Performance Evaluation of Automatically Generated Data-Parallel Programs

L. Massari

DIS
Università di Pavia
via Ferrata 1
27100 Pavia, ITALIA

Y. Mahéo

IRISA
Campus de Beaulieu
Avenue du Général Leclerc
35042 Rennes Cedex, FRANCE

Abstract

In this paper, the problem of evaluating the performance of parallel programs generated by data-parallel compilers is studied. These compilers take as input an application written in a sequential language augmented with data distribution directives and produce a parallel version, based on the specified partitioning of data.

A methodology for evaluating the relationships existing among the program characteristics, the data distribution adopted, and the performance indices measured during the program execution is described. It consists of three phases: a "static" description of the program under study, a "dynamic" description, based on the measurement and the analysis of its execution on a real system, and the construction of a workload model, by using workload characterization techniques. Following such a methodology, decisions related to the selection of the data distribution to be adopted can be facilitated.

The approach is exposed through the use of the PANDORE *environment, designed for the execution of sequential programs on distributed memory parallel computers. It is composed of a compiler, a runtime system and tools for trace and profile generation. The results of an experiment explaining the methodology are presented.*

1 Introduction

Performance evaluation activities are required in many studies, involving the design and the tuning of a system, as well as the debugging of its performance. Whatever is the objective of the study, the workload, that is the set of programs submitted to the system, is one of the major components which determine system performance, in that strict relationships exist between the obtained performance and the hardware and software components of the system itself. Workload characterization is the basis of every performance evaluation study, in that it provides systematic methods for a quantitative description of the load which, together with the systems architectural aspects, is responsible of the performance.

In parallel environments, in particular, the performance evaluation process is more crucial and difficult than in sequential ones. The analysis and characterization of the workload in parallel systems require special care because of their complex architectures, the large number of hardware and software components interacting, and the new programming paradigm adopted.

In this paper, we study the problem of evaluating the performance of parallel programs generated by data-parallel compilers. Such compilers that takes as input an application written in a sequential language augmented with data distribution directives like HPF [9] and produce a parallel version, based on the specified partitioning of data (*e.g.* [12, 18, 1]).

The "parallelizing" compiler represents one of the most important system's software component influencing the program execution. However, even if the compiler manages in a transparent way code distribution, data distribution and communication among tasks, the parallelization decisions are definitively left to the user who specifies the partitioning of the data and their mapping onto the available processors. These factors highly influence the performance attained by the system.

The paper is organized as follows. In Sect. 2, the motivations for such a study and the problems related to the selection of the appropriate data distribution are presented. A methodology for the evaluation of the relationships existing between the workload characteristics and the obtained performance is proposed. In Sect. 3, the environment used for the automatic generation of data parallel programs is briefly described, together with the profiler tool providing measurements of the program execution. Finally, an experiment showing the feasibility of the approach is presented in Sect. 4.

2 Performance Evaluation of Data Parallel Programs

From the performance point of view, the major characteristic of a data parallel program is the dependence of the obtained performance from the specified data distribution, which influences computation, as well as communication activities. Optimal system performance is obtained by balancing the computation activity among different processors but also by minimizing communication traffic, and achieving a large degree of parallelism.

As already pointed out, the user is still responsible for decisions related to data distribution. Hence, due to the large number of distributions available and to the number of parameters affecting such decisions, an help in selecting the appropriate data decomposition which maximizes performance has emerged as an essential requirement. This help can benefit to the user, who has to choose the distribution, or it can serve the development of automatic data management in parallelizing compilers (see [6, 16]).

2.1 Methodology

Generally, depending on the objective of the study and on the availability of the system, different methods can be applied for performance evaluation [11]. They range from the construction of models having an analytical solution, to very detailed simulation models, to measurement based approaches. The major benefit in using performance modeling is when measurements are not available, for example, in performance prediction studies or during design activities. A number of studies have been done in the framework of data-parallel compilers to statically estimate the performance of the generated parallel programs [7, 2]. However, also in this case, an interaction with performance measurement activity is required in order to test if the model is realistic, or to verify the validity of the parameterization of the model and to drive its input.

When the system is available, a measurement-based study is generally carried out; measurements represent the basis for the workload characterization process (see [5, 8]). Following this approach, the performances of the real system are obtained: they contain also the influence of hardware and software components which would be otherwise difficult to capture even in a very detailed model. When dealing with programs generated by a parallelizing compiler, in particular, many are the decisions to be taken and the factors which interact and which make a measurement based approach particularly suitable.

Thus, the approach that we propose is based on various statistical analyses of the data measured during the program execution. The aim is to relate the program characteristics with that of the data partitioning and mapping, in order to optimize performance. Three phases have been identified. First, the program under study can be statically described by means of a few parameters. Then, a "dynamic" description, based on the measurement and the analysis of its execution on a real system is obtained. Finally, workload characterization techniques are applied to the obtained workload for constructing a model. In what follows, the three phases are described in more detail.

By statically analyzing the program structure, a workload description in terms of parameters that can affect the choice of the data distribution can be obtained. The attention has been focused on DO loops, and on array data structures, because the most part of scientific codes are based on loops working on data arrays, and the major part of the parallelism inherent in an application resides in them. Indeed, efficient code has to be generated especially for such structures; they form also the major source of performance degradation in numerical applications, due to communication.

Parameters which are strictly related to the loop structure, and others which characterize the whole program have been defined. As an example, the loop can be defined by the size of each of the referenced arrays, by the order of the index in the nested loops, and by the depth of the loop. The selected data distribution, which can be identified by the size of the blocks into which the data structures are partitioned, and the mapping strategy adopted are other parameters characterizing the program. Finally, the number of allocated processors can be chosen as a parameter reflecting the influence of the system hardware configuration.
Note that the parameter selection is a critical point, in that they have to accurately describe the program characteristics.

Once the program under study has been defined according to some of the previously described parameters, the second phase deals with the measurement and the analysis of its execution on a real architecture. In order to trace the program execution, appropriate monitoring instrumentation is required (see [17, 18]). Very low level data, related for example to the start and to the end of an event, are produced by monitoring tools and collected into trace files. The generally huge amount of data obtained makes them difficult to interpret. At this point, a pre-elaboration phase is needed in order to derive a more intuitive and compact description of the program behavior. Starting from measurements collected by the monitoring tools, a set of higher-level parameters are identified, describing the behavior of the program in terms of computation and communication activities. Such parameters deal, for example, with the number of messages exchanged, the execution, computation and communication times.

At the end of these two phases, the whole program execution can globally be represented by a set of n parameters, that is, those related to its "static" structure and to the system configuration, and those reflecting its "dynamic" behavior, that is, measured.

Each execution of the program is a *workload component*, which can be represented as a point in a n-dimensional space [8]. Then, the *workload* processed by the system is constituted by a collection of workload components, obtained varying the values of the parameters, for example, changing the dimension of the blocks into which data have been distributed, the mapping policy, or the problem dimension.

The aim and the core of the methodology, at this point, is to construct a model of the workload, that is, a compact representation able to capture and reproduce the behavior of the system. Workload characterization techniques have to be applied, so that "typical" behaviors of the program can be identified; multidimensional analysis techniques are required for this purpose. A description of the techniques applied for workload characterization is presented in the next section.

2.2 Workload Characterization

The characteristics of the workload have to be studied by applying different statistical analyses. A complete description of workload characterization techniques applied to performance evaluation studies of various system architectures, can be found in [5, 4].

The idea in our study is to find relationships among the program, the system configuration, and the achieved performance.
In order to reduce the number of parameters, and thus the complexity of the analysis, the correlation among the parameters can be analyzed. The correlation matrix helps in selecting the most appropriate parameters by expressing the dependencies among them: a correlation index between two parameters close to one reflects an equivalent behavior. Just one parameter amongst the highly correlated parameters can then be considered for the description of the application, hence reducing the dimension of the n-dimensional space.

Clustering analysis (see [10]) revealed very important for the identification of workload components having "similar" behavior, and is most commonly applied to the workload characterization problem.
The workload is considered as a set of points (components) in a space with a number of dimensions equal to the number of parameters used to describe each component.
Clustering algorithms partition the workload into clusters, such that components with similar characteristics belong to the same cluster. These algorithms have to identify the partition which better represents the characteristics of the original measurements. The goodness of a partition is given by an optimality measure, based on a selected metric, for example the euclidean distance.
Then, according to some specified criteria, a few components are extracted from every cluster and are considered as the representatives of the measured workload. The centroid, that is, the geometric center of the clusters, is generally chosen.

The partitioning of the workload means that "typical" behaviors can be identified among the various program executions. This gives the possibility to relate the static parameters describing the program and the system configuratio, with the dynamic ones, that is, with the performance indices. Centroids represent a model of the workload; hence, once the static configuration of the program has been chosen, the corresponding performance indices can be "expected".

Furthermore, clustering algorithms provide an analysis of the behavior of the components belonging to a cluster. Basic statistics, such as minimum, maximum, average values and standard deviation give a first idea of the "average" behavior of the program. For example, when considering parameters related to communication activities, a very high value of the standard deviation, compared with the mean value, is a synonym of variabilities. This means that unbalanced conditions due to uneven work or data distribution are detected.

In Sect. 4, the application of our methodology is

described through a test example.
In the following section, the environment used for parallelizing the program and for monitoring its execution is presented.

3 The Pandore Environment

Pandore is an environment designed for the execution of sequential programs on distributed memory parallel computers [1]. It is composed of a compiler, a runtime system and tools for trace and profile generation.

3.1 The Language

The Pandore language is based on a sequential imperative language using a subset of C (excluding pointers) as a basis. We have added a small set of simple and well-defined data distribution features in order to describe frequently used decompositions.

A Pandore program is a sequential program which calls distributed phases. The sequential part is in charge of all I/O operations and is executed on the host processor (if exists) or on one specific node of the distributed computer. Each distributed phase is spread over the processors of the target machine and is executed in parallel according to the owner-writes rule.

Distributed phases are declared like procedures preceded by the keyword `dist`. To each formal parameter of the distributed phase is assigned a distribution. The distributed parameter list allows the specification of the partitioning and the mapping of the data used in the distributed phase.

The array is the only data type that may be partitioned, scalars are replicated. The means to decompose an array is to split it into blocks. The specification of the partitioning for a d-dimensional array is given by the construct `block` $(t_1, ..., t_d)$ where t_i indicates the size of the blocks in the i^{th} dimension.

Then, the mapping of the blocks onto the architecture will be achieved in a regular or cyclic way according to the mapping parameters (`regular` or `wrapped`). In Pandore, we consider only one dimensional processor arrays whose size is not specified in the source code but used as a parameter by the compiler. As we allow the mapping of multidimensional decompositions, it is needed to indicate the order for the mapping of blocks by providing an ordered list of dimension numbers: for instance, `(1,0)` states for column first, `(0,1)` states for row first.

The last specification given in the parameter list concerns the transfer mode for values between the caller and the distributed phase: allowed modes are `IN`, `OUT` and `INOUT`. This specification is similar to the one found in Ada or Fortran90.

For example the formal parameter declaration

```
int A[N][N] by block(1,N) map wrapped(0,1) mode INOUT
```

states that array `A` is decomposed into `N` lines mapped cyclically onto the processors. The value of the elements of the array must be transferred from the host at the beginning of the distributed phase and must be sent back to it at the end of the phase.

3.2 The Compiler and the Runtime System

From the source program, the Pandore compiler automatically generates a machine independent SPMD code according to the owner-writes rule: a processor modifies only the variables that have been assigned to it by the distribution specification.

Two compilation schemes are embedded in the compiler. For reductions and parallel loops, the compiler applies an optimized scheme [14] performing loop bounds reduction and message vectorization, based on static domain analysis. For statements that cannot be optimized, the compiler relies on the well-known *runtime resolution* technique: masks and communication operations are introduced at the statement level to fetch distant data and select the processor responsible for the computation.

An original distributed array management based on paging [15] has been developed to support both schemes. Each block of a distributed array is decomposed into pages so that the array is represented on a processor by a table of pages that contains both *local pages* (pages of the blocks owned by the processor) and *distant pages* (copies of pages owned by other processors). Such a management leads to both efficient accesses and reasonable memory overhead.

The code generated by the compiler is a SPMD *C* code containing calls to the runtime system. The goal of the runtime system is to implement memory and process management, communication of data elements between processes, and distributed data accesses. It is build upon a virtual machine that permits the execution of Pandore programs on a wide range of parallel platforms.

3.3 The Pandore Profiler

The Pandore profiler allows the user to collect a number of quantitative measures on his program's execution with minimal intervention. It may be complemented by a trace generator for more qualitative measurements [3]. The use of profiling restrains the amount of storage needed; the number of counters to be updated is of the order of the number of variables declared in the source program. Sensors are inserted in modified versions of some runtime primitives, thus the compiler generates a similar code whether an instrumentation is demanded or not. An enhancement of mere profiling is actually used: in addition to their occurrences, the durations of events may also be cumulated [13]. Measurements are performed on each node and counters are brought back to the host at the end of the execution and then written down into a file that can be exploited by appropriate tools.

The links between the source and the evaluation results are established two different ways: first the user bounds fragments of the distributed phases he wants to be evaluated by defining some *instrumentation zones*, typically loop nests. Moreover, output figures are associated with objects of the source program such as arrays, scalars or conditional statements.

Twelve types of information are available that can be divided in two categories:

- *Information relative to each instrumentation zone*

  For each distributed variable v and each couple of processors (p_1, p_2), the number of messages from p_1 to p_2 required for the assignment of v is computed as well as the corresponding volume transferred and the cumulated waiting time on p_2. Identical information is collected for broadcast messages due to assignments to replicated scalars and due to the evaluation of conditional expressions.

  For the entire zone, the number of local accesses to a distributed array element and the number of accesses that required communication is computed for each processor. Identically, the number of purely local assignments and the number of assignments that required distant data is reported.

  The execution time for the zone is also given.

- *Information relative to each distributed phase*

  For each distributed variable, the time spent receiving the corresponding initial value from the host is reported for each processor. The waiting part of this time is also given.

  Different times are measured globally for the phase: the time for the triggering of the phase by the host, the time for transferring `IN` variables from the host, the time for executing the statements of the phase and the time for transferring `OUT` variables back to the host.

4 Experiments

In order to test the feasibility of the proposed methodology, a few experiments have been carried out based on data obtained using the profiler integrated into the PANDORE environment, described in the previous section. An iPSC/2 with 32 nodes has been chosen as target architecture. As a test program, the Jacobi algorithm has been considered (see Fig. 1).
Following the methodology presented in Sect. 2.1 raw data collected by the profiler have been analyzed, by applying various statistical techniques.

The algorithm has been statically described as follows: the size of the array (`Size`) has been chosen for the definition of the loop, while the number of allocated processors (`Npr`) and the number of blocks (`NBl`) into which the array are partitioned represent the data distribution policy.
Different runs of the algorithm have been executed, and monitored using the PANDORE profiler, obtaining 32 components for the workload. Each workload component has been obtained by executing the algorithm varying from 2 to 16 the number of allocated processors, the problem size, and the number of blocks allocated per processor.

At the beginning, 10 parameters have been extracted from the measurements for the description of the achieved performance. They are (see Sect. 3.3) the mean number of messages exchanged between a couple of processors (NM), the waiting time (Wait), the execution time for the zone (Tex), the time for the triggering of the phase (Trigg) by the host, the time for transferring variables from the host (Tin) and back to the host (Tout) and parameters related to the number of accesses (NRL, NRD, NAD, NAR).

```
#define N      256
#define P      4
 ...
dist jacobien(double B[N][N] by block(N/P,N)
                             map wrapped(0,1)
                             mode INOUT
          )
double A[N][N] by block(N/P,N) map wrapped(0,1);
{
 int i,j;
  for (i=1; i<(N-1); i++)
     for (j=1; j<(N-1); j++) {
        A[i][j] = ( B[i-1][j] + B[i+1][j] +
                    B[i][j-1] + B[i][j+1] ) +
                    V * B[i][j] ;
     }
}
```

Figure 1: The Jacobi algorithm considered.

The pre-analysis phase dealt with the definition of the relevant parameters able to capture the parallel characteristics of the load. Indeed, each component, has been represented in a 11-dimensional space.

The aim of the experiments was to analyze the average behavior of the program; the value of the parameters for each component have been computed by averaging the values on all the processors on which the data have been distributed.
Analyzing the correlation matrix (see Fig. 2), 4 highly correlated parameters have been discovered. In the further analysis, only 7 parameters, namely, the number of processors, the array size, the number of blocks per processor, the number of exchanged messages, the execution time, and the I/O times, have been considered (see Fig. 3).

Cluster analysis yield an optimal partition of the workload into 2 clusters, whose statistics are shown in Fig. 4. As can be seen, cluster 1 contains the major part of the workload components, and is characterized by low execution times and number of exchanged messages.
Looking at the composition of each cluster, we can see that, independently on the number of the allocated processors, cluster 2 groups executions with high problem size and high number of blocks per processor. Then, we can conclude that the performance obtained when executing the jacobi algorithm with high problem size and a partitioning of the data in small blocks are independent on the number of allocated processors.

5 Conclusions and Future Work

A methodology for evaluating the relationships existing among the program characteristics, the data distribution adopted and the measured performance indices has been presented. The results of an experiment in which the Jacobi algorithm has been analyzed, have been shown.

Following such methodology, decisions related to the selection of the data distribution to be adopted can be facilitated.

The methodology has to be tested on real application programs. Then, a more detailed description of the loop has to be studied, reflecting, for example, the dependences among the statements, can be useful for representing the influence that such factors have on the selected data distribution. If different experimentations are carried out while varying the data distribution, the data mapping and the problem description, groups of programs with similar performance behavior can be identified. Then, given the program description, a prediction of the performance can be obtained, based on the specified data distribution.

References

[1] F. André, M. Le Fur, Y. Mahéo, and J.-L. Pazat. The Pandore Data Parallel Compiler and its Portable Runtime. In *International Conference and Exhibition on High-Performance Computing and Networking, HPCN Europe'95*, number 919 in Lecture Notes in Computer Science, Milan, Italy, May 1995. Springer Verlag.

[2] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer. A Static Performance Estimator to Guide Data Partitioning Decisions. In *3rd ACM SYGPLAN Symposium on Principles and Practice of Parallel Programming*, Williamsburg, VA USA, June 1991.

[3] C. Bareau, Y. Mahéo, and J.-L. Pazat. Parallel Program Performance Debugging with the Pandore II Environment. *Parallel Computing*, September 1993.

[4] M. Calzarossa and L. Massari. Measurement-Based Approach to Workload Characterization. In G. Haring, R. Marie, and G. Kotsis, editors, *Performance and Reliability Evaluation. Tutorials Papers at the 7th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, OCG Schriftenreihe, pages 123–148. Oldenbourg Verlag, 1994.

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

[6] B. Chapman, T. Fahringer, and H.P. Zima. Automatic Support for Data Distribution on Distributed Memory Multiprocessor Systems. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Proc. Sixth International Workshop on Languages and Compilers for Parallel Computing*, number 768 in Lecture Notes in Computer Science, pages 184–199. Springer-Verlag, 1993.

[7] T. Fahringer and H. Zima. A Static Parameter Based Performance Prediction Tool for Parallel Programs. In *International Conference on Supercomputing*, Tokyo, Japan, July 1993. ACM Press.

[8] D. Ferrari, G. Serazzi, and A. Zeigner. *Measurement and Tuning of Computer Systems*. Prentice-Hall, Inc., 1983.

[9] High Performance Fortran Forum. High Performance Fortran Language Specification, Version 1.0. Technical report, Rice University, Huston, Texas, 1993.

[10] J.A. Hartigan. *Clustering Algorithms.* John Wiley & Sons, New York, 1975.

[11] P. Heidelberger and S.S. Lavenberg. Computer Performance Evaluation Methodology. *IEEE Trans. on Computers*, C-33(12):1195–1220, 1984.

[12] S. Hiranandani, K. Kennedy, and C-W. Tseng. Compiling Fortran D for MIMD Distributed-Memory Machines. *Communications of the ACM*, 35(8), August 1992.

[13] C. Kesselman. *Tools and Techniques for Performance Measurement and Performance Improvement in Parallel Programs.* PhD thesis, UCLA, July 1991.

[14] M. Le Fur, J.-L. Pazat, and F. André. An Array Partitioning Analysis for Parallel Loop Distribution. In *International Conference on Parallel Processing, Euro-Par'95*, Stockholm, Sweden, August 1995. Springer Verlag.

[15] Y. Mahéo and J.-L. Pazat. Distributed Array Management for HPF Compilers. In *High Performance Computing Symposium*, Montreal, Canada, July 1995.

[16] J. Ramanujam and P. Sadayappan. Compile-Time Techniques for Data Distribution in Distributed Memory Machines. *IEEE Trans. on Parallel and Distributed Systems*, 2(4):472–482, October 1991.

[17] P.H. Worley. A New PICL Trace File Format. Technical Report TM-12125, Oak Ridge National Laboratory, 1992.

[18] H. Zima and B. Chapman. Compiling for Distributed-Memory Systems. *Proc. of the IEEE*, 81(2):264–287, 1993.

	Npr	Size	NBl	NM	Wait	Tex	Tin	Tout	NRL	NRD	NAD
Npr	1.00	0.00	0.00	0.04	0.49	0.44	-0.59	0.45	-0.58	0.04	-0.58
Size	0.00	1.00	0.00	0.38	0.35	0.44	0.58	0.49	0.58	0.38	0.58
NBl	0.00	0.00	1.00	0.87	0.42	0.45	0.01	0.33	-0.01	0.87	0.00
NM	0.04	0.38	0.87	1.00	0.62	0.68	0.18	0.55	0.16	1.00	0.18
Wait	0.49	0.35	0.42	0.62	1.00	0.99	-0.21	0.83	-0.21	0.62	-0.20
Tex	0.44	0.44	0.45	0.68	0.99	1.00	-0.14	0.83	-0.14	0.68	-0.12
Tin	-0.59	0.58	0.01	0.18	-0.21	-0.14	1.00	-0.04	0.99	0.18	0.99
Tout	0.45	0.49	0.33	0.55	0.83	0.83	-0.04	1.00	-0.05	0.55	-0.04
NRL	-0.58	0.58	-0.01	0.16	-0.21	-0.14	0.99	-0.05	1.00	0.16	1.00
NRD	0.04	0.38	0.87	1.00	0.62	0.68	0.18	0.55	0.16	1.00	0.18
NAD	-0.58	0.58	0.00	0.18	-0.20	-0.12	0.99	-0.04	1.00	0.18	1.00

Figure 2: Correlation matrix.

Npr	Size	NBl	NM	Tex	Tin	Tout
16	128	1	236.25	2960.40	77.02	63.85
16	128	2	488.25	6961.27	95.80	2115.51
16	128	4	992.25	14693.36	42.19	2472.90
16	128	8	1984.50	19382.80	42.29	2162.21
16	256	1	476.25	13169.92	152.69	151.59
16	256	2	984.25	30699.33	180.96	7571.21
16	256	4	2000.25	64488.50	134.20	8231.70
16	256	8	4032.25	131149.64	137.57	8556.59
8	128	1	220.50	2024.08	81.45	148.48
8	128	2	472.50	4124.83	100.27	1212.01
8	128	4	976.50	8086.10	125.39	1475.52
.						
.						

Figure 3: Workload components.

1 th cluster of 2			n. of observations= 23.0000	
param.	center	min	max	st.dev
Npr	7.130	2.000	16.000	5.286
Size	166.957	128.000	256.000	60.220
NBl	3.043	1.000	8.000	2.549
NM	796.457	126.000	1984.500	608.624
Tex	7370.958	1411.711	19382.811	5579.477
Tin	257.511	42.192	946.695	248.937
Tout	1231.401	63.854	3622.539	872.672

2 th cluster of 2			n. of observations= 9.0000	
param.	center	min	max	st.dev
Npr	8.444	2.000	16.000	6.064
Size	256.000	256.000	256.000	0.000
NBl	5.556	2.000	8.000	2.404
NM	2712.861	984.250	4032.250	1208.181
Tex	44796.430	7039.291	131149.641	38947.316
Tin	441.284	134.206	964.395	328.209
Tout	4743.210	2250.381	8556.596	2654.015

Figure 4: Statistics of the two clusters obtained.