# Fundamental parallel program issues to improve parallel computations

B. Lecussan, O. Poitou

*Supaero and ONERA/DTIM, 10. av. E. Belin, 31055 Toulouse cedex, FRANCE*

{lecussan, poitou}@cert.fr, http://www.supaero.fr/lecussan

**Abstract**

The parallel techniques to improve computer performances form the common base to the evolution of the processors and the modern information processing systems. Performance improvement of a multiprocessor led to consider two complementary actions to face latencies; first is latencies reduction by mechanisms of replication which will involve requirements in memory bandwidth, data coherence and consistency protocols, then is residual latencies tolerance by overlapping techniques of computation and communications. The computation/communication ratio is the essential parameter of the effectiveness of the multiprocessor computers; near future technologies makes it possible to carry out locally 10.000 sequential instructions during the necessary time to send the first bit of a message from a transmitter towards a receiver. It is probable that the gap between the cycle time of the processor and the memory cycle continue to widen then requiring techniques to decrease and tolerate latencies of memory accesses; consequently the pressure is put on the bandwidth of the memory hierarchy. The programmer must know the consequences of the parallel paradigms that it uses to distribute computation on a set of processors (or on a set of clusters of processors) and to ensure himself (herself) the parallel program efficiency. Whatever the technological development, the implementation models of parallel programming will remain confronted with the problems of the data accesses, the execution scheduling of the various computations and the synchronisation techniques. This paper will discuss the fundamental parallel program issues to improve parallel computations and presents an algorithmic technique that leads to a natural memory distribution among the computing nodes implying better locality and a lot less communications compared to conventional approaches.

## 1 Introduction

The parallel techniques to improve the computer performances form the common base to the evolution of the processors and the modern information

processing systems. The parallelism is found inside the processor, which is able simultaneously to treat on average 2.5 instructions per clock cycle (1Ghz) by exploiting several functional units. It is the domain of the processors designers and compilers manufacturers to generate codes exploiting these resources as well as possible. The mother boards of today computers contain two or four processors which can be managed effectively by a multithreaded operating systems to obtain an acceptable efficiency of the parallel computer; parallelism is then, essentially, with the responsibility of the operating system. The systems with more than eight processors and up to 48 are currently present in all high performance servers, for computations or data management. The management of this parallelism is then the responsibility of the user who must produce an effective parallel program to exploit these resources as well as possible. To understand the key factors of a parallel application performance on a multiprocessor we have to examine the time spent in each component of the computer to carry out several threads of instructions, namely the access to the data of the memory hierarchy and the co-ordination of the activities with the other processors. The execution time (1) of a sequential program can be summarised by the following formula:

$$Ts = N \times CPI \times tc + Tam \tag{1}$$

**N** the total number of instructions executed by the processor
**CPI** the average number of Cycle Per Instructions
**Tc** the processor cycle time
**Tam** memory hierarchy access time

On a parallel architecture, the execution time (2) includes five parameters, which are:

**Ti=Ni×CPIi×tc** the sequential computation time inside the i-th processor
**Opar(i)** the overhead introduced by parallel tasks
**Tam(i)** the access time to local memory hierarchy
**RTam(i)** the access time to remote memory hierarchy
**Sync(i)** the synchronisation overhead between parallel tasks

Then, the computation speed-up of the parallel program (p) obtained by a multiprocessor is:

$$Speed - up(p) = \frac{Ts}{\max\left(Ti + Opar(i) + Tam(i) + Rtam(i) + Sync(i)\right)} \tag{2}$$

The performance improvement for the single processor consists in reducing the access time to the hierarchy memory (Tam) and in improving the value of CPI by bringing more than one instruction by cycle in the processor (two, four even eight instructions per cycle); the effectiveness of a multiprocessor is more complex, the challenge covering three aspects:

(1) characteristics of the bare machine leading to a minimum computation granularity, the cost of synchronisation primitives, the memory bandwidth, the data coherence and consistence protocol management
(2) characteristics of the application summarised by the frequency of the exchanges, the load balancing on each processor
(3) definition of an analytical model which makes it possible for the programmer to predict the performances of the parallel application.[1]

The programmer must know the consequences of the parallel paradigms that it uses to distribute computation on a set of processors and to ensure himself (herself) of the parallel program efficiency. In shared memory this is difficult because all costs are masked, in particular data coherence and consistency protocol costs that can induce very significant latencies. In distributed memory, the problem is simpler but a remote access cost is such as the parallel program must produce the least possible number of messages, especially short messages.

For multiprocessors, the initial model PRAM becomes insufficient because it does not allow interacting with performance. The speed-up formula (2) shows that the output of the parallel machine is related to parameters that it is advisable to take into account at application design time and when writing the parallel program. The value of Opar(i) is incompressible and corresponds to the additional code with the sequential program to make it parallel; Tam(i) contains the latency of the access to the local data, Rtam(i) represents the latency of the access through a communication network and Sync(i) contains the synchronisation costs so that the parallel application is semantically correct. Performance improvement of a multiprocessor led to consider two complementary actions to face latencies:

(1) Latencies reduction by mechanisms of replication which will involve requirements in memory bandwidth and data coherence and consistency protocols
(2) Residual latencies tolerance by overlapping techniques of computation and communications which will involve exchanges of memory blocks with zero copy, anticipations of communication and switches to parallel threads [2].

The latency reduction can take place on three levels:

- Reduction in the access times to each level of the hierarchy memory and the communication
  The cache controller must answer very quickly to detect a cache miss and to reduce the latency of access at the higher level. The objective is to minimise the following formula:

$$Tam = Hitrate \times Hittime + (1 - Hitrate) \times Misstime \quad (3)$$

For first level cache with hit ratio higher than 95% the Miss time may be 20 times the Hit time and for lower-level cache it will still be an order of magnitude. This encourages the designer to increase the hit ratio in particular by increasing the size of the lines of cache and to improve data cache locality. The network interface must be strongly coupled with the node of computation and be designed to format, deliver and manage quickly the transactions on the network. Time to transfer N bytes from an emitter (E) towards a receiver (R) has four components:

$$\begin{aligned} T(n)E\_R = \quad & Overhead \\ & + RoutingDelay \\ & + ChannelOccupancy \\ & + ContentionDelay \end{aligned} \tag{4}$$

Overhead is the time taken by the processor to deposit a message in the communication interface.
Routing is the time to transmit the first bit of a message towards a receiver.
Channel Occupancy is a function of the communication medium, of the number of control information inside a packed and of the time to apply control signals to carry out an exchange between two switches.
Contention Delay expresses that one message may collide with others and contend for resources.
With the three first parameters the network can be seen like a simple pipeline with a cost of initialisation, a depth and a processing time per stage. The strategies of commutation and routing change the effective structure of the pipeline whereas topology, the bandwidth of each link and the fragmentation of the message determine the depth and time by stage. However a message can enter in collision with another and require shared resources (buffers). Application then represents an additional overhead that ensures that at every moment a channel is occupied by one and only one message. The application adds times to the basic routing time.

- Structuring the system to reduce the frequency of the high latency accesses
  The cache mechanism is the basic component to bring closer the data of the operator who consumes them. This mechanism exploits the space locality and the temporal locality of the programs. The localisation of the data is then uncoupled from its physical address; this architecture authorises a migration of the data adapted to a management of the dynamic loads of a set of processes.

- Structuring the application to reduce the frequency of the high latency accesses
  This point is the responsibility of the application designer: it consists to break up and allocate computation with the processor in order to reduce the communications and to improve the space and temporal localities [2].

By example, dynamic memory allocation and data structures construction as late as possible during the program computation is a way to be taken. Development of this technique on cases studies is explained in this paper. Part2 will present fundamental parallel program issues to improve parallel computer speed-up, part 3 and part 4 describes two case studies based on lazy construction of program data structures reporting basic results.

## 2 Fundamental parallel program design issues

The computation/communication ratio is the essential parameter of the effectiveness of multiprocessor computers; current technology makes it possible to carry out locally 10.000 sequential instructions during the necessary time to send the first bit of a message from a transmitter towards a receiver. The restrictive factor of the formula (4) is the *Overhead*. Certain predict that this value can pass to 20.000 quickly even 100.000 when the processors have several hundreds of million transistors. In 10 years, the processor clock frequency was multiplied by 10 and transistor number by microprocessor was multiplied by a factor 30; on another side DRAM cycle time was only improved of a factor 2. Thus, it is probable that the gap between the cycle time of the processor and the memory cycle continue to widen then requiring techniques to decrease and tolerate latencies of memory accesses; **consequently the pressure is put on the bandwidth of the hierarchy memory**. The interconnection bandwidth and the switches results in supporting exchanges with 1 Gigabits/s and up to 10 Gigabits/s, the optical fibre having to impose like the major support for high speed links, but these links are connected to I/O sub-systems which become the bottlenecks of the multiprocessor. Whatever the technological development, the implementation models of parallel programming will remain confronted with the problems of the data accesses, the execution scheduling of the various threads and the synchronisation techniques.

As the issue addressed by this paper is the efficiency of the distributed computer, the main parameters to be evaluated are the load imbalance of the parallel algorithm, the memory required by the solution on each processor and the speed-up gained by the parallel computation.

Let the parallel computation time $T_p$ expressed by the following formula:

$$T_p = \max_i t_i \tag{5}$$

where i indexes the set of computers and $t_i$ is the i-th computer computation time.

The parallel computer efficiency E using p processors is:

$$E = \frac{T_s}{p \times T_p} \tag{6}$$

where $T_s$ is the best sequential algorithm to solve the problem.

A minimum of the computation imbalance occurs when all computers complete their work at the same time. In this case, this minimum occurs at:

$$T_{\min} = T_{seq} + \frac{T_{par}}{p} \tag{7}$$

where $T_{seq}$ and $T_{par}$ respectively are the non-parallel and the parallel part of the computation time. This suggests an objective function to measure the effectiveness of any candidate solution $s$to any instance of the load-balancing problem. The quality of $s$ can be measured by the ratio of imbalance that it produces and can be expressed by the following formula:

$$Load\ imbalance = \frac{T_p - T_{\min}}{T_{\min}} \tag{8}$$

The evaluation of E (6), then the evaluation of the Load Imbalance (8), and the amount of required memory to implement the solution will demonstrate the efficiency of the proposed solution.

At a coarse grain, the parallel algorithm behaviour is:

Part 1: Initialise data and initiate the parallel execution

Part 2: Distribute the whole computation

Part 3: Compute locally each task inside each processor

Part 4: Write the output result and end

Part 1 and Part 4 contribute to the Overhead (Opar) in formula (2). Part 2 and Part 3 are concerned by load balancing strategies and computation time evaluations. So the formula (7) applied to the parallel algorithm decomposition becomes:

$$T_{\min} = T_{part2} + \frac{TS_{part3}}{p} \tag{9}$$

where $TS_{part3}$ is the execution time of Part3 on a uniprocessor computer. This formulation shows that, as the number of processors p increases, the efficiency

of the parallel computer is very sensitive to the value of $T_{part2}$. For example, the efficiency of a computer with 100 processors drops to 0.5 if $T_{part2}$ represents only 1% of the total execution time. In this paper a solution to distribute data is presented in order to reduce significantly the sequential part of the algorithm.

## 3 Case study 1 : a raytrace algorithm

### *3.1 Raytracing oveview*

Ray tracing is a computational technique that can perform wave simulation at comparatively low costs. This technique can be applied in cases where the frequency of the simulated wave can be adapted to the minimal object size. Ray tracing has been used for years in various fields like image synthesis, seismic simulation, radar reflections and others.

A first algorithm will be used to spot the issue raised by the parallel raytrace algorithm and will be referred to for the coming improved algorithm evaluations. The image is uniformly split into as many blocks as computing resources. The octree is entirely built at the beginning of the computation by each node to avoid communication during this step. Each node hence has the entire model and voxelisation information in its local memory: so the computation is achieved without communication. The memory requirement for each node is the same as on the single node of a sequential computer. The only positive aspect of this first trivial algorithm is that it does not need any communication.

#### *3.1.1 Improving load-balancing by a dynamic distribution of blocks*

To deal with the irregularity of the raytrace algorithm, a first improvement is to achieve a thinner static splitting of the image and a dynamic distribution of blocks. The algorithm is:

- At the master mode : *Let N be the number of processors*

  Split_image(block_size) // block_size is set to obtain a large number of blocks
  **for** i=1..N Assign(a_Block, node (i)) // Assignment of the first N blocks
  **while** non_computed_block_remains
  Wait_for_a_job_termination
  Assign(a_new_block, node (requester))
  **end while**
  **for** i=1..N Send(termination_signal, node(i))

- At each slave node :

```
Wait_for_a_job(job)
while not_the_termination_signal
  Compute(job)
  Send(job_termination, master_node)
  Wait_for_a_job(job)
end while
```

This should improve the load balancing, but the local memory requirement problem is not addressed yet.

#### *3.1.2 Saving memory, reducing computation and improving locality : the lazy raytrace*

### *3.2 The lazy algorithm*

At the beginning of a simulation, i.e. before the first ray is cast, the octree is reduced to a single leaf voxel. The octree will be actually built during the ray tracing process. Each time a ray hits a voxel, it is to be decided whether the polygon description of the voxel is sufficient or not for an analytic computation. If boundary condition is not reached, the intersection with all child voxels along the ray path has to be computed. If the voxel is a leaf voxel, it is evaluated in order to transform it into a node voxel. So, the lazy octree is a potentially infinite tree. Contrary to method using a static octree, most voxels actually built were hit at least once by a ray, and no useless voxel was built. More detail on lazy construction of voxels could be found in [3].

Inside each processor, the algorithm is the following:

```
Propagate(rays)
  for each ray
    Intersection(ray,octree_root)
    if (intersection ≠ nil) then
      Apply Snell Descartes laws to determine secondary rays
      if (secondary rays ≠ nil) then
        Propagate (secondary_rays)
      end if
    end if
  end for
end Propagate
```

The algorithm of the lazy recursive function Intersection is:

```
Intersection(ray,octree_elt)

  //First step: Actions on the octree element if it is a leaf
  if is_a_leaf(octree_elt) then
    if boundary_conditions(octree_elt) then //there is no need to explore deeper
      if (object_list(octree_elt)≠nil) then // the element contains surfaces
        Flag_as_terminal_node(octree_elt)
      else // the element contains no surface
        Flag_as_empty(octree_elt)
      end if
    else //deeper exploration is necessary
      Flag_as_node(octree_elt)
      Create leaf sons(octree_elt)
    end if
  end if

  // Second step: Action to take according to the flag of the element as it can no more be a leaf
  case typeof(octree_elt)
    empty:  return(nil)
    node:   if is_a_terminal_node then
              compute_intersection;
            else
              return merge(
                   if hit_by_ray then Intersection(ray,son 1(octree_elt) else nil,
                   if hit_by_ray then Intersection(ray,son 2(octree_elt) else nil,
                   if hit_by_ray then Intersection(ray,son 3(octree_elt) else nil,
                   if hit_by_ray then Intersection(ray,son 4(octree_elt) else nil,
                   if hit_by_ray then Intersection(ray,son 5(octree_elt) else nil,
                   if hit_by_ray then Intersection(ray,son 6(octree_elt) else nil,
                   if hit_by_ray then Intersection(ray,son 7(octree_elt) else nil,
                   if hit_by_ray then Intersection(ray,son 8(octree_elt) else nil,
            end if
  end case
end Intersection
```

This algorithm shows the following properties: first, a child node is evaluated only if it contains necessary data for the computation; then, the node evaluation results is definitively stored in the octree and will be reused for neighbour ray computation. Thereby, the algorithm exploits spatial ray coherence.

The main drawback of the algorithm is the remaining data replication. To reduce this undesirable replication, neighbour ray's coherence hypothesis is

taken. It states that neighbour rays have a high probability to cross a lot of common voxels and only few different voxels. A proximity support in the assigned ray choice for each node is so achieved to minimise data replication phenomenon. Nevertheless some rays assigned to different nodes may need common voxel evaluation and generate data replication.

### *3.3 Lazy raytrace algorithm performance results*

Results were obtained using 16 Sun Ultra 10 with 256 MB of memory interconnected by an Ethernet 100 Mb/s network. MPI1.1 was used to distribute the computation. Benchmarks come from very well known images (Teapot12 and Tetra9) and proprietary scenes to extend computation complexity [Tab.1].

| SCENE NAMES | MODEL SIZE | NB OF SURF. | PICT. SIZE | SEQUENTIAL TIMES | |
|---|---|---|---|---|---|
| | | | | STANDARD | LAZY |
| TEAPOT 12 | 1.19 MB | 9,408 | 2048X2048 | 220s | 163s |
| TETRA 9 | 18.24 MB | 262,144 | 2048X2048 | 194s | 155s |
| GEN8 | 26.21 MB | 786,438 | 1024X1024 | 793s | 201s |
| BIG_GEN8 | 104.84 MB | 3,145,728 | 2048X2048 | mem overflow | 3,992s |

Table 1
Test scenes overview

The local memory requirement is the first point where laziness has a significant impact. The lazy version of the raytrace algorithm uses less memory as the non lazy one; as the number of references to the memory hierarchy decreases the total computation time decreases dramatically. Huge computation (3 millions surfaces, 2K by 2K image) could not be computed on a single processor with conventional data management but has been computed with the lazy version of the raytrace algorithm using a computer with 256 MB of memory in about one hour.

Efficiency is in the range [1..0.93] for small scene [Tab.2] as the number of processors increases. For medium size image efficiency is in the range of [1..0.61] and is always better with the lazy algorithm.

With the non lazy implementation of the raytrace algorithm each node has the entire model and voxelisation data in its local memory; so the computation is achieved without communication but the local memory requirement is constant and maximum whatever the number of nodes is. The measured efficiency of this algorithm shows that global computation time gets far higher than desirable as the number of nodes increases showing an important load imbalance between nodes [Tab.3].

| EFFICIENCY | NON LAZY | | | | LAZY | | | |
|---|---|---|---|---|---|---|---|---|
| Nb of PE's | 2 | 4 | 8 | 16 | 2 | 4 | 8 | 16 |
| TEAPOT 12 | 0.9 | 0.8 | 0.75 | 0.6 | 0.98 | 0.97 | 0.95 | 0.93 |
| TETRA 9 | 0.82 | 0.6 | 0.3 | 0.3 | 0.9 | 0.76 | 0.7 | 0.61 |
| GEN8 | 0.6 | 0.71 | 0.68 | 0.5 | 0.97 | 0.9 | 0.8 | 0.64 |
| BIG_GEN8 | Memory overflow | | | | 0.98 | 0.97 | 0.97 | 0.95 |

Table 2
Algorithm efficiencies (Eq.6)

| LOAD IMBALANCE (in %) | NON LAZY | | | | LAZY | | | |
|---|---|---|---|---|---|---|---|---|
| Nb of PE's | 2 | 4 | 8 | 16 | 2 | 4 | 8 | 16 |
| TEAPOT 12 | 15 | 22 | 40 | 60 | 0.5 | 1.2 | 2.2 | 2.8 |
| TETRA 9 | 15 | 60 | 220 | 240 | 11 | 40 | 42 | 52 |
| GEN8 | 60 | 40 | 50 | 95 | 5 | 10 | 22 | 55 |
| BIG_GEN8 | Memory overflow | | | | 2 | 8 | 10 | 13 |

Table 3
Algorithm load imbalance ratio (eq.8)

The memory requirement decreasing rate is about 25% each time the number of nodes doubles [Tab.4]. The memory is now distributed among the computing nodes thanks to the laziness added to the base algorithm. On the GEN8 test a memory saving can be observed with the sequential lazy computation; it come from important useless parts of the octree that is evaluated by the classic algorithm and not by the lazy algorithm. The speed-up obtained on the GEN8 test is equal to 15 with 16 processors and for the huge test BIG_GEN8 is 31 with 32 processors and the memory is well distributed among each processor.

| MEMORY NEEDS (MB) | NON LAZY | | | | LAZY | | | |
|---|---|---|---|---|---|---|---|---|
| Nb of PE's | 2 | 4 | 8 | 16 | 2 | 4 | 8 | 16 |
| TEAPOT 12 | 8 | 8 | 8 | 8 | 6.5 | 5.7 | 4.1 | 3.7 |
| TETRA 9 | 80 | 80 | 80 | 80 | 55 | 38 | 30 | 23 |
| GEN8 | 140 | 140 | 140 | 140 | 40 | 30 | 22 | 17 |
| BIG_GEN8 | Memory overflow | | | | 250 | 240 | 200 | 170 |

Table 4
Memory requirements

## 4 Case study 2 : Adaptive mesh refinement for Computational Fluid Dynamic simulation

Adaptive mesh refinement is a numerical technique that dynamically places high resolution numerical grid in the region of a numerical simulation where higher accuracy is needed [4]. The method is crucial in solving PDE's where large gradients and dynamic scales are present [Fig.5]. Problems in computational fluid dynamic exhibit such gradients and widely varying dynamics (Euler or Navier-Stokes equations). Dynamically adaptive methods for solution of differential equations which employ locally optimal approximations have been shown to yield highly advantageous ratios for cost/accuracy where compared to methods based upon static uniform approximations. Adaptive methods start with a base coarse grid with minimum acceptable resolution that covers the entire computational domain. As the solutions progress, regions in the domain requiring additional resolution are identified and finer grid are generated on the identified regions of the coarse grid. Refinement proceeds recursively for regions on the finer grid requiring more resolution; to link computation inside one grid with outside it is necessary to introduce virtual cells at the grid boundary; the size of this belt of cells depends of the equations to be solved. The virtual cells are initialised then a grid can be integrated within a time step independently with others grids.

Lazy evaluation will be used to define the adaptive data structure; like with the previous example the data structure is a tree dynamically built. Nodes of the tree represent grids at a particular level with three different status:

(1) Constant :The grid contains constant values which are already computed
(2) Leaf : The grid has to be refined and its child grid are not yet built
(3) Node : The grid has been refined and its child grid are already built

Lazy evaluation will allow a leaf grid to be transformed into a node grid. This process is called grid evaluation. At the beginning of a simulation, the tree is reduced to a single leaf grid. The tree will be actually built during the time step evolution.

During the grid evaluation process, all the algebraic equations for the unknowns have to be calculated. A lazy process would delay those computations, allowing some child grid to be fully computed and some others not.

Each time step, it is to be decided whether the grid description is sufficient or not for an analytic computation. Usually this condition is based on error estimation. If this condition is not reached, time integration is performed on each component grid using a specified operator (Each component grid may have its own operator and can be integrated independently) then the component grids is advanced at a particular level of refinement in time. If the grid

is a leaf grid, it is evaluated in order to transform it into a node grid.

Contrary to methods using a static tree, most grids actually built were needed by the calculus and no useless grid was built. This can result in memory saves and also improve locality as refinement is applied locally.

A recursive formulation of the adaptive method is the following [5] :

```
Amr (level l, time dt)
  if (is_regrid_time(l)) then // true if the grid is a leaf
    Refine_level(l)
  end if
  if (is_non_plus_fin) then // true if the grid is a node
    Creer_CL_niveau_fin(l) // create virtual cells
  end if
  Init_CL_niveau(l) // initialise virtual cells
  Integrer_Niveau(l, dt) // time integration
  if (Is_non_plus_fin) then
    for k=1..r do
      Amr(l+1, dt/r)
      Corriger_flux_niveau(l) // conservative correction scheme
      Projeter_niveau_fin(l) // update level l with value computed on level l+1
    end for
  end if
end
```

The primary source of parallelism in adaptive method is data-parallelism that can be exploited by decomposing the computational grid across the processing elements and concurrently operating on the local portion of this domain. Different decomposition methods have to be defined to optimise computation time and load balancing.

The base grid is created using specified bounding box information representing region in the computational domain. The server decomposes this bounding box into blocks with granularity greater than an architecture specific minimum. A sufficient number of blocks are generated so that they can be uniformly distributed among the available processors. The associated data-storage is created on each processor as the first level. The integration algorithm defines inter-grid communications between component grids at different level of the grid hierarchy; with the lazy tree these communications are local to each block and hence can be performed without any remote access.

```
Integrer_Niveau (level l, time dt)
  for all grids at level l do
    Calculer_flux //compute flows between cells
```

```
    Evoluer_solution(dt) //advance computation with flows
    Sauver_flux_fin //save flows to finer level
    Sauver_flux_Grossier //save flows to upper level
  end for
end
```

Preliminary evaluation shows that the lazy algorithm has no significant overhead and allows to compute problems with one order of magnitude in size compared to methods based upon static uniform approximation [5]. The resulting partitions of the adaptive grid hierarchy require no communications during one time step. Several grid distribution algorithms are under evaluation to optimise the load balancing and the final execution time on clusters and metacluster architectures.

## 5 Conclusion

Efficient parallel solutions on distributed computers must reduce communications to the minimum, as they constitute a very important overhead. Dynamic hierarchical data structures and lazy evaluation leads to a natural memory repartition among computing nodes implying improvement in data localities and reducing dramatically the need of communication. Laziness has two fundamental advantages allowing creation of data at right time and at right place in the local memory of the processor that need effectively the data. Non lazy algorithms made data reservation at compile time or during the initialisation phase that may create useless data structures. However laziness could imply redundancy in distributed memory, as part of the data and computation could be duplicated, compared to global memory allocation and distribution, but it is the cost to be paid to have an efficient computation of large simulation on a distributed memory multiprocessor. Experiences have shown that this cost is negligible compared to the cost to access remote global memory through high bandwidth, low latency networks.

## References


[1] David E. Culler and J.P. Singh, *Parallel computer Architecture* (Morgan Kaufmann Publishers Inc., 1999).

[2] P. Sainrat and B.Lecussan, *L'architecture du noeud de la grappe : état de l'art et prospective* (Ecole d'hiver iHPerf2000 : Applications Hautes Performances). *Analyse, Conception et utilisation des grappes homogènes*

*ou hétérogènes de calculateurs* (4-8 Décembre 2000 Aussois- France : http://www.irisa.fr/iHPerf2000/documents.html).

[3] S. Bermes, B. Lecussan, C. Coustet *MaRT : Lazy Evaluation for Parallel Ray tracing* (High Performance Cluster Computing, Vol 2, Prentice Hall 1999).

[4] Manish Parashar and James C. Browne *On partitioning Dynamic Adaptive Grid Hierarchies* (HICSS-29, January 1996).

[5] L. Le Saint et N. Hardouin *Etude et implémentation d'une stratégie d'enrichissement automatiques de maillages structurés : Application aux équations d'Euler* (Projet d'Initiation à la Recherche - Supaero, Juin 2001).