# Paradys:
# A scalable infrastructure for parallel circuit simulation

J.-L. Lafitte, D. Mlynek
Integrated Systems Laboratory
Swiss Federal Institute of Technology
Lausanne

February, 2001

**Abstract**

*We detail the design of a scalable infrastructure, called Paradys, developed for parallel circuit simulation. Early measurements of its scalability (some 0.9x of parallel efficiency) are encouraging signs to measure on larger parallel configurations as well as to envision its application for simulation of deep sub-micron technology. This good scalability is, in great part, achieved thanks to a dynamically managed memory gap, called SHMC++, reducing the number of memory accesses in the given shared memory environment. Actual measurements show an increase, due to ShMC++, of the overall speedup of Paradys parallel infrastructure going from 14% to 78% depending on the original memory access rate.*

## 1 Introduction

This paper is to reflect the design of a scalable infrastructure, called Paradys, developed for parallel circuit simulation. Early measurements of its scalability (some 0.9x of parallel efficiency) are encouraging signs to measure on larger parallel configurations as well as to envision its application for simulation of deep sub-micron technology.

From the very beginning, the main goals assigned to Paradys are to:

1. develop algorithms for dynamic load balancing on parallel computers and adaptive algorithms for merging subcircuits in correlation with the instantaneous distributed power of the processing elements.

2. develop parallel algorithms for the detection of subcircuits with an analog or digital behavior in ULSI.

# 2 Overall Structure

The overall design of Paradys is lead by the following basic assumptions:

**scalability** is the main ingredient: at time of trade-offs it should be the winning part,

**generality** should always be kept opened,

**portability** : though developed on a concrete hardware configuration (an IBM SP2, herein known as SP), everything is done in order to easily port the code unto other parallel (or distributed) platforms.

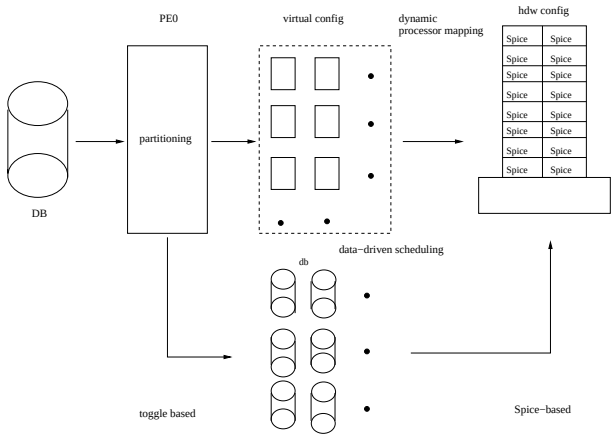


Figure 1: Paradys overall data-driven design

As depicted in Figure in 1, the different components of Paradys infrastructure are the followings:

**Paravis** is related to Paradys design component and runs in a specific node (PE0) of parallel configuration: it allows to display the events in the parallel infrastructure as well as interacting with the running simulation going on. A dedicated paragraph gives a thorough description of this graphical interface.

**Data Base** is a simple data base containing Spice sentences (also known as netlists). It is splitted (i.e. partitioned) in ”mini db’s” (i.e. kind of sub-netlists) by the Partitioning phase of the infrastructure.

**Partitioning** is developed based on the Toggle algorithm (cf. [11], [1]) and is intended to produce a picture of the different subcircuits that can be simulated in parallel - the set of *ultimate grains* - along with the way they are linked together.

These *ultimate grains* are potentially re-gathered together based on the

**actual SP configuration selected:** Given the ratio $\frac{\text{grain (nbr of subcircuits)}}{\text{PE (nbr of PE's)}}$ estimated at minimum around 5, there is an upper bound for the number of grains per available PE; the lower bound is being driven by the number of transistors per grain, which is empirically determined between 100 and 200 (indeed, smaller subcircuits trigger too high a latency in - the available - Spice processing).

**feedback detection:** Feedbacks are common place in circuits. In Partitioning, when the maximum granularity is searched, they are not considered. Later on, during the Regrouping phase, they are detected: whenever they can be inserted within a subcircuit, it is done so in regrouping the concerned original grains. This regrouping based on feedback is limited by the size of the resulting subcircuit as well as by the number of resulting subcircuits. Though they complexify the overall process, *large feedbacks* are kept outside the subcircuits (i.e., they are visible to the Paradys infrastructure and handled accordingly). Further developments are considered in order to handle them in a more general manner.

**analog vs digital criteria:** Contiguous original ultimate grains of the same nature (analog or digital) are also potentially regrouped, while always keeping in balance the number of subcircuits and their respective sizes.

This *granularity adaptation* -or *regrouping* - is handled within PE0 as the final initialisation phase.

During the Partitioning and Regrouping phases, a *configuration pre-tayloring* is going on in all the other available PEs, checking the different parameters (or adapters) of each PE so they can be set at their optimal value (for both latency and bandwidth) for the actual parallel part of Paradys processing.

## 2.1 Data-driven scheme

During the design phase of Paradys, an alternate architecture has been thorougfully considered. Indeed, despite the Paradys objectives of being scalable up to several thousand PEs, *realpolitik* obliged to consider small and medium size (from 10's to 100's PEs) configurations.
In an attempt to address also those small configurations, a distributed model has been considered for a while. Given the complexity and the programming load it would have triggered for the project, this model has been discarded in favor of a unique data-driven model addressing the long term objective of Paradys (i.e. 10.000 PEs) which is described in the following paragraphs.

### Data-driven scheduling

At the beginning of Paradys processing, a given processing element (PE0) has a specific role, in the sense that it runs the initialising phase of the simulation, followed by the partitioning phase (follow-on versions of Paradys are considering to parallelize also this latter, exploiting space-filling curve algorithms such as proposed in References [3], [2]), producing the *ultimate grains* and the way they are linked together. Further refinement known as *Regrouping* is also handled to fit the different criteria, as expressed in the previous paragraph.

From then on, this specific role of PE0 is terminated, it then plays the role of the Paravis PE (as described in a specific paragraph) for the rest of the simulation. Each other PE of the available configuration has an instance of the scheduler, which is hence distributed. Such a distributed structure of the scheduler, avoiding a single point of failure as well as a *hostnode* structure as advocated in Reference [9], is expected to achieve the scalability objectives required of Paradys.

This distributed scheduler memorizes the db's mapping, and keeps track of the advancement of the simulation based on the contents of a piece of shared memory, containing the overall subcircuits status/mapping. Each block in the Shared Memory is either a SC_CB (for SubCircuit_ControlBlock, a C++ object) representing the different signals required by a given subcircuit to be simulated, or a control block representing the electrical nodes. Those SC_CBs reside in the shared memory and form the base upon which the firing rules are applied in order to build up an overall data-driven scheduling; those firing rules are being triggered by properly overloaded C++ operators.

**Load Balancing**

The simulation time for each subcircuits being quite variable, it is expected that the conjunctions of

1. the *ultimate grains* approach (regrouped according to the available SP configuration, feedback detection and Analog-vs-Digital criteria) provided by the partitioning component,
2. the data-driven scheduling scheme

provides the adequate load balancing required to achieve the expected Paradys scalability.

The way the SC_CB's are chained, represents the static ordering of the subcircuits simulations. The way by which they are dynamically (i.e. actually) ordered is mainly due to three factors:

1. relative speed of simulation execution for each subcircuits
2. PEs availability
3. subcircuits simulation convergence.

## 2.2 Loads and Convergence

The fact of partitioning a circuit into subcircuits, decouples each subcircuit from one another. This creates a situation where each subcircuit is 'electrically isolated' from the others. To compensate this, waveform relaxation technics such as those developed and proposed by A. Ruehli [11], [1] and A. Vachoux [12], allows the existence of 'virtual inputs' (also called *loads*) in addition to the 'normal inputs' of each subcircuit. Each subcircuit to which the outputs of a given subcircuit A are connected, triggers the existence of an equivalent so-called *load*. At the time of the first iteration, all loads are set to zero. As the first iteration is progressing among the different subcircuits, proper

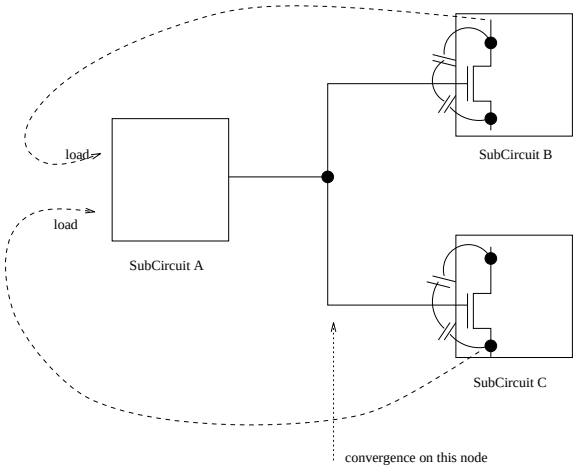


Figure 2: Loads and convergence

loads are reflected as inputs to the appropriate subcircuit(s). This whole process triggers the need to iterate through the simulation of each subcircuits, until each one of them has converged (i.e. the *loads* have stabilized their effects on each of them).

> A subcircuit is flagged as **converged** when all its input signals and loads have converged **and** when all its output signals have converged.

## 2.3 Overall design

After the initialisation phase (i.e. Partitioning and Regrouping), the main components of Paradys involved in the infrastructure and depicted in Figure 3, are as follows:

**NetList Manager:** the existence of both inputs and loads to be managed by Paradys between subcircuits implies the existence of a permanent process, called a *NetList Manager*, present in each PE. The NetList Manager is in charge of maintaining actual values of inputs and loads (as they become available) within the netlists provided to Spice. It has been preferred to have such a permanent process in order to smooth the communication load between the different PEs. The other term of the alternative would have been to wait for the inputs and loads to be available to *pull* them unto the appropriate PE, which would have triggered a peak in communication load between the PEs.

**Simulator:** in a first approach, Spice is run as a distinct application in each node of the SP configuration, as each ”mini db” can be perceived as an actual DB (i.e. netlist) by each Spice instance.

It should be noted that in the case of a SP node made of n processors (i.e. a **SMP**), n instances of the simulator are simultaneously running in the given SP node. In such a case, we are dealing with n logical PEs, each of them being implemented as a pair of Distributed_Scheduler / Netlist_Manager processes, described later.

**Resulting Waveforms** are produced by the simulation of each subcircuit, as dictated by the data-driven scheme, and are kept on the PE where the given iteration has just been exercised. The PE number is set into the SC_CB representing the given subcircuit. When time comes of scheduling the next iteration (i.e. next time when the firing rule is to be applied for the given subcircuit), this PE number is considered by the distributed scheduler as the preferred PE to be selected, hence providing maximum data locality.

At design time of such an infrastructure, a multi-tasking problem is to be solved: how multiple processors should be running an application in parallel?, question known as the question of *task scheduling* (i.e. how to assign specific tasks to specific processors).
Four different factors are to be considered in order to pick up the best suited method:

- Processor Utilization
  The different subcircuits, results of the circuit partitioning, have to be assigned to the different PEs such that the workload is balanced in order to obtain the optimal processor utilization.

- Communications
  Communications between PEs are important for performance since they may cause processors to wait for each other results. This is accomplished in trying

  – to avoid unnecessary waiting times

  – to reduce as much as possible the interprocessor communications.

  – to avoid any communication bottlenecks.

- Recovery
  The scheduler supervises the processes activity. When a simulator process or a PE fails, the scheduler has to be informed so it can take the appropriate actions in order to recover the failure.

- Scalability
  A system is said to be scalable if it can handle the addition of users and resources without suffering a noticeable loss of performance or increase in administrative complexity. In our case it means that the simulator should be performing as well for small circuits as for very large ones and with few PEs as with many.

To reach our objectives, a data driven model is chosen. The scheduler is distributed over all PEs and the data necessary for the scheduling are stored on a piece of shared memory.

The scheduling data (in the shared memory) contains the overall subcircuits status/mapping and a queue is maintained containing the subcircuits ready to be simulated.

Two processes are used to manage the scheduling (each pair of processes is present in each PE, cf. Figure 3):

- the Distributed Scheduler: either it is waked up or it picks up a subcircuit from the ready queue; it gets the necessary input (netlist) from the mini db's maintained by the Netlist Manager and provides it to Spice for simulation
- the Netlist Manager: it updates the netlist files with the values of the new waveforms resulting from the simulations just run; it updates the corresponding data in the subcircuits status/mapping and appropriately issues the firing rule.

This separation of tasks between two different processes enables a better distribution of interprocessor communications.

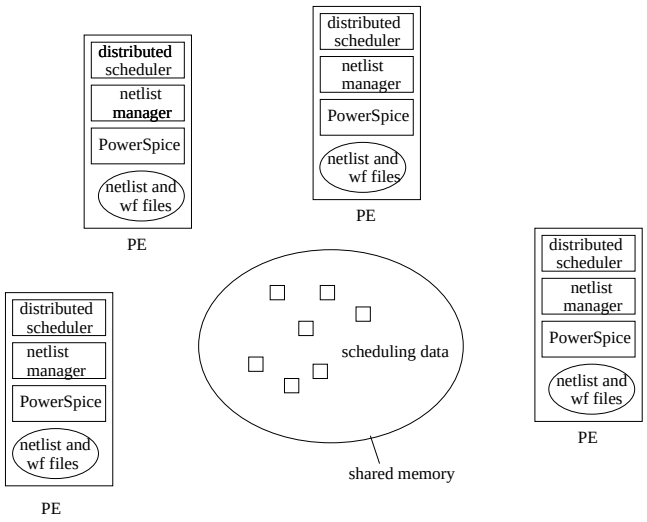


Figure 3: Overall View

### Scheduling data

Each subcircuit is represented in the shared memory by a SubCircuit Control Block (SC_CB). For each SC_CB, the input and output signals are represented respectively by an input list and an output list. Each signal of the input list points to the SC_CB, where this signal comes from (predecessor subcircuit). Each signal of the output list contains a list representing the subcircuits where this signal is sent to (successor subcircuits in the successor list). Each element of the successor list points to the corresponding SC_CB and contains a representation of the loads calculated in the successor subcircuit.

The subcircuits which are fired for simulation, are represented in a ready queue (if no PE's are available when the firing rule is issued). Each queue element (ready element) points to the corresponding SC_CB (see Figure 4).

### Convergence Calculation

The convergence calculation is done by the Distributed Scheduler. It compares the output waveforms of current iteration with those from the preceeding iteration. The current iteration waveforms

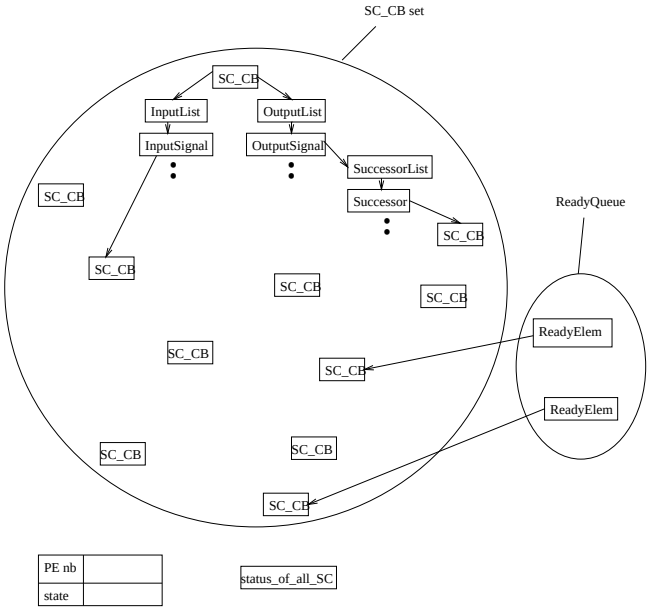


Figure 4: Shared Memory Overview

have just been calculated by the local Spice simulator and stored into a file. The preceeding iteration waveforms are also stored locally in a file (it was carried over from a remote PE if necessary by the Netlist Manager during the subcircuit simulation). For each output node, the following equation has to be true for convergence:

$$|\text{volt(i)} - \text{volt(i}-1)| < \epsilon$$

where i is the iteration number and $\epsilon$ the convergence criteria as specified by the user.

**Scalability**

Beyond the overall design, the following points in Paradys infrastructure are given considerable attention for achieving the expected scalability:

- when the Distributed Scheduler or the Netlist Manager searchs a particular SC_CB in the set in order to read or write information
- when the Distributed Scheduler searchs an element in the ready queue with particular criteria in order to select it for simulation

To reach the needed scalability, a novative structural approach is being exploited as described in paragraph 3. In the search for scalability, we also used at the beginning of the project, some classes from the IBM Class Library (key sorted Set and Relation). Those latter classes, though of good quality, are not portable. When the Standard Template Library (STL) has become widely available, their usage was converted to C++ iterators over containers.

# 3 Dynamically managing the memory gap

The growing gap between processor and memory has been exacerbated by the way we are using computers: in order to obtain an actual data for a given instruction, it can take several, if not a lot of, different memory accesses, dependent upon the way data are structured.

As part of the Paradys project, we were lead by design to build up a shared memory. As an assist to the Paradys infrastructure written in C++, and transparent to this programming model, the author developed a software layer in order to access objects resident in the required shared memory. Such a software layer, coined **ShMC++**, is a first step implementing, in Paradys parallel environment, the device proposed in references [7, 8].

## 3.1 Structured data across the wall

As argued in so many instances [5, 4, 6], the requirements are numerous to handle dynamic complex data structure. However current designs (either at the processor or at the system level) do not address the point in a satisfactory manner, witness vector processors or High Performance Fortran inadequacies to exploit processor speed for these kinds of data.

To give a concrete case of the matter, it has actually been measured that on a superscalar processor, able to potentially run 5 instructions per cycle, the achieved rate for High Energy Physics (HEP) data is, in fact, 0.8 instruction per cycle[1]: data not being accessed at the necessary rate because too many indirections have to be handled along the von Neumann bottleneck.
Profiling of HEP applications[10] shows it clearly: in such environments, 50% of the time is spent handling indirections in order to access the actual data.

Given the discrepancy between microprocessors performance increase per year and DRAM much slower speed increase per year[2], this leads to an enlarging gap, the effect of which is much more exacerbated by dynamic complex data structures accessed in a non-local manner.

Encouraged by preliminary results obtained by software simulation of the CERN Benchmark Jobstream[8][1], we developed a concrete case around a shared memory access. Such an access is important for parallelism in multithreading environments as well as in the perspective of a NUMA architecture.

We now describe how the principles proposed in [7, 8] are implemented as an assist in order to dynamically handle access to Paradys shared memory.

[1] Personal communication of Sverre Jarp, Atlas experiment, CERN.

[2] respectively estimated by F. Baskett at 80% and 7% (in his keynote address at the International Symposium on Shared Memory Multiprocessing, April 1991), they have recently been rated at 60% and 10% by Maurice Wilkes (cf. reference [13]).

## 3.2 ShMC++ motivations

In the context of accessing Paradys shared memory, one needs to reduce the latency for both read and write. This implementation of the principles proposed in [7, 8], shows that a dynamically managed access to complex data stuctures (ShMC++) residing in a shared memory, can substantially help the overall latency.

Control blocks implemented as tiles[3] within the shared memory are representing the subcircuits as well as the different signals required to simulate them. This set of control blocks resides in shared memory as a set of tiles and forms the base from which firing rules are triggered, forming Paradys overall data-driven scheduling. Such control blocks being C++ objects, the firing rules are ideally triggered by properly overloading a C++ operator[4].

## 3.3 ShMC++, principles

As part of Paradys design, the availability of a shared memory was assumed. As the local available hardware (an IBM SP2) doesn't embody such a mechanism, we developed a home grown software-based distributed shared memory. As an assist to the Paradys infrastructure written in C++[1], and transparent to this programming model, the author developed a software layer in order to access objects resident in the shared memory. Such a software layer, coined ShMC++, is an implementation, in a parallel environment, of the proposed device[7, 8] along the gap between processing elements and a shared memory.
ShMC++ allows the parallel application to define the required tilings as well as to program their access.

[3] We take the term *tile* in its general meaning as an element filing up with some others, the shared memory content.

[4] C++ overloading, allowing to freely modify operators semantic during read/write access to objects, has extensively been exploited in ShMC++ as a mean to develop new type of object access, while remaining transparent to the programming model.

[1] C++, even as standardized, does not offer a semantics in accessing shared objects either from different threads within a process or from different processes running in parallel - either in an SMP or an MPP environment.

Figure 5 depicts the split introduced by ShMC++:

**(a)** a classical way to access shared memory, where data and structures are identically accessed within the shared memory;

**(b)** the segregation, introduced by ShMC++, between the algorithmic part written in C++ and the shared memory access; the part labeled "ShMC++" in the given Figure contains the structures (such as all the pointers) of the C++ objects being accessed, the shared memory containing only the actual data (such as the *integer*, *Flag* or *SC_status* fields in Example 1) of the C++ object being represented.

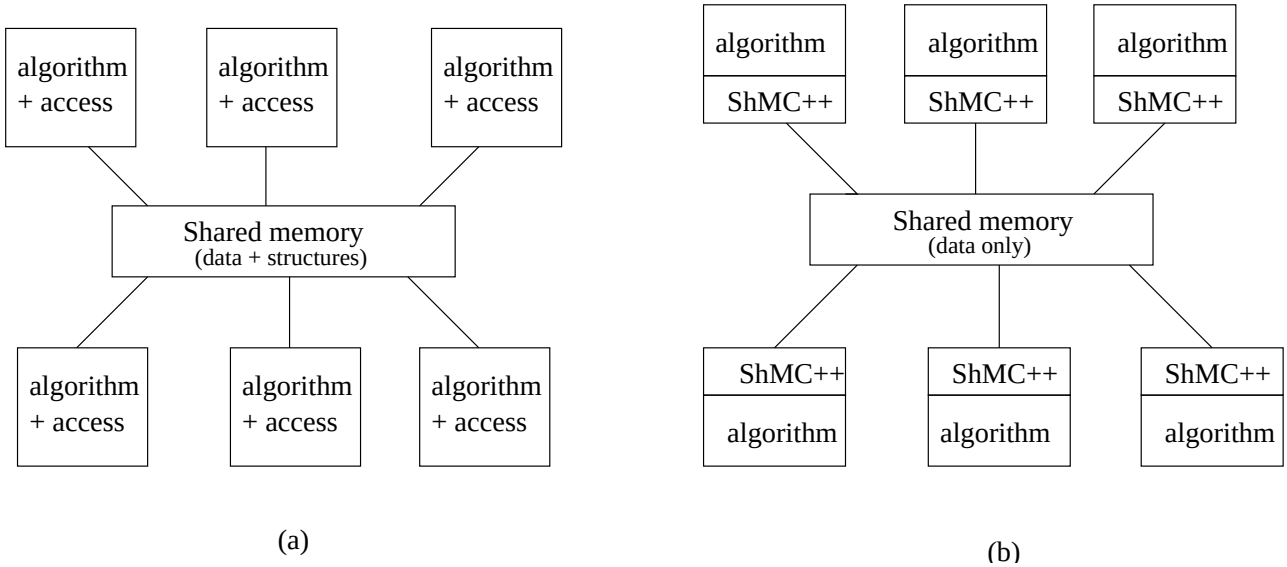


Figure 5: Shared Memory access from C++: (a) classical, (b) with ShMC++

When each participating process starts, their ShMC++ part contains no reference (i.e. knowledge) of the shared objects. As such process starts to reference the shared objects, the structures of these latters are automatically built up for the given process in the local node, which potentially gives for each process the same perspective of the shared memory content. When referenced, the structures of the shared objects are built up in the ShMC++ part, the fields corresponding to the actual data (such as the *integer*, *Flag* or *SC_status* fields in Example 1) instead containing the address to be used for accessing such fields in the shared memory.

Example 1 is an instance of a C++ object (called SC_CB) used in Paradys:

```
#ifndef SC_CB_H
#define SC_CB_H

#include <iostream.h>
..............

//ShMC++ Shared Memory tilered
//
//ShMC++ 1eSC_CB   <--> C++_Class
//ShMC++ 1eInputs   :=  ( Input_free,  0,  Nb_inputs  )
//ShMC++ 1eOutputs  :=  ( Output_free, 0,  Nb_outputs )
//ShMC++ 1eTile    <--- ( 1eSC_CB, 1eInputs, 1eOutputs) gcat

 ..............................
//-------------------------------------------------------------
class SC_CB
{
private:
SC_CB();                          //.Default constructor

//ShMC++  Id
  int         SC_id;              // subcircuit identification (>=1)

  int         last_iter_PE;       //.PE number on which last
                                  // iteration was done, or on which
                                  // actual simulation is running
  int         sched_iter_num;     //.Number of iterations already done
  int         signals_counter;    //.Number of signals yet to be received fo
                                  //    the SC to be ready

  Flag        all_inputs_ready; //.ON: all inputs are ready
  Flag        all_loads_ready;  //.ON: all loads are ready
  Flag        first_conv_cond;  //.First convergence condition:
                                  // all predecessor SC have converged

  SC_Status   sched_status;     //.Subcircuit status

  InputList*  inputs;           // list of input signals
  OutputList* outputs;          // list of output signals

.......................
#endif SC_CB_H
```

The ShMC++ sentences are implemented as C++ directives that specify how the data structure of the given object should be tiled in the shared memory. The sentences which contain the following verb:

$< -- >$ allow to identify the space *1eSC_CB* with a C++ class structure,

$:=$ allow to define two spaces and their respective contents,

$< - - -$ allow to define the tile (*1etile*) as being made of the concatenation (*gcat*) of three different spaces (*1eSC_CB*, *1eInputs* and *1eOutput*),

The directive **//ShMC++ Id** specifies that the following variable (namely in the given case, *SC_id*), must be used as a unique identifier for the tile to be built.

Such ShMC++ sentences allow to direct how to build up tiles in the shared memory that only contain the actual data for each instance of the given C++ object. The set of pointers building up the structure of normal C++ objects are not placed in the shared memory but are dynamically managed in each separate participating process.

Overall, a given shared object can actually be represented by the contents of two locations:

1. its structure known and defined in the ShMC++ parts of a certain number of processes (hence, at a given moment, there are n such structures, n being $<=$ the number of participating processes),
2. the set of actual data it is made of, uniquely stored and maintained in the shared memory without any structure.

# 4 Paravis

Paravis is the visual component of the Paradys infrastructure. It keeps running during the whole simulation in a given node, called PE0. Its main objectives are to display information regarding the simulation progress.

It is made of several panels, which are continuously updated in order, for the user, to follow how his(her) simulation is going on. By *continuously* it is meant 'as soon as the information is brought into PE0'.

There are three quite different flows of message going on in Paradys:

1. MPI messages between the PE's, explicitly issued by the C++ programs making up Paradys,
2. other MPI messages used in implementing the home-made distributed shared memory,
3. information messages brought to Paravis in PE0, for logging and display purposes.

Given that the built-up Paradys infrastructure is dependent upon available levels of software/tools which do not allow to prioritize such quite distinct flows of messages, the flow of information messages to Paravis arrives when possible, and the panels are refreshed accordingly.

A new level of software should be soon available, based on DPCL defined by the 'ptools' consortium, that might help enhance that aspect.

The main Paravis panel is reproduced in Figure 6 as a snapshot after 10+% of the subcircuits have converged, as shown in its **Convergence** field. At the top of the panel

- number of **Simulators** available,
- number of **Sub-Circuits** being handled,
- and different **Times** are displayed.

The four different graphics of the main Paravis panel are as follows:

**Speedup(p):** shows the speedup reached so far.

**Time:** shows, in seconds, the time so far used by:

**Tparad:** the Paradys infrastructure,

**Tidle:** the amount of time the Distributed Scheduler has so far been idle, waiting for SC_CB to be ready.

**Thruput(p) & Convrate(t):** respectively the number of subcircuits being currently simulated and the number of subcircuits already converged.

**Sub Circuit(s) ready and Simulator(s) idle:** the instantaneous number of subcircuits in the ready queue, and its opposite, the number of PE idle, waiting for a subcircuit to be ready.

At the top of the main panel, three pull-down menus are available, the **Iterations** one being displayed as a snapshot in Figure 7: The first column shows the different iterations going on (**Id** from 0 to 2) and for each of these iterations, the number of subcircuits already converged (**done**), with some graphical representations of the iteration progress displayed on the right side.

Equivalent information are available for **Simulators** and **Queues**. Another panel displays the different subcircuits, those already converged being red-colored and the user having the interactive possibility to dynamically change the netlist of a given subcircuit and to restart the simulation at the required level, depending upon the connections of the modified subcircuits with other surrounding subcircuits.

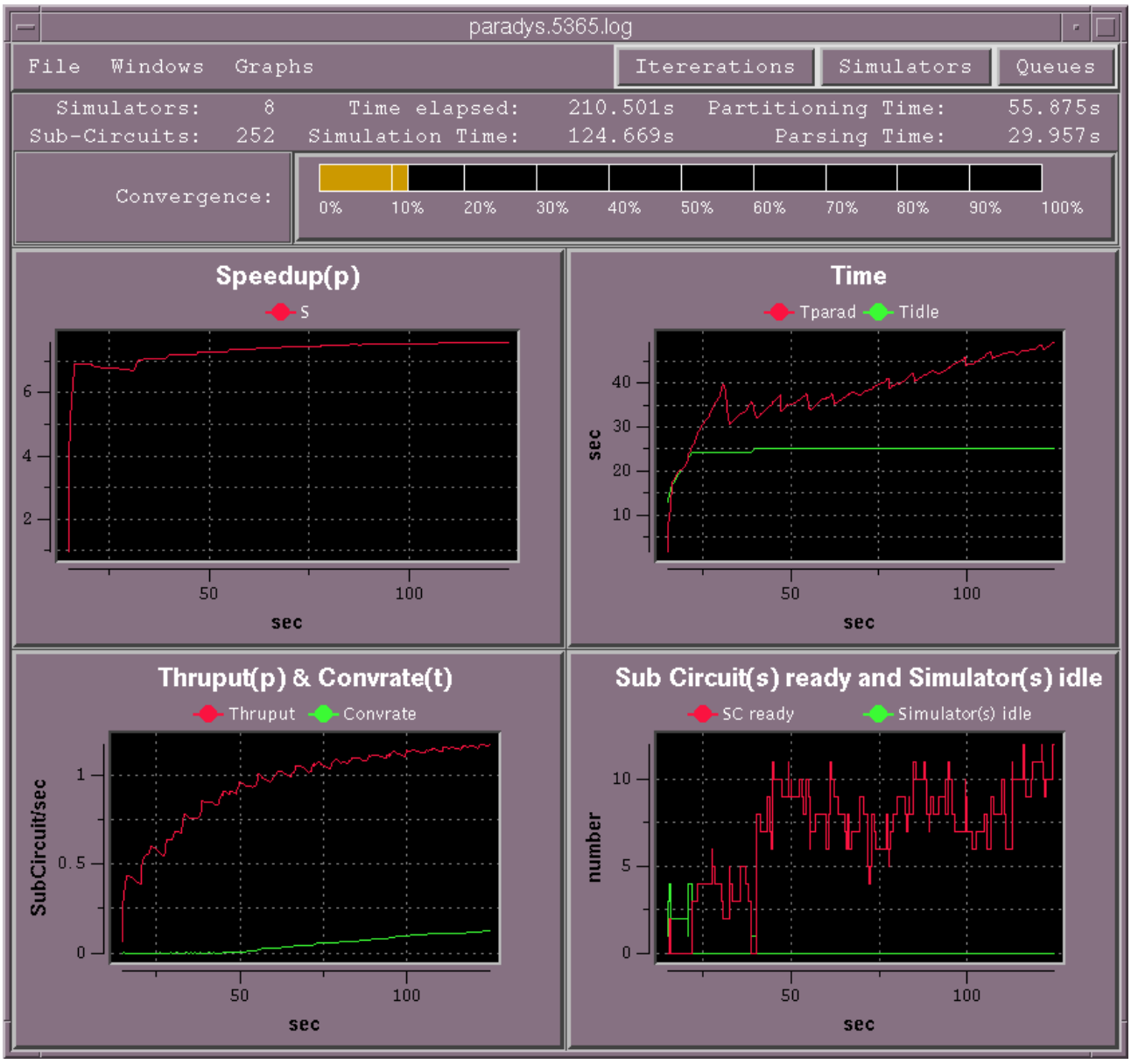


Figure 6: Paravis main panel: at 10% of the simulation


Figure 7: Paravis panel: levels of iteration

# 5 First measurements

Our implementation has been exerciced for different circuit size, on 2, 4 and 8 PE's configuration. A PE (Processor Element) is made of two UNIX processes working together, a Netlist Manager and a Distributed Scheduler The actual configuration on which the different runs (three times each for reproductibility checking) have been carried, is an IBM SP2 with 5 nodes connected by an SPS switch. 4 nodes are identical UP's, each running one PE. The 5th node is a 4-way SMP node capable of handling 4 PE's. A 2 PE's configuration is thus made of 2 UP's nodes, a 4 PE's of 4 UP's nodes, a 8 PE's of the SMP node and 4 UP's nodes.

The software distributed shared memory is evenly distributed on the considered PE's. Its structure is quite simple, first of all for lack of programming resources in developing a full blown distributed shared memory, but also the considered shared memory is filled up once at the end of partitionning, after what its structure does not change, only values are modified. We are thus not dealing with a sophisticated memory management scheme, cache coherency, etc... and its possible interference with the measured device.

The Paradys infrastructure is written in C++, compiled with gcc 2.95 19990728. The link-edit is done with *mpcc* a specific script provided with that machine. At run-time, the global variables related to the parallel environment are: MP_LABELIO=yes, MP_PROCS=n, MP_TRACELEVEL=0, MP_EUILIB=ip, MP_EUIDEVICE=css0.

For each circuit size, measurements are carried in an equal stable environnment.

## 5.1 ShMC++ effects on speedup

Given the major objective of the project (scalability), scalability points of control are placed in Paradys code in order to measure their respective effects. In this perspective, we detail, in this paragraph, effects of ShMC++ as measured on the Paradys infrastructure.
In those specific (i.e., ShMC++) scalability measurements, partitioning is kept under control in

order to obtain subcircuits of the same size: in consequence, we are expressing the circuit size in term of number of subcircuits. Each subcircuit is made up of around 120 similar transistors, so that the complexity of a subcircuit simulation is identical and does not interfere with the targeted objectives of the measurement.

For each circuit size, measurements are carried in an equal stable environnment, for both normal case (i.e., *w/o ShMC++*) and with the proposed device (i.e., *with ShMC++*). The difference of the presence of the studied device can be characterized by a decrease in the number of shared memory access, e.g. the memory pressure, or the traffic on the von Neumann bottleneck.

| Number of subcircuits | w/o ShMC++ | | with ShMC++ | | memory pressure decrease | speedup increase |
|---|---|---|---|---|---|---|
| | memory pressure | speedup | memory pressure | speedup | | |
| 125 | 48 | 1.59 | 16 | 1.89 | -32 | +19% |
| 252 | 49 | 1.64 | 16 | 1.92 | -33 | +17% |
| 500 | 45 | 1.64 | 15 | 1.92 | -30 | +17.5% |
| 1008 | 45 | 1.65 | 15 | 1.93 | -30 | +17% |
| 1960 | 43 | 1.69 | 14 | 1.94 | -29 | +15% |
| 4410 | 32 | 1.7 | 11 | 1.94 | -21 | +14% |

Table 1: ShMC++ effect on speedup (2 PE's)

The experiment data show the variation in speedup calculated here as $S(p) = \frac{T(1)}{T(p)}$ with T(1) being the elapsed time to simulate a given circuit on 1 PE, and T(p) the elapsed time to simulate the very same circuit in parallel on p PE's.

| Number of subcircuits | w/o ShMC++ | | with ShMC++ | | memory pressure decrease | speedup increase |
|---|---|---|---|---|---|---|
| | memory pressure | speedup | memory pressure | speedup | | |
| 125 | 96 | 2.47 | 32 | 3.68 | -64 | +49% |
| 252 | 95 | 2.65 | 32 | 3.74 | -63 | +41% |
| 500 | 93 | 2.66 | 31 | 3.75 | -62 | +41% |
| 1008 | 85 | 2.71 | 28 | 3.80 | -57 | +40% |
| 1960 | 71 | 2.74 | 24 | 3.82 | -47 | +39% |
| 4410 | 54 | 2.84 | 18 | 3.85 | -36 | +35.5% |

Table 2: ShMC++ effect on speedup (4 PE's)

| Number of subcircuits | w/o ShMC++ | | with ShMC++ | | memory pressure decrease | speedup increase |
|---|---|---|---|---|---|---|
| | memory pressure | speedup | memory pressure | speedup | | |
| 125 | 247 | 3.83 | 82 | 6.86 | -165 | +78% |
| 252 | 228 | 4.06 | 76 | 7.2 | -152 | +77% |
| 500 | 197 | 4.24 | 66 | 7.25 | -131 | +71% |
| 1008 | 153 | 4.24 | 51 | 7.2 | -102 | +70% |
| 1960 | 125 | 4.46 | 42 | 7.46 | -83 | +67% |
| 4410 | 114 | 4.6 | 36 | 7.52 | -78 | +63% |

Table 3: ShMC++ effect on speedup (8 PE's)

**Note** that,

**1.** the speedup increase is directly correlated with the number of shared memory access per second (i.e. the *memory pressure*). It is coherent that the more the memory is sollicitated, the more benefit one can expect from the proposed model.

**2.** the measured effects are typical of the relative speed between the given processors and the time to access the shared memory (a home grown software distributed structure);

**2.** when parallel resources are fixed for a growing number of subcircuits (sizeup), the gain starts to stall, as if the system was choking;

**4.** further analyses are required in order to figure out a predictive model of ShMC++ benefits.

## 5.2 Paradys scalability

Based on ShMC++ benefits, measurements of Paradys scalability are displayed in Table 4 where the experiment data shown are as follows:

| Number of subcircuits | 2 PE's | | 4 PE's | | 8 PE's | |
|---|---|---|---|---|---|---|
| | S(p) speedup | E(p) efficiency | S(p) speedup | E(p) efficiency | S(p) speedup | E(p) efficiency |
| 125 | 1.89 | 0.94 | 3.68 | 0.92 | 6.86 | 0.85 |
| 252 | 1.92 | 0.96 | 3.74 | 0.93 | 7.20 | 0.90 |
| 500 | 1.92 | 0.96 | 3.75 | 0.93 | 7.25 | 0.90 |
| 1008 | 1.93 | 0.96 | 3.80 | 0.95 | 7.20 | 0.90 |
| 1960 | 1.94 | 0.97 | 3.82 | 0.95 | 7.46 | 0.93 |
| 4410 | 1.94 | 0.97 | 3.85 | 0.96 | 7.52 | 0.94 |

Table 4: Paradys speedup and parallel efficiency

**speedup** calculated here as $S(p) = \frac{T(1)}{T(p)}$ with T(1) being the elapsed time to simulate a given circuit on one PE, T(p) being the elapsed time to simulate the very same circuit in parallel on p PE's,

**parallel efficiency:** taken as $E(p) = \frac{T(1)}{pT(p)}$ with T(1) being the elapsed time to simulate a given circuit on one processor, T(p) being the elapsed time to simulate the very same circuit in parallel on p PE's, with p varying from 2 to 4 to 8 for our case.

**Note** that,

1. there is a great imbalance of speed and connectivity between the nodes of the 8 PE's confguration (4 UP's and a 4way SMP node);
2. the 8 PE's configuration might be lacking *grains* (i.e. subcircuits) to be properly 'fed' and reach 0.96 or 0.97 level of parallel efficiency; in order to provide more *grains*, partitioning should be enhanced by exploiting such approach as spacefilling curves as proposed in Reference [3].

# 6 Conclusion

On the available configuration, measurements of Paradys scalability, (the main objective of the built-up infrastructure) give encouraging figures. Among different things, it shows that the memory wall in front of us can greatly be battered, specially in accessing the worst case: complex structured data, through the use of tailorable I-units. Indeed, making available an I-unit dynamically tailorable to complex structured data, allows to alleviate data accessing and to substantially narrow the memory gap.

This new approach, of dynamically managing the memory gap, gives substantial gains mainly in decreasing the traffic along the von Neumann bottleneck which Paradys takes advantage of in order to achieve its main goal: scalability.

# 7 Follow-on activities

As mentionned in the different parts of this paper, follow-on activities are considered for Paradys:

**Performance** enhancements could be reached when proper software would allow better tuning of the different flows of messages;

**Recovery** infrastructure available in the IBM SP2 (known as High Availability or Phoenix) has to be used only by *root* users ... who cannot then issue MPI messages;

**Large feedback** loops should be handled in a more general manner exploiting recursive data structures;

**Partitioning** should be enhanced by exploiting spacefilling curves as proposed in Reference [3].

**Dynamically managing** the memory gap should be pursued in developing and analyzing a novel hardware device in order to alleviate the memory wall problem.

# 8 Acknowlegments

The author wishes to thanks the members of the Paradys team who plainfully contributed to its success: P. Debefve for his advice as well as his sense of humor, C. Alexandre for successfully sniffing around, C. Dupuis for carving out the first C++ objects of this project, Y. Pizzuto for developing and testing a large portion of the code, D. Steffen for his (alas temporary) boosting effect. Remote and knowledgeable support has been greatly appreciated: T. Johnson, an old timer for this kind of approach, as well as A. Rühli, a gentleman on the matter. Overall, this project would not have existed without the realistic enthusiasm of Professor D. Mlynek.

# References

[1] T. A. Johnson A. E. Rühli, *Waveform-relaxation-based circuit simulation*, Tech. report, Research Report, IBM Yorktown, 1997.

[2] A.B. Kahng C.A Alpert, J.-H. Hsuang, *Multilevel circuit partitioning*, Design Automation Conference, 1997.

[3] A.B. Kahng C.J. Alpert, *Multi-way partitioninh via spacefilling curves and dynamic programming*, 31st Design Automation Conference, San Diego, CA, 1994, pp. 652–657.

[4] William D. Gropp and David E. Keyes, *Semi-Structured Refinement and Parallel Domain Decomposition methods*, Tech. report, Argonne National Laboratory, 1991.

[5] J. Knupe H. Fischer and C. Troger, *FIRE on BMW's nCUBE2*, Speedup (1992).

[6] Steven W. Hammond and Robert Schreiber, *Mapping Unstructured Grid Problems to the Connection Machine*, Tech. report, NASA Ames Research Center, 1992.

[7] J.-L. Lafitte, *On Structured Data Handling in Parallel Processing*, ACM Computer Architecture News **23** (1995), no. 3, 11–18.

[8] _____, *A Generalized Mapping Device to help the Memory Latency*, ACM Computer Architecture News **26** (1998), no. 5.

[9] J.-L. Lafitte M. Röthlisberger, *Hostnode vs hostless structures: a case study*, to be published.

[10] Eric McIntosh, *Benchmarking Computers for High Energy Physics*, Tech. report, CERN, september 1992.

[11] A. E. Rühli T. A. Johnson, *Parallel waveform relaxation of circuits with global feedback loops*, Proceedings of the 29th ACM/IEEE Design Automation Conference, 1992.

[12] Alain Vachoux, *Analyse temporelle de grands circuits intégrés mos par relaxation de formes d'onde*, Ph.D. thesis, Ecole Polytechnique Fédérale de Lausanne, 1988.

[13] Maurice Wilkes, *The Memory Gap and the Future of High Performance Memories*, ACM Computer Architecture News **29** (2001), no. 1.