ANDES: Evaluating Mapping Strategies with Synthetic Programs

João Paulo Kitajima and Brigitte Plateau
Laboratoire de Génie Informatique

Pascal Bouvry and Denis Trystram
Laboratoire de Modélisation et Calcul

Projet APACHE/IMAG-INPG
46, avenue Félix Viallet
38031 – Grenoble CEDEX 1 – France
(contact e-mail: `Kitajima@imag.fr`)

Abstract

This paper presents the *ANDES* performance evaluation tool. *ANDES* is based on the synthetic execution of parallel programs and it is used for the evaluation of mapping strategies. The Meganode, a distributed memory parallel computer, is considered as our target architecture. *ANDES* takes into account a benchmark of quantitative models of parallel algorithms and a set of mapping strategies (greedy and iterative algorithms are used). We show how this tool allows an extensive comparison of mapping strategies by using the benchmark, the mapping strategies and different cost functions.

1 Introduction

Distributed memory multiprocessors (DMM) are the current trend of high-performance parallel computers. They represent a good balance between cost and performance, mainly because of the connection of several commercial, general and relatively cheap microprocessors. A distributed memory multiprocessor is a computer composed of autonomous processors connected by a high speed communication network. Each processor has its own address space. The Intel Paragon, the Thinking Machines CM-5, the IBM SP-1 and the Cray T3D are typical examples of this generation of parallel computers. The different processing elements inside such a multiprocessor simultaneously execute different pieces of the parallel code, and each piece demands some machine resources, like the processor itself, the memory and the inter-processor communication media. The low level programming paradigm imposed by this kind of architecture is based on message passing, considering that no memory is physically shared by the processors.

Although more abstract programming models have been developed for DMMs (e.g., paradigms based on data parallelism, remote procedure calls, logical and functional programming, etc.), the programmer, the compiler or the operating system is always faced with the problem of choosing which processor should execute each of the different pieces of the compiled parallel program. In this allocation problem, known as *mapping*, the workload is represented by a quantitative and structural model of the parallel program to be executed. On the other side, the DMM is composed of a set of specific resources: processors with a certain processing power (e.g., expressed in MFLOPS or MIPS), distinct memories with a finite capacity and a communication network with a defined bandwidth. Finally, one or more performance criteria (like execution time or processor load) should be optimized.

The mapping problem takes into account exact or approximate models of the parallel program and of the parallel computer. These entities, mainly the DMMs, can however be very complex and their respective models can be somewhat far from the reality. Therefore, it would be interesting to compare the value of a performance criterion computed by the mapping algorithm against the same criterion measured from a true execution of the parallel program on the real DMM. It is desired, however, to stay in the domain of performance prediction, that is, evaluation should be done without executing the *real*

parallel program on the real parallel machine. It is well known that coding and debugging of parallel algorithms are a quite expensive task.

ANDES is a tool that supports performance evaluation of parallel programs at the prediction level, which considers the existing complex overheads of parallel computers. This is achieved through the use of *synthetic parallel executions* directly on the parallel machine. In a synthetic parallel execution, the resources of the DMM are used in a controlled way, but no code is generated. From a parallel program and a parallel machine model, all the steps from the model interpretation until the synthetic execution on the real multiprocessor and the computation of the performance indices, taking also into account the use of the mapping strategies, are automatically managed by *ANDES*.

The next section presents the mapping problem and some strategies used to solve it. Two aspects of *ANDES* are then explained: the parallel algorithm modelling language and the synthetic execution manager. A comparison of the mapping algorithms is presented in order to show that the tool is useful. Finally, some conclusions and perspectives are presented. *ANDES* is an evolution of the *ALPES* environment presented in [7], which was based on the generation of source files of synthetic programs. The new approach is based on a more efficient synthetic execution, controlled by a kernel that accepts a synthetic workload described in an intermediate format.

2 The mapping problem

The main goal of a good mapping is to minimize the execution time of the whole program. Other objectives can also be achieved, for instance in a multi-user processor network, it is interesting to minimize the average use of the different resources. The goal to be reached is often represented by a cost function.

Let us denote: T, the set of tasks

P, the set of processors

$calc(t)$, the total computing time of task t

$comm(t, t')$, the total communication time between t and t'

the function $alloc(t)$ returns the processor where task t is allocated

Formally defined, a mapping is an application from T to P which associates to each task a unique processor. The cardinality of all possible solutions is $|T|^{|P|}$. Even if this number can be slightly decreased due to some symmetry considerations, it remains too large for practical problems.

Several algorithms can be found in the literature for solving the mapping problem. We can roughly distinguish two classes of methods, namely, exact algorithms and heuristics [3]. Exact algorithms can only be used when the space of solutions is small enough, for instance when only a few tasks have to be allocated to a machine with a small number of processors. Exact algorithms give the optimal solutions but in practical cases they cannot be used because of the combinatorial explosion of the number of solutions.

The goal of heuristic algorithms is to give good solutions in relatively reasonable time. Two subclasses of heuristic algorithms have mainly been explored: *greedy* algorithms which progressively construct the solution and *iterative* algorithms whose principle is to improve an existing solution.

2.1 Greedy algorithms

In a greedy algorithm, the mapping is done without backtracking (a choice already made can never be reconsidered). The allocation of the i^{th} task is based on a criterion depending on the mapping of the first $(i-1)$ tasks. Two kinds of greedy algorithms can be envisaged for the mapping problem: the first ones are based on empirical methods and the second ones come from the relaxation of classical graph theory algorithms which are optimal for some restricted cases.

Examples of such criteria are given below:

- LPTF (Largest Processing Time First) is a heuristic whose criterion is restricted to load balancing. It is well-known that its performance in the worst case is about $\frac{4}{3}$ from the optimal when considering independent tasks [8].

- Lo presents in [11] an algorithm based on a maximal matching which minimizes the costs of

communications between tasks. This algorithm is optimal for UET (Unitary Execution Time) tasks if the number of tasks is less than twice the number of processors and if at most two tasks are allocated to one processor.

- Algorithms based on a minimal cut of bi-parted graphs can also be used [10] [14].

Greedy algorithms are easy to implement and have a polynomial complexity (often less than $O(|T|^3)$, for instance LPTF is of $|T| \log |T|$ complexity).

2.2 Iterative algorithms

All iterative algorithms start from an initial solution and try to improve it. Note that the initial solution can be found using a greedy algorithm. Usual iterative algorithms try to exchange tasks between processors to locally improve the solution. Most such algorithms use random perturbations to leave local minima and to jump to better solutions.

A well-known iterative algorithm is the Bokhari algorithm [1]. Its cost function (called *cardinality*) takes into account the number of tasks correctly mapped on the processor network and uses pair-wise exchanges of tasks to improve it. The basic hypothesis is that the number of tasks must be equal to the number of processors. Grouping methods must be used to take into account a greater number of tasks.

2.2.1 Hill climbing

The basic iterative algorithm, called *hill climbing* consists of starting from a given solution and to improve it iteratively using a neighborhood relation. This solution leads directly to a local optimum.

2.2.2 Simulated annealing

One of the most popular iterative methods is simulated annealing [2][13]. This method is based on an analogy with statistical physics: The annealing technique allows a metal with the most regular structure as possible to be obtained. It consists of heating the metal and reducing the heat slowly so that it keeps its equilibrium. When the temperature is low enough, the metal is in a equilibrium state corresponding to the minimal energy. At high temperature, there is a lot of thermic agitation which can

locally increase the energy of the system. This phenomenon occurs with a given probability decreasing with the temperature. It corresponds mathematically to giving a chance to leave a local minimum of the function to optimize.

2.2.3 Tabu search

Tabu search is an iterative meta-heuristic [5]. It tries to find the best neighbor of a given solution. To avoid cycling and local optima, a tabu list is established. This tabu list contains information concerning the last moves. A tabu move is not allowed except if an aspiration criterion is satisfied (i.e. if the proposed neighbor gives a better value of the objective function).

2.3 Mixed strategies

The two preceding classes of heuristic mapping algorithms are not the only ones but are the most present in the literature. Other solutions can be obtained by mixing the preceding algorithms: an initial solution could be obtained by simulated annealing and then tabu or algorithms like the branch&bound algorithms [6] could be used to improve the mapping.

2.4 Quality of the solution

Most solutions of the mapping problem are based on the optimization of cost functions. Let us denote by z such a function.

Under the previous assumptions in the model, we have two opposite criteria to take into account: the first one is to minimize inter-processor communications (1) and the second one to balance the computations between processors (2). These criteria are opposite in the sense that minimizing external communications leads to group all tasks on one processor and balancing the computing costs leads to distribute the tasks on all available processors (if $|T| > |P|$).

The cost functions given below correspond respectively to the two previous criteria and try to minimize the most loaded processor (in terms of communication, computation or both):

Minimum communications :

$$z = \max_{p \in P} \sum_{t|alloc(t)=p} \sum_{t'|alloc(t') \neq p} comm(t, t')$$

Load Imbalance :

$$z = \max_{p \in P} \sum_{t|alloc(t)=p} calc(t)$$

Trade-off :

$$z = \max_{p \in P} \sum_{t|alloc(t)=p} \left(calc(t) + \sum_{t'|alloc(t') \neq p} comm(t, t') \right)$$

This criterion that we have chosen to consider is a trade-off between both previous criteria.

In the cost functions previously described, no overlap between communications and computations was considered. If all communications can occur at the same time as computation (i.e. with the help of specialized processors), the cost function could be adapted by trying to find the maximum between the processor which communicates the more and the processor which computes the more, giving a second cost function. In fact, the cost function should be adapted following the characteristics of the target machine (for instance, distance between processors can be taken into account by analyzing routing tables, etc. . .).

3 Modelling parallel algorithms in *ANDES*

Parallel algorithms are modeled in *ANDES* as precedence valued DAGs. The graph vertices model computations and the arcs model the precedence and possibly a communication between computations. A numerical value is associated to the vertices in order to quantify a processing load (e.g., number of instructions to be executed). Also, a numerical value is associated to the arcs, in order to quantify a communication load (e.g., number of integers exchanged). These values are closely related to the

application costs. They can be converted to a normalized cost, for example to costs expressed in time units, if a parallel machine model is associated. A more detailed description of *ANDES* can be found in [7].

The DAG used in *ANDES* is not given "as is" (for example, as a communication matrix or as a formatted file). Indeed, it is extracted from a more abstract textual description which models a family of parallel algorithms. This abstract description, named *DG-ANDES*, is a C program using a specific library, which allows the representation of computation nodes and of precedence between these computation nodes. A computation node is more than a single task: it is composed of an input, of an output and of a computation description. These annotations allow the modelling of more complex relationships among the tasks. The concept of "computation" is variable. It may be a single arithmetic operation or a complex algorithm.

The basic type in a *DG-ANDES* is `Node`. This type is needed when declaring the computation nodes of a parallel algorithm model and it is used as a classical C data type. Two basic functions are used to build the graph: `comp_node` and `prec`. `comp_node` identifies the input, the computation and the output descriptions. For example,

```
gauss[0][1] = comp_node ("gauss task",
              input_desc, comp_desc, output_desc);
```

creates the computation node `gauss[0][1]` (of `Node` type), whose input, computation and output descriptions are `input_desc`, `comp_desc` and `output_desc`. These descriptions are classical C procedures. The string `"gauss task"` is a comment.

`prec` is the procedure which defines a precedence between two computation nodes. For example,

```
prec ("Precedence", gauss[0][1], 0, gauss[1][1], 3);
```

defines a precedence between `gauss[0][1]` (output port 0) and `gauss[1][1]` (input port 3). Input and output ports numbers identify specific input and output precedences of a computation node and they are important if a reference is needed. For example, it may be necessary to model constructions like the Occam2 alternative where some action is taken depending on which channel a message arrived. In

this case, the identification of the port is necessary in order to associate the port with the corresponding action. The string `"Precedence"` is a comment.

Finally, there are three methods used for input, computation and output descriptions: `type_input`, `type_oper` and `type_output`. For example,

```
void input_desc (n);
Node n;
{type_input ("inp", n, AND, SYSTEM_SIZE+1);}

void comp_desc (n);
Node n;
{type_oper ("comp", n, uniform(100*INT));}

void  output_desc (n);
Node n;
{type_output ("out", n, OR, 3, 1*int_size, 0.4,
                               3*int_size, 0.1,
                               5*int_size, 0.5);}
```

details the input, computation and output descriptions for the previous `gauss[0][1]` computation node (see above paragraphs, in the example of `comp_node`). The strings `"inp"`, `"comp"` and `"out"` are comments. The parameter `n` is a formal parameter used by the library. This variable is a pointer to the current computation node being described. It is important because the same input/output/computation description can be used by different computation nodes. In the above example of `type_input`, an `AND` input with `SYSTEM_SIZE+1` inputs (`SYSTEM_SIZE` being a constant of the application description) is described. This means that all the inputs of `gauss[0][1]` should arrive before executing the computation associated to the computation node `gauss[0][1]`. In `comp_desc`, a computation of average 100 integer instructions, distributed uniformly, is represented. Finally, in `output_desc`, an output of type `OR` is described. This kind of output means that only one output will be chosen: an output of 1 integer with probability 0.4, an output of 3 integers with probability 0.1, or an output of 5 integers with probability 0.5.

Originally, three types of inputs/outputs can be described: (1) boolean descriptions (like `AND`, `OR`). An `AND` input models a join of control threads, and an `OR` input models an Occam2 alternative [9]. An `AND` output models a fork of control threads, and an `OR` output models the C `case` instruction; (2) global

operations, like data broadcast. A data broadcast can be considered as an *AND* output, but the data are the same for each output; and (3) grouping input/output, that is, two computation nodes linked by a grouping arc should be executed by the same processor. This kind of description is important to model, for example, sequential pieces of code that share the same address space and that should be mapped to the same processor. Another application of grouping inputs/outputs is the possibility of clustering several tasks of the graph in order to study the impact of this grouping on the final performance of the application. In other words, grouping task is useful to perform a granularity analysis of a parallel program.

The quantitative costs can also be described as variable quantities: (1) as a constant distribution, in order to model well-known workload, as for example, message sizes; (2) random variables, in order to model, for example, branching probabilities; and (3) dependent variables, that is, costs that depend on other costs of the graph or on the inputs of the algorithm.

Very compact descriptions can be created using the native iterative constructions of the C language. Also, the C `random` function can be employed to create random variables. Standard structures of algorithms (e.g., trees, grids, diamonds) can be "canned" in C procedures which also allow the representation of recursive and hierarchical constructions. In a *DG-ANDES*, a loop is unrolled in order to fit the directed graph representation. The iterative instruction of the host language are suitable for the representation of loops. For example,

```
for (i=0; i<LOOP_LIMIT; i++)
   iteration[i] = comp_node ("iteration",
          input_desc, comp_desc, output_desc);
```

describes a loop in the model. When compiling and executing the program describing the model, the `for` instruction is executed and consequently the modelled loop is unrolled. The constant `LOOP_LIMIT` can be substituted by a random function or a value related to another attribute of the *DG-ANDES*.

The *DG-ANDES* is then a description of a family of parallel algorithms. Indeed, in a *DG-ANDES*, uncertainties can be modeled (e.g., average costs, branching probabilities, `OR` outputs). A *DG-ANDES* can be compiled and executed (it is a C program). For the work presented here, this execution produces

a DAG with all the vertices having AND inputs and outputs and all the (computation and communication) costs being constants. This means that all uncertainty is removed. The DAG is used by the mapping strategy and by the synthetic execution manager in order to produce a synthetic execution from which the required performance indices are computed.

4 *ANDES* and the synthetic execution manager

From the DAG until the synthetic execution, *ANDES* performs four main steps: (1) cost conversion, that is, given a parallel machine model, the graph costs (e.g. number of instructions to be executed, the number of exchanged integers) are reduced to the same unit (e.g., microseconds); (2) the DAG clustering; (3) the choice of which processor will execute each cluster (execution of the mapping strategies); and (4) file treatment in order to be read by the synthetic execution manager. These four steps plus the compilation of the *DG-ANDES* are executed sequentially on a Sparc Sun/4 workstation. The synthetic execution manager executes on the Meganode, a DMM with 128 Transputers and a statically reconfigurable interconnection network.

4.1 DAG pre-processing

The parallel algorithm quantitative model for a mapping algorithm is a valued undirected graph, where the vertices model processes and the edges model communication between processes (see section 2). The DAG is a richer application model, due to the precedence. However, it does not seem sensible to give a DAG for a mapping strategy. Therefore, a clustering algorithm is used to group the computation nodes of the DAG into clusters. In *ANDES*, clustering is done using the PYRROS DSC (Dominant Sequence Clustering) Algorithm [15]. The DSC algorithm "performs a sequence of clustering refinement steps and at each refinement step, it tries to zero an edge to reduce the parallel time" [15]. It has $O((v+e)logv)$ time complexity and $O(v+e)$ space complexity (v is the number of DAG tasks and e is the number of arcs). Clustering is done considering an unbounded number of processors of a completely connected architecture. The mapping strategies presented in the section 2 are then used to choose which processor

executes each cluster. Clustering in this study is *only* used to adapt the models described by a directed graph in an understandable representation for the mapping algorithms (undirected graphs). Currently, no detailed study is conducted in order to compare different clustering strategies and the impact of these strategies on the application performance. Finally, the DAG and the mapping information are stored in a file that is read by the synthetic execution manager running on the Meganode. It is important to remark that during this DAG pre-processing, there is an available *Meganode computation and communication model* which allows estimation of the computation and communication costs (expressed in time units) used by the clustering and mapping algorithms. These models are linear functions that associate number of operations to time in computation models and message size and topology to time in communication models. During this phase, it is also possible to estimate the total computation demand of the the DAG tasks (their costs and the machine computation model are known - see tap_p below).

4.2 Synthetic execution

The synthetic execution performed in order to obtain the desired performance indices is executed on the Meganode, a DMM that contains 128 Transputers (T800/20 MHz and 1 Mb of memory – communication links adjusted to 10 Mbits/s), interconnected by a hierarchical and reconfigurable network. The 128 Transputers are grouped in 4 *tandems* of 32 processors each. There is an internal switch connecting these 32 processors. Another switch interconnects the 4 tandems. This second level switch also interconnects the tandems to the host machine, a Sparc Sun/4 with a 4-Transputer board (each Transputer is a T800/20 MHz and 4 Mb of memory), 3 of them used for program development and only one connected to the Transputer network. The available Meganode is a single-user machine: there is no operating system that allows more than one application to execute on it. The available programming languages are Occam2 and C (mainly the InmosC toolset IMS-D4214). Routing is done by software using VCR (Virtual Channel Router), Version 2.0k [4].

The synthetic execution manager is the kernel that controls the synthetic execution on the parallel computer. It is a SPMD parallel program written in Inmos C using routing facilities provided by VCR. There are two types of processes (Figure 2): one executing on the root interface Transputer inside the

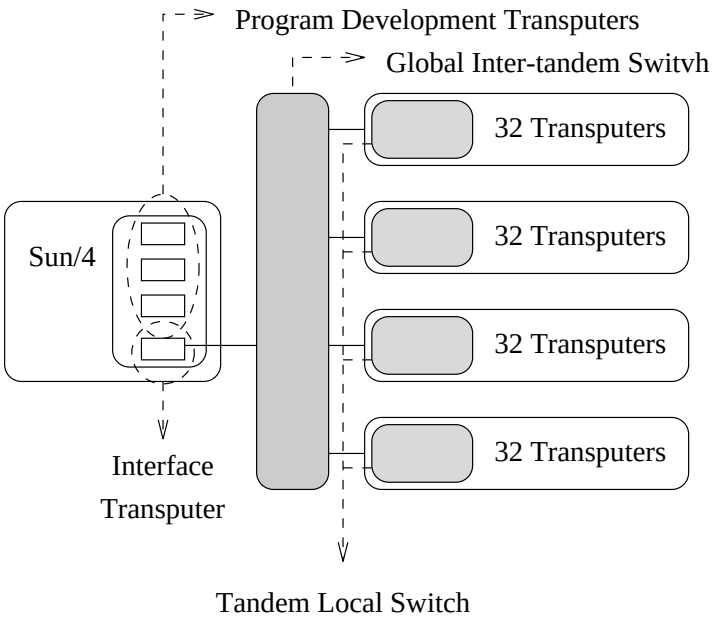

Figure 1: The Meganode.

host workstation (the `root` task) and one executed on each Transputer of the network (the `manager` tasks). These tasks (all the `manager` tasks and the `root`) are virtually completely connected, that is, there is a virtual communication channel between any pair of processes.

The root process reads the application quantitative DAG plus the mapping information and sends, to each `manager` process on the network, the information of the nodes (tasks) of the DAG mapped 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 mapped 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 load information.

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,

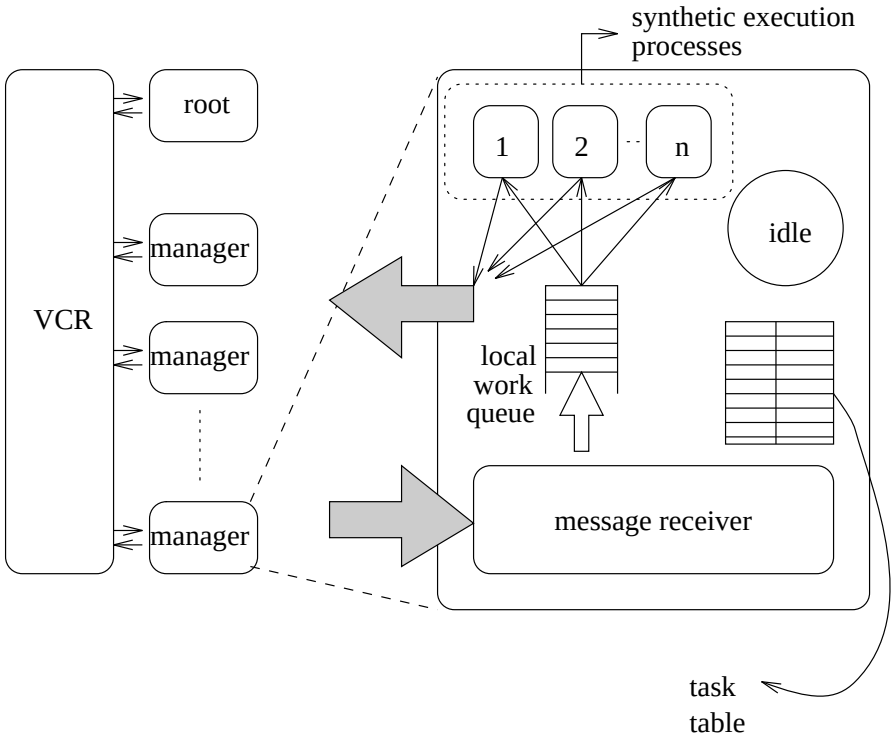

Figure 2: Process structure of the synthetic execution manager.

number of successor tasks) and puts them into a table (the *T-Table*). The second part consists of effectively managing the synthetic execution. It is only started if there are tasks placed on the associated processor. 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 output 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.

After the execution of the kernel, some data are obtained concerning the execution of the synthetic application. The time of the synthetic execution (T_{ex}) is measured. The time spent by the iterations of the `idle` process in processor p ($tidle_p$) can be obtained through the linear model of number of iterations versus execution time, specific for a given processor. In this way, $twork_p = T_{ex} - tidle_p$ gives the time a processor p worked. $twork_p$ can still be decomposed in two parts: (1) the time a processor spends executing the computations modelled by the input DAG (tap_p); and (2) the time a processor spends communicating, managing the synthetic execution and doing housekeeping (the overhead time toh_p). tap_p is computed before the synthetic execution. toh_p is then computed doing $T_{ex} - tap_p - tidle_p$, for each processor p. For a given application, the estimated overhead on processor p in microseconds due to the tool (not taking into account VCR and communication overheads) is $204 * NT + 215 * MD + 140 * MSGA + 40 * MSGD + 80$, where NT is the number of tasks placed on p, MD is the "multiprogramming degree" on p, $MSGA$ is the number of arriving messages in p, and finally, $MSGD$ is the number of messages leaving p.

5 Evaluation of the mapping strategies

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 (this model tries to model an irregular application); (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 total computation cost, the total communication cost (and the ratio computation/communication), the standard deviation of the costs among the tasks, the number of inputs/outputs of a group, its granularity (i.e., the mean time interval between two external communications of a group) and its virtual parallelism (the width of the *DG-ANDES*). For example, executing the program corresponding to the *ANDES* model of the Strassen benchmark, the following values for the main parameters are obtained: 1140 groups, a total computation cost of 8623185 microseconds, a total communication cost of 11564613 microseconds and a virtual parallelism of 92.69.

There are four types of cost functions:

- ADD is an additive cost function (non-overlap of computation and communication);
- MAX is a maximum cost function (overlap computation-communication);
- ROUT considers non-overlap of computation and communication and a store-and-forward routing;
- TOR considers non-overlap of computation and communication and a packet switching routing.

Considering the above 4 cost functions used to evaluate the quality of the placement, there are 4 greedy strategies (modulo, LPTF, LPTF with a quantitative criterion, and LPTF with a structural criterion). There are also 2 iterative algorithms: a simulated annealing and a tabu search, both starting from a solution defined by a greedy algorithm (LPTF with a quantitative criterion). These algorithms

are available in our research environment. Each model of the benchmark, clustered automatically or manually, is given to each mapping strategy and the corresponding synthetic charge is executed 100 times in order to obtain an average execution time. For each model, the communication/computation ratio is modified three times. The total number of different workloads is 102.

The target architecture is the Meganode multiprocessor itself, configured as a 4x4 torus. Topologies with more processors are considered (larger tori) in order to evaluate the scalability of the obtained measures. Also, the multiprogramming degree can be changed (if more than one task of the *DG-ANDES* is able to execute, they are executed in a time-sliced fashion).

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, the fraction of the execution time corresponding to the execution of the application tasks, the fraction corresponding to the overhead (due to the communication and to the tool itself) and the fraction corresponding to the idle time. The performance indices that can be analysed when using these strategies are the distribution of the groups on the processors and the reduction of the communication costs between groups due to the mapping (the communication cost between two groups mapped to the same processor is considered null).

For example, graphics like those presented in Figure 3 support the comparison between the synthetic execution and the performance given by the mapping strategies (the values are normalized: the reference value is that given by the modulo strategy). Figure 4 presents an example of a graph representing the load of the machine during the synthetic execution. On a given bar, the lower region represents the useful work, the middle region corresponds to the overhead and the upper region corresponds to the idle time of the processor. The large overhead fraction is caused by an excessive degree of the application communications, considering that the overhead caused by the tool is forced to be inferior to 10% of the total computation cost of the program model.

Results. A linear regression was performed between the set of cost function values (which estimates the execution time of the synthetic workload) and the set of measured execution times. The cost function TOR which represents well the behavior of the Meganode is the best cost function. The linear model

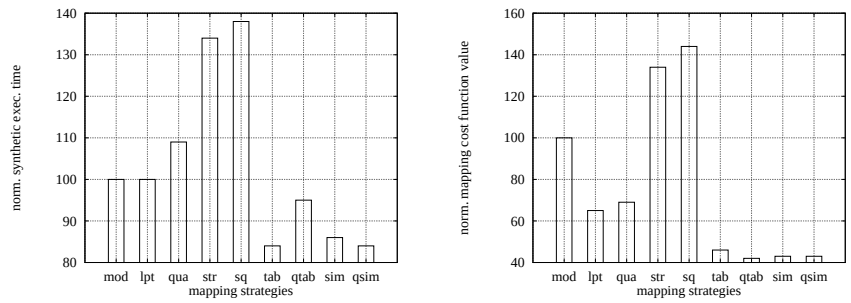

Figure 3: Synthetic execution time and cost function values per mapping strategy for Strassen benchmark (mod=modulo, lpt=standard lpt, qua=quantitative lpt, str=structural lpt, sq=structural and quantitative lpt, tab=tabu, qtab=tabu from a quantitative solution, sim=simulated annealing, qsim=simulated annealing from a quantitative solution).

given by the regression is the closest if compared with the execution time (i.e., cost function value close to the measured execution time). The cost function ADD (non-overlap of communication and computation) is also good. The table below gives the slope of the obtained linear model and the quality of the regression (a good regression has quality close to 1):

	ADD	MAX	TOR	ROUT	Clustering
slope	1,213	1,620	1,185	1,636	Pyrros
quality	0,923	0,894	0,907	0,947	Pyrros
slope	1,370	2,147	1,302	1,664	manual
quality	0,734	0,796	0,794	0,771	manual

Considering a given program model (among 17), a given communication/computation ratio (there are 3 for each model) and a given clustering (there are 2 types of clustering), 6 mapping strategies were applied. In 76% of the cases, the worst mapping algorithms (the algorithm that gives the worst execution time) are modulo and LPTF. In 12% of the cases, modulo and LPTF are not the worst strategies but the difference between the best strategy and the worst is not superior to 10%. When this difference is greater than 10%, the worst strategy is iterative (it may go to a local minimum that is not a global minimum). The conclusion is that all the strategies that do not consider the communication as mapping criterion (modulo and LPTF) are not good in practice. In 81% of the cases, the best strategy is one that considers the existence of the communications. We verify also that the best speed-ups (the ratio between the total sum of the computation costs of the graph and the measured execution time) are obtained for the

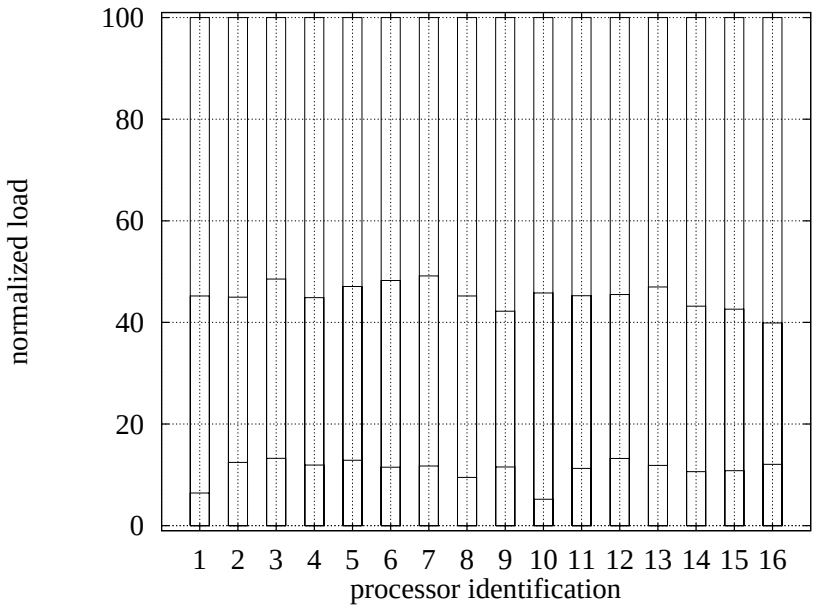

Figure 4: The load profile of the Strassen benchmark for qtabu mapping strategy.

models having the lowest communication/computation ratio. A very interesting conclusion is that the iterative algorithms do not improve reasonably the mapping given by a greedy algorithm. It is verified that for the given benchmark the improvements obtained with the simulated annealing and the tabu are inferior to 10%. A possible reason of this behavior is the regularity and symmetry of the 17 models. A modification of a already done mapping (e.g., by pair exchanging) on these program models do not represent a considerable gain in terms of execution time. Taking into account that a benchmark is a representative set of models of *real* programs, a greedy mapping algorithm is enough to map tasks on processors.

6 Conclusion 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 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 [12].

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

Meganode is a Telmat trademark. Sun and Sparc Sun/4 are Sun Microsystems Inc. trademarks. Occam, Occam2 and Transputer are Inmos, Inc. trademarks. The *ANDES* environment is related to the *ALPES* (*AL*gorithms, *P*arallelism and *E*valuation of *S*ystems) research group inside the *APACHE* project. The *ALPES* group is supported by the CNRS, INPG, PRC C^3, MESR, CNPq/Brazil and the Rhône-Alpes Region.

References

[1] S. H. Bokhari. On the mapping problem. *IEEE Transactions on Computers*, 1981.

[2] S. W. Bollinger and S. F. Midkiff. Processor and link assignment in multicomputers using simulated annealing. In *Int. Conf. Par. Proc.*, 1988.

[3] T.L. Casavant and J.G. Kuhl. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Transactions on Software Engineering*, 1988.

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

[5] Fred Glover and Manuel Laguna. *Tabu Search, a chapter in Modern Heuristic Techniques for Combinatorial Problems.* W.H. Freeman, N-Y, 1992.

[6] H. Kasahara and S. Narita. Practical multiprocessor scheduling algorithms for efficient parallel processing. *IEEE Transactions on Computers*, 1984.

[7] João Paulo Kitajima, Cécile Tron, and Brigitte Plateau. *ALPES*: a tool for the performance evaluation of parallel programs. In Jack J. Dongarra and Bernard Tourancheau, editors, *Environments and Tools for Parallel Scientific Computing*, pages 213–228, Amsterdam, 1993. North-Holland.

[8] C-L. Lee. Parallel machines scheduling with nonsimultaneous machine available time. *Discrete Applied Mathematics*, 1991.

[9] INMOS Limited. *Occam 2 reference manual*. Prentice-Hall International Series in Computer Science. Prentice-Hall International, New York, 1988.

[10] V.M. Lo. Heuristic algorithms for task assignment in distributed systems. In *Proc. 4th Int. Conf. Dist. Comp. Systems*, 1984.

[11] V.M. Lo. Algorithms for static task assignment and symetric contraction in distributed systems. In *ICPP88*, 1988.

[12] Michael G. Norman and Peter Thanisch. Models of machines and computation for mapping in multicomputers. *ACM Computing Surveys*, 25(3):262–302, September 1993.

[13] J. L. Pazat. *Outils pour la programmation d'un multiprocesseur à mémoires distribuées*. PhD thesis, Université de Bordeaux I, 1989.

[14] H.S. Stone. Multiprocessor scheduling with the aid of network flow algorithms. *IEEE Transactions on Software Engineering*, 1977.

[15] 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.