On the usefulness of object tracking techniques in performance analysis

Germán Llort, Harald Servat, Judit Giménez and Jesús Labarta

Barcelona Supercomputing Center
Universitat Politècnica de Catalunya
e-mail: {gllort, harald, judit, jesus}@bsc.es

***Abstract*—Object tracking in computer vision has been applied for many years in the context of applications that require the recognition of a moving object in an image or every frame of a sequence of scenes, with practical uses in a wide range of fields, including human-computer interaction, security and surveillance, traffic control, augmented reality and more.**

In performance analysis, it is crucial to understand the details of a parallel application if we are to tune it to achieve its maximum performance. However, the actual application behavior is directly bound to its execution conditions: input data sets, configuration variables, number of processes, specific hardware, etc. Then, beyond the details of a single-experiment, a far more interesting question arises. How does the application behavior change along with the execution parameters?

In this paper, we leverage object tracking techniques to analyze how the behavior of a parallel application evolves through multiple experiments where the execution conditions change. This method provides apprehensible insights on the influence of the application parameters on its behavior, and helps to identify performance trends of the different computing phases and how do they scale.

I. Introduction

When starting out the performance analysis of a parallel scientific application, the user usually faces a wide range of choices to make. How many processes to run the application with, the inputs to use, the number of timesteps to simulate, the minimum accuracy of the results, hardware settings, and many other configuration options 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.

While many useful information can be extracted from the analysis of a single experiment, it is not until we understand how all these variables affect the performance of the program that we get a complete understanding of its behavior. To this end, the user will often require to run multiple experiments with different configurations.

At the moment that you try to compare two or more experiments with different configurations, one has to take into account that the performance characteristics of the different parts of the application can change significantly. For instance, consider the same application executed twice, doubling the number of processors. If the work distribution is well balanced, one would expect the number of instructions executed in the different computing regions to drop by a half.

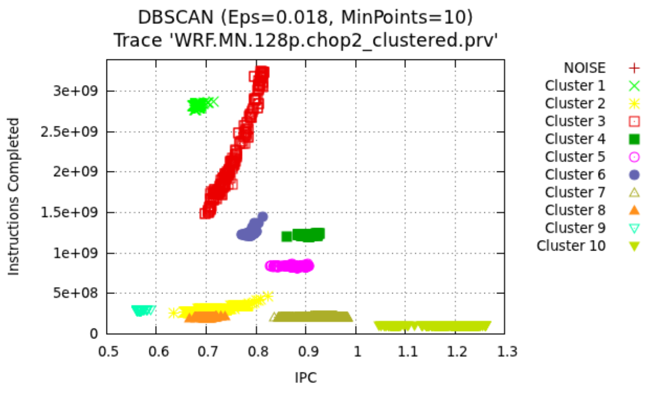
In addition to changes in the application scale, there is a good variety of factors that may have an impact on the execution, such as the size of the problem and the physical mapping of tasks onto nodes; and multiple aspects that might be interesting to evaluate, including the temporal evolution of the program, CPU throttling adjustments, or different parallel implementations (i.e. MPI vs. OpenMP).

Understanding the effect of these variations is important not only to get better knowledge of the application behavior, but also to quantify how much performance improves or degrades, and to foresee tendencies of the different parts of the code. Such type of analysis indicates the expert the right direction to focus all the optimization effort.

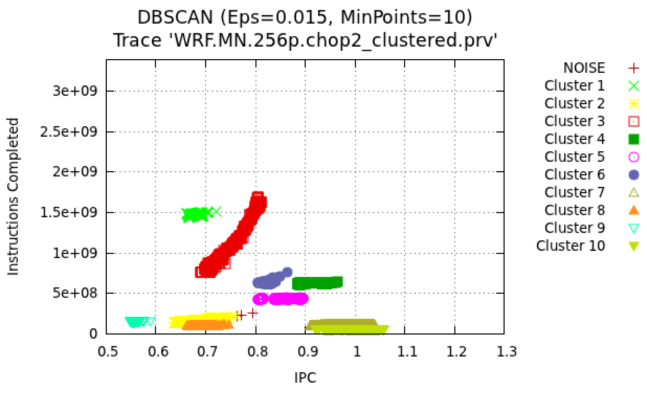
The main contribution of this paper is to bring the concept of object tracking into the world of performance analysis, with the objective of studying performance variations due to changes in the execution conditions across multiple experiments. Traditionally, tracking techniques are used to locate a moving object in an image or sequence. Similarly, we are expressing the different parts of the application as trackable objects in a space whose dimensions are not the actual physical dimensions of height, length and breadth, but performance metrics.

Difficulties in tracking objects arise due to abrupt object motion and changes of appearance. Tracking code regions across the performance space can borrow certain assumptions about the object's direction and motion speed that makes the task easier. Generally speaking, an application performance will neither radically change all of a sudden, nor behave diametrically opposed as it used to. In this sense, it is feasable to detect the evolution of every region of the code across different experiments. Then, for each region we can compute automatic metrics to evaluate their performance trends and scalability.

The rest of the paper is structured as follows. Section II describes the way we express the application performance metrics as trackable objects. Section III describes the tracking algorithm adapted to this particular use case. Finally, we compile the conclusions and outline the future directions in our research in Section IV.



(a) Running 128 tasks



(b) Running 256 tasks

Fig. 1: Structure of WRF computing bursts

II. Clustering for structure detection

Cluster analysis has proved useful in the context of performance tools to characterize the structure of an application. The approach presented in [1] applies density-based clustering with respect to several performance metrics (i.e. instructions completed, IPC, cache misses) in order to identify behavioral trends in the computing phases. A practical use case of this technique is, for instance, to differentiate invocations to the same routine with variable performance.

The clustering algorithm takes as input the computing regions (i.e. CPU bursts) of the application, which refer to the sequential computations between MPI communications. This information is extracted from a Paraver trace [2], a sequence of time-stamped events that records performance measurements of the application during the run. Every CPU burst is defined by its duration and a set of performance metrics read at the beginning and end of the region. By grouping together all computations that have similar performance with respect to these metrics (i.e. instructions completed, IPC, cache misses, etc.), the structure of all tasks of the application gets clearly exposed.

Figure 1a shows the structure of the ten most time-consuming computing regions of the WRF application [**?**]. Clusters are formed according to similarities in the achieved performance (X-axis) and number of instructions executed (Y-axis). Those that stretch vertically (i.e. cluster 3) denote instructions imbalance, while those that stretch horizontally (i.e. 7 and 10) reflect IPC variations. Multiple computations happening simultaneously that perform the same amount of work but executed at variable speeds, or vice-versa, can be the source of significant load-imbalances in a parallel application, and thus they are interesting to be studied. Time and space variations can be studied over the trace timeline. Figure 2 shows the clusters distribution across time and processes (X- and Y-axis, respectively). As you can see, this application follows a marked SPMD model where all processes perform the same type of computation at the same time, in a clear pattern that repeats over iterations.

The way clusters are formed mainly depends on the structure of the data being analyzed. When the execution conditions change, so will the application performance, and thus will the clustering results. Figure 1b shows the computations structure of the same application, but doubling the number of cores in the execution. The number of instructions executed per cluster has reduced in inverse proportion, and so all clusters have moved down across the Y-axis. A few clusters have slightly improved their performance (i.e. 4, 5, 6 and 7 moved right to higher IPC's), while cluster 10 significantly degraded.

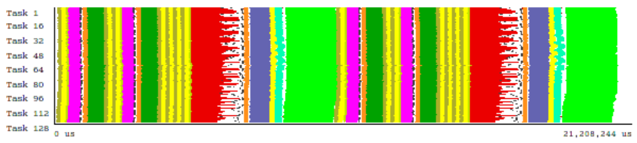


Fig. 2: Clusters distribution of WRF over time

Clusters can not only move long distances in the space or change their shape, but they can vary their density, split, or merge together. Because of these variations, it is possible that the same computing regions are represented by different clusters in different experiments, and might not be easy to relate how much did they change from one experiment to another.

The objective of this work is to perform an automatic detection of these variations across different experiments in relation to changes in the execution conditions. To this end, we extrapolate the concept of recognizing moving objects in a sequence of images to clusters. Clustering the application can be seen as an image that describes its global structure. Subsequent clusterings result then in a sequence of images that can be compared to see changes in the application behavior. Every cluster, that represents a region of code, can move or change its appearance across images, which reflects performance variations. Tracking their evolution across different experiments, enables us to study the performance characteristics of the different computing phases of the application, and to understand how the different execution conditions get to influence the behavior of the application.

III. Performance Tracking

The objective of this mechanism is to automatically determine which clusters are equivalent across different experiments with changing conditions (i.e. scale the number of processes, size of the problem, different hardware, etc.). The main difficulty lies in reacting to consequent long displacements and changes in their appeareance (shape, density, etc) across the sequence of clusterings.

Even though the execution conditions of an application change, its general behavior will usually neither turn to be very different, nor change abruptly. For instance, the expected outcome of doubling the size of the problem computed, would be the execution to take twice the time to complete. In general, changes in the execution conditions will mean proportional, sometimes predictable, changes in the application behavior. In this sense, it is reasonable to assume certain restrictions regarding the direction and speed of the moving objects between scenes.

When comparing two clusterings, we are generally bound to expect slight cluster movements between executions, or the whole cluster space to move as a pack in a common direction (i.e. all clusters improve performance or execute less instructions). This assumption allows to establish relations between clusters based on a proximity criteria, being the nearest counterpart cluster the more likely to be the same. Our first approach to the tracking problem uses a nearest-neighbor Cross-Classification algorithm to measure the distances covered by moving clusters and correlate those that are closer.

Considering only the minimum distances might not be enough to find an exact correlation between all clusters. In some cases, they can move long distances with variable directions, which can lead to incorrect matches. In addition to this, we pay attention to other structural characteristics of the clusters that can be extracted from the information comprised in the trace, that can help us to evaluate whether two clusters are equivalent:

- Callstack references. Clusters are linked to concrete regions of code if using optional callstack information, that expresses from which point of the code a given cluster happens. There will be an explicit relation between two clusters that are executed in the same code regions.
- SPMD alignment. If the application is SPMD, all processes will be executing the same code regions simultaneously. At any point in the execution sequence, all clusters that are happening simultaneously should refer to the same region of code.

The following sections describe the previous evaluators in detail and how they are combined to build a decision heuristic that determines whether two clusters are the same or not despite structural differences. This heuristic is contrasted with all resulting clusters in order to determine a global sequence that shows how does each cluster evolve across all experiments.

A. Cross-classification

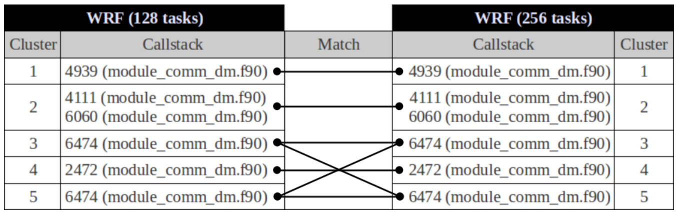
This evaluator takes a pair of clusterings and performs a nearest-neighbor classification of all points from the first into the latter, and viceversa. The classification criteria is based on euclidean distances, so that all points will get classified to the nearest counterpart cluster.

The idea that lies behind this process supports on the fact that the behavior of a parallel application will not radically change across experiments. Consider the previous example where we doubled the number of processes we run the application WRF with (see Figure 1. One can then expect a proportional decrease in the number of instructions executed per each cluster. Geometrically, all clusters uniformly move down in the instructions axis of the clustering space, and the relative distances between keep constant. This results to be true for the general case, thus cross-classifying to the nearest cluster should usually result in a one-to-one mapping association of clusters.

However, there are frequent exceptions where the points that conform a given cluster will split into two or more, as there are multiple target candidate clusters that are close enough. This can be seen in Figure 3, where the matrices show the percentage of points from clusters in one of the experiments classified among clusters of the other. Only those cells with a value of 100% express that for the given cluster an unequivocal counterpart is found.

Also, there are infrequent cases where some clusters can move a long way in the space, as can be the case of Cluster 10 from 1a to 1b. In these situations, cross-classification based on distance is likely not to assign the points to the correct cluster, but we can then use the complementary evaluators to decide whether the cluster is the same or not.

B. Callstack references

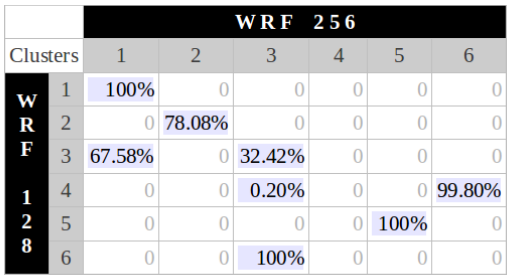


WRF (128 tasks)			WRF (256 tasks)	
Cluster	Callstack	Match	Callstack	Cluster
1	4939 (module_comm_dm.f90)		4939 (module_comm_dm.f90)	1
2	4111 (module_comm_dm.f90) 6060 (module_comm_dm.f90)		4111 (module_comm_dm.f90) 6060 (module_comm_dm.f90)	2
3	6474 (module_comm_dm.f90)		6474 (module_comm_dm.f90)	3
4	2472 (module_comm_dm.f90)		2472 (module_comm_dm.f90)	4
5	6474 (module_comm_dm.f90)		6474 (module_comm_dm.f90)	5

Fig. 4: Clusters tracking through callstack information

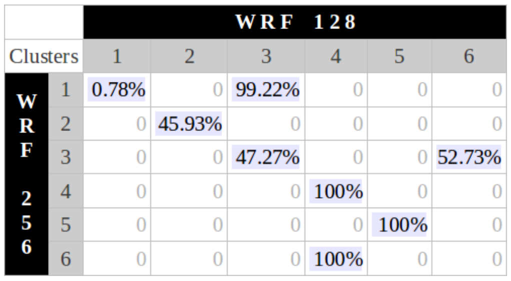
The Extrae tracing toolkit [2] can optionally collect callstack information that points to which function, file and source code line is a given instrumentation probe called from. In this way, every computing burst is unequivocally linked to a particular code point.

If two given clusters are equivalent, it is clear that the computations than conform them have to refer to the same source code regions. Therefore, this evaluator prunes the search space looking for clusters that share the same code references. In the particular case where two clusters are executed from the same unique code region, we are positevely sure that both clusters



		WRF 256					
	Clusters	1	2	3	4	5	6
WRF 128	1	100%	0	0	0	0	0
	2	0	78.08%	0	0	0	0
	3	67.58%	0	32.42%	0	0	0
	4	0	0	0.20%	0	0	99.80%
	5	0	0	0	0	100%	0
	6	0	0	100%	0	0	0

(a) Classify data from the 128 tasks experiment into the 256



		WRF 128					
	Clusters	1	2	3	4	5	6
WRF 256	1	0.78%	0	99.22%	0	0	0
	2	0	45.93%	0	0	0	0
	3	0	0	47.27%	0	0	52.73%
	4	0	0	0	100%	0	0
	5	0	0	0	0	100%	0
	6	0	0	0	100%	0	0

(b) And vice-versa, from 256 to 128

Fig. 3: Cross-classification example

are the same. But for this to be true, there must not be other clusters sharing the same callstack reference.

Figure 4 illustrates the relations that can be outlined between clusters from their code references. Both clusters 3 and 5 share the same reference to the line of code 6,474 in the file module_comm_dm.f90, meaning that this code region presents two different behaviors. Though on its own, this information does not allow to differentiate whether these clusters are respectively the same, or whether they have swapped from one experiment to the other.

C. *SPMD alignment*

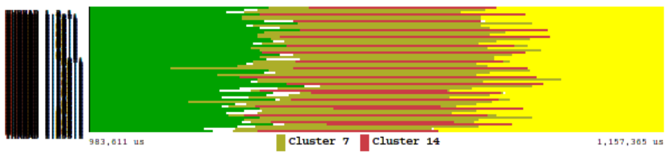


Fig. 5: Time sequence of clusters in WRF

When the application model is SPMD, all processors are expected to be executing the same code region at a time. In this case, if two different clusters happen simultaneously, they are likely to refer to the same code region, although there might be structural differences that make them separate (i.e. the application presents work imbalance and some processes execute more instructions than others). For instance, Figure 5 shows that different processes are executing Clusters 7 and 14 simultaneously, while they actually refer to the same code region but slight performance variations make them separate, and should be seen just as one single cluster.

Whether the application is SPMD is evaluated with the sequence algorithm presented in [3]. This algorithm computes a score that quantifies the degree of SPMD-ness, and generate the execution sequence of clusters for every task of the application. Aligning these sequences we can detect whether two or more clusters get executed simultaneously at different application tasks. The drawback of this evaluator is that it requires the application to be highly SPMD. Otherwise, the cluster sequences might not align correctly, which would lead to false positives.

D. *Multi-criteria heuristic*

The evaluators introduced above consider different characteristics of the clusters, but they are not infallible and have their own strengths and weaknesses each. Callstack linking can be univocal, but it is not always; rely on clusters alignment is limited to SPMD applications; and cross-classification might miss clusters if their structure changes significantly. While the latter is always applicable, the first two depend on the availability of extra information and the application structure. On their own, it is not difficult to find test cases to prove them wrong. However, it is possible to combine all three to determine a precise correlation of clusters across different experiments with a very small margin of error.

If callstack information is present and the code references are univocal, there is an unequivocal relation between clusters and no extra analysis is required. When there are multiple counterpart candidate clusters, we then intersect the results of the calltack and cross-classification evaluators. This makes matches to the nearest clusters, amongst all that refer to the same code region. For the set of clusters that results from the intersection, we then use the alignment evaluator to see whether any other cluster not already included in the set gets executed simultaneously with any of those in the set, and such cluster is also added. This can be seen as finding the relation between different clusters that refer to the same code region, but they are structurally so different that the cross-correlation fails to match them.

Clusters correlations are computed for every pair of subsequent clusterings. Then the tool builds a global sequence illustrating the evolution of each cluster across the different experiments. Having detected which clusters are equivalent, we perform a recoloring process and produce scatter plots that show the structure of the application keeping the same color for clusters that represent the same code regions, as shown in Figure 6. These plots can be displayed together or in simple animation, so that it is very easy to identify alterations in the clustering space, and whether the clusters move, merge, split or change their shape. This enables the study of the repercussion of the changes in the execution conditions on every particular region of code.

In addition, we compute averaged metrics per cluster for all the clustering dimensions and experiments. These metrics are drawn in a tendency line chart that shows how does a cluster

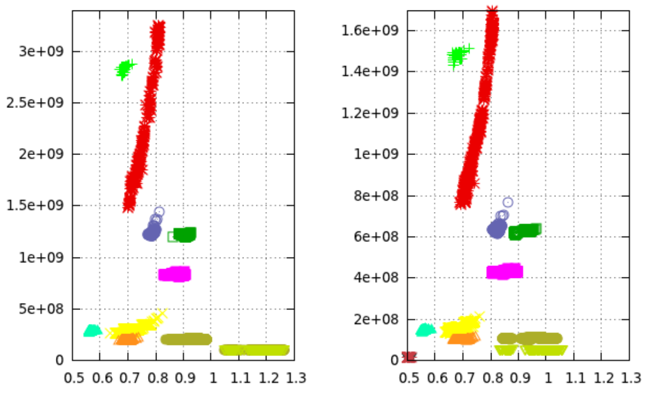
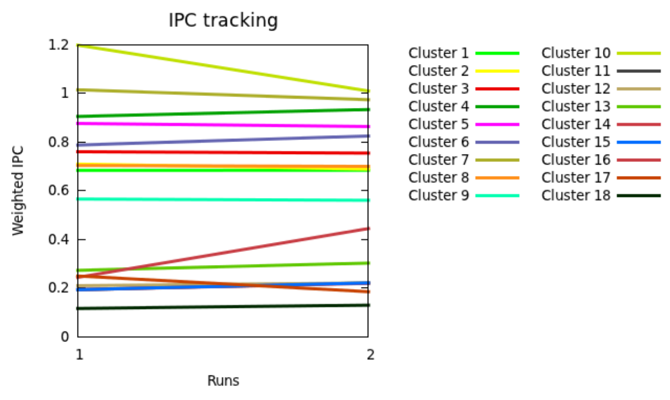
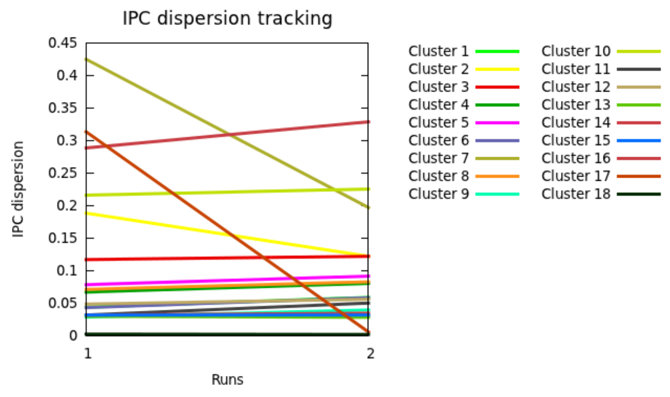


Fig. 6: Recolored multiplot



(a) IPC tendency



(b) IPC dispersion tendency

Fig. 7: Tendency lines for WRF clusters

evolve according to every particular performance parameter. Figure 7a shows the evolution in *Instructions per Cycle* of WRF from 128 to 256 tasks. It is easy to see that clusters 10 and 17 present the most significant degradation, while 16 improves performance. The metric displayed in 7b represents the dispersion in IPC, which decreases for clusters 7 and 17, and remains more or less constant for the rest. The fact that the dispersion does not increase with the number of processes suggests that the application is well balanced.

This information can be used to gain understanding of the influence of the execution parameters on the application performance, predict the outcome of future experiments and study how well the application scales.

IV. Conclusions

There is a good number of parameters that can potentially modify the execution of an application: the number of processes that it is run with, the size of the problem to solve, specific inputs and particular configuration variables, hardware, etc. In order to get better knowledge of the application behavior, it is important to understand how all these parameters can affect the execution, which will often require multiple experiments with different adjustments.

Applying cluster analysis to the computing regions of the application we can identify their relevant characteristics and gain insight into their structural properties. But modelling clusters based on performance data is strongly tied with the application behavior. If the execution conditions change, so will the performance of the application, and so will the resulting clusters. Multi-experiment cluster analyses provides useful information to study how any parameter will affect the application performance, but it is necessary to keep track on how do clusters change along with the execution conditions, so that we can see the evolution of the different computing regions.

In this paper we have presented a mechanism that analyzes multiple clusterings that result from different execution conditions, and automatically correlates the clusters that refer to the same computing regions across all clusterings. To perform this who-is-who correlation we use different evaluators that take into account different characteristics of the clusters: the code region they refer to, their likeness in terms of the clustering dimensions, and the SPMD-ness of the application. Combining their use, we identify a global sequence for each cluster that shows the evolution of its characteristics across the different experiments, no matter changes in their shape, density, performance, etc.

Over the clusters sequences we automatically compute averaged metrics for each clustering parameter (i.e. IPC, instructions executed, cache misses, etc.), and draw tendency lines that shows the direction in which every cluster progresses. We also generate scatter plots that show the structure of the computing regions, and perform a recoloring process to keep the clusters that refer to the same region of code represented with the same colors. In this way, a simple animation can easily display changes in the clustering space.

All in all, this work enables the analyst to understand how the different execution parameters have an inpact on the performance; predict the outcome of future experiments; study how well the application scales; and ultimately helps to gain full understanding of the application behavior, much beyond what can be learned from a single specific experiment.

Acknowledgment

The work is granted by the IBM/BSC MareIncognito project and by the Comisión Interministerial de Ciencia y Tecnología (CICYT), contract TIN2007-60625.

References

[1] 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.

[2] "BSC Tools," http://www.bsc.es/paraver.

[3] 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*, ser. PDCAT '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 138–145. [Online]. Available: http://dx.doi.org/10.1109/PDCAT.2009.52