Performance Debugging of Parallel Programs[1]

M. Calzarossa, L. Massari, A. Merlo, D. Tessera
Dipartimento di Informatica e Sistemistica
Università di Pavia
I-27100 PAVIA

A. Malagoli
Department of Astronomy
University of Chicago
CHICAGO, IL 60637

ABSTRACT

The performance of programs executed on parallel systems is influenced by a large number of factors related to the match between system and program characteristics. Performance debugging helps in understanding the behavior of the programs. Possible bottlenecks and unbalanced work distributions among the processors are identified. The portions of the code which lead to poor performance can then be modified and optimized. Performance debugging techniques are applied in a case study, where a real application solving a turbulence flow problem is considered.

1 Introduction

Performance evaluation of a computer system is a basic component to be taken into account in many studies involving its design, configuration and tuning. In particular, the execution of programs on parallel environments reveals a non–deterministic behavior, in that it depends not only on its structure, but also on the interactions of various hardware and software components. The programming paradigm and language adopted, the code optimizations achieved by means of specialized compilers, the interconnecting topology of the system and the scheduling policies are a few examples of such components.
Two enhancements to the development of parallel codes have been introduced: parallel extensions of sequential programming languages (for example, Fortran 90 [ANSI91] and Vienna Fortran [CMZ91]) and programming environments for multiprocessor and distributed systems (see e.g., PARADE [PV94], PVM [BDGM91] and p4 [BL92]).
Parallel languages reflect the new programming paradigms adopted for such systems. For example, Vienna Fortran is based on the so called SPMD computational model, in which the data arrays are partitioned and mapped onto different processors which execute the same program

[1]This work was supported in part by the the Italian Research Council (C.N.R.) "Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo" under grants N. 93.01592.PF69 and N. 94.00409.CT12, by the Italian M.U.R.S.T. under the 40% Project and by a NASA Grand Challenge Grant under the HPCC program.

on different data sets. The programmer has to select the most appropriate data distribution. Based upon these languages, parallelizing compilers have been developed with the effort of performing code optimization (see e.g., [BCFH93], [CFZ93]).
Message–passing communication libraries, such as, PVM and p4, represent the second approach towards the development of parallel code. Basically, the programmers have to specify inter-processor communications in a machine–independent fashion, since the high–level commands for exploiting the exchange of data are directly mapped onto the corresponding low–level, machine–dependent system calls.

As a result of the complex interactions of all these components, the evaluation of the performance of parallel programs requires methods and tools that have to be part of an integrated environment which allows the programmer to observe, analyze, and visualize the behavior of his program.
Measurements are a widespread approach used in performance debugging activities, that is, when the program has to be tuned to a given architecture in order to optimize its performance [LSVC89], [SG93]. This is generally an iterative procedure, involving measurements as well as modification of the source code.
Measurements generated and collected by means of monitoring tools need to be analyzed through statistical techniques, and visualized into graphical forms to highlight significant performance indices. The observation of program behavior requires some sort of tracing, which captures the events occurred (e.g., wait for a message, beginning of a communication) during its execution. This tracing activity can generate vast amount of data, creating problems during the analysis and interpretation phases.
The manipulation of large traces and the creation of graphical representations require automatic support for their analysis and visualization (see [HE91], [LMF90], [LS92]).

In this paper, a performance debugging study of a real code executed in two parallel environments is presented. Starting from measurements obtained from its executions, a study of its performance has been carried out, for identifying possible problems in obtaining the expected performance indices.
The paper is organized as follows. Section 2 describes some general issues on the performance debugging methodologies. Section 3 presents a case study where a real application which solves a turbulent flow problem is considered. The performance issues related to its execution on two parallel systems are described with the aim of identifying possible tuning actions. Finally, in Section 4 future developments are outlined.

2 Performance Debugging Techniques

It is known that the performance of the programs processed by parallel systems is much more critical than by traditional ones. It is difficult to obtain good predictions of the performance of the programs prior to their execution because the architectural components of parallel systems may play some unpredictable roles. The identification of possible bottlenecks and of "unbalanced" work distributions among the processors is almost impossible before running the program.

Profiling of the source code on sequential systems provides preliminary insights into the composition of the program in terms of the relative weights of its components (e.g., subroutines, functions, procedures). Hints about possible parallelization strategies (e.g., optimal number of processors, data distributions) can be derived. Two major drawbacks are associated to profiling. Such technique is not always applicable in practice because of the prohibitive computation time and memory requirements of real applications which make impossible their execution on uniprocessor systems. In general, some rough information may be extrapolated by profiling small portions of the source code. Furthermore, there is a complete lack of any details concerning the communication and the synchronization patterns.

Hence, performance debugging techniques need to be applied for understanding and explaining the behavior of the program with the aim of improving its performance. Indeed, the execution times of the programs have to be optimized, since they are usually executed many times. Even small savings in their elapsed time, after a few runs aimed at measuring the performance, are very important. The inherent parallelism of the code has to be exploited and the delays due to message exchanges among the processors and synchronization constraints have to be reduced. A trade-off between computation and communication activities has to be reached.

Performance debugging requires various phases including a tracing of the source code and the analysis of a large amount of raw data produced at run-time by the monitoring tools. As already pointed out, such procedure is iterative in that "optimal" program performance can be achieved by successive modifications and refinements of the code itself. Figure 1 summarizes the various phases required by a performance debugging methodology.

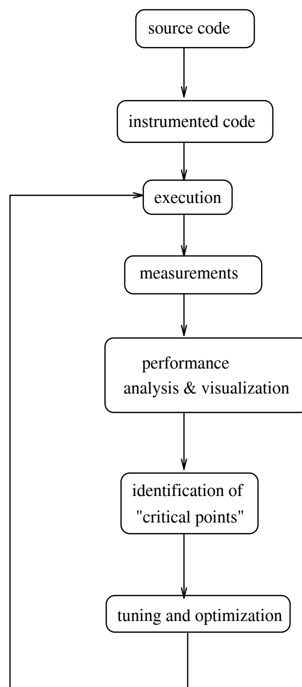

Figure 1: Phases of a performance debugging methodology.

Once the source code has been manually or automatically instrumented and the program has been executed on a parallel system, a post-processing of the collected measurements has to be applied. Such phase makes the data more manageable and easily understandable by the programmer who is the main responsible of the tuning actions aimed at improving program performance.

From the events stored during the program tracing, performance metrics and statistical indices are derived [CS93]. Execution, computation and communication profiles represent, as a function of the elapsed time, the number of processors which are simultaneously "active", computing or communicating, respectively. Such profiles are suitable for discovering phases in the program by showing how the execution evolves over time. For example, particular communication patterns and synchronization barriers which may be responsible of poor performance are easily identified by applying visualization techniques. Such techniques provide an intuitive description of the analyzed phenomenon. Then, in the case of SPMD programming paradigm, the most suitable data distribution can be derived. In the case of functional parallelism, an appropriate distribution of the program among the available processors is obtained.

Parameters, such as, speedup, efficiency, efficacy and processor working set [EZL89], [GST91], are computed when measurements of various runs of the program with different number of processors are available. Such parameters, which describe the behavior of the program with respect to its parallelism, provide information about the optimal number of processors to be used.

Statistical techniques can also be applied for the characterization of the measurements. Basic statistics, such as, means and standard deviations, help in discovering variabilities which may be a synonym of unbalanced conditions due to uneven work or data distributions.

Similarities in the measurements are discovered by applying clustering techniques. For example, a subdivision of the communication activities, e.g., send, receive, with respect to their parameters, e.g., number of bytes transferred, communication time, is very useful for identifying portions of the code characterized by "irregular" communication patterns where the tuning actions have to be focused. A better understanding of the program behavior is also achieved by applying, to the obtained clusters, a functional description with the aim of analyzing their composition with respect to the activities performed.

More detailed information are also obtained by subdividing and analyzing the communications according to their types, e.g., blocking versus non-blocking, send versus receive communications. Indeed, each type may exhibit different impact on the program performance.

An application of the various techniques of the performance debugging which allow the identification of portions of code where tuning and optimization activities have to be focused, is presented in the next section.

3 Case Study

In this section the methodology previously introduced is applied to a case study where a real code, solving a turbulent flow problem is executed, with different sizes, on two parallel systems. Before presenting the details of the case study and discussing the performance results

obtained, brief descriptions of the analyzed code of the programming environment and of the system architectures are provided.

3.1 Turbulent Flow Problem

The code analyzed, developed at the University of Chicago in collaboration with the Argonne National Laboratory, solves the turbulent convection and mixing in highly stratified, compressible plasmas. These kinds of study are the most computationally intensive in today's real life applications. Many efforts have been spent in order to exploit parallel computing resources on it.

The code under test uses finite difference methods and domain decomposition techniques for solving the flow model equations on massively parallel systems. Domain decomposition techniques are based on the idea of splitting the physical domain of the problem into smaller subdomains and to solve each subproblem on a processing node. This approach, together with the use of finite difference discretization methods, requires almost entirely nearest neighbourgh communications between processors. Therefore, it can be expected that under suitable circumstances good computation versus communication ratios, which are well suited for parallel systems, will be achieved.

The code solves the equations of compressible fluid dynamics with the inclusion of an external gravitational force g and a temperature dependent thermal conduction.

A mathematical model of the problem, which takes care of the conservation laws and the Fourier equation, is shown below:

$$\begin{aligned}
\partial_t \rho + \nabla \cdot \rho u &= 0 \\
\partial_t \rho u + \nabla \cdot \rho u u &= -\nabla P + g \rho z + Q_{visc} \\
\partial_t u + u \cdot \nabla T + (\gamma - 1) T \nabla \cdot u &= \frac{1}{\rho C_v} [\nabla \cdot (k(T) \nabla T)] + H_{visc}
\end{aligned}$$

where ρ, u and T are the density, the velocity and the temperature of the gas, respectively; γ is the ratio of specific heats and C_v is the heat capacity at constant volume; Q_{visc} and H_{visc} represent the rates of viscous momentum dissipation and viscous heating, respectively.

Mathematically speaking, the equations without the right–hand side of the energy equation form a system of hyperbolic equations, which can be solved with very robust algorithms based on higher–order Godunov methods. One such method, which is also used in our program, is the Piecewise Parabolic Method (PPM) of Colella and Woodward. The thermal conduction operator on the right-hand side of the last equation requires the solution of a nonlinear elliptic equation, for which a Crank–Nicholson implicit scheme and a multigrid method have been used.

The main phases identified in the program can be summarized as follows:

- initialization of the source term for the equation of heat transfer;

- solution of the Riemann shock tube problem and time advancement of the fluid variables with the higher-order Godunov method;

- solution of the non linear heat-conduction equation with the multigrid solver using the temperatures computed by the Godunov method;

- dumping of the physical variables of the model.

More details on physical and mathematical descriptions of the problem can be found in [DMCL94]. In what follows the various phases of the program are briefly presented with the objective of pointing out their performance aspects.

In general, finite difference operators can be thought of as having a stencil which determines the domain of dependency of one grid point from its nearest neighbors. The number of nearest neighboring points required depends on the details of the numerical scheme, but it is usually limited to a few. Boundary conditions are implemented by adding extra arrays of "ghost" points around the boundaries of the physical mesh. The mapping of a physical grid on more than one processor requires some form of domain decomposition, where each subdomain can be mapped on one processor. Each subdomain has its own ghost points, where the edge points from one subdomain are copied into the ghost points of the adjacent subdomain. Each processor takes care of a portion of the global domain and solves the equations only on it.

The PPM method used for the fluid motion requires the exchange of the neighbor points two times, one after sweeping the x direction and the other after sweeping the y direction on each solution step. In order to ensure the stability of the scheme a global maximum is also required to adjust the time step length according to the Courant Friedrichs Lewys condition.

The multigrid algorithm has two different sections. The first one, similar to the finite difference algorithm, uses a domain decomposition technique. For each relaxation step, at each level, the ghost points are updated. The second section begins when the grid size at each processor becomes too small. This means that the update of ghost points becomes more expensive than the computation. Then, a global collection of all the subdomains is performed, and a copy of the complete grid is provided to all the processors. If the number of processors is large enough, then the collected grid is likely to be too large to be solved exactly, and a sequential multigrid is started on each node until the lowest level is reached. Since this operation is performed redundantly (but concurrently) on each node, there is no further need to redistribute the solution back to the nodes.

The dumping of the physical variables is a very I/O intensive step which requires a global synchronization of all the processors in order to make a single ordered output stream by collecting the data scattered on each node. This phase requires big amount of storage space since at every solution step several megabytes of data need to be stored. For example, a problem of 1024×1024 dumps 18Mbytes of data at every step.

An example of the physical results obtained by executing our application is shown in Figure 2, where the dynamics of turbulent mixing on astrophysics on a 1024×512 problem are presented.

The figure shows the temperature fluctuations in a layer of unstable, turbulent gas (upper half) convecting onto a layer of stable gas (lower half). Thermal plumes inpinging onto the lower layer generate gravity waves, which cause the gas to mix also in the lower, convectively stable region. This process of convective penetration plays a fundamental role for the structure

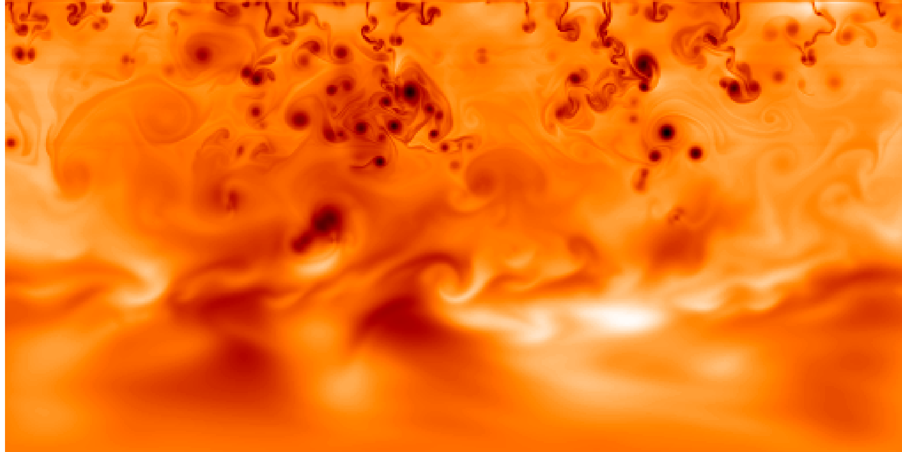

Figure 2: Temperature fluctuations on a 1024×512 problem

and evolution of stellar interiors.

3.2 Test Environment

The test program considered in our study is a Fortran code which employes the Chameleon library [GS93] for handling the communications among the processors. The dumping routines and the dynamic memory allocation functions are performed by C language procedures. The parallelism is exploited by means of a SPMD programming paradigm.

The choice of the Chameleon library is related to its portability and scalability features. It is known that message–passing represents a common framework for programming parallel systems. Unfortunately, even if a standard Message–Passing Interface (MPI) has been recently proposed, a number of different (and often incompatible) communication libraries are available, each with its own strengths and weaknesses.

Chameleon implements a collection of routines aimed at providing a consistent and easy–to–use communication paradigm and supports both native communication libraries (e.g., Intel NX and IBM EUI) and several highly portable packages, such as, PVM [BDGM91], PICL [Worl90], and p4 [BL92].

Chameleon, together with the BlockComm library [Grop94], ensures efficient data communications between processors. Various types of communications, such as, nonblocking and ordered blocking send and receive, can be selected within the BlockComm. Rather than viewing each communication among elements of a distributed data structure as a separate operation, BlockComm implements aggregate operations for dynamically managing the exchange of blocks of data between processors.

In order to aid the debugging and performance evaluation activities, Chameleon provides the possibility of collecting detailed information on the behavior of parallel programs by means of event monitoring. In our case, such feature has been used for tracing the program execution and obtaining measurements on two different multiprocessor architectures, namely, an IBM

Sp1 of Argonne National Laboratory and an Intel Paragon XP/S of IRISA, where our program has been tested.

The IBM Sp1 is a high–performance switch–connected cluster of RS/6000 workstations configured with 128 nodes and two compiler servers, used for dispatching the programs being executed. Each node has 128 Mbytes of main memory and 1 Gbyte local disk. Nodes exchange data by means of an Omega network.

The Intel Paragon XP/S is a distributed–memory, wormhole–routed, mesh–connected parallel system configured with 56 computing nodes. Each node is equipped with two Intel i860 processors, namely, a Message CoProcessor for communication handling and a Computing Processor, and 16Mbytes of main memory. Access to network interfaces and disks are transparently provided by dedicated I/O nodes.

3.3 Experimental Results

In order to understand how our program behaves and to discover the most computation intensive components of the code, a profiling of the program has been performed. The weight of each component (i.e., routine) has been obtained by executing the application on a sequential system.
Note that the code profiling depends on the problem size, since the program uses an iterative method, that is, the relaxed solver encapsulated in the multigrid code.

For the sake of simplicity, unless otherwise stated, we will present the results of our performance debugging study for a problem of size 512×256.

Figure 3 shows the results of our profiling activity. The percentage of the time spent by the program in each routine is plotted in Fig. 3 (a). Since performance debugging is aimed at helping the programmer in identifying the portions of code to be modified, the diagram presents only the routines which are part of it. Hence, system library functions, such as, `log`, `exp`, which account for about the 21% of the total time, have not been plotted. In the figure, the 17 "heaviest" routines, which contribute for about 67.4% of the total execution time, plus the remaining routines, denoted as "others", which accounts for about 11.6%, have been shown.

Another parameter derived by the profiling is the number of calls to each routine (see Fig. 3(b)). Such parameter together with the percentage of time consumed gives a more precise description of the weights of the various program components. As can be seen from Fig. 3, the "heaviest" routine is the `sweep`, which accounts for about 14% of the time. However, such routine has a call percentage of only 0.2%. Hence, the `sweep` routine is characterized by 527.3 ms/call. On the other side, the `interp` routine, which is characterized by a large number of calls, takes 1.81 ms/call. In performance debugging studies, a trade-off exists between all these performance measures.
In parallel environments there are different approaches to achieve better performance. Whenever possible, a divide and conquer methodology is applied to the components with the largest time per call. This is the case of our application. An alternative approach is based on the redistribution of the routine calls according to their number and their time per call.

After the sequential profiling, various experiments performed on the two parallel systems above described have been carried out. The program has been executed under different condi-

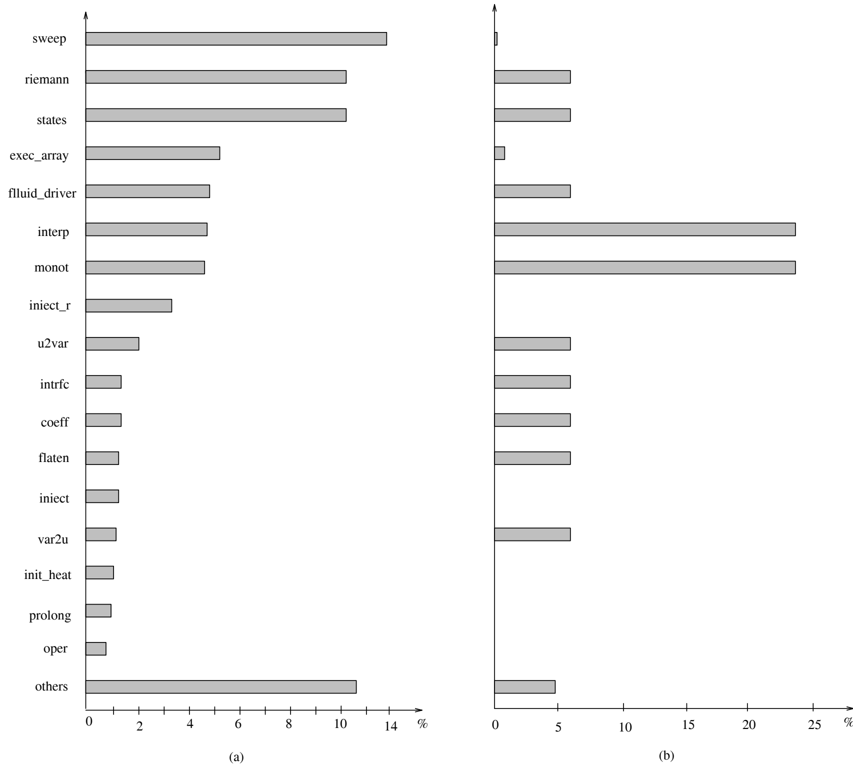

Figure 3: Percentages of the time spent by each component (a) and of the number of calls (b) obtained by profiling the program on a sequential system.

tions, namely, varying the number of allocated processors, the problem size, the communication types and the number of dumping phases. As already pointed out, the execution and communication activities have been monitored by means of the Chameleon facilities. The overhead due to the tracing activities accounts less than 1% of the program elapsed time.
Then, the collected measurements have been analyzed by MEDEA, a software tool developed at the University of Pavia for the analysis and visualization of the performance of parallel programs [CMMP94].

The objective of our study was to identify possible sources of poor performance within the code itself and as a function of the underlying architecture.

The problem has been executed on the Paragon with number of processors ranging from 1 to 32. Note that due to the domain decomposition technique adopted, such number has to be a power of two. The execution time of our problem is equal to 36 minutes on one processor. The speedup is linear up to 8 processors, then it goes down because the communication times become larger, while the computation times are reduced. Hence, a loss in performance results. Figure 4 shows the speedup curve for a 512×256 problem. As can be seen, there is a sharp decrease when the number of processors is increased from 8 to 16. Such phenomenon has less influence on the execution time of larger size problems. For example, the times of a 1024×1024 grid are 5 hours and 50 minutes and 3 hours and 57 minutes on 16 and 32 processors, respectively. Indeed,

in such cases, the subdomains solved by each processor are equal to 256×256 and 256×128, respectively.

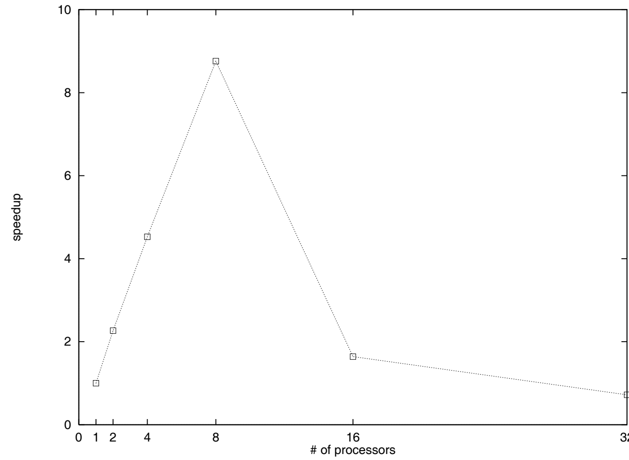

Figure 4: Speedup curve for a 512×256 problem executed with variable number of processors on the Paragon.

The same problem has also been executed on the Sp1. In such a case, due to the architectural characteristics of the system and of its interconnection network, a speedup decrease is noticed with a larger number of processors (i.e., 64). This is also because of the poor communication versus computation ratios.

Figures 5 and 6 represent the communication profiles for our problem executed on four processors of the Paragon, with blocking and non-blocking communications, respectively.

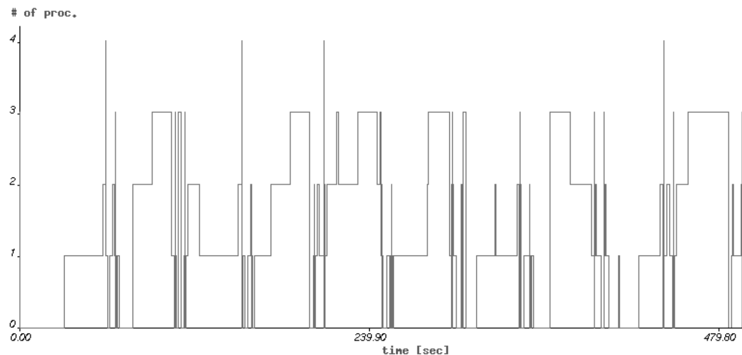

Figure 5: Communication profile of a 512×256 problem on a four processor partition of the Paragon with blocking communications.

As can be seen, there are communication patterns between the processors. Groups of communications repeated 10 times, one for each solution time step, are identified. It can also be noticed that there are a few asymmetries in the processor timings. It is likely to find three processors in a communication phase waiting for the fourth one. This phenomenon is better

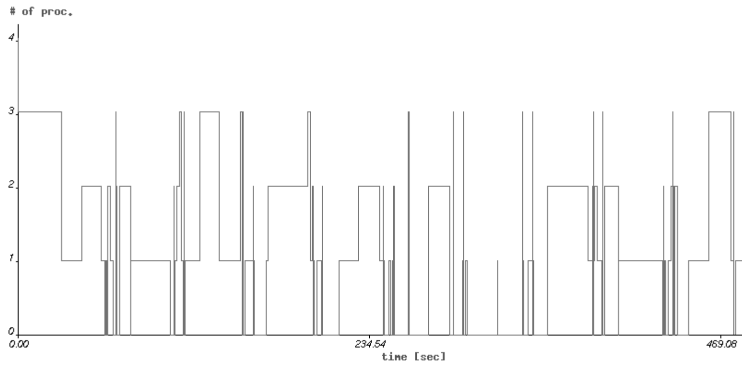

Figure 6: Communication profile of a 512×256 problem on a four processor partition of the Paragon with non-blocking communications.

displayed in Fig. 7 where a zooming of the diagram of Fig. 5 is shown.

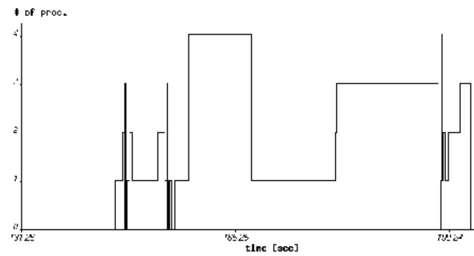

Figure 7: Zooming of the communication profile of Fig. 5.

A zoomimg over one phase of the communication profile obtained on the Sp1 is presented in Figure 8.

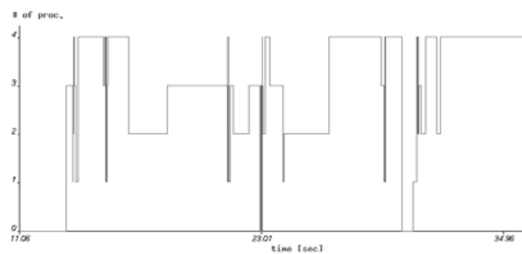

Figure 8: Zooming of one phase of the communication profile obtained on the Sp1.

After the visualization of the program performance, the communications activities have also been analyzed by means of statistical techniques in order to discover irregular patterns and similarities. Mean values together with standard deviations and distributions of the times spent by the various communication events have been computed. For example, the average and

the standard deviation are equal to 1.88 ms and 56.6 ms, respectively.
We then focused on on the activities performed by each processor. In particular, a communication event has been described by the time spent per processor, that is, as a point in a $p-$dimensional space, where p is the number of allocated processors. In Table 3.3 basic statistics obtained on a eight processor partition of the Paragon are presented. Unbalanced conditions have been found. Even and odd numbered processors have very similar behavior between each other. This is related to the concept of processor "leader" adopted in the code. There are processors which are also responsible of coordinating the activities of a few other processors.

Processor	Average	St. Dev.	Min	Max
1	0.378	5.675	0.015	465.160
2	2.293	61.171	0.015	5612.889
3	1.882	33.423	0.015	2111.845
4	2.492	60.789	0.015	5612.873
5	1.756	32.774	0.015	2111.911
6	2.366	60.439	0.015	5612.918
7	1.644	33.066	0.015	2111.901
8	2.262	60.601	0.016	5612.920

Table 1: Basic statistics of the communication times over four processors of the Paragon. The times are expressed in ms.

The correlations among the times spent by each processor have also been derived. There are no correlations between processor leaders, which are responsible of the coordination of a few other processors. Indeed, such an activity leads to unbalanced timings and delays in the program execution. High positive correlations are found between non-leader processors.

The clustering, applied to the 12606 communication events collected on eight processors and represented in a eight-dimensional space, yields a subdivision into two groups characterized by different parameters, i.e., communication times. The two groups are very unbalanced either with respect to the number of events and to their times. The first cluster contains the most part of the events which are very similar between each other. For example, the averages of the communication times for processors 5 and 6 are equal to 0.81 ms and 0.96 ms, respectively. In order to obtain a better understanding of the composition of this big group, cluster analysis has been applied once more on it. The new results enforce the previous ones. No better subdivisions have been obtained.

The second group is very small and contains just the 0.26% of the events whose times are two orders of magnitude larger than the global averages. Such events refer to particular communications where the processors have to wait for each other. The functional description of this cluster supports these conclusions. In Figure 9 the distribution of communication types belonging to this cluster is shown. As can be seen, there are only three different types, namely, Wait_Receive, Global_Min, Sync. Note that the Sync events are related to the dumping phase of the code.

The dumping activities have a large influence on the program performance. For example, for the 512$\times$256 problem, the global size of the dumping of the physical variables at each

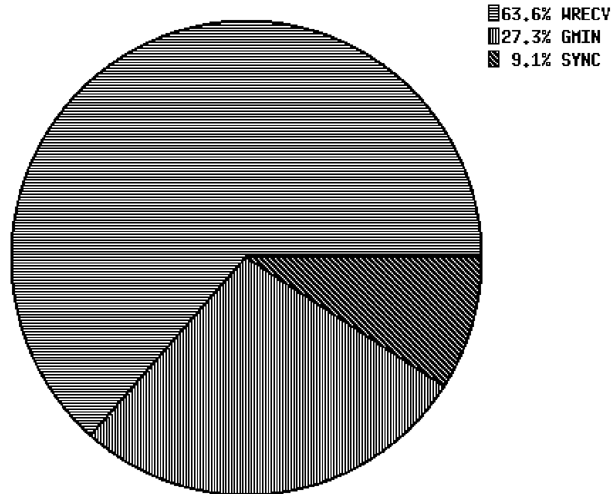

Figure 9: Functional description of the cluster analysis applied to measurements collected on eight processors with a non-blocking communication paradigm.

program step, that is, ten times, is about 25 Mbytes. The overhead due to such operations, e.g., I/O time plus time spent for contentions on system resources, accounts for more than 31% of the global elapsed time, that is, 98 seconds on an eight processor partition of the Paragon. Indeed, the transfer of large amounts of data is a critical factor for the interconnection network of the parallel systems, especially in the case of message passing architectures. This heavy data flow consumes resources on the intermediate processors because of the routing handling. Even in architectures with dedicated communication processors, such as the Intel Paragon, the I/O activities require the allocation and the use of communication buffers, which steal main memory to the computation processors. Furthermore, a slow down of the local communications between the processors may be introduced with consequent performance degradations due to synchronizations.
It is important to notice that dumping phases are an integral part of the analyzed application. They catch snapshots of each computation step and can be used as an archive as well as for restarting the computation at intermediate steps.

Acknowledgments

The authors would like to thank IAN-CNR for the use of a SUN SC-21002 where all the sequential profiling of the code has been performed. Special thanks go to IRISA (France) for making possible the experimental part of the paper by providing the access to the Paragon system. Authors gratefully acknowledge use of the Argonne High-Performance Computing Research Facility, funded principally by the U.S. Department of Energy Office of Scientific Computing.

References

[ANSI91] Fortran 90. X3j3 internal document s8.118, American National Standard Institute, May 1991.

[BCFH93] Z. Bozkus, Choudhary A., G. Fox, T. Haupt, and S. Ranka. A Compilation Approach for Fortran 90D/HPF Compilers. In Banerjee U., D. Gelernter, A. Nicolau, and D. Padua, editors, *Proc. of the 6th International Workshop on Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science, pages 200–215. Springer–Verlag, August 1993.

[BDGM91] A. Beguelin, J. Dongarra, G. A. Geist, R. Manchek, and V. Sunderam. A User's Guide to PVM: Parallel Virtual Machine. Technical Report ORNL/TM–11826, Oak Ridge National Laboratory, 1991.

[BL92] R. Butler and E. Lusk. User's Guide to the p4 Parallel Programming System. Technical Report ANL–92/17, Argonne National Laboratory, 1992.

[CFZ93] B.M. Chapman, T. Fahringer, and H. Zima. Automatic Support for Data Distribution on Distributed Memory Multiprocessor Systems. In *Proc. Sixth Annual Workshop on Language and Compilers for Parallel Computing*, Lecture Notes on Computer Systems, Portland, 1993. Springer Verlag.

[CMMP94] M. Calzarossa, L. Massari, A. Merlo, M. Pantano, and D. Tessera. MEDEA: A Tool for Workload Characterization of Parallel Systems. (in preparation), 1994.

[CMZ91] B. M. Chapman, P. Mehrotra, and H. P. Zima. Vienna Fortran – A Fortran Language Extension for Distributed Memory Multiprocessors. Compilers and Runtime Software for Scalable Multiprocessors. Elsevier, 1991.

[CS93] M. Calzarossa and G. Serazzi. Workload Characterization: A Survey. *Proc. of the IEEE*, 81(8):1136–1150, 1993.

[DMCL94] A. Dubey, A. Malagoli, F. Cattaneo, and Levine D. Portable and Efficient Parallel Algorithms for Compressible Hidrodynamics. (in preparation), 1994.

[EZL89] D.L. Eager, J. Zahorjan, and E.D. Lazowska. Speedup Versus Efficiency in Parallel Systems. *IEEE Trans. on Computers*, 38(3):408–423, 1989.

[Grop94] W. Gropp. BlockComm: Data Exchange on Parallel Computers. Argonne National Laboratory, June 1994.

[GS93] W. Gropp and B. Smith. User's manual for the chameleon programming tools. Technical report, Argonne National Laboratory, 1993.

[GST91] D. Ghosal, G. Serazzi, and S.K. Tripathi. The Processor Working Set and its Use in Scheduling Multiprocessor Systems. *IEEE Trans. on Software Engineering*, 17(5):443–453, 1991.

[HE91] M.T. Heath and J.A. Etheridge. Visualizing the Performance of Parallel Programs. *IEEE Software*, 8:29–39, 1991.

[LMF90] T.J. LeBlanc, J.M. Mellor-Crummey, and R.J. Fowler. Analyzing Parallel Program Executions Using Multiple Views. *Journal of Parallel and Distributed Computing*, 9(6):203–217, 1990.

[LSVC89] T. Lehr, Z. Segall, D.F. Vrsalovic, E. Caplan, A.L. Chung, and C.E. Fineman. Visualizing Performance Debugging. *IEEE Computer*, pages 38–51, October 1989.

[LS92] Lenzi P. and Serazzi G. PARMON: Parallel Monitor - Release 1.0. Technical Report CNR Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo n. 3/95, 1992.

[PV94] E. Pozzetti and V. Vetland. Parallel Programming with PARADE. In *Proc. Annual Conference AICA*, 1994.

[SG93] S. R. Sarukkai and D. Gannon. SIEVE: A Performance Debugging Environment for Parallel Programs. *Journal of Parallel and Distributed Computing*, 18:147–168, 1993.

[Worl90] P. H. Worley. A New PICL Trace File Format. Technical Report ORNL/TM–12125, Oak Ridge National Laboratory, 1992.