A Methodology Towards Automatic Performance Analysis of Parallel Applications

Maria Calzarossa, Luisa Massari, Daniele Tessera
Dipartimento di Informatica e Sistemistica
Università di Pavia
I-27100 Pavia, Italy
{mcc,massari,tessera}@alice.unipv.it

ABSTRACT

Tuning and debugging the performance of parallel applications is an iterative process consisting of several steps dealing with identification and localization of inefficiencies, repair, and verification of the achieved performance. In this paper, we address the analysis of the performance of parallel applications from a methodological viewpoint with the aim of identifying and localizing inefficiencies. Our methodology is based on performance metrics and criteria that highlight the properties of the applications and the load imbalance and dissimilarities in the behavior of the processors. A few case studies illustrate the application of the methodology.

Keywords: parallel application; performance analysis; performance metrics; load imbalance.

1 Introduction

The performance achieved by a parallel application is the result of complex interactions between the hardware and software resources of the system where the application is being executed. The characteristics of the application, e.g., algorithmic structure, input parameters, problem size, influence these interactions by determining how the application exploits the available resources and the allocated processors. In this framework, tuning and debugging the performance of parallel applications become challenging issues [16].

A typical approach to address these issues is experimental, that is, based on instrumenting the application, monitoring its execution and analyzing its performance either on the fly or post mortem. Many tools have been developed for this purpose (see e.g., [1], [2], [7], [17],

[18]). These tools analyze the measurements collected at run-time and provide statistics and diagrams describing the performance of the application and of its activities, e.g., computation, communication, I/O. The major drawback of these tools is that they fail to assist users in mastering the complexity inherent in this analysis.

To overcome this drawback, various methodological approaches have been proposed and tools have been developed out of these approaches with the aim of identifying performance bottlenecks, that is, the code regions, e.g., routines, loops, of the applications critical from the performance viewpoint. The Poirot project [8] proposed a tool architecture to automatic diagnose parallel applications using a heuristic classification scheme. The Paradyn Parallel Performance tool [12] dynamically instruments the applications to automate bottleneck detection at run-time. The Paradyn Performance Consultant starts a hierarchical search of the bottlenecks and refines this search by using stack sampling [14] and by pruning the search space considering the behavior of the application during previous runs [9]. The Kappa-Pi tool [3] deals with a post mortem automatic performance analysis of message passing applications based on PVM. The analysis of processor utilizations leads to the identification of performance bottlenecks classified by means of a rule based knowledge system. Aksum [4] automatically performs multiple runs of a parallel application and detects performance bottlenecks by comparing the performance achieved varying the problem size and the number of allocated processors.

In this paper, we address the analysis of the performance of parallel applications from a methodological viewpoint with the aim of identifying and localizing performance inefficiencies. We define new performance metrics and criteria that highlight the properties of the applications and the load imbalance and dissimilarities in the behavior of the allocated processors. These metrics rely on the measurements collected by monitoring at run-time the applications. The integration of this methodology into a performance tool will help users in interpreting the performance achieved by their applications.

The paper is organized as follows. Section 2 presents the methodology and introduces metrics and criteria for the evaluation of the overall behavior of a parallel application. Section 3 focuses on the behavior of the processors allocated to the application. Section 4 presents an application of the methodology on a few case studies. Finally, Section 5 summarizes the methodology and discusses its integration into a performance analysis tool.

2 Characterization of Performance Properties

Tuning and debugging the performance of a parallel application can be seen as an iterative process consisting of several steps, dealing with the identification and localization of inefficiencies, their repair and the verification and validation of the achieved performance. Our objective is to define performance metrics and criteria for explaining the properties and the behavior of an application by identifying and localizing its performance inefficiencies.

As already stated, these metrics rely on the performance measurements collected at runtime. Various types of parameters can be measured. They include timings parameters, such as, wall clock times, as well as counting parameters, such as, number of I/O operations, number of bytes read/written, number of memory accesses, number of cache misses, number of bytes sent/received. These parameters can be measured at different levels of granularity, that is, they can refer to the whole application, to its activities, e.g., computation, communication, memory accesses, I/O, or to its code regions, e.g., loops, routines, code statements.

Our methodology follows a top down approach which first focuses on the overall behavior of the application in terms of its activities. Then, the individual code regions of the application and the activities performed within each of them are considered.

Note that the granularity of the measurements determines the level of details of the metrics that can be obtained by applying our methodology. However, as the parameters to be measured are typically defined when the applications are instrumented and monitored, it is out of the scope of this work to address these issues.

In what follows, we assume that the measurements refer to an execution of a parallel application with P processors, where K different activities and N code regions have been monitored. Moreover, not to clutter the presentation, we will focus on timings parameters. The extension of the methodology to counting parameters is straightforward and will be discussed on one of the case studies (see Sect. 4.1). Table 1 summarizes the notations and definitions used in this paper.

Parameter	Description
P	number of allocated processors
K	number of activities
N	number of code regions
T	wall clock time of the application
T_j	wall clock time of activity j ($j = 1, 2, \ldots, K$)
t_i	wall clock time of code region i ($i = 1, 2, \ldots, N$)
t_{ij}	wall clock time of activity j in code region i ($i = 1, 2, \ldots, N$; $j = 1, 2, \ldots, K$)
t_{ijp}	wall clock time of processor p for activity j in code region i ($i = 1, 2, \ldots, N$; $j = 1, 2, \ldots, K$; $p = 1, 2, \ldots, P$)

Table 1: Notations and definitions.

A coarse grain characterization of a parallel application is based on the breakdown of its wall clock time T into the times T_j spent in the various activities. We define the activity with the maximum T_j as the dominant, that is, "heaviest", activity of the application.

Our next step is to describe each code region i by its wall clock time t_i. We then identify the dominant, i.e., "heaviest", code region of the application as being characterized by the maximum t_i. The analysis has then to be focused on the dominant code region and activity as they could be critical for the performance of the application. Hence, we consider the breakdown of each t_i into the times t_{ij} spent into the various activities. It might be difficult to understand which activity better explains the behavior and the performance of the application. We can identify the code region with the maximum time in the dominant activity of the application. Moreover, for each activity j we can identify the worst and the best code regions, that is, the code regions with the maximum and minimum t_{ij}, respectively. Clustering techniques help in summarizing and interpreting this information by identifying patterns or groups of code regions characterized by a similar behavior. Each code region i, described by its wall clock times t_{ij}, is then represented as a point in a K-dimensional space. Clustering partitions these points into groups of code regions with similar characteristics, among which the candidates for possible performance tuning can be identified.

3 Characterization of Processor Dissimilarities

The coarse grain characterization of the performance properties of parallel applications is followed by a fine grain characterization that focuses on the behavior of the processors with the objective of identifying the most imbalanced activity and code region.

Load balancing is an ideal condition for an application to achieve good performance by fully exploiting the benefits of parallel computing. Programming inefficiencies might lead to uneven work distribution among processors that, in turn, leads to poor performance because of loss of synchronization, dependencies, and resource contentions among the processors.

Our methodology focuses on identifying and localizing whether and where an application experienced poor performance because of load imbalance. For this purpose, we consider the dissimilarities in the behavior of the processors by analyzing the spread of the t_{ijp}'s, that is, the wall clock times spent by the various processors to perform activity j within code region i. In particular, our methodology is based on:

- the definition of metrics to detect and quantify dissimilarities;
- the definition of criteria to assess their severity.

We derive the metrics for evaluating the dissimilarities according to the majorization theory [11], [15], that provides a framework for measuring the spread of data sets. Such a theory is based on the definition of indices for partially ordering data sets according to the dissimilarities among their elements as to identify the data sets that are more spread out than the others. Dissimilarities can be measured by different indices of dispersion, such as, variance, coefficient of variation, Euclidean distance, mean absolute deviation, maximum, sum of the elements of the data sets. The choice of the most appropriate index of dispersion depends on the objective of the study and on the type of physical phenomenon to be analyzed. In studying processor dissimilarities, the index of dispersion has to measure the spread of the times spent by the processors to perform a given activity with respect to the perfectly balanced condition, where all processors spend exactly the same amount of time. The Euclidean distance between the wall clock time of each processor and the corresponding average is then well suited for our purpose.

Having defined the metrics for evaluating dissimilarities, we have to select criteria for their ranking such as to identify and localize the activity and code region characterized by the largest load imbalance. Possible criteria to assess the severity of the dissimilarities among processors are the maximum of the indices of dispersion, some percentiles of their distribution, or some predefined thresholds. The choice of the criteria depends on the level of details required by the analysis.

The study of the dissimilarities can be summarized by the following steps:

- standardization of the wall clock times;
- computation of the indices of dispersion;
- ranking of the indices of dispersion.

As the indices of dispersion have to provide a relative measure of the spread of the wall clock times, we first standardize the t_{ijp}'s with respect to the wall clock times spent by all processors to perform activity j in code region i. The standardized times $\tilde{t}_{ijp}$ are given by:

$$\tilde{t}_{ijp} = \frac{t_{ijp}}{\sum_{p=1}^{P} t_{ijp}} \quad .$$

To quantify the dissimilarities of the times spent by the various processors to perform activity j within code region i, we define the index of dispersion ID_{ij}, that is, the distance between the $\tilde{t}_{ijp}$'s and their average:

$$ID_{ij} \ = \ \sqrt{\sum_{p=1}^{P} (\tilde{t}_{ijp} - \tilde{t}_{ij})^2} \ .$$

We then focus on the dissimilarities of the various activities and code regions.

Let ID_A_j be the index of dispersion for activity j, that is, the measure of the load imbalance within the activity. This index is defined as the weighted average of the ID_{ij}'s with respect to the fraction of the wall clock time of activity j spent within code region i:

$$ID\_A_j \ = \ \sum_{i=1}^{N} \frac{t_{ij}}{T_j} \ ID_{ij} \ .$$

To provide a measure of the dissimilarities that takes into account the wall clock times of the activities, we define the scaled index of dispersion SID_A_j given by:

$$SID\_A_j = \frac{T_j}{\sum_{j=1}^{K} T_j} \ ID\_A_j \ .$$

A similar approach is adopted for characterizing the dissimilarities of the code regions. Let ID_C_i be the index of dispersion of code region i, that is, the measure of the load imbalance within the code region. This index is defined as the weighted average of the ID_{ij}'s with respect to the fraction of the wall clock time of code region i due to activity j:

$$ID\_C_i \;=\; \sum_{j=1}^{K} \frac{t_{ij}}{t_i}\, ID_{ij}\ .$$

The scaled index of dispersion SID_C_i, that takes into account the wall clock times of the code regions, is then given by:

$$SID\_C_i \;=\; \frac{t_i}{\sum_{i=1}^{N} t_i}\, ID\_C_i\ .$$

Once we have computed the indices of dispersion for the various activities and code regions, we rank them to assess the severity of the dissimilarities. The activity and the code region with the maximum of the SID_A_j and SID_C_i are the most suitable candidates for performance tuning.

4 Case studies

In this section, we discuss our methodology on three case studies dealing with the identification of the inefficiencies of two kernels from the NAS Parallel Benchmarks 2.3 suite [13] and of a computational fluid dynamic application [10].

These case studies illustrate the application of our methodology on programs with different characteristics and on measurements collected at different levels of granularity. Note that to derive preliminary insights into the behavior of the processors the case studies dealing with the two kernels rely on some visualization, whereas the analysis of the computational fluid dynamic application does not rely on any visualization because of its complex algorithmic structure.

4.1 Integer Sorting kernel

The first case study focuses on the NAS Integer Sorting (`IS`) kernel [13], a kernel that performs a distributed integer sorting of 8,388,608 elements (i.e., class A benchmark). The case study considers an execution of the kernel on 64 processors of an IBM NetFinity Linux cluster. The characterization relies on measurements collected by the MPICH profiling library [6], and referring to the eight code regions corresponding to the MPI communication statements used in the kernel.

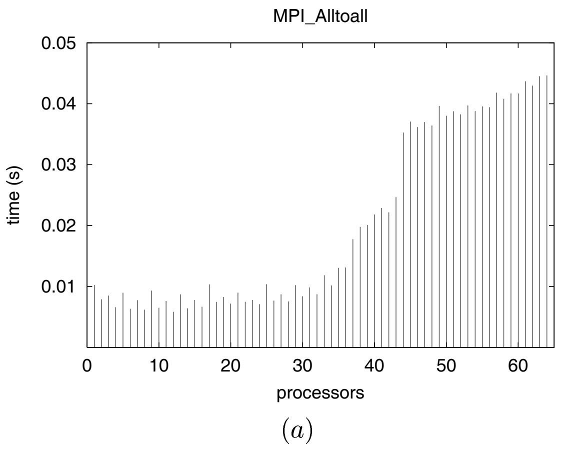
These statements refer to the collective communications (i.e., `MPI_Allreduce`, `MPI_Alltoall`, `MPI_Alltoallv`) used to distribute the elements to be sorted among the processors, and to the point-to-point communications (i.e., `MPI_IRecv`, `MPI_Recv`, `MPI_Send`, `MPI_Wait`), used to exchange elements, i.e., the local maxima, between neighboring processors. Moreover, the kernel uses the `MPI_Reduce` statement for timing purposes.

For each code region, measurements contain both timings and counting parameters. In particular, the parameters refer to the wall clock time and the occurrences of each code region, and to the number of bytes sent/received.

From the analysis of the wall clock times, we notice that the `MPI_Alltoallv` is the dominant code region, as it accounts for 0.175 seconds, that is, about 51% of the total wall clock time of the code regions.

To characterize the dissimilarities in the behavior of the 64 processors allocated to the kernel, we have first analyzed the wall clock times spent by the processors in the various code regions. Figure 1 plots the times spent by each processor in the `MPI_Alltoall` and `MPI_Allreduce` code regions. We notice that in the case of the `MPI_Alltoall` statement (Fig. 1(a)), the times of about one third of the processors are much longer, whereas we do not notice such a behavior in the case of the `MPI_Allreduce` statement (Fig. 1(*b*)). Moreover, the range of the times of the `MPI_Alltoall` statement is much smaller.

To derive a quantitative characterization of the dissimilarities experienced by each communication statement, we compute the corresponding indices of dispersion. Table 2 shows the indices of dispersion for the various communication statements. Note that the table reports only the communication statements whose indices of dispersion and wall clock times are non negligible.



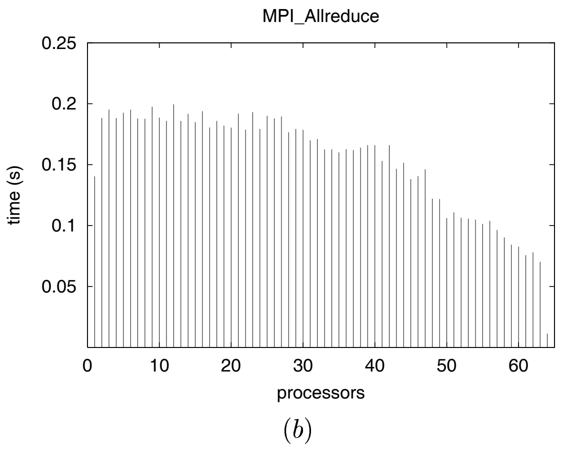


Figure 1: Times spent in the `MPI_Alltoall` (*a*) and the `MPI_Allreduce` (*b*) code regions.

We notice that the index of dispersion of the `MPI_Alltoall` statement is about three times larger than the index of dispersion of the `MPI_Allreduce`. Moreover, the `MPI_Allreduce` statement is characterized by the maximum scaled index of dispersion, that is equal to 0.0142307.

Code region	ID_C	SID_C
`MPI_Allreduce`	0.0338871	0.0142307
`MPI_Alltoall`	0.0894876	0.0052846
`MPI_Alltoallv`	0.0212838	0.0102718
`MPI_Reduce`	0.2758272	0.0025301

Table 2: Indices of dispersion of the code regions of the `IS` kernel.

The characterization of the kernel with the counting parameters shows that `MPI_Allreduce`, `MPI_Alltoall`, and `MPI_Reduce` statements are perfectly balanced, that is, each processor exchanges exactly the same amount of data with the same number of occurrences. The `MPI_Alltoallv` statement is the most imbalanced with respect to the volume of exchanged data. Hence, we can conclude that the `MPI_Alltoallv` statement, that is also the dominant code region, and the `MPI_Allreduce`, that is characterized by the maximum scaled index of dispersion, are good candidates for performance tuning.

4.2 Multigrid kernel

The second case study focuses on the NAS multigrid (`MG`) kernel [13], a kernel that solves the scalar discrete Poisson equation on a 256×256×256 grid (i.e., class A benchmark). This case study considers an execution of the kernel on 256 processors of an IBM NetFinity Linux cluster.

The measurements refer to the wall clock times of seven code regions of the kernel. These code regions correspond to the `interp`, `psinv`, `resid`, `rprj3`, `bubble`, `norm2u3`, and `zero3` routines that implement the computational core of the multigrid algorithm.

Table 3 reports the wall clock time, in seconds, of each code region. As can be seen from the table, the `resid` routine, that evaluates the residual error of approximate solutions, is the dominant code region and accounts for about the 33% of the total wall clock time of the code regions (that is, 1.90926 seconds).

`bubble`	`interp`	`norm2u3`	`psinv`	`resid`	`rprj3`	`zero3`
0.000067	0.210597	0.208532	0.546485	0.630935	0.296276	0.016373

Table 3: Wall clock time, in seconds, of each code region.

To derive preliminary insights into the behavior of the processors we have applied some visualization. Figure 2 shows a distribution of the processors as a function of their wall clock times. For each code region, the figure shows the fraction of processors whose wall clock time is in the lower 15% (light gray) or in the upper 15% (dark gray) interval of the range of the wall clock time. The fraction of processors whose wall clock time is outside these intervals is represented in white.

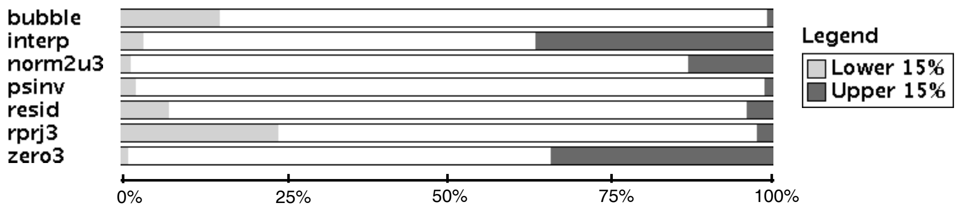


Figure 2: Distribution of the processors according to their wall clock times.

The figure shows that the behavior of the processors across the code regions is quite different. For example, in the case of the `rprj3` routine, about 24% of the processors is characterized by a wall clock time within the lower 15% interval and 2% only of the processors is characterized by a wall clock time within the upper 15% interval. In the case of the `interp` and `zero3` routines, more than 34% of the processors is characterized by a wall clock time within the upper 15% interval.

From the figure it is difficult to draw any conclusion about the processor dissimilarities. We can also look at the densities of the wall clock times of the processors. Figure 3 shows the densities for two code regions, namely, the `interp` routine (Fig. 3(a)) and the `rprj3` routine (Fig. 3(b)). As can be seen, the behavior of the processors is very different for the two routines although both perform a trilinear finite element projection.

To quantify the dissimilarities experienced by the processors in the various code regions we compute the indices of dispersion (see Table 4). From the table we notice that the spread of the times of the `interp` routine is about twice the spread of times of the `rprj3` routine. The

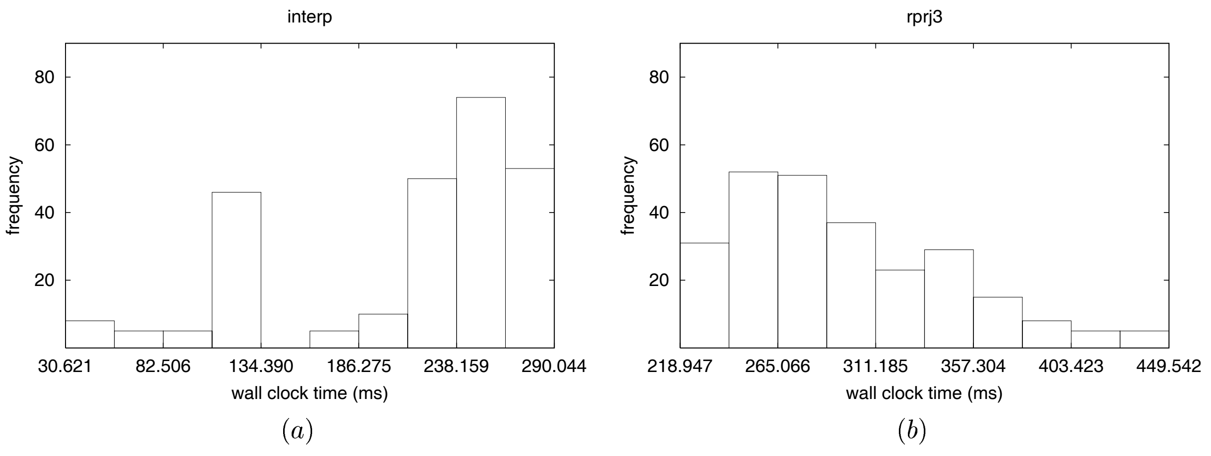


Figure 3: Density of the wall clock times of the processors.

Code region	ID_C	SID_C
`bubble`	0.0132119	0.0000005
`interp`	0.0200752	0.0022144
`norm2u3`	0.0040778	0.0004454
`psinv`	0.0056927	0.0016294
`resid`	0.0047494	0.0015695
`rprj3`	0.0111050	0.0017233
`zero3`	0.0062990	0.0000540

Table 4: Indices of dispersion of the seven code regions of the `MG` kernel.

indices of dispersion of these routines are equal to 0.0200752 and 0.0111050, respectively.

Table 4 also reports the scaled indices of dispersion used to assess the severity of the dissimilarities in the processor behavior of each code region. The `interp` routine, that interpolates the correction from the coarser grid to the actual approximation, is the code region with the maximum scaled index of dispersion, that is, with the most severe imbalance. As reported in the table, both its index of dispersion and its scaled counterpart are the maxima. Hence, the `interp` routine, although its wall clock time is shorter than the wall clock times of other routines, is a good candidate for performance tuning.

4.3 Fluid dynamic application

The third case study focuses on a message passing computational fluid dynamic application [10] that uses various numerical algorithms to solve the Navier-Stokes equations. The measurements refer to an execution of the application on 16 processors of an IBM Sp2 and are associated with nine code regions corresponding to the main routines and loops of the application. In particular, measurements refer to the wall clock times of these code regions and of the four activities performed within each of them, namely, computation, point-to-point communications, collective communications, and synchronizations. In what follows, for the sake of simplicity, the code regions are identified with a number, ranging from 1 to 9.

Table 5 presents the breakdown of the wall clock time of the application into the times spent into its four activities. As can be seen, the dominant activity, i.e., computation, accounts for approximately 59% of the wall clock time of the application, that is, 69.924 seconds.

computation	point-to-point	collective	synchronization
41.56	13.69	14.60	0.074

Table 5: Wall clock times, in seconds, of the activities performed by the application.

Table 6 presents the wall clock time of each code region with the breakdown into the times of its activities. We notice that the dominant code region, that is, code region 1, accounts for about 27% of the wall clock time of the application. This code region, that is the core of the application, is characterized by the longest time in the dominant activity of the application, i.e., computation, as well as in collective communications and synchronizations, whereas it does not perform any point-to-point communication. The code region which spends the longest time in point-to-point communications is code region 3. Moreover, only three code regions perform

Code region	wall clock time	wall clock time breakdown			
		computation	point-to-point	collective	synchronization
1	19.051	12.24	-	6.75	0.061
2	14.22	7.90	-	6.32	-
3	10.90	5.22	5.68	-	-
4	10.54	8.03	2.51	-	-
5	9.041	7.53	0.07	1.43	0.011
6	3.38	-	3.38	-	-
7	1.79	-	1.72	0.07	-
8	0.692	0.36	0.33	-	0.002
9	0.31	0.28	-	0.03	-

Table 6: Wall clock times, in seconds, of the code regions and of their activities.

synchronizations.

The application of clustering techniques to the code regions, described by the wall clock times of their activities, yields a partition of two groups. The heaviest code regions, that is, code regions 1 and 2, belong to one group, whereas the remaining code regions belong to the second group.

To characterize the imbalance of code regions and activities, we compute the indices of dispersion ID_{ij} (see Table 7). As can be seen, the behavior of the processors is highly imbalanced when performing synchronizations. The value of the index of dispersion corresponding to code region 5 is equal to 0.30571. Code region 1 is the most imbalanced with respect to the times spent by the processors for performing collective communications, whereas code region 8 is characterized by the largest indices of dispersion in two activities, namely, computation and point-to-point communications.

To measure the load imbalance of each code region, we compute the indices of dispersion ID_C_i and SID_C_i (see Table 8). We notice that code region 8 is characterized by the maximum index of dispersion, that is equal to 0.13720. However, as this code region accounts for a very

code region	computation	point-to-point	collective	synchronization
1	0.03674	-	0.06793	0.12870
2	0.01095	-	0.00318	-
3	0.00672	0.02833	-	-
4	0.01615	0.10742	-	-
5	0.00933	0.08872	0.04907	0.30571
6	-	0.00293	-	-
7	-	0.07002	0.01036	-
8	0.05017	0.23200	-	0.16163
9	0.00719	-	0.01138	-

Table 7: Indices of dispersion ID_{ij} of the activities performed by the code regions.

short wall clock (see Table 6), the corresponding scaled index of dispersion is equal to 0.00136 only. Code region 1 is a good candidate for performance tuning as it is the core of the application and it is also characterized by large values of the index of dispersion and its scaled counterpart.

Code region	ID_C	SID_C
1	0.04809	0.01310
2	0.00750	0.00153
3	0.01798	0.00280
4	0.03789	0.00571
5	0.01655	0.00214
6	0.00293	0.00014
7	0.06769	0.00173
8	0.13720	0.00136
9	0.00760	0.00003

Table 8: Indices of dispersion of the code regions.

Table 9 presents the indices of dispersion of the activities. As can be seen, the synchronization is characterized by the maximum index of dispersion, whereas its corresponding scaled index of dispersion is negligible. Hence, this activity is not a suitable candidate for tuning. The computation seems more suitable, as it is the dominant activity of the application and of

activity	ID_A	SID_A
computation	0.01904	0.01132
point-to-point	0.04701	0.00920
collective	0.03766	0.00786
synchronization	0.15559	0.00016

Table 9: Indices of dispersion of the activities.

the most imbalanced code region, i.e., code region 1.

5 Conclusions

Performance analysis of parallel applications is quite challenging. Many factors influence the performance and it is difficult to assess whether and where the applications have experienced poor performance.

The methodological approach presented in this paper is in the framework of automatic performance analysis of parallel applications and is aimed at the identification and localization of their performance inefficiencies. The methodology provides users with some guidelines for the interpretation of the performance achieved by their applications. We define metrics and criteria that characterize the performance of the applications. The metrics, derived as a result of the analysis of measurements collected at run–time, highlight the performance properties of the applications and the load imbalance and dissimilarities in the behavior of the allocated processors. The criteria are used to identify the activity and code region experiencing the most severe performance inefficiencies.

We are currently developing a prototype of a performance tool that computes the metrics and implements the criteria proposed in this paper. We believe that the integration of our methodology into a performance tool represents a good enhancement towards automatic performance analysis. Users expect from the tools answers to their performance problems and our methodology tries to provide a few of these answers.

As a future work, we also plan to assess the sensitivity of the metrics and criteria used for the identification of performance inefficiencies. For this purpose, we will analyze a large set of measurements collected on different parallel systems for a large variety of numerical and scientific applications [5].

References

[1] M. Calzarossa, L. Massari, A. Merlo, M. Pantano, and D. Tessera. Medea: A Tool for Workload Characterization of Parallel Systems. *IEEE Parallel and Distributed Technology*, 3(4):72–80, 1995.

[2] L. DeRose, Y. Zhang, and D.A. Reed. SvPablo: A Multi-Language Performance Analysis System. In R. Puigjaner, N. Savino, and B. Serra, editors, *Computer Performance Evaluation - Modelling Techniques and Tools*, volume 1469 of *Lecture Notes in Computer Science*, pages 352–355. Springer, 1998.

[3] A. Espinosa, T. Margalef, and E. Luque. Automatic Performance Evaluation of Parallel Programs. In *Proc. 6-th Euromicro Workshop on Parallel and Distributed Processing*, pages 43–49. IEEE Press, 1998.

[4] T. Fahringer, M. Geissler, G. Madsen, H. Moritsch, and C. Seragiotto. On Using Aksum for Semi-Automatically Searching of Performance Problems in Parallel and Distributed Programs. In *Proc. 11-th Euromicro Workshop on Parallel and Distributed Processing*, pages 385–392. IEEE Press, 2003.

[5] K. Ferschweiler, S. Harrah, D. Keon, M. Calzarossa, D. Tessera, and C. Pancake. The Tracefile Testbed - A Community Repository for Identifying and Retrieving HPC Performance Data. In *Proc. 2002 International Conference on Parallel Processing*, pages 177–184. IEEE Press, 2002.

[6] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. High-performance, portable implementation of the MPI Message Passing Interface Standard. *Parallel Computing*, 22(6):789–828, 1996.

[7] M.T. Heath and J.A. Etheridge. Visualizing the Performance of Parallel Programs. *IEEE Software*, 8:29–39, 1991.

[8] B. Helm, A. Malony, and S. Fickas. Capturing and Automating Performance Diagnosis: the Poirot Approach. In *Proc. of the 1995 International Parallel Processing Symposium*, pages 606–613, 1995.

[9] K.L. Karavanic and B.P. Miller. Improving Online Performance Diagnosis by the Use of Historical Performance Data. In *Proc. SC'99*, 1999.

[10] A. Malagoli, A. Dubey, F. Cattaneo, and D. Levine. A Portable and Efficient Parallel Algorithm for Astrophysical Fluid Dynamics. In *Parallel Computational Fluid Dynamics*, pages 553–560. North–Holland, 1995.

[11] A.W. Marshall and I. Olkin. *Inequalities: Theory of Majorization and Its Applications.* Academic Press, 1979.

[12] B.P. Miller, M.D. Callaghan, J.M. Cargille, J.K.H. Hollingsworth, R.B. Irvin, K.L. Karavanic, K. Kunchithapadam, and T. Newhall. The Paradyn Parallel Measurement Performance Tool. *IEEE Computer*, 28(11):37–46, 1995.

[13] NAS Parallel Benchmarks. http://www.nas.nasa.gov/NAS/NPB/.

[14] P.C. Roth and B.P. Miller. Deep Start: A Hybrid Strategy for Automated Performance Problem Searches. In B. Monien and R. Feldman, editors, *Euro-Par 2002. Parallel Processing*, volume 2400 of *Lecture Notes in Computer Science*, pages 86–96. Springer, 2002.

[15] M. Shaked and J.G. Shanthikumar. *Stochastic Orders and Their Applications.* Academic Press, 1994.

[16] M.L. Simmons, A.H. Hayes, J.S. Brown, and D.A. Reed, editors. *Debugging and Performance Tuning for Parallel Computing Systems.* IEEE Computer Society, 1996.

[17] J.C. Yan and S.R. Sarukkai. Analyzing Parallel Program Performance Using Normalized Performance Indices and Trace Transformation Techniques. *Parallel Computing*, 22(9):1215–1237, 1996.

[18] O. Zaki, E. Lusk, W. Gropp, and D. Swider. Toward Scalable Performance Visualization with Jumpshot. *The International Journal of High Performance Computing Applications*, 13(2):277–288, 1999.