ANDES: a Tool for Evaluating Parallel Systems

João Paulo Kitajima
Departamento de Ciência da Computação
Universidade de Brasília
Campus Universitário - Asa Norte
70910-900, Brasília - DF - Brazil
e-mail: kita@guarany.cpd.unb.br

Brigitte Plateau
LMC-IMAG-INPG
46, avenue Félix Viallet
38031 - Grenoble CEDEX 1 - France
e-mail: Brigitte.Plateau@imag.fr

Abstract

This paper presents the *ANDES* environment, a tool for supporting the performance evaluation of parallel systems. *ANDES* is structured as a modular tool. The parallel program, the parallel computer and the strategies used for implementing the software on the hardware are described separately. The interaction among modules occur when using *ANDES*. This environment is employed during the early phases of the software or hardware design. The evaluation technique currently adopted is based on the execution of synthetic workloads. However, *ANDES* can be used with other techniques that are compatible with the information described inside each module. A first prototype of the tool is running on a Transputer network. A more recent version runs on a computer network supporting PVM (Parallel Virtual Machine). *ANDES* was used to compare different mapping algorithms.

1 Introduction

It is already well established the importance of parallel processing in the search for computational power and also as a new way to solve problems. Parallel algorithms, parallel languages, parallel operating systems and parallel computers are available in order to make this parallel processing real. However it is necessary to constantly *evaluate* this new technology in order to improve it and to make it worthy.

This work concerns the *performance evaluation* of parallel systems. By "parallel systems", we mean a parallel computer executing a parallel program (i.e., a parallel *workload*). There are several methodologies and techniques for the performance analysis of such systems. There are also different moments in the life cycle of a parallel program development when performance evaluation is necessary. Therefore, we restrict the scope of our research.

First of all, it is intended to evaluate the performance of parallel programs during the design phase. In other words, we want to *predict* the performance of these programs, before coding them. We believe that the implementation of parallel programs (coding it, running it and obtaining results from it) is an expensive task. In this way, the detection of performance "bugs" should be avoided in the late phases of a parallel program life cycle. If the stress is "prediction" then we work rather with *models* of parallel programs and with *models* of parallel machines, as seen later.

Another characteristic of our work is the use of *synthetic workloads* as the technique for obtaining performance indices. A synthetic workload is also known as "exerciser" [8, 3]. It uses effectively a computer, but no real problem is solved. The advantage of this approach is the realistic execution environment. The synthetic workload executes on a *real* machine with a *real* operating system or runtime system on it. The overheads (e.g., scheduling, other users using the system) are real and not artificial or absent as in other techniques, like analytical modeling and simulation. In general, these overheads are difficult to model and at the same time they influence strongly the performance of the system. On the other hand, this approach has drawbacks. One of them is the need of a *real* parallel computer. During the 80's, the availability of a parallel machine was not trivial. However, with the growing of Internet and the development of standard communication libraries (e.g., PVM - Parallel Virtual Machine [9]), this problem has been minimized. Another drawback is the absence of control of the overhead. It is difficult, even impossible, for example, to increase the quantum of the multiprogramming or to reduce in a controlled fashion the interference of the operating system. There are also some restrictions imposed by the used parallel machine (the *target* computer). If there is only one target parallel computer, it can be difficult to evaluate the performance of a program on a parallel machine different from the target one. If we can evaluate different

parallel programs, we would like also to evaluate different parallel computers. To minimize this problem, *emulation* is done. To emulate a parallel computer or a parallel operating system corresponds to modify some characteristics of the parallel program, of the machine and of the operating system in order to mimic a different computer. A typical example is the change of the computation costs of a parallel program in order to emulate a parallel computer with a processor having a different processing power if compared with the processor of the target machine. This approach will be detailed later. Although the synthetic approach is used, we would like to support eventually other techniques for obtaining performance indices.

Another restriction in our work is the chosen paradigm of parallel processing. Parallelism can be expressed and implemented in different ways. The scope of this research is related mainly with message-passing parallel computers. In the same way we want to support different evaluation techniques, we intend to evaluate different types of parallel systems too. However, stress is done on message-passing programs running on distributed memory multiprocessors [5]. Nowadays, this is a strong trend towards research in parallelism.

Taking into account the considerations above, our goal is the development of a computer-aided environment for performance evaluation of parallel systems. This environment is called *ANDES* and it is based currently on the execution of synthetic parallel workloads. In the next section, we present the structure of *ANDES*. After the structure, we present the methodology to implement it. A current version of the environment is presented. We hope that this work should be useful. In order to achieve this, we use the environment to compare static mapping strategies. Some results are presented. Finally, we analyze the research effort and give perspectives to our work.

2 Global Structure of *ANDES*

The structure of *ANDES* is modular and it is based on *models* of parallel programs and *models* of parallel machines. Taking into account that we work at the prediction level, the parallel program is not yet implemented. The target parallel machine is real, but we have seen that eventually this parallel machine would emulate a different multiprocessor. Therefore, models are still important when describing the emulated parallel machine. Finally, there is a module where the implementation strategies are described. These strategies may be static like mapping or dynamic like load balancing.

2.1 Modeling the parallel program

The parallel program is modeled as an annotated DAG (Direct Acyclic Graph) called *DG-ANDES*. The nodes of this graph represent the *computations* to be done. An arc of the *DG-ANDES* represents the *precedence* between the nodes connected by it. Eventually this arc models a *communication* between these nodes. By the fact we want to employ the *DG-ANDES* graphs in a performance evaluation environment, it is necessary to quantify their needs in terms of computation, memory and communication resources. Therefore, we add *costs* to the nodes and to the arcs in order to quantify the needs of the parallel program. Computation and memory costs are associated to the nodes. Communication costs are associated to the arcs. The computation costs can be expressed in MIPS (millions of instructions per second) or MFLOPS (millions of floating point operations per second). Memory and communication costs are expressed in bytes. Considering that we are interested in evaluating message-passing programs, the communication costs are often associated to message sizes.

With a DAG structure and with costs, it is possible to model quantitatively deterministic parallel programs. However, there are nondeterministic programs too. These programs, for example, contain loops whose limits are not known before executing the program: these loops are data-dependent. Another example is the existence of decision structures like `if` and `switch` structures [4]. It is not possible to model exactly these situations. We cannot even know all the possibilities of data inputs because these possibilities can explode combinatorially. The solution adopted to this problem is to work with probabilities and with dependent costs. A cost can be a *constant*, if we know exactly the load, *random* in the case where the load can be modeled by a random variable and *dependent* when the cost depends on another cost or characteristic of the graph. For example, if we know the distribution of the limits of a loop, this computation structure can be modeled quantitatively using random variables. Often also, a task computation cost depends on the size of input messages. In this case, we can use dependent costs.

Another characteristic of the *DG-ANDES* is the possibility of modeling different relationships between connected nodes. As a matter of fact, a computation node is decomposed in three "logics": a *computation logic*, an *input logic* and an *output logic*. The computation logic has computation costs associated to it. The input and output logics are related with communications and the relationships between computation nodes. The input and output logics have descriptors which define a relationship pattern. A descriptor can be *simple*, *boolean* or *global*. A simple descriptor characterizes a computation node with one input and/or one output. A boolean descriptor characterizes a pattern of activation of inputs and outputs. For example, an and output means that all the successors of a computation node should be executed. On

the other hand, an `or` output models a decision. If there are only two `or` outputs, one models the "true" possibility and the other models the "false" possibility. By the fact we are working with models, exclusive probabilities (summing up to 1 or 100%) are associated to each output. A global input or output is associated to global communications like broadcast. A broadcast output works as a boolean `and` logic, but the passing data are the same for all outputs. Global communications often occur in parallel programs and we believe important to model them.

Finally, it is possible to build hierarchical *DG-ANDES* where a computation logic can be exploded in another *DG-ANDES*. This feature should be employed carefully in order to make compatible the inputs and outputs between different levels of the hierarchy. Hierarchical models can be useful when developing the model and when evaluating different granularities of the parallel program.

A *DG-ANDES* is described textually using the language *ANDES-C*. This language is indeed a library built on standard C [4]. This library supports the construction of annotated direct graphs through procedures. A new type, `Node`, is defined in order to model a computation node. A `Node` object is composed by some "attributes" like input, computation and output logics, costs and probabilities. The attributes can be accessed by procedures. The generated program (using *ANDES-C*) is then compiled and a workload is generated.

2.2 Modeling the parallel architecture

The parallel architecture model is the other module inside *ANDES*. This module allows the user to model quantitatively a parallel machine on which the modeled program should execute. The data contained in this module are related basically to the structure and behavior of the modeled parallel computer added with eventual overheads generated by an operating system executing on this architecture.

The structural data are:

- *for the processors*: number of processors;
- *for the communications*: topology of the communication network;
- *for the memory*: size of principal memory on each processor;
- *for the input/output devices*: number of devices.

The behavioral data are:

- *for the processors*: processing power in MIPS and MFLOPS, scheduling overhead;

- *for the communications*: link bandwidth and communication models (latency in function of the message size, distance and average charge of the communication media), routing strategy;
- *for the memory*: access time (write/read);
- *for the input/output devices*: input/output latencies in function of the distance, data size and charge.

These data are furnished to the target parallel machine (i.e. the available parallel machine). The target machine changes its own parameters in order to emulate a different parallel machine. This emulation is based mainly on modifying the basic parameters of the models. These changes should answer questions like "How much should the original message sizes be modified in the parallel program model in order to emulate a communication network twice faster than the communication network of the target machine"? Or "How should we change the dummy synthetic loops in order to emulate a faster or a slower machine"? This emulation is not trivial. It can be observed that changes can be done in the parallel application model, in the hardware of the target computer or in the target operating system in order to emulate the parallel computer described in this module.

2.3 The implementation strategies

The implementation strategies are related to all the necessary tasks, executed by the designer, by the operating system or even by the target parallel machine, to allow the implementation of a parallel program on a parallel computer. The typical strategies are those performing mapping or scheduling. Mapping is characterized by the choice of a processor to execute a task of the parallel program, according to a criterion. This criterion can be, for example, the execution time. In this way, the best mapping corresponds to the task allocation which gives the fastest execution time for the whole program. The scheduling is a similar problem, but more complex. It defines where (i.e., which processor - as in the mapping problem) and *when* the tasks should be executed on each processor. This temporal problem can be also viewed as defining an execution order of tasks on each processor. These problems can be solved *statically*, i.e., *before* the execution of the parallel program or *dynamically*, i.e., *during* the execution of the parallel program. The dynamic scheduling and mapping are also known generally as "load balancing" or "load sharing". The dynamic strategies are better in theory because they solve the allocation problem according to the actual charge of the system. Also, these strategies are better adapted to the nondeterminism of the applications. However, they are expensive in time and space and often they hide the benefits of the parallelism. More than expensive, these strategies are in

general very complex. On the other hand, the static strategies can be very fast to be executed, but they are less precise. Inside *ANDES*, these strategies are "black boxes" that can be changed. Only the interface should be defined between the mappers/schedulers and *ANDES*.

2.4 Putting all the modules together

How the modules described above interact with each other inside the environment *ANDES* ? The Figure 1 presents the modules and their interaction in order to generate a synthetic workload to be executed by a target parallel machine.

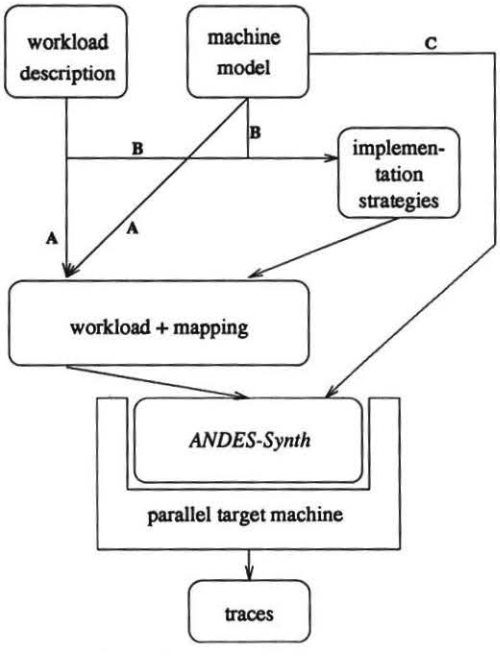

Figure 1: The structure of *ANDES*.

As we have seen, the quantitative model of the parallel program (i.e., the workload) is a C program that is compiled and executed. In this stage, some program costs can be modified in order to reflect a different architecture of the target machine. These changes are naturally based on the parameters

of the modeled parallel computer. This interaction is represented by the arcs **A**. The static implementation strategies need also information about the parallel program and the parallel machine. This fact is represented by the arcs **B**. The execution of the C program generates a synthetic workload which is mapped (or scheduled) on the modeled parallel machine. *ANDES-Synth* is the kernel program, executing on the target machine, that executes synthetically the workload. This execution can also be controlled in order to emulate a different machine parameter (arc **C**). The synthetic execution produces traces that can be analyzed and visualized for performance analysis.

3 Implementation of *ANDES-Synth*

The current implementation of *ANDES-Synth* executes on the target machine *Méganode* composed of 128 Transputers interconnected by a reconfigurable network [6]. A newer version is available on target machines using PVM (Parallel Virtual Machine), a standard and portable communication library. However, the description presented in this session is related to the Transputer version.

3.1 The general scheme

The parallel program models are described using *ANDES-C*. A benchmark of parallel program models exists in order to allow a systematic performance evaluation of classical parallel programs. The parallel machine model corresponds directly to the *Méganode* itself. In this first step of the research, the problem of parallel machine emulation is not tackled. The modeled and the target parallel computer are the same. The implementation strategies employed are the grouping and the mapping. Grouping is necessary in order to make the directed *DG-ANDES* compatible and the undirected models needed by the mapping strategies. Two grouping strategies are used: a manual grouping and an automatic one, done by PYRROS, a grouping program developed at Rutgers [10]. Static mapping is done using algorithms developed in the APACHE project in Grenoble, France [1]. The grouped workload (the `andes.data` file) is then given to the synthetic execution manager which executes the workload in a synthetic fashion. Traces are generated in an internal format in order to be used for performance evaluation.

3.2 The structure of *ANDES-Synth*

The kernel of *ANDES* is the synthetic execution manager that runs on the Transputer network. This manager is called *ANDES-Synth*. Roughly speaking, this kernel reads a file containing the *DG-ANDES* with the computation

and the communication costs plus a chosen map (the `andes.data` file), executes the synthetic program corresponding to the `andes.data` file, and stocks the performance data obtained during the synthetic execution.

3.2.1 The structure of the manager

The synthetic execution manager is a SPMD (*Single Program, Multiple Data* [5]) parallel program written in Inmos C using routing facilities provided by VCR (*Virtual Channel Router*, developed at the University of Southampton [2]). There are two types of processes (Figure 2): one executing on the root interface Transputer inside the host workstation (the `root` task) and one executing on each Transputer of the network (the `manager` tasks). These tasks (all the `manager` ones and the `root`) are virtually completely connected, that is, there is a virtual communication channel between any pair of processes.

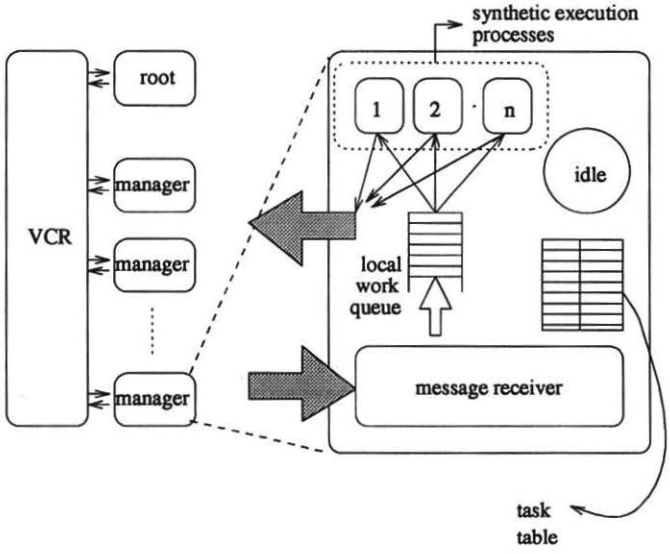

Figure 2: Process structure of the synthetic execution manager.

3.2.2 The root process

The root process reads the *DG-ANDES* plus the mapping information and sends, to each **manager** process on the network, the information of the nodes (tasks) of the DAG placed on the receiver **manager** process. In this way, each **manager** has a partial knowledge of the complete DAG. The **root** keeps the processor identification of the DAG tasks which do not have predecessors. After each network processor has received from the root processor all the information of the tasks placed on it, the **root** process starts the synthetic execution and measures time. This **root** process sends a starting signal to all Transputers containing DAG tasks with no predecessors. After this communication, the **root** waits for a terminal signal from all **manager** processes, stops measuring time and receives from all **manager** processes the charge information.

3.2.3 The manager process

The manager process is composed of two distinct parts. The first one receives, from the **root** process, the DAG information (computation and communication costs, number of predecessor tasks, number of successor tasks) and put them into a table (the *T-Table*). The second part consists in effectively managing the synthetic execution. It is only started if there are tasks placed on the associated processor. This part is the kernel of the environment and it is described in the following paragraphs.

Three types of (sub)processes are created just before managing synthetic execution (Figure 2):

1. a *message receiver process*: this process receives *all* the messages for the tasks residing on the processor. When a message arrives, this process verifies the identification of the receiver DAG task. The local DAG T-table is then consulted in order to check whether all the incoming messages for the receiver tasks have arrived. If not, the incoming message counter is decremented, and the process waits for a new message. If yes, a synthetic task execution is signaled (through a local work queue) to the *synthetic execution process*;

2. a set of *synthetic execution processes*: when a DAG task is to be executed, the message receiver process informs one of the synthetic execution processes that a synthetic DAG task can be executed. This last process loops for a specific amount of time defined by the application quantitative DAG and the out communications are done. Opposite to the reception of messages, there is not a process that manages the emission of messages: the synthetic execution process itself sends the appropriate messages to the successor DAG tasks. One important remark

is that the number of synthetic execution processes is specified by the user of the kernel. This parameter is known as the "multiprogramming degree", and the higher is this degree, the more exploited is the Transputer time-sharing capacity;

3. an *idle process*: this process runs only when no other process is running (including the VCR internal processes). The *idle process* only increments a counter. The final value of this counter is used to estimate the processor idle time.

The *message receiver process* and all the *synthetic execution processes* runs on high priority, like the internal VCR processes. The *idle process* is the only low priority process on a network Transputer. The *message receiver process* receives a message, looks up the concerned task information in the T-table (using direct access, no sequential nor binary search is done) and puts the task id in a local work queue in the case a synthetic task should be executed. It is expected that these activities do not take significant time of the processor. On the other hand, extreme care has to be taken when a synthetic task is executed in high priority. A synthetic task executing in high priority is *a priori* not interruptible. In order to give the processor to other processes, reschedule instructions are interleaved in the synthetic loop executed by the *synthetic execution processes*. In other words, at each x iterations of the synthetic loop (which mimics a true computation), a `ProcReschedule()` is executed. The current process is put at the end of the Transputer active processes queue. The x parameter can be modified, but a good value corresponds approximately to the slice provided by the CPU when executing low priority processes. In this way, a forced rescheduling of synthetic tasks are executed in order to enable message treatment by VCR and by the *message receiver process*.

The end of the `manager` process is achieved when no more DAG tasks are to be executed. All the (sub)processes finish and a signal is sent to the `root` process. Finally, the value of the counter incremented by the *idle process* is sent to the `root` process in order to be stored for post-mortem treatment.

4 Application of the Environment

ANDES is then used in the evaluation of task mapping strategies. In this experimental approach, the set of experiments described below has been performed. The starting point is a benchmark composed of 17 models of parallel algorithms (*DG-ANDES*). This benchmark is derived from different sources (the literature and real benchmarks). We hope that it is representative of scientific computing. The models are of (1) the Bellman-Ford iterative algorithm for computing the path length in a graph; (2)-(5) 4 systolic diamond-

shaped computations; (6) a divide-and-conquer; (7) one-dimensional FFT; (8) Gaussian elimination; (9) a generic iteration; (10) master-slave; (11) master-slave followed by Gaussian elimination; (12)-(13) two partial differential equation iterative algorithms; (14) a tree computation; (15) a quantum dynamics algorithm; (16) the recursive Strassen algorithm for matrix multiplication, and (17) the Warshall algorithm for finding the transitive closure of an adjacency matrix. For each program of this benchmark, some parameters are considered important for performance evaluation: the number of groups generated by PYRROS, the computation cost, the communication cost (and the ratio communication/computation), the number of inputs/outputs of a group, its communication regularity (i.e., the mean time interval between two external communications of a group) and its virtual parallelism (the width of the *DG-ANDES*). The target architecture is the *Méganode* configured as a 4×4 torus.

Four greedy and two iterative mapping algorithms are evaluated: modulo (i^{th} task on i *modulo* p processor, p is the number of available processors), LPTF (*Largest Processing Time First* [1]), LPTF with a quantitative criterion, LPTF with a structural criterion, one implementation of the simulated annealing algorithm (iterative) and one implementation of the tabu algorithm (iterative) [1]. The starting solution for the iterative algorithms is given by executing a greedy LPTF with a quantitative criterion. Four cost functions are defined:

1. SUM: it considers an additive cost function (non-overlap of computation and communication);
2. MAX: it considers a maximum cost function (overlap of computation and communication);
3. ROUT: it considers an additive cost function, but the communication costs are multiplied by the distance between the two placed tasks;
4. TOR: it considers an additive cost function, but the communication costs are obtained from a precise communication model of the target machine.

Finally, the following indices can be obtained when using *ANDES*: the value of the cost function of the final solution given by the mapping strategies, the execution time of the mapping strategy, the execution time of the synthetic program and the charge of each processor of the parallel machine. We are interested firstly in the execution time of the synthetic program and the related speedup (ratio between the total sum of the computation costs of the DAG tasks and the measured execution time). The speedup is used for comparing the different synthetic loads associated with the benchmark.

The quality of the cost functions is assessed through a linear regression between the cost function values (obtained when executing the mapping algorithms) and the real execution times. The best cost function is that whose obtained values are near of the real execution times. The Table 1 presents, for each cost function, the regression slope and the correlation coefficient. It is clear that the best cost function is TOR (slope close to 1).

	ADD	MAX	TOR	ROUT
slope	1,213	1,620	1,185	1,636
linear correlation coefficient	0,923	0,894	0,907	0,947

Table 1: Linear regressions for cost functions.

The practical (in opposition to the theoretical) evaluation of the mapping strategies is done through an experimental process. For each model of the benchmark (17 models), three different communication-computation ratios are considered (therefore 17 models $\times$ 3 ratios). Six mapping strategies are applied (4 greedy algorithms and 2 iterative methods). In this way, we have $17 \times 3 \times 6$ experimental values. Each one of these experimental values corresponds to the average execution time of a sample of 100 executions. With this sample size, the obtained averages are accurate.

From the experiences, we got some global conclusions:

- the best cost function is TOR which models the quantitative behavior of the VCR routing scheme;
- the communication-computation ratio has a strong influence on the speedup. This is verified through the obtained linear correlation coefficients among the different parameters of the benchmark models;
- the communication regularity has a weaker influence on the speedup but remarkable;
- the best mapping strategies are those which consider the communication and computation costs of the tasks (LPTF with a quantitative criterion and the two iterative algorithms). The improvement of the iterative algorithms on the greedy algorithms is not remarkable. We think that this is due to the regularity of the graph of groups. If the graph is regular, the pairwise exchange done by the iterative algorithms may not improve the cost function because the groups are equivalent;
- the models with a very high or a very low communication-computation ratio "smooth" the behavior of the mapping strategies. Too much communications imply a very loaded network. Even the most "intelligent"

algorithms are not able to manage the high congestion in the parallel system. On the other hand, too little communications imply that any load balancing strategy is good enough.

5 Conclusions and Perspectives

ANDES is a performance evaluation tool based on synthetic programs and was developed initially for support of the evaluation of different mapping strategies. The tool is being used intensively in order to acquire knowledge of the strategies behavior. The goal is to obtain some rules of thumb about the choice of the best mapping strategy given a specific parallel program and a specific parallel architecture. However, *ANDES* has been designed for a wider use. Other implementation and execution strategies can be evaluated like scheduling and load balancing, implying a change of the synthetic execution manager.

The choice of the synthetic approach was done in order to take into account the real overheads of the execution of a parallel program on a parallel machine. These overheads (for example, those associated with the communication system of a multiprocessor) are sometimes difficult to model when using analytical and simulation models. In this way, *ANDES* allows performance evaluation at the model level, but with some realistic (or almost realistic) components. This experimental approach is rather new, considering that normally mapping strategies are compared according to different values of the cost function [7].

Future work is planned. *ANDES* currently runs on a Transputer machine. It will be ported on the IBM SP-1 multiprocessor. With the SP-1 version, mapping, scheduling and load balancing strategies will be evaluated. A toolbox of the best strategies will compose the kernel of the parallel programming environment currently being developed inside the APACHE project. This environment is based on Athapascan, a programming language based on Remote Procedure Calls. Later, *ANDES* will be used inside this programming environment as a tool used for performance prediction. With the version on the SP-1, *ANDES* models will be described using C++ (instead of C, as done today). This language seems to be more adequate to model objects like tasks. C++ will also be used to describe machine models.

Acknowledgements

Méganode is a Telmat trademark. Transputer is an Inmos, Inc. trademark. The *ANDES* project is supported by the CNRS, INPG, PRC C^3, MESR, CNPq/Brazil and the Rhône-Alpes Region.

References

[1] Pascal Bouvry. *Placement de Tâches sur Machine Parallèle à Mémoire Distribuée.* PhD thesis, Institut National Polytechnique de Grenoble, 1994. Directeur de thése (advisor) : Denis Trystram.

[2] Mark Debbage, Mark B. Hill, and Denis A. Nicole. The Virtual Channel Router. *Transputer Communications*, 1(1):3–18, August 1993.

[3] Raj Jain. *The art of computer systems performance analysis: techniques for experimental design, measurement, simulation, and modeling.* Wiley Professional Computing. John Wiley and Sons, New York, 1991.

[4] Brian W. Kernighan and Dennis M. Ritchie. *The C programming language.* Prentice-Hall Software Series. Prentice-Hall, Englewood Cliffs, second edition, 1988.

[5] Ted G. Lewis and Hesham El-Rewini. *Introduction to Parallel Computing.* Prentice-Hall International, Englewood Cliffs, 1992.

[6] Denis A. Nicole. ESPRIT project 1085: reconfigurable Transputer processor architecture. In C. R. Jesshope and K. D. Reinartz, editors, *CONPAR88*, pages 81–89, Cambridge, 1989. British Computer Society, Cambridge University Press.

[7] Michael G. Norman and Peter Thanisch. Models of machines and computation for mapping in multicomputers. *ACM Computing Surveys*, 25(3):263–302, September 1993.

[8] D. A. Poplawski. Synthetic models of distributed memory parallel programs. Technical Report ORNL/TM – 11634, Oak Ridge National Laboratory – Martin Marietta, ORNL – Oak Ridge, Tennessee 37831 – USA, 1990.

[9] V. S. Sunderam, G. A. Geist, J. Dongarra, and R. Manchek. The PVM concurrent computing system: evolution, experiences, and trends. *Parallel Computing*, 20(4):531–546, April 1994.

[10] Tao Yang and Apostolos Gerasoulis. PYRROS: static scheduling and code generation for message passing multiprocessors. In *Proceedings of the 6th ACM International Conference on Supercomputing*, pages 428–437. ACM, July 1992.