Parallel Program Performance Debugging with the Pandore II Environment

C. Bareau, Y. Mahéo, J.-L. Pazat[a]

[a] PAMPA Team, IRISA, 35042 RENNES Cedex, France (pandore@irisa.fr)

In this paper, we present the overall design of PANDORE II, an environment dedicated to the experimentation of distribution of sequential programs for their execution on distributed memory parallel architectures. The emphasis is then put on two performance analysis tools integrated in this environment.

1. INTRODUCTION

Programming distributed memory parallel computers (DMPCs for short) is a difficult task because of the management of parallel processes. The intricateness between computation and interprocess communication is often very tight, leading to very cumbersome programs. So the wish of most users is to be freed from all these "low level" details. In many cases they do not want to take into account the distributed aspects of their programs, which they also want to be machine independent.

A solution to this problem is given by the use of sequential programming languages extended with data distribution features like High Performance Fortran [7]. In this case, the compiler is in charge of generating communicating parallel processes from the sequential code and the data distribution specification. This solution has been validated [4], and implemented in the PANDORE II environment [1]. Yet, efficiency issues necessitate to develop specific performance analysis tools.

2. THE PANDORE II ENVIRONMENT

PANDORE II is an environment designed for parallel execution of imperative sequential programs on DMPCs. It comprises a compiler, libraries for different DMPCs and execution analysis tools including a profiler and a trace generator.

2.1. The source language

The source language for this environment is a subset of *C* augmented with features for data decomposition purpose – similar characteristics appear in the recently defined High Performance Fortran language. No specific knowledge of the target machine is required of the user: only the specification of data decomposition is left to his duty.

A PANDORE II program is a collection of *distributed phases* which may be seen like functions. The specification of data distribution is expressed as attributes of the formal parameters. Arrays can be partitioned into rectangular blocks and their mapping on processors can be regular or cyclic. For example :

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

specifies that the array `A` is to be partitioned into columns mapped cyclically onto the processors. The `INOUT` mode specifies that `A` is to be read at the beginning of the phase and written back at the end. Figure 1 shows an example of a PANDORE II program.

```
#define  N   128
#define  P   4
float A[N][N], float V[N];

dist myphase( float A[N][N] by block(N/P,N) map regular(0,1) mode INOUT,
              float V[N] by block(N/P) map regular(0) mode INOUT)
{ int i,j;
  for (i=0; i<N; i++)                        /*                        */
     for (j=0; j<N; j++)                     /* Instrumentation zone 1 */
        V[i] = f(V[i],A[i][j]);              /* ______________________ */
  for (j=0; j<N; j++)                        /*                        */
     for (i=1; i<N-1; i++)                   /* Instrumentation zone 2 */
        A[i][j] = g(A[i+1][j], A[i-1][j]);   /*                        */
}
main()
{ myphase(A,V); }
```

Figure 1. Example of PANDORE II source program

2.2. The compiler

The PANDORE II compiler generates parallel processes according to the data decomposition specified by the programmer. The compilation scheme is based on the locality of writes, on the host/node and the SPMD model. The host process executes the code of the `main()` part of the program, whereas node processes execute the SPMD code produced by the compiler from the distributed phases. A node process is in charge of its local variables: it updates their values and sends them to other processes when needed, according to the original sequential program. For example the assignment $A[i] = B[i+1] + C[i]$ is translated into:

```
refresh({tmp1,tmp2}, {B[i+1],C[i]}, owner(A[i]))
exec(owner(A[i]), tmp1+tmp2)
free({tmp1,tmp2})
```

When executing the `refresh` macro, owners of $B[i+1]$ and $C[i]$ send their values to the owner of $A[i]$. This process receives the values into buffers *tmp1* and *tmp2*. The `exec` macro is a guarded command that insures that only the owner of $A[i]$ executes the statement $A[i]$ =*tmp1*+*tmp2*. If the assigned variable is replicated on all the processors (as scalars for example), distant values are broadcasted through the network.

This basic translation scheme is not very efficient but optimization techniques based on the same model exist, for example [2] carries out a static domain analysis of loops to generate efficient code.

2.3. The runtime library

The runtime permits the execution of object code on different DMPCs. Its goal is to implement memory and process management, communication of data elements between processes, distributed data management and efficient index translation. It relies on machine dependent libraries and is implemented using macro definitions.

The machine model is a fully connected network of processing elements, communicating through reliable FIFO channels. Sends are non-blocking, whereas receives are blocking. The design and the implementation of this library increase the system portability and facilitates the instrumentation of the code.

2.4. Need for performance debugging

The performances of the code generated by the PANDORE II system are dependent on the appropriateness of the chosen data distribution to the algorithm but also on strategies directing the compiler and the runtime implementation. It appears that evaluating the influence of these parameters is necessary in order to guide the system designers and to provide the user with tools helping him to distribute his data. A dynamic evaluation – as opposed to a static estimation [3, 6] – offers the advantage of being applicable to every type of program and yields precise results.

Two techniques are used for performance measurement in the PANDORE II environment: tracing and profiling. These two techniques differ in their aim and in their implementation. Tracing permits the recording of events to which are assigned at least a type and a timestamp. With this method, the parallel activity can be recorded in order to rebuild the program behavior; we present here an extension of usual tracing for extracting information on all the potential behaviors of the program. The counterpart of this method is profiling, whose aim is to gather enough statistics for execution analysis. A number of counters related to events are updated during program execution. In PANDORE II, an enhancement of mere profiling is used: in addition to their occurrences, the durations of events may also be cumulated [9].

3. TRACE GENERATION AND ANALYSIS

3.1. Principle of the Pandore II trace analyzer

The principle of performance analysis by trace generation is to study the causal structure of the program that takes into account the load balancing induced by the chosen data distribution and the compilation scheme. Indeed, from an intuitive point of view, "a distributed program is very parallel when, most of the time, most processors can actually perform an action, i.e. are not blocked waiting for a message". Now these blockings, due to the asynchronism of the communications, happen when an event is causally dependent on an event on a different processor; thus, a tool that makes these dependences between processors clearly visible gives an estimation of their importance in terms of efficiency. Moreover, by focusing on events rather than on relations between them, one may get an estimation of the load balance of the program.

Our tool permits to visualize whether an action creates a bottleneck or can be performed in parallel with many other actions on other processors. For this purpose, we build the lattice of all the possible behaviors of the program: it is a graph in which each vertex represents an instant of the execution and the outgoing edges are the actions that may be performed at this instant by the unblocked processors. Hence each path from the initial vertex to the final one corresponds to one of the interleaving of the actions of the program.

For instance, from the following program, where x[i] and y[j] are placed on processor P1 and z[i] on processor P2,

```
Source code              execution on P1:                 on P2:

z[i]:=y[j]+1;            send(y[j],P2);          (a)      receive(y[j],P1);  (e)
x[i]:=5;                 x[i]:=5;                (b)      z[i]:=y[j]+1;      (f)
y[j]:=x[i]+3*z[i];       receive(z[i],P2);       (c)      send(z[i],P1);     (g)
z[i]:=z[i]-3;            y[j]:=x[i]+3*z[i];      (d)      z[i]:=z[i]-3;      (h)
```

we get, giving a direction to each processor, the lattice of executions of figure 2(a).

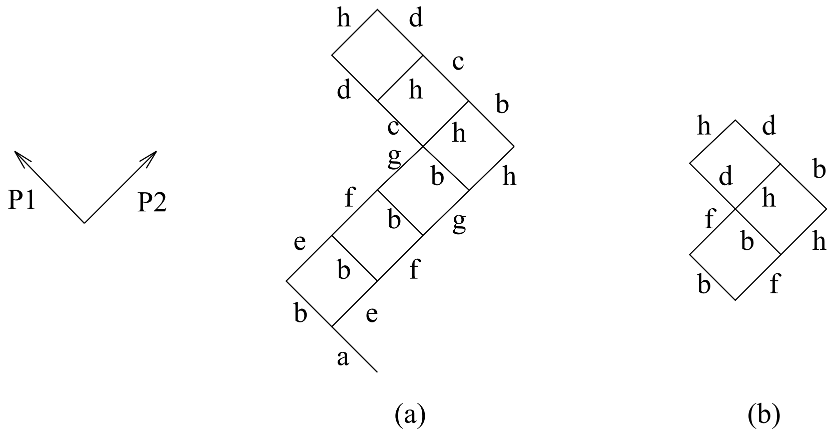

Figure 2. Lattices of executions

The main drawback of the lattice of executions is its size. Furthermore, events are not all interesting: for instance when an assignment always follows a reception, one of these two events could be abstracted without loss of precision. The user may also want to analyze only parts of its program. In that respect, we allow him to include his own observation points either in the PANDORE II source program or in the generated code. In the previous example, if we choose to observe only assignments, we obtain the new lattice shown in figure 2(b).

3.2. Construction

The construction of the lattice of executions is based on results of order theory, using the fact that a distributed execution can be seen as a partial order. An algorithm [10] that computes vector timestamps coding this order has been implemented in ECHIDNA [8], a programming environment for execution of Estelle specifications on DMPCs, networks of workstations, or by simulation on monoprocessors[1]. In order to use this tool, an Estelle-

[1] Estelle is an ISO language for protocol specification.

code generator has been added as new back-end to the PANDORE II environment. The execution times of the Estelle code are of course very different from those obtained with the C code, but the notion of intrusion is here irrelevant for we focus on the very structure of the algorithm, on the causal dependences between events that ensue only from the source, not from the low-level mechanisms.

The lattice of executions is actually the lattice of the ideals of an execution seen as a partial order, therefore it can be created from only one execution. We use an algorithm described in [5]; this algorithm is online: during an execution, the observed events generate traces with timestamps coding the partial order, and the lattice is incrementally constructed from these traces.

3.3. Applications

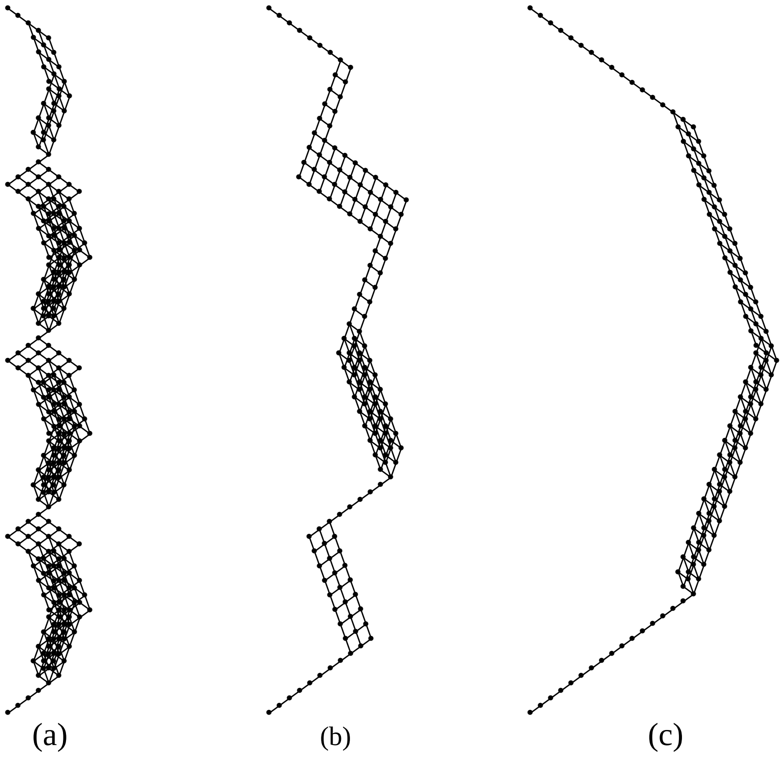

Figure 3. Matrix product lattices of executions

With our tool, the user is given an intuitive look over the performance of his distributed program. The lattice of executions highlights the bottlenecks as well as the "fully parallel" parts of the program, therefore it is possible to have an overall estimation of the performance, and thus to compare several data distributions. Furthermore, when a bottleneck appears in the lattice, the user immediately knows which events are causing it, and so which part of the program should be reconsidered.

As an example, we show in figure 3 the lattices corresponding to a matrix product on four processors with three different distributions. The observed events are the assignments of the program. It can be seen that in the distribution (c) there are many points with only one or two outgoing edges: parallelism is here very weak, especially at the beginning and at the end of the execution when only one processor achieves assignments, the others performing only communication actions. On the contrary, in (a), most points correspond to instants when three or four processors are "ready to work", leading to a greater amount of potential parallelism. This approach may seem unrealistic if we consider lattices with many processors; however we observe that such lattices present the same overall shape (for example the juxtaposition of the same pattern in distribution (a)), so the observations made with few processors are pertinent.

4. THE PANDORE II PROFILER

4.1. Instrumentation

The Pandore II profiler allows the user to collect a number of quantitative measures on his program's execution with minimal intervention. 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. This profiler has been implemented on a 32-node iPSC/2 but is easily portable. Sensors are inserted in modified versions of some runtime macros, thus the compiler generates a similar code whether an instrumentation is demanded or not. Lapses of time are measured with a software microsecond clock.

As most information for updating counters is available at compile time, the level of intrusion remains low (limited to a few percent execution overhead). 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 host code is not instrumented due to the lack of precision of time measurement on a time-shared multi-user system.

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 to objects of the source program such as arrays, scalars, conditional statements or loops.

4.2. Results

Besides execution times and the load balance, the main results produced by a profiled execution are related to communications and synchronizations. They may be classified in two categories: measures specific to distributed phases (communication with host at the beginning and at the end of each phase, phase triggering) and measures concerning assignments within instrumentation zones.

These statistics give information about the efficiency of the runtime implementation especially for message passing. Moreover, with the last class of results, data distribution for a given algorithm can be evaluated. An assignment of a distributed array element in which another distributed array reference appears in the right hand side may generate a message from the owner of the right hand side to the owner of the left hand side. The aim of the measurement is to globally build a directed graph where vertices are array partitions and arcs describe the traffic between partitions. Arcs are valued by the number of messages, the transferred volume or the waiting time on reception. For example, the assignment $A[3,5] = B[4]$ will increase the value of the arc $(B1 \rightarrow A2)$ if element $A[3,5]$ is on processor 2 and $B[4]$ on processor 1.

In a similar way, an assignment of a replicated variable in which a distributed array reference appears in the right hand side will generate a broadcast message from the owner of the array element. For the entire execution, broadcasting is described (number of messages, transferred volume, waiting time on reception) for each pair (*var,part*), where the partition *part* represents the source and *var* the assigned replicated variable. For example, the assignment $x = A[3,5]$ will increase the value of the counters related to the pair $(x, A1)$ if $A[3,5]$ is located on processor 1.

The produced results may be analyzed as such or treated by a specific tool that can give partial and abstracted views (e.g. by selecting or grouping processors or variables).

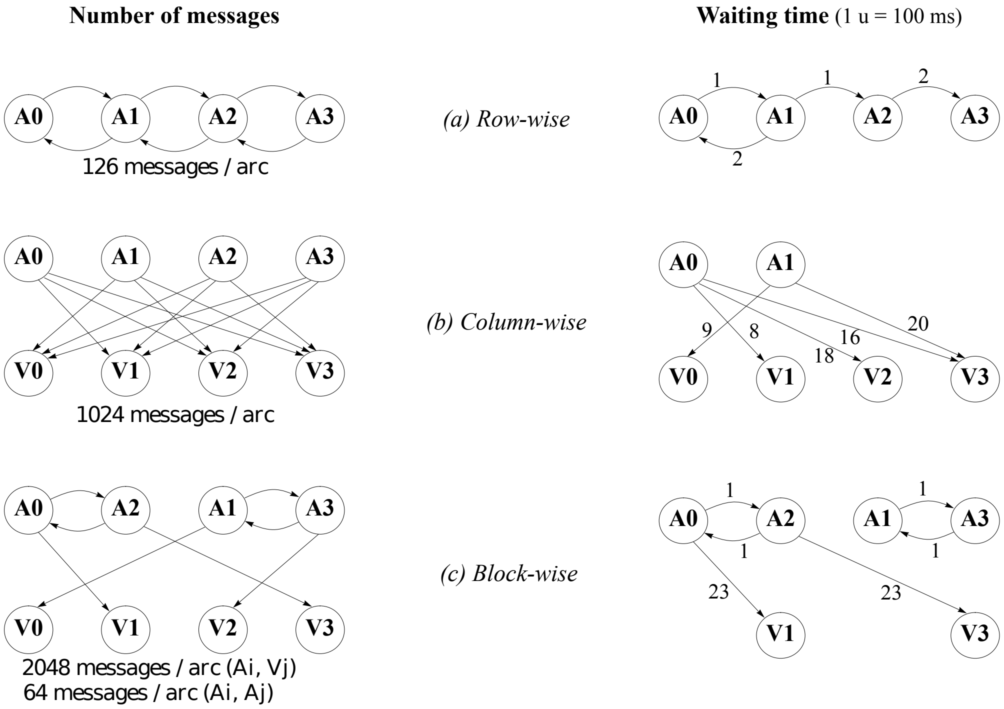

Figure 4. Communication graphs for 3 distributions

For illustrating the use of the profiler, let us consider the Pandore II program in figure 1 executed on 4 processors. After examining the first loop nest, vector V is decomposed into blocks of N/P elements and matrix A into groups of N/P rows (distribution *a*). Another choice would be to first look at the second loop nest. This would lead to the decomposition of A into groups of N/P columns (distribution *b*). One could also think to an intermediate solution: decomposition of A into $(2{*}N/P, 2{*}N/P)$ blocks (distribution *c*). Figure 4 gives the communication graphs for the three distributions. The row-wise distribution seems preferable. This is confirmed by the waiting time graphs which show strong synchronization for the column-wise and block-wise distributions.

5. CONCLUSION

The approach of distribution of sequential programs by data distribution is now recognized. We have presented PANDORE II, a complete environment for experimenting this method. As efficiency is a key issue, there is a great need for performance evaluation. However, because of the specificity of the codes generated by systems like PANDORE II, usual performance debugging tools are not well adapted. Therefore, we have designed new tools and integrated them in our environment. They are based on two complementary techniques of execution analysis (tracing and profiling) which permit qualitative and quantitative evaluation. They are aimed to help the user to distribute his program's data and to give information to the system designers. They have already been employed to improve the compiler and the runtime; nevertheless, experimentation must be pursued in order to tune these tools as well as the compilation and runtime techniques involved in the environment.

REFERENCES

1. F. André, O. Chéron, and J-L. Pazat. Compiling Sequential Programs for Distributed Memory Parallel Computers with Pandore II. In Jack J. Dongarra and Bernard Tourancheau, editors, *Environments and Tools for Parallel Scientific Computing*, Elsevier Science Publishers B.V., 1993.
2. F. André, M. Le Fur, and J-L. Pazat. *Static Data Domain Analysis for Compiling Nested Commutative Loops.* Technical Report to appear, IRISA, 1993.
3. V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer. A Static Performance Estimator to Guide Data Partitioning Decisions. In *The Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, June 1991.
4. C. Bareau, B. Caillaud, C. Jard, and R. Thoraval. Correctness of automated distribution of sequential programs. In *Proc. PARLE'93, LNCS 694*, Springer Verlag, June 1993.
5. C. Diehl, C. Jard, and J.X. Rampon. Reachability analysis on distributed executions. In *Proc. TAPSOFT'93, LNCS 668*, Springer–Verlag, April 1993.
6. T. Fahringer and H.P. Zima. *A Static Parameter based Performance Prediction Tool for Parallel Programs.* Technical Report APCP/TR 93-1, Austrian Center for Parallel Computation, University of Vienna, January 1993.
7. High Performance Fortran Forum. *High Performance Fortran Language Specification.* Technical Report Version 1.0, Rice University, May 1993.
8. C. Jard and J-M. Jézéquel. ECHIDNA, an Estelle-compiler to prototype protocols on distributed computers. *Concurrency Practice and Experience*, 4(5), August 1992.
9. C. Kesselman. *Tools and Techniques for Performance Measurment and Performance Improvement in Parallel Programs.* PhD thesis, UCLA, July 1991.
10. F. Mattern. Virtual time and global states of distributed systems. In Cosnard, Quinton, Raynal, and Robert, editors, *Proc. Int. Workshop on Parallel and Distributed Algorithms, Bonas, France, Oct. 1988*, North Holland, 1989.