# Studying Performance Changes with Tracking Analysis

G. Llort, H. Servat, J. Gonzalez, J. Gimenez, and J. Labarta

**Abstract** Scientific applications can have so many parameters, possible usage scenarios and target architectures, that a single experiment is often not enough for an effective analysis that gets sound understanding of their 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 experiments to see how the behavior of the program changes with respect to the varying settings and over time. This methodology 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.

## 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 include the size of the input problem, the number of processes running in parallel, the physical mapping and sharing of the resources, the parallel programming model used, and many other

Barcelona Supercomputing Center — Polytechnic University of Catalonia — BarcelonaTech
Jordi Girona 31, 08034, Barcelona, Spain
e-mail: {gllort | harald | jgonzale | judit | jesus}@bsc.es

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 very 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, object tracking techniques become a natural and intuitive way to detect the performance changes sustained by each part of the code automatically, and represent the information in a clear and visual manner.

While previous approaches for comparing experiments or phases [17, 28, 29] 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.

## 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 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 behavior, that can be modeled into metrics to 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.

## *2.1 Application structure characterization*

Analysis tools usually 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 providing 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 [10], 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 [26, 27].

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 described 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 [14], 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 technique presented would as well be applicable using other abstractions.

### 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 video or 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 density-based cluster analysis [14, 16] 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, we have selected as example the BT-MZ benchmark [5], a solver for block tri-diagonal systems that performs computations of uneven size. Table 1 shows the

Table 1: User functions profile for BT-MZ

| | 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 2: Clusters profile for BT-MZ

| | 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% |

average IPC, total 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 computational 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 1 shows the performance image generated for these functions, with each point 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: all 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 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 all clusters, which was masked in a traditional function-based profile. Column *% Time* shows the fraction of the total execution time that these computational behaviors cover, and it is clear that their weight is not negligible, and thus the importance of being aware of these variabilities. This example highlights the importance of focusing on the dynamic

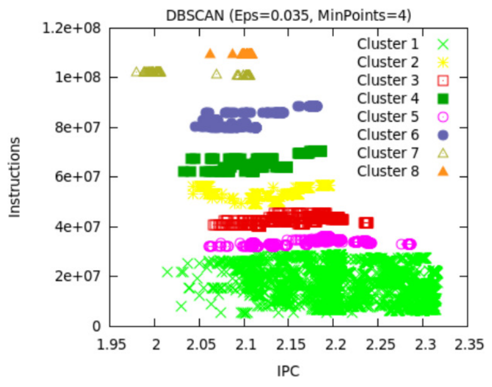


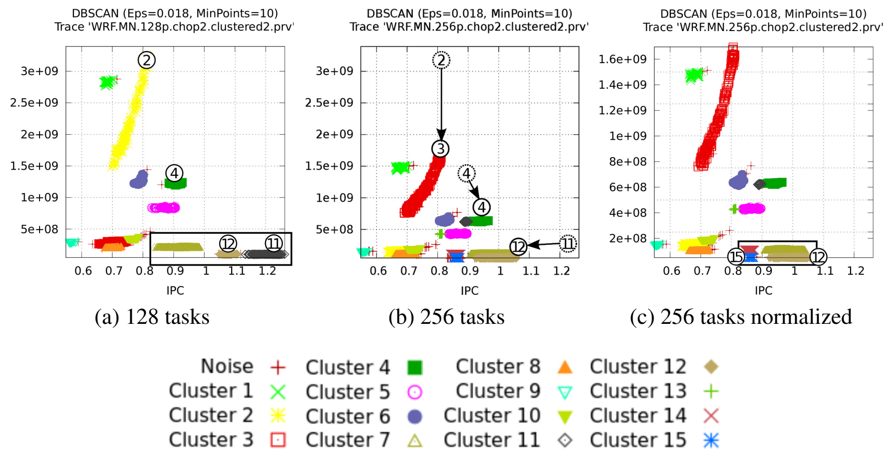
Fig. 1: Clusters for 3 main functions of BT-MZ (Class B, 4 tasks)

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.

### 2.3 Tracking difficulties

The main difficulty in the use of tracking techniques arises due to abrupt object motions and noise in the images. When applied to performance analysis the problem is the same. 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 2a shows the structure of the twelve most time-consuming regions of WRF [8] 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 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



(a) 128 tasks (b) 256 tasks (c) 256 tasks normalized

Fig. 2: Structure of WRF computing bursts

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 is not obvious in the general case, and so the benefits of an automated mechanism able to detect abrupt changes amongst many clusters become palpable.

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 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 the objects of interest (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.

### *2.4 Implementation details*

The current implementation uses the Extrae tracing toolkit [2] to automatically instrument MPI and/or OpenMP codes through library preloading techniques. For each entry and exit point of the parallel runtime, the tool writes a per-thread timestamped event trace, and collects hardware counters data through PAPI [9], and source code references by using libunwind [4] to walk the call stack and GNU binutils [3] 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 [14], 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 process up to 100K points under 1-2 minutes.

As reported in the literature, tracing tools already scale to hundreds of thousands of cores [13], and parallel density-based algorithms are able to manage millions of points [24]. 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.

## 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 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 ob-

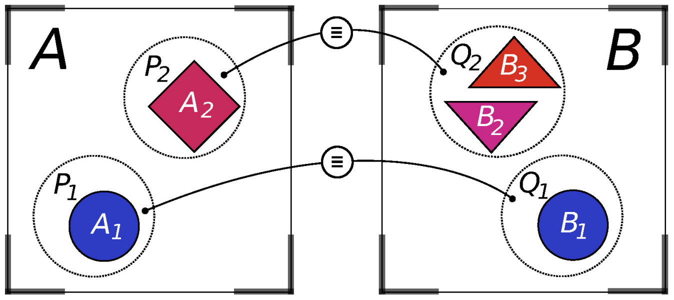


Fig. 3: Tracking scheme

jective 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:

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

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.

## *3.1 The tracking heuristics*

Recalling the difficulties to apply tracking on performance data that we previously discussed in Section 2.3, deciding whether two clusters from different experiments represent the same computational behavior requires to consider several 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 instrumentation phase). 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:

- *Distance 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 CPU burst in a cluster to the function, file and line in the source 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 to the same point in the source code.
- *Clusters density.* If there are performance variabilities that make the clusters split, there must be a combination of the split clusters so that the sum the computations that form each cluster equals the total number of computations that form the equivalent unsplit cluster in the previous experiment.
- *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 3.1.1 to 3.1.5 describe each of the heuristics in more detail. Then in Section 3.2 we explain how the information provided by the different heuristics is combined to maximize the number of objects successfully tracked.

### 3.1.1 Distance 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 2a and 2c). The general structure for both experiments hardly differs, with very slight movements.

There are situations where a cluster may split into two or more. For example, when new zones of imbalance appear and separate one region into several distinct performance behaviors. This case can be seen in Figure 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 2a to regions

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

Fig. 4: Cross-classification between WRF-128 and WRF-256

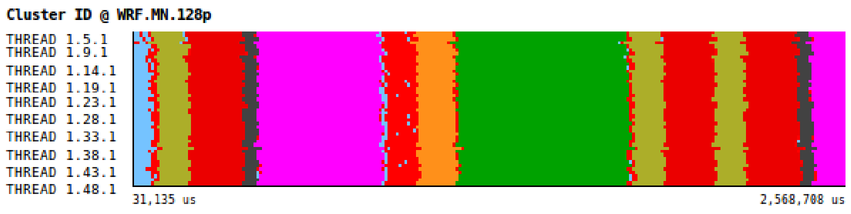
12 and 15 in Figure 2c, respectively. In these situations, cross-classification based on distance is likely not to assign the points to the correct cluster (both get assigned to 12 because 15 is too far away, which illustrates a mapping error), but we can then use the next heuristics to discern whether those regions are the same or not.

### 3.1.2 SPMDiness

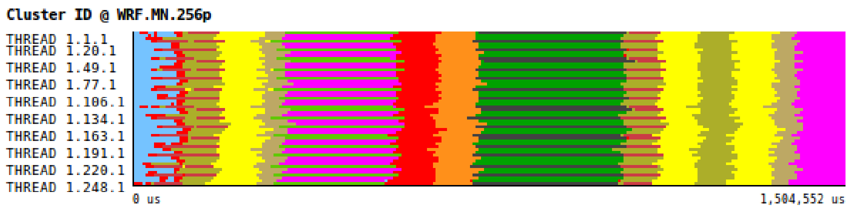
This heuristic exploits the SPMD structure of the applications to match computing regions that happen simultaneously in different processes. Assuming this execution model, all processors are expected to be executing the same phase of code at a time. In this case, if multiple processes are executing different 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).

Figure 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 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 [15]. The algorithm takes as input the sequence of clusters for every task of the application, and performs a Multiple Sequence Alignment (MSA). Clusters from different tasks that fall into the same position of the globally aligned sequence are those that get executed simultaneously, and we use this information to mark them as equivalent. If the application follows a programming model that may result in different processes running different parts of the code at the same time (e.g. task-based parallelism), this heuristic alone may lead to inconclusive decisions.



(a) SPMD computations for WRF-128



(b) SPMD computations for WRF-256

Fig. 5: Correlations from SPMDiness heuristic for WRF

### 3.1.3 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 clusters from two different frames do not share code references, they are certainly not equivalent.

Table 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 the selected metrics to generate the performance images, 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 multi-modal behaviors, it will appear as part of multiple clusters. This information alone is insufficient to discriminate more, but effectively reduces the combinatorial explosion.

### 3.1.4 Clusters density

This heuristic is applicable when comparing experiments that have computed the same number of CPU bursts. In those cases, the aggregate of computations of all the clusters in each performance image 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

Table 3: Correlations from call stack heuristic for WRF

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

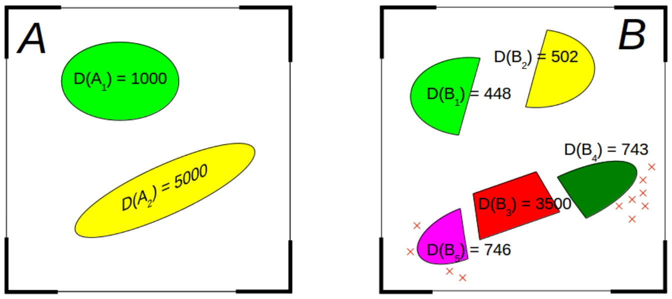


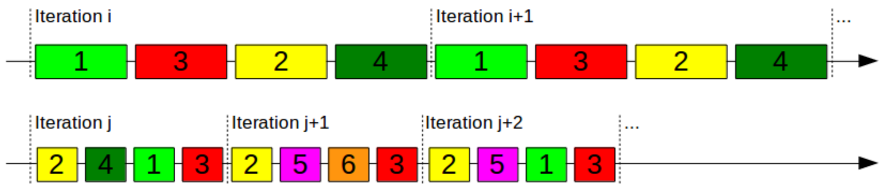
Fig. 6: Correlations from clusters density heuristic. The aggregate density of the split clusters on the right is lower or equal than the merged clusters on the left.

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 6.

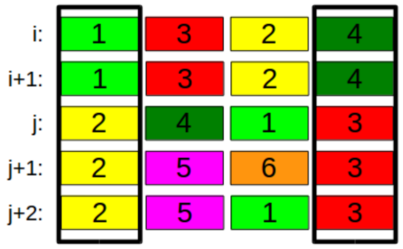
This problem can be formulated as a variant of the 0/1 knapsack problem [18]: given a cluster $i$ in the experiment $A$ with a certain density $D(A_i)$, find the combination of sub-clusters in the second experiment $B$ that maximizes the sum of their densities $D(B_{sum}) = D(B_1) + D(B_2) + ... + D(B_N)$ so that the aggregate density $D(B_{sum})$ is lower or equal than the limit density $D(A_i)$.

### 3.1.5 Chronological sequence

This heuristic assumes that the program execution order will not change between experiments, and so the sequence of computing bursts over time will preserve the same chronological order. If the execution flow of the program varies between experiments (e.g. the program is dependent on the input data set, and triggers different algorithms optimized for specific data sets), then this heuristic is not applicable.



(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.

Fig. 7: Correlations from chronological sequence heuristic

When the execution order is preserved, it is possible to determine equivalent code regions by looking into the position where the computations appear in the execution sequence and matching those in the same position.

The sequence alignment technique referred in [15] 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 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 7a.

As we have discussed earlier in Section 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 regions 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 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 the 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.

## 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 4 shows the correlations computed by the *density* 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 *SPMDiness* heuristic one correlation 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 *call stack* heuristic 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 *density* heuristic represents groups of clusters that have the same aggregate density. Lastly, the *chronological sequence* heuristic reflects the percentage of times where computations $A_i$ and $B_j$ happen in the same order of occurrence. In all cases, non-zero cells evince that a given pair of objects are the same with a certain probability, according to that heuristic. Occurrences with a very small probability (5% by default) are neglected as outliers.

Since every heuristic considers different properties of the objects, there might be contradictions on the correspondences found, and the results have to be combined to complement the correspondences that a given heuristic might fail to discern. To this end, a combination algorithm extracts from each correlation matrix a set of rules in the form $A_i \equiv B_j$ expressing which objects between two frames are equivalent, and reduces the rules applying a series of union and intersection operations. The union operation computes a logical *OR* and can be seen as complementing the results of different heuristics (e.g. one heuristic finds that $A_1 \equiv B_1$, and another finds that $A_1 \equiv B_2$, so we add up the results and consider that $A_1 \equiv B_1 \cup B_2$). In this case, an equivalence between two objects is kept always that at least one heuristic confirms it. The intersection operation computes the logical *AND*, and can be seen as the agreement between heuristics (e.g. in the previous example, there is no valid correspondence for $A_1$ because the heuristics did not agree). In this case, an equivalence betwen two objects is kept only if all the heuristics find that same correspondence, or discarded otherwise.

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. 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 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 *chronological 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.

## *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 8 (in logarithmic scale for better readability, refer to Figure 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 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

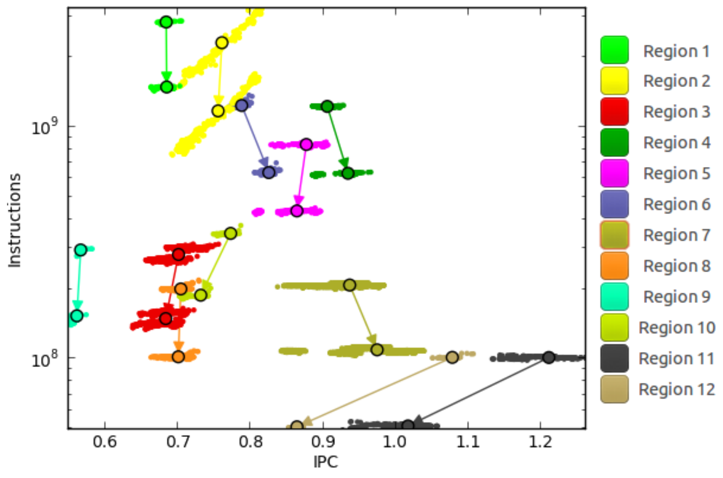


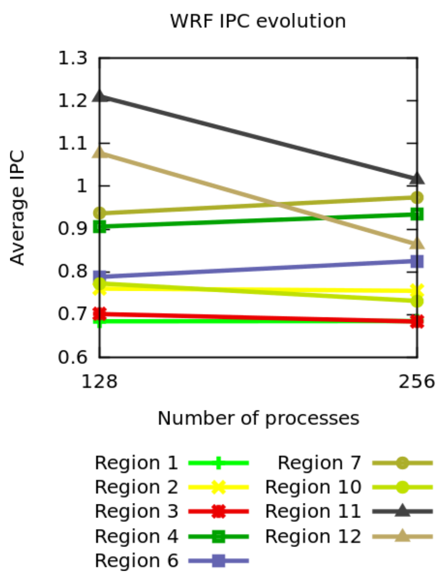
Fig. 8: Trajectories of clusters from WRF-128 to WRF-256

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.

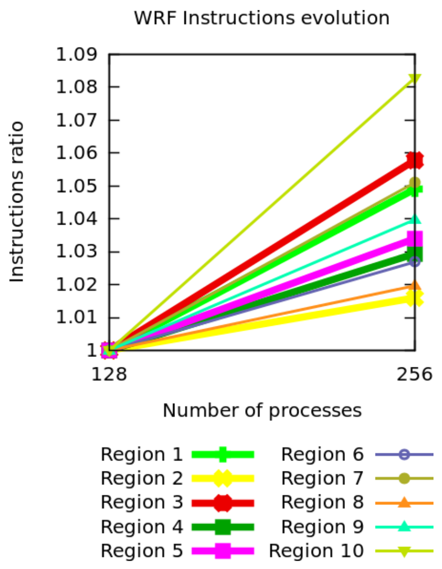
Amongst the regions that do not deteriorate due to the scale increase, Region 2 stands out for covering all alone the 15% of the execution time, and exhibiting an elongated cluster in the Y-axis that reflects large instructions imbalance, within a dynamic range that doubles from 1.5e9 for the 128-tasks 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 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 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 infor-



(a) IPC evolution



(b) Instructions evolution

Fig. 9: Performance trends for WRF code regions

mation 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.

## 4 Cases of study

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.

Table 4: Summary of experiments

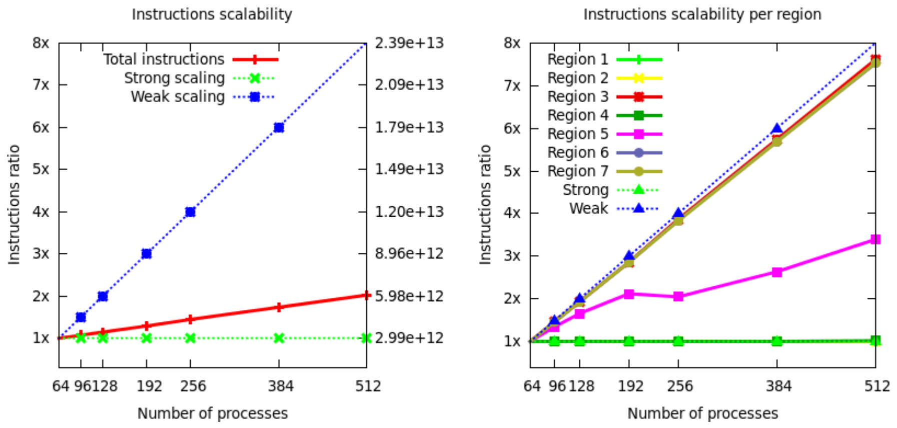
| Application | Input images | Tracked regions | Coverage |
|---|---|---|---|
| Gadget | 2 | 8 | 88% |
| QuantumE | 2 | 6 | 66% |
| WRF | 2 | 12 | 100% |
| Gromacs | 3 | 5 | 100% |
| CGPOP | 4 | 2 | 66% |
| NAS BT | 4 | 6 | 100% |
| OpenMX | 7 | 7 | 100% |
| Hydro | 8 | 3 | 100% |
| MR-Genesis | 12 |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

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 [1]. 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 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 seven case studies in more detail.

## 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 [6], a software package designed for the realization of large-scale ab initio calculations. To this 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.



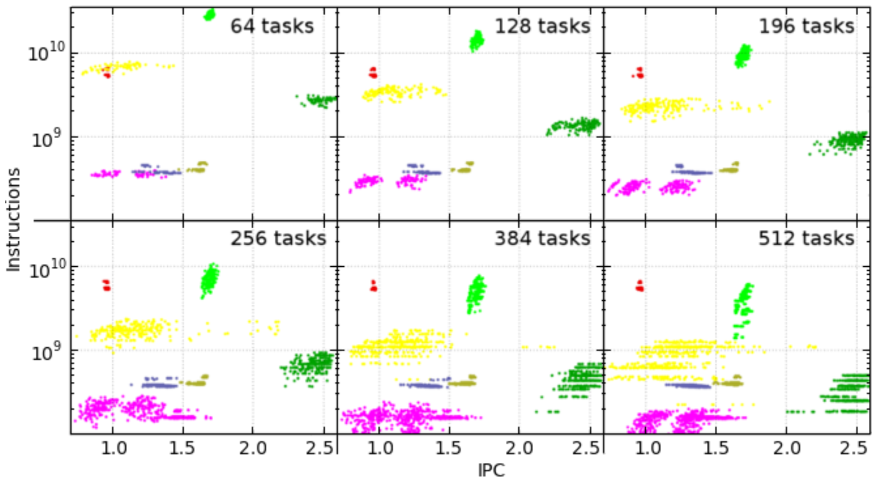
(a) Application scalability (b) Computations scalability

Fig. 10: Scalability of OpenMX

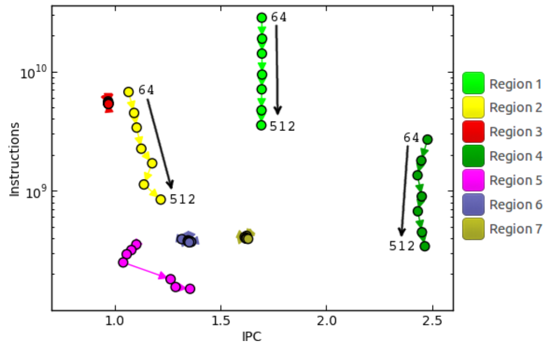
As we are running a strong-scale test (fixed-size problem on a varying number of processors), the application would ideally see the execution time reduced inversely proportional to the number of processors used. However, multiplying by 8 the number of tasks, the speedup achieved in a single time-step 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 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. Applying 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 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 in the Instructions and IPC axes 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 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.



(a) Sequence of output images from the tracking algorithm



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

Fig. 11: Tracking results for OpenMX

Figure 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* [11] 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.

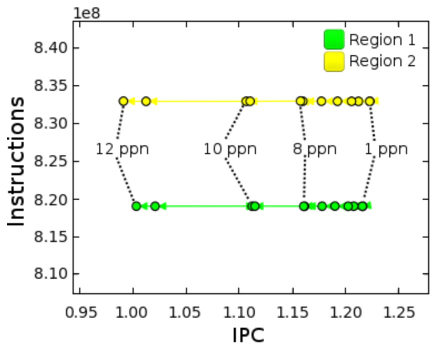
As a final remark, the detected hazards could have been inferred just from the first three frames in the sequence, and so our technique can be used with few cores to anticipate problems at higher scales, saving on time and resources.

## 4.2 Studying the impact of multi-core sharing

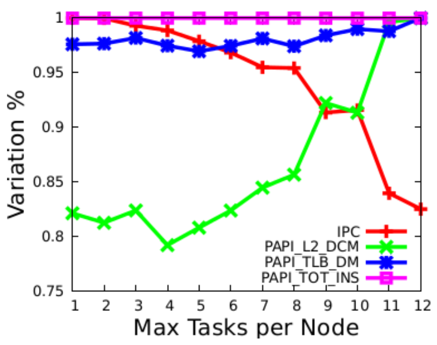
MR-Genesis [22] employs a finite volume approach in order to evolve the Relativistic Euler equations combined with a Constrained Transport scheme to account for the divergence free evolution of the dynamically included magnetic field. MR-Genesis was run in MinoTauro using 12 processes, changing the maximum 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 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 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.



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



(b) Region 1 evolution

Fig. 12: Tracking results for MR-Genesis

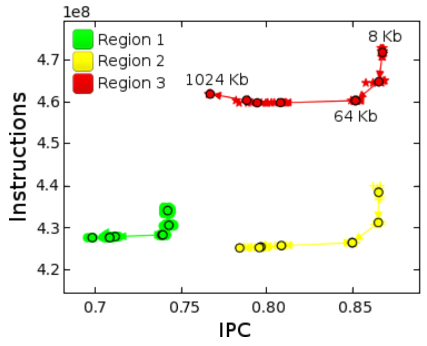
In this case, a fair trade-off between maximum utilization of the resources and the application performance is met at two-thirds of the node occupation.

## *4.3 Studying the impact of the program block size*

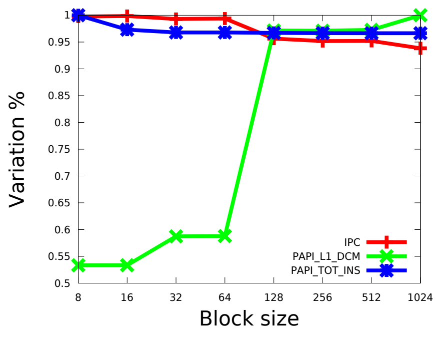
HYDRO [20] is a proxy benchmark that solves a large scale structure and galaxy formation problem using a rectangular 2D space domain split in blocks. HYDRO was run in MinoTauro, and the sequence of images in this case is built doubling the block size from 8 to 1024 Kb. The objective of this experiment is to determine which is the best setting for a particular parameter of the program to minimize the execution time.

Figure 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 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.



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



(b) Region 1 evolution

Fig. 13: Tracking results for Hydro

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.

### *4.4 Studying the impact of the problem input size*

The NAS Parallel Benchmarks [5] 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 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 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

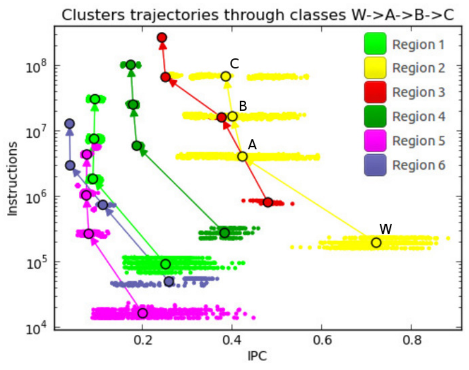
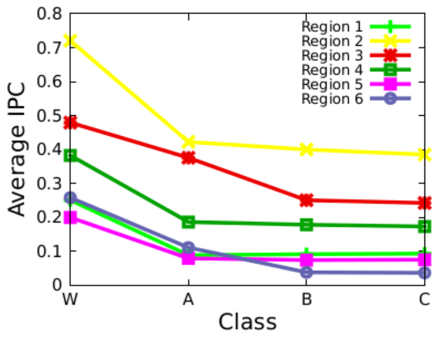
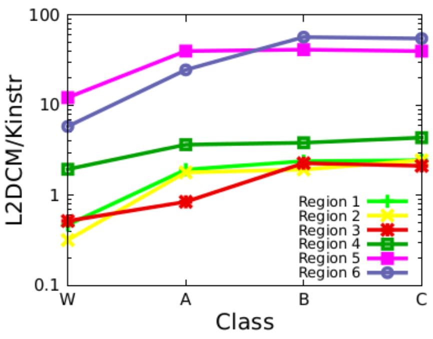


Fig. 14: Trajectories of clusters through BT experiments



(a) Evolution of IPC



(b) Evolution of L2 Cache Misses

Fig. 15: Performance trends for NAS BT code regions

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 15b.

### *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 the machine where it is executed and also changing from a generic to an architecture specific compiler. This test shows that even in very different scenarios that may result in large performance variations, the tracking algorithm is able to follow the evolution of the clusters.

Table 5: CGPOP performance results

| | | MareNostrum II | | 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 |

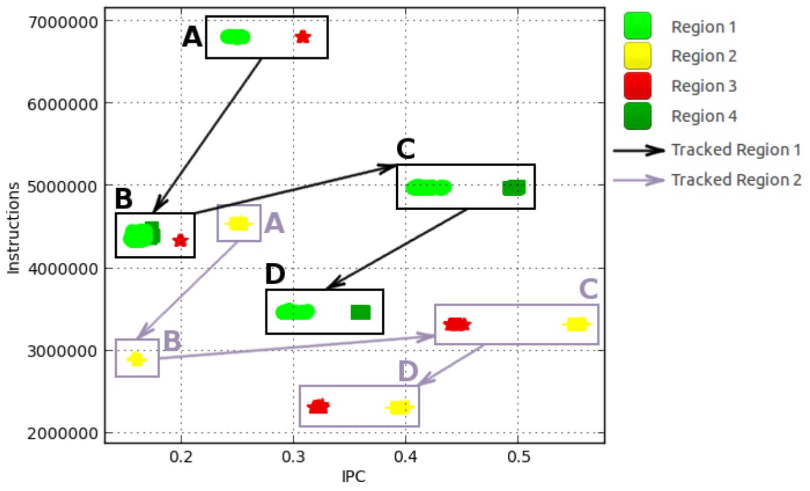


Fig. 16: Trajectories of clusters through CGPOP experiments: (A) MareNostrum GNU (B) MareNostrum IBM XL (C) MinoTauro GNU (D) MinoTauro Intel.

CGPOP [7] is a proxy application of the Parallel Ocean Program [19]. 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 an aggressive optimization flag (-O3) and debug (-g).

Figure 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 16b) all computations see the number of instructions significantly reduced (36% and 33%, respectively) compared to using gfortran (see 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 16c and 16d), with an overall improvement in terms of less instructions executed and

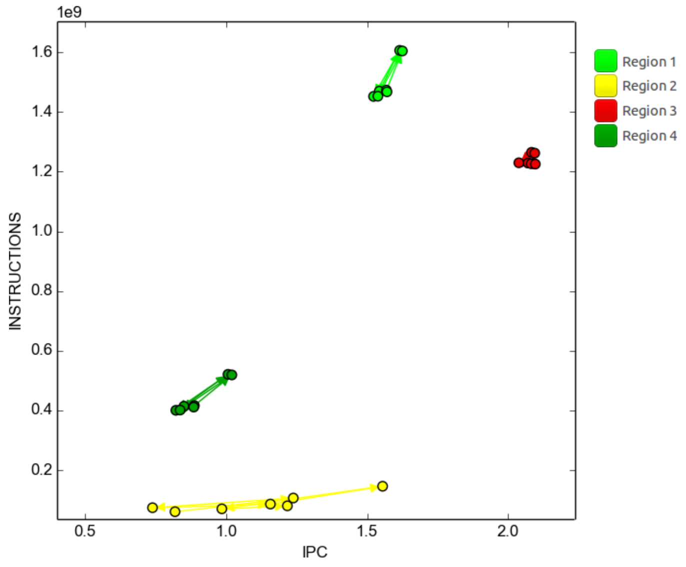


Fig. 17: Trajectories of clusters through HACC weak scale experiments

higher IPC achieved, yet the same degradation 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 5 summarizes the averages for IPC and instructions for both tracked regions, and their elapsed execution 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 but 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 $\pm 0.03\%$.

## 4.6 Studying the effect of optimal and non-optimal grid geometries

HACC (Hardware/Hybrid Accelerated Cosmology Code) is a framework that melds particle and grid methods to satisfy the requirements of cosmological surveys, ex-

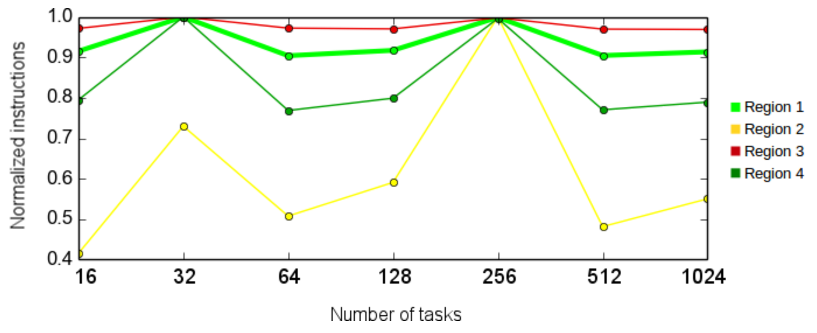


Fig. 18: Average instructions executed per computing region in HACC

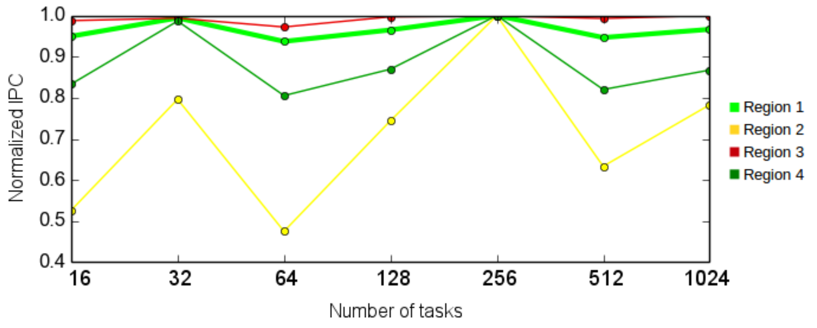
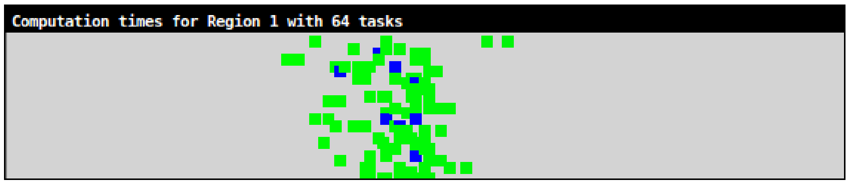


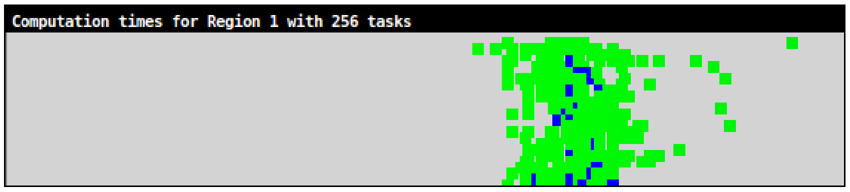
Fig. 19: Average IPC achieved per computing region in HACC

ploiting hybrid and accelerator-based architectures with millions of cores, including CPU/GPU, multi/many-core, and Blue Gene systems. HACC is designed to scale weakly by dividing the working data set 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 III, 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 memory caches sharing), using the Intel MPI message passing library, and without support for threads.

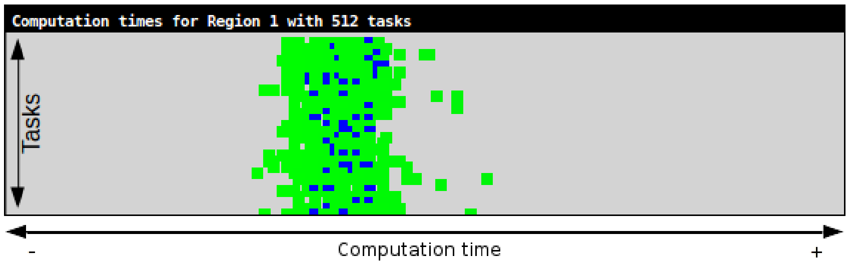
Figure 17 shows the trajectories of the main computing regions of HACC. Here we can observe zigzag movements back and forth: as we increase the number of tasks (and so the size of the problem in proportion), all regions move upward (more instructions executed) and rightward (more IPC achieved). However, 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 18 and 19 show this effect more clearly. Figure 18 shows the amount of instructions executed per region across experiments. The lower workload is found at experiments 3 and 6 (the cubic cases with 64 and 512 tasks). Correlating with Figure 19, these two experiments are also the ones achieving lower IPC.



(a) 64 tasks



(b) 256 tasks



(c) 512 tasks

Fig. 20: Histograms of durations for computing region 1 in HACC

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 20, that compares the computation times for the main computing region of the program (Region 1) in the cubic runs (64 and 512 tasks), and an uneven intermediate case (256 tasks). In these histograms, the rows represent processes and the columns are bins of computation durations increasing from left to right, and the colors represent the time spent on a particular bin, ranging from green (low time) to blue (high time). In the cubic cases (see Figures 20a and 20c), the computing times are very similar in the range of 310 to 323 ms, but in the 256 tasks case (see Figure 20b), all computations are shifted to the right, having increased their 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 see that the percentage of

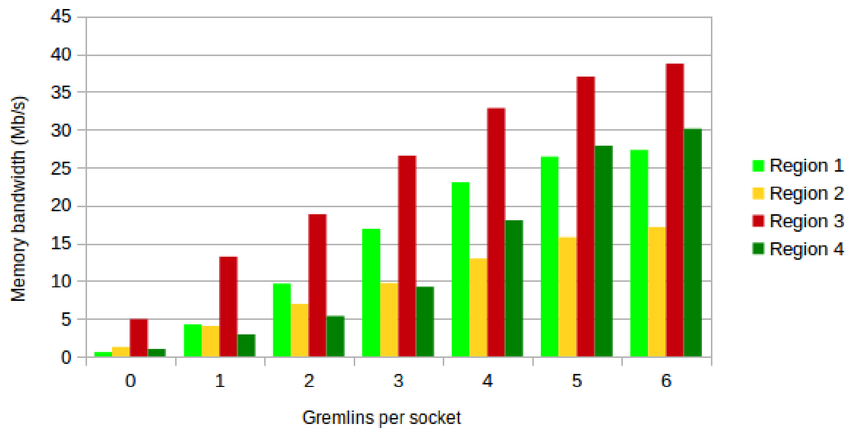


Fig. 21: Memory bandwidth consumed by the main computing regions of LULESH with increasing levels of interference

time spent in computations decreases from 60 to 45%, and so the communications start to dominate the execution time. Analyzing the trace, the time spent in MPI_Wait calls increases 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 communications with computations, reducing the serializations.

## *4.7 Studying the effects of memory congestion*

LULESH is a shock hydro mini-app. While designed to test many machine and hardware features in particular it stresses compiler vectorization and OpenMP overheads. LULESH performs a hydrodynamics stencil calculation using both MPI and OpenMP to achieve parallelism. In general the compute performance properties of LULESH are more interesting than messaging as on a typical modern machine only about 10% of the runtime is spent in communication.

In this experiment we measure how sensitive are the computations of this program to memory congestion. To do so, we interfere the execution collocating gremlin processes [12] in the same nodes where the application is running, constantly consuming a large amount of memory bandwidth by contaminating the L3 shared memory cache. One gremlin is activated at a time every few seconds across the execution. So the application starts running clean, then it is interfered by one gremlin

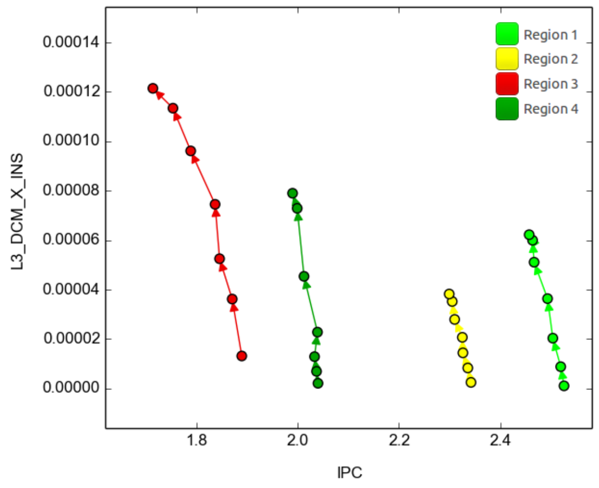


Fig. 22: Trajectories of clusters through LULESH experiments (from bottom to top, 0 to 6 gremlins per socket)

after a while, then by a second gremlin and so on until a maximum of 6 gremlins per socket and node are interfering. Every single gremlin consumes an average memory bandwidth of 600 Mb/s. The bandwidth has been approximated with the following formula: $BW = (L3_{miss} * L_{size})/T_{burst}$; where $L3_{miss}$ is the number of misses in the last level cache, and so is the number of times that data is fetched from the main memory; $L_{size}$ is the size of the cache line, and $T_{burst}$ is the duration of the computing region.

Figure 21 shows the average bandwidth consumed by the different computing regions of LULESH with increasing number of gremlins. As the reader can see, each new gremlin increases the average bandwidth consumed by the application due to the extra memory accesses needed to cope with the higher number of cache misses.

Figure 22 shows the trajectories of the main 4 computing regions of LULESH with respect to the IPC achieved (X-axis) and the number of L3 misses (Y-axis). All regions move upwards (meaning that the number of L3 cache misses increase with the number of interfering gremlins), and also move leftwards (meaning that all regions loose performance with higher levels of interference). Region 3 (red) is the one that moves further in both axes, which means that is the computation most affected by the interferences, and corresponds to the computations performed at the CommMonoQ routine. This routine copies data from several structures with a non-consecutive stride, making more misses because the application is not taking advantage of the temporal data locality.

## 5 Related work

Our work draws inspiration from a motion detection algorithm of moving biological objects that are similar but non-homogeneous [23]. They apply multi-feature contour segmentation and flux tensors for identifying the boundaries of biological objects and the detection of 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 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 and obstacle avoidance. [30] 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 [25] is a post-mortem performance tool that includes 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 [29] 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 [17] 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 [28] 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 [21] 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.

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 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 distances in the performance space, the SPMDiness of the application, the code region they refer to, the clusters densities and the chronological 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 bottlenecks 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.

## References

1. BSC Facilities. http://www.bsc.es/marenostrum-support-services.
2. BSC Tools. http://www.bsc.es/paraver.
3. GNU Binutils. http://www.gnu.org/software/binutils.
4. The libunwind project. http://www.nongnu.org/libunwind.
5. NAS Parallel Benchmarks. http://www.nas.nasa.gov/Software/NPB.
6. Open source package for Material eXplorer. http://www.openmx-square.org.
7. The CGPOP Miniapp website. http://www.cs.colostate.edu/hpc/cgpop.
8. The Weather Research & Forecasting model. http://www.wrf-model.org.
9. 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.
10. B. Buck and J. K. Hollingsworth. An API for Runtime Code Patching. *The International Journal of High Performance Computing Applications*, 14:317–329, 2000.
11. 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.
12. 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.
13. 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.
14. J. González et al. Automatic Detection of Parallel Applications Computation Phases. In *IPDPS: 23rd IEEE International Parallel and Distributed Processing Symposium*, 2009.
15. J. González et al. Automatic Evaluation of the Computation Structure of Parallel Applications. In *PDCAT: Proceedings of the 2009 International Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 138–145, 2009.
16. J. González et al. Performance Data Extrapolation in Parallel Codes. In *ICPADS: 16th IEEE International Conference on Parallel and Distributed Systems,*, pages 155–163, 2010.
17. K. A. Huck and A. D. Malony. PerfExplorer: A Performance Data Mining Framework For Large-Scale Parallel Computing. In *Proceedings of the conference of Supercomputing*, page 41, 2005.
18. O. H. Ibarra and C. E. Kim. Fast approximation algorithms for the knapsack and sum of subset problems. *J. ACM*, 22(4):463–468, Oct. 1975.
19. P. Jones. Parallel Ocean Program (POP) user guide. Technical report, Los Alamos National Laboratory, March 2003.
20. P.-F. Lavalléea et al. HYDRO. http://www.prace-ri.eu.
21. J. Mellor-Crummey. HPCToolkit: Multi-Platform Tools for Profile-Based Performance Analysis. In *APART*, November 2003.
22. 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).
23. K. Palaniappan et al. Moving Object Segmentation Using the Flux Tensor for Biological Video Microscopy. In *PCM*, page 483, 2007.
24. 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.
25. P. C. Roth. ETRUSCA: Event Trace Reduction Using Statistical Data Clustering Analysis. Master's thesis, University of Iowa, 1992.

26. H. Servat et al. Detailed Performance Analysis Using Coarse Grain Sampling. In *PROPER*, 2009.
27. H. Servat et al. Unveiling Internal Evolution of Parallel Application Computation Phases. In *ICPP*, pages 155–164, 2011.
28. S. S. Shende and A. D. Malony. The Tau Parallel Performance System. *Int. J. High Perform. Comput. Appl.*, 20:287–311, 2006.
29. F. Song et al. An Algebra for Cross-Experiment Performance Analysis. In *ICPP*, pages 63–72, 2004.
30. A. Yilmaz et al. Object Tracking: A Survey. *ACM Comput. Surv.*, 38(4), Dec. 2006.