comparing the structure of the computing regions of WRF between two experiments with different scale

7.2.3. Tracking difficulties

The main difficulty in the use of tracking techniques arises due to abrupt object motions. Even though one would normally expect the application performance not to radically change all of a sudden, performance variations may result in large changes of behavior, preventing us from borrowing any assumption about the clusters' position, direction or shape in the performance space.

The clustering process of a frame assigns numbers and colors to every cluster identified. Since this is an independent, non-supervised process, the clustering of a second, different frame does not necessarily have to result in the same number of objects, assign the same identifiers, or exist a direct correspondence between their numberings. Figure 7.2a shows the structure of the twelve most time-consuming regions of WRF [164] ran with 128 processes. Clusters are formed according to similarities in the achieved performance (X-axis) and number of instructions (Y-axis). Those that stretch vertically (e.g. Region 2) denote instructions imbalance, while those that stretch horizontally (e.g. 7 and 11) reflect IPC variations. Figure 7.2b shows the structure of WRF doubling the number of cores. The number of instructions executed per core has reduced in inverse proportion, and so all clusters have moved downwards the Y-axis. Intuitively, we can see that cluster 2 (yellow) turned into 3 (red). And a few clusters have slightly improved their performance (e.g. 4 and 6 moved right with higher IPC), while cluster 11 significantly degraded. But some changes are far from evident: zooming into the boxed areas, you can see a fourth cluster appearing. Is that the left-most cluster in the 128-task case redistributed into the two small ones on the left of the 256-task case? Or these two come from split parts of the two left-most clusters?

With changing scenarios that may affect the application performance, clusters can not only move long distances or change their shape between frames, they can also vary in density, split, or merge together. And if the configurations that differentiate the experiments vary significantly, the frames to compare can be remarkably different, which makes even more difficult to detect the interesting regions and see how they change from one frame to the next. Although in some cases it would be possible to determine "who-is-who" by visual inspection, this will not be obvious in the general case, and so the benefits of an automated mechanism able to detect abrupt changes amongst many clusters become evident.

The first difficulty in determining which objects within a frame correspond to the ones in the next lies on the fact that the respective scales may be different, so they can not be compared directly. For example in a strong-scaling case, when the number of cores increases, the number of instructions executed per core will decrease in proportion. A step prior to track the evolution of the objects consists in normalizing the performance scales so that they are comparable. Such metrics that are correlated with the number of processes of the application (e.g. Instructions) are weighted by the number of cores, while the scale for the rest (e.g. IPC) is adjusted to the minimum and maximum values seen along all experiments. Figure 7.2c shows the 256-tasks case with the performance scales normalized. The relative distances compared to the base 128-tasks case are kept almost constant, and the experiments can now be easily compared.

In the next section we present a tracking algorithm that performs an automatic correlation of equivalent code regions that are subject to performance variations along multiple experiments. To this end, we extrapolate the concept of recognizing moving objects in a sequence of images to the displacement of clusters within the metrics space across experiments. Clustering the application performance can be seen as identifying objects (regions of code with a certain behavior) in a single frame. Subsequent clusterings result in a sequence of images that can be compared to see how these objects move, shape-shift, merge or split in the performance space, reflecting changes in the application behavior. Tracking their evolution across experiments enables us to study the performance characteristics of the different code regions, and to understand how the different configurations get to influence their behavior.

7.2.4. Implementation details

The current implementation uses the Extrae tracing tool [53] to automatically instrument MPI and/or OpenMP through library preloading techniques. For each entry and exit point of the parallel runtime, the tool stores a per-thread timestamped event trace, collecting hardware counters data through PAPI [22]; and source code references by using libunwind [28] to walk the call stack and GNU binutils [165] to fetch human-readable debugging information from the binary. In our experiments, the size of the traces generated ranged from tens of MB to tens of GB.

The clustering tool extracts the CPU bursts data comprised in the trace and runs a basic DBSCAN algorithm to identify the main computing trends. In this process, bursts with very short duration are considered negligible and discarded, so as to avoid the high cost of processing many small points. In [114], the authors prove that one can discard up to 80% of the data, while preserving the 99% of the computation representativity. This clustering tool can handle up to 100K points under 1-2 minutes.

As reported in the literature, tracing tools already scale to hundreds of thousands of cores [47], and parallel density-based algorithms are able to manage millions of points [166]. Once the data has been reduced to representative clusters in the performance image, the tracking algorithm presented next works with a very reduced number of objects, enabling low response times from few seconds to few minutes, and so the technique presented, relying on large-scale tracing and clustering tools, is perfectly applicable with large volumes of data and totally scalable.

7.3. The tracking algorithm

The objective of this algorithm is to automatically correlate equivalent computational components that are subject to performance variations, tracking how they move along a sequence of images that represent the application's performance behavior. Let A and B be two images, as depicted in Figure 7.3, where n and m objects are respectively detected, say $A = \{A_1, A_2, ..., A_n\}$ and $B = \{B_1, B_2, ..., B_m\}$. The objective is to find the maximum number of relations k, so that exists a k-partition $P = \{P_1, ..., P_k\}$ of A, and a k-partition

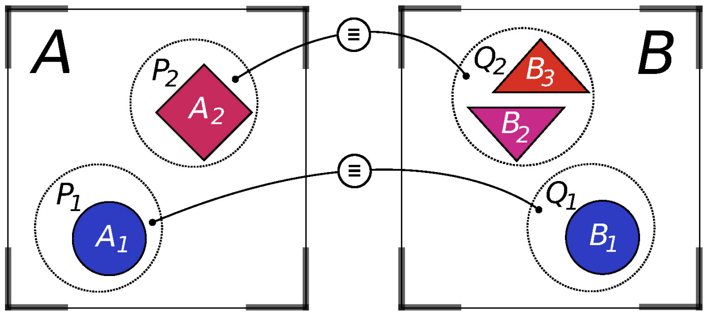


Figure 7.3.: Tracking algorithm scheme

$Q = \{Q_1, ..., Q_k\}$ of B, that fulfill the condition:

$$\forall i : 1 \leq i \leq k : P_i \equiv Q_i$$

Where the optimal k is bounded above by the image with the fewer number of objects detected, i.e. $min(n, m)$, and the equivalence relation $P_i \equiv Q_i$ is the assumption that objects in partition P_i correspond to those in partition Q_i.

In order to determine whether two clusters are equivalent, there are three principal properties of the computations that can be considered: the position in the performance image, the position in the source code, and the position in the execution trace. Based on these characteristics, we define five complementary heuristics to evaluate the clusters equivalences that are detailed in the next section.

7.3.1. The tracking heuristics

Recalling the difficulties to apply tracking on performance data that we previously explained in Section 7.2.3, deciding whether two clusters from different experiments represent the same computational behavior requires to consider different characteristics of the computations. In our implementation, each characteristic is evaluated with a different heuristic. Applying just a single heuristic is generally not enough, because as we will discuss throughout this section, most of the characteristics inspected are to some extent ambiguous and do not allow to perfectly differentiate between the objects. Moreover, not all the information required to apply all the heuristics is always present (that depends on the system and the amount of information collected during the tracing process). Therefore, we employ multiple heuristics and combine their results to decide the equivalences between all objects. Each heuristic focuses on a particular characteristic of the computations:

- *Direction of the movement.* Clusters can move in any direction of the space as a consequence of performance variations, but in the general case, these will manifest as smooth, directed transitions rather than swift leaps. For example, if we keep increasing the size of the workload, we can expect the total number of instructions executed in all computations to increase as well, and make certain assumptions on the directions of the movements.

- *SPMDiness.* In SPMD applications, all processes must be executing the same code phase simultaneously. If two different clusters happen at the same time, since the application is SPMD they can not refer to different code phases, and so they must be the same code phase that is presenting multi-modal behavior.
- *Call stack references.* Call stack information links every computation in the cluster to the point in the code where it is executed. Different clusters can not be the equivalent if the computations that form them do not share any call stack reference that points to the same point in the source code.
- *Clusters density.* If two experiments produce the same amount of computations, but they classify in different clusters because there are changes of behavior or splits, there must be a superset of the split clusters so that the sum of their computations equals the amount of computations in the equivalent unsplit cluster.
- *Chronological sequence.* Two experiments running the same program will show the same time-ordered sequence of computations, so those that appear in the same order of occurrence must be equivalent.

The following Sections 7.3.1 to 7.3.1 describe each of the heuristics in more detail. Then in Section 7.3.2 we explain how the information provided by the different heuristics is combined to maximize the number of objects successfully tracked.

Direction of the movement

This heuristic takes a pair of images and performs a cross-classification of every computing burst from the first into the latter, and vice versa. The classification is based on a nearest-neighbor criteria, so that all points will get classified to the nearest counterpart cluster. This can be seen as projecting each object from one image to the next, and see which object in the second image is closer.

The idea that lies behind supports on the fact that the behavior of a parallel application will not radically change along images, and so the objects displacements will generally be short. This assumes a certain ordering in the pairs of images that are compared, as the more different they are, the more difficult becomes to find correspondences. However, for the majority of analyses an implicit order emerges. Consider again the previous example where we doubled from 128 to 256 the number of cores in WRF (see Figures 7.2a and 7.2c). The resulting structure for both experiments hardly differs, with very slight movements.

There are situations where a cluster splits into two or more. For example, when new zones of imbalance appear and separate one region into two distinct behaviors. This case can be seen in Figure 7.4, where region A_4 shifts to two behaviors, namely B_4 and B_{11}. Also, there are cases where clusters can move a long way in the space, which is the case of regions 11 and 12 in Figure 7.2a to regions 12 and 15 in Figure 7.2c, respectively. In these situations, cross-classification based on distance is likely not to assign the points to the correct cluster (both get assigned to 12 because 15 is too far away, which illustrates a mapping error), but we can then use the next heuristics to discern whether those regions are the same or not.

$$\begin{array}{c|cccccccccc|}
 & B_1 & B_2 & B_3 & B_4 & B_5 & B_6 & \dots & B_{10} & B_{11} & B_{12} \\
\hline
A_1 & 100\% & 0 & 0 & 0 & 0 & 0 & & 0 & 0 & 0 \\
A_2 & 0 & 0 & 100\% & 0 & 0 & 0 & & 0 & 0 & 0 \\
A_3 & 0 & 99\% & 0 & 0 & 0 & 0 & & 1\% & 0 & 0 \\
A_4 & 0 & 0 & 0 & 34\% & 0 & 0 & & 0 & 65\% & 0 \\
A_5 & 0 & 0 & 0 & 0 & 100\% & 0 & & 0 & 0 & 0 \\
A_6 & 0 & 0 & 0 & 0 & 0 & 100\% & & 0 & 0 & 0 \\
\vdots & & & & & & & & & & \\
A_{11} & 0 & 0 & 0 & 0 & 0 & 0 & & 0 & 0 & 100\% \\
A_{12} & 0 & 0 & 0 & 0 & 0 & 0 & & 0 & 0 & 100\%
\end{array}$$

Figure 7.4.: Tracked correlations between WRF-128 and WRF-256 according to the movement-based heuristic

SPMDiness

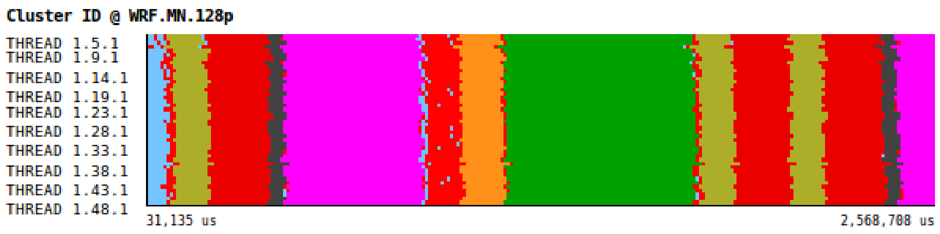
This heuristic exploits the SPMD structure of the applications to match computing regions that happen simultaneously in different processes. Assuming this execution model, all processors are expected to be executing the same phase of code at a time. In this case, if multiple processes are executing different types of computations concurrently, they are likely to refer to the same code region, although there might be performance variations that make them shift apart (e.g. the application presents work imbalance and some processes execute more instructions than others).

Figure 7.5a shows a detailed view of the temporal sequence of clusters at the beginning of one iteration of WRF 128-tasks. All processes (Y-axis) execute the same computations over time (X-axis). The same pattern can be seen in Figure 7.5b for the 256-tasks case, meaning that the code phases and the order in which they get executed are the same in both runs. However, in this case some processes are undergoing duration imbalances and execute longer computations, shown as stride lines with distinct colors. The new behavior is identified as a different cluster, but these are actually the same computing phases and can be linked together.

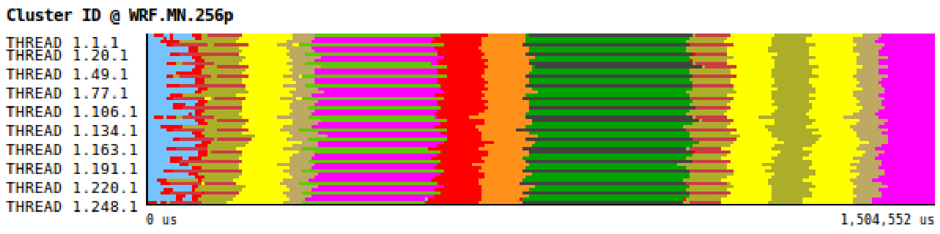
The application SPMDiness is evaluated with the technique presented in [159]. The algorithm takes as input the sequence of clusters for every task of the application, and performs a global sequence alignment. Clusters from different tasks that fall into the same position of the global sequence are getting executed simultaneously, and we use this information to mark them as equivalent.

Call stack references

This heuristic prunes the search space by discarding matchings between regions that do not have call stack references in common. Call stack information points to the function, file and source code line where the CPU burst starts, linking them to specific points of code. If two regions from two different frames do not share code references, they are certainly



(a) SPMD computations for WRF-128



(b) SPMD computations for WRF-256

Figure 7.5.: Tracked correlations between WRF-128 and WRF-256 according to the SPMD-iness heuristic

not equivalent.

Table 7.3 illustrates a subset of the relations that can be outlined between regions from their code references. The reason why some relations are ambiguous is because the clustering process groups computations based on their similarity with respect to selected performance metrics, so it is possible that different points of code behave the same and get grouped under the same cluster. Also, if a single code region presents different behaviors, it will also appear as part of multiple clusters. This information on its own is not enough to discriminate more, but effectively reduces the combinatorial explosion.

Clusters density

This heuristic is applicable when comparing different experiments that produce the same amount of data. In those cases, the aggregate of points of all clusters in each clustering will be the same. If the points distribution in the performance space does not change between experiments, the densities of the clusters will also be the same. When a cluster splits, two or more sub-clusters will have formed, and the sum of their densities will equal the density of the original super-cluster that contained them all, as illustrated in Figure 7.6.

This problem can be formulated as a variant of the 0/1 knapsack problem [167]: given a cluster from one experiment with a certain density D_A, find the combination of sub-clusters in the second experiment whose aggregate densities $D_B = D_B^1 + D_B^2 + \ldots + D_B^N$ are lower or equal to the limit density D_A.

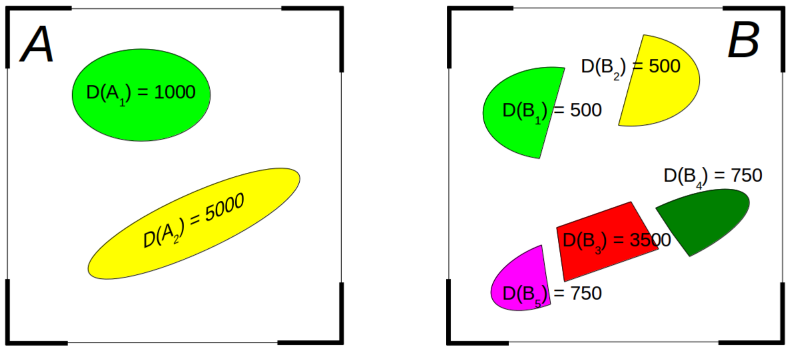
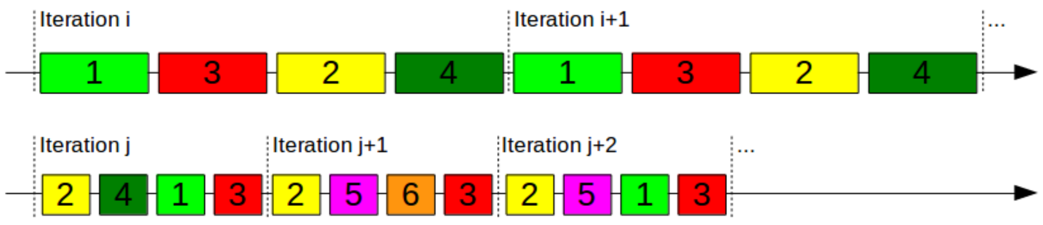
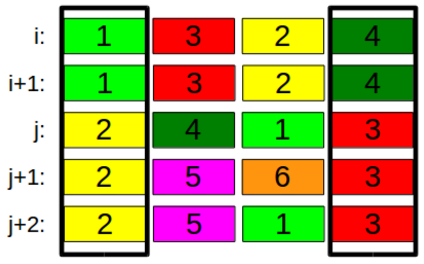


Figure 7.6.: Tracked correlations according to the clusters density heuristic. Aggregated density of clusters $B_1 + B_2 = A_1$, and $B_3 + B_4 + B_5 = A_2$.



(a) Sequence of computations in two different experiments



(b) Aligned subsequences between selected pivots, given that cluster 1 and 4 in the first sequence correspond to 2 and 3 in the second. Attending to their chronological order all clusters that fall the same column would be equivalent.

Figure 7.7.: Tracked correlations according to the chronological sequence heuristic

128 tasks	Callstack references	256 tasks
Region 1	4939 (module_comm_dm.f90)	Region 1
Region 2 Region 5	6474 (module_comm_dm.f90)	Region 3 Region 5 Region 13
Region 3	6060 (module_comm_dm.f90)	Region 2
Region 4	2472 (module_comm_dm.f90)	Region 4 Region 11
Region 7 Region 11 Region 12	5734 (module_comm_dm.f90) 6275 (module_comm_dm.f90)	Region 7 Region 12 Region 15

Table 7.3.: Tracked correlations between WRF-128 and WRF-256 according to the call stack heuristic

Chronological sequence

This heuristic assumes that, unless there are changes that alter the execution flow of the program, the code executed along different experiments will be the same, and so the sequence of computing bursts over time will preserve the same chronological order. Looking into the position where the computations appear in the sequences and matching those in the same position, it is possible to determine equivalent code regions.

The sequence alignment technique referred in [159] is applied now on two experiments, and we then compare the order of occurrence of the computations. For example, consider an experiment that executes a loop comprising 4 computing regions with different performance behavior, and so these get classified in 4 different clusters. The top timeline in Figure 7.7a depicts 2 iterations of this loop, with each computation colored according to the cluster to whom it belongs. A second experiment that uses more processes and a bigger problem size results in shorter computations and more iterations of the loop, as illustrated in the bottom timeline in Figure 7.7a.

As we have discussed earlier in Section 7.2.3, the clustering process applied to different experiments can result in different clusters, hence having the same clusters colors or identifiers does not necessarily imply that they represent the same computing region, and so these sequences can not be compared directly. However, if we could guarantee some correspondences between clusters, for example, that clusters 1 and 4 in the top experiment correspond to 2 and 3 in the second, then we can split the sequences between this points and align all the resulting subsequences, as shown in Figure 7.7b. Now if we only pay attention to the order of occurrence of the computations, all those that appear in the same column are equivalent with respect to their chronological order.

In order to decide which are points to split the sequences, this heuristic uses the matchings discovered so far by the previous heuristics to establish pivots in both sequences, and align the subsequences with respect to these points of reference to discover new matchings.

7.3.2. Combining tracking heuristics

Build upon the combination of these five heuristics, the tracking algorithm proceeds as follows to determine a global matching between all clusters. Every heuristic is applied separately and reports one or more correlation matrices representing relations between objects. Depending on the heuristic, what these matrices express is different. Figure 7.4 shows the correlations computed by the first heuristic for experiments WRF-128 (A) and WRF-256 (B). In this case, it indicates the percentage of computations that conform object A_i for which object B_j is closer. As you can see, there are cases where one object is close enough to two others or more, so it is not immediate to determine the appropriate correspondences when the objects are moving arbitrarily around the performance space. For the second heuristic one matrix per frame is built, each expressing the probability of two different computations to be executed at the same time by different processes within the same experiment. The third calculates the percentage of computations that are part of object A_i whose call stack references point to the same source code than those of object B_j. The fourth reflects the percentage of occurrence where computations A_i and B_j happen in the same chronological order. In the last case, the matrix represents combinations of clusters that have the same aggregate density. In all cases, non-zero cells evince that a given pair of objects are the same according to that heuristic, with a certain probability. Occurrences with a very small probability (5% by default) are neglected as outliers.

Since every heuristic considers different properties of the objects, they have to cooperate to complement the correspondences that a given one might fail to discern. To this end, the combination algorithm extracts from each correlation matrix a set of rules in the form $A_i \equiv B_j + B_k$, expressing which objects between two images are equivalent according to that heuristic. Some of the rules found can be contradictory (i.e. A_i and B_j are very close according to the *distance* heuristic, but they do not share any common references according to the *call stack* heuristic, so they can not be the same) or they can be complementary (i.e. A_i and B_j are very close according to *distance*, and B_j and B_k always appear together according to the *SPMDiness* heuristic, so they must merge). In order to combine all the equivalence rules from the different heuristics, the combination algorithm performs a series of union and intersection operations between them.

The first rules to take into account are always those found by *distance*, because the information required to compute the distances between objects is always present in the frames. Then the resulting rules are united with those found by *SPMDiness*. For example, if the first finds that the nearest object for A_5 is B_5, and the latter finds that B_5 and B_{13} always happen simultaneously, all objects merge into a more general relation $A_5 \equiv B_5 \cup B_{13}$. The *call stack* and *density* rules are then intersected to prune incorrect relations that may appear due to mapping errors in the former heuristics. The intersection operation can be seen as an agreement between heuristics: an equivalence betwen two objects is kept only if all the heuristics find that same correspondence, or discarded otherwise. For example, all related clusters must share the same references to the source code, so we discard those not having any in common.

We search for correspondences between objects reciprocally, this is to say, comparing frame A with B and vice versa, extracting a final set of rules that correlate the objects

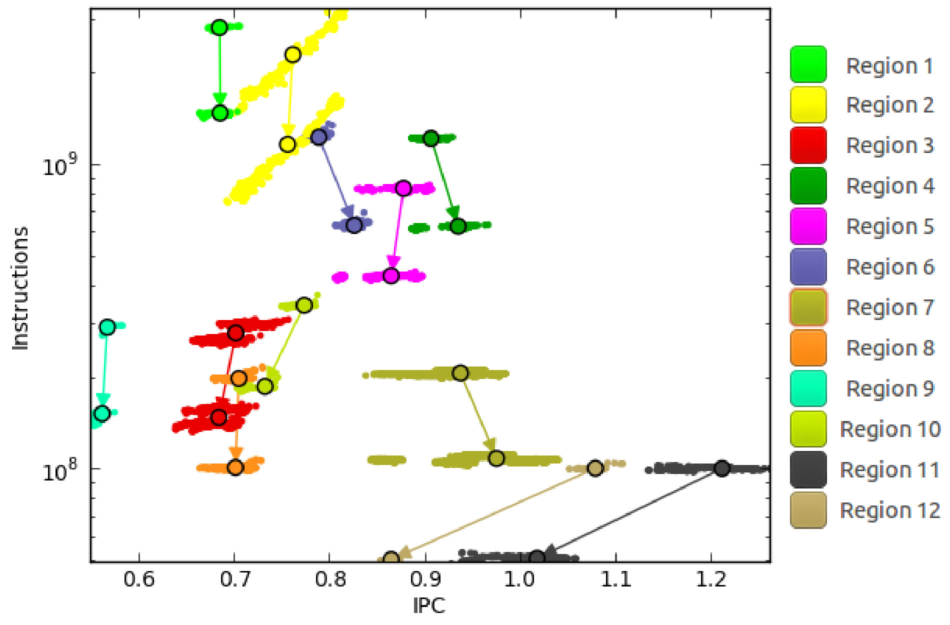


Figure 7.8.: Trajectory of clusters from WRF-128 to WRF-256

between both frames. When the information available leads the heuristics to not be able to clearly distinguish one region from another, the regions in doubt are grouped together, resulting in wide relations of multiple objects. The *execution sequence* heuristic is finally used to refine the results, splitting wide relations into more specific ones.

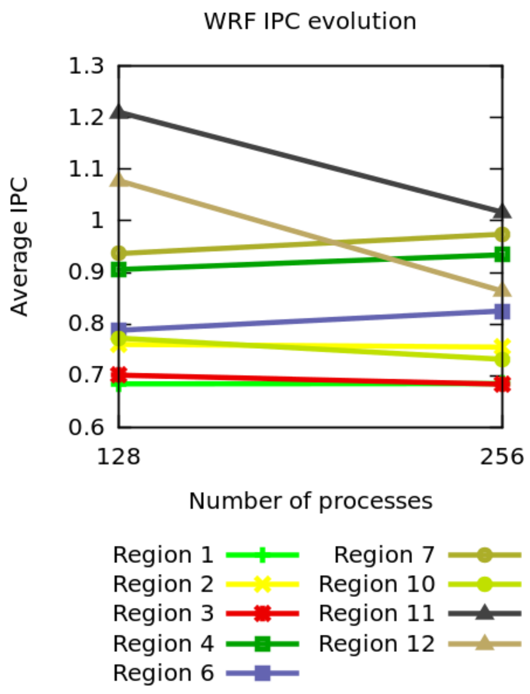
The analysis is repeated for every pair of consecutive frames, obtaining in the end k *tracked regions*, relations of objects that are equivalent along the whole sequence of images. Additionally, the tool generates plots describing the evolution of each *tracked region*. Next section gives an overview of the results of the tracking algorithm.

7.3.3. Tracking results

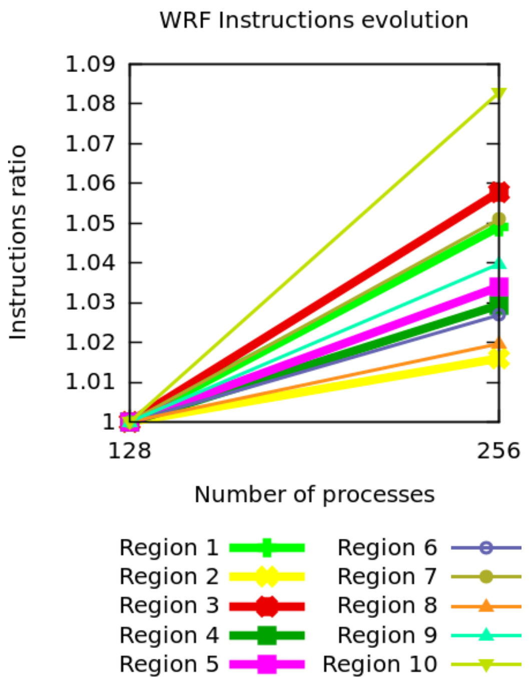
In this section we present the results of the tracking algorithm, following on from the WRF example used to guide the explanation of the technique through the former sections. For the two configurations presented, runs with 128 and 256 tasks, we will conduct a brief scalability study to explain how the tracking results yield practical insights that help in understanding and improving the code.

First, the tool reconstructs the input images for the tracking algorithm with all objects identifiers renamed, so that all equivalent regions keep the same numbering and color. The whole sequence of images can be displayed in a simple animation, or in a single plot showing the trajectory that every different object follows, so that is very easy to identify variations in the performance space, as shown in Figure 7.8 (in logarithmic scale for better readability, refer to Figure 7.2 for the real scales).

Here we can observe two main trends: clusters whose shape hardly varies between experiments (e.g. Regions 1 to 3), and those that become more distorted when the scale increases (e.g. Regions 4, 5 and 7). Focusing on the latter which are most affected by the



(a) IPC evolution



(b) Instructions evolution

Figure 7.9.: Performance trends for WRF code regions

scale, the developers made an effort to balance the amount of work, as they appear as flat clusters with low variation in the instructions axis. However, they present large IPC variability that increases at higher scale. In the 256-tasks case Regions 4, 5 and 7, that cover altogether the 30% of the total time, split into new zones of imbalance on their left with lower performance. Clusters becoming more disperse indicate an increasing problem of time imbalance.

Amongst the regions that do not deteriorate due to the scale increase, Region 2 stands out for covering all alone the 15% of the execution time, and exhibiting an elongated cluster in the Y-axis that reflects large instructions imbalance, within a dynamic range that doubles from 1.5e9 for the 128-tasks case (top), and 8e8 for the 256-tasks case (bottom). Despite the IPC variability partially compensates the instructions imbalance and the performance is maintained at scale, this region was already inefficient from the start.

In addition, the tool presents the evolution of every computing region from the first scenario to the last, with respect to the metrics selected to generate the images. Figure 7.9a shows a trend chart displaying the evolution in IPC for the 128 and 256-tasks runs of WRF. For better readability, only the most significant regions and those with higher IPC variations (above 3%) are depicted. While there is a slight improvement for regions 4, 6 and 7 under 4%, regions 10 to 12 present a sharp decline up to 20%. Regions 1 to 3 remain constant, yet is important to remark that being the most important computations covering 50% of the total time, these are also the ones achieving lower IPC around 0.70. Figure 7.9b shows the evolution in the number of instructions for the regions that execute the most, as the percentage over the 128-tasks base case. When the number of cores increases, so does the total number of instructions, revealing code replication below 8% in all regions of the program, which is reasonable but warns us about an increasingly detrimental effect at higher scales, in particular for regions 3 and 10.

For a production class application with a long-term development, a brief analysis of the clusters trajectories and the metrics trends has quickly diagnosed several performance weaknesses and potential problems at higher scales. In general, the information presented allows to perform parametric studies on the influence of different configurations, as well as to study the evolution of a single experiment over time, enabling an intuitive analysis that gets straight to the points of interest and their major causes of inefficiency. Having call stack references associated to every cluster, it is possible to connect the observed performance artifacts to specific points in the code and extract useful recommendations on which way to direct the optimization process.

7.4. Experimental validation

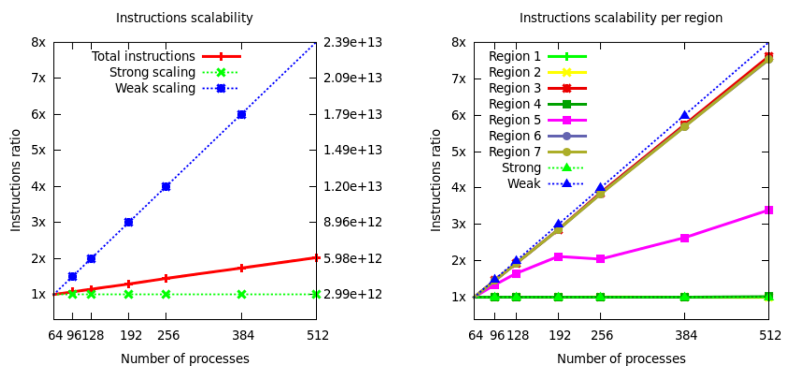
The aim of this section is to demonstrate the added value of using tracking, where the importance lays on understanding how and why the performance of the application changes along multiple experiments. We want to highlight the versatility of the technique for a variety of parametric studies, tossing ideas about the kind of cases of study that could be interesting for the analyst. To this end, we have selected configurations that would produce unpredictable sets of clusters and arbitrary displacements to prove the algorithm working under stress. Moreover, we present a real-case study to show that this technique can be useful to provide valuable insights to the users and successfully lead to improvements in their codes. Optimizing the applications is beyond the scope of this paper.

Therefore, a variety of proxy and production codes from different fields such as astrophysics, molecular dynamics and meteorology; were run in MareNostrum II, MareNostrum III and MinoTauro. MareNostrum II is a cluster of 2,560 nodes, each containing 2 IBM PowerPC 970MP 2-Core at 2.3 GHz with 8 GB of RAM. MareNostrum III comprises 3,028 nodes, each containing 2 Intel SandyBridge-EP E5-2670 8-Core at 2.6 GHz with 32 GB of RAM. MinoTauro comprises 126 nodes, each containing 2 Intel Xeon E5649 6-Core at 2.53 GHz with 24 GB of RAM.

Table 7.4 illustrates the ability of the algorithm to identify and keep track of the different computing regions in 11 studies. The objects detected are automatically reduced to the ones considered more relevant, those that represent a high percentage of the total application time, usually above 5-10%. *Coverage* is calculated as the percentage of objects tracked with respect to the maximum number of identifiable objects in the input images. 100% in *coverage* denotes that the algorithm has been able to find unambiguous correspondences between all the objects. Values below the optimal reflect that there were nearby objects in the input images that the tracking heuristics could not distinguish as separate individuals with the information available, grouping them as a single entity. On average, the algorithm successfully discriminates 90% of the objects. The following sections present five case studies in more detail.

Application	Input images	Tracked regions	Coverage
Gadget	2	8	88%
QuantumE	2	6	66%
WRF	2	12	100%
Gromacs	3	5	100%
CGPOP	4	2	66%
NAS BT	4	6	100%
OpenMX	7	7	100%
Hydro	8	3	100%
MR-Genesis	12	2	100%
NAS FT	15	2	100%
Gromacs	20	4	80%

Table 7.4.: Tracking experiments



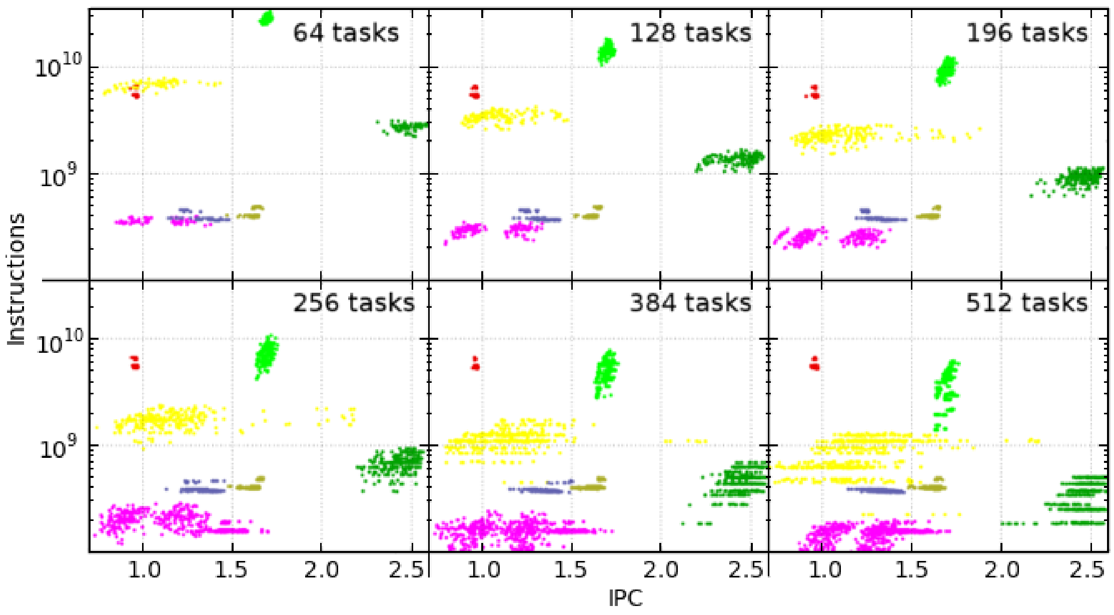
(a) Application scalability (b) Computations scalability

Figure 7.10.: Scalability of OpenMX

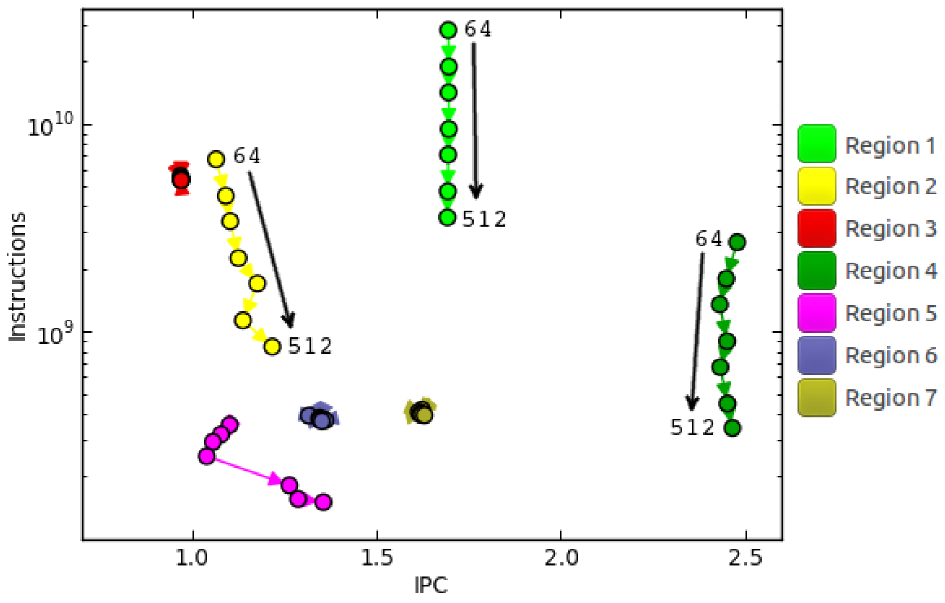
7.4.1. Studying the scalability of the computing regions

The objective of this experiment is to conduct a real-case study of the scalability of the computing regions of an application. The selected code is OpenMX [168], a software package designed for the realization of large-scale ab initio calculations. To that end, we run OpenMX v3.6p1 in MareNostrum III increasing the number of MPI tasks from 64 to 512 using a single OpenMP thread per task.

As we are running a strong-scale test (fixed-size problem on a varying number of processors), the application would ideally see the execution time reduced inversely proportional to the number of processors used. However, multiplying by 8 the number of tasks, the speedup achieved in a single time-step is lower than 2. In terms of work executed, the total number of instructions should have got evenly distributed amongst all processes, and thus remain constant when the scale increases. Withal, Figure 7.10a shows the total number of instructions increasing by 100% from the 64 to the 512-tasks case, which is far from the ideal scaling and too significant to be due to a problem of code replication. Ap-



(a) Sequence of output images from the tracking algorithm



(b) Trajectory of clusters from 64 to 512-tasks runs

Figure 7.11.: Tracking results for OpenMX

plying tracking, we can now break-down this aggregate for the whole program and study the evolution of the relevant code regions per separate, to understand which parts prevent the application from scaling better.

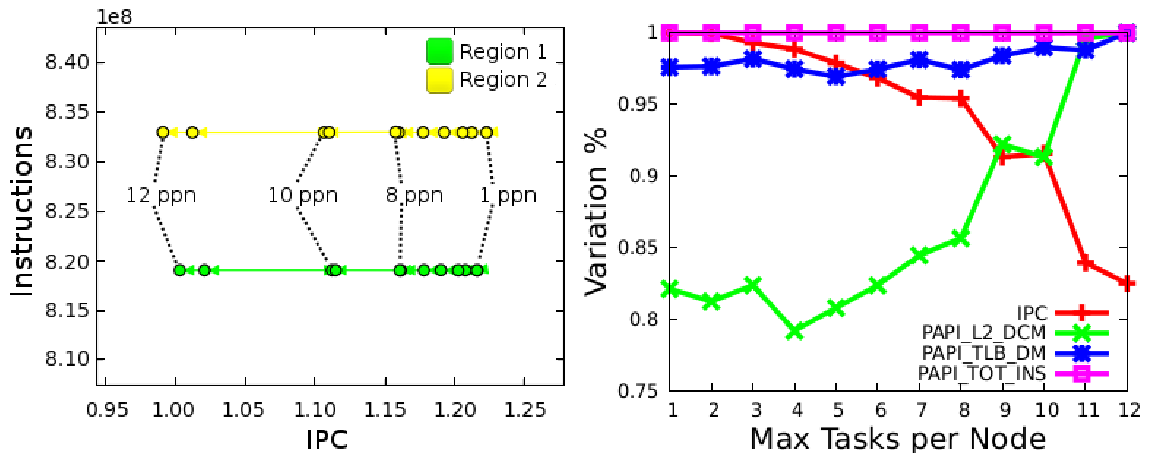
The input to the tracking algorithm is the collection of images that depict the performance of each individual experiment. Unlike in other experiments where the images are two-dimensional (Instructions and IPC), in this case we used the metric *L1 data cache misses* as a third dimension to cluster the data, which results in a more precise characterization of the relevant computational behaviors. Figure 7.11a shows the result of the tracking algorithm applied to the sequence of experiments from 64 to 512 tasks (only 6 out of 7 depicted due to space constraints, and plotted in 2D for clarity).

A quick glance at the evolution of the main behaviors reveals two main issues: First, most regions progress vertically downwards the Y-axis (instructions decrease), as one would expect for a strong-scaling case. Figure 7.11b shows the trajectories that follow the different regions from one experiment to the next, represented by their centroids. It is easy to see that regions 3, 6 and 7 do not move, meaning that they perform constant work despite the scale, as if they were ran in a weak-scaling mode.

Figure 7.10b shows the ratio of surplus work executed per region with respect to the ideal case where all regions scaled perfectly. In the 512-tasks case, regions 3, 6 and 7 which should have seen reduced their work by a factor of 8, actually execute 7.5 times more work than the expected. With this progression, these three regions that represented altogether the 20% of the iteration time in the 64-tasks case, now dominate the iteration representing the 65% of the total time, and have become the main bottlenecks to the computation scalability. Namely, these correspond to the computing phases starting at lines 289, 589 and 129 of routine Set_XC_Grid. Here, the programmer has put effort to use shared memory programming, but has not taken advantage of distributing the workload amongst processes. Likely, the developers considered more efficient to replicate this code to avoid the cost of communications, which may be worthwhile at small scales, but the increasing costs do not pay off at larger scales. These observations were reported to the developers, suggesting to study the feasibility of partitioning the work so as to fully exploit the distributed resources.

The second important observation is that most behaviors grow more and more disperse. In particular, it is the regions that scale better the ones that present more variability, namely 1, 2 and 4. The *parallel efficiency* [143] of these regions decreases from 0.80 in the 64-tasks case to 0.60 in the 512-tasks case, meaning that the 40% of resources are wasted due to time imbalances, where some processes have to wait for others to finish their work, and such imbalance gets absorbed in subsequent synchronizations. These correspond to the computing phases starting at lines 732 of Krylov_Col, 256 of Set_Hamiltonian, and 288 of Set_Density_Grid. In this case, a second precise recommendation could be made to the user to study the load-balancing characteristics of these particular regions.

As a final remark, the detected hazards could have been inferred just from the first three frames in the sequence, and so our technique can be used with few cores to anticipate problems at higher scales, saving on time and resources.



(a) Clusters trajectories mapping from 1 to 12 processes per node

(b) Region 1 evolution

Figure 7.12.: Tracking results for MR-Genesis

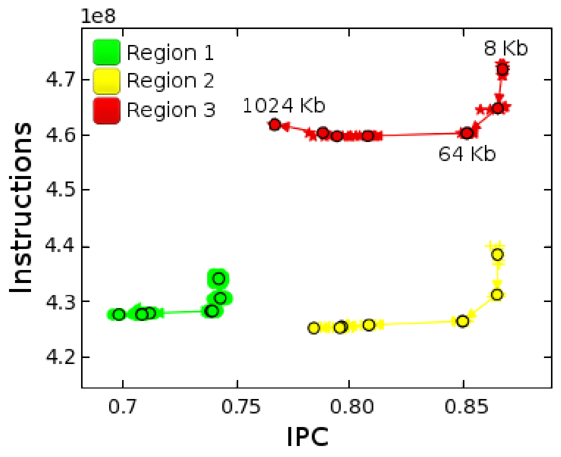
7.4.2. Studying the impact of multi-core sharing

MR-Genesis [169] employs a finite volume approach in order to evolve the Relativistic Euler equations combined with a Constrained Transport scheme to account for the divergence free evolution of the dynamically included magnetic field. MR-Genesis was run in MinoTauro using 12 processes, changing the maximum number of processes allowed per node from 1 to 12. Being 12 the number of available cores per node in MinoTauro, the configuration for the first experiment corresponds to 12 different nodes running a single process each, and a single full node for the last experiment, with all the intermediate cases also tested. The objective is to study the effect of memory bandwidth and caches contention on the application performance when sharing resources.

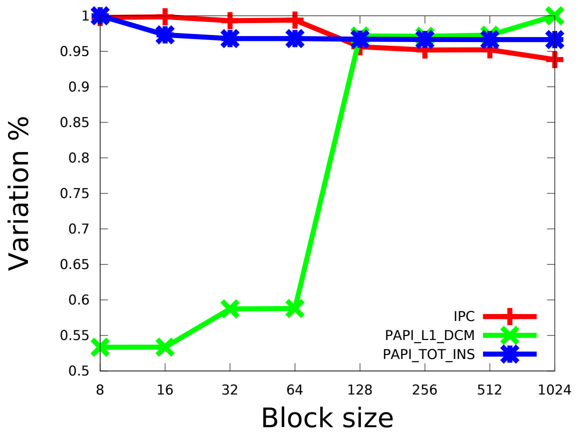
Figure 7.12a shows the result of the tracking algorithm applied to the sequence of experiments from 1 to 12 processes per node, which reveals two main computing phases with analogous behavior. Since it is only the physical mapping of processes what changes, the total number of instructions executed remains constant in all trials. However, as nodes get more populated, the achieved performance of the application decreases. Up to the 66% of the node occupation (8 tasks per node) the IPC presents a slight reduction under 1.5% from one experiment to the next, but starts presenting sharper drops beyond this point, with an 8.5% loss when an additional process is collocated in the node. Overall, the achieved IPC degrades a total of 17.5% when the node is full.

Figure 7.12b correlates all performance metrics for Region 1. The Y-axis reflects the percentage of variation of each metric with respect to its maximum value for all trials. The number of L2 cache misses grows inversely to the IPC degradation rate, and the TLB misses also increase as the node gets more populated.

In this case, a fair trade-off between maximum utilization of the resources and the application performance is met at two-thirds of the node occupation.



(a) Clusters trajectories doubling the block size from 8 to 1024 KB



(b) Region 1 evolution

Figure 7.13.: Tracking results for Hydro

7.4.3. Studying the impact of the program block size

HYDRO [170] is a proxy benchmark that solves a large scale structure and galaxy formation problem using a rectangular 2D space domain split in blocks. HYDRO was run in MinoTauro, and the sequence of images in this case is built doubling the block size from 8 to 1024 Kb. The objective of this experiment is to determine which is the best setting for a particular parameter of the program to minimize the execution time.

Figure 7.13a shows the evolution of the three main computing phases of the application, which actually refer to the same source code region with tri-modal behavior. The trajectories reflect the number of instructions initially decreasing for all three regions with drops from 1% to 3% up to a block size of 32 (movement downwards the Y-axis), and keeps steady beyond this point. IPC also decreases with a total deviation of 5% for Region 1, and 10% for Regions 2 and 3, all presenting a sharp dip when the block size increases from 64 to 128 (movement leftwards the X-axis). At this point, the number of L1 data cache misses rockets 40% more, as shown in Figure 7.13b.

Using small block sizes the application gets more blocks to compute, which entails executing more control instructions. Since the blocks are bi-dimensional and store 8-bytes elements, when the block size is set to 64 the limit of the L1 cache is reached, which is 32 KB. With bigger sizes, the block does not fit in the cache, and so the miss rate increases to the detriment of IPC.

Correlating the evolution of all metrics, the point where highest performance and lowest workload and cache misses converge is at a block size of 16, which results in the fastest execution of all proposed setups, so this one would be the most recommendable to minimize the response time.

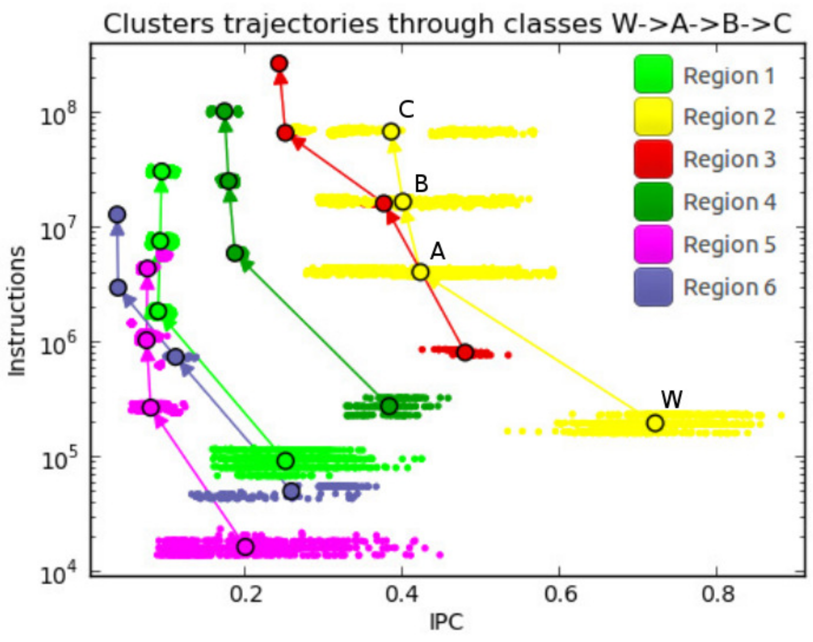


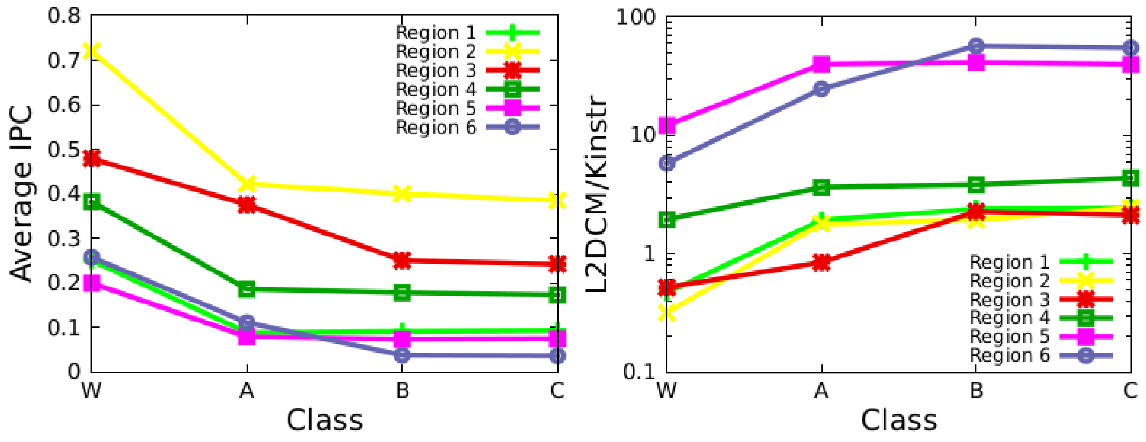
Figure 7.14.: Trajectory of clusters through BT experiments

7.4.4. Studying the impact of the problem input size

The NAS Parallel Benchmarks [141] are a small set of programs designed to assess the performance of parallel supercomputers. In this experiment we evaluate version 2.3 of the BT solver with increasing problem sizes. Problem sizes are predefined and indicated as different classes, where Class W corresponds to a small workstation problem size, and A, B and C correspond to standard test problems with a 4X size increase going from one class to the next. For all classes, BT was run in MareNostrum II with 16 processes.

Figure 7.14 shows the trajectories of the clusters through classes W to C. The starting experiment corresponds to Class W, which can be located at the bottom part of the plot. Class W presents large variability in IPC, which is depicted with the elongated clusters in the X-axis. As the experiments move forward, all clusters move to the top-left part of the plot. This transition shows a large dynamic increase of two orders of magnitude in the number of instructions from Class W to Class C. Also, clusters become more compact, indicating a reduction in the IPC variability except for Region 2, which corresponds to the Gaussian elimination performed in routines $[x|y|z]\_solve\_cell$.

In contrast, the achieved performance in all code regions degrades as the size of the problem increases. Figure 7.15a shows there are two decreasing trends for the IPC. For regions 1, 2, 4 and 5, a sharp loss ranging from 40% to 65% happens as soon as we move from Class W to A and then stabilizes, while for regions 3 and 6 the IPC keeps decreasing and does not stabilize until Class B. Correlating the evolution of all available metrics, we can see that this IPC degradation can be explained due to an increase in the L2 data cache misses, as shown in Figure 7.15b.



(a) Evolution of IPC (b) Evolution of L2 Cache Misses

Figure 7.15.: Performance trends for NAS BT code regions

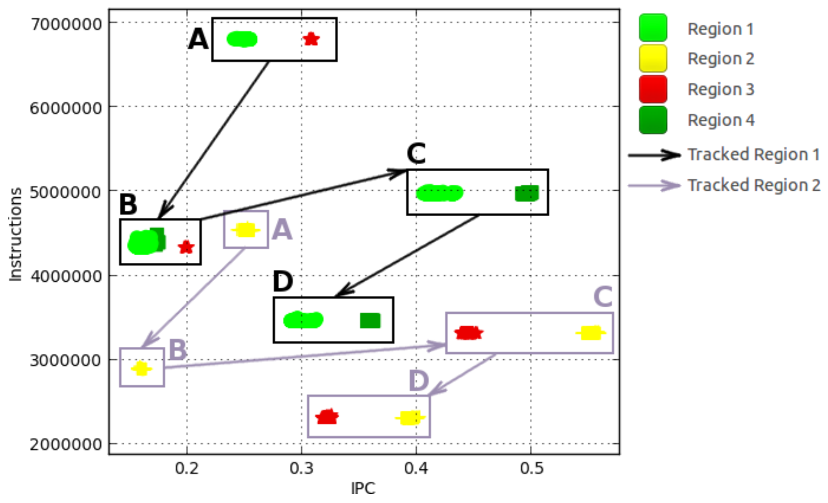
7.4.5. Studying the impact of different hardware and compilers

In this experiment we are going to stress the performance variations in the application changing machines and also changing from a generic to an architecture specific compiler. This test shows that even in very different scenarios, the tracking algorithm is able to follow the evolution of the clusters.

CGPOP [171] is a proxy application of the Parallel Ocean Program [172]. POP simulates the global climate model and is a component of the Community Earth System Model. CGPOP was run with 128 processors both in MareNostrum II and MinoTauro, and compiled with GNU Fortran 4.1.2 (gfortran) and IBM XL Fortran 12.1 (xlf) in MareNostrum, and GNU Fortran 4.4.4 and Intel Fortran 12.0.4 (ifort) in MinoTauro. In all cases, the application was compiled with the optimization flag -O3 and debug.

Figure 7.16 shows the trajectories that follow the two main computing behaviors with respect to the number of instructions, which are subdivided into several regions due to differences in the achieved IPC. In MareNostrum, when the application is compiled with xlf (see 7.16b) all computations see the number of instructions significantly reduced (36% and 33%, respectively) compared to using gfortran (see 7.16a), but the IPC degrades practically in the same proportion and the overall execution time remains almost constant. The situation in MinoTauro is very similar (see 7.16c and 7.16d), with an overall improvement in terms of less instructions executed and higher IPC achieved, yet the same effect when changing compilers can be easily identified.

Changing the platform also alters the behavior of code, as can be seen for Region 2 in MareNostrum which splits into Regions 2 and 3 in MinoTauro, no matter the compiler used. They all refer to the same point in the code, but it now presents two distinct behaviors. The tracking algorithm automatically identifies and groups together those regions that are equivalent despite the performance variations, as illustrated by the bounding boxes, and then numerically calculates their evolution along experiments. Table 7.5 summarizes the averages for IPC and instructions for both tracked regions, and their elapsed execution



(A) MareNostrum GNU (B) MareNostrum IBM XL (C) MinoTauro GNU (D) MinoTauro Intel.

Figure 7.16.: Trajectory of clusters through CGPOP experiments

		MareNostrum		MinoTauro	
		gfortran	xlf	gfortran	ifort
Region 1	IPC	0.25	0.16	0.42	0.30
	Instructions	6.8M	4.3M	5M	3.5M
	Duration	12.09s	12.11s	4.82s	4.68s
Region 2	IPC	0.25	0.16	0.50	0.36
	Instructions	4.5M	3M	3.3M	2.3M
	Duration	2.13s	2.14s	0.71s	0.69s

Table 7.5.: Performance results for CGPOP using different compilers

time.

In this case, the specialized compilers xlf and ifort attain a reduction of 36% and 30% of the number of instructions with respect to gfortran in both machines, but at the expense of an average IPC loss of 36% in MareNostrum and 28% in MinoTauro. Likely, they reduced index arithmetic through reassociation. However, the performance did not change much because the computation is still memory bound. The integer instructions saved were likely traded for idle issue slots while waiting for the memory hierarchy, leading to negligible variations in the execution times lower than $\pm 0.03\%$.

7.4.6. Studying a weak scalability problem

The HACC (Hardware/Hybrid Accelerated Cosmology Code) is a framework that melds particle and grid methods to satisfy the requirements of cosmological surveys, exploiting hybrid and accelerator-based architectures with millions of cores, including CPU/GPU,

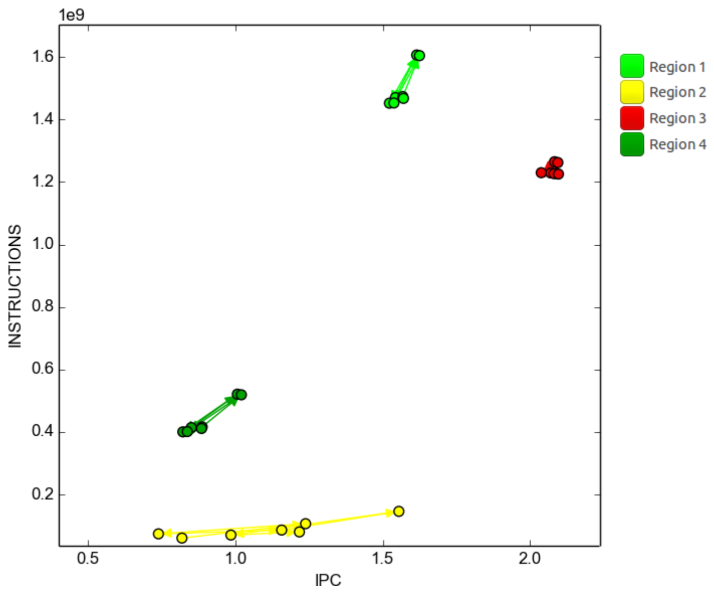


Figure 7.17.: Trajectories of clusters through HACC weak scale experiments

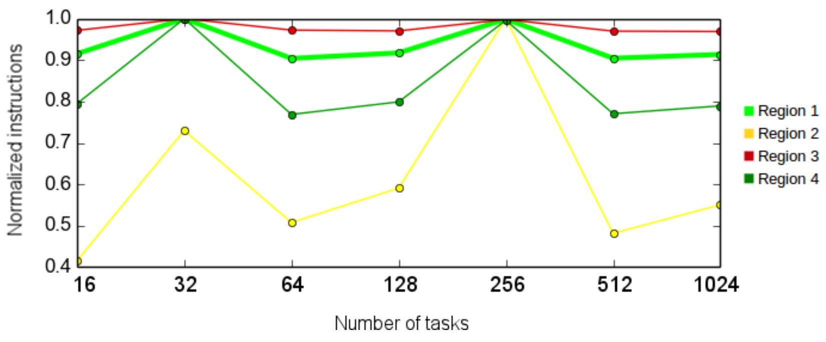


Figure 7.18.: Average instructions executed per computing region in HACC

multi/many-core, and Blue Gene systems.

HACC is designed to scale weakly by dividing the work in cubes. In this experiment we stressed the application setting different geometries other than a perfect cube, in order to see how much is the performance affected. The program was run in MareNostrum, doubling the number of tasks from 16 to 1024 tasks, as well as the size of the problem, with 1 single MPI task per node (so neither multi-core nor L1 to L3 caches sharing), using the Intel MPI message passing library, and without support for threads.

Figure 7.17 shows the trajectories of the main computing regions of the application. Here we can observe diagonal movements back and forth: as we increase the number of tasks (and so the size of the problem in proportion), all regions move upward (more instructions executed) and rightward (more IPC achieved). However, when the number of tasks is cubic (i.e. 64 and 512 tasks), the regions move back in the opposite direction (down and left; meaning less instructions executed and less IPC achieved). Figures 7.18 and 7.19 show this effect more clearly. Figure 7.18 shows the amount of instructions executed per region across experiments. The lower workload is found at experiments 3 and 6 (64 and

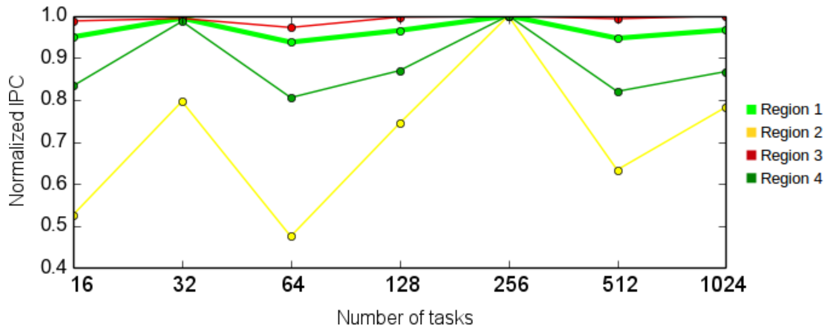


Figure 7.19.: Average IPC achieved per computing region in HACC

512 tasks). Correlating with Figure 7.19, these two experiments are also the ones achieving lower IPC.

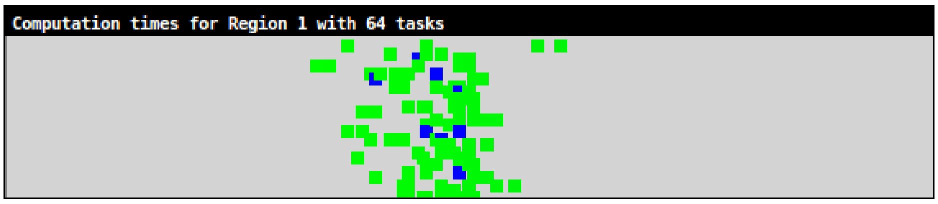
The differences in the number of instructions can be explained due to the work distribution scheme: when the number of tasks is not cubic there is extra work to distribute among the available tasks. Although the IPC achieved also becomes higher, the increase in performance does not compensate for the increase of work, and the computation time becomes higher in the uneven cases. This can be seen in Figure 7.20b, that compares the computation times for the main computing region of the program in the cubic runs (64 and 512 tasks), and an uneven intermediate case (256 tasks). In these histograms, the rows and processes and the columns are bins of computation durations increasing from left to right. In the cubic cases 7.20a and 7.20c, the computing times are very similar in the range of 310 to 323 ms, but in the 256 tasks case shown in Figure 7.20b, all computations are shifted to the right, having increased times ranging from 323 to 343 ms.

Even though the overall performance of the computations is better in the cubic cases, we can also observe that the time to solution degrades as we increase the scale. Comparing the two cubic cases with 64 and 512 tasks, we can see that the percentage of time spent in computations decreases from 60 to 45%. The problem in this case resides in the complementary communications. In particular, the time spent in MPI_Wait calls rockets from 30 to 50% because of the serializations in the program caused by a pipelined communication pattern, where some processes can not progress until they have received messages they are waiting on. One recommendation that could be given in this case to improve the scalability of the program is to change the communication pattern so as to overlap computations and communications, reducing the serializations.

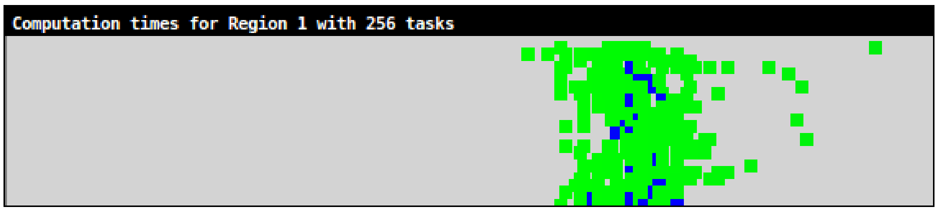
7.5. Related work

Our work draws inspiration from a motion detection algorithm of moving biological objects that are similar but non-homogeneous [173]. They apply multi-feature contour segmentation and flux tensors to identify the boundaries of biological objects and detect deformable motion and complex behaviors (e.g. cell crawling or division) along a time-lapse collection of images.

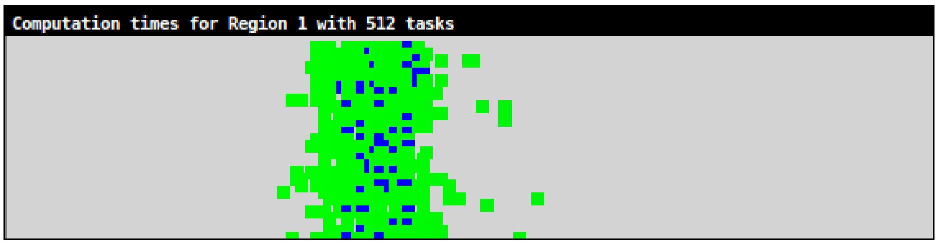
In a broader sense, object tracking is applied in the context of applications that require to



(a) Experiment using 64 tasks



(b) Experiment using 256 tasks



(c) Experiment using 512 tasks

Figure 7.20.: Histograms of computations duration for Cluster 1 in HACC. The data points in the 256-task case that appear more to the right indicate that the performance of the computations decreases in experiments with non-cubic task distribution.

associate target objects in consecutive frames to detect how they move around the scene. Practical applications include: automated surveillance, gesture recognition, traffic monitoring or path planning. [174] presents an extensive review of the state-of-the-art of tracking methods, and discusses related issues including the use of appropriate image features, motion models and object recognition.

ETRUSCA [175] included a jitter reduction analysis that attempted to relate the clusters found in one time interval with the clusters found in the next interval. Selecting a representative process in each interval, they would minimize the data captured. Our approach does not look for representative processes, but representative behaviors for all computing phases within all processors, and track how they change not only across time intervals, but also across experiments with different configurations.

Multi-experimental analysis has been approached by several performance analysis tools. SCALASCA [126] includes a tool called performance algebra that can be used to merge, subtract, and average the data from different experiments and view the results as a single derived experiment. PerfExplorer [113] supports data mining analyses on multi-experiment parallel performance profiles. Its capabilities include general statistical analysis of performance data, dimension reduction, clustering and correlation of performance data, and multi-experiment data query and management. TAU [45] incorporates the concept of phase profiling for the study of the evolution within a single experiment. This is an approach to profiling that measures performance relative to a phase of execution, having its entry and exit marked by the user. HPCToolkit [41] merges profile data from multiple performance experiments into a database file and perform various statistical and comparative analyses.

While they compute averages for predefined metrics and fixed phases such as functions, iterations or sections marked beforehand, we report arbitrary metrics at the level of computing regions. By doing so, we abstract the structure of the application to the behavior of its computing phases, taking into account the performance measurements of every single computation rather than profiled averages that may hinder their actual behavior.

Studies have also used non-predefined program constructs to characterize the program behavior. In [176], BBV-based analysis is applied to detect phases within an execution. This lowers the granularity to the instruction-level, which moves away from the semantics of the code and is not of common use for the analysis of parallel production codes. Further discussion on the suitability of CPU bursts to characterize the program behavior can be found in [114].

The fundamental difference that distinguishes our approach from the previous ones is that we do not merely report the outcome of different experiments together. We automatically determine the regions of interest and track their evolution along multiple executions. To this end, we translate performance data from different execution scenarios into a sequence of images, detect structure in each image and automatically correlate them.

7.6. Closing remarks

In this chapter we have demonstrated that it is possible to draw an analogy between tracking techniques applied to the automatic detection of an object's motility, and the performance analysis of a parallel application's evolution along multiple execution scenarios. This approach mimics the common phase structure of a tracking algorithm, including the generation of a sequence of images, object recognition within each frame and motion analysis across scenes.

Different scenarios are represented as a sequence of performance images that expresses the evolution of the application either along different experiments with changing configurations, or along time intervals within the same experiment. Computing regions of the application are represented as objects in these images, described by how they behave in terms of selected performance metrics. Then, we find a correspondence between objects along the whole sequence of images, keeping track of their possible motions and structural changes due to performance variations. To this end, we use a variety of heuristics that take into account different characteristics of the computing regions: the displacements in the performance space, the SPMDiness of the application, the code region they refer to, and the execution sequence. Combining their use, we are able to automatically identify the global evolution of the main computational behaviors and illustrate their performance trends.

Our technique offers a different viewpoint to the task of analysis that is more agnostic of the syntax of the code, but brings into focus the main performance characteristics of the program and the nature of their inefficiencies, enabling the identification of the most appropriate solution for the artifacts observed. Then, these observations can be correlated with the source code, to know which sections exhibit a given behavior. There are two remarkable benefits to this approach. First, the same solution can be applied to multiple code sections that present the same deficiency, without having to reappraise the same problems repeatedly. Second, we are able to detect multi-modal behavior and variations along time and processors, two important effects often masked by profiling tools. In this way, a single code section undergoing performance variabilities will be expressed as divergent behaviors that can be studied separately, revealing more room for improvement.

All in all, this work presents a versatile tool applicable in very varied scenarios, enabling the analyst to study the impact of virtually any configuration on the application performance without prior knowledge of the program; compare and correlate performance data between experiments; determine the best setup to meet specific performance requirements; and ultimately helps to gain better understanding of the application behavior, much beyond what can be learned from a single experiment.

This work opens up interesting lines of future research. On one hand, predictive models could be built next that would enable us to foresee the performance of experiments beyond the sample space. On the other hand, further on-line integration could be developed, in order to analyze the evolution over time of adaptative applications automatically.

Chapter 8

Active measurement techniques to improve the productivity of the analysis

Current interest in multi-experiment performance analysis is highly motivated by several purposes: benchmarking, procurement evaluation, modeling, prediction or application optimization. However, in-depth analysis of a variety of experimentation and evaluation scenarios poses two fundamental problems. First, and more importantly, it becomes essential to have sophisticated tools to store, process and draw useful conclusions from the multiple performance datasets generated from the different experiments. Second, even if you have these tools, recurring executions of a large-scale parallel program results very quickly in in an enormous expense of computing resources, energy and time; and the users happen to be often restrained by narrow computing budget allocations.

In previous chapters, we have thoroughly discussed on the importance of leveraging techniques from data mining, machine learning and statistics to focus the analysis on the zones of real interest, and so we have successfully applied cluster and spectral analysis to detect the application structure and identify behavioral archetypes. But being the applications eminently repetitive, if we focus the analysis just on a few representative regions, a lot of performance data is underused, and the computing power consumed to produce and process this data becomes essentially wasted. However, if the conditions of the execution changed over time, the application would be exposed to different scenarios, enabling us to evaluate the performance behavior under different circumstances all under the same run.

Active measurement techniques actively interfere the application's use of key resources like memory capacity and bandwidth, network bandwidth and computing power. Then, the application's sensitivity to reduced resource availability is measured by observing the effect of interference on the application performance. In this chapter, we combine this kind of techniques with all the technology that we have developed for the on-line analysis of parallel applications, in order to add controlled interferences to the program so as to emulate different execution scenarios at different time intervals of the same run. Then, tracking can be applied to analyze the evolution of the program behavior across the different scenarios. Overall, we condense all the information that would be extracted from multiple experiments in just a single run, minimizing the amount of resources necessary to produce all the required data for an effective performance analysis.

8.1. Background and motivation

In the previous chapter, we have seen that performance evaluation of parallel programs benefits from analyzing data taken from multiple experiments and different execution scenarios. This allows us to understand and anticipate the computation and communication needs in applications, identify current and future bottlenecks, drive program optimizations, and assess their effectiveness.

Three of the most common approaches to test the program performance under different scenarios are (i) analytical models, which characterize the application behavior under several known environments and enable to extrapolate the results to a new scenario [177]; (ii) parameter sweep executions, which allow to explore the real effects produced on the performance by different settings of the algorithm [178]; and (iii) simulations, which enable to study how the program would behave under different conditions [179]. While the solutions are different, all three approaches share the necessity of executing the program several times or running one simulation per scenario, and this entails two main shortcomings.

The first handicap is directly related to the accessibility to the computing resources. Large-scale parallel executions may use a large number of cores and the execution time may be long, so each experiment is costly to run. Moreover, the access to computing resources is often limited by job scheduling policies, computing time budgets and shared allocations, which taken altogether often results in long wait times and overly delayed results.

The second issue is related to the optimal exploitation of the resources used. Our previous research has focused on leveraging clustering and signal processing techniques to identify relevant phases of the program, reducing all the performance measurements taken throughout the whole execution to small representative areas. Since the applications are very repetitive, this actually results in a big portion of the measurements to be discarded because they are essentially redundant and do not provide any additional information. Thus the computing resources devoted to produce all this unused performance data are basically wasted, especially when many experiments are required and the user has to pay every time for the full cost of going through the wait lines to get all the necessary resources and make another full run.

To overcome these limitations, we could take advantage of the repetitive behavior of an application to expose it to many different scenarios during different time intervals of the same execution. In this way, we could extract from a single run all the necessary performance data that would otherwise require to perform multiple independent experiments.

To this end, active measurement techniques [180] emulate the behavior of anticipated future architectures on current machines. In particular, they focus on the impact of resource limitations by artificially restricting them, enabling to capture low-level execution details in real time to observe the application behavior under different scenarios. In this work, we combine the use of active measurement techniques with on-line techniques to detect the application structure at run-time, using the detected repetitive program phases to simulate different machine conditions at selected execution intervals. Then, tracking-based comparative analysis [9] can be applied to study how the program behavior evolves through the different simulated scenarios. The benefits are twofold: on one hand, just a single execution is needed to perform a multi-experiment analysis, and so the resources become more accessible; and on the other hand, less measurements are wasted because of being redundant, and so the productivity of the resources increases.

Combining the use of all these techniques on-line, it is possible to go one step further and use the results of the analysis to provide dynamic feedback to the execution run-time to improve the application performance. In this work, we started this line of research by integrating the on-line analysis framework with the OmpSs parallel run-time [21], adding the capability to give hints about the most optimal strategy to balance threads and tasks.

8.2. On-line application of active measurement techniques

In this Section we describe the integration of the on-line analysis framework with the GREMLINS framework [180], a collection of modules that are loaded into the execution environment of the target MPI application, to emulate different machine conditions in a single experiment. Then we describe how to extrapolate this idea to make the on-line analysis framework interact directly with the underlying parallel run-time, enabling to make dynamic adjustments to the execution. We show two different experiments to demonstrate the utility of studying which resources are more critical to the application, and which is the most optimal strategy to balance threads and tasks.

The experiments were run in Marenostrum 3 [181], a cluster of 3,056 compute nodes based on dual Intel SandyBridge 8-core processors at 2.6 GHz, iDataPlex Compute Racks, a Linux operating system and an Infiniband interconnection network.

8.2.1. Simulating different conditions in a single run

In this Section we describe the integration of the on-line analysis with the GREMLINS framework [180], a collection of modules that are loaded into the execution environment of the targeted MPI application to emulate different aspects. The framework provides different classes of GREMLINS: power GREMLINS reduce the power budget using DVFS or cap power on a per node basis; thermal GREMLINS enable thermal throttling; resilience GREMLINS inject faults into target applications and enable us to study the efficiency and correctness of recovery techniques; bandwidth and memory GREMLINS limit resources in the memory hierarchy such as cache size or memory bandwidth by running interference threads; latency GREMLINS degrade memory latency; and noise GREMLINS inject periodic or random noise events to emulate system and OS interference.

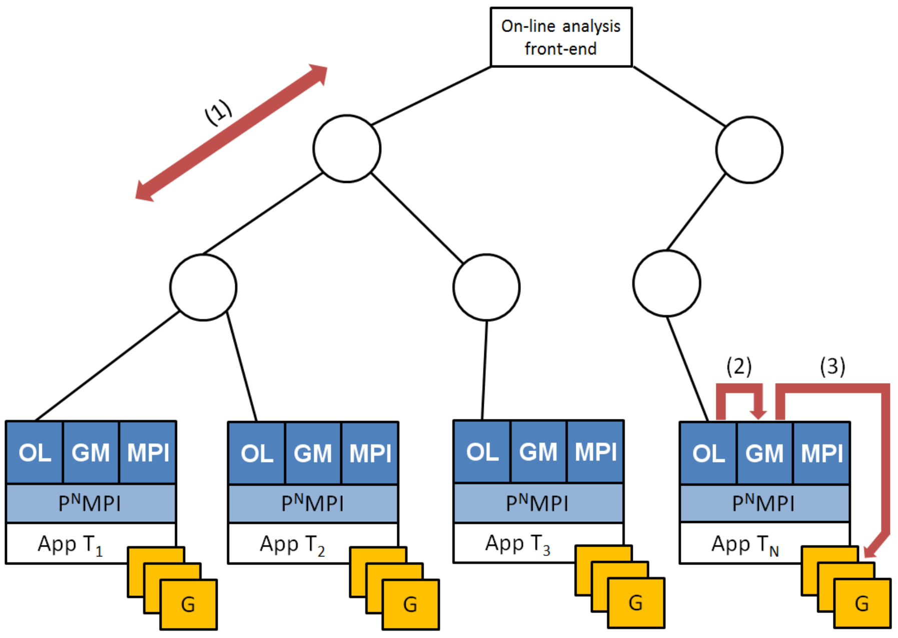
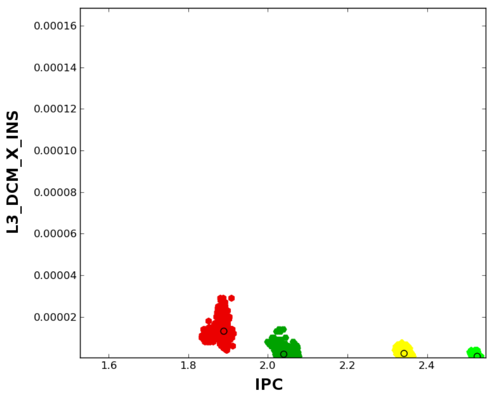


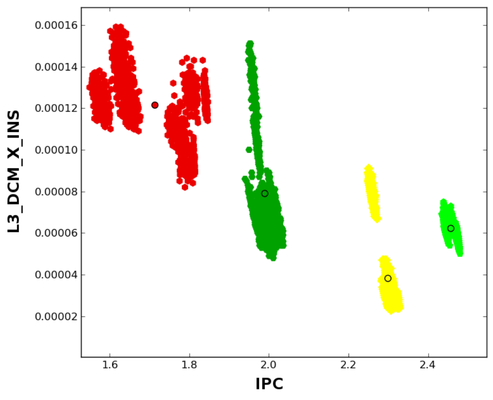
Figure 8.1.: On-line active measurement protocol workflow

Multiple GREMLINS can be used together, providing the illusion of a system that is modified in several different aspects. For example, this enables to emulate a combined effect of power limitations and reduced memory bandwidth. To accomplish this transparently to the application and for an arbitrary combination of GREMLINS, the framework relies on P^{N}MPI [182]. P^{N}MPI is a virtualization layer for PMPI [26], the standardized interface for tools to wrap MPI calls and thereby to monitor the communication of the message-passing applications. P^{N}MPI extends the PMPI interface enabling to load and to stack arbitrary PMPI tools dynamically and run them concurrently. This mechanism, in addition to allowing to inject several GREMLINS into the application, also allows to plug-in a monitoring system, and through this mechanism we connect the on-line analysis system, as shown in Figure 8.1.

When the application starts executing and calls the MPI initialization routine `MPI_Init`, the control is transferred first to the GREMLINS manager (GM), which triggers the appropriate GREMLINS into the execution. Next, control is transferred to the on-line analysis framework (OL), which starts the analysis front-end, the reduction tree and the tracing back-ends, as seen in Chapter 3. Then, the control returns to the application that starts running normally, with one or more GREMLINS interfering the resources in parallel, and the monitoring system taking performance measurements. Periodically, the analysis front-end will pause the execution (1), and communicate with the GREMLINS manager (2; the interaction between both systems is made through signals) to change the active GREMLINS to simulate a different scenario (3). The decision on when to change the simulation conditions can be taken arbitrarily, e.g. every certain amount of seconds, or intelligently, e.g. considering the result of a previous analysis to detect the application's periodic patterns (see Chapter 5).



(a) LULESH experiment without interferences



(b) LULESH experiment with interferences

Figure 8.2.: Structure of the main computing regions of LULESH with (right) and without (left) interferences

8.2.2. Studying the effects of memory congestion in LULESH

The objective of this experiment is to study which parts of the LULESH 1.0 benchmark [183, 184] are more sensitive to memory restrictions. To do so, we interfere the execution collocating GREMLIN processes in the same nodes where the application is running, constantly stealing memory capacity by randomly filling and invalidating the L3 shared memory cache. To this end, the on-line analysis framework will orchestrate the GREMLINS manager to use a variable number of memory GREMLINS at different execution intervals.

LULESH represents a typical hydrodynamics code, which describes the motion of materials relative to each other when subject to forces. LULESH is a highly simplified application, hard-coded to only solve a simple Sedov blast problem with analytic answers, and approximates the hydrodynamics equations discretely by partitioning the spatial problem domain into a collection of volumetric elements defined by a mesh. In this experiment, we run LULESH with 8 MPI tasks distributed over 2 nodes. Each node has 2 CPU sockets with 8 cores each. Then, the MPI tasks are mapped so that each of the 4 CPU sockets available runs 2 MPI tasks. In the remaining slots will run the GREMLINS, and the mapping is important so as to impact all the application's processes evenly. Throughout the execution, the number of memory GREMLINS increases at a rate of 1 per CPU socket every several iterations. In this way, in the same run there is an initial execution interval without memory interferences, a final region with very reduced cache capacity with all the CPU sockets populated with up to 6 GREMLINS, and all the intermediate configurations also tested. At the same time, the on-line analysis framework is taking performance measurements for the subsequent time intervals with varying number of GREMLINS. And at the end of the execution, all the data is correlated using tracking (see Chapter 7), enabling us to evaluate how the performance behavior of the program changes in relation to the degree of memory interference.

The experimental setup has a baseline L3 cache capacity available of 20 MB, and each

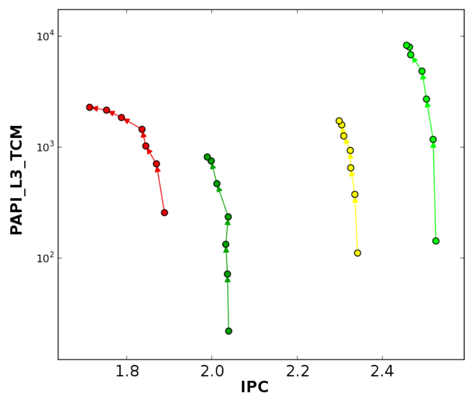
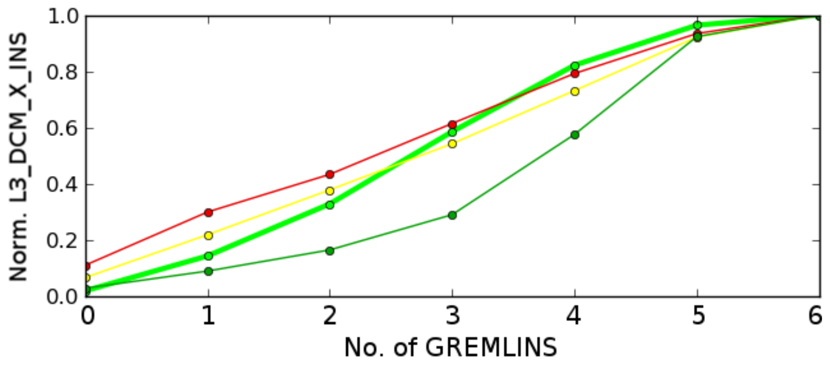


Figure 8.3.: Trajectory of clusters through LULESH experiments with increasing memory interferences

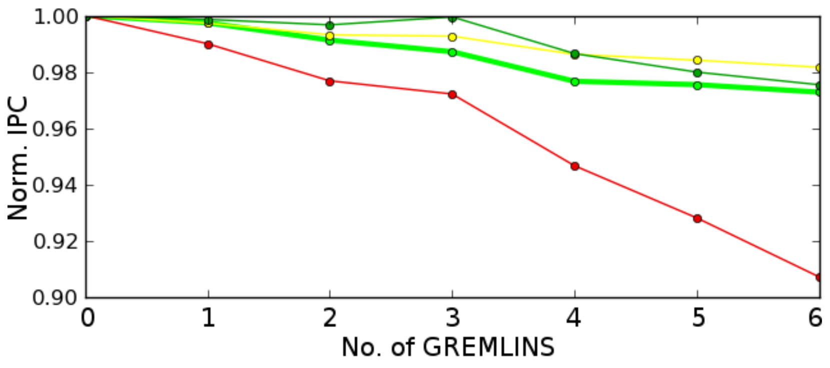
each memory GREMLIN is configured to use 4 MB buffers, which results in minimal memory bandwidth consumption (600 MB/s out of 40 GB/s), and a reduction of the effective cache capacity to approximately 15 MB, 12 MB, 7 MB, 5 MB and 2.5 MB when using from 1 through 5 GREMLINS, respectively (refer to [185] for details on predicting the impact of adding new GREMLINS).

Figure 8.2a shows the performance characteristics of the program when there are no GREMLINS active. We can observe 4 main computing trends, characterized for having significantly high performance (IPC > 1.8, X-axis), and a low number of L3 data cache misses per instruction (Y-axis). Figure 8.2b shows the performance behavior of the same code regions when the maximum number of GREMLINS are running concurrently (6 per CPU socket). Here we can observe that the clusters have become much more disperse, especially in the vertical axis, meaning that the intervention of the GREMLINS is contaminating the memory caches and causing the application to transfer data from the main memory more often. Figure 8.3 shows the evolution of these code regions considering all the experiments from 0 to 6 GREMLINS per CPU socket. X-axis represents performance (IPC), and Y-axis represents L3 total cache misses. All regions present similar behavior, amid which stands all clusters moving upwards (the more GREMLINS active, the higher data cache misses), and all clusters moving left (the more GREMLINS, the lower performance).

Figure 8.4a depicts the evolution of the L3 data cache misses per instruction metric across experiments, which shows an increase of almost a 100% in all cases. Figure 8.4b shows the evolution of the IPC, which shows a decrease of at least 2%. In particular, Cluster 3 (red) shows a 10% decrease in performance, meaning that this is the part of the program that would suffer a more significant performance loss if the memory resources are limited. Thanks to the callstack information that is also extracted in the measurement phase, we can attribute this behavior to the calculations performed in the routine `CalcMonotonicQForElems`. A conclusion that we can extract from these observations is that in a future exascale system where memory will be a more scarce resource due to the competition of a high number of threads per node, this particular part of the program will



(a) Evolution of L3 data cache misses per instruction



(b) Evolution of IPC

Figure 8.4.: Evolution of performance metrics for all clusters across all LULESH experiments

become a major bottleneck.

8.2.3. Dynamic interaction with the paralell run-time

In this Section we describe the integration of the on-line analysis framework with the OmpSs parallel run-time. OmpSs [21] is a programming model that extends OpenMP [20] with new directives to support asynchronous parallelism and heterogeneity (devices like GPUs). The OmpSs parallel run-time works on top of Nanos++ [186], a run-time service that provides mechanisms to support task parallelism using synchronizations based on data-dependencies. In turn, Nanos++ runs on top of DLB [187], a dynamic library designed to speed up hybrid applications with nested parallelism by improving the load balance inside each computational node. Similarly to the way we have used the on-line analysis framework to dynamically control the active GREMLINS to simulate different machine scenarios throughout the execution, these tools combined enable the on-line analysis to dynamically change the parallel run-time internal settings, and then study which settings yield a better program performance.

In particular, we have experimented with tuning the low-level DLB's balancing decisions (i.e. how many threads are assigned to each process), trying different resources distribution schemes throughout the run. To this end, DLB provides mechanisms to assign or remove threads from a process, which have been extended with a new API that enables the on-line analysis framework to adjust the thread mapping over time.

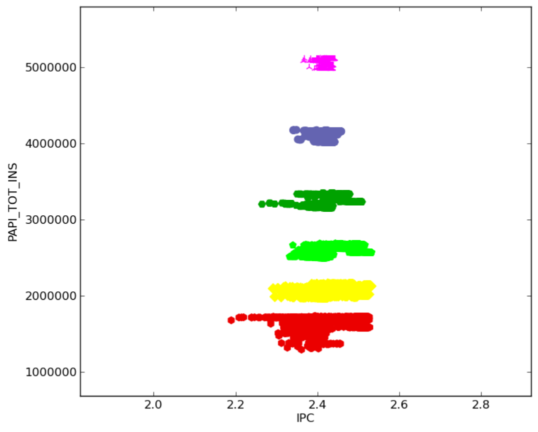
8.2.4. Studying the effect of thread-balancing strategies in BT-MZ

The objective of this experiment is to study how the thread mapping affects the performance of the NAS BT-MZ benchmark. The NAS Parallel Benchmarks (NPB) [141] are a small set of programs designed to help evaluate the performance of parallel supercomputers. Problem sizes in NPB are predefined and indicated as different classes. BT-MZ consists in a block tri-diagonal solver derived from computational fluid dynamics (CFD) applications, with uneven-size zones within a problem class, and increased number of zones as problem class grows. In this case, we have run the benchmark class B using 2 MPI tasks, and a variable number of OpenMP threads per process ranging from 1 to 8. In total, the execution never exceeds 8 concurrent threads, which means that we can set a perfectly balanced mapping with 4 threads assigned to each process, or a very unbalanced case with 7 threads assigned to the first process and only 1 thread assigned to the second process.

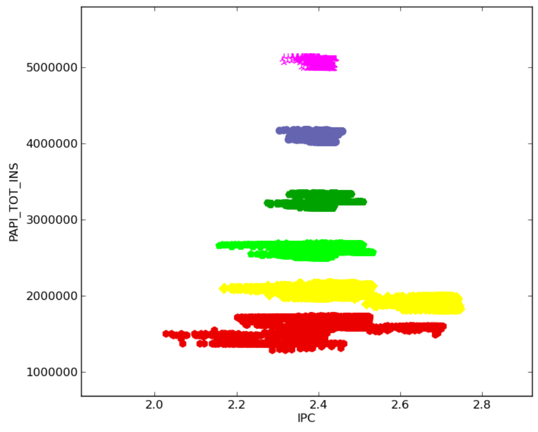
In this experiment, the on-line analysis system changes the thread mapping every few iterations of the program, making a full sweep that considers all possible distribution schemes, reallocating 1 single thread at a time. The starting point is the most unbalanced scenario, having 7 threads assigned to the first process. For the following experiments, threads assigned to the first process are reassigned to the second, until we achieve the most balanced scenario with 4 threads per process. From this point on, we keep reassigning threads from the first process to the second to generate the mirrored unbalanced scenario, with 7 threads assigned to the second process.

Figure 8.5a shows the performance characteristics of the program when the thread mapping is very unbalanced, with 7 threads assigned to the first process, and only 1 thread assigned to the second process. In this case, we can observe that the program presents 6 main computing trends, characterized for having a very high IPC (around 2.4, X-axis), and a wide dynamic range in the number of instructions executed (Y-axis). These behaviors reflect the different workloads for the uneven-size zones of the benchmark. Figure 8.5b shows the same structure for the perfectly balanced case, with 4 threads assigned per process. In this case, we can observe that the clusters elongate horizontally, meaning that there is more variability in the IPC in the balanced case.

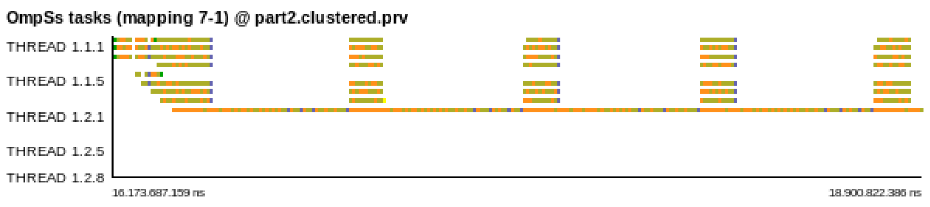
This happens because in the unbalanced case, the second process that runs with a single thread introduces a large serialization in the computation, as shown in Figure 8.5c. The first process (7 threads) finishes the computations fast, and has to wait for the second process (1 thread) to finish its part, which is executing the majority of the time alone without any resource competition. In the balanced case, see Figure 8.5d, the work is better balanced so the are no serializations, but since all threads are executing concurrently all the time, the competition for resources is higher and the threads interfere with each other, degrading the performance. Table 8.1 shows the total time percentage that the program spends executing at different levels of thread concurrency, where the reader can see that



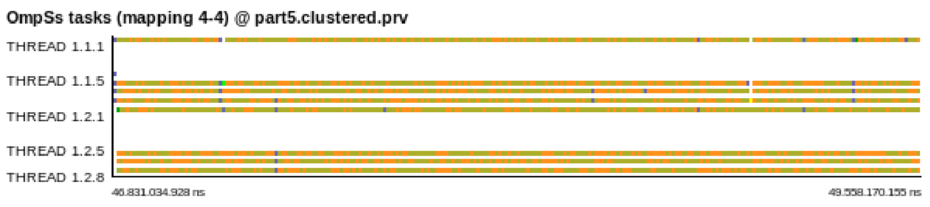
(a) BT-MZ experiment with unbalanced threads



(b) BT-MZ experiment with balanced threads



(c) Tasks distribution with unbalanced threads



(d) Tasks distribution with balanced threads

Figure 8.5.: Structure of the main computing regions of BT-MZ with unbalanced threads configuration (a) and balanced (b), and their respective timeline distribution (c, d)

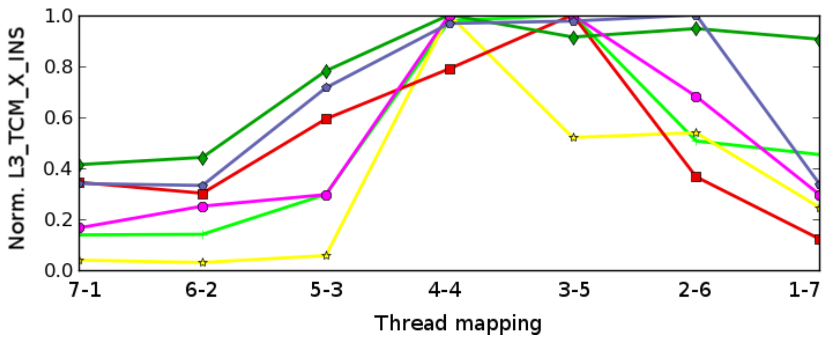


Figure 8.6.: Evolution of L3 total cache misses per instruction across all BT-MZ experiments

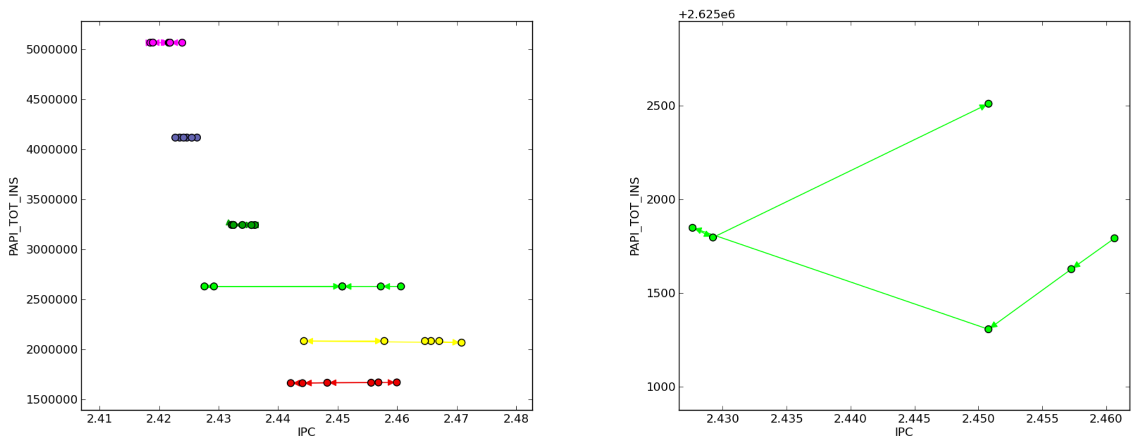
	Execution time %	
No. concurrent threads	Unbalanced mapping (7-1)	Balanced mapping (4-4)
1	68.35%	7.36%
2	2.40%	8.46%
3	2.19%	37.58%
4	1.97%	41.88%
5	4.36%	1.33%
6	12.49%	1.39%
7	8.24%	2.01%

Table 8.1.: Comparison of the execution time percentage with different thread concurrency levels in NAS BT-MZ between an experiment with unbalanced thread mapping (all threads to a single process) and balanced thread mapping (same threads per process)

there is a single thread executing alone about 70% of the time in the unbalanced case, while in the balanced case almost 80% of the time there are 3 or 4 threads executing concurrently.

The higher concurrent thread interaction results in an increased L3 memory miss rate. Figure 8.6 shows that the average L3 total cache misses per instruction (Y-axis) is maximum for all code regions in the balanced case, and lower in the unbalanced scenarios, which explains the IPC degradation in the most balanced case.

Figure 8.7a shows the tracking results for the series of experiments reallocating one thread at a time, starting with the mapping 7-1 and moving to the opposite mapping 1-7. Here we can observe a bouncing movement. At first, all threads are assigned to one process, and the clusters centroids are on the right-hand of the plot with a high average IPC. As we start removing threads from the first process and assigning them to the second, the IPC degrades (clusters move left) because the concurrent execution of several threads results in higher cache misses and so the average performance decreases. However, the overall load balance is better and the total execution time also decreases. After the balanced scenario, we start unbalancing the threads again in favor of the second process, which results in a mirror scenario and the average IPC improves again although the balancing is better, and the total execution time becomes slower. This effect can be seen more clearly in Figure



(a) General view of all clusters (b) Detailed view of Cluster 1

Figure 8.7.: Trajectory of clusters through BT-MZ experiments with different thread-balancing configurations

8.7b, which zooms in the trajectory followed by Cluster 1 (green). All other clusters show the same bouncing behavior.

For Clusters 1 to 3 (green, yellow and red), the variation in IPC is more significant, i.e. their displacements in the horizontal axis are larger, especially for Cluster 1 (green). These computing regions correspond to the line solves in the X, Y and Z directions performed in routines `x_solve`, `y_solve` and `z_solve`. One conclusion that we can extract from their behavior is that these tasks are most sensitive to the competition for resources of many threads, and this information could be useful for the run-time to try avoiding scheduling many tasks of this type together to minimize the impact of sharing resources.

8.3. Related work

Many tools have demonstrated the importance of multi-experimental analysis [59, 188, 41, 189, 126, 190]. However, previous techniques require to run the program several times to obtain all the necessary performance data for the analysis, a solution which is timely to get results and computationally expensive. The technique presented here overcomes these limitations by producing all the necessary performance data in a single execution.

To this end, we have proposed the use of active measurement techniques to simulate different machine conditions over time [185]. Similar approaches use interference threads to steal available cache storage and bandwidth [191, 192, 193, 194], but in general they fall short in precisely controlling the exact resources they are interfering, biasing the performance measurements obtained.

Then we have extended this idea to study not only different machine conditions, but different parallel run-time settings. Most run-times offer mechanisms to tune their internal activity. For example, MPI implementations [129, 195] typically allow to configure the eager limit (i.e. the method of sending short messages) through environment variables. Also, some scheduling decisions in OpenMP [20] applications can be delegated to the user through extended annotations in the code. The OmpSs [21] task scheduling strategy can also be controlled manually through the `resources` directive. But in order to get any benefit by tuning these settings the user often requires some trial and error to find which settings are more optimal. Our method extends these functionalities enabling to try different run-time settings automatically and provide feedback to the parallel run-time dynamically.

Sharing the idea of testing several configurations under the same run, we find tools like AutoTune [127], a plugin-driven framework that automatically runs the program one or more times to try different configurations of compiler flags, energy efficiency parameters and different execution patterns, and returns recommendations to tune the code. Similarly, Active Harmony [128] is an automated runtime tuning system that puts emphasis on dynamically adapting to changing resource capacities and application requirements. Our goal is different: to extract all the detailed measurements necessary to conduct an in-depth analysis of the data, instead of reporting general and predefined recommendations. Moreover, our approach enables to simulate on-line both different hardware and runtime settings, requiring only a single program run.

Following a different direction, on-line performance modeling tools like PEMOGEN [102] could also be used to make dynamic predictions of the impact of different machine configurations on the program performance. Rather than interfering the execution to limit the available resources and then measuring the effect, it would be possible to rely on architecture-based performance models (e.g. [196, 197, 198]) to predict changes in the program behavior. The COMPASS framework [105] relies on the performance modeling language Aspen [106] for automated static analysis of the target application. Their prototype system is integrated on top of OpenARC, a compiler and runtime system, enabling the study of runtime relevant parameters, such as predicting runtimes under different application and system configurations, or finding optimal decision points for a performance target with multiple constraints (e.g., whether it is faster to offload a function and pay data transfer costs or to execute the function on the host CPU). Using performance model-based predictions, the comparative analysis methodology proposed that employs tracking techniques could still be applied to effectively compare the different scenarios. Nevertheless, the main advantage of applying active measurement techniques over modeling lays on avoiding the difficulties, both in terms of cost and accuracy, of building a model that is general and portable enough to infer the impact of a wide number of machine and runtime parameters over many different architectures and software versions.

8.4. Closing remarks

Multi-experiment analysis is important for perfomance evaluation and application tuning, but it is a timely and computationally expensive task. In this chapter, we have proposed to combine several techniques to take advantage of the highly iterative behavior of parallel applications to obtain from a single experiment all the performance data that would otherwise require to run the program multiple times, saving on time and resources.

On one hand, the on-line analysis framework monitors the application execution, providing the ability to detect the application's periodic structure on the fly, and deciding when is the right moment to start simulating different machine conditions. On the other hand, the GREMLINS framework is responsible for injecting controlled interferences into the target application that can affect one or more sources like the CPU, memory or network. By having both tools working together, we are able to simulate different scenarios over different time intervals within the same execution. Then tracking can be applied to study how the program's performance behavior changes over the different scenarios. In this way, we are defining a new methodology to analyze very detailed trace data, considering all the time and space varying behavior, and correlating the data from multiple experiments, without having to generate and analyze the independent trace of each experiment per separate.

Furthermore, we have proposed to extrapolate this idea to interact with the parallel run-time. Being able to communicate our whole analysis infrastructure with the underlying run-time system, we can make dynamic adjustments of the run-time internal settings to see how they impact the program's performance. And if the analysis is perfomed on-line, we could even provide dynamic feedback to the run-time to stick with the most optimal settings.

This work has opened several interesting lines of research. Adaptative applications are an interesting subject of study, to see if we can dynamically decide whether the application could benefit from adjustments considering the current use of the available resources. Also, the tracing process could be guided by the impact of the interferences, i.e. if limiting a given resource does not affect the program performance, then it is probably not worth to obtain performance measurements regarding this particular resource as it is not going to pose a performance bottleneck. Finally, it would be interesting to incorporate into the trace the information of which degree of interference each part of the program is able to handle before it starts degrading, to enable in-depth analysis of the resource needs correlated with the program source code.

Part IV.

Conclusions

Chapter 9

Conclusions

In this thesis, we have presented the research to improve the traces scalability problem. Trace-based tools provide much more detailed information than any other approach in performance analysis, yet the amount of information collected can grow so large that traditional tracing techniques have become no longer applicable in the large scale. This work addressed the challenging problem of enabling trace-based tools to be usable, effective and applicable for the analysis of large scale parallel applications, and helped lay the foundation of intelligent tracing as a valuable technique for performance analysis in the exascale.

The main goal was to minimize the amount of captured data while maximizing the amount of relevant information presented to the analyst. To this end, we proposed and validated new instrumentation techniques and analysis methodologies to overcome the limitations of trace-based analysis from two sides.

On one hand, we proposed a set of automatic mechanisms to intelligently focus the analysis on the most relevant data, obtaining manageable amounts of information even for large-scale experiments. On the other hand, we proposed a variety of techniques to facilitate and improve relevant aspects of the task of analysis, with regard to (i) finding new ways to represent, interpret and extract useful observations, (ii) extending the scale of the analysis, (iii) facilitating comparative and multi-experiment studies, and (iv) increasing the productivity of the overall analysis process.

Our approach to the problem has focused on leveraging data analytics techniques from other fields like data mining, machine learning, signal processing, object tracking and active measurement; to automatically extract useful insight from the data. We demonstrated the utility of the techniques introduced to conduct different cases of study of parallel applications, reaching useful and comprehensible conclusions towards feasible programs' optimizations. Table 9.1 shows a summary of the experiments that have been reported in this thesis.

Technique	System	Tested applications
On-line clustering	Marenostrum 2 [181]	GROMACS, Leslie3D, MILC, SPECFEM3D, Zeus-MP
On-line spectral	Marenostrum 2	NAS-BT, PfloTran
	Jaguar [199]	PEPC, PfloTran
Parallel clustering	BlueWaters [155]	AMG2013, AVBP, CPMD, GTC, GROMACS, HACC, IPIC3D, LINPACK, NEKBONE, PEPC, PFLoTran, SAMRAI, Zeus-MP
Tracking	Marenostrum 2	CGPOP, GADGET, GROMACS, NAS-BT, NAS-FT, Quantum Espresso, WRF
	Marenostrum 3 [181]	HACC, OpenMX
	Minotauro [181]	CGPOP, Hydro, MR-Genesis
On-line active measurement	Marenostrum 3	LULESH, NAS-BT-MZ

Table 9.1.: Summary of reported experiments

This chapter compiles all the conclusions and discusses potential future lines of research.

9.1. Enabling on-line analysis of extreme scale parallel applications.

The core development of our research consists in an on-line analysis framework that combines a tracing system and an overlay analysis network. The target of the framework is to automatically extract performance data from the execution of a parallel application, and analyze it on-the-fly. On top of this software infrastructure, we proposed two automatic analysis techniques, based on clustering and signal processing, to supply the tracing process with intelligence to select the most interesting information to produce a *smart trace*: a minimal set of data that best describes the overall program behavior.

To achieve this, we made several design choices. We structured the analysis network in a tree-like topology, where the leaves of the tree are threads that attach to each process of the parallel application, extract performance measurements from them, and provide mechanisms to dynamically filter, summarize and manipulate the data. The intervention of these monitors is globally orchestrated by the root node of the tree, a central process that has the ability to decide when to extract performance data, how to perform the analysis of this data, and how to interpret the results. Between both end-points of the tree, the intermediate nodes operate applying reductions on the data that is sent through the network.

Application	No. tasks	Full trace size	Smart trace size	Reduction %
PFloTran	32	8.5 GB	275 MB	96.84%
GROMACS	64	20 GB	200 MB	99.02%
NAS BT	64	1.5 GB	20 MB	98.70%
SPECFEM3D	64	3 GB	100 MB	96.74%
PFloTran	128	1.3 GB	93 MB	93.01%
MILC	245	5.5 GB	200 MB	96.45%
Zeus-MP	256	22 GB	350 MB	98.45%
Leslie3D	512	82 GB	600 MB	99.29%
PEPC	1,024	7.0 GB	700 MB	90.23%
PEPC	2,048	0.5 TB	5 GB	99.02%
PFloTran	16,384	28 TB	2.6 GB	99.99%
PFloTran	32,768	78 TB	7.3 GB	99.99%

Table 9.2.: Summary of reported trace data reductions

This structure provides three main advantages: (i) enables independent tracing behavior for each application process, i.e. take different performance measurements for the different computing resources to maximize the spectrum of metrics evaluated; (ii) allows taking decisions having a global view of the overall state of the application; and (iii) keeps the system scalable by reducing logarithmically the set of linear data that is extracted from each process.

Beyond the uses cases that we implemented, the modular construction of the system enables to easily extend its functionalities by connecting new plug-ins of analysis to the different components of the framework. To ease the development of new analysis plug-ins we have developed *Synapse*, a middleware that can be seen as a simple programming paradigm to implement parallel computations that require master-slave coordination and that can take advantage of the hierarchical reduction model employed (refer to Annex B).

The proposed implementation follows an extremely scalable design that has been proven to handle very large executions up to tens of thousands of cores, and achieve large trace data reductions of multiple orders of magnitude, as shown in Table 9.2. Furthermore, our approach would as well accomodate different types of performances tools other than trace-based, for example, highly scalable profilers, debuggers or expert systems.

9.2. Detecting relevant phases through on-line clustering analysis.

We have explored the application of cluster analysis techniques to analyze the performance data extracted from a parallel application at run-time. Although the use of cluster algorithms in performance analysis is not new, we have found in these techniques a different possibility of application that serves for a new purpose. Traditionally, cluster analysis has been applied to detect representative processes in the parallel execution, aiming at reducing the volume of data just by discarding the non-representative ones. This approach eliminates all the variability in the data between processes, and thus undermines the analysis precision and the ability to detect certain performance problems that are directly related to the interaction between processes.

More recent research focused on applying fine-grain clustering to reveal the detailed structure of the program's computations. Trace-based analysis benefits from this approach, as it enables a more in-depth examination of time and space variabilities, but this method of using clustering did not achieve any degree of data reduction per se.

In our research, we have given the latter strategy a novel use. Clustering is repeatedly applied to expose the performance structure of the main computing regions of the program over consecutive time intervals of the execution. Subsequent clustering results are then compared to see whether changes in the application performance happen.

Given the iterative nature of the vast majority of codes, the algorithms tend to quickly converge into an iterative pattern over which the achieved performance presents minor fluctuations. When the observed behavior differs significantly from the baseline, this is indicative of the program undergoing perturbations or entering into a different algorithmic phase.

Periodically assessing the performance structure of the clusters we automatically identify relevant phases of the program while it is executing. This information enables us to produce a compact trace comprising selected representative regions of the whole execution, yet preserving all the important fine-grain trace information to support a posterior detailed analysis.

9.3. Generating multi-detail traces through on-line spectral analysis.

Previously, we proposed the use of cluster analysis to dynamically infer when the program had entered into an important computing phase, and use the information to intelligently select a small representative region to be traced in detail. As we proved, our approach effectively reduced the volume of traced data, and was able to preserve meaningful information to analyze the major hotspots of the program. Nevertheless, we identified potential improvements in two areas.

In the first place, the method lacked precision to delimit the exact boundaries of the traced region, because the exact start and ending points of every iteration inside the loops of the program were not known. In the general case, the traced region would contain several iterations of the main loop of the program. But in the worst case, this could result in substandards like capturing incomplete iterations, tracing many more iterations than what is actually needed for an effective analysis, or mistaking a small sub-phase of the program with the main computing phase, which could lead to discard potentially important information. Secondly, the previous approach focused straight on the most important regions of the execution, directly discarding the remaining data without any further consideration.

In order to complement the previous work, we proposed the use of signal processing techniques to expose the precise periodic pattern of a parallel application at run-time. Wavelet transform, cross-correlation function and mathematical morphology techniques are applied to precisely identify the iterative behavior of the program, unveiling where are the periodic and non-periodic program phases, the start and end points for each loop iteration, and the zones that are less perturbed by performance artifacts. Then, we used this new information to dynamically decide at which level of granularity the data was stored in the trace, ranging from a general view of the periodic structure of the application to every single detail for specific time intervals.

We demonstrated the usefulness of the proposed approach by producing execution traces of several tens of thousands of processes, achieving size reductions of three and more orders of magnitude. We also proved that the analysis of the traced information can successfully lead to important improvements, as any recurring problem that can be observed in a few representative iterations and corrected, will positively impact the performance throughout the whole periodic phase of the program.

The addition of signal processing techniques to our on-line analysis framework transfers the responsability of deciding which are the best areas to trace from the study of the structure of the computing phases through cluster analysis, to the study of the periodicity structure of the program through signal processing. By no means this implies that cluster analysis falls in disuse. Quite the opposite, in combination with clustering, we can now incorporate into the trace the internal structure of the computing phases for selected representative regions of the execution, better controlling the size of the resulting trace, and adding a new layer of very fine-grain data to the trace that brings with it the added value of cluster-based analysis methodologies and the performance models that can be derived from this information.

The application of the proposed techniques can be seen as a first step in the analysis methodology, obtaining the minimum amount of information that is relevant for the analysis. This opens the way for other analysis mechanisms that could then be applied (e.g. automatic analysis, expert systems or performance modeling), not having to deal with large amounts of data and all the difficulties that are inherently associated.

9.4. Complementary performance reports to enrich the analysis.

In order to face the analysis of a large volume of data, it is vital to find efficient representations of the data to permit direct lookup without requiring expensive post-processing treatments; and so have a well-defined methodology to inspect the results, make understandable interpretations and draw useful conclusions. To this end, the on-line analysis mechanisms are not only limited to producing a succinct trace. As a result of the analysis, additional performance reports, trend plots and breakdown models are produced periodically, providing extra insight about the behavior of the application and its evolution over time.

On one hand, the multi-detail data collection approach extracts performance information that fully covers the whole execution, enabling a top-down analysis methodology that goes first from a very abstract view for the whole run to get general understanding of the program behavior, to delving into the small details within a small time interval and subset of processes to understand any microscopic phenomena that may impact the performance.

On the other hand, the study of the clusters gives us a new perspective to interpret the application performance concisely. The shape of elongated clusters can be an indicative of potential load imbalances. The density and temporal distribution of the clusters highlight the zones of the program that are more significant in terms of the time they consume. And the composition of the clusters enables to characterize multi-modal behavior, i.e. different parts of the program with similar performance behavior and vice-versa.

All this new information, combined with the analysis of the detailed event trace, allows us to reverse the typical direction of the analysis process for a more effective outcome. We can now first evaluate the performance of the most relevant program phases, characterize the nature of their inefficiencies, and ultimately propose a solution that is applicable to all the parts of the program that present the same behavior, instead of having to reappraise routine per routine the same commonly known performance problems to see whether the observed performance is satisfactory or not, as the most traditional performance tools would do.

While the study of the clusters provides extra insight to better understand the application behavior, some of the observations that can be made still require to inspect the trace to corroborate them. For example, if a cluster presents an elongated shape because it comprises unbalanced computations that may run faster or slower, it is important to see how these different computing behaviors distribute over time and space to determine whether they pose a real problem of imbalance or not.

We can not have certainty of some potential performance issues just by studying the shape of the clusters because the scatter-plot representations used to characterize the performance do not include the temporal and spatial information that is present in the trace. We proposed a new scatter-plot representation for the clusters based on the idea of heat maps. In heat maps, the temperature indicates how many parallel processes support the different areas of the clustering space. The temperature of the clusters enabled quick identification of the most frequent performance behaviors among parallel tasks, and easy characterization of the source of variabilities behind very disperse clusters, pointing out whether the behavioral differences occurr between processes or over time.

Additionally, we extended existing performance hardware counters multiplexing techniques to compute global statistics of the clusters in parallel, complementing the clusters characterization. We used many more metrics than those that can be gathered in a single run, and condensed all the extra information in comprehensible break-down models that describe how the resources are spent. For example, the architecture impact model indicates which hardware components are most used and how often the processor stalls due to lack of resources; and the instructions mix model indicates the different types of operations that are necessary to perform a given computation.

In any case, the purpose of these reports is not to replace the task of analysis of the trace, but to enrich the interpretation of the available trace data, and to enable to reach quick conclusions before a more in-depth analysis.

9.5. Extending the scalability of the analysis.

When the amount of performance data grows large, cluster-based analysis becomes a serious bottleneck. In this thesis, we assessed the cost of the basic density-based cluster algorithm DBSCAN, and empirically proved that the processing of just a hundred thousand data points can already require several hours to compute. This is extremely expensive if we consider that current supercomputers already have the same order of magnitude of cores, which can produce performance data at sustained rates of thousands of measurements per second, and so the cost of analyzing all the data from a large scale run easily exceeds any reasonable time frame. Thus, the applicability of cluster analysis at run-time is indispensably subject to the availability of more sophisticated algorithms able to process large data volumes in near-real time to be usable on-line.

We have explored two main research directions to overcome the scalability limitations of the DBSCAN algorithm. First, we proposed a sampling-based approach to accelerate the clustering process. In our research we found that because of the iterative nature of scientific applications and their highly regular and repetitive structure, even a very small sample space with few measurements taken from few processes was enough to reliably characterize the performance characteristics of the program. This solution effectively gained ground to the scalability limits, but only in the short term. As the scale keeps increasing, a small percentage of samples taken from a large number of processes still is a large volume of data. At this point, further reducing the sample space easily results in a biased subset of points that poorly describe the complete data set, and the clustering results become distorted and unrepresentative.

Then we approached the problem differently, considering to design a parallel version of the cluster algorithm. There are parallel implementations present in the literature, but virtually all of them rely on a pre-processing phase where optimizations are often made to distribute the data so that the dense areas are kept close. This is done to facilitate the main operation of the subsequent cluster algorithm, which consists in searching for neighbour points along the data space. But in the on-line scenario, where each process is producing its own performance data, the data is implicitly distributed from the start. Going through a redistribution phase that would keep the dense areas close would require a massive movement of data between all processes, which would be as expensive as the cluster algorithm itself and unworkable to be used at run-time.

This led us to a new parallelization strategy for the DBSCAN algorithm that does not require an explicit phase of data redistribution. It is designed to work efficiently in those scenarios where the dense areas are highly distributed, which is precisely the case of large scale parallel applications. Our algorithm follows the hierarchical tree structure deployed for the on-line analysis framework, taking full advantage of all the resources available to perform the analysis. In this way, the leaf processes run a basic DBSCAN with their local data, the resulting clusters are modelled with simple geometries, and they are combined in parallel in the intermediate nodes of the tree to obtain a global clustering model. Finally, the local data of each process is classified using a nearest neighbor algorithm with the global model as reference.

We validated our proposal with a series of experiments scaling up to several thousands of cores and millions of data points, measuring response times of few seconds, which represents an overhead low enough to be used on-line. Furthermore, the approach presented could as well be adopted by other cluster algorithms families that can also take advantage of a distributed structure to easily parallelize their computation.

9.6. Facilitating comparative and multi-experiment analysis.

The aforementioned research addressed the scalability problem of trace-based analysis from the standpoint of reducing the size of the stored data, by intelligently selecting relevant regions of the execution.

From the analysis of the captured data we learned that, throughout the execution, there may be several important phases, because of performance perturbations that the application may undergo at different intervals, or just because the program may enter a different algorithmic phase. It then became necessary to have an effective way to compare and contrast performance observations belonging to different execution phases, to see how the program behavior changes and evolves.

Moreover, a single experiment is often not enough to fully understand the behavior of the program. In the general case, there are a wide range of tunable parameters and possible tweaks that can potentially alter the performance of the program: the number of parallel processes or threads used, the strategy to distribute the workload, the use of specific hardware like accelerators, different data access and algorithmic patterns, as well as many other settings. It is important to understand how all these choices impact the execution, which will often require multiple experiments trying different configurations.

Contrasting the information from several data sources and interpreting the results for large amounts of uncorrelated data is a difficult task. Thus, it is necessary to develop techniques to correlate the information and present it in an efficient and comprehensible manner. To this end, we proposed a novel approach that leverages object tracking techniques from the field of computer vision applied to the performance analysis of a parallel application's evolution along multiple execution scenarios. We have extrapolated the concept of following the movement of objects through a physical space in a video sequence, to track the changes of behavior of the different parts of the code through a multi-dimensional performance space. Different scenarios are represented as a sequence of performance images that expresses the evolution of the application either along different experiments with changing configurations, or along time intervals within the same experiment. Computing regions of the application are represented as objects in these images, described by how they behave in terms of selected performance metrics. Then, we find a correspondence between objects along the whole sequence of images, keeping track of their possible motions and structural changes due to performance variations. To do so, we use a variety of heuristics that take into account different characteristics of the computing regions: the displacements in the performance space, the SPMDiness of the application, the density of the clusters, the code region they refer to, and the execution sequence. Combining their use, we are able to automatically identify the global evolution of the main computational behaviors and illustrate their performance trends.

The proposed technique offers a different viewpoint to the task of analysis that is more agnostic of the syntax of the code, but brings into focus the main performance characteristics of the program and the nature of their inefficiencies, enabling the identification of the most appropriate solution for the artifacts observed. Then, these observations can be correlated with the source code, to know which sections exhibit a given behavior. There are two remarkable benefits to this approach. First, the same solution can be applied to multiple code sections that present the same deficiency, without having to reappraise the same problems repeatedly. Second, we are able to detect multi-modal behavior and variations along time and processors, two important effects often masked by profiling tools. In this way, a single code section undergoing performance variabilities will be expressed as divergent behaviors that can be studied separately, revealing more room for improvement.

All in all, this work presents a versatile technique applicable in very varied scenarios, enabling the analyst to study the impact of virtually any configuration on the application performance without prior knowledge of the program; compare and correlate performance data from different sources; determine the best setup to meet specific performance requirements; and ultimately helps to gain better understanding of the application behavior.

This work addressed an important problem in performance analysis, where the sheer amount of data that can be collected expands very rapidly in a potentially high dimensional space with many metrics available, and we have provided a technique to manage this complexity and identify coarse properties that change when parameters are varied to target tuning and more detailed performance studies.

9.7. Improving the productivity of the analysis.

Executions in a production-class supercomputing facility are often subject to many restrictions, including any sort of batch scheduling software that runs the submitted jobs based upon resource constraints and limited computing time allocations, shared with many other users. Often, a job pending execution may sit in the queue for a very long time, and when many experiments are needed, the time required to produce all the data can easily prolong for several days as it is difficult to access the high amount of resources that are needed to run the experiments and produce all the analysis data. In order to improve the productivity of the analysis process, it is important to gather as much performance data in the less number of runs, ideally, producing all the necessary data in just a single execution.

To this end, we proposed the application of active measurement techniques to simulate in a single run different experimental conditions that would otherwise require multiple runs to try. Our approach consists in introducing controlled interferences into the program that impact the performance of one or more specific resources, such as CPU, network or memory. Affecting these resources at different levels of intensity, we can simulate the effect of having, for example, a slower processor or narrower network and memory bandwidth. Then, applying the tracking techniques proposed before, we can study how the application behaves under different situations like node oversubscribing, or network or memory congestion.

The main benefit of this approach is that it enables us to obtain all the information necessary to understand the impact of one or several parameters in the program performance from a single run. Moreover, to do this we can apply all the techniques and methodologies proposed in this thesis. This is to say, the application's performance data can be analyzed on-line. At each analysis step, signal processing techniques can be applied to find the program's repetitive phases. To each representative phase, clustering analysis can be applied to detect the most relevant computations. This process can be repeated at different time intervals, dynamically changing the level of interference. Lastly, we can apply tracking techniques to see what is the evolution of the behavior of the main computing phases across the different zones of interference with respect to the parameters being studied. In this way, we are providing a methodology to analyze very detailed trace data, considering all the time and space varying behavior, and correlating the data from multiple experiments, without having to analyze the independent trace of each experiment per separate.

Going one step beyond, we made experiments to dynamically interact with the parallel run-time. Weaving all these pieces together, it is possible to study the impact of run-time relevant parameters in the application performance. As a proof of concept, we integrated the on-line analysis framework with the OmpSs run-time. We extended the on-line analysis framework to guide the runtime underlying balancing mechanism, deciding at the analysis layer how the available threads are assigned to physical cores, and automatically assessing which distribution is better. By doing so, we obtained useful information about which tasks would be more convenient to be scheduled alone to reduce the competition for the shared resources.

This line of work created an opportunity for providing dynamic feedback to the parallel run-time to dynamically set the most optimal settings to achieve better performance. Moreover, this approach facilitates the application of trace-based analysis tools at large scale, by reducing the necessity for computing time and resources, which in current systems is already an important bottleneck.

9.8. Future work

One of the main arguments of this thesis is that in-depth performance analysis is crucial to parallel program development. The concept of accurate and detailed analysis has been customarily bound together with tracing, a specialized technique to record very low level information about a program's execution. But the traditional approaches to implement trace-based analysis techniques have reached their ceiling [200]. The ever-growing size and power of high-end supercomputers led tracing tools to be increasingly difficult to apply, because the amount of information generated in such systems has grown so exponentially fast that storing, processing and extracting useful and timely conclusions from the data became a challenging task.

The exascale will bring even higher core counts, and the increase of hardware components and their interactions will definitely result in a greater likelihood of failures and performance flaws. In this scenario, the ability to see in detail which are the causes of poor efficiency will be more important than ever. And this is precisely the reason why trace-based analysis will continue to be necessary in the field.

In this thesis, we have addressed the scalability limitations of trace-based performance analysis approaches to keep them applicable and efficient in large-scale scenarios. To this end, we took two important steps forward. First, we worked on incorporating intelligence to the mechanisms of data extraction, automatically controlling the volume of data that would be generated. In turn, this has enabled the application of state-of-the-art performance analytics techniques at run-time, putting aside the trace-centric approach of post-processing all the recorded data.

Although analyzing the details is important, this does not necessarily mean to keep storing traces in the traditional way. On the contrary, we focused on applying on-line analysis techniques to automatically extract useful insight from detailed performance measurements, taking into account the time-varying behavior of the program over time and across processes. The information extracted has enabled us to understand the program behavior and the nature of its inefficiencies better, and for the most part, storing the trace would be no longer necessary.

The success of the techniques employed leads to think that the research in this field will continue to apply automatic analysis techniques to obtain the most useful information from the data, but the trace seen as a physical container that records the whole execution is very likely deemed to disappear.

In this direction, new interesting lines of research are being recently explored that would fit into this purpose. For example, *folding* is a novel technique that combines coarse grain sampled and instrumented information to provide the detailed node-level performance within a computing region [163]. This technique applied on-line would provide a new level of analysis detail, being able to display very fine-grain instantaneous performance metrics precisely correlated with the program source code lines.

The design of new performance models is also a very important line of research. With the daunting volume of metrics available, the high dimensional complexity of the data, and the high intercorrelations between performance factors, performance models strive to capture the application's performance characteristics in a human-understandable form. Their applications today are many and varied, for example, they are used to compare system performance, validate large system installations, to detect anomalies and degradation, to guide optimizations or refactoring of applications, and to some extent, allow predicting performance of future experiments or on a target architecture. Models based upon hardware performance counters have shown promise for on-line analysis [201] to quantitatively understand performance and power consumption, thanks to the growing use of hardware counter events in performance data, their portability to a large number of systems and because of being reasonably accurate.

Our research paves the way for new studies in areas like expert systems for machine learning. All the insight obtained with the techniques mentioned above also serves as valuable information to feed, refine, and specialize expert systems with better knowledge rules to be more elastic to the variations in computation and communication delays, and more efficient to perform automatic bottleneck detection. Nevertheless, in order to describe the knowledge rules that govern an expert system, we still need the ability to detect the microscopic effects that result in a performance problem first, understand what is the root cause, and identify which performance variables may be early markers to diagnose the problem. This is why in-depth performance analysis based on detailed event data will still be a fundamental cornerstone in the design of new architectures, programming models and better software.

Part V.

Appendices

Appendix A

Extrae On-line User Guide

A.1. Introduction

Extrae On-line is a new module developed for the Extrae tracing toolkit [53], available from version 3.0, that incorporates intelligent monitoring, analysis and selection of the traced data. This tracing setup is tailored towards long executions that are producing large traces. Applying automatic analysis techniques based on clustering, signal processing and active monitoring, Extrae gains the ability to inspect and filter the data while it is being collected to minimize the amount of data emitted into the trace, while maximizing the amount of relevant information presented.

Extrae On-line has been developed on top of Synapse (refer to Annex B), a framework that facilitates the deployment of applications that follow the master/slave architecture based on the MRNet software overlay network [87]. Thanks to its modular design, new types of automatic analyses can be added very easily as new plug-ins into the on-line tracing system, just by defining new Synapse protocols.

This document briefly describes the main features of the Extrae On-line module, and shows how it has to be configured and the different options available.

A.2. Automatic analyses

Extrae On-line currently supports three types of automatic analyses: fine-grain structure detection based on clustering techniques, periodicity detection based on signal processing techniques, and multi-experiment analysis based on active monitoring techniques. Extrae On-line has to be configured to apply one of these types of analyses, and then the analysis will be performed periodically as new data is being traced.

A.2.1. Structure detection

This mechanism aims at identifying the fine-grain structure of the computing regions of the program. Applying density-based clustering, this method is able to expose the main performance trends in the computations, and this information is useful to focus the analysis on the zones of real interest. To perform the cluster analysis, Extrae On-line relies on the ClusteringSuite tool [53].

At each phase of analysis, several outputs are produced:

- A scatter-plot representation that illustrates the behavior of the main computing regions of the program, that enables a quick evaluation of potential imbalances.
- A summary of several performance metrics per cluster.
- On supported machines, a CPI stack model that attributes stall cycles to specific hardware components.
- And a trace that is augmented with the clusters information, that allows to identify patterns of performance and variabilites.

Subsequent clustering results can be used to study the evolution of the application over time. In order to study how the clusters are evolving, the xtrack tool can be used (refer to Annex C.

A.2.2. Periodicity detection

This mechanism allows to detect iterative patterns over a wide region of time, and precisely delimit where the iterations start. Once a period has been found, those iterations presenting less perturbations are selected to produce a representative trace, and the rest of the data is basically discarded. The result of applying this mechanism is a compact trace where only the representative iterations are traced in full detail, and for the rest of the execution we can optionally keep summarized information in the form of phase profiles or a "low resolution" trace.

Please note that applying this technique to a very short execution, or if no periodicity can be detected in the application, may result in an empty trace depending on the configuration options selected (see Section A.3).

A.2.3. Multi-experiment analysis

This mechanism employs active measurement techniques in order to simulate different execution scenarios under the same execution. Using gremlins [185], Extrae On-line is able to add controlled interference into the program to simulate different computation loads, network bandwidth, memory congestion and even tuning some configurations of the parallel runtime (currently supports MPI Dynamic Load Balance (DLB) runtime). Then, the application behavior can be studied under different circumstances, and tracking can be used to analyze the impact of these configurations on the program performance (refer to Annex C). This technique aims at reducing the number of executions necessary to evaluate different parameters and characteristics of your program.

A.3. Configuration

In order to activate the On-line tracing mode, the user has to enable the corresponding configuration section in the Extrae XML configuration file (see [202] for a complete description). This section is found under *trace-control* > *remote-control* > *online*. The default configuration is already ready to use:

```
<online enabled="yes"
      analysis="clustering"
      frequency="auto"
      topology="auto">
```

The available options for the <online> section are the following:

- **enabled**: Set to “yes” to activate the On-line tracing mode.
- **analysis**: Choose from “clustering”, “spectral” and “gremlins”.
- **frequency**: Set the time in seconds after which a new phase of analysis will be triggered, or “auto” to let Extrae decide this automatically.
- **topology**: Set the desired tree process tree topology, or “auto” to let Extrae decide this automatically.

Depending on the analysis selected, the following specific options become available.

A.3.1. Clustering analysis options

```
<clustering config="cl.I.IPC.xml"/>
```

- **config**: Specify the path to the ClusteringSuite XML configuration file (refer to [203]).

A.3.2. Spectral analysis options

```
<spectral max_periods="0" num_iters="3" min_seen="0" min_likeness="85">
  <spectral_advanced enabled="no" burst_threshold="80">
    <periodic_zone detail_level="profile"/>
    <non_periodic_zone detail_level="bursts" min_duration="3s"/>
  </spectral_advanced>
</spectral>
```

The basic configuration options for the spectral analysis are the following:

- **max_periods**: Set to the maximum number of periods to trace, or “all” to explore the whole run.

- **num_iters**: Set to the number of iterations to trace per period.
- **min_seen**: Minimum repetitions of a period before tracing it (0 to trace the first time that you encounter it)
- **min_likeness**: Minimum percentage of similarity to compare two periods equivalent.
- **min_likeness**: Minimum percentage of similarity to compare two periods equivalent.

Also, some advanced settings are tunable in the <spectral_advanced> section:

- **enabled**: Set to "yes" to activate the spectral analysis advanced options.
- **burst_threshold**: Filter threshold to keep all CPU bursts that add up to the given total time percentage.
- **detail_level**: Specify the granularity of the data stored for the non-representative iterations of the periodic region, and in the non-periodic regions. Choose from none (everything is discarded), profile (phase profile at the start of each iteration/region) or bursts (trace in bursts mode).
- **min_duration**: Minimum duration in seconds of the non-periodic regions for emitting any information regarding that region into the trace.

A.3.3. Gremlins analysis options

```
<gremlins start="0" increment="2" roundtrip="no" loop="no"/>
```

- **start**: Number of gremlins at the beginning of the execution.
- **increment**: Number of extra gremlins at each analysis phase. Can also be a negative value to indicate that you want to remove gremlins.
- **roundtrip**: Set to "yes" if you want to start adding gremlins after you decrease to 0, or vice-versa, start removing gremlins after you reach the maximum.
- **loop**: Set to "yes" if you want to go back to the initial number of gremlins and repeat the sequence of adding/removing gremlins after you have finished a complete sequence.

Appendix B

Synapse User Guide

B.1. Introduction

Synapse is a framework to facilitate the development of MRNet [87] applications. Synapse takes its name from the neurological structure in the nervous system, because following the analogy, MRNet applications resemble a neural network where inputs are passed from one neuron (processor) to the next, and each processor applies a function on the passing data.

As stated in the MRNet documentation [204], "MRNet applications are characterized for MRNet is a customizable, high-throughput communication software system for parallel tools and applications with a master/slave architecture. MRNet reduces the cost of these tools' activities by incorporating a tree-based overlay network (TBON) of processes between the tool's front-end and back-ends. MRNet uses the TBON to distribute many important tool communication and computation activities, reducing analysis time and keeping tool front-end loads manageable.

MRNet-based tools send data between front-end and back-ends on logical flows of data called streams. MRNet internal processes use filters to synchronize and aggregate data sent to the tool's front-end. Using filters to manipulate data in parallel as it passes through the network, MRNet can efficiently compute averages, sums, and other more complex aggregations on back-end data".

Implementing and running an MRNet application requires to follow a series of steps that are common to most programs. These steps include: request resources for the TBON; map the available resources in a tree-like topology; start the front-end process; instantiate the MRNet internal nodes; and in the case where MRNet runs in the "back-end attach" mode, spawn the back-ends manually, pass the network connectivity information from the front-end to the back-ends externally (via the environment, shared filesystems or other information services), connect the back-ends; and load the filters into the network's internal processes and announce them.

MRNet provides a C++ API to facilitate each of these operations, yet the logical sequence of operations is the same for all programs and is rather long. Furthermore, MRNet support for BlueGene architectures requires of an entirely separate and slightly different API (known as *lightweight*), which is written in pure C. Synapse encapsulates all these common operations to all MRNet applications, and provides wrappers to hide the complexity of using two different API's depending on the architecture, making the development of a new program as easy as writing a few lines of code.

Sound understanding of the MRNet software is necessary to follow this manual, so please refer to [204] for a complete description of MRNet.

B.2. Abstractions

Synapse provides a set of libraries and classes that can be extended to define the interactions between your front-end, filters and back-end processes. Synapse handles all the operations necessary to initiate and connect all the processes in the network, and just leaves to the user the task of implementing the logic of the algorithm that these processes will perform. To this end, Synapse provides the *Protocol* abstraction. A *Protocol* is the set of operations that determine how the MRNet processes interact. For example, imagine a trivial p̈ing-pongp̈rotocol, where the back-ends send an ACK to the front-end, and the front-end replies with how many ACK's it has received. This and any other protocol has two parts: the part that is run at the back-ends, and the part that is run at the front-end. In this simple case, the back-ends have to send the ACK and receive the reply, while the front-end has to receive the ACK's and send the reply. In order to implement the logic of the front-end and the back-ends, Synapse provides two separate classes: FrontProtocol and BackProtocol. When the user wants to define its own protocol to run on top of the MRNet, he has to write the code that refers to the operations performed by the front-end extending the FrontProtocol class, and the complementary operations performed by the back-ends extending the BackProtocol class. More specifically, the user has to implement the Run() method, and write inside this method all the computations and communications that take place in the protocol.

Once this work is done, that protocol can be easily loaded into any MRNet application through the FrontEnd::LoadProtocol and BackEnd::LoadProtocol methods, and triggered anytime through FrontEnd::Dispatch. Synapse will start the protocol in all the end-points of the MRNet application, and ensure they meet a synchronization point once the protocol is over so that the next one can start.

B.3. A simple example

A detailed description of the Synapse API is in Section B.4. This section shows a brief example of a MRNet application that implements a "ping-pong" protocol. In this example, the back-ends send an ACK to the front-end, and the front-end replies with how many ACK's were received.

Front-End sample code

This piece of code creates the front-end end-point of the MRNet application, loads the front-end side of the ”ping-pong“ protocol, and then starts the protocol.

```
#include <iostream>
#include "FrontEnd.h"
#include "Ping_FE.h"

int main(int argc, char *argv[])
{
  /* Start the front-end side of the network */
  FrontEnd *FE = new FrontEnd();
  FE->Init("topology_1x4.txt", "./test_BE", NULL);

  /* Load the protocols the network understand */
  FrontProtocol *prot = new Ping();
  FE->LoadProtocol( prot ) ;

  /* Execute protocol "PING" */
  int status;
  FE->Dispatch("PING", status);

  /* Shutdown the network */
  FE->Shutdown();

  return 0;
}
```

Back-End sample code

This piece of code creates the back-end end-point process of the MRNet application, loads the back-end side of the ”ping-pong“ protocol, and waits for the front-end to orchestrate the start of the protocol.

```
#include "BackEnd.h"
#include "Ping_BE.h"

int main(int argc, char *argv[])
{
  /* Start the back-end side of the network */
  BackEnd *BE = new BackEnd();
  BE->Init(argc, argv);

  /* Load the protocols the network understand */
  BackProtocol *prot = new Ping();
  BE->LoadProtocol(prot);

  /* The back-end enters the main analysis loop,
     waiting for commands from the front-end */
  BE->Loop();

  return 0;
}
```

Front-End side of the ”ping-pong“ protocol sample code

The following code defines the front-end side of the protocol. The user has to extend the Setup() method to initialize all the streams that are going to be used during the execution of the protocol, and also the Run() method that implements the protocol. In this case, the front-end first receives all the ACK's (through the aggregation stream stAdd), and then broadcasts to the back-ends the total number of ACK's received.

```
#include <iostream>
#include "Ping_FE.h"

/**
 * In the Setup function we have to register all the streams we want
 * to use for this protocol. When the function returns, all streams
 * pushed to the registeredStreams queue are automatically published
 * to the back-ends. The Register_Stream is a wrapper to
 * net->new_Stream() that creates a new stream in the FE and pushes
 * it to the queue.
 */
void Ping::Setup()
{
  stAdd = Register_Stream(TFILTER_SUM, SFILTER_WAITFORALL);
  cout << "[FE] Created new stream " << stAdd->get_Id() << endl;
}

/**
 * Implement the front-end side of the protocol.
 * It is expected to return 0 on success; -1 otherwise.
 */
int Ping::Run()
{
  int tag, countPongs = 0;
  PacketPtr p;

  cout << "[FE] Sending PING to " << stAdd->size()
      << " back-ends through stream " << stAdd->get_Id() << endl;
  MRN_STREAM_SEND(stAdd, TAG_PING, "");
  cout << "[FE] Waiting for PONG from " << stAdd->size()
      << " back-ends..." << endl;
  MRN_STREAM_RECV(stAdd, &tag, p, TAG_PONG);
  p->unpack("\%d", &countPongs);
  cout << "[FE] " << countPongs << " PONGs received!" << endl;

  if (countPongs == stAdd->size())
  {
    cout << "[FE] Addition filter ran successfully! :)" << endl;
    return 0;
  }
  else
  {
    cout << "[FE] Addition filter FAILED! :(" << endl;
    return -1;
  }
}
```

Back-End side of the ”ping-pong“ protocol sample code

The following code defines the back-end side of the protocol. The user has to extend the Setup() method to initialize all the streams that are going to be used during the execution of the protocol, and also the Run() method that implements the protocol. In this case, the back-end first sends an ACK's to the front-end (through the aggregation stream stAdd), and then receives the reply with the total number of ACK's that the front-end received.

```
#include <iostream>
#include "Ping_BE.h"

/**
 * The streams created in the front-end are received here,
 * in the same order that were created.
 */
void Ping::Setup()
{
  Register_Stream(stAdd);
}

/**
 * Implement the back-end side of the protocol.
 * It is expected to return 0 on success; -1 otherwise.
 */
int Ping::Run()
{
  int tag;
  PACKET_new(p);

  MRN_STREAM_RECV(stAdd, &tag, p, TAG_PING);
  MRN_STREAM_SEND(stAdd, TAG_PONG, "\%d", 1);

  PACKET_delete(p);

  return 0;
}
```

B.4. Synapse API

Synapse provides several classes to help building front-end and back-end MRNet processes and the protocols they execute. These are bundled into two libraries (synapse-frontend and synapse-backend) for the respective end-point processes.

B.4.1. Class FrontEnd

An instance of this class is needed to make your front-end process. This class provides basic methods to start the network.

Init (normal mode)

Synopsis

```
int Init(const char *TopologyFile, const char *BackendExe,
         const char **BackendArgs);

int Init(const char *BackendExe, const char **BackendArgs);
```

Description The basic method that is used to create the network and spawn the back-ends (normal instantiation mode). *TopologyFile* is the path to a configuration file that defines the desired process tree topology. The specification format is the same that the one produced by the mrnet_topgen tool. *BackendExe* is the path to the binary that will act as the application back-end process. *BackendArgs* is a null terminated list of arguments to pass to the back-end application upon creation.

If *TopologyFile* is not given, this information is read from the environment variable MRNAPP_ TOPOLOGY.

Return value Returns 0 if the MRNet starts successfully; -1 otherwise.

Init (back-end attach mode)

Synopsis

```
int Init(const char *TopologyFile, unsigned int numBackends,
         const char *ConnectionsFile, bool wait_for_BEs=true);

int Init(bool wait_for_BEs=true);
```

Description Starts the MRNet except the back-ends (back-end attach mode), and waits for these to connect. *TopologyFile* is the path to a configuration file that defines the desired process tree topology (not including the back-ends). *numBackends* is the number of back-ends that will be created outside the MRNet application. *ConnectionsFile* is the path to a file where the necessary information for the back-ends to connect to the network (hosts and ports of their parent processes) will be written to. *wait_for_BEs* is an optional argument (set by default). When set, the front-end will wait for all the back-ends to connect and then complete all the initializations. Otherwise, the user will have to call to Connect() later to complete the initialization.

If *TopologyFile*, *numBackends* and *ConnectionsFile* are not given, this information is read from the environment variables MRNAPP_TOPOLOGY, MRNAPP_NUM_BE and MRNAPP_BE_CONNECTIONS.

Return value Returns 0 if the MRNet starts successfully; -1 otherwise.

Connect

Synopsis

```
int Connect();
```

Description In the back-end attach instantiation mode, this is the second part of the Init() function. Init() can call this function automatically if specified, otherwise you have to call it manually. This is implemented to support the use-case where the front-end and the back-ends are threads of the same MPI process and you need to get the control back after the front-end initialization to distribute the pending connection information among the MPI tasks to start the back-ends.

Return value Returns 0 on success; -1 otherwise;

isConnectionsFileWritten

Synopsis

```
bool isConnectionsFileWritten();
```

Description This method exists to control in multi-threaded programs not to start parsing the file before all the necessary information has been totally dumped.

Return value Returns true if the necessary information for the back-ends to connect to the network (hosts and ports of their parent processes) has been written completely.

ConnectedBackEnds

Synopsis

```
int ConnectedBackEnds(void);
```

Return value Returns the number of back-ends that are connected to the network.

LoadProtocol

Synopsis

```
int LoadProtocol(Protocol *prot);
```

Description Loads the user-defined protocol *prot* for the front-end side of the MRNet.

Return value Returns 0 on success; -1 otherwise;

LoadFilter

Synopsis

```
int LoadFilter (string filter_name);
```

Description Looks for the filter shared object specified by *filter_name* (appending .so) in the paths specified with the environment variable MRNAPP_FILTER_PATH. If the filter is found, it is loaded into the network.

Return value Returns the filter identifier; or -1 if can not be found or loaded.

Dispatch

Synopsis

```
int Dispatch(string protID, int &status, Protocol *& prot);
int Dispatch(string protID, int &status);
```

Description Tells the back-ends the next protocol the network is going to execute and runs it. *prot_id* is the protocol identifier. *status* is a parameter by reference that captures the return code of the protocol. *prot* is an optional parameter that returns the instance of the protocol that was executed, that the user can use to retrieve results from the execution of the protocol.

Return value Returns 0 on success; -1 otherwise.

Shutdown

Synopsis

```
void Shutdown(void);
```

Description Notifies the back-ends to exit and shutdowns the MRNet.

isUp

Synopsis

```
bool isUp();
```

Return value Returns true if the network has started correctly and the front-end is ready to issue protocols.

B.4.2. Class BackEnd

Instances of this class are needed to make your back-end processes. This class provides basic methods to connect the leaves of the network.

Init (normal mode)

Synopsis

```
int Init(int argc, char *argv[]);
```

Description Starts the network back-end. argc and argv are the number of arguments and a NULL-terminating list of arguments that the back-end binary receives, respectively.

Return value Returns 0 on success; -1 otherwise.

Init (back-end attach mode)

Synopsis

```
int Init(int wRank, const char *connectionsFile);

int Init(int wRank, char *parHostname, int parPort, int parRank);

int Init(int wRank);
```

Description Starts the network attaching the pending back-ends (BE attach mode). *wRank* is the back-end rank identifier.

connectionsFile is the path to a file that contains all the parent's hosts and ports where each back-end has to connect.

Alternatively, you can manually provide the host, port and rank by setting *parHostname*, *parPort* and *parRank*. This variant was implemented to avoid stressing the filesystem by reading the connections file simultaneously from many back-ends.

If none of these arguments is provided, the path to the connections file is read from the environment variable MRNAPP_BE_CONNECTIONS.

Return value Returns 0 on success; -1 otherwise.

LoadProtocol

Synopsis

```
int LoadProtocol(Protocol *prot);
```

Description Loads a user-protocol for the back-end side of the MRNet.

Return value Returns 0 on success; -1 otherwise.

Loop

Synopsis

```
void Loop();

void Loop(callback_function preProtocol, callback_function postProtocol);
```

Description The back-end enters a loop waiting for requests from the front-end. When the front-end dispatches protocol, the back-end executes the counterpart back-end side of the same protocol. The loop exits when the front-end dispatches a message with tag TAG_EXIT (when calling Shutdown()).

If *preProtocol* and *postProtocol* are given, these callbacks are executed before and after the protocol is executed. This was introduced to prepare input data and retrieve results.

Shutdown

Synopsis

```
void Shutdown();
```

Description Gracefully disconnects the back-end end-point from the MRNet.

B.4.3. Class MRNetApp

Common class inherited both from FrontEnd and BackEnd. Provides common methods to identify the processes.

NumBackEnds

Synopsis

```
unsigned int NumBackEnds (void);
```

Description Queries the MRNet Network Topology for the total number of back-ends.

Return value Returns how many back-ends are in the network.

WhoAmI

Synopsis

```
unsigned int WhoAmI(bool return_network_id=false);
```

Description If *return_network_id* is set, returns the real ID for the backends in the network topology (range from 1000000 to 1000000+N). Otherwise, returns the logical ID for the remote back-ends (range from 0 to N).

Return value Returns the rank of the current MRNet process.

isFE

Synopsis

```
bool isFE (void);
```

Return value Returns true if the calling process is the network front-end.

isBE

Synopsis

```
bool isBE (void);
```

Return value Returns true if the calling process is a network back-end.

B.4.4. Class Protocol

All user-defined protocols extend this class through the inheritance of FrontProtocol and BackProtocol, that are necessary to define the logic executed at the different end-points of the network. When writing your FrontProtocol and BackProtocol objects, you must provide an implementation for the following methods.

ID

Synopsis

```
string ID (void);
```

Description Sets a textual identifier for the protocol (e.g. P̈INGPONG)̈. The identifier must coincide in the front-end side of the protocol (FrontProtocol object) and the back-end side of the protocol (BackProtocol object).

Setup

Synopsis

```
void Setup (void);
```

Description When a given protocol starts executing, its Setup() method will be executed first. In the Setup() method you have to register all the streams (and filters) that are going to be used during the execution of the protocol. In order to register streams, it is necessary to make calls to FrontProtocol::Register_Stream (in the front-end side of the protocol) and BackProtocol::Register_Stream (in the back-end side of the protocol), **in the same order**.

Run

Synopsis

```
int Run (void);
```

Description The protocol to execute has to be defined inside this method.

Return value Returns the return code of the protocol.

B.4.5. Class FrontProtocol

The front-end side of a protocol has to inherit this class, and implement the generic methods `ID`, `Setup` and `Run` (see B.4.4).

Barrier

```
int Barrier (void);
```

Description Blocks the front-end protocol until `BackProtocol::Barrier` is called by all back-ends.

Return value Returns 0 on success; -1 otherwise.

Register_Stream

Synopsis

```
STREAM * Register_Stream(int up_transfilter_id, int up_syncfilter_id);

STREAM * Register_Stream(string filter_name, int up_syncfilter_id);
```

Description Front-end wrapper for MRNet::Network::new_Stream that stores the newly created stream in a registration queue. When a protocol is loaded, all registered streams are published automatically to the back-ends. *up_transfilter_id* is the transformation filter to apply to data flowing upstream (the default is not to apply any filter, TFILTER_NULL). *up_syncfilter_id* is the synchronization filter to apply to upstream packets (the default is to wait for all children, SFILTER_WAITFORALL).

If *filter_name* is specified, the filter identified by *filter_name* is loaded into the network and linked to the new stream.

Return value Returns the new stream.

B.4.6. Class BackProtocol

The back-end side of a protocol has to inherit this class, and implement the generic methods ID(), Setup() and Run() (see B.4.4).

Barrier

```
int Barrier (void);
```

Description Blocks the calling back-end waiting for the remaining back-ends to call BackProtocol::Barrier() and the front-end to call FrontProtocol::Barrier().

Return value Returns 0 on success; -1 otherwise.

Register_Stream

Synopsis

```
void Register_Stream(STREAM *& new_stream);
```

Description Retrieves a new stream that was registered in the front-end. The streams have to be registered in the same order than in the front-end!

Return value Returns the stream that was registered in the front-end.

B.4.7. Class PendingConnections

This clas provides generic methods to exchange the connections information from the front-end to the back-ends outside of the MRNet application. Currently it supports distribution of this information through shared filesystems and through an MPI network.

PendingConnections

Synopsis

```
void PendingConnections (string ConnectionsFile);
```

Description Constructor that receives the file where the connections information has to be written (front-end), or from where has to be read (back-ends).

Write

Synopsis

```
  int Write(NETWORK *net, unsigned int numBackends);
```

Description This front-end method queries the connection information from the network and dumps it into a file to share with the back-ends via a shared filesystem.

Return value Returns 0 on success; -1 otherwise.

GetParentInfo

Synopsis

```
int GetParentInfo(int rank, char *phost, char *pport, char *prank);
```

Description This back-end method retrieves the *host*, *port* and *rank* where a given backend has to connect from the connections file.

Return value Returns 0 on success; -1 otherwise.

ParseForMPIDistribution

Synopsis

```
int ParseForMPIDistribution(int world_size, char *&sendbuf,
                             int *&sendcnts, int *&displs);
```

Description This back-end method parses the connections file and serializes the data into arrays to be distributed through MPI scatter. This method is meant for the use-case where the back-ends are spawned through MPI, and just one process reads the connection information and distributes the data to the rest through MPI. This was implemented to avoid stressing the filesystem because of many back-ends reading the same file simultaneously.

Parameter *world_size* is the total number of back-ends. *sendbuf* is the address of send buffer to store the data to send to each process. *sendcnt* is the address of the integer array to specify in entry i the number of elements to send to processor i. *displs* is the address of the integer array to specify in entry i the displacement (relative to sendbuf) from which to take the outgoing data to process i.

Return value Returns 0 on success; -1 otherwise.

B.5. Synapse wrappers

Synapse provides several wrappers to unify the lightweight API for BlueGene architectures and the standard C++ API. These wrappers are defined as macros in upper case. There are wrappers available for the basic MRNet objects Network, Stream and Packet:

- NETWORK

- NETWORK_PTR
- STREAM
- STREAM_PTR
- PACKET
- PACKET_PTR

There are also wrappers for the basic methods of these objects. The wrappers take the same arguments as the MRNet routines:

- STREAM_recv(stream, tag, data, block)
- STREAM_get_Id(stream)
- STREAM_send(stream, tag, format, args...)
- STREAM_flush(stream)
- STREAM_is_Closed(stream)
- STREAM_delete(stream)
- PACKET_unpack(p, fmt, args...)
- PACKET_new(p)
- PACKET_delete(p)
- NETWORK_recv(net, tag, data, stream, block)
- NETWORK_CreateNetworkBE(argc, argv)
- NETWORK_get_LocalRank(net)
- NETWORK_waitfor_ShutDown(net)
- NETWORK_delete(net)
- NETWORK_get_NumBackEnds(net, num_be)

And additionally provides a few communication primitives:

- `MRN_STREAM_SEND(stream, tag, format, args...)`

Broadcast to all back-ends.

- `MRN_STREAM_SEND_P2P(stream, be_list, tag, format, args...)`

Sends a message to the subset of back-ends in the stream specified in *be_list*.

- `MRN_STREAM_RECV(stream, tag, data, expected_tag)`

Receive from a specific stream (blocking receive).

- `MRN_STREAM_RECV_NONBLOCKING(stream, tag, data, expected_tag)`

Receive from a specific stream (non-blocking receive).

- `MRN_NETWORK_RECV(net, tag, data, expected_tag, stream, blocking)`

Receive from any stream.

B.6. Synapse configuration tool

Synapse provides a configuration tool that helps compiling and linking a MRNet-based application. It can be queried (for example, from a Makefile or building system) for the following information:

Synopsis

```
synapse-config <option>
```

Options

- –prefix: print Synapse installation directory
- –fe-cflags: prints pre-processor and compiler flags for the front-end
- –fe-libs: prints library linking information for the front-end
- –be-cflags: prints pre-processor and compiler flags for the back-ends
- –be-libs: prints library linking information for the back-ends
- –cp-cflags: prints pre-processor and compiler flags for the filters
- –libdir: prints the library directory
- –rpath: prints run-time search path flags for the shared libraries
- –libtool-rpath: prints the rpath flags used by libtool
- –mrnet: prints MRNet installation directory
- –version: prints version information

Appendix C

xtrack User Guide

C.1. Introduction

xtrack is a tool for comparing multiple experiments or different time intervals within the same experiment. xtrack takes as input the output produced by the ClusteringSuite tool [53] as well as the output produced by the Extrae On-line module when configured to run automatic structure detection analysis (refer to Annex A). The input of the tool is a series of clustering scatter-plots. Each clustering scatter-plot (or frame) is the result of clustering the computations of one execution of a parallel application with respect to selected performance metrics. Then, the clusters in one frame represent the main performance trends of the computations of the program. Comparing multiple frames, we can study how the behavior of the computing regions change between experiments. If we are changing a given parameter between experiments, this is useful to study the impact of this parameter in the program performance.

The tool has two parts: the tracking algorithm and the GUI. The *tracking algorithm* takes as input the sequence of frames and performs a "who-is-who" correlation between the clusters that appear in all the frames. To do this, the tool applies several heuristics that look for different characteristics that can distinguish certain clusters from the others. Then, the results of the tracking algorithm can be visualized with the *xtrack GUI*.

In this document we briefly introduce the user to both parts of the tool, and show how to use them and the different settings available.

C.2. The Tracking tool

The first step to perform this comparative analysis is to apply the tracking algorithm to the sequence of frames that results from the application of cluster analyses to different traces (or sub-traces). To do so, the tool takes as input the list of clustered traces and some optional arguments that define which heuristics will be used to do the tracking. A detailed description of each of the available heuristics can be found here [10]. By default, the tool tries to apply all the heuristics that are applicable with the information comprised in the trace, as well as default threshold levels that are already usable.

SYNTAX

```
tracking [OPTIONS] [-l LIST | TRACE1 TRACE2 ...]
```

OPTIONS

```
-a MIN_SCORE
    Minimum SPMD score to use the alignment tracker
-c CALLER_LEVEL
    Enable the callers heuristic at the specified stack depth.
-d MAX_DISTANCE
    Maximum Epsilon distance to use the cross-classifier tracker
-m MIN_TIME_PCT
    Discard the clusters below the given duration time percentage.
-s DIM1,DIM2...
    Select the dimensions to scale with the number of tasks.
-o OUTPUT_PREFIX
    Set the prefix for all the output files.
-r Enable the trace reconstruction with tracked clusters.
-t THRESHOLD
    Minimum likeliness percentage in order to match two clusters
    (special values: all | first).
-v[v] Run in verbose mode (-vv for extra debug messages).
-x CLUSTERING_CONFIG_XML
    Specify the clustering configuration to automatically
    cluster the traces.
```

As a result, the tool generates:

- Trajectory lines for all the clusters in the frames, that show how the behavior of the clusters change across experiments.
- Recolored frames: a new set of scatter-plots where the identifiers of the clusters and their colors have been changed to make them match across experiments for easy comparison.
- Tracked traces: the input trace is reconstructed, changing the clustering events so that the clusters identifiers and their colors are the same in all experiments for easy comparison.
- The **.xtrack* definition file. This file contains all the information about “who-is-who” between experiments, and is the input for the xtrack GUI.

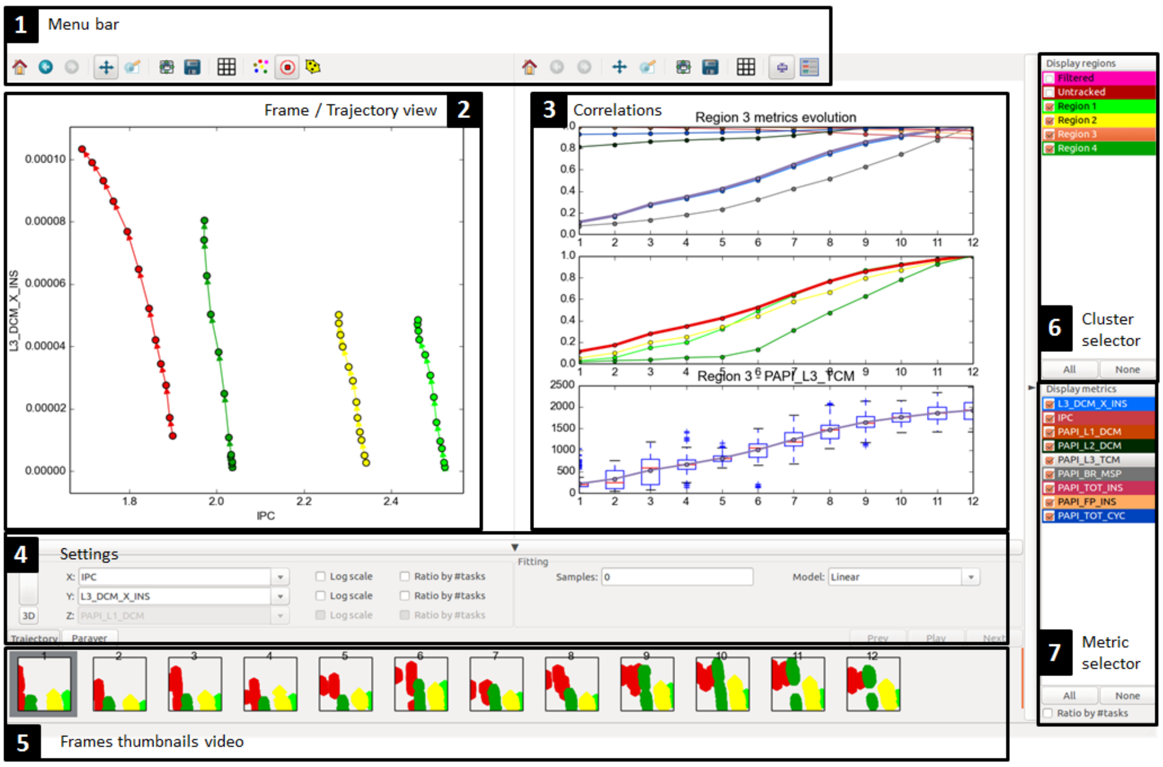


Figure C.1.: xtrack GUI

C.3. The xtrack GUI

The xtrack GUI takes as input the results of applying the tracking algorithm, and displays a graphical comparison of the different experiments. Figure C.1 shows the main view of the tool.

C.3.1. Menu bar

The menu bar contains controls to manipulate the plotting areas 2 and 3. From left to right, the common controls in both bars are:

- **Reset** the plot to the default view.
- **Undo** the last pan or zoom.
- **Redo** the last pan or zoom.
- **Pan** the graph towards any direction.
- **Zoom-in** dragging a box with the left mouse button, or **zoom-out** with the right mouse button.
- Adjust the plot **margins**.

- **Save** the visible plot.

The specific controls for the left menu bar (that controls plotting area 2) are:

- Show/hide a **grid** in the plot.
- Show/hide the **centroids** of the clusters.
- Show/hide the **perimeters** of the clusters.

And the specific controls for the right menu bar (that controls plotting area 3) are:

- Show/hide **boxplots** for dispersion plot in pane 3 (dispersion graph).
- Show/hide plots **legends**.

C.3.2. Frame/Trajectory view

This is the main view of the tool. When the *frame view* setting is selected in pane 4, it displays the cluster results for one single frame of the input sequence of experiments, the one that is selected in pane 5. When the *trajectory view* is selected in pane 4, it displays instead all the clusters from all the frames at the same time, and draws trajectory lines that show how the clusters are moving from one experiment to the next. In the image, the trajectory view is set by default.

C.3.3. Correlations

This pane shows correlations for different clusters and metrics across the sequence of experiments (X-axis). The top plot shows a correlation of all metrics that are checked in pane 7, for the cluster that is selected in pane 6. The middle plot shows a correlation of the selected metric in pane 7, for all the clusters that are checked in pane 6. Since these two plots can display different metrics or very different ranges for the same metric in the Y-axis, the value for this axis is normalized. The bottom plot shows the dispersion of the selected metric in pane 7 for the selected cluster in pane 6. In this case, the Y-axis shows real values.

C.3.4. Settings

The settings is divided in two parts. The *Axes* configuration (left) determine the metrics that are used to plot the performance space in pane 2. It is possible to select any performance counter that was used to cluster the trace or extrapolated, and also to select a third metric to change the view into a 3D plot. The *Log scale* checkbox can be ticked to draw each axis in logarithmic scale. The *Ratio by #tasks* can be selected to weight that axis metric with respect to the number of tasks that were used in the selected frame. This is useful when the metric is related to the number of instructions executed (e.g. total instructions, floating point instructions, etc.) for doing strong scaling or weak scaling comparisons, and studying code replication issues.

The *Fitting* settings (right) are used to do predictions based on the experiments we have. You can select a fitting model between linear, quadratic, cubic, logarithmic or log-linear (right), and a number of experiments to predict (left), and all the correlation plots in pane 3 will extend their X-axis to predict how the series will continue according to the selected model.

C.3.5. Frame thumbnails video

This pane shows a carousel of frames, where each frame is the clustering result of every single experiment. This representation provides a quick view on the changes across experiments. Also, this pane allows to select a single frame to be inspected in detail in pane 2.

C.3.6. Cluster selector

Ticking the checkbox of each cluster we can control whether that cluster has to be displayed/hidden in the plots in panes 2 and 3. Also selecting a single cluster from the list, changes the plots in pane 3 to display the metrics correlations for the selected cluster only.

C.3.7. Metric selector

Analogously, ticking the checkbox of each metric we can control whether that metric has to be displayed/hidden in the plots in pane 3. Also, selecting a single metric from the list, changes the plots in pane 3 to display the correlations of clusters for the selected metric only.

Bibliography

[1] J. Schaye, R. A. Crain, R. G. Bower, M. Furlong, M. Schaller, T. Theuns, C. Dalla Vecchia, C. S. Frenk, I. G. McCarthy, J. C. Helly, A. Jenkins, Y. M. Rosas-Guevara, S. D. M. White, M. Baes, C. M. Booth, P. Camps, J. F. Navarro, Y. Qu, A. Rahmati, T. Sawala, P. A. Thomas, and J. Trayford. **The EAGLE project: Simulating the Evolution and Assembly of Galaxies and their Environments**. *Monthly Notices of the Royal Astronomical Society*, **446**:521–554, January 2015.

[2] **Human Brain Project**. `http://www.humanbrainproject.eu`.

[3] D. Torrents. **Somatic mutation finder (SMuFin) to identify single-nucleotide variants (SNVs) and structural variants in tumor DNA**. SciBX: Science-Business eXchange.

[4] **TOP500 Supercomputer Sites**. `http://www.top500.org`.

[5] J. J. Dongarra. **Performance of Various Computers Using Standard Linear Equations Software**. *SIGARCH Comput. Archit. News*, **20**(3):22–44, June 1992.

[6] P. E. Ross. **Why CPU Frequency Stalled**. *IEEE Spectr.*, **45**(4):72–72, April 2008.

[7] G. Llort, J. Gonzalez, H. Servat, J. Gimenez, and J. Labarta. **On-line detection of large-scale parallel application's structure**. In *24th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010, Atlanta, Georgia, USA*, pages 1–10, 2010.

[8] G. Llort, M. Casas, H. Servat, K. Huck, J. Gimenez, and J. Labarta. **Trace Spectral Analysis Towards Dynamic Levels of Detail**. In *17th IEEE International Conference on Parallel and Distributed Systems, ICPADS 2011, Tainan, Taiwan*, pages 332 – 339, 2011.

[9] G. Llort, H. Servat, J. González, J. Giménez, and J. Labarta. **On the Usefulness of Object Tracking Techniques in Performance Analysis**. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 29:1–29:11, New York, NY, USA, 2013. ACM.

[10] G. Llort, H. Servat, J. Gonzalez, J. Gimenez, and J. Labarta. **Studying Performance Changes with Tracking Analysis**. In *Tools for High Performance Computing 2014*, pages 175–209. Springer International Publishing, 2015.

[11] G. Llort, J. González, H. Servat, J. Gimenez, and J. Labarta. **Distributed tree-based implementation of DBSCAN cluster algorithm for on-line performance analysis**. Technical report, Polytechnic University of Catalonia - BarcelonaTech, 2015.

[12] G. Llort, M. Casas, M. Garcia, V. Lopez, H. Servat, J. Gimenez, and J. Labarta. **On-line Active Measurement to Perform All-In-One Series of Experiments**. Technical report, Polytechnic University of Catalonia - BarcelonaTech, 2015.

[13] H. El-Rewini and M. Abd-El-Barr. *Advanced Computer Architecture and Parallel Processing (Wiley Series on Parallel and Distributed Computing)*. Wiley-Interscience, 2005.

[14] **RFC 1057 - RPC: Remote Procedure Call Protocol specification**. `http://www.faqs.org/rfcs/rfc1057.html`.

[15] **CORBA: Object Management Group specifications**. `http://www.omg.org/spec/index.htm`.

[16] **Whitepaper: Java Remote Method Invocation - Distributed Computing for Java**. Java RMI - Oracle Documentation.

[17] **D-Bus Documentation**. `http://dbus.freedesktop.org`.

[18] **MPI: A Message-Passing Interface Standard**. `http://www.mpi-forum.org`.

[19] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, S. Kumar, E. Lusk, R. Thakur, and J. L. Träff. **MPI on a Million Processors**. In *Proceedings of the 16th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 20–30, Berlin, Heidelberg, 2009. Springer-Verlag.

[20] **The OpenMP API specification for parallel programming**. `http://www.openmp.org`.

[21] **The OmpSs Programming Model**. `http://pm.bsc.es/ompss`.

[22] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. **A Portable Programming Interface for Performance Evaluation on Modern Processors**. *The International Journal of High Performance Computing Applications*, **14**:189–204, 2000.

[23] **Performance Co-Pilot Programmer's Guide**. `http://www.pcp.io`.

[24] 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.

[25] A. D. 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 *Proceedings of the 2011 International Conference on Parallel Processing*, ICPP '11, pages 176–185, Washington, DC, USA, 2011. IEEE Computer Society.

[26] E. Karrels and E. Lusk. **Performance Analysis of MPI Programs**. In *In: Workshop on Environments and Tools for Parallel Scientific Computing*, 1994.

[27] A. Eichenberger, J. Mellor-Crummey, M. Schulz, M. Wong, N. Copty, R. Dietrich, X. Liu, E. Loh, and D. Lorenz. **OMPT: An OpenMP Tools Application Programming Interface for Performance Analysis**. In A. Rendell, B. Chapman, and M. Müller, editors, *OpenMP in the Era of Low Power Devices and Accelerators*, **8122** of *Lecture Notes in Computer Science*, pages 171–185. Springer Berlin Heidelberg, 2013.

[28] **The libunwind project**. `http://www.nongnu.org/libunwind`.

[29] **Stackwalker API**. `http://www.dyninst.org/stackwalker`.

[30] Microsoft. **Examining and Tuning Disk Performance**. `https://technet.microsoft.com/en-us/library/cc938959.aspx`.

[31] M. Hähnel, B. Döbel, M. Völp, and H. Härtig. **Measuring Energy Consumption for Short Code Paths Using RAPL**. *SIGMETRICS Perform. Eval. Rev.*, **40**(3):13–17, January 2012.

[32] S. L. Graham, P. B. Kessler, and M. K. McKusick. **Gprof: A Call Graph Execution Profiler**. *SIGPLAN Not.*, **17**(6):120–126, June 1982.

[33] B. Mohr, A. D. Malony, S. Shende, and F. Wolf. **Towards a Performance Tool Interface for OpenMP: An Approach Based on Directive Rewriting**. In *In Proceedings of the Third Workshop on OpenMP (EWOMP'01*, 2001.

[34] D. Lorenz, B. Mohr, C. Rössel, D. Schmidl, and F. Wolf. **How to Reconcile Event-Based Performance Analysis with Tasking in OpenMP**. In M. Sato, T. Hanawa, M. Müller, B. Chapman, and B. de Supinski, editors, *Beyond Loop Level Parallelism in OpenMP: Accelerators, Tasking and More*, **6132** of *Lecture Notes in Computer Science*, pages 109–121. Springer Berlin Heidelberg, 2010.

[35] B. Buck and J. K. Hollingsworth. **An API for Runtime Code Patching**. *The International Journal of High Performance Computing Applications*, **14**:317–329, 2000.

[36] L. DeRose, K. Ekanadham, J. K. Hollingsworth, and S. Sbaraglia. **SIGMA: A Simulator Infrastructure to Guide Memory Analysis**. *SC*, **00**:1, 2002.

[37] V. J. Reddi, A. Settle, D. A. Connors, and R. S. Cohn. **PIN: A Binary Instrumentation Tool for Computer Architecture Research and Education**. In *Proceedings of the 2004 Workshop on Computer Architecture Education: Held in Conjunction with the 31st International Symposium on Computer Architecture*, WCAE '04, New York, NY, USA, 2004. ACM.

[38] H. Servat, G. Llort, J. Giménez, and J. Labarta. **Detailed Performance Analysis Using Coarse Grain Sampling**. In *PROPER*, 2009.

[39] T. Ilsche. **Combining Instrumentation and Sampling for Trace-based Application Performance Analysis**. In *In proceedings of the 8th International Parallel Tools Workshop (IPTW'14)*, Stuttgart, Germany, 2014.

[40] **20 Years of PLDI (1979 - 1999): A Selection**. `http://www.cs.utexas.edu/users/mckinley/20-years.html`.

[41] J. Mellor-Crummey. **HPCToolkit: Multi-Platform Tools for Profile-Based Performance Analysis.** In *5th International Workshop on Automatic Performance Analysis (APART)*, November 2003.

[42] M. Schulz, J. Galarowicz, D. Maghrak, W. Hachfeld, D. Montoya, and S. Cranford. **Open SpeedShop: An Open Source Infrastructure for Parallel Performance Analysis**. *Sci. Program.*, **16**(2-3):105–121, April 2008.

[43] M. Gerndt, K. Fürlinger, and E. Kereku. **Periscope: Advanced Techniques for Performance Analysis**. In *PARCO*, pages 15–26, 2005.

[44] **mpiP: Lightweight, Scalable MPI Profiling**. `http://mpip.sourceforge.net`.

[45] S. S. Shende and A. D. Malony. **The TAU Parallel Performance System**. *Int. J. High Perform. Comput. Appl.*, **20**(2):287–311, 2006.

[46] D. Becker, F. Wolf, W. Frings, M. Geimer, B. Wylie, and B. Mohr. **Automatic Trace-Based Performance Analysis of Metacomputing Applications**. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Long Beach, California, 2007.

[47] M. Geimer, P. Saviankou, A. Strube, Z. Szebenyi, F. Wolf, and B. J. N. Wylie. **Further improving the scalability of the SCALASCA Toolset**. In *Proceedings of the 10th international conference on Applied Parallel and Scientific Computing - Volume 2*, PARA'10, pages 463–473, Berlin, Heidelberg, 2012. Springer-Verlag.

[48] Z. Szebenyi, F. Wolf, and B. Wylie. **Space-efficient time-series call-path profiling of parallel applications**. In *High Performance Computing Networking, Storage and Analysis, Proceedings of the Conference on*, pages 1–12, Nov 2009.

[49] W. E. Nagel, A. Arnold, M. Weber, H. C. Hoppe, and K. Solchenbach. **VAMPIR: Visualization and Analysis of MPI Resources**. *Supercomputer*, **12**(1):69–80, 1996.

[50] A. Knüpfer, C. Rössel, D. Mey, S. Biersdorff, K. Diethelm, D. Eschweiler, M. Geimer, M. Gerndt, D. Lorenz, A. Malony, W. Nagel, Y. Oleynik, P. Philippen, P. Saviankou, D. Schmidl, S. Shende, R. Tschüter, M. Wagner, B. Wesarg, and F. Wolf. **Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir**. In H. Brunst, M. S. Müller, W. E. Nagel, and M. M. Resch, editors, *Tools for High Performance Computing 2011*, pages 79–91. Springer Berlin Heidelberg, 2012.

[51] D. Mey, S. Biersdorf, C. Bischof, K. Diethelm, D. Eschweiler, M. Gerndt, A. Knüpfer, D. Lorenz, A. Malony, W. Nagel, Y. Oleynik, C. Rössel, P. Saviankou, D. Schmidl, S. Shende, M. Wagner, B. Wesarg, and F. Wolf. **Score-P: A Unified Performance Measurement System for Petascale Applications**. In C. Bischof, H.-G. Hegering, W. E. Nagel, and G. Wittum, editors, *Competence in High Performance Computing 2010*,

pages 85–97. Springer Berlin Heidelberg, 2012.

[52] 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**. In *Applications, Tools and Techniques on the Road to Exascale Computing / ed.: K. De Bosschere, E.H. D'Hollander, G.R. Joubert, David Padua, Frans Peters, Mark Sawyer, IOS Press, 2012, Advances in Parallel Computing, Vol. 22. - 978-1-61499-040-6. - S. 481 - 490*, 2012. Record converted from VDB: 12.11.2012.

[53] **BSC Tools**. `http://www.bsc.es/paraver`.

[54] B. S. Center. **Paraver Trace Generation Manual. TRACEFILE DESCRIPTION**. `http://www.bsc.es/paraver`.

[55] **Oracle Solaris Studio**. `http://www.oracle.com/technetwork/server-storage/solarisstudio/overview/index.html`.

[56] N. R. Tallent, J. Mellor-Crummey, M. Franco, R. Landrum, and L. Adhianto. **Scalable Fine-grained Call Path Tracing**. In *Proceedings of the International Conference on Supercomputing*, ICS '11, pages 63–74, New York, NY, USA, 2011. ACM.

[57] *IBM System Blue Gene Solution: Performance Analysis Tools. Chapter 5: CPU profiling using Xprofiler.* IBM Redbooks, 24 November 2008.

[58] R. Bell, A. Malony, and S. Shende. **ParaProf: A Portable, Extensible, and Scalable Tool for Parallel Performance Profile Analysis**. In H. Kosch, L. Böszörményi, and H. Hellwagner, editors, *Euro-Par 2003 Parallel Processing*, **2790** of *Lecture Notes in Computer Science*, pages 17–26. Springer Berlin Heidelberg, 2003.

[59] K. Huck, A. Malony, R. Bell, and A. Morris. **Design and Implementation of a Parallel Performance Data Management Framework**. In *Parallel Processing, 2005. ICPP 2005. International Conference on*, pages 473–482, June 2005.

[60] 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 *In Parallel Computing: Architectures, Algorithms and Applications: Proc. Parallel Computing (ParCo'07, Jülich/Aachen*, pages 645–652. IOS Press.

[61] H. Brunst and B. Mohr. **Performance Analysis of Large-scale OpenMP and Hybrid MPI/OpenMP Applications with Vampir NG**. 2005.

[62] 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.

[63] J. K. Hollingsworth, B. P. Miller, and J. Cargille. **Dynamic Program Instrumentation for Scalable Performance Tools**. Technical Report CS-TR-1994-1207, 1994.

[64] **Dyninst API web site**. `http://www.dyninst.org`.

[65] A. Srivastava and A. Eustace. **ATOM: A system for building customized program analysis tools**. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 196–205, New York, NY, USA, 1994. ACM Press.

[66] G. Llort. **OMPItrace over SiGMA**. Technical report, Universidad Politècnica de Catalunya. Departamento de Arquitectura de Computadores., 2006.

[67] M. Gerndt and A. Krumme. **A Rule-based Approach for Automatic Bottleneck Detection in Programs on Shared Virtual Memory Systems**. In *HIPS '97: Proceedings of the 1997 Workshop on High-Level Programming Models and Supportive Environments (HIPS '97)*, page 93, Washington, DC, USA, 1997. IEEE Computer Society.

[68] M. Schulz, J. May, and J. C. Gyllenhaal. **DynTG: A tool for interactive, dynamic**

instrumentation. In *In 5th International Conference on Computation Science,* **Part II**, pages 140–148, Atlanta, GA, United States, May 22-25 2005.

[69] F. Mueller, T. Mohan, B. de Supinski, S. McKee, and A. Yoo. **Partial data traces: Efficient generation and representation**, 2001.

[70] J. Marathe, F. Mueller, T. Mohan, B. R. de Supinski, S. A. McKee, and A. Yoo. **METRIC: tracking down inefficiencies in the memory hierarchy via binary rewriting**. In *CGO '03: Proceedings of the international symposium on Code generation and optimization*, pages 289–300, Washington, DC, USA, 2003. IEEE Computer Society.

[71] A. Knüpfer. **A new data compression technique for event based program traces**. In *Proceedings of the ICCS 2003,* **2659** of *Springer LNCS*, pages 956–965, 2003.

[72] A. Knüpfer and W. E. Nagel. **Compressible memory data structures for event-based trace analysis**. *Future Gener. Comput. Syst.*, **22**(3):359–368, 2006.

[73] A. Knüpfer, H. Brunst, and W. E. Nagel. **High Performance Event Trace Visualization**. In *PDP '05: Proceedings of the 13th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP'05)*, pages 258–263, Washington, DC, USA, 2005. IEEE Computer Society.

[74] A. D. Malony and W. E. Nagel. **Open trace—The open trace format (OTF) and open tracing for HPC**. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 24, New York, NY, USA, 2006. ACM Press.

[75] P. A. González. **Implementación de un traductor de OTF a PRV**. Facultad de Informática de Barcelona, 2007.

[76] M. Wagner, A. Knupfer, and W. E. Nagel. **Enhanced Encoding Techniques for the Open Trace Format 2**. *Procedia Computer Science*, **9**(0):1979 – 1987, 2012. Proceedings of the International Conference on Computational Science, {ICCS} 2012.

[77] M. Noeth, P. Ratn, F. Mueller, M. Schulz, and B. R. de Supinski. **ScalaTrace: Scalable Compression and Replay of Communication Traces for High-Performance Computing**. *J. Parallel Distrib. Comput.*, **69**(8):696–710, August 2009.

[78] P. Havlak and K. Kennedy. **An Implementation of Interprocedural Bounded Regular Section Analysis**. *IEEE Trans. Parallel Distrib. Syst.*, **2**(3):350–360, July 1991.

[79] O. Zaki, E. Lusk, W. Gropp, and D. Swider. **Toward Scalable Performance Visualization with Jumpshot**. *Int. J. High Perform. Comput. Appl.*, **13**(3):277–288, 1999.

[80] A. Chan, W. Gropp, and E. Lusk. **Scalable Log Files for Parallel Program Trace Data - DRAFT**. Technical Report IL 60439, Argonne National Laboratory, Argonne, 2000.

[81] D. Kranzlmüller, M. Scarpa, and J. Volkert. **DeWiz - A Modular tool Architecture for Parallel Program Analysis**. In *Parallel Processing, 9th International Euro-Par 2003. Proceedings*, pages 74–80, 2003.

[82] C. Schaubschläger, D. Kranzlmüller, and J. Volkert. **Event-based Program Analysis with DeWiz**, 2003.

[83] C. Glasner, E. Spiegl, and J. Volkert. **PARADIS: Analysis of Transaction-Based Applications in Distributed Environments**. In *International Conference on Computational Science (2)*, pages 124–131, 2005.

[84] 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.

[85] B. Mohr and F. Wolf. **KOJAK - A Tool Set for Automatic Performance Analysis of Parallel Programs**. In *Proceedings of the International Conference on Parallel and Distributed Computing (Euro-Par 2003)*, pages 1301–1304, Klagenfurt, Austria, August 2003.

[86] S. Moore, F. Wolf, B. Mohr, and J. Dongarra. **Improving Time to Solution with Automated Performance Analysis**. In *Proceedings of the 2nd Workshop on Productivity and Performance in High-End Computing (PPHEC-05)*, pages 20–26. Productivity and Performance in High-End Computing, 2005.

[87] P. C. Roth, D. C. Arnold, and B. P. Miller. **MRNet: A Software-Based Multicast/Reduction Network for Scalable Tools**. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 21, Washington, DC, USA, 2003. IEEE Computer Society.

[88] A. D. Malony and A. Nataraj. **TAU meets Dyninst and MRNet: A long-term and short-term affair**. Paradyn / Condor Week, May 2007.

[89] D. Arnold, D. Ahn, B. de Supinski, G. Lee, B. Miller, and M. Schulz. **Stack Trace Analysis for Large Scale Debugging**. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–10, March 2007.

[90] F. Wolf and B. Mohr. **Automatic Performance Analysis of MPI Applications Based on Event Traces**. In *Euro-Par '00: Proceedings from the 6th International Euro-Par Conference on Parallel Processing*, pages 123–132, London, UK, 2000. Springer-Verlag.

[91] F. Wolf and B. Mohr. **Automatic performance analysis of hybrid MPI/OpenMP applications**. *J. Syst. Archit.*, **49**(10-11):421–439, 2003.

[92] F. Wolf and B. Mohr. **EARL - A Programmable and Extensible Toolkit for Analyzing Event Traces of Message Passing Programs**. In *HPCN Europe '99: Proceedings of the 7th International Conference on High-Performance Computing and Networking*, pages 503–512, London, UK, 1999. Springer-Verlag.

[93] A. Espinosa, T. Margalef, and E. Luque. **Automatic Detection of Parallel Program Performance Problems**. In *VECPAR*, pages 365–377, 1998.

[94] J. Jorba, T. Margalef, and E. Luque. **Performance Analysis of Parallel Applications with KappaPI 2**. In *Parallel Computing: Current & Future Issues of High-End Computing, Proc. of PARCO 2005*, **33** of *NIC*, pages 155–162, Research Centre Jülich, 2005. John von Neumann Institute for Computing.

[95] T. Fahringer, M. Gerndt, G. R. Jesper, and L. Traff. **Specification of Performance Problems in MPI Programs with ASL**. In *ICPP '00: Proceedings of the 2000 International Conference on Parallel Processing*, page 51, Washington, DC, USA, 2000. IEEE Computer Society.

[96] T. Fahringer, A. Jugravu, S. Pllana, R. Prodan, J. Clovis Seragiotto, and H.-L. Truong. **ASKALON: A Tool set for Cluster and Grid computing: Research Articles**. *Concurr. Comput. : Pract. Exper.*, **17**(2-4):143–169, 2005.

[97] H. Truong and T. Fahringer. **SCALEA: A Performance Analysis Tool for Distributed and Parallel Programs**, 2002.

[98] T. Fahringer and J. Clovis Seragiotto. **Aksum: a performance analysis tool for parallel and distributed applications**. pages 189–208, 2004.

[99] S. Pllana, I. Brandic, and S. Benkner. **Performance Modeling and Prediction of Parallel and Distributed Computing Systems: A Survey of the State of the Art**. In *Complex, Intelligent and Software Intensive Systems, 2007. CISIS 2007. First International Conference on*, pages 279–284, April 2007.

[100] T. Hoefler, W. Gropp, W. Kramer, and M. Snir. **Performance Modeling for Systematic Performance Tuning**. In *State of the Practice Reports*, SC '11, pages 6:1–6:12, New York, NY, USA, 2011. ACM.

[101] A. Calotoiu, T. Hoefler, M. Poke, and F. Wolf. **Using Automated Performance Modeling to Find Scalability Bugs in Complex Codes**. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 45:1–45:12, New York, NY, USA, 2013. ACM.

[102] A. Bhattacharyya and T. Hoefler. **PEMOGEN: Automatic Adaptive Performance Modeling During Program Runtime**. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 393–404, New York, NY, USA, 2014. ACM.

[103] O. Morajko, A. Morajko, T. Margalef, and E. Luque. **On-Line Performance Modeling for MPI Applications**. In *Proceedings of the 14th International Euro-Par Conference on Parallel Processing*, Euro-Par '08, pages 68–77, Berlin, Heidelberg, 2008. Springer-Verlag.

[104] A. Morajko, O. Morajko, T. Margalef, and E. Luque. **MATE: Dynamic Performance Tuning Environment**. In M. Danelutto, M. Vanneschi, and D. Laforenza, editors, *Euro-Par 2004 Parallel Processing*, **3149** of *Lecture Notes in Computer Science*, pages 98–107. Springer Berlin Heidelberg, 2004.

[105] S. Lee, J. S. Meredith, and J. S. Vetter. **COMPASS: A Framework for Automated Performance Modeling and Prediction**. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS '15, pages 405–414, New York, NY, USA, 2015. ACM.

[106] K. L. Spafford and J. S. Vetter. **Aspen: A Domain Specific Language for Performance Modeling**. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 84:1–84:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[107] C. Rosas, J. Giménez, and J. Labarta. **Scalability prediction for fundamental performance factors**. *Supercomputing frontiers and innovations*, **1**(2), 2014.

[108] O. Y. Nickolayev, P. C. Roth, Daniel, 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.

[109] D. A. Reed, R. A. Aydt, R. J. Noe, P. C. Roth, K. A. Shields, B. W. Schwartz, and L. F. Tavera. **Scalable Performance Analysis: The Pablo Performance Analysis Environment**. In *Proceedings of the Scalable parallel libraries conference*, pages 104–113. IEEE Computer Society, 1993.

[110] 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.

[111] H. Abdi and L. J. Williams. **Principal component analysis**. *Wiley Interdisciplinary Reviews: Computational Statistics*, **2**(4):433–459, 2010.

[112] R. L. Gorsuch. *Factor Analysis*. 1983.

[113] 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. IEEE Computer Society.

[114] J. González, J. Giménez, and J. Labarta. **Automatic Detection of Parallel Applications Computation Phases**. In *IPDPS'09: 23rd IEEE International Parallel and Distributed Processing Symposium*, 2009.

[115] M. Ester, H. peter Kriegel, J. S, and X. Xu. **A density-based algorithm for discovering clusters in large spatial databases with noise**. pages 226–231. AAAI Press, 1996.

[116] F. Freitag, J. Corbalan, and J. Labarta. **A Dynamic Periodicity Detector: Application to Speedup Computation**. In *IPDPS '01: Proceedings of the 15th International Parallel and Distributed Processing Symposium (IPDPS'01)*, page 10002.1, Washington, DC, USA, 2001. IEEE Computer Society.

[117] J. Corbalán. *Coordinated Scheduling and Dynamic Performance Analysis in Multiprocessor Systems*. PhD thesis, Universitat Politècnica de Catalunya, 2002. http://people.ac.upc.es/juli/thesis.pdf.

[118] M. Casas and J. Labarta. **Automatic analysis of parallelization efficiency of MPI**

applications. Submitted to "SC '07: 2007 ACM/IEEE conference on Supercomputing", 2007.

[119] A. Knüpfer, B. Voigt, W. E. Nagel, and H. Mix. **Visualization of Repetitive Patterns in Event Traces**. In *Proceedings of Para'06*, 2006.

[120] J. Labarta, J. Giménez, E. Martínez, P. González, H. Servat, G. Llort, and X. Aguilar. **Scalability of Visualization and Tracing Tools**. In *Parallel Computing: Current & Future Issues of High-End Computing, Proc. of PARCO 2005*, **33** of *NIC*, pages 869–876, Research Centre Jülich, 2005. John von Neumann Institute for Computing.

[121] X. Xu, J. Jäger, and H.-P. Kriegel. **A Fast Parallel Clustering Algorithm for Large Spatial Databases**. *Data Min. Knowl. Discov.*, **3**(3):263–290, September 1999.

[122] Y. He, H. Tan, W. Luo, H. Mao, D. Ma, S. Feng, and J. Fan. **MR-DBSCAN: An Efficient Parallel Density-Based Clustering Algorithm Using MapReduce**. In *Proceedings of the 2011 IEEE 17th International Conference on Parallel and Distributed Systems*, ICPADS '11, pages 473–480, Washington, DC, USA, 2011. IEEE Computer Society.

[123] B.-R. Dai and I.-C. Lin. **Efficient Map/Reduce-Based DBSCAN Algorithm with Optimized Data Partition**. In *Proceedings of the 2012 IEEE Fifth International Conference on Cloud Computing*, CLOUD '12, pages 59–66, Washington, DC, USA, 2012. IEEE Computer Society.

[124] M. A. Patwary, D. Palsetia, A. Agrawal, W.-k. Liao, F. Manne, and A. Choudhary. **A New Scalable Parallel DBSCAN Algorithm Using the Disjoint-set Data Structure**. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 62:1–62:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[125] B. Welton, E. Samanas, and B. P. Miller. **Mr. Scan: Extreme Scale Density-based Clustering Using a Tree-based Network of GPGPU Nodes**. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 84:1–84:11, New York, NY, USA, 2013. ACM.

[126] F. Song, F. Wolf, N. Bhatia, J. Dongarra, and S. Moore. **An Algebra for Cross-Experiment Performance Analysis**. In *Proceedings of the 2004 International Conference on Parallel Processing*, ICPP '04, pages 63–72, Washington, DC, USA, 2004. IEEE Computer Society.

[127] R. Miceli, G. Civario, A. Sikora, E. César, M. Gerndt, H. Haitof, C. Navarrete, S. Benkner, M. Sandrieser, L. Morin, and F. Bodin. **AutoTune: A Plugin-Driven Approach to the Automatic Tuning of Parallel Applications**. In P. Manninen and P. Öster, editors, *Applied Parallel and Scientific Computing*, **7782** of *Lecture Notes in Computer Science*, pages 328–342. Springer Berlin Heidelberg, 2013.

[128] C. Ţăpuş, I.-H. Chung, and J. K. Hollingsworth. **Active Harmony: Towards Automated Performance Tuning**. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, SC '02, pages 1–11, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[129] **Open MPI: Open Source High Performance Computing**. `http://www.open-mpi.org`.

[130] **Intel® MPI Library Reference Manual**. `https://software.intel.com/en-us/articles/intel-mpi-library-documentation`.

[131] **The IBM Parallel Environment (PE)**. IBM Redbooks, September 2013.

[132] A. K. Jain, M. N. Murty, and P. J. Flynn. **Data Clustering: A Review**. *ACM Comput. Surv.*, **31**(3):264–323, September 1999.

[133] P. Bose and T. M. Conte. **Performance Analysis and Its Impact on Design**. *Computer*, **31**(5):41–49, May 1998.

[134] **CPMD: Carr-Parrinello Molecular Dynamics**. `http://www.cpmd.org`.

[135] **GROMACS: GROningen MAchine for Chemical Simulations**. `http://www.gromacs.org`.

[136] **SPEC MPI2007 Benchmark Suite**. `http://www.spec.org/mpi2007`.

[137] J. C. Hayes, M. L. Norman, R. A. Fiedler, J. O. Bordner, P. S. Li, S. E. Clark, A. ud Doula, and M.-M. M. Low. **Simulating Radiating and Magnetized Flows in Multiple Dimensions with ZEUS-MP**. *The Astrophysical Journal Supplement Series*, **165**(1):188, 2006.

[138] **SPECFEM3D**. `https://geodynamics.org/cig/software/specfem3d`.

[139] **CPI analysis on POWER5**. `http://www.ibm.com/developerworks/library/pa-cpipower1`.

[140] M. Casas, R. Badia, and J. Labarta. **Automatic structure extraction from MPI applications tracefiles**. In *In Euro-Par Conference*, August 2007.

[141] **NAS Parallel Benchmarks**. `http://www.nas.nasa.gov/Software/NPB`.

[142] **PEPC - Pretty Efficient Parallel Coulomb-solver**. `http://www2.fz-juelich.de/jsc/pepc`.

[143] M. Casas, R. Badia, and J. Labarta. **Automatic analysis of speedup of MPI applications**. In *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*, pages 349–358, New York, NY, USA, 2008. ACM.

[144] **PFloTran: A Massively Parallel Reactive Flow and Transport Model for describing Surface and Subsurface Processes**. `http://www.pflotran.org`.

[145] F. Freitag, J. Corbalan, and J. Labarta. **A Dynamic Periodicity Detector: Application to Speedup Computation**. In *In Proceedings of International Parallel and Distributed Processing Symposium (IPDPS*, 2001.

[146] K. Mohror and K. L. Karavanic. **Evaluating similarity-based trace reduction techniques for scalable performance analysis**. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 55:1–55:12, New York, NY, USA, 2009. ACM.

[147] A. Knüpfer, B. Voigt, W. E. Nagel, and H. Mix. **Visualization of repetitive patterns in event traces**. In *Proceedings of the 8th international conference on Applied parallel computing: state of the art in scientific computing*, PARA'06, pages 430–439, Berlin, Heidelberg, 2007. Springer-Verlag.

[148] C. W. Lee, C. Mendes, and L. Kale. **Towards Scalable Performance Analysis and Visualization through Data Reduction**. In *IPDPS'08: Proceedings of the 22nd IEEE International Parallel and Distributed Processing Symposium*, pages 1–8, April 2008.

[149] F. P. Preparata and S. J. Hong. **Convex Hulls of Finite Sets of Points in Two and Three Dimensions**. *Communications ACM*, **20**:87–93, February 1977.

[150] M. Auguin and F. Larbey. **OPSILA: an advanced SIMD for numerical analysis and signal processing**. In *Euromicro-83 Symp.*, pages 311–318, Sept. 14–16 1983.

[151] **CGAL, Computational Geometry Algorithms Library**. http://www.cgal.org.

[152] H. Blum. **A transformation for extracting new descriptors of shape**. In W. Wathen-Dunn, editor, *Proc. Models for the Perception of Speech and Visual Form*, pages 362–380, Cambridge, MA, November 1967. MIT Press.

[153] G. Voronoi. **Nouvelles applications des paramètres continus à la théorie des formes quadratiques. Deuxième mémoire. Recherches sur les parallélloèdres primitifs.** *Journal für die reine und angewandte Mathematik*, **134**:198–287, 1908.

[154] O. Aichholzer, F. Aurenhammer, D. Alberts, and B. Gärtner. **A Novel Type of Skeleton for Polygons**. In H. Maurer, C. Calude, and A. Salomaa, editors, *J.UCS The Journal of Universal Computer Science*, pages 752–761. Springer Berlin Heidelberg, 1996.

[155] **Blue Waters supercomputer**. `https://bluewaters.ncsa.illinois.edu`.

[156] **NEKBONE: Thermal Hydraulics mini-application**. `https://cesar.mcs.anl.gov`.

[157] M. Levin. **B. Mirkin, Mathematical Classification and Clustering**. *J. of Global Optimization*, **12**(1):105–108, January 1998.

[158] M. Meilă. **Comparing Clusterings: An Axiomatic View**. In *Proceedings of the 22Nd International Conference on Machine Learning*, ICML '05, pages 577–584, New York, NY, USA, 2005. ACM.

[159] J. Gonzalez, J. Gimenez, and J. Labarta. **Automatic Evaluation of the Computation Structure of Parallel Applications**. In *Proceedings of the 2009 International Conference on Parallel and Distributed Computing, Applications and Technologies*, PDCAT '09, pages 138–145, Washington, DC, USA, 2009. IEEE Computer Society.

[160] A. Wissink, R. Hornung, S. Kohn, S. Smith, and N. Elliott. **Large Scale Parallel Structured AMR Calculations Using the SAMRAI Framework**. In *Supercomputing, ACM/IEEE 2001 Conference*, pages 22–22, Nov 2001.

[161] J. Gonzalez, J. Gimenez, and J. Labarta. **Performance Data Extrapolation in Parallel Codes**. In *Proceedings of the 2010 IEEE 16th International Conference on Parallel and Distributed Systems*, ICPADS '10, pages 155–163, Washington, DC, USA, 2010. IEEE Computer Society.

[162] S. Zhou, A. Zhou, J. Cao, J. Wen, Y. Fan, and Y. Hu. **Combining Sampling Technique with DBSCAN Algorithm for Clustering Large Spatial Databases**. In *Proceedings of the 4th Pacific-Asia Conference on Knowledge Discovery and Data Mining, Current Issues and New Applications*, PADKK '00, pages 169–172, London, UK, UK, 2000. Springer-Verlag.

[163] H. Servat, G. Llort, J. Gimenez, K. Huck, and J. Labarta. **Unveiling Internal Evolution of Parallel Application Computation Phases**. In *Proceedings of the 2011 International Conference on Parallel Processing*, ICPP '11, pages 155–164, Washington, DC, USA, 2011. IEEE Computer Society.

[164] **The Weather Research & Forecasting model**. `http://www.wrf-model.org`.

[165] **GNU Binutils**. `http://www.gnu.org/software/binutils`.

[166] M. A. Patwary, D. Palsetia, A. Agrawal, W.-k. Liao, F. Manne, and A. Choudhary. **A New Scalable Parallel DBSCAN Algorithm Using the Disjoint-Set Data Structure**. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 62:1–62:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[167] O. H. Ibarra and C. E. Kim. **Fast Approximation Algorithms for the Knapsack and Sum of Subset Problems**. *J. ACM*, **22**(4):463–468, October 1975.

[168] **Open source package for Material eXplorer**. `http://www.openmx-square.org`.

[169] P. Mimica, D. Giannios, and M. A. Aloy. **Deceleration of arbitrarily magnetized GRB ejecta: the complete evolution**. Technical Report arXiv:0810.2961, Oct 2008. Comments: 13 pages, 10 figures, revised version sent to the referee (first version submitted on 6th of August).

[170] P.-F. Lavalléea, G. C. de Verdièreb, P. Wauteleta, D. Lecasa, and J.-M. Dupays. **Porting and optimizing HYDRO to new platforms and programming paradigms – lessons learnt**. Available online at `www.prace-ri.eu`.

[171] **The CGPOP Miniapp**. `http://www.cs.colostate.edu/hpc/cgpop`.

[172] P. Jones. **Parallel Ocean Program (POP) user guide**. Technical report, Los Alamos National Laboratory, March 2003.

[173] K. Palaniappan, I. Ersoy, and S. K. Nath. **Moving Object Segmentation Using the Flux Tensor for Biological Video Microscopy**. In *Proceedings of the multimedia 8th Pacific Rim conference on Advances in multimedia information processing*, PCM'07, pages

483–493, Berlin, Heidelberg, 2007. Springer-Verlag.

[174] A. Yilmaz, O. Javed, and M. Shah. **Object Tracking: A Survey**. *ACM Comput. Surv.*, **38**(4), December 2006.

[175] P. C. Roth. *ETRUSCA: EVENT TRACE REDUCTION USING STATISTICAL DATA CLUSTERING ANALYSIS*. Master's thesis, University of Iowa, 1992.

[176] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. **Automatically Characterizing Large Scale Program Behavior**. *SIGARCH Comput. Archit. News*, **30**(5):45–57, October 2002.

[177] T. Hoefler. **Bridging Performance Analysis Tools and Analytic Performance Modeling for HPC**. In *Proceedings of the 2010 Conference on Parallel Processing*, Euro-Par 2010, pages 483–491, Berlin, Heidelberg, 2011. Springer-Verlag.

[178] F. Chirigati, V. Silva, E. Ogasawara, D. de Oliveira, J. Dias, F. Porto, P. Valduriez, and M. Mattoso. **Evaluating Parameter Sweep Workflows in High Performance Computing**. In *Proceedings of the 1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies*, SWEET '12, pages 2:1–2:10, New York, NY, USA, 2012. ACM.

[179] S. Girona, J. Labarta, and R. Badia. **Validation of Dimemas Communication Model for MPI Collective Operations**. In J. Dongarra, P. Kacsuk, and N. Podhorszki, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, **1908** of *Lecture Notes in Computer Science*, pages 39–46. Springer Berlin Heidelberg, 2000.

[180] **The GREMLINS Framework: Emulating Exascale Conditions on Today's Platforms**. `https://scalability.llnl.gov/gremlins`.

[181] **Barcelona Supercomputing Center: HPC Facilities**. `www.bsc.es/marenostrum-support-services`.

[182] M. Schulz and B. de Supinski. **PNMPI tools: a whole lot greater than the sum of their parts**. In *Supercomputing, 2007. SC '07. Proceedings of the 2007 ACM/IEEE Conference on*, pages 1–10, Nov 2007.

[183] I. Karlin, A. Bhatele, B. L. Chamberlain, J. Cohen, Z. Devito, M. Gokhale, R. Haque, R. Hornung, J. Keasler, D. Laney, E. Luke, S. Lloyd, J. McGraw, R. Neely, D. Richards, M. Schulz, C. H. Still, F. Wang, and D. Wong. **LULESH Programming Model and Performance Ports Overview**. Technical Report LLNL-TR-608824, December 2012.

[184] **Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH)**. `https://codesign.llnl.gov/lulesh.php`.

[185] M. Casas and G. Bronevetsky. **Active Measurement of Memory Resource Consumption**. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, IPDPS '14, pages 995–1004, Washington, DC, USA, 2014. IEEE Computer Society.

[186] **Nanos++**. `http://pm.bsc.es/nanox`.

[187] **Dynamic Load Balancing**. `http://pm.bsc.es/dlb`.

[188] K. L. Karavanic and B. P. Miller. **A Framework for Multi-Execution Performance Tuning**. In T. Ludwig and B. P. Miller, editors, *On-line Monitoring Systems and Computer Tool Interoperability*, pages 61–89. Nova Science Publishers, Inc., Commack, NY, USA, 2003.

[189] R. Prodan, T. Fahringer, M. Geissler, G. Madsen, F. Franchetti, and H. Moritsch. **On Using ZENTURIO for Performance and Parameter Studies on Cluster and Grid Architectures.** In *PDP*, pages 185–192. IEEE Computer Society, 2003.

[190] V. Taylor, X. Wu, and R. Stevens. **Prophesy: An Infrastructure for Performance Analysis and Modeling of Parallel and Grid Applications**. *SIGMETRICS Perform. Eval. Rev.*, **30**(4):13–18, March 2003.

[191] C. Xu, X. Chen, R. Dick, and Z. Mao. **Cache contention and application performance prediction for multi-core systems**. In *Performance Analysis of Systems Software (ISPASS), 2010 IEEE International Symposium on*, pages 76–86, March 2010.

[192] X. Chen, C. Xu, R. P. Dick, and Z. M. Mao. **Performance and Power Modeling in a Multi-programmed Multi-core Environment**. In *Proceedings of the 47th Design Automation Conference*, DAC '10, pages 813–818, New York, NY, USA, 2010. ACM.

[193] D. Eklov, N. Nikoleris, D. Black-Schaffer, and E. Hagersten. **Cache Pirating: Measuring the Curse of the Shared Cache**. In *Proceedings of the 2011 International Conference on Parallel Processing*, ICPP '11, pages 165–175, Washington, DC, USA, 2011. IEEE Computer Society.

[194] D. Eklov, N. Nikoleris, D. Black-Schaffer, and E. Hagersten. **Bandwidth Bandit: Quantitative characterization of memory contention**. In *Code Generation and Optimization (CGO), 2013 IEEE/ACM International Symposium on*, pages 1–10, Feb 2013.

[195] **MPI/CHameleon: High-Performance Portable MPI implementation**. `http://www.mpich.org`.

[196] X. Zhang. **Performance measurement and modeling to evaluate various effects on a shared memory multiprocessor**. *Software Engineering, IEEE Transactions on*, **17**(1):87–93, Jan 1991.

[197] M. Pavlovic, Y. Etsion, and A. Ramirez. **On the memory system requirements of future scientific applications: Four case-studies**. In *Workload Characterization (IISWC), 2011 IEEE International Symposium on*, pages 159–170, Nov 2011.

[198] L. Ma and R. Chamberlain. **A Performance Model for Memory Bandwidth Constrained Applications on Graphics Engines**. In *Application-Specific Systems, Architectures and Processors (ASAP), 2012 IEEE 23rd International Conference on*, pages 24–31, July 2012.

[199] **Jaguar supercomputer**. `https://www.olcf.ornl.gov`.

[200] F. Wolf, F. Freitag, B. Mohr, S. Moore, and B. J. N. Wylie. **Large Event Traces in Parallel Performance Analysis**. In *Proc. 8th Workshop on Parallel Systems and Algorithms (PASA), Frankfurt, Germany,* **P-81** of *Lecture Notes in Informatics*, pages 264–273. Gesellschaft für Informatik, March 2006.

[201] C. Isci and M. Martonosi. **Runtime Power Monitoring in High-End Processors: Methodology and Empirical Data**. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, pages 93–, Washington, DC, USA, 2003. IEEE Computer Society.

[202] B. S. Center. **Extrae User Guide manual**. Available online at http://www.bsc.es/computer-sciences/performance-tools/documentation.

[203] B. S. Center. **ClusteringSuite Introductory Manual**. Available online at http://www.bsc.es/computer-sciences/performance-tools/documentation.

[204] U. of Wisconsin. **MRNet API Programmer's Guide**. http://www.paradyn.org/mrnet.