Distributed Simulation of Parallel Computers

Loïc Prylli, Bernard Tourancheau

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Distributed simulation of parallel computers

Loïc Prylli and Bernard Tourancheau

N 2767

Janvier 1996

PROGRAMME 1

ISSN 0249-6399

Distributed simulation of parallel computers

Loïc Prylli and Bernard Tourancheau*

Programme 1 — Architectures parallèles, bases de données, réseaux
et systèmes distribués
Projet ReMaP

Rapport de recherche n°2767 — Janvier 1996 — 26 pages

Abstract: We propose a method for emulating the behavior of your favorite MIMD computer on a network of workstations, so that programs written for the MIMD target can run on the distributed emulated version with just a recompilation. This is achieved by executing the computational parts as in the native run, with the message passing being simulated.

The hardware of the target machine is simulated so that the behavior of the application is identical to a native run on the simulated computer with virtual timings and trace files. Moreover, our complexity analysis sets up the conditions required to achieve a good speedup as a function of the number of simulation hosts, the network latency and the granularity of the application.

Key-words: distributed simulation, parallel computers, performance estimation, non-intrusive monitoring

(Résumé : tsvp)

Le projet *ReMaP* est un projet commun CNRS – ENS Lyon – INRIA. Ce travail est financé par le CNRS contrat PICS, le GDR-PRC PRS action EXEC, la CEE programme EUREKA contrat EUROTOPS.

*Laboratoire LIP, URA CNRS 1398, Ecole Normale Supérieure de Lyon, F - 69364 LYON Cedex 07, e-mail: firstname.lastname@inria.fr

Unité de recherche INRIA Rhône-Alpes
46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1 (France)
Téléphone : (33) 76 57 47 77 – Télécopie : (33) 76 57 47 54

Simulation distribuée d'ordinateurs parallèles

Résumé : Cet article présente nos travaux sur la simulation distribuée, dirigée par les événements, d'ordinateurs parallèles à mémoire distribuée. Nous décrivons nos algorithmes de simulation distribuée et l'analyse théorique des conditions nécessaires pour obtenir une bonne accélération de la simulation.

Notre implémentation d'un tel simulateur permet, à partir d'une application écrite pour un ordinateur MIMD, une exécution sur un ensemble de stations de travail avec seulement une recompilation du code. La machine cible est simulée au niveau matériel de manière à ce que le comportement de l'application soit le même que sur l'ordinateur simulé avec le maintient de temps d'exécution virtuels et la construction des fichiers de traces correspondants.

Les résultats obtenus corroborent notre étude de complexité et suggèrent le nombre de processeurs de la machine simulante, la granularité de l'application parallèle qui permettront d'obtenir une bonne efficacité de la simulation parallèle.

Mots-clé : simulation distribuée, ordinateurs parallèles, estimation de performances, monitoring non-intrusif

Introduction

As parallel computers have become more widely available, a lot of tools have been developed for them. These tools try to examine: the application performances, the load-balancing, the effective parallelism and the communication network problems (contentions, links utilization).

We propose a new tool, that makes it possible to one to simulate a MIMD[1] computer in order to run real parallel applications by just recompiling the source code on a workstations cluster, or a small parallel machine. This simulation will give the application results and all the timing and monitoring information corresponding to the simulated computer.

We try to be as general as possible to be able to simulate a wide range of parallel computers and provide several application programming interfaces (APIs), including NX[2][14] and PVM[3][9] (notice the emerging standard Message Passing Interface (MPI) exists on top of both preceding APIs).

All the code of the application is in fact truly executed as in a native run, but all the functions related to message-passing or to time measurement are treated by an engine that simulate the timings and behavior of the target machine. This simulation is based on conservative discrete event simulation.

The result of the simulation can be exploited either by a trace-file generated during the simulation (which can be visualized with classical tools like Paragraph[11]), or by time measurements made inside the application that reflect the virtual time.

We wanted to simulate a wide range of parallel computers, but we also wanted to simulate real applications, that is to say, applications that deal with a great amount of data and that are also quite demanding on CPU power. Consequently we parallelized the simulation (of a machine that is itself a parallel computer) and this is the main novelty of our work. Last, we wanted it to be portable.

Moreover, we studied the theoretical limitation of distributed simulation under our assumptions (cf.§3). Hence, we could predict, for a given simulation machine the conditions necessary to achieve a good level of efficiency (§3.3). For instance, this is especially helpful in determining the number of nodes that should be used to

[1] Multiple Instruction flow Multiple Data flow.

[2] The Intel Paragon and iPSC "old" version of the operating system which is still well spread as an API for communications.

[3] The well know Parallel virtual Machine system which provides (one of) the more often used API for communications.

achieve the maximum simulation speedup on a LAN[4] of workstations. This is the second novelty of our work

Until now there is no general tool that allows the simulation of a parallel computer at the application level. That is to say to answer the question: "how much time will *this application* take on *this computer*?". Other work done on this subject is either dedicated to only one machine, or focuses on a very high accuracy of the simulation at the hardware level and so is restricted to simulate "toy" applications. Moreover, as we parallelize the simulation itself a severe limitation of traditional application-level simulators is avoided: the limited amount of memory of one workstation and the elapse time. By using a network of workstations or a parallel computer, we can now deal with larger problems and decrease the execution time of the simulation as well. Parallel simulation is commonly used for network studies but to our knowledge it has never been done for general MIMD computers at the application source code level (except for some work in progress for the Paragon).

The interest of the simulation tool and some backgrounds on this subject are described in section 1 and section 2 then the theoretical study of its efficiency in section 3. The specification and implementation are described in section 4 and 5 and we show in section 6 experiments that proved the quality of the results obtained as well as simulation speedups.

1 Why is simulation useful?

Simulation is useful development or application tests without the target machine. In the case of modular or custom made parallel systems, it is also useful for the design and choice of the hardware to be bought or made. Moreover, in order to develop and optimize parallel applications, the most commonly used methods have been instrumentation and runtime tracing. There are a lot of different ways to do this monitoring [21], the instrumentation of the program can be done more or less automatically, and there are several approaches to visualization[18, 11]. The main problem with observational analysis is neutrality. It is actually difficult to instrument and collect data without disrupting a lot the timings of the target application. In the worst case, even the behavior of the application can be changed due to the non-determinism introduced by parallelism. So a lot of effort has been made, in software as well as in hardware to ensure as much neutrality as possible[5].

Hence, our parallel simulation tool allows the following new features:

[4] Local Area Network.

- It insures neutral observation.
- It allows development without access to the real machine.
- It makes it possible to one to design and study a machine without (or before) building it.
- It allows for the testing of massive parallelism on real applications without requiring the target machine and without a huge execution time.

1.1 Sequential simulation of parallel computers at application level

Some tools already exist to simulate the execution of a parallel application on some peculiar hardware. Generally, the application is automatically transformed into a sequential program. This program will simulate all events that would have occurred on the target machine during a real run of the application. The result of the simulation can be examined with the appropriate tools. Among them are:

Protéus [3] : This tool allows for quite a realistic simulation. First, at compile-time the cost of each basic block of the application is evaluated to be able to take it into account during the simulation. Then the application is sequentially simulated with a simulation engine, which is responsible for: maintaining a virtual clock, sharing the simulator CPU among the different simulated virtual nodes and simulating of the communications.

EPPP simulator [17] : EPPP is a complete programming environment, including a simulator based on Protéus. The evaluation of computation time has been improved. The compile-time analysis is done by looking at each assembly code basic block generated for the target machine.

EPG-sim [15] : This is an integrated set of tools allowing trace generation, serial execution-driven simulation, and trace-driven simulation.

Beside those cited, some other work has been done: **Tango**[6], **PEET**[10].

1.2 Parallel simulation of MIMD computers

In the case of trace-driven simulation, there are already some tools and studies that allow one to achieve an efficient parallel simulation. The classical methods are based on Chandy-Misra-Byrant protocols[4], "time warp" techniques[12], or "windows"

techniques[13] and results have been obtained either for specific or parameterized networks[2, 15].

Some execution-driven simulation can use the same technics than trace-driven simulation, in the case where the set of messages exchanged between processors is deterministic, and where the execution path of each application process is completely insensitive to timings. This is equivalent to generate and treat the trace online (for instance EPG-sim is able to use this mode).

Another tool adapts its behavior dynamically :

Lapse [7] : LAPSE is a parallel simulator for the Paragon. It assumes that the behavior of applications is insensitive to timings most of the time. It uses "windows" technics, that work well in this case. If the application becomes sensitive to timings, the windows become so small that this part of the simulation becomes sequential.

2 Simulation of parallel program by discrete events

We will present a quick overview of the sequential discrete-event simulation of parallel programs with our assumptions and point of view. The next section shows how we exploit parallelism into our simulation algorithm.

2.1 Structure of the simulation engine

Events are unavoidable to simulate a complex system where some parts evolves separately and interacts at certain times, they are a representation of these interactions.

The global state of the simulated computer at a given time is represented by a set of variables. The simulation progresses by way of transitions, one transition modifies the variables to set a new machine state and increases the time by a specific amount. Hence the global state of the machine is changed only at precise countable points in time, that's why we speak of *discrete* events simulation.

One important structure that is maintained is a queue of *events*. Two attributes are associated with each event, its nature and its time of occurrence (called its time-stamp).

In here, we consider all changes to the state of the system to be atomic. Then actions that last will be represented in the model by two events, one at the beginning of the action and one at the end. The evolution of the real system in between is supposed not to be meaningful in the scope of the simulation.

At a given time t, let Q be the queue of events and S be the state of the system. The simulation engine consists basically in the following algorithm:
While true

1. Remove an event e with the smallest time-stamp from Q .
2. Modify S by taking into account the occurrence of e. During this modification, events can be created that are inserted into Q.

EndWhile

At each stage of the algorithm, the virtual time of the simulation is given by the time-stamp of the event e that is currently being processed.

The representation of the state of a machine depends a bit of its architecture, but we can generally reduce the state of all machines with our assumptions (section 4.1) as follows:

- the state (active or idle) of each link of the communication network.
- the list of messages blocked at each router, waiting for a link availability.
- the list of links currently monopolized by each message in transit on the network.
- the state of each application process (data, stack, program counter), which is represented by a real process (or just a thread) on the simulating host, linked with a library that redirects message-passing calls to routines that interact with the simulation engine.
- for each processor node, a list of events blocked waiting for resources, a list of events corresponding to the received messages not already grabbed by an application process, and some others structures depending on which flow control protocol is used.

This is a simple model, but notice that we can extend it easily; for instance, if the routing function is more complex then the router maintains some additional state, etc ...

2.2 The fundamental events and their associated actions

We describe in the following the most representative types of events and the corresponding actions that are processed when treating them (depending on the simulated architecture). We will describe the simulation of a circuit-switched network and, from this example, the worm-hole case is straightforward.

Treatment of a "transmission" event. Let s be the source site, t the sending date, d the target site. The following algorithm is implemented :

1. Acquire on s the resources needed for emission, eventually this can lead to "sleep" (see explanation below) if the resource is not available at once.

2. Compute t' the date at which the resources are ready to use (there can be a "switching time" associated with some resources). Insert an event at date t' of type routing into Q (the event queue).

This algorithm is considered atomic if all resources are available. On the contrary, the information necessary to do the rest of the processing is inserted into the queue of blocked waiting events associated with the resource. The algorithm will resume when another event frees the resource.

Treatment of a "routing" event. Let n be the node on which the message arrives :
If n is the final destination of the message then try to acquire on this node the resources needed for delivery as in the case of a transmission event, compute the time at which the delivery will be terminated and insert an event of type "end of transmission" in Q.
Or else, (n is an intermediate node), compute the next node n' with the appropriate routing function, wait for the availability of the resource corresponding to the link between n and n', add a switching time to obtain the final date of the event of type routing (for the node n'), insert it in Q.

Treatment of an "end of transmission" event. All the resources used for the message are freed, in particular the links along the path between the sender and the receiver, (this can lead to renewal of some actions that were waiting for the corresponding resources). The message is delivered to the application.

3 Constraints for the parallelization of the simulation

We have seen in the preceding section our sequential simulation algorithm. We will present the main problems that occur when we parallelize it in this section.

We investigate how several events can be proceeded in parallel. The classical problem that arises is the constraint of coherence in a parallel simulation :
The simulation algorithm must ensure that the results of transitions on the machine

state are exactly the same as if they had been processed sequentially in chronological order.

It is not possible to apply a "Time Warp" or any optimistic parallel simulation method because we have seen that in the state of the machine, we consider real application processes and it is not possible to save and restore the state of such processes (or if possible, the cost would be prohibitive because with C or F77 languages, all the program state has to be saved, i.e. the data and the stack).

We cannot use a simple conservative window algorithm either because in a window algorithm, we have to compute at a time t an interval δ during which it is assumed that no event is generated, and in our case we cannot predict the service time of such an event. Moreover the service time is very low for components such as routers compared to the typical inter-events application time. So the only possible window interval δ would be too small to allow any parallelism. Notice that this restriction is removed if we suppose that both the pattern of messages exchanged by the application and the execution path of the processes are completely deterministic, thus allowing an efficient look-ahead (see [16] for more details).

3.1 Using the latencies of the simulated machine

We describe here each element of the simulated machine as having a service time. In practice, the usual components present too small a service time in comparison with the inter-event occurrences to permit the ability of exploiting parallelism with a window-type strategy (see [16]).

We define, between any couple of components (p, q): the latency $l(p, q)$ to be the minimum time possible between an event that occurs at p and an event that occurs at q who is a consequence of the first one.

Let h_x design the time-stamp of an event x. At each stage of the simulation algorithm, one can choose for the next transition, to compute any event $e \in Q$ verifying $h_e < h_{e_0} + l_{s_0,s}$ where e_0 is the event with the smallest time-stamp and s, s_0 are the locations where e and e_0 occurred. The simulation algorithm become non-deterministic and so has an inherent potentiality for parallelism.

In the corresponding distributed algorithm, Q and S are in fact distributed among a certain number of processes. Each one is responsible for a site and then deals with all the transitions associated with this particular site. On each process the following algorithm is executed:

1. Let t be the smallest time-stamp among the events owned locally.

2. For every other site q, wait until the process associated with q reach time $t - l_{q,p}$ where p is the local site.

3. Modify S by taking into account the occurrence of e. During this modification, events can be created that are dispatched to the appropriate site.

The approximate parallelism provided by this algorithm will depend essentially on the ratio of the typical interval between two events on the same site other the typical information transfer time between sites. In practice in our case, the only sites where the local state progresses independently of the other nodes are the compute processes; every other sites (routers, links) are essentially driven by external events (i.e. events not generated on the same site). It roughly means that there is a synchronization with the neighborhood at each event.

A simple complexity study shows there is no parallelism exploitable at this level :

Let n be the number of events for a unit of time. Let β the communication delay, and δ the service time of the event (i.e. the time during which the event cannot influence the rest of the system). Let t be the time of processing of the event. The sequential time of the simulation of one unit of time is $T_{seq} = n \times t$. Assuming an ideal machine with infinite number of processors and constant communication multicast time, every event can be on a different processor, thus parallel simulation is $t + \beta$ for a δ period of time hence $T_{par} = \frac{t+\beta}{\delta}$ for one unit of time. The maximum speedup is then

$$\frac{T_{seq}}{T_{par}} = \frac{nt\delta}{t+\beta}$$

With classical parameters instantiation, the speedup is less than 1.

But it will be a useful optimization when used in conjunction with the algorithm described below.

3.2 Master slaves organization

Anyway we can try to preserve the parallelism inherent to the application by distributing the computations of the different application processes.

With that aim, one node of the simulation machine takes charge of several nodes of the simulated machine. To take into account contentions and network constraints, a master process simulates the communication hardware and delivers in order (chronologically with regards to the virtual time) acknowledgments of the messages that are exchanged on the network between application processes that we have called the

slaves. Each slave has to inform the master of the virtual time reached by the nodes it simulates. Notice that the master is able to run on a node with several slaves.

3.2.1 Cutting the computation phases

Computation processes have a local evolution of their state between two calls to the message-passing library. But if we stay with our previous model, a computation phase is only considered as one atomic transition leading to no parallelization at all![5]

Hopefully a computation phase is something that can be decomposed. Then in the middle of a computation the other simulation processes can be informed of what simulation point is reached. Hence, they can eventually start/continue their computation phases, which would proceed in parallel with the rest of the current computation. The problem is that there is no obvious decomposition granularity, and it would be too costly to decompose the application at the instruction level, or on the contrary, a loss of parallelism would occur at too large a granularity.

Our solution to this problem is to allow an interrupt-driven decomposition. That means, when the master must wait for a slave to reach a certain point in time, for instance before allowing the delivery of a message which will start a computation phase on another node, it interrupts the slave 's computation phase. The slave answers if it has or has not reached that corresponding point in time and if not, it sets a timer in order to inform the master as soon as it reaches this critical point. Moreover when several virtual nodes are simulated on a single real node, we will see later that it is essential to be able to switch between the processes (representing virtual nodes) in the middle of computation phases. So from now on we will assume that computation phases can be dynamically cut into several parts.

3.2.2 Algorithm on an ideal simulation host

Let V (for Virtual) be the number of nodes of the simulated machine and R (for Real) the number of nodes of the simulating machine. We assume from now on that the *simulating* host has the following properties:

- Computation phases can be cut into "infinitely" small parts without overhead.
- Communication can fully overlap with computation.

[5] this is a consequence of 3.1, where t is large but cannot be predicted

- The average latency of small messages between a slave process and the master process is β.

3.2.3 Algorithm of the master

Our simulation model has changed a bit since §2.1 with the introduction of interruptions in computation phases, but we have still a set of events Q that is completely managed by the master.

The algorithm on the master is then:

While true

1. Let e be the first event in Q (i.e. with the smallest time-stamp).
2. Wait until all application process to be inactive (waiting for a message or some information from the master) or until a point later than the time-stamp of e is reached (more precisely later than the time-stamp of e minus the latencies β corresponding to the communication path of short messages, see §3.1).
3. Run the action corresponding to e (it can result in the starting of computations on slaves).

EndWhile

3.2.4 Algorithm of the slave

Let $N = \frac{V}{R}$ be the average number of virtual nodes managed by a slave. We can represent the state of the slave by $(t_i)_{1 \leq i \leq N}$, each t_i represents the virtual time reached by one of the managed nodes. The vector t represents the advancement of the simulation on one slave.

When a virtual node reaches a communication point, it must wait for the master to inform it that all slaves have reached "this point". We will say that the virtual node is *blocked*.

The algorithm of a slave is composed of the following actions:

1. Let S be the set of nodes that are not blocked. Advance their computation uniformly: i.e. it runs the virtual node of S with the smallest t_i. As we assumed that we can switch with an infinitely small granularity between nodes of S, there will be a subset of the t_i that will increase at the same time.

2. When a virtual node becomes blocked (at a receive call for instance), it sends the corresponding t_i value to the master. Hence the master can inform all the slaves when a condition on the t_i has been reached.

Notice that at any time, a message from the master can unblock a node. For instance, by ensuring there will be no more delivery of application messages with a stamp less than the point of blocking t_i. The node can then continue its computation, running some computations based on messages available at this point.

3.3 Efficiency analysis

We will study the efficiency of our algorithm. For the sake of simplicity, this will be done on an average virtual application defined as follows:

- The computational phases have an average duration of time M between two message-passing calls.
- All process are busy at any given time.

The execution profile of each process will be computation phases with an average of $\frac{1}{M}$ communication points by unit of time.

Lets consider a particular slave and try to determine the conditions necessary to avoid idle states in the simulation.

We can consider a "computation cycle" beginning at a time when we receive a message from the master unblocking a node. Let t be the state of the slave as defined in §3.2.4. The critical path to go to the next "computation cycle" consists of several repetitions of the following:

1. The master unblocks a node of a slave.
2. The computation on this node progresses until the next global blocking point, on average that means a $\frac{M}{V}$ computation (at some conditions, see below).
3. The slave returns its status to the master.

These steps are represented on the space-time diagram example of figure 1.

Property 1 *There will be on average R steps before returning to the initial slave, that means a critical path of length $R \times (\frac{M}{V} + 2\beta) = \frac{MR}{V} + 2R\beta$. As the amount of computation of one cycle is M, there is no idle time if $M \geq \frac{MR}{V} + 2R\beta$.*

In practice we will have $V \gg R$, so the remaining condition is $M > 2R\beta \Leftrightarrow R < \frac{M}{\beta}$.

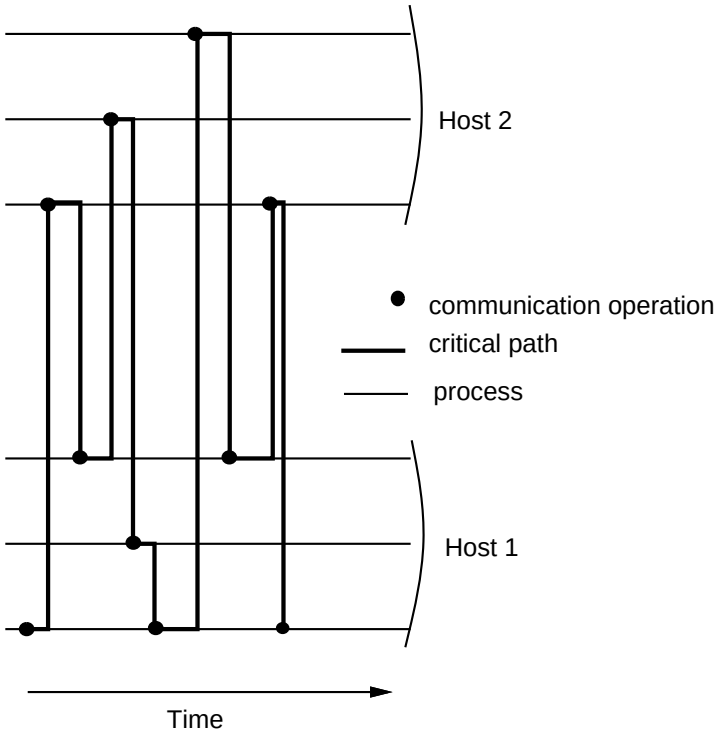

Figure 1: Critical path representation

We must now demonstrate that average considerations lead to a valid result.

Let W_a denote the amount of computation already done "in anticipation", i.e. if the node just unblocked is at time t_0, $W_a = \sum_i t_i - t_0$. Let W be the total amount of work that can be done from t_0, $W = \sum_i t'_i - t_0$ where t'_i is the next blocking date of node i. On average $W = \frac{MV}{2R}$. Figure 2 represents W_a and W on a space time diagram.

In the computation of the critical path length, we said that the time from a blocking point to the next global blocking point was $\frac{M}{V}$, and for that we assumed that at the time the master sends the unblocking message, it knows also the time of the next blocking point. A sufficient condition for that assumption is that for all slaves, the W_a are greater than M.

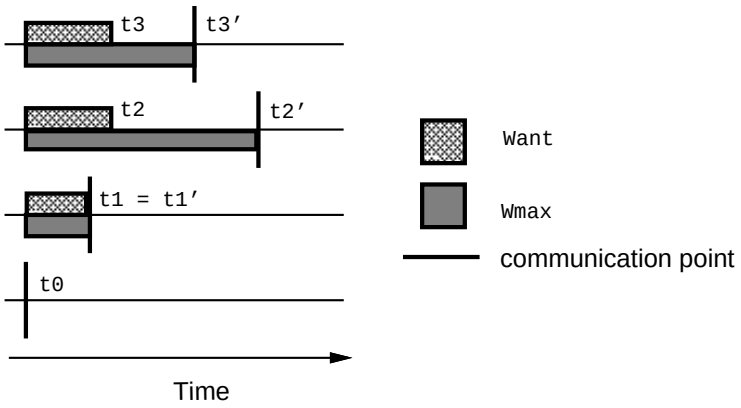

Figure 2: W and W_a at beginning of a cycle on one slave

Let's have look at what happens across several cycles. If the W_a are too small, the critical paths are longer but then, before leading to idle states, the W_a increases. When the $W_a >= M$ and as we must have $R < \frac{M}{\beta}$, the critical path between two cycles is smaller than M so the W_a decreases. So on average the W_a values will stabilize themselves somewhere below M. Moreover, if $V \gg 2R$, the maximum values for W_a is several times M which ensures that reasoning with average values is valid.

Note that average values are taken among different nodes *at one time* and not on one node throughout time. If all nodes do small computations at the same time, then the efficiency will drop.

Now that our average evaluation is validated, we have to prove that we can generalize our analysis of a special kind of application (see [16]) to more general cases. The point is to see how idle times influence the performance of the simulation. Let M now represent the interval between two computational phase starting points. M will be decomposed into a computation part M_c and a idle part M_i. Of course we will still consider average values. The algorithm does not change at all, there are some nodes that are simply idle instead of busy and the new complexity formulae for W is now $W = \frac{V}{R} \times (\frac{M}{2} - M_i)$. Hence we need $M_c > M_i$ and preferably $M_c \gg M_i$ (W must be several times greater than M). The condition on V and R becomes $\frac{V}{R} \gg 2\frac{M}{\frac{M}{2}-M_i}$.

All the other reasoning remains valid and then the efficiency will be optimal at the same condition $R < \frac{M}{\beta}$. Thus, we can generalize our previous result:

Property 2 *The efficiency of the simulation is directly related to the "granularity" of the application. The efficiency is close to 1 at the condition* $R < \frac{M}{\beta}$.

3.4 Simulation on a realistic machine

The preceding section studies the algorithm on a ideal machine. The strong assumption was that one CPU of the simulation host could be shared between different logical processes (representing virtual nodes of the application) with an infinitely small granularity.

In practice the processor CPU can be shared but with context switching between threads or processes with a granularity g depending on the system and the implementation (logical processes representing virtual nodes can be managed by several Unix processes or by several threads into a single Unix process).

The previous study can take g into account. The only important modification during the computation of the critical path length is when the time necessary to reach the next global point is computed. This time was $\frac{M}{V}$ and it must now be replaced by $\max(\frac{M}{V}, g)$, thus the supplementary condition becomes $Rg < M \Leftrightarrow R < \frac{M}{g}$.

Property 3 *In order to ensure that validity of* $R < \frac{M}{g}$, W_a *must now be greater than* $M + gR$, *but this change is not very important for the stability of* W_a *as long as* $R < \frac{M}{g}$ *which implies* $M + gR < 2M$.

3.5 Limitations due to the application

All the efficiency considerations we discussed until now did not take into account the time necessary to transfer the data messages between the different simulated nodes. We just spoke about the messages necessary for the coherence of the simulation. It is quite obvious that if an application cannot be run on the simulating host because of the bottleneck of its communication with a normal message-passing library like NX, or PVM, there is no hope to compensate by adding coherency constraints and sequential ordering on the simulation machine. So what we have determined, are the conditions at which the simulation could be done with the same order of speed as with a simple message-passing library on the simulation machine.

4 Specification of the simulation tool

4.1 Architecture modeling

We deal exclusively with distributed memory machines, linked with a classical communication network: point to point, multi-stage, or crossbar.

Our tool has to be parameterizable enough to model the target machines in a fixed format. The parameters we chose are the following:

Power of computing processors: For a wide range of problems, an estimation of the computation time for a given processor can be obtained with the scaling of the simulator's processor computation time. It gives an acceptable accuracy, especially for processors of the same family (for instance RISC). Moreover, the lack of precision introduced is often not greater than those introduced by a compiler change or even just a compiler option change. So our choice is quite simple, the computing processor is modeled by just one scalar obtained by benchmarks adapted to the type of computation[6]. The table 1 shows how we obtain this ratio of efficiency for 5 processors in 3 distinct domains: the LINPACK benchmark (dense linear algebra), the Whestone benchmark (aggregate melting of integer and floating-point operations), and the Dhrystone benchmark (integer operations)[7]

	LINPACK	Whestone	Dhrystone
Alpha	100	100	100
MIPS	20	28	19
Sparc	19	26	20
i860	37	38	24
RS6K	130	77	126

Table 1: Benchmark numbers in order to determine the scaling ratio estimation (the unit is not meaningful, Alpha is given as 100 for reference)

Notice that an approach like the one used in [17] allows more accuracy but is beyond the scope of our approach.

[6] The only limitation is that some timings effects due to processor specific strengths will not be reflected in the simulation (for instance cache size).

[7] Of course for a given processor, different processor clock rates will give different ratios and other benchmarks can be used, such as SpecInt and SpecFp.

The topology of the communication network and the routing strategy: The currently available topologies are: *ring, mesh and hypercube, complete network (i.e. switch)* with classical routing strategies.

Communication protocol: It can be *store and forward, circuit switched or worm-hole*. Depending on the protocol, several others parameters must specified: the packet size if messages are split, the flit-size for worm-hole communication, etc..

Communication numerical constants: These parameters specify the band-width of the links, the switching time of the routers, the different software and hardware overheads implied in message transfers.

Parameters for the upper communication layer: These are some options to specify the size of communication buffers used and the type of control flow if any.

All these parameters are given by the user in a configuration file filled either manually or with a graphical interface. The file is read at the beginning of the simulation.

4.2 Related issues

The different message-passing Application Programming Interfaces (API) generally provide some high-level operations such as collective operations. Our simulator tried to reproduce the behavior of the corresponding software layer of the target machine by dividing our runtime library into two layers emulating one API. The upper layer implements the calls of the API and use the lower layer to simulate the sending of messages on the hardware. Although we cannot guarantee that we always use the constructor algorithms, we can reasonably assume that the discrepancies introduced in the upper layer are quite small. It is also in this layer that we will take into account the control flow protocol used on some machines.

There are some operating system (OS) features that we do not take into account: on a multi-tasked node, the OS will determine the scheduling policy, and on some systems it manages a virtual memory space with possibly paging or swapping. Given the number of different operating systems, it is not reasonable to take all possibilities in a general tool into account. So we adopted a conservative approach, we made some simple choices (for instance, we assume a simple round-robin scheduling if there are several threads or processes on one processor) and we assume no swapping or paging. Anyway, our approach seems useful in a lot of cases for several reasons:

- Most parallel applications do not rely on paging because in order to provide acceptable performance, it is necessary that all code and data can be stored in physical memory. Hence system impact due to memory management is negligible and it appears reasonable to ignore it in simulation.
- A lot of parallel applications using message-passing do not rely much on the operating system except at load time and for input-output operations. In particular, there is only one application process per node which most of the time eliminates the problems related to scheduling policies. In regards to input-output operations, it is true that our simulator will not give any hint when such operations are predominant over computation (there is also no benchmark of that type).
- On a machine that allows only a single application at a time, the operating system hardly has any influence on the behavior of the application. On a multi-user machine, such as a LAN of workstations, the operating system and the load introduces "random" disruptions. It is not the scope of our project to study this kind of disruptions. On the contrary, they make analysis and performance tuning of the application more difficult. So the fact that the machine we simulate is more deterministic than the real one is an advantage most of the time.

5 Implementation presentation

5.1 Simulator

For portability and simplicity reasons, our environment is built on top of PVM [9, 8].

There are 3 kinds of PVM tasks: the "Slaves" noted S that manage the different virtual nodes, the "main simulation engine" noted M (for Master), that simulates the communications on the virtual hardware, and there will be a certain number of application processes noted T (like Thread), each of them representing a virtual node. A slave and its attached nodes are all run on the same CPU.

These different entities are interconnected by several kinds of communication channels (cf. figure 5.1):

- The channels M$\leftrightarrow$S allow the slaves to cooperate with the master to progress in the simulation.

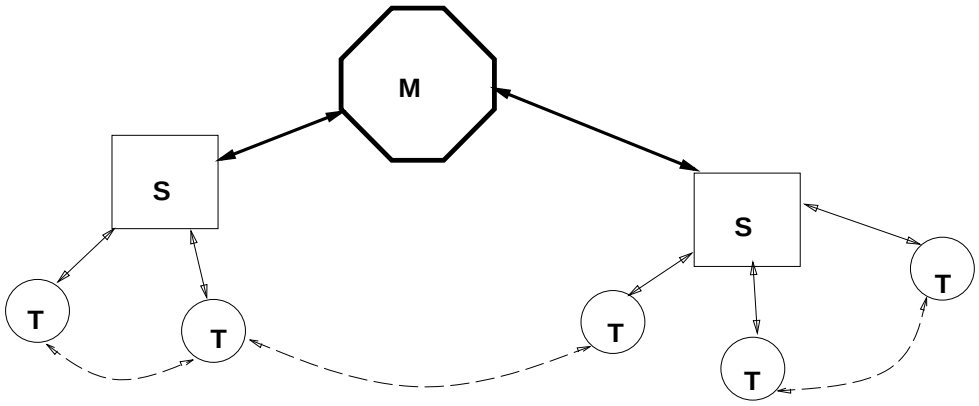

Figure 3: overview of the simulator organization

- The channels T↔S allow a slave to dispatch the CPU between the differents threads and to gather application messages that are eventually sent to the master.
- The channels T↔T allow raw data of the application to transit directly between application processes. Only information about such messages pass by the master node.

In a future version, we will perhaps change a bit this implementation in order to run a slave and all its attached processes within one single PVM task. Anyway this is only a technical detail to minimize switching time between the different processes on one CPU.

5.2 Trace generation

The simulation can be exploited by two means. On one hand the timing measurements on the application are virtual times (identical to those that would have been measured on the target machine) and so that allows us to analyze the application "manually". And on the other hand, there is the possibility of generating a trace file during the simulation. This file can then be examined with existing tools to do a post-mortem analysis. Note that the trace-file obtained corresponds to a per-

fect neutral observation of the execution (which is almost impossible with a real machine!).

We chose the PICL[20, 22] trace file format that allows the use of the Paragraph[11] and PIMSY/VIST[21, 19] tools to display the results of the simulation.

The trace generation is actually done at the master site, which has all the information about circulating messages and computing processors activities.

6 Validation of the simulator

In this part, we present several results obtained with our simulator. For our test, we took several programs written for the Intel iPSC860 and ran them both on the real iPSC860 and with the simulator on a LAN of SUNs.

The first test (figure 4) was a communication "ping-pong" test. This simple test validated the accuracy of the parameters for the target machine.

We then took two algorithms that come from the SCALAPACK library package [1] that provided numerical linear algebra routines on parallel computers.

The first one (figure 5) does a LU decomposition of a matrix, and then solves several linear equations by using this decomposition. We present the execution time of these two phases for several matrix sizes.

The second program (figure 6) does a QR decomposition, and then solves a linear system. As for LU, we indicate each phase time.

These results shows that the simulator has good accuracy. In our case we were simulating on a LAN of workstations with Sparc processors and a power ratio issued from the LINPACK benchmark. The difference between the real and simulated execution time is essentially due to the non constant power ratio between the two processors while using different vector sizes[8]. Nevertheless the biases stay relatively small (within 10%).

The table 2 gives a measurement of the efficiency/speedup of the simulation. Simulations are performed with different numbers of Sparc 5 workstations for two different applications (based on SCALAPACK routines again). The applications use different numbers of nodes. We give the execution time of the simulation in seconds for the different tests. The latency between these workstations is about 2 milliseconds and the computation between message-passing calls for the applications was quite large (about 100ms). The results are not perfect and we hope to get closer to the theoretical bound by simulating all processes on one host within a single

[8] A cache optimized version of the Basic Linear Algebra Subroutines (BLAS) is provided on the Intel machines.

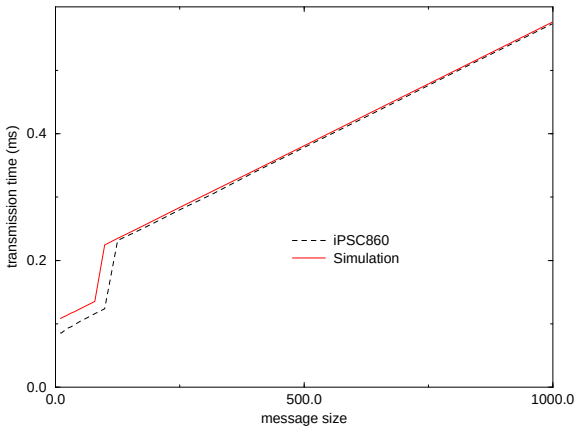

Figure 4: ping-pong simulation for the iPSC860

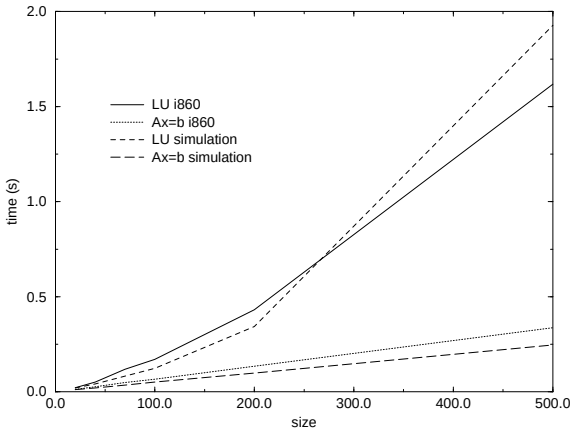

Figure 5: LU simulation on 16 nodes

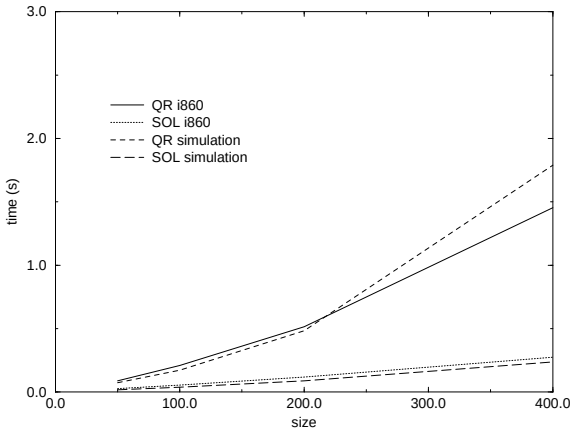

Figure 6: QR simulation with 16 nodes

address space (using threads instead of processes). Nevertheless, those examples show already the effectiveness of our simulator on a LAN of workstations.

Number of workstations	Appli1 with 16 nodes		Appli2 with 25 nodes	
	Time (sec)	Speedup	Time (sec)	Speedup
1	50	-	88	-
2	29	1.72	58	1.52
4	18	2.77	38	2.32
8	14	3.57	27	3.26

Table 2: Speedups obtained for the simulation of 2 applications with different number of processor in the simulator.

7 Conclusion

At this stage of our work, we obtained some promising results. This tool seems to be useful in several cases: for the development of parallel applications without having

an account on the target machine, for the neutral analysis of an application run, and to help in the design and study of a future parallel machine.

The tests that have been done with both simulation and native execution show that good accuracy is obtained easily with the simulator.

The other interest of this work is the theoretical study of the parallelization efficiency of our distributed simulator. We are able to characterize the type of simulation that can be done in parallel as a function of the application granularity and the communication latency of the simulation host.

Today's implementation with processes does not follow completely the assumptions of our theoretical study (the granularity is too big). Hence, the speedups growth in Table 2 slows rapidly. But we expect them to follow more closely our theoretical upper bound with the threads implementation that is in progress and in practice, with today's parallel computers' parameters, the speedup should increase steadily with up to a dozen workstations for a wide range of parallel linear algebra applications.

Notice also that finer grained applications can be simulated using a small parallel computer that provides a much smaller communication latency than a LAN.

Some work is in progress to allow the use of the simulator with more APIs, to reduce simulation overheads using threads instead of processes and to group the threads into one PVM task.

References

[1] E. Anderson, A. Benzoni, J. Dongarra, S. Moulton, S. Ostrouchov, B. Tourancheau, and R. Van de geijn. Lapack for distributed memory architecture progress report. In *Fifth SIAM Conference on Parallel Processing for Scientific Computing,*, 1991.

[2] B. Berkman and R. Ayani. Parallel simulation of multistage interconnection networks on an simd computer. *Advances in Parallel and Distributed Simulation*, 23:133–140, Jan. 1991. SCS Simulation Series.

[3] E. A. Brewer. Aspects of a parallel-architecture simulator. Technical Report MIT/LCS/TR-527, Massachusetts Institute of Technology, Laboratory of Computer Science, February 1992.

[4] K.M. Chandy and J. Misra. Distributed simulation : a case study in design and verification of distributed programs. *IEEE Trans. on Software Engineering*, 5(5):440–452, September 1979.

[5] A. Couch and D. Krumme. Portable execution traces for parallel program debugging and performance visualization. In *Proceedings of the 7th Conference in High Performance Computers, communications and application (HCCA7)*, May 1992.

[6] H. Davis, S. Goldschmidt, and J. Hennessy. Multiprocessor simulation and tracing using tango. In *ICPP*, 1991.

[7] Phillip M. Dickens, Philip Heidelberger, and David M. Nicol. Parallelized direct execution simulation of message-passing parallel programs. Technical Report 94-50, Institute for Computer Application in Science and Engineering, NASA Langley Research Center Hampton, VA 23861-0001, June 1994.

[8] Jack Dongarra and Robert Manchek. *PVM3 User's Guide and Reference Manual*. Oak Ridge National Laboratory, 1994.

[9] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Mancheck, and Vaidy Sunderam. *PVM: Parallel Virtual Machine*. Scientific and Engineering Computation. MIT Press, 1994.

[10] D. Grunwald, G. J. Nutt, A. M. Sloane, D. Wagner, and B. Zorn. A testbed for studying parallel programs and parallel execution architectures. Technical report, University of Colorado, April 1992.

[11] M. T. Heath and J. A. Etheridge. Visualizing performance of parallel programs. Technical report, Oak Ridge National Laboratory, 1991.

[12] D.R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.

[13] D. Nicol. The cost of conservative synchronisation in parallel discrete-event simulation. *Journal of the ACM*, 40(2):304–333, April 1993.

[14] P. Pierce. The NX/2 Operating System. In *Third Conference on Hypercube Concurrent Computers and Applications*, pages 384–390. ACM Press, 1988.

[15] David K. Poulsen and Pen-Chung Yew. Execution-driven tools for parallel simulation of parallel architectures and applications. In *SUPERCOMPUTING*, 1993.

[16] L. Prylli. Distributed simulation of parallel computers. Technical Report 95-12, LIP/ENS-LYON, 46 allee d'Italie 69364 LYON FRANCE, 1995.

[17] Eric Reiher, Herbert H.J. Hum, and Ajit Singh. Simulating networks of superscalar processors. Technical report, Centre de recherche informatique de Montréal, 1994.

[18] Bernhard Ries. The pararagon performance monitoring environment. In *SUPERCOMPUTING*, 1993.

[19] Bernard Tourancheau and Xavier-François Vigouroux. Parallel trace file management on top of PVM. In *PVM UG*, Oak Ridge, TN, May 1994. University of Tennessee, Knoxville.

[20] M. van Riek and B. Tourancheau. The trace-formats that are used in picl, paragraph and gpms. Technical Report 92-02, LIP – Ecole Normale Sup rieure de Lyon, 1992.

[21] M. van Riek, B. Tourancheau, and X. Vigouroux. The massively parallel monitoring system (a truly approach to parallel monitoring). In G. Haring, editor, *Performance Measurement and Visualization of Parallel Systems*, Moravany, CZ, October 1992. Elsevier Sciences Publisher.

[22] P. Worley. A new PICL trace file format. Technical Report TM-12125, Oak Ridge National Laboratory, October 1992.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399