Laziness, a way to improve distributed computation of the ray tracing algorithm

Olivier POITOU, Bernard LECUSSAN, Sébastien BERMES
ONERA-CERT/DTIM
2 Av. E. BELIN
31055 TOULOUSE Cedex

{poitou, lecussan, bermes}@cert.fr

ABSTRACT

This paper raises the issue of computational workload and memory load balancing to ray trace large scenes efficiently on a network of workstations. The task splitting is done on the image to be produced and not on the scene data to obtain a high performance level. To deal with the high memory requirements of such a distribution strategy, laziness is added to the base algorithm. This reduces the computing local memory requirements to the locally computed part of the image. It also reduces the sequential parts of the algorithm by making the voxelization process parallel. Many comparisons have been done using a manager/worker distribution algorithm on different scenes computed on a conventional network of workstations. Performance, load imbalance, communication overhead, and memory requirement results are given and discussed in this paper. Furthermore, this paper demonstrates that the proposed solution improves the results obtained with conventional algorithms, no matter what network used or however complex the image is.

I. INTRODUCTION

Ray tracing is a realistic synthesis image rendering technique which requires a high computing power and a large amount of memory to compute realistic images. Many papers have been proposed since 1980 to improve ray tracing algorithm computations [GLA89,SCR97]. This algorithm exhibits naturally a high degree of parallelism, considering independent rays propagation or independent image computation, but with irregular workload of computation leading to inefficient usage of a parallel computer. The well known improvement of the base algorithm comes from space voxelisation [AMW87] , but this single technique is not sufficient for an efficient parallel computation.

Starting from an existing sequential algorithm, this paper introduces a new method and associated algorithms to compute ray tracing wave simulation in parallel. Experiments are made with image computing but the proposed solutions could be applied in others area based on wave propagation simulation (electromagnetism, seismic, etc. [BER98]).

This paper starts with a quick presentation of related researches in the field, pointing the main drawbacks of the proposed approaches; then the followed methodology bypasses these drawbacks introducing additional algorithms, especially for memory management and workload distribution.

Section II introduces the related works on the ray tracing application parallelisation, section III presents the proposed solutions to improve load balancing and memory management of a distributed computation, section IV explains experimental conditions and the evaluation criteria choices, section V exhibits the algorithms used and finally section VI presents results obtained with a network of workstations. The conclusion of the paper presents a synthesis of the experiments and possible extends to improve the validation of the proposed algorithms.

II. RELATED WORKS

Two kind of parallelisation strategies of the ray tracing algorithm are conventionally presented: either distribute the database among the nodes memory, called data oriented distribution, or distribute the computation of all rays to each computing node, called image oriented distribution.

With data oriented approach, each node owns only one part of the model enabling the rendering of large scene by static decomposition of the whole data structure. It produces a higher communication rate since the data for a ray computation is not in the local memory. The communication overhead leads to poor performance of the distributed computation. [DMS84 and KNK88].

The image oriented approach gives best performances but suffers a serious drawback, which is the amount of required memory. Ideally, the entire model and the voxel structure have to be loaded in each local memory. The memory requirements dramatically limits the maximum size of the model that may be computed with algorithms implementing this strategy. The Shared Virtual Memory solution was proposed to distribute data among all local storages in order to deal with the model size limitation and to be able to compute realistic cases. The solution implies higher communication rate and consequently lower performance if communication overhead is not overlapped by efficient computation [BKP96]. Multithreading may be a solution to this problem but strongly dependent on the scene structure.

In [PRM98], for instance, the Shared Memory Model of the CRAY T3D is used to access to the whole database from every node while only keeping one copy of the database in the system. The memory requirement is so reduced from $N*DB_{size}$ to $1*DB_{size}$ for N processors but the parallel computer efficiency seems far from an optimum.

In [BBP94], an advanced algorithm is used, based upon a shared memory emulation. Only the octree is entirely stored in each local memory, the main database is loaded on demand during the computation. This solution, implemented on a high performance multi-processor, strongly relies on the network efficiency and doesn't seem so efficient on a workstation network.

III. COMPUTING THE RAY TRACING ALGORITHM ON A DISTRIBUTED MEMORY COMPUTER

Previous works on parallel ray tracing computation shows that the image oriented distribution strategy with the model data entirely stored in each local memory leads to efficient results but at the cost of a large amount of memory. To solve this problem some authors have proposed a Shared Virtual Memory concept distributed on each local memory. Thus, each node has a part of the model in its local memory and is able to access the others parts, stored in other local memory of each node via messages. The fundamental parameters of the actual technology implies that the access cost of a non-local data via a network varies in the range of 4,000 to 40,000 processor cycles according to the processor throughput, to the network latency and the communication protocol overhead [SBJ99]. Such latency has to be overlapped by computing effective code to reach efficient parallel computation. Multithreaded implementation minimizing the task communication costs or speculative algorithms predicting code to be computed are presented in previous papers. The efficiency of those solutions strongly relies on the technology and the CPU_time/Communication_latency ratio. Furthermore it seems very hard to find solutions which are not dependent on the image complexity.

The strategy followed in the research presented here aims to produce the smallest number of messages possible while reducing memory requirement for voxel storage. The main idea is to build the octree dynamically on demand, only when and where the part of the octree is needed [BLC99].

Voxelisation is usually achieved during the ray tracing initialization. The model discretisation is contained in the leaves of the produced octree [GLA84]. Rays are then cast on the octree and intersection computation is computed on the ending voxels. The octree construction is a sequential task that must be achieved before the task repartition. The octree distribution is a bottleneck that dramatically impacts the expected performance of the parallel machine.

The lazy octree is a potentially infinite tree; voxels have three different status :

- empty : the voxel contains no surfaces
- node : the voxel is not empty and its eight children's voxels are already built
- leaf : the voxel is not empty, but its children are not yet built

At the beginning of a simulation, i.e. before the first ray is cast, the octree is reduced to a single leaf voxel. Lazy evaluation will allow a leaf voxel to be transformed into a node voxel : this process is called voxel evaluation and it is done dynamically along simulation. 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. This boundary depends on the number of surfaces in the voxel, or on the relative size of the voxel compared to the ray solid angle. If this boundary is not reached, the intersection with all children 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. In order to avoid expensive tests during ray-voxel intersection dynamic boundary conditions can flag a voxel.

Contrarily to conventional methods using static octrees, most voxels actually built were at least hit by one ray, and no useless voxel was built. This can result in large amounts of memory saves due to hidden parts of a scene : as no ray cross those areas, no voxels are built there. If the ray uses a solid angle method to avoid a deeper exploration, it can also save voxel construction in deep octree branches.

This algorithm has interesting sequential properties to save memory, as only the needed part of the octree is built. Furthermore on a parallel computer local data structure will be built inside each local memory at the demand of the ray computed by each processor. In that way, voxelisation process can be distributed on each processor without any message.

The methodology of performance evaluation of the proposed solution is summarized in four sets of results obtained by :

1) A trivial algorithm of data distribution which needs a large amount of memory but without communication overhead.
2) An improved algorithm to improve the load balancing of the distributed computer without solving the memory issue.
3) The implementation of the lazy algorithm which reduces the amount of required memory with static splitting of images and dynamic distribution algorithm.
4) An improvement of the previous solution with a dynamic splitting algorithm.

Comparisons will be done considering that algorithm 1) gives the best solution at the communication overhead point of view, then solution 2) improves the load balance of the previous solution; solution 3) gives an answer to reduce the total amount of required memory and solution 4) is a final optimization combining the main qualities of each previous algorithms.

IV. EVALUATION CRITERIA

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 speedup gained by the parallel computation.

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

$$T_p = \max_i t_i \quad (1)$$

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

The parallel computer efficiency E using p processor is :

$$E = \frac{T_s}{p \times T_p} \quad (2)$$

where T_s is the best sequential algorithm known 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 :

$$T_{min} = T_{seq} + \frac{T_{par}}{p} \quad (3)$$

where T_{seq} and T_{par} respectively are the non-parallelisable and the parallelisable part of the sequential computation time (Amdahl's law, performance improvement to be gained from using faster mode of execution is limited by the fraction of the time the faster mode can be used). 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 :

$$\text{Load Imbalance} = \frac{T_p - T_{min}}{T_{min}} \quad (4)$$

The efficiency of the proposed solution will be demonstrated by the evaluation of E (Eq. 2) then the evaluation of the Load Imbalance (Eq. 4), and the amount of required memory to implement the solution.

At a coarse grain, the algorithm behavior is :

Part 1 : *Initiate the parallel execution and read the model data*

Part 2 : *Distribute the model voxelisation*

Part 3 : *Compute the image*

Part 4 : *Write the output image file and end*

Part 1 and *Part 4* are Input/Output operations and are not considered in this paper. *Part 2* and *Part 3* are concerned by load balancing strategies and computation time evaluations. On previous approaches *Part 2* is fully sequential and *Part 3* is entirely parallelisable, so the formula (Eq. 3) applied to the ray tracing computation becomes :

$$T_{min} = T_{part2} + \frac{TS_{part3}}{p} \quad (5)$$

where TS_{part3} is the execution time of *Part 3* 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 Tpart2 represents only 1% of the total execution time. In this paper a solution to distribute voxelisation is presented in order to reduce significantly the sequential part of the algorithm.

The communication ratio indicates the cost of the parallelization algorithm.

Results were obtained using 16 Sun workstations (Ultra 10 with 256 Mo of memory) interconnected by an Ethernet 100 Mb/s network. An additional validation on a PC cluster with Myrinet is in progress, to show the independence of the proposed solution to the technology.

MPI ver. 1.1 [MPI2] will be used to distribute the computation.

V. ALGORITHM DESCRIPTIONS

1. The trivial repartition algorithm

A first trivial algorithm will be used to spot the issue raised by the ray tracing parallelisation 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 computation ends when the workload heaviest block is computed.

The measured efficiencies of the static trivial algorithm shows that global computation time gets far higher than desirable as the number of nodes increases (Figure 1, Figure 5, Figure 9).

The high load imbalance values confirm the ray tracing algorithm irregularity (Figure 2, Figure 6, Figure 10). The differences between node computation times are significant (they have reached 40 seconds for a global execution time of 120 s), showing an important load imbalance between nodes. As limiting the computation time imbalance is a key issue to performance, this will be the first problem to address.

Furthermore the local memory requirement is constant and maximum whatever the number of nodes is; memory is not distributed at all. The memory requirement for each node is the same as on the single node of a sequential computer. This memory waste is the second point to improve.

The only positive aspect of this first trivial algorithm is that it does not need any communication. In fact, the parallel computer rate is unacceptable while there is no communication overhead.

2. Improving load-balancing by a dynamic distribution of the blocks

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

- **At the master node**

```
// let N be the number of processors
Split_image(block_size)
// with block_size adjusted to obtain number of
// blocks >> N
For i=1..N Assign(a_Block, node(i))
// this was the assignment of the first N blocks
While non_computed_blocks_remain
  Wait_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. Saving memory and reducing computation : the lazy ray tracing

3.1. The lazy algorithm

Inside each processor, the algorithm is the following :

```
Propagate(rays)
  For each rays
    Intersection(ray,octree_root)
    If (intersection<>nil)
      Apply Snell Descartes laws to determine
      secondary rays
      If (secondary_rays<>nil)
        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)
    If boundary_conditions(octree_elt)
    // there is no need to explore deeper
      If (object_list(octree_elt)<>nil)
        // 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
    is_empty(octree_elt) :
      return(nil);
    is_a_node(octree_elt) :
      if is_a_terminal_node
        compute_intersection;
      else
        return merge(
  if hit_by_ray Intersection(ray,son1(octree_elt) else nil,
  if hit_by_ray Intersection(ray,son2(octree_elt) else nil,
  if hit_by_ray Intersection(ray,son3(octree_elt) else nil,
  if hit_by_ray Intersection(ray,son4(octree_elt) else nil,
  if hit_by_ray Intersection(ray,son5(octree_elt) else nil,
  if hit_by_ray Intersection(ray,son6(octree_elt) else nil,
  if hit_by_ray Intersection(ray,son7(octree_elt) else nil,
  if hit_by_ray Intersection(ray,son8(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 neighbor ray computation. Thereby, the algorithm exploits spatial ray coherence.

The main drawback of the algorithm is the remaining data replication. Although it is reduced by a proximity support in the assigned ray choice for each node, some rays assigned to different nodes may need common voxel evaluation and generate data duplication. This cost is to be evaluated as it strongly relies on the considered application.

3.2. Static splitting, dynamic distribution algorithm

The hybrid algorithm uses the repartition algorithm introduced in section V.2, a nearest new block choice and a static image splitting. It has been tested with and without implementing the lazy evaluation; it leads to the following results :

Efficiencies are better than the previous one especially when the number of nodes increases, lazy evaluation does not significantly interfere on performance (Figure 1, Figure 5, Figure 9).

The repartition quality of the hybrid algorithm evolves linearly. Its values are always lower than those of the trivial algorithm. The two versions obtain the same results. (Figure 2, Figure 6, Figure 10).

The local memory requirement is the first point where laziness has a significant impact. The non lazy version obtains the same results as the first trivial algorithm with a constant and maximal memory requirement for all test configurations. On the opposite, the lazy algorithm offers decreasing memory requirements as the number of nodes increases. The memory requirement decreasing rate is about 20% each time the number of nodes doubles (Figure 3, Figure 7, Figure 11). The memory is now distributed among the computing nodes thanks to the laziness added to the base algorithm. On the last gen8 test an important memory requirement reduction can be observed on the sequential lazy execution. It is due to useless parts of the octree evaluated by the classic algorithm and not by the lazy algorithm (Figure 11).

The introduced communication is correct for gen8 test (Figure 12), just acceptable for teapot12 test (Figure 4) and clearly too high for tetra9 test (Figure 8). The results are the same on both versions of the hybrid algorithm.

Results indicate a sensible execution time improvement with both versions of the hybrid algorithm and a significant memory saving with the lazy version. Lazy evaluation solution always leads to memory requirements and execution time improvements on parallel machines but also on sequential machines when the computed scene contains hidden parts.

4. Dynamic image splitting

However, using both dynamic splitting and dynamic repartition improves data and computation locality. Moreover it may reduce the data replication too.

Computing a single large block instead of the four smaller blocks that compose it, ensures this

computation is done by a single node. It really takes advantage of the locality and it also fully uses the original sequential algorithm efficiency. Moreover it reduces the number of needed messages. But it may involve a greater execution time imbalance between nodes being assigned blocks that generates very different workloads.

On the opposite, the use of small blocks often implies more data replication and generates more messages. But it induces very close execution times which means load balancing improvement.

To take advantage of both large and small blocks the following dynamic splitting and dynamic repartition algorithm is used :

```
// let N be the number of processors
// let NS be the number of different block sizes
Dynamic_repartition()
// first splitting/repartition
Block_side = Image_side / N;
Split(Image,Block_side); // N*N blocks created
Distribute N blocks
  // start the loop
  While it_stays_blocks_to_compute
    // wait and count for achieved blocks
    Wait_for_a_job_termination;
    Assign(a_new_block, requesting_node);
    Waited_blocks = Waited_blocks - 1;

    // if conditions true prepare next block size
    If (Waited_blocks = 0 and Block_side ≥ minimum_side)
    Then
      // prepare the next block size
      Block_side = Block_side / reduction_factor;
      Split(Image,Block_side);

      // schedule the re-split
      Blocks_before_resplitting =      blocks to cover
                                  1/NS of the image surface;
      Waited_blocks = Blocks_before_resplitting;
    End if
  End While
End Dynamic_repartition
```

The minimum size of a block must be chosen to ensure that the communication time for the block will never be greater than its computation time; so evaluating it remotely would cost more than evaluating it locally [HLL96].

An efficiency improvement can be observed in particular when the number of node increases (Figure 1, Figure 5, Figure 9). Efficiency is almost linear with a value of 0.93 on the 16 node configuration of teapot12 test (Figure 1).

The load imbalance values are a bit lower than previous ones when the number of node increases. Therefore starting he computation with larger blocks do not impact the load-balancing (Figure 2, Figure 6, Figure 10).

The memory requirements are the same as those of the lazy hybrid algorithm (Figure 3, Figure 7, Figure 11).

Communication ratio have been hardly reduced (Figure 4, Figure 8, Figure 12).

VI. RESULTS

The presented results correspond to the three following scenes (Table 1 :Test scenes).

Scenes	Model size (in MB)	Number of surfaces	Picture size	Sequential computation time	
				Non lazy	Lazy
Teapot12	1.19	9,408	2048x2048	220 s	163 s
Tetra9	18.24	262,144	2048x2048	194 s	155 s
Gen8	26.21	786,438	1024x1024	793 s	201 s

Table 1 :Test scenes

The two first scenes are part of the well known SPD, the last one is proprietary. Due to paper size limitation only three tests are presented. Results are presented in the following pages.

VII. CONCLUSION

Efficient parallel solutions on workstation networks must reduce communications to the minimum, as they constitute a very important overhead. However this reduction implies more local data knowledge and thus more local memory requirements. The use of a lazy evaluation base algorithm leads to a natural memory repartition among the computing nodes and implies a lot less communications while computing complex scenes with a high level of performance.

Lazy evaluation benefit might be contested because it can be annihilated in the case of scene with reflecting surfaces imposing a lot of rays to cross the entire scene. However, even in the worst theoretical case in which the whole data set would be needed on each node, the lazy evaluation would not perform worse then classical solution but would be absolutely equivalent to it.

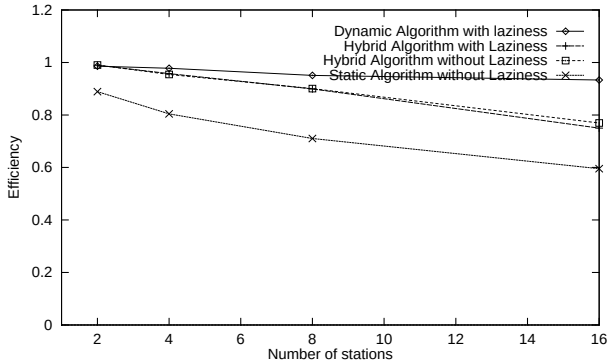

Figure 1 : Teapot12 algorithm efficiencies

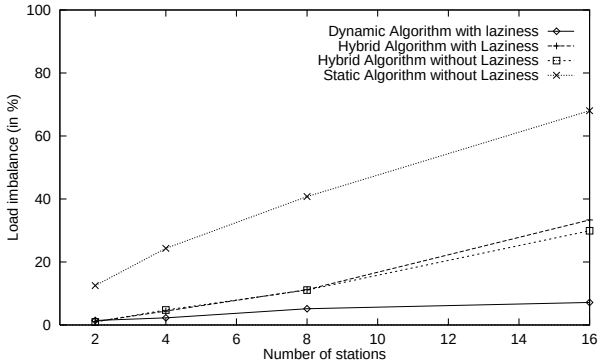

Figure 2 : Teapot12 algorithm load imbalance ratio

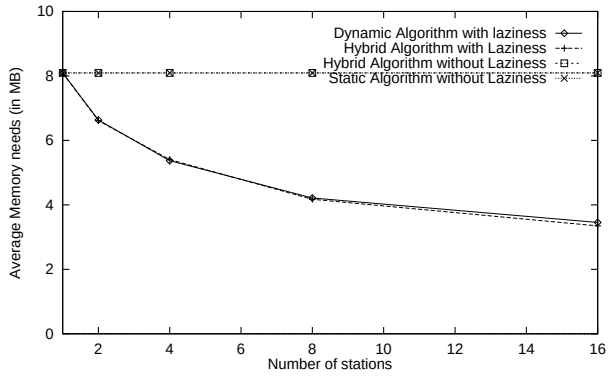

Figure 3 : Teapot12 local memory requirement

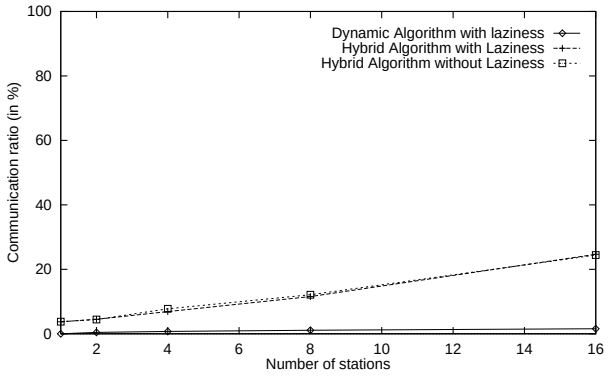

Figure 4 : Teapot12 communication ratio

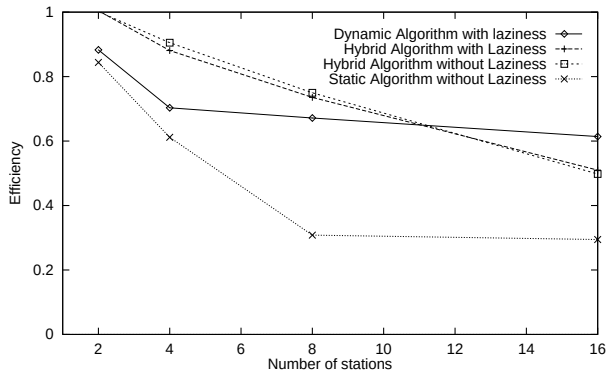

Figure 5 : Tetra9 algorithm efficiencies

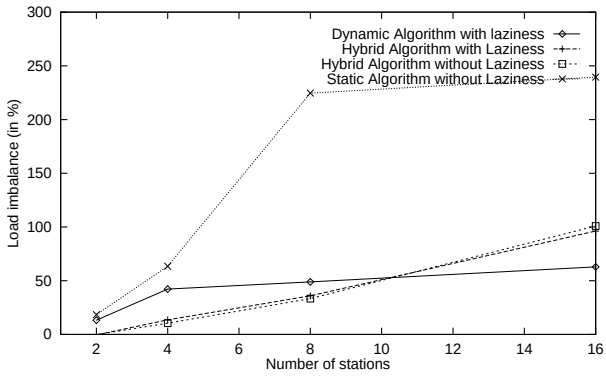

Figure 6 : Tetra9 algorithm load imbalance ratio

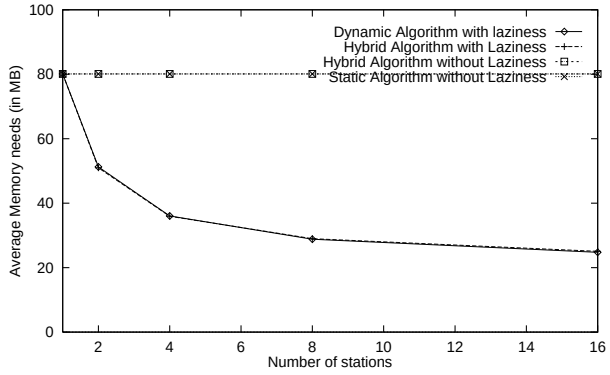

Figure 7 : Tetra9 local memory requirement

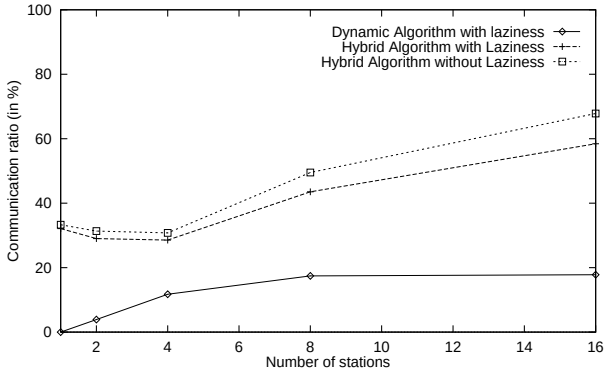

Figure 8 : Tetra9 communication ratio

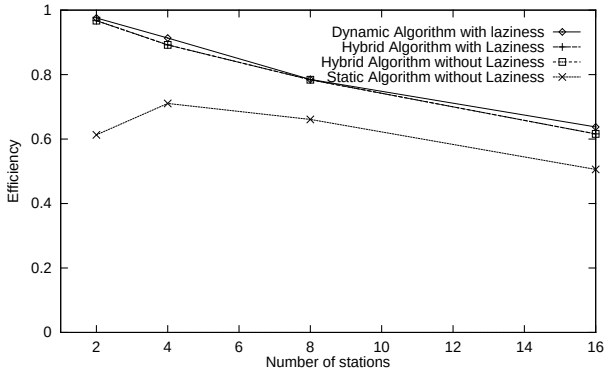

Figure 9 : Gen8 algorithm efficiencies

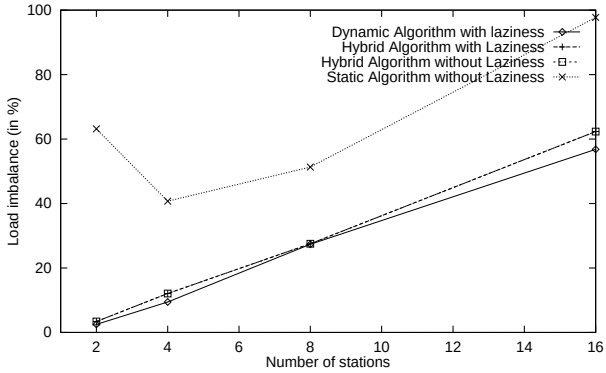

Figure 10 : Gen8 algorithm load imbalance ratio

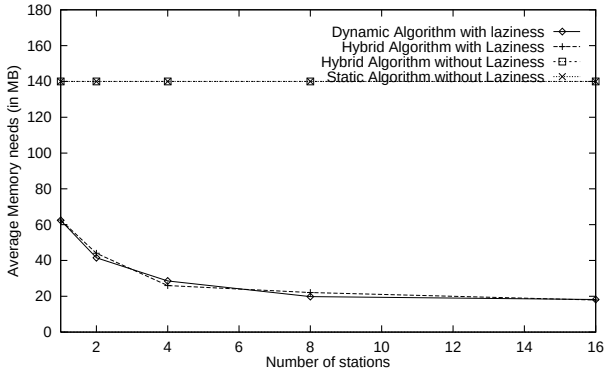

Figure 11 : Gen8 local memory requirement

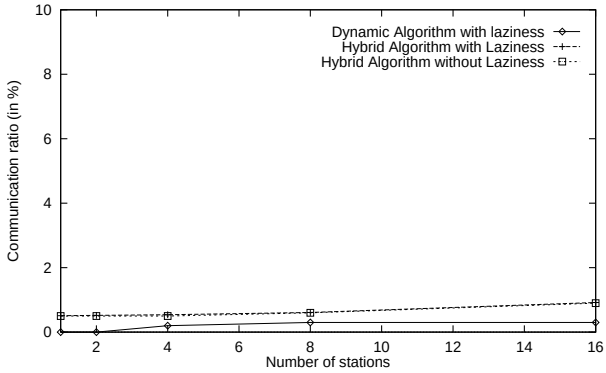

Figure 12 : Gen8 communication ratio

VIII. References

[GLA89] Glassner, Andrew. *An Introduction to Ray Tracing.* Palo Alto C.A. : Academic Press 1989

[SCR97] Schutt, Robert. *Ray tracing 101.* Colgate University Department of Natural Sciences Ray Tracing on Parallel Microprocessors.

[AMW87] John Amanatides, Andrew Woo. *A fast Voxel Traversal Algorithm for Ray tracing.* Proceedings of Eurographics 1987

[BER98] Sébastien Bermes, *Les arbres octaux paresseux : une méthode dynamique de subdivision spatiale pour le lancer de rayons.* Thesis, 1998.

[DMS84] Dippé M.A.Z & Swensen J. *An adaptative subdivision algorithm and parallel architecture for realistic image synthesis.* Christiansen ed., Computer Graphics (SIGGRAPH '84 proceedings), Vol.18, p.149-158, 1984

[KNK88] Kobobayashi H., Nishimura S., Kubota H., Nakamura T. & Shigei Y. *Load balancing strategies for a parallel raytracing system based on constant subdivision*, The Visual Computer 4(4), 1988.

[BKP96] R.Bianchini, L.I. Kontothanassis, R.Pinto, M. De Maria, M.Abud and C.L. Amorim. *Hiding Communication Latency and coherence overhead in Sofware DSMs.*

[PRM98] Igor-Sunday Panzic, Michel Roethlisberger, Nadia Magnetat Thalmann. *Parallel raytracing on the IBM SP2 and CRAY T3D.* MIRALab Copyright Information 1998.

[BBP94] Badouel D., Bouatouch K., Priol T. *Distributing data and control for Ray Tracing*. Computer Graphics and Applications, p.69-77, 1994.

[SBJ99] J.P. Singh, A. Bilas, D. Jiang and Y. Zhou. *Limits to the performance of Software Shared Memory : A Layered Approach.* 1999

[BLC99] Bermes Sébastien, Lécussan Bernard, Coustet Christophe. *MaRT : Lazy Evaluation for Parallel Ray Tracing*. High Performance Cluster Computing, Vol.2, Programming and applications.

[GLA84] Andrew S. Glassner. *Space subdivision for fast Ray-Tracing.* IEEE Computer Graphics and Applications, Vol.4, N. 10., pp 15-22, Oct. 84

[MPI2] *MPI : A Message-Passing Interface Standard.* June 12, 1995

[HLL96] Tsan-Sheng Hsu, Joseph C. Lee, Dian Rae Lopez, William A. Royce. Task Allocation on a Network of Processors. 1996.