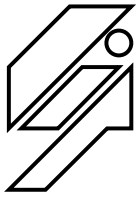
Distributed simulation of parallel computers.

Loïc Prylli



Laboratoire de l'Informatique du Parallélisme

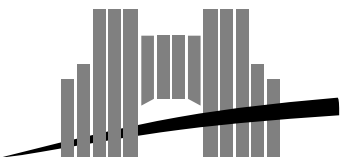
Ecole Normale Supérieure de Lyon
Unité de recherche associée au CNRS n°1398

Distributed simulation of parallel computers

Loïc PRYLLI

May 1995

Research Report N^{O} 95-12



Ecole Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : (+33) 72.72.80.00 Télécopieur : (+33) 72.72.80.80
Adresse électronique : lip@lip.ens−lyon.fr

Distributed simulation of parallel computers

Loïc PRYLLI

May 1995

Abstract

Our work deals with simulation of distributed memory parallel computers. The tool we realized allows to take an application written for say an Intel Paragon and run it on a workstations cluster by just recompiling the code. The hardware of the target machine is simulated so that the behavior of your application on the workstations is identical to a native run on the simulated computer (except for total execution time!). We present here this tool as well as a mathematical analysis of the conditions required about the simulation host, the simulated host and the application to be able to distribute efficiently the simulation.

Keywords: simulation, parallel computers, performance analysis

Résumé

Nous nous intéressons ici à la simulation distribuée d'ordinateurs eux-mêmes parallèles. Nous avons réalisé un outil permettant d'exécuter une application développée pour une machine parallèle (par exemple le Paragon d'Intel) sur un réseau de stations de travail par le simple biais d'une recompilation. Les composantes matérielles de la machine cible sont simulées, de sorte que le comportement de l'application est identique à celui obtenu par une exécution native sur la machine simulée (hormis le temps total d'exécution !). Nous présentons ici cet outil ainsi qu'une analyse mathématique des conditions sur la machine simulante, l'application et la machine simulée qui permettent de distribuer efficacement la simulation.

Mots-clés: simulation, ordinateurs parallèles, analyse de performances

Table of Contents

Introduction

As parallel computers have become more widely available, a lot of tools have been developed for them. These tools try to fill several needs: studying application performances, load-balancing and effective exploitation of parallelism and problems related to the communications network (contention, links utilization).

What we propose is a new tool that hopefully has its place in this domain. It allows to simulate a MIMD computer on a workstations cluster and to run real parallel applications with just recompiling the source code. We try to be as general as possible so that we can simulate a wide range of parallel computers and provide as well several application programmer interfaces (including Chorus, NX, PVM and MPI). A virtual clock (representing the time on the simulated computer) is maintained and 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), or by time measurements made inside the application that will reflect the virtual clock.

What is new is our approach is the parallelization of the simulation itself. This avoids a severe limitation of traditional simulators: the limited amount of memory of one workstation and the elapse time. By using a network, we can now deal with larger problems and decrease as well the time of execution of the simulation.

This document will present the tool as well as a theoretical study of the efficiency we can hope to achieve by distributing the simulation.

1 Backgrounds

1.1 Why is simulation useful?

To develop and optimize parallel applications, the most used methods have been instrumentation and runtime tracing. There are a lot of different ways to do this, instrumentation can be done more or less automatically, and there are a lot of different approaches to visualization. The main problem with observational analysis is neutrality. It is actually difficult to instrument and collect data without perturbing 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 efforts have been made, in software as well as in hardware to ensure as much neutrality as possible. Another problem that come with instrumentation and tracing is that, like workstations, parallel computers tends to become multi-tasked and multi-user. So the study of the application will be disturbed by events external to the application

So although observational analysis is useful, simulation can be more suited in some cases and of moreover both can be combined to insure neutral observation. Parallel simulation allows also new features :

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

1.2 Parallel programming on workstations

Some tools already exist to ease the development of parallel applications on a cluster of workstations. But they aim at executing as fast as possible a parallel application without bothering simulating a peculiar MIMD computer.

PVM [GBD+94] : This is a library that allows to developp parallel applications using a message passing paradigm. It is available either on networks of workstations or on real parallel machines.

NXLib [SLL93] : You can use this environment to execute applications written for the Paragon. But, although it does provides the application programmer interface of the paragon it doesn't simulate the behavior of this machine.

Trollius [BDV94, Bur88] : Trollius is both an operating system and an API for parallel programs. You can use the trollius environment on workstations with "Trollius-LAM" which provides also the MPI interface.

1.3 Tools to simulate a parallel computer

Some tools already exists to simulate execution of a parallel application on some peculiar hardware. The application is transformed either manually or automatically, in 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.

Protéus [Bre92, BDCW91] : This tool allows a quite realistic simulation. First at compile time the cost of each basic block of the application is evaluated and some code is inserted to be able to take it into account during simulation time. Then the application is sequentially simulated with a simulation engine, which is responsible for:

- maintaining a virtual clock.
- sharing the CPU of the simulating machine among the different virtual nodes simulated.

- simulating the communications.

EPPP simulator [RHS94] : EPPP is a complete programming environment, including a simulator based on Protéus. The evaluation of computation time had been improved by doing the compile-time analysis of each basic block of the assembly code generated for the target machine.

Beside those cited, some other works have been done :
EGP-sim[PY93], **Tango**[DGH91], **PEET**[GNS+92].

2 Specification of the simulation tool

Our work aims at providing a tool able to simulate a wide range of parallel computers, so it has to be quite parameterizable. Moreover, we want to be able to simulate real applications, that is to say, applications that deal with a great amount of data and also that are quite demanding on CPU power. Consequently we want to parallelize the simulation (of a machine that is itself a parallel computer) and this is probably the main specificity of our work. Last, we want it to be portable.

Nevertheless, even if we want to deal with big applications, the main objective is the accuracy of the simulation and the time necessary to do the simulation comes only second. Of course, it has to be enough reasonable to test applications with massive parallelism.

Trace generation is a major feature of the simulator. The generated trace files can then be analyzed with classical tools (cf. 6).

This work is original for two reasons, until now there is no general tool that allow simulating a parallel computer at the application level that's to say answer at the question: "how much time this application will take on this computer?". Well, others work are in progress, but either dedicated to only one machine, or they focus on very high accuracy at of simulation at the hardware level and so they are restricted to simulate "toy" applications. The second specificity is that we are the first one to parallelize the simulation, which eliminate the necessity of a huge computer to deal with real applications, in fact there is some theoretical reasons (cf.§4.1) that probably prevented other people to do so before. But we will show how we can circumvent these in our specific case.

2.1 Modeling of an architecture

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

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

- The power of the computing processors,
- The topology of the communication network and the routing strategy,
- The protocol used for communication (circuit-switched, store-and-forward, worm-hole),
- The bandwidth of the links,
- The switching time of the routers,
- The timings associated with the initialization of a transmission (without taking CPU time),
- The CPU timings associated with a transmission,
- The packet size if packeting is used,
- The flit size in the case of worm-hole routing,
- And some ad-hoc options to deal with peculiar buffering schemes.

All these parameters are simply given by the user in a configuration file, that is read at the beginning of the simulation.

Let's look more closely at these different options:

Power of computing processors: Our choice is quite simple, the computing processor is modeled just by one scalar (for instance the Mflops as given by the **Linpack** benchmark). This choice is a limitation, the relative timings of different CPU depends also of the type of computation. An approach like the one used in the EPPP project allows more accuracy but is beyond the scope of our approach.

Nevertheless for a wide range of problems, an estimation of the computation time for a processor obtained with the scaling of the computation time regarding the simulator's processor gives an acceptable accuracy, this is particularly true for processors of the same family (for instance RISC). And in fact, the lack of precision introduced is not greater than those obtained when you change from one compiler to an other for the same processor, or when you just change compiler options. The table 1 shows the worst case where we compare efficiency of some processors in very distinct domains: with the Linpack benchmark (dense linear algebra), the whetstone benchmark (aggregate melting of integer and floating-point operations), and the dhrystone benchmark (integer operations).

	linpack	whetstone	dhrystone
Alpha	100	100	100
Mips	20	28	19
Sparc	19	26	20
i860	37	38	24
RS6K	130	77	126

Table 1: processors comparison, the numbers give an power estimation (unit not meaningful, Alpha is given as 100 for reference)

Topology and routing: Topology and routing are indissociable so they are represented by only one parameter. Of course in some cases you can have several choices of routing strategy for the same topology. The currently available topologies are: *ring, mesh and hypercube*, but provision has been made so that it's very easy to add new ones. Classic routing strategies have been implemented, XY routing for the mesh or e-cube routing for the hypercube. But more complicated routing like Hot Potato routing can also also be implemented easily.

Communication network protocol: The simulator can simulate different routing protocols: worm-hole with a specified flit size, circuit-switched or a theoretical idealized protocol with which you don't really simulate the physical transmission on the network but instead you assume that the transfer time follows the law $\beta+L\tau+d\alpha$ where L is the length of the message and d is the number of links between the two and β,α,tau are numerical constants characterizing the network.

Also it takes into account as well half duplex link or full-duplex links.

Numerical constants for the network: There is little to say about that, you can simply notice that we have taken into account separately the operations that require the computing processor and those that occur in parallel with it.

Packet size: This optional parameter allows to signal that a message is cut into fixed size fragments before transmission on the network.

Others parameters: It is necessary to do a compromise between having a generic tool dealing with a wide range of parallel computers and simulating as accurately as possible a peculiar machine. That's why we introduce some ad-hoc parameters that are useful to take into account features specific to one machine, for instance:

- The iPSC860 buffering protocol to insure that room is available on the destination node before sending a message [Int].

- The behavior of the Paragon that has also a peculiar buffering protocol. [PR94].

2.2 The available application programmer interfaces

It is important to deal with existing applications but they are written for different machines with different APIs (CMMD, NX, PVM, MPI, ...). So we decided to provide several APIs.

For historical reasons, we first provide a library conforming to the Chorus interface specification. For practical reasons, the other interfaces have been implemented on top of the Chorus ones with the help of some functions to insure that the introduction of a intermediate level doesn't introduce any discrepancies in the simulation.

At this time, the Chorus and the NX interfaces are available.

2.3 Related issues

In this part we will not consider the message-passing facilities as part of the operating system. Otherwise saying, the "operating system" will name the functionality provided to the application by the execution environment, exception made of the message-passing library.

We have seen so far how we model an architecture and what range of applications we can simulate, it appears that we have left the modeling of the operating system of the target machine. This is something important so far, as the operating system can influence dramatically a machine performance in some cases. On a multi-tasked node, it will determine the scheduling policy. On some systems it manages a virtual memory space with eventually paging or swapping. Given the number of different operating systems, it is not reasonable to take into account all possibilities in a general tool. So we adopt a conservative approach, we made some simple choices (for instance, we assume a simple round-robin scheduling if there is several threads or processes on one node) we assume no swapping or paging. Anyway, our approach seems acceptable for several reasons:

- Most parallel applications don't rely on paging because generally to provide acceptable performance, it is necessary that all code and data can be stored in physical memory. So system impact due to memory management will generally be negligible and it appears reasonable to ignore it in simulation.
- Most parallel applications using message-passing doesn't rely much on the operating system except at load time and for input-output operations. In particular, most of the time there is only one application process per node

which eliminates problems related to scheduling policies. As regards input-output operations, that is true that our simulator will not give any hint for applications where such operations are predominant over computation.

- On machine that allow only a single application at a time, the operating system has hardly any influence on the behavior of the application. On multi-user machine which tends to spread, the operating system introduces "random" perturbations, but in the scope of our project we are not interested by reproducing this kind of perturbations. 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 that the real one is most of the time an advantage.

3 Simulation by discrete events

We will present here the method of simulation we used, an overview is first given at an abstract level and then the notion introduced will be further explained just after.

3.1 Structure of the simulation engine

First we need a set of variables that represent the global state of the simulated computer at a given time. Simulation progress by way of transitions, one transition modify the variables to represent a new state of the machine, and increase the time by a specific amount, so in fact 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, the nature of this event and the time at which it will occur (called the time-stamp of the event). Events are unavoidable to simulate a complex system where some parts evolves separately and interacts at certain times, they are in fact a representation of these interactions as will show the examples below.

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 changes, one at the beginning of the action and one at the end, the evolution of the real system in between should not 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 consist basically in the following algorithm:

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

At each stage of the algorithm, the virtual time of the simulation is given by the time-stamp of the event e we are dealing with.

3.2 Simple example of simulation

Let's took a simple example with the following situation: we have a computer with three nodes A, B, C on a row. Node A and node B each compute something and then send the result to C. We will took in this section an oversimplified model. To communicate three steps are necessary, first we must acquire successively the links necessary to reach the destination node. If a link is free, we can acquire it instantaneously. Then the communication time is constant. Let's suppose the computation of A lasts 2 unit of time and the one of B lasts 1 units, the communication lasts 2.

Here follows the different stages of the simulation:

A state	B state	$A \leftrightarrow B$ link	$B \leftrightarrow C$	Q
-	-	Free	Free	(init computation,0)
Busy	Busy	Free	Free	(end B,1) (end A,2)
Busy	Idle	Free	Active	(end A,2) (end $B \leftrightarrow C$,3)
Idle	Idle	Active	Active	(end $B \leftrightarrow C$,3)
Idle	Idle	Active	Active	(end $A \leftrightarrow C$,5)

3.3 Choice of a time scale

When doing discrete event simulation you have generally to choose either a discrete time model where time increases by multiple of a specified time unit or a contiguous time model where the time can take any value. In our case, in a parallel machine every node are asynchronous, so there is no appropriate discrete time (as for instance a global clock signal) on which to base the simulation, so we choose a contiguous time model (in fact this can be considered as an extreme case of the discrete time case where the chosen unit would be very small compared to the typical intervals of time used in the model).

Simultaneous occurrence of two events: In a contiguous time model, the probability of two events occurring simultaneously is zero (unless a strong dependency exists between them, but this is taken into account in the model). So we choose to ignore this kind of situation if it occurs in our simulation (events can always be ordered). But in a computer there are a lot of things based on clocks, so for some hardware parts, all events occurring in one period should be considered simultaneous. We don't take into account this kind of thing because it will be inconsistent with the level of our model.

3.4 Representing the state of the machine

After having presenting the general concepts, we will now describe more precisely how we apply them in our case.

The representation of the state of a machine depends a bit of its architecture, but we can roughly decompose the state of all machines entering in the classification of section 2.1 as follows:

- For each link of the communication network, we store his state: active or idle, and at the attached routers the list of messages blocked, waiting the availability of this link.
- For each message in transit on the network, we must know the list of links it is currently monopolizing.
- The real code of an application process is in fact executed in a real process with a library that redirects message-passing calls to interact with the simulation engine.
- For each node, we maintain a list of requests blocked waiting for some resources to be processed, a list of 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 note that we can extend it easily, for example if the routing function is not fixed then the router must maintain some additional state, etc ...

3.5 Some examples of events

We will now describe the most representative types of events that can occur and the corresponding actions that must be taken when treating them (they depend on the architecture we are simulating). We will use again the notation Q to design the set of events managed by the simulation engine (cf. §3.1).

Treatment of an application send request. This one is generated when an application process reach a "send" library call. If the packet-splitting option is on, the message is first cut and transformed into several requests each corresponding to a new event of type internal transmission. It is also here that we eventually deal with special buffering protocols.

3.5.1 circuit-switched case

Treatment of an internal transmission request. Let s be the source site, t the sending date, d the target site. The following work:

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

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

The action we described here is atomic if we don't need to "sleep". If we have to wait for the availability of some resources, then we insert the information necessary to do the rest of the processing into the queue of blocked requests associated with the resource. It will be processed when another event frees the resource.

Treatment of a routing event. Let n be the node on which the message is arriving.
If n is the final destination of the message then try to acquire on this node the resources needed for delivery like in the case of a transmission event, compute the time at which the delivery will be terminated and insert in Q an event of type "end of transmission".

If n is an intermediate node, then compute the next node n' with the appropriate routing function. Then wait for the availability of the resource corresponding to the link between n and n', add a switch time to obtain the final date at which to insert a event of type routing in Q (for the node n') .

Treatment of end of transmission. All resources used for this message are freed (in particular the links along the path between sender and receiver). This can lead to execute some actions that were waiting for the corresponding resources. Depending on the required simulation precision, the resources can be freed successively instead of all at one time, in this case several events are in fact generated. The state of the destination node is changed so that the new message is taken into account, if an application process was blocked waiting for such a message, it is resumed.

3.5.2 wormhole case

Treatment of internal transmission request. As for the circuit-switch case, we acquire the resources necessary to reach the second node (basically a link), we compute the date at which these resources will be ready to use, and we insert an event of type routing in Q. Of course, the timings constants will generally not be the same as for the circuit-switched case.

Treatment of a routing event. If the end of the message has left the source node, free the last link used by the tail of the message.
Then we execute the same operations as in the circuit-switched case: if we are not at the final destination, compute the next node with the appropriate routing function, acquire the necessary resources and insert a event of type routing in Q. If we are at the final node, insert a event of type transmission end phase.

Treatment of transmission end phase. If the message is entirely arrived on the final node then the actions taken are similar as for an end of transmission in the circuit-switch case. Else do one step of transmission: the message progress according to the flit size; if the queue of the message has left the source node, free the last link at the tail of the message; insert an event of type end of transmission in Q at the date corresponding to the delay of achievement of the previous operations.

4 Parallelization of the simulation

4.1 Constraints

We have seen that there was a simple sequential simulation algorithm. There several points before starting with parallelization:

- It is first quite obvious that we cannot parallelize the evaluation of one transition because they either consists in very simple action or in execution of the code of an application process which is by nature sequential.
- Then we have to investigate how we can proceed in parallel several events. The problem that arises is the problem of coherence. The simulation algorithm must ensure that the results of the state transitions are exactly as if they have been processed sequentially in chronological order.

The coherency constraint implies that all sites of the simulating machine have generally to be synchronized at each time jump which in practice will prevent any actual parallelism. We will examine different methods in the following subsections to remove partially this constraint.

4.2 Using the latencies of the simulated machine

We will use some topological knowledge of the machine we are simulating. After decomposing we can associate a localization to every event. Then we will use the fact that there is a minimal latency time of propagation between the different components of the simulated machine. More formally, if we represent the components of the machine by a connected graph (the vertices will represent the computing nodes, and the routers of the machine), we will design the different components by $s_1, s_2, \ldots$. Then we will have the following property that is a consequence of the hardware latency:

- For each edge (s_i, s_j), there exists a minimal latency $l_{i,j}$ such that every event generated when evaluating a transition on s_i and associated with the site s_j has a time-stamp greater than $t_i + l_{i,j}$ where t_i is the time-stamp of the current transition.

We can now generalize the latency to any couple of components, even if there are not directly connected by saying the corresponding latency is equal to the one of a path of minimal latency between the two components. graph (the closest path). Let h_x design the time-stamp of an event x. At each stage of the simulation algorithm, we can choose for the next transition to compute any event $e \in Q$ verifying $h_e < h_{e_0} + l_{s,s_0}$ where e_0 is the event with the smallest time-stamp and s, s_0 are the vertices associated to e and e_0. The simulation algorithm become non-deterministic and so has an inherent potential for parallelism.

In 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 deal with all transitions associated with this particular site. On each process the following algorithm is executed:

1. Let be t the smallest time-stamp among the events owned locally.
2. For every other site q, wait that 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 of two factors: the ratio of the typical interval between two events on the same site against the typical latencies between sites.

In practice in our case, the only sites where the local state progress independently of the other nodes are the compute nodes; every other sites (routers, links) are essentially driven by external events (that mean events not generated on the same site), that roughly mean that there is a synchronization at each event with its neighborhood. That means the latency here will be limited, an upper bound for the efficiency being the diameter of the graph if the communication costs in the simulation host are null, in practice there is no parallelism exploitable at this level between such site. Although it seem we cannot gain much parallelism here, the features described here can be useful when used in conjunction with the algorithm described below.

4.3 Master slaves organization

There is little hope to parallelize the simulation of the communication network (in fact we could parallelize it using predictive action but that would not be an effective solution in our case), but we can try to conserve the parallelism inherent to the application by distributing the computations done by the different application processes. One node of the simulation node will take in charge several nodes of the simulated machine. There will be a master process that will simulate the communication hardware and will deliver in order (chronologically with regards to the virtual time) the messages that are exchanged on the

network to the applications process (that will be called the slaves). Each slave inform the master of the virtual time reached by the nodes it simulates.

4.4 Cutting the computation phases

On the compute nodes, we have a local evolution of process between two calls to the message-passing library. But at this point if we stay with our clean model of §3.1 then a computation phase is just considered like one transition and the problems of §4.1 will not be solved. But a computation phase is something that can be decomposed. If we do so then in the middle of a computation we can inform the other simulation sites what simulation point we reach so that they can eventually start other computation phases that would proceed in parallel with the rest of the current computation. The problem is that there is no obvious decomposition, it would be too costly to decompose at the instruction level and that will cause problems to evaluate computation time. On the contrary, if we decompose the computation with a too heavy granularity, we will loose any parallelism. The 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 duration before starting a computation phase on an other node, it interrupts the computation phase, the slave answers if it has reach or not the critical point and if not sets a timer so that it can 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 several processes (representing a virtual node) in the middle of computation phases. So from now we will assume that computation phases can be dynamically cut into several parts.

4.5 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 consider here that the *simulating* host has the following properties:

- Computation phases can be cut into infinitely small parts without overhead.
- Communication fully overlap with computation.
- The average latency of small messages between a slave and the master is β.

4.5.1 Algorithm of the master

Our simulation model has changed a bit since §3.1 with the introduction of interruptions in computation phases, but we have still a set of events Q that would be completely managed by the master.

The algorithm on the master is then:

1. Let e be the first event in Q.
2. Wait that every application process are inactive (waiting for a message or some information of the master) or that we know it has reach a point later than the time-stamp of e (more precisely less than the time-stamp of e to which we subtract the latencies described in §4.2).
3. Take into account the first event of e. That can result in starting a computation on a slave.
4. Go to step 1.

4.5.2 Algorithm of the slave

Let $N = \frac{V}{R}$ be the 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 representing the virtual time reached by one of the nodes managed. The vector t represents the advancement of the simulation on one slave.

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

The algorithm is composed of the following actions:

- Let S be the set of nodes not blocked, advance in their computation uniformly: that means run the node of S with the smallest t_i. We supposed we can switch with an infinitely small granularity between nodes of S, that means there will be a subset of the t_i that will increases at the same time.
- When a node becomes blocked, we inform the master that it should inform us when the corresponding t_i has been reached globally.
- Messages received from the master unblock a node.

4.5.3 Efficiency analysis

Now we will study the efficiency of this algorithm. It will be done on a virtual application that we define as follows:

- Compute phases have average duration of time M.
- All process are busy at any time.

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

We now consider a particular slave. Let try to determine the conditions necessary to avoid idle states in the simulation. We consider a cycle beginning at a time where we receive a message from the master unblocking a node. Let t be the state of the slave as defined in 4.5.2. The critical path to go to the next cycle will consists of several repetition of the following:

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

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

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

As practically we will have $V \gg R$ the remaining condition is $M > 2R\beta \Leftrightarrow R < \frac{M}{\beta}$.

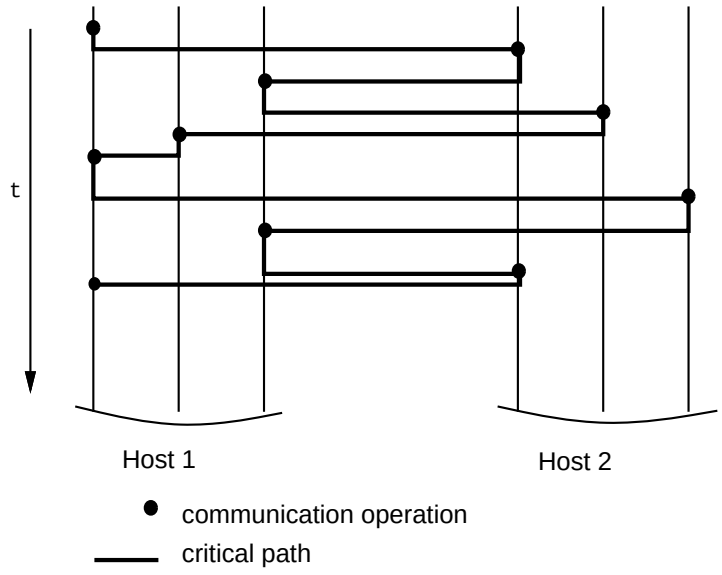


Figure 1: Critical path representation

Now, we must justify that just average considerations lead us to a valid result. For that at a beginning of a cycle, on a particular slave, let W_a denote

the amount of computation already done "by anticipation", that means 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 $W = \frac{MV}{2R}$. An example of what represent W_a and W is given in figure 2.

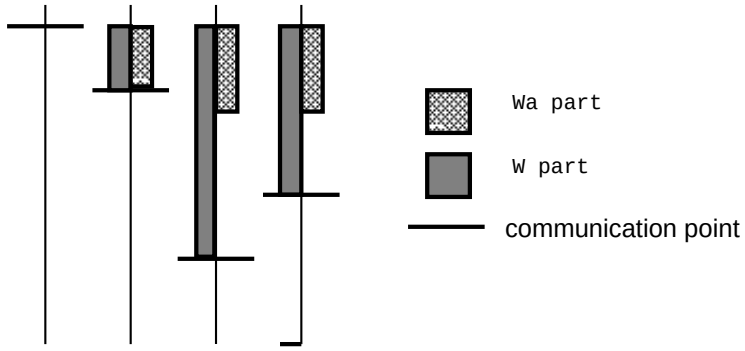


Figure 2: W and W_a at beginning of a cycle on one slave

In the computation of the critical path length, we said that the time from a blocking point to the next global blocking point was $\frac{M}{V}$, for that we assumes that at the time the master send the unblocking message, it knows also the point in time of the next blocking point. A sufficient condition for that is that for all slaves, the last W_a were greater than M. So now we have to look at what happen 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 will increase. When the W_a are greater than M and as we must have $R < \frac{M}{\beta}$, the critical path between two cycles is smaller than M so the W_a decrease. So in average the W_a values will stabilize themselves somewhere below M. If $V \gg 2R$, the maximum values for W_a is several times M which ensure that it is valid to reason with average values. Note that average values are taken among different nodes *at one time* and not along time. If all nodes do small computations at the same, then the efficiency will drop.

Last we have to see if we can generalize our analysis of a special kind of application to more general cases. The point is to see how idle times in the virtual case influence the performance of the simulation. Let now consider that M will represent in fact the interval between two computations starting points, so M will be decomposed into a computation part M_c and a idle part M_i. Of course we always consider average values. The algorithm does not change at all, simply there are some node that are idle instead of busy and then the new formulae for W id now $W = \frac{V}{R} \times (\frac{M}{2} - M_i)$ so we need that $M_c > M_i$ and preferably $M_c \gg M_i$ (W must several times greater them M). The condition on V and R becomes $\frac{V}{R} \gg 2\frac{M}{\frac{M}{2} - M_i}$. All the other reasoning remain valid and

then the efficiency will be optimal at the same condition $R < \frac{M}{\beta}$.

4.6 Simulation on a realistic machine

We just study the algorithm on a ideal machine, the strong assumption was that we could share one CPU of the simulation host between different logical processes representing the nodes of the application with an infinitely small granularity. In reality what we can do is to switch between threads or processes with a granularity g depending of the system and the implementation (logical processes representing nodes can be managed by several unix processes or by several threads into a single unix process).

So we could take again the previous study with g, the only important modification is during the calculus of the critical path length, when we take into account the time necessary to reach the next global point. That was $\frac{M}{V}$, that must be now replaced by $\max(\frac{M}{V}, g)$, so then we have the supplementary condition $Rg < M \Leftrightarrow R < \frac{M}{g}$. To ensure the validity of this limit W_a must now be greater than $M + gR$, but his change is not very important for the stability of W_a as long as $R < \frac{M}{g}$ which implies $M + gR < 2M$.

4.7 Limitations due to the application

All the efficiency considerations we discussed until now didn't take into account the time necessary to transfer the data messages between the different simulated nodes. We just spoke about the messages necessary to the coherency of the simulation. It is quite obvious that if an application can't be run on the simulating host because of the bottleneck of the communication with a normal message-passing library like NXlib, MPICH or PVM (cf. §1.2), there is no hope to compensate that by adding the coherency constraints and ordering of the target machine simulation. So what we determined are the conditions at which the simulation could be done with the same order of speed than with a simple message-passing library on the simulating host.

5 Implementation presentation

For portability and simplicity reasons, our environment is built on top of PVM.

There will be three kind of PVM tasks: the "slaves" noted S that will manage the different virtual nodes, the "main simulation engine" noted M (like master), that will simulate the communications on the virtual hardware. Last there will be a certain number of application processes noted T (like thread), each representing a virtual node. A slave and its attached nodes will all be run on the same CPU.

These different entities will be interconnected (cf. figure 5) by several kinds of communication channels:

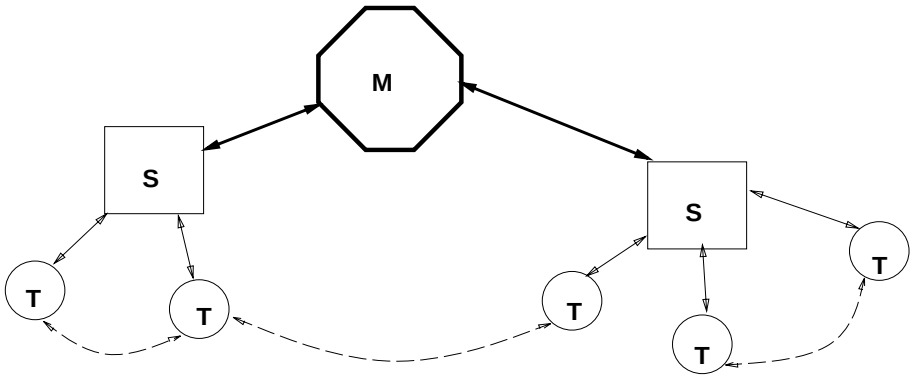


Figure 3: overview of the simulator organization

- The channels M↔S allow the slaves to cooperate with the master to allow progression of the simulation.
- The channels T↔S allow a slave to dispatch the CPU between the differents threads and gathering application messages information that is further 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 transit by the master.

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

6 Trace generation

The simulation can be exploited by two means. On one hand the timings measured by the application are virtual times (identical to those that would have been measured on the target machine) and so that allows to analyze superficially the application. On the other hand, there is the possibility to generate 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 correspond to a neutral observation of the execution (what is almost impossible with a real machine!).

We choose the PICL[vRT92, Wor92] trace file format that allows us to use Paragraph[HE91] to displays the result of the simulation.

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

7 Validation of the simulator

In this part, we will present several results obtained with the simulator. To do these tests, we took several programs written for the iPSC860 and ran them both on the real iPSC860 and with the simulator.

The first test (figure 4) is a "ping-pong" test. It is a simple test to verify that the parameters are correct for the target machine.

Then we took two algorithms that comes from the SCALAPACK package that deals with numerical linear algebra operations on parallel computers. [ABD+91].

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

The second program (figure 6) is doing a QR decomposition, then also solve a linear system. As for LU, we indicate each phase time.

This results shows that the simulator has a good accuracy. In our case we were simulating on some workstations with sparc processors. The difference between the real and simulated execution time is essentially due to the non constant power ratio between this two processors but nevertheless we can see that it does cause a small bias.

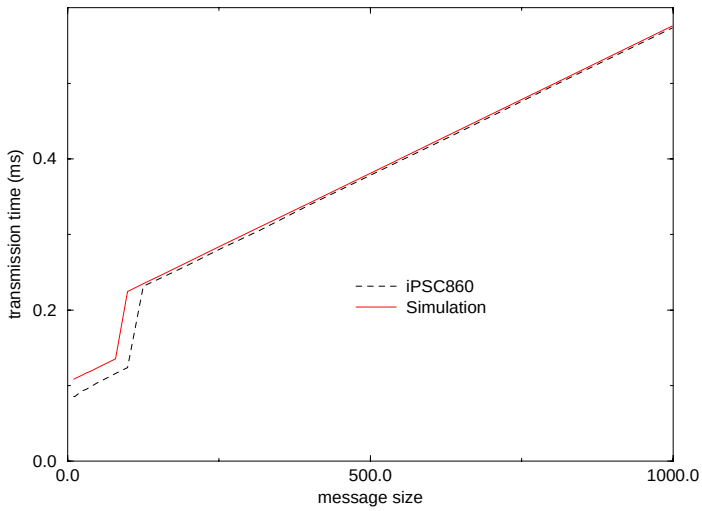


Figure 4: ping-pong simulation for the iPSC860

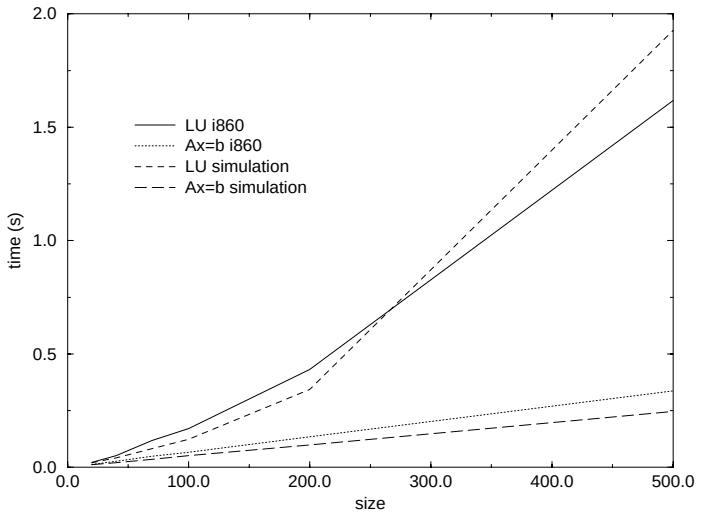


Figure 5: LU simulation on 16 nodes

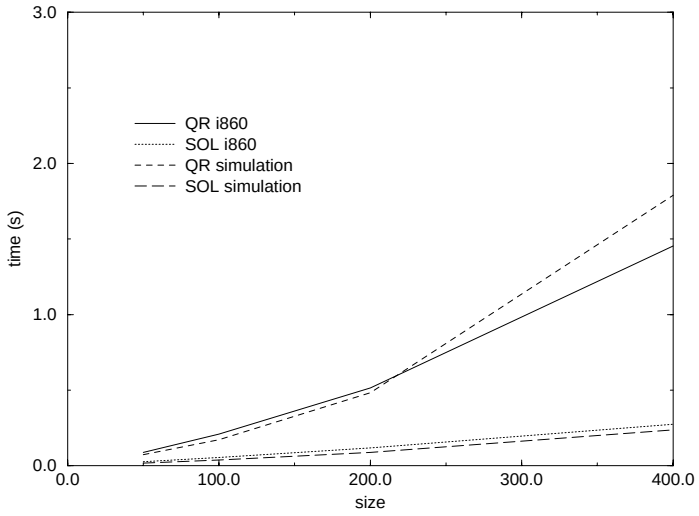


Figure 6: QR simulation with 16 nodes

8 Conclusion

Even at this stage of our work, we obtain 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 the design and study of a parallel machine.

The tests that have been done with both simulation and native execution seems to show that a good accuracy can be obtained.

Some work is in progress to allow the use of the simulator with different APIs.

The other interest of this work is the theoretical study of the parallelization efficiency. We are now able to characterize the type of simulation that can be done in parallel depending on the granularity of the application and the parameters of the simulation host.

References

[ABD+91] 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.

[BDCW91] E. A. Brewer, C. N. Dellarocas, A. Colbrook, and W. E. Weihl. Proteus: A high-performance parallel-architecture simulator. Technical Report MIT/LCS/TR-516, Massachusetts Institute of Technology, Laboratory of Computer Science, September 1991.

[BDV94] Greg Burns, Raja Daoud, and James Vaigl. *LAM: An Open Cluster Environment for MPI*, 1994.

[Bre92] 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.

[Bur88] Gregory Burns. Trillium operating system. In *Third Conference on Hypercube Concurrent Computers and Applications*, pages 374–376, 1988.

[DGH91] H. Davis, S. Goldschmidt, and J. Hennessy. Multiprocessor simulation and tracing using tango. In *ICPP*, 1991.

[GBD+94] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Mancheck, and Vaidy Sunderam. *PVM: Parallel Virtual Machine.* Scientific and Engineering Computation. MIT Press, 1994.

[GNS+92] 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.

[HE91] M. T. Heath and J. A. Etheridge. Visualizing performance of parallel programs. Technical report, Oak Ridge National Laboratory, 1991.

[Int] Intel Corporation. *iPSC/2 and iPSC/860 Source Code Product : Internal Product Specification.*

[PR94] Paul Pierce and Greg Regnier. The paragon implementation of the nx message passing interface. In *SHPCC*, 1994.

[PY93] David K. Poulsen and Pen-Chung Yew. Execution-driven tools for parallel simulation of parallel architectures and applications. In *SUPERCOMPUTING*, 1993.

[RHS94] Eric Reiher, Herbert H.J. Hum, and Ajit Singh. Simulating networks of superscalar processors. Technical report, Centre de recherche informatique de Montréal, 1994.

[SLL93] Georg Stellner, Stefan Lamberts, and Thomas Ludwig. *NXLIB User's Guide*. Institut für Informatik, Technische Universität München, 1993.

[vRT92] 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.

[Wor92] P. Worley. A new PICL trace file format. Technical Report TM-12125, Oak Ridge National Laboratory, October 1992.