On the usefulness of object tracking techniques in performance analysis

Germán Llort Harald Servat Judit Giménez

Jesús Labarta

Barcelona Supercomputing Center

Universitat Politècnica de Catalunya

{gllort, harald, judit, jesus}@bsc.es

Abstract

Understanding the behavior of a parallel application is crucial if we are to tune it to achieve its maximum performance. Yet the behavior the application exhibits may change over time and depend on the actual execution scenario: particular inputs and configuration variables, the number of processes running, or hardware-specific problems. Then, beyond the details of a single experiment a far more interesting question arises. How does the application behavior respond to changes in the execution conditions?

In this paper, we demonstrate that object tracking concepts have huge potential to be applied in the context of performance analysis. We leverage tracking techniques to analyze how the behavior of a parallel application evolves through multiple scenarios where the execution conditions change. This method provides comprehensible insights on the influence of different parameters on the application behavior, enabling to identify the most relevant code regions and their performance trends, variabilities and bottlenecks.

1. Introduction

The execution of a scientific code is dependent on a variety of parameters that may have a strong impact on its performance: the size of the problem, the number of processes running in parallel, their physical mapping onto nodes, the choice of parallel programming model, and many other hardware and software adjustments that are significant for the particular application. The combination of these factors constitute an unique execution scenario, which directly influences the application behavior and results.

Often, it is very difficult to anticipate the impact of different configurations on the final performance, work balancing or memory usage. Analyzing these effects is important not only to get better understanding of the program behavior, but also to quantify the improvements or degradations so as to foresee trends in the application performance. To this end, it is necessary to provide the users with tools to compare different scenarios and correlate observations between them.

The main contribution of this paper is a technique for easily performing very diverse parametric and evolutionary studies, correlating performance information either from multiple runs with different configurations, or different time intervals within the same experiment. Our approach focuses on 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. To this end, we present a tool where the object tracking and performance analysis worlds converge.

Traditionally, tracking techniques are used to locate a moving object in an image or video sequence. Practical examples include human-computer interaction, security and surveillance or traffic control. A first step to these problems is to delimit the objects of interest within the scene. Therefore, object recognition algorithms (e.g. image segmentation and edge detection) will look for appearance characteristics (such as color, intensity, or texture) and distinguishing features that identify them. Then, consecutive frames are compared to find a correspondence between objects and their possible displacements.

Analogously, we will represent multiple execution scenarios as a sequence of images expressing the evolution of the application metrics. Code regions will be 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 will highlight changes in the application performance, which can be modeled into metrics that evaluate the performance trends of the different regions of code.

Difficulties in tracking mainly arise due to abrupt object motion, and tracking code regions is not exempt from this risk. 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 object's position, direction or shape in the performance space. To tackle this problem, we present an algorithm that automatically tracks the evolution of code regions along multiple scenarios despite their possible performance variations.

While previous analysis techniques for comparing experiments or phases [10, 18, 20] have been presented, our work goes one step further and presents a novel technique that does not rely on pre-selected 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 different behaviors, thus making averages will hide divergent performance trends. Furthermore, access to the sources may be restricted, or the user may have no prior knowledge of the application to decide which sections are meaningful to instrument. Our position is that it is necessary not to consider averages, but every independent instance to detect structure and capture multi-modal variability. To this end, we characterize every different type of performance behavior abstracting from the code regions that present them. In this scenario, the use of tracking is a natural and intuitive tool that allows us to measure the evolution of these behavioral trends automatically.

We prove this technique useful to discover and understand valuable performance insights in very diverse cases of analysis, of which we report as examples, the study of the impact of: different architectures and software, increasing problem sizes, memory bandwidth and cache contention, multi-core sharing and scalability.

The rest of the paper is structured as follows. Section 2 describes the process we follow to build a sequence of images from different experiments, each representing how the different code regions perform with respect to several metrics. Section 3 describes the algorithm that tracks the evolution of

the different code regions along the images. Section 4 shows how this technique can be applied to obtain interesting performance observations for very diverse studies. Section 5 gives a brief overview of similar works in the field. Finally, we compile the conclusions from our research in Section 6.

2. From performance data to trackable objects

Analysis tools usually choose to display performance data to the user in the form of profiles at the level of program subroutines (loops, or user-defined sections). This has the advantage of being a very natural and understandable representation, but also carries a few drawbacks along. Prior knowledge of the application is 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. And when no dynamic instrumentation mechanisms are available, 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 [17, 16].

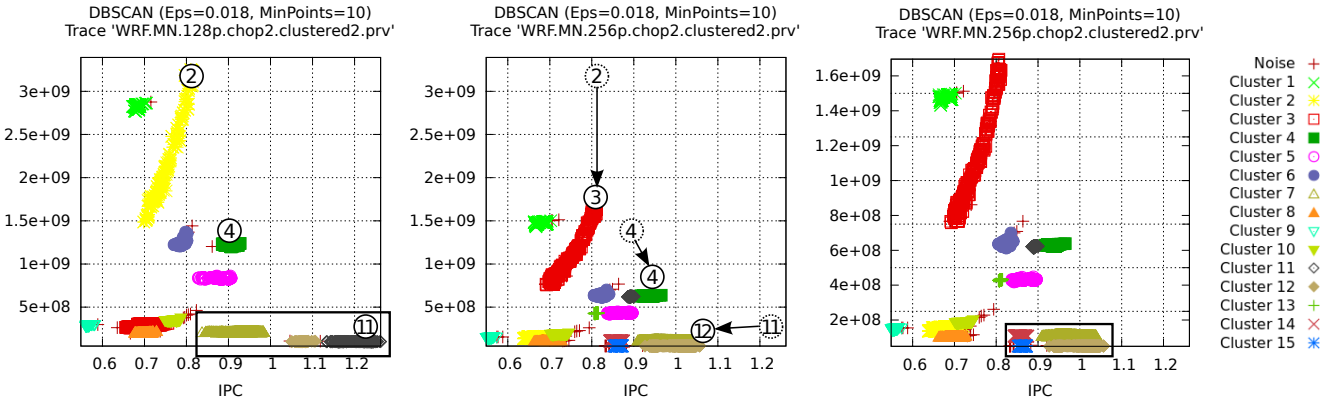
A different granularity to characterize the application performance are the computing regions (i.e. CPU bursts). A CPU burst is defined as the sequential computation between calls to the MPI or OpenMP runtimes. Delimiting these regions only requires library interposition to instrument the parallel programming 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 code region, and a vector of hardware counter metrics which describe how it performed. Considering every CPU burst rather than simple averages, we can see whether variability distributes across processes or 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 [7], the authors demonstrate that this granularity is useful for the analysis of parallel

applications, as it reflects an intermediate point of view between the very low level characterizations (i.e. basic blocks or instruction-level simulators) and higher abstractions (i.e. functions, loops or user-defined code blocks). Regardless of our current implementation, the technique presented would as well be applicable at different granularities.

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 the experiments 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. While the experiments described hereafter define these two dimensions, the whole process can be likewise applied to any arbitrary number of dimensions.

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 and their frequent SPMD organization, 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 [7, 9], that uses density-based clustering in order to group similar CPU bursts with respect to the metrics selected.

The result of the analysis 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 code regions, and both terms will be indistinctly used for clarity, but this connection is not necessarily univocal: 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) 128 tasks (b) 256 tasks (c) 256 tasks, scales normalized

Figure 1: Structure of WRF computing bursts

a behavioral trend, independently of the code region that exhibits it. Being the objective to improve the application performance, characterizing the dynamic behavior of the regions rather than static code structures guarantees that we direct the analysis towards the zones of real interest.

Figure 1a shows the structure of the twelve most time-consuming regions of the WRF application [6] 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 (i.e. Region 2) denote instructions imbalance, while those that stretch horizontally (i.e. 7 and 11) reflect IPC variations. Computations with high amount of work but low performance are an interesting subject of study, as well as those with the same amount of work at different speeds, or vice-versa, as these indicate potential load-imbalances.

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 1b shows the structure of WRF, doubling the number of cores in the execution. The number of instructions executed per core has reduced in inverse proportion, and so all clusters have moved down along the Y-axis. Intuitively, we can see that cluster 2 (yellow) turned into 3 (red). And a few clusters have slightly improved their performance (i.e. 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 the number of clusters increasing from 3 to 4. 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 is that 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 parameters 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 would be possible to determine who-is-who by visual inspection, the benefits of an automated mechanism able to detect abrupt changes among 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, if the number of processors doubles, the number of instructions executed per core will typically decrease in proportion. A step prior to track the evolution of the objects consists in transforming 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 1c shows the 256-tasks case with the performance scales normalized. The relative distances compared to the base 128-tasks case are actually 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 computing phases of the code, and to understand how the different configurations get to influence their behavior.

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 space. Let A and B be two images, as depicted in Figure 2, 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 $Q = \{Q_1, ..., Q_k\}$ of B, that fulfill the condition:

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

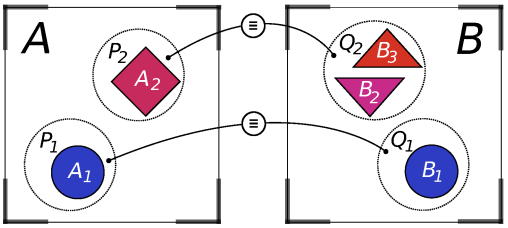


Figure 2: Tracking scheme

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 our case, this operation is implemented using four heuristics that evaluate different characteristics of the computing regions:

- *Displacements in the performance space.* 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.

- *SPMD simultaneity.* In SPMD applications, all processes will be executing the same phase simultaneously. So if two different clusters happen simultaneously in different processes, it is likely that they refer to the same code.

- *Call stack references.* Call stack information links every computation to the point in the code where it starts and finishes executing. There will be an explicit relation between any pair of clusters whose computations belong to the same code region.

- *Execution sequence.* The temporal order in which all computations get executed can be expressed as an ordered sequence that can be analyzed to see which code phases happen one after another.

Every evaluator is run separately and produces one or more correlation matrices representing correspondences between objects. Depending on the evaluator, what these matrices express is different. Figure 3 shows the correlations computed by the first evaluator 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 evaluator 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. In the last case, the matrix reflects the percentage of occurrence where computations A_i and B_j happen in the same chronological order. In all cases, non-zero cells evince that a given pair of objects are the same according to that evaluator, with a certain probability. Occurrences with a very small probability (5% by default) are neglected as outliers.

Since every evaluator considers different properties of the objects, they have to cooperate to complement the correspondences that a given one might fail to discern. The combination algorithm starts from the set of relations found by the *displacements* evaluator, and enhances its results with the findings from the *SPMD* evaluator. 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 get merged into a more general relation $A_5 \equiv B_5 \cup B_{13}$. The *call stack* evaluator is then used to prune incorrect relations that may appear due to imprecisions in the former heuristics. All related regions must share the same references to the source

code, so we discard those not having any in common.

We search for correspondeces reciprocally, this is to say, comparing frame A with B and vice versa, extracting a final set of rules that correlate the objects between both frames. When the information available leads the evaluators 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 last heuristic is finally used to refine the results, trying to split 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 that describe the evolution of each *tracked region*. The following Sections 3.1 to 3.4 describe the above evaluators in more detail, and Section 3.5 explains the results of the tracking algorithm.

3.1. Displacements in the performance space

This evaluator 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 objects displacements will generally be short. Consider again the previous example where we doubled from 128 to 256 the number of cores in WRF (see Figures 1a and 1c). The resulting structure for both experiments hardly differs, with very slight movements.

However, there are situations where a given region 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 3, where region 4 shifts to two behaviors, namely 4 and 11. Also, there are cases where clusters can move a long way in the space, as can be the case of regions 11 and 12 in Figure 1a to regions 12 and 15 in Figure 1c, 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), but we can then use the remaining evaluators to discern whether those regions are the same or not.

	B_1	B_2	B_3	B_4	B_5	B_6	...	B_{10}	B_{11}	B_{12}
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
⋮										
A_{11}	0	0	0	0	0	0		0	0	100%
A_{12}	0	0	0	0	0	0		0	0	100%

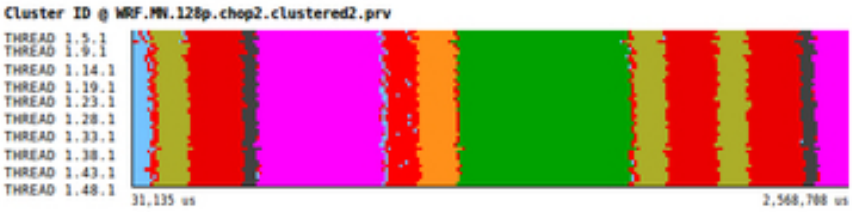
Figure 3: Correlations from displacements evaluator between WRF-128 (rows) and WRF-256 (columns)

3.2. SPMD simultaneity

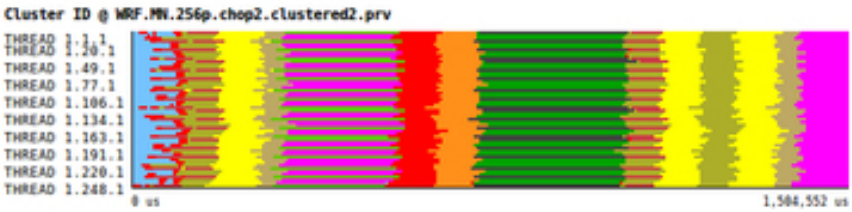
This evaluator exploits the SPMD structure of the applications to match computing regions that happen simultaneously in different processes. Assuming this execution model, all processors are expected to be executing the same phase of code at a time. In this case, if multiple processes are executing different 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 4a shows a detailed view of the temporal sequence of clusters at the beginning of one iteration of WRF 128-tasks. The same pattern can also be observed in the 256-tasks case in 4b, meaning that the code phases and the order in which they get executed are the same in both experiments. In the first case, all processes (Y-axis) execute the same computations over time (X-axis), but some variability appears in the latter, where some processes execute different computations simultaneously. These are the same regions of code, although they present slight performance variations.

Whether the application is SPMD is evaluated with the technique presented in [8]. 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 add an equivalence between them.



(a) SPMD computations in the 128-tasks experiment



(b) SPMD computations in the 256-tasks experiment

Figure 4: Correlations from SPMD evaluator for WRF

3.3. Call stack references

This evaluator 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 computation starts, linking them to specific points of code. If two regions from two different frames do not share any code reference, they can not be considered equivalent.

Table 1: Correlations from call stack evaluator for WRF

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 1 illustrates a subset of the relations that can be outlined between regions from their code references. The reason why some relations are not univocal is because the clustering process groups

computations based on their similarity with respect to selected metrics, so it is possible that different points of code behave the same and get grouped under the same cluster. Also, if a single 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.

3.4. Execution sequence

This evaluator 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 [8] is applied now on two experiments, and we then compare their respective execution sequences. Keeping in mind that equivalent objects might have different identifiers between experiments, the sequences can not be compared directly. Instead, we use the matchings discovered so far by the previous evaluators to establish *pivots* in both sequences and align them with respect to these points of reference, as exemplified in Figure 5. For example, if the former evaluators conclude that region 1 in the first experiment becomes region 2 in the second, we can infer from the sequences that regions 2 and 3 will correspond to 3 and 4, respectively.

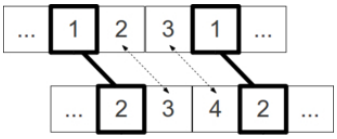


Figure 5: Correlations from execution sequence evaluator

3.5. Tracking results

The tool reconstructs the input images for the tracking algorithm with all objects identifiers renamed, so that all the equivalent regions keep the same numbering and color along the whole sequence of images. Figure 6 shows the resulting images for the executions of WRF with 128 (left) and 256 (right) tasks. These scatter plots can be displayed in a simple animation, so that it is very easy to identify

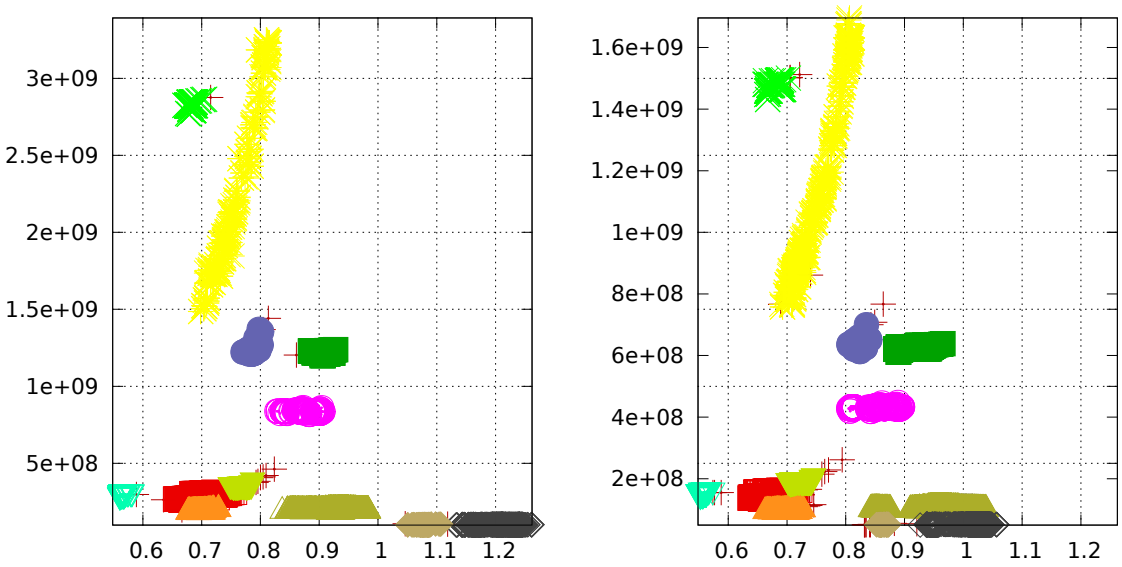


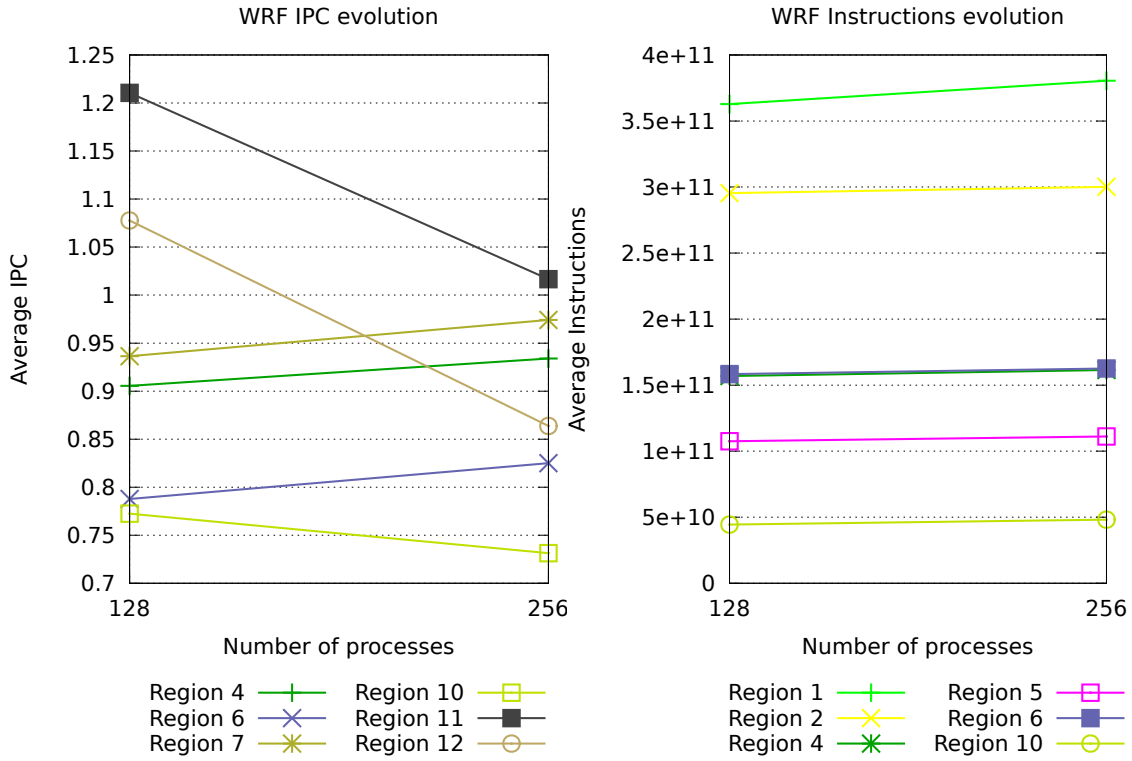
Figure 6: Sequence of images for WRF 128-tasks (left) and 256-tasks (right), tracked regions renamed

variations in the performance space, which enables the study of the repercussion of the changes in the execution conditions on the structure of every particular region of code.

In addition, the tool illustrates the evolution of every computing region from the first scenario to the last, with respect to the performance metrics selected to generate the images. Figure 7a shows a trend line chart displaying the evolution in IPC (Y-axis) for the 128 and 256-tasks runs of WRF (X-axis). For better readability, only the regions with higher IPC variations (above 3%) are depicted. Regions 11 and 12 present a 20% decline, while there is a slight 5% improvement for regions 4, 6 and 7.

Figure 7b shows the evolution in the total number of instructions for the regions that execute the most. When the number of cores increases, the total amount of work gets evenly distributed, and thus the number of instructions executed per process remains constant. The increasing trend for Region 1 denotes a 5% of code replication.

The information from these plots can be used to perform parametric studies on the influence of software or hardware changes in the achieved performance along multiple experiments, as well as to study the evolution of the application over time within a single experiment, enabling to extract recommendations on which way to direct the optimization process. Having call stack references associated to every cluster, it is possible to connect the observed performance artifacts to specific points in the code. Also,



(a) IPC evolution (b) Instructions evolution

Figure 7: Performance trends for WRF code regions

these results can be used as a model to predict the outcome of future experiments.

Table 2: Summary of experiments

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

4. Experimentation

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. Also, we want to stress that the proposed technique is a powerful and versatile tool that is applicable for a variety of studies, which include but are not limited to the impact of: using different compilers and hardware architectures, scaling the size of the problem, memory and caches contention and sharing resources. Optimizing the applications is beyond the scope of this paper.

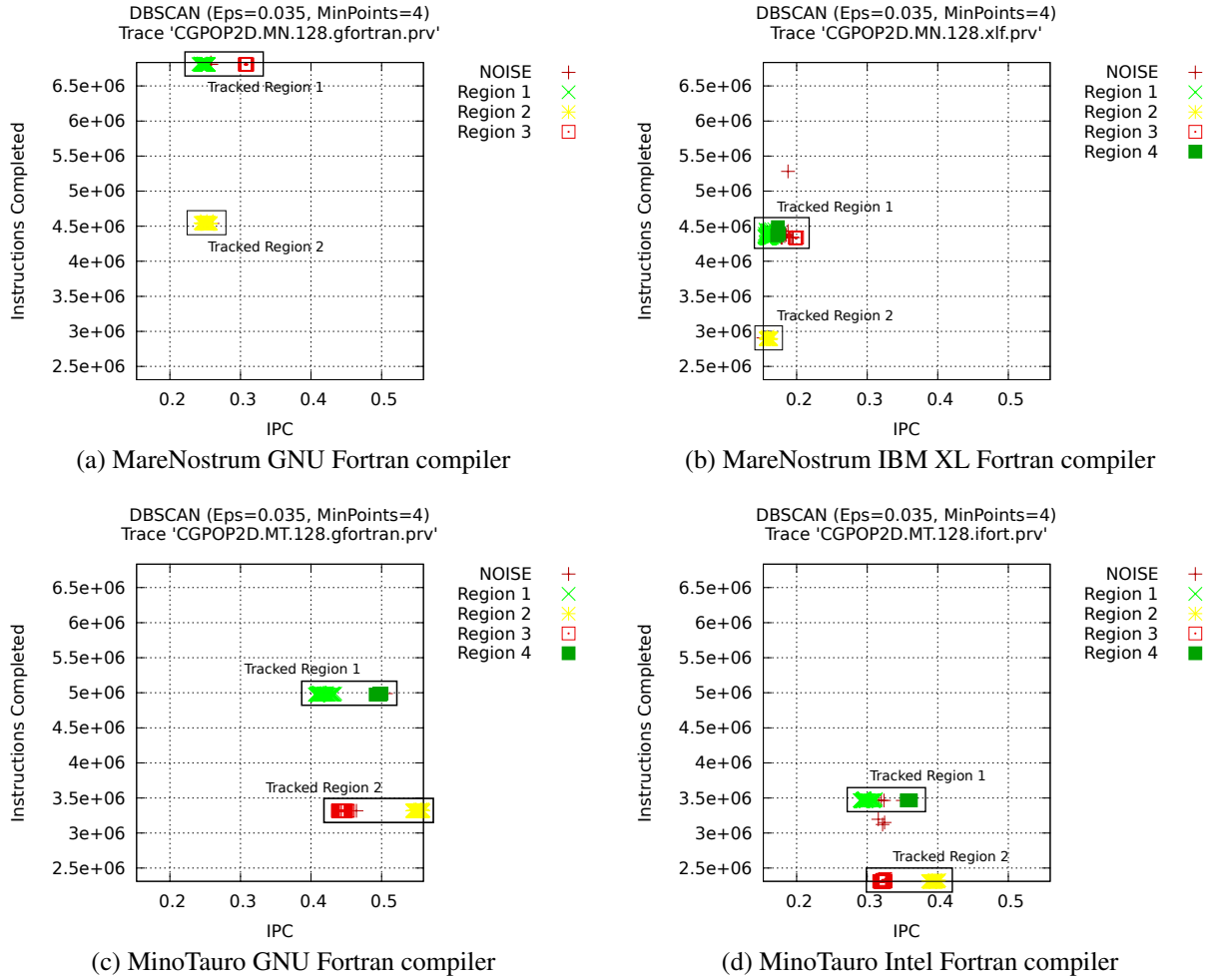
To this end, a variety of real applications and benchmarks were run in MareNostrum [1] and MinoTauro [2] supercomputers. MareNostrum is a cluster of 2560 JS21 compute nodes each containing 2 dual core IBM PowerPC 970MP processors running at 2.3 GHz with 8GB of RAM memory. MinoTauro comprises 126 compute nodes each containing 2 Intel Xeon E5649 6-Core processors running at 2.53 GHz with 24 GB of RAM memory. The applications were linked using the default MPI implementation installed in each system, and the ABI was set to 64 bits.

Table 2 illustrates the ability of the algorithm to identify and keep track of the different computing regions in 10 different case-studies. The objects detected are automatically reduced to the ones considered more relevant, those that represent a high percentage of the application time, usually 10-30% of the total. *Coverage* is calculated as the percentage of objects tracked by the algorithm 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 univocal 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 four case studies in more detail.

4.1. Studying the platform and compiler impact

CGPOP [5] is a proxy application of the Parallel Ocean Program [11]. 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 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. In this experiment we are going to stress the performance variations in the application due to the different architectures and also study the impact of using a generic versus an architecture specific compiler.

The input to the tracking algorithm is the collection of images that depict the performance of each individual experiment, shown in Figure 8. In all four, there are two main trends with respect to the num-



(a) MareNostrum GNU Fortran compiler

(b) MareNostrum IBM XL Fortran compiler

(c) MinoTauro GNU Fortran compiler

(d) MinoTauro Intel Fortran compiler

Figure 8: Sequence of input images to the tracking algorithm for CGPOP

ber of instructions, divided into several regions due to differences in the achieved IPC. In MareNostrum, when the application is compiled with xlf (see 8b) all computations see the number of instructions significantly reduced (36% and 33%, respectively) compared to using gfortran (see 8a), 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 8c and 8d), 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 3 summarizes the averages for IPC and instructions for both tracked regions, and their elapsed execution time.

The specialized compilers xlf and ifort attain a reduction of 36% and 30% of the number of instructions executed with respect to gfortran in both machines, but at the expense of an average IPC loss of 36% in MareNostrum and 28% in MinoTauro, which leads to negligible changes in the execution times of the computing regions, with variations smaller than $\pm 0.03\%$. In this case, the election of the compiler may affect the computational complexity of the execution but does not have meaningful repercussions in the total execution time.

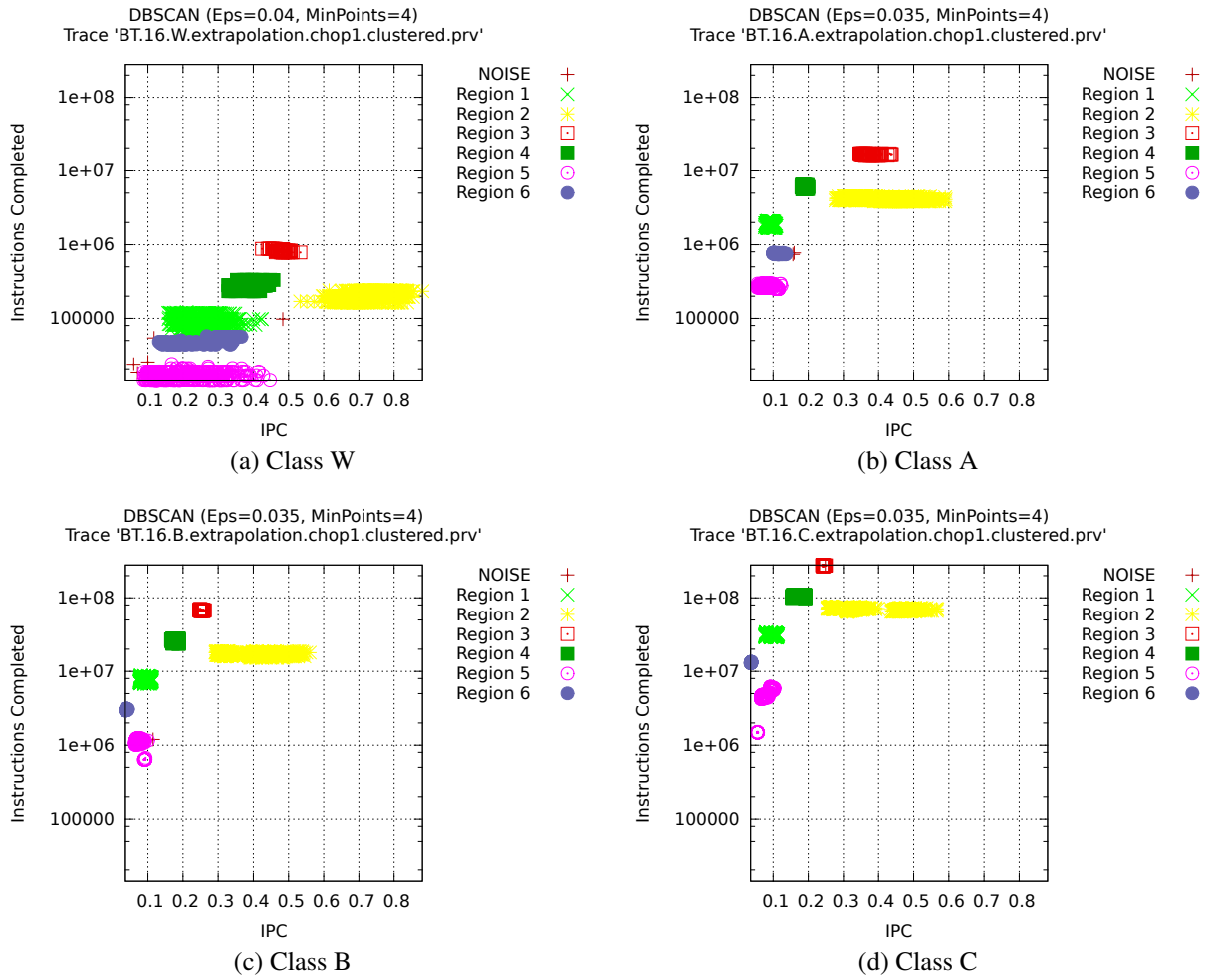
Table 3: CGPOP performance results

		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

4.2. Studying the problem size impact

The NAS Parallel Benchmarks [3] 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 in NAS 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 with 16 processes.

Figure 9 shows the outcome of the algorithm applied to the sequence of experiments with increasing problem size, where all tracked regions have been colored the same. The plots show large dynamic



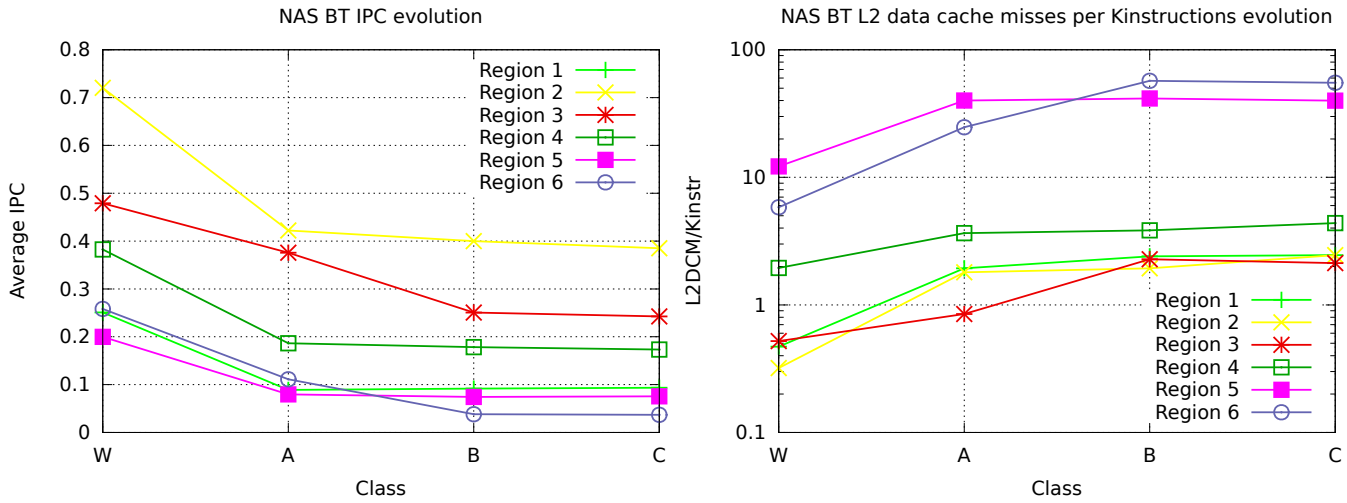
(a) Class W (b) Class A

(c) Class B (d) Class C

Figure 9: Sequence of output images from the tracking algorithm for NAS BT

range in the number of instructions, increasing two orders of magnitude from the bottom of Figure 9a for Class W to the top of Figure 9d for Class C. Class W also presents large variability in IPC, which greatly reduced in the following experiments except for Region 2. Still, the same main six computing regions can be easily identified in all cases.

The achieved performance degrades as the size of the problem increases. Figure 10a 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. Figure 10b shows that this IPC reduction is related to an increase in L2 data cache misses.



(a) Evolution of IPC

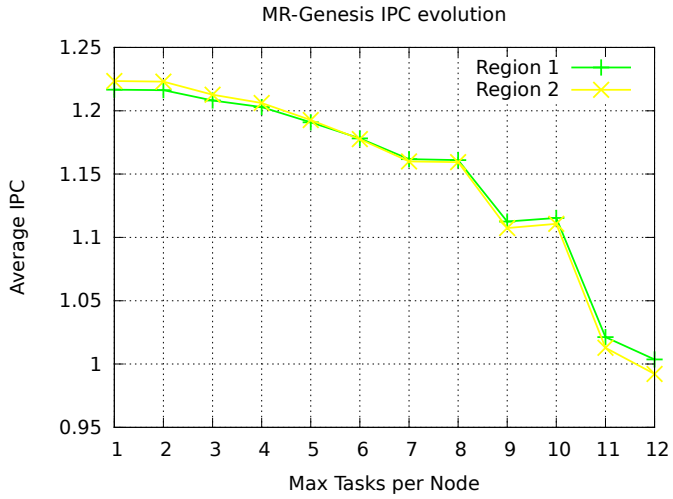
(b) Evolution of L2 Cache Misses

Figure 10: Performance trends for NAS BT code regions

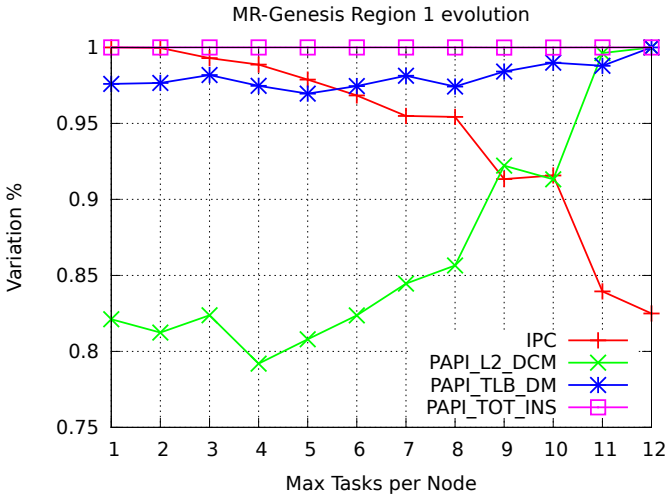
4.3. Studying the multi-core resource sharing impact

MR-Genesis [14] 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 node with all processes running in it for the last experiment, with all the intermediate cases also tested. The purpose of this experiment is to study the effect of memory bandwidth and caches contention on the application performance when sharing resources.

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 with processes, the achieved performance of the application decreases. Figure 11a shows the progression of IPC for the two main computing regions of the application, which present the same behavior. Up to the 66% of the node occupation (8 tasks per node) the IPC presents a slight downslope 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.



(a) Evolution of IPC in MR-Genesis experiments



(b) Evolution of Region 1 in MR-Genesis experiments

Figure 11: Performance trends for MR-Genesis code regions

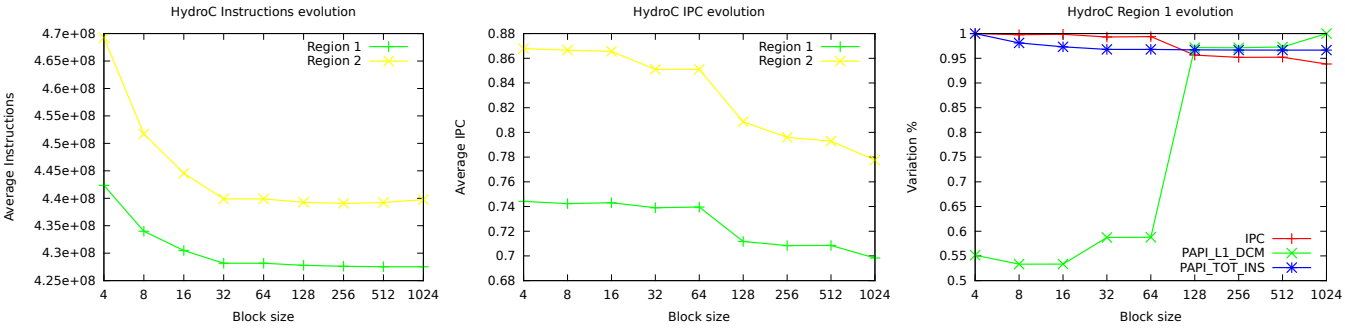
Figure 11b correlates all the performance metrics that describe Region 1. The Y-axis reflects the percentage of variation of each metric with respect to its maximum value for all trials. It can be seen that the number of L2 cache misses grows inversely to the IPC degradation rate, and the number of TLB misses also increase as the node gets more populated.

4.4. Studying the block size impact

HYDRO [12] is a proxy benchmark of the RAMSES [4] application, 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 varying the block size, doubling it from 4 to 1024. The objective of the experiment is to study the impact of the block size on the performance.

The application presents a single computing phase with bimodal behavior. Figure 12a shows the number of instructions gradually decreasing for both regions with drops from 1% to 3% up to a block size of 32, and keeps constant beyond this point. IPC also decreases (see Figure 12b), with a total deviation of 5% for Region 1, and 10% for Region 2, both regions presenting a sharp dip when the block size increases from 64 to 128. At this point, the number of L1 data cache misses rockets 40% more, as shown in Figure 12c.

Using small block sizes, the application ends up having more working sets to compute, which entails



(a) Evolution of Instructions (b) Evolution of IPC (c) Evolution of Region 1

Figure 12: Performance trends for HydroC code regions

executing more control instructions. Since the blocks are bidimensional, 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 block sizes, the working set does not fit in the cache, and so the miss rate increases to the detriment of IPC.

5. Related work

Our work draws inspiration from the motion detection algorithm of moving biological objects that are similar but non-homogeneous, presented in [15]. They apply multi-feature contour segmentation and flux tensors for identifying the boundaries of the biological objects and the detection of deformable motion and complex behaviors (e.g. cell crawling or division) along a time-lapse collection of images. This problem shares domain with multimedia video analytics.

In a broader sense, object tracking is applied in the context of applications that require to associate target objects in consecutive frames to detect how they move around the scene. Practical applications include: motion-based recognition, automated surveillance, gesture recognition, traffic monitoring or path planning and obstacle avoidance. [21] 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.

Multi-experimental analysis has been approached by several performance analysis tools. SCALASCA [20] includes a tool called performance algebra that can be used to merge, subtract, and average the data from different experiments and view the results in the form of a single derived experiment. PerfExplorer

[10] 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 [18] 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 [13] 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 events 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 instance of computation rather than profiled averages that may hinder their actual behavior.

Some studies have also employed non-predefined program constructs to characterize the program behavior. In [19], 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 computing regions to characterize the program behavior can be found in [7].

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.

6. Conclusions

In this paper 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 of the objects 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 execution conditions, 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 different computing regions and illustrate their performance trends.

Our technique enables the analyst to identify the most interesting computing regions and the nature of their inefficiencies precisely, without prior knowledge of the application and automatically, enabling the identification of the most appropiate solution for the performance artifacts observed. To this end, we have studied the effect of changes in software such as different compilers and program versions, different testing platforms, sharing multi-core resources, problem and working set sizes, and the program scalability.

All in all, this work presents a versatile tool that can be applied in very different scenarios, enabling the analyst to study how the different execution parameters have an impact on the application performance; analyze the evolution of each code region independently; and ultimately helps to gain better understanding of the application behavior, much beyond what can be learned from a single experiment.

As future work, we consider interesting to extend this mechanism to build predictive models able to foresee the performance of experiments beyond the sample space.

References

[1] MareNostrum system architecture. http://www.bsc.es/support/MareNostrum-ug.pdf. 16

[2] MinoTauro system architecture. http://www.bsc.es/support/MinoTauro-ug.pdf. 16

[3] NAS Parallel Benchmarks. http://www.nas.nasa.gov/Software/NPB. 18

[4] RAMSES. http://web.me.com/romain.teyssier/Site/RAMSES.html. 21

[5] The CGPOP Miniapp website. http://www.cs.colostate.edu/hpc/cgpop. 16

[6] The Weather Research & Forecasting model. http://www.wrf-model.org. 6

[7] 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. 4, 5, 23

[8] 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. 11, 13

[9] 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. 5

[10] 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 of Supercomputing*, page 41, Washington, DC, USA, 2005. IEEE Computer Society. 3, 23

[11] P. Jones. Parallel Ocean Program (POP) user guide. Technical report, Los Alamos National Laboratory, March 2003. 16

[12] P.-F. Lavallea, G. C. de Verdireb, P. Wauteleta, D. Lecasa, and J.-M. Dupays. HYDRO. Available online at http://www.prace-ri.eu. 21

[13] J. Mellor-Crummey. HPCToolkit: Multi-platform tools for profile-based performance analysis. In *5th International Workshop on Automatic Performance Analysis (APART)*, November 2003. 23

[14] 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). 20

[15] 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. 22

[16] 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. 4

[17] H. Servat, G. Llort, J. Giménez, and J. Labarta. Detailed performance analysis using coarse grain sampling. In *PROPER 2009*, August 2009. 4

[18] S. S. Shende and A. D. Malony. The Tau Parallel Performance System. *Int. J. High Perform. Comput. Appl.*, 20(2):287–311, 2006. 3, 23

[19] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. *SIGARCH Comput. Archit. News*, 30(5):45–57, Oct. 2002. 23

[20] 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. 3, 22

[21] A. Yilmaz, O. Javed, and M. Shah. Object tracking: A survey. *ACM Comput. Surv.*, 38(4), Dec. 2006. 22