A PERFORMANCE PREDICTION TOOL FOR PARALLEL RAY TRACING

M. Krajecki, Z. Habbas, F. Herrmann, Y. Gardan
Laboratoire de Recherche en Informatique de Metz
Ile du Saulcy
57045 Metz Cedex, France
Tél. : +33 3 87 31 54 44
Fax : +33 3 87 31 53 33
E-mail : {krajecki, zineb, herrmann, gardan}@lrim.univ-metz.fr

ABSTRACT

This paper proposes a general matrix iterative model to represent a range of dynamic load balancing for MIMD parallel ray tracing. Different measure parameters are expressed only on theoretical data. A parallel ray tracing application is implemented. Both results obtained from the model and from parallel implementation are used to compare different load balancing strategies and validate the model. As a main result, we observe no significant error between model and measured value. We can deduce that our model is an interesting and accurate tool to select the best algorithm before implementing it.

KEYWORDS: load balancing, parallel algorithms, matrix iterative model, parallel ray tracing, parallel and distributed graphics, MIMD application.

1 INTRODUCTION

Ray tracing is a well-known algorithm in the visualization area for its image quality and its simplicity. Unfortunately, it is computationally expensive. Several sequential optimizations have been proposed in order to reduce the computational cost of this algorithm. But in despite of all these improvements, it is still too much time consuming on sequential computers. On the other hand, parallel ray tracing has been intensively explored. We intend to propose a general tool in which different parallel ray tracing performances are theoretically measured. More precisely, we introduce a mathematical framework to represent and validate different load balancing strategies for MIMD parallel ray tracing.

We address this issue and develop a performance tool which analyses the matching of load balancing algorithm to the parallel ray tracing. This model is derived from approaches [Cyben89] and [Frank96]. But the two authors restrict their works to SIMD and SPMD applications. In our study, we develop this model in order to parametrize full distributed MIMD load balancing algorithms. In fact, studying asynchronous load balancing schemes along the lines of asynchronous application has never been presented in literature and this is our contribution of our paper.

Section 2 describes our target application example: the parallel ray-tracing algorithm. Section 3 presents the proposed load balancing model, performance measures can be derived from this model to evaluate the different algorithms. It also adds two load balancing schemes that are represented by the model. At last, in section 4 the model is validated and used to compare the two mentioned load balancing algorithms and to evaluate their efficiency.

2 THE RAY-TRACING ALGORITHM

Ray tracing is a well known algorithm in the visualization area for its image quality and its simplicity. Unfortunately, it is computationally expensive. This algorithm takes into account multiple reflections and transparencies to visualize a scene. We describe here the basic ray tracing algorithm of Whitted [Whitt80]. It processes all the pixels of the screen in the same way. The intensity of each pixel is computed by a primary ray that is opposite to the observer's point of view. The visualization of a given pixel consists in determining the nearest intersection point with the objects of the scene. If no intersection point exists, the pixel is not displayed. Otherwise, the illumination at this intersection point is computed. To do so, the algorithm needs to cast a shadow ray in the direction of each light source in order to determine whether the intersection point is in the shadow with respect to that light. Finally, the indirect illumination is computed by casting two secondary rays : a transmitted one and a reflected one. All types of rays are processed in the same way so that underlying structure of each pixel forms a tree.

Determining the closest intersection point between a ray and an object can be computationally expensive since generally the same ray is tested against all the objects of the scene.

Several techniques have been studied in order to reduce the cost of this algorithm [Glasn89]. They aim at : reducing the number of rays and intersection points or reducing the calculus time of an intersection point between a ray and an object [Roth82, Glasn84]. Despite these numerous improvements, the ray tracing algorithm remains too slow on sequential computers. In addition, ray tracing algorithm should be easy to parallelize because each pixel can be evaluated independently.

2.1 Parallel ray tracing algorithm

Several parallel ray tracing algorithms have been developed in literature [Green90, Sekha95]. They can be divided into two categories : the oriented ray-flow and the oriented object-flow algorithms.

Generally, ray-flow algorithms subdivide the scene in order to distribute it over local memories [Priol89, Aykan94, Kobay93]. Each processor owns at least one region in its local memory and the objects it contains. It computes the intersection points between these objects and a given ray. If no intersection exists, the ray is transmitted to the neighbor processor which is in the path of the ray.

In the object-flow approach, the communications deal with objects instead of rays [Badou94, Kraje96b]. The processors are responsible for a certain number of rays and ask the neighbor processors to send them the lacking objects *i.e.* the objects which are not in their region.

It is also possible to write an hybrid algorithm. W. Lefer defines an hybrid algorithm with a dynamic load balancing strategy in [Lefer93]. His algorithm uses a centralized client-server scheme. The author distinguishes two kinds of independent task: the first one defines a subset of objects which might have an intersection with a given ray and the second one computes the intersection points. When a processor is overloaded, it can choose to send a subset of its load to the server. If a processor is underloaded, it asks for work to the server.

2.2 Our parallel ray tracing algorithm

We present in this subsection the main hypothesis of our parallel ray tracing algorithm. As far as we know, the best parallel ray tracing algorithm is based on a centralized client-server load balancing policy with a virtual shared memory [Badou94]. In our parallel ray tracing algorithm, the whole scene is duplicated on each processor. This conjecture does not limit our study because we focus on load balancing strategies. However, each processor computes only some regions of the scene. The total number of regions is a priori known since the whole screen is regularly subdivided. Each processor receives initially the same number of regions to perform. Despite this, the load balancing is not guaranteed as long as the computing time changes from a region to another. In order to solve the load balancing problem, our parallel ray tracing algorithm can use different dynamic load balancing schemes.

3 LOAD BALANCING MODEL

3.1 Assumptions

In the remainder of our study, we develop a general matrix iterative model for dynamic load balancing strategies applied to MIMD application classes. We take for granted that the whole application can be subdivided into elementary independent tasks like many numerous algorithms and more particularly for the ray-tracing algorithm which is our study case. This induces that $nbtask$, the number of tasks (but not their execution time) is *a priori* known at the beginning of the execution. Our model can be parametrized by any initiated load balancing strategy. The processors may have different computation and communication abilities: it would be easy to take into account the different computing power of each processor as in [Frank96]. But without loss of generality, we consider here a massively parallel machine with a set of available processors with the same computing power. We have to bear in mind that analytic model for asynchronous algorithm is not yet developed elsewhere. Some assumptions are necessary if we want to solve this problem.

We consider a message passing MIMD computer with n PE numbered 0 through $n-1$. The processors operate asynchronously but to simplify, we assume the existence of a global time. A message packet can be communicated from a PE to any other processor in a given time. In a MIMD application, processors can be either in computation phase or in communication phase. In addition, several pairs of PE can communicate simultaneously. We here discretize the time such that, at a given time t, at most one pair of processors is concerned with the communication.

We concentrate on receiver initiated load balancing algorithms. So, underloaded processors initiate load balancing step and send a request for task to an overloaded processor defined by the partner selection function. A processor is underloaded if its load (the number of tasks in its queue) is less than a given threshold named $LowThreshold$. In the same way, a processor is overloaded if its load is superior to another threshold named $HighThreshold$.

3.2 The matrix iterative model for MIMD application

The matrix iterative model tracks the task distribution of each processor in the system at each of these instants k where a given pair of communication is achieved.

The state of the system at time k is represented by the task distribution vector $w(k)$ where $w_i(k)$ is the number of tasks on processor P_i at k. The model computes the task distribution $w(k)$ for all k in terms of task transfer between processors due to load balancing and due to the departure of achieved tasks at each processor. Task distribution at time $(k+1)$ noted $w(k+1)$ can be expressed as a function of the three elements :

- $w(k)$;
- the task transfer due to load balancing ;
- the departure of tasks between the k^{th} and the $(k+1)^{th}$ steps.

We recall that our model does not accept dynamical arriving of tasks. More formally the dynamic task model for each processor P_i can be written as follows:

$$\begin{aligned} w_i(k+1) &= w_i(k) - \textit{tasks sent by } p_i \\ &+ \textit{tasks received by } p_i \\ &- \textit{tasks departing during } [k, k+1] \end{aligned}$$

which can be rewritten as the following form:

$$w_i(k+1) = w_i(k) + \alpha_{ij}(k)(w_j(k) - w_i(k)) - c_i$$

where j is the number of i's partner at the instant k if this partner exists, α_{ij} is a non negative constant such that $\alpha_{ij}(k)(w_j(k) - w_i(k))$ means the amount of exchanged work if a partner of P_i exists at the instant k. Else, $\alpha_{ij} = 0$. c_i is the number of achieved tasks by the processor i due to the application.

Notice that due to our time discretization assumption, our formulas are simplified, compared to the Cybenko and Govindan ones.

The previous equation can be reformulated as:

$$w(k+1) = M(k)w(k) - C$$

where the load system at the instant k is represented by the vector $w(k) = \begin{pmatrix} w_0(k) \\ \vdots \\ w_i(k) \\ \vdots \\ w_{n-1}(k) \end{pmatrix}$;

and at each time k the matrix $M(k)$ reflects the transfer of tasks initiated by the load balancing algorithms and can be parametrized to represent various load balancing schemes.

$$M(k) = \begin{pmatrix} M_{00} & M_{01} & \dots \\ M_{10} & M_{11} & \dots \\ \vdots & \vdots & \ddots \end{pmatrix} \text{ of size } n \times n.$$

In order to built the matrix M two cases are to be distinguished:

- case 1 All processors are loaded;
- case 2 A single processor is underloaded.

Case 1: If $\forall i,\ w_i(k) > LowThreshold$ then $M(k) = Id$.

All processors are active and then no communication between them is needed. Hence, M is the identity matrix.

Case 2: If $\exists i,\ w_i(k) < LowThreshold$,

$\exists!\ j\ such\ that\ w_j(k) > HighThreshold$ and P_j is a selected partner for P_i following a given dynamic load balancing strategy.

Let be $\bar{w}(k) = \begin{pmatrix} \bar{w}_0(k) \\ \vdots \\ \bar{w}_{n-1}(k) \end{pmatrix}$ the uniform work distribution, and

$$\bar{w}_i(k) = \begin{cases} \sum_j w_j(k)/n + 1 & \text{if } i < \sum_j w_j(k) \bmod n \\ \sum_j w_j(k)/n & \text{else.} \end{cases}$$

When more than one processor is overloaded, $M(k)$ reflects the transfer of tasks initiated by the different underloaded processors.

The effect of load balancing strategy at time k can be defined by the distance $q(k) = d(\bar{w}(k), w(k))$. Hence,

$$q(k) = \sqrt{\sum_{i=0}^{i=n-1} (\bar{w}_i(k) - w_i(k))^2}.$$

Smaller is $q(k)$, best are the effect of load balancing.

3.3 Performance Measures

The basic performance measure used is the total sequential execution time T_{total}. Parallel execution time T_p, the time among p processors, can be exhibited to evaluate the algorithm. Under a given load balancing strategy, we measure in addition the number of load balancing steps $N_{load-bal}$, the number of asked processors $N_{ask-proc}$, the number of communicated tasks $N_{com-task}$. In all cases, we also consider the speedup and time efficiencies.

All these measures are defined as follows and are related to a given load balancing strategy and a given partner selection scheme. Let be $nbtask$ the total number of tasks of the whole application and $ftask(i)$ the computation time of the task i.

- T_{total}: it is given by the following expression $T_{total} = \sum_{i=1}^{nbtask} ftask(i)$.
- T_p: this measure corresponds to the consumed time until step k when the last processor determines the termination, on a p processors system. This time depends on the computation time due to application and the communication time due to load balancing as well. In other words, if P_j is the processor which detects the termination, *i.e.* P_j is the last processor invoking an unsuccessful load balancing step, let be nb_j the number of tasks performed by P_j and $Tlb_p(j)$ the time for load balancing spent by processor P_j on a p processors system. Then, $T_p = \sum_{l=1}^{nb_j} ftask(l)\ + Tlb_p(j)$.
- $N_{load-bal}$: it corresponds to the number of steps where $\exists\ P_i\ such\ that\ w_i(k) = 1$ and $w_i(k+1) = 0$
- $N_{ask-proc}$: This measure tightly depends on the partner selection scheme and on the state of the task distribution vector. It indicates the number of attempted partners. We illustrate this parameter later by a particular partner selection scheme. Let be $N_{ask-proc}(k)$ the number of requested processors at the time k, that is the number of asked processors until an overloaded processor p_l is found. Then $N_{ask-proc} = \sum_{k=1}^{N_{load-bal}} N_{ask-proc}(k)$
- $N_{com-task}$: This parameter is related to the task distribution vector and to the task splitting strategy. $N_{com-task} = \sum_{k=1}^{N_{load-bal}} \alpha_{il} w_l(k)$ where l is the number of the loaded processor found at step k by processor i.
- speedup: $speedup = \frac{T_{total}}{T_p}$
- efficiency: $efficiency = \frac{T_{total}}{T_p \times p}$

3.4 A server initiated algorithm based on annealing partner selection

In this server initiated strategy, the load balancing request is initiated by idle processors which ask for some work from the overloaded processors. We consider here that a processor is idle or underloaded if its load is null *i.e.* $LowThreshold = 0$. Similarly, we consider a processor is overloaded iff its load is greater than 1 *i.e.* $HighThreshold = 1$. In order to find an overloaded processor, the same algorithm is used by each idle processor.

We may notice that whenever a load balancing step proves unsuccessful, no more load balancing step can be successful. In fact, a processor P_i, having a number of tasks equal to zero, realises $n-1$ failed communications if, and only if, it asks all the $n-1$ different processors of the system without any success. This is possible only if all the processors are idle. In other words, all processors have a number of tasks equal or inferior to 1 : therefore, they cannot definitively share it.

In order to built the matrix M for the server initiated algorithm, two cases are to be distinguished:

- case 1: all processors are loaded ;
- case 2: a single processor is underloaded.

Case 1: If $\forall i,\ w_i(k) > 0$ then $M(k) \ = \ Id$.

Clearly all the processors are active and then no communication between processors is needed. Hence, M is the identity matrix.

Case 2: If $\exists i,\ w_i(k) < 1$,

$\exists!\ j\ such\ that\ w_j(k) > 1$ and j is a selected partner for i following the server initiated strategy.

Processor P_i is underloaded and P_j the first asked processor by P_i is able to share its load.

Processor P_i receives then from P_j a fraction α_{ij} of its tasks. Then:

$$M_{ij} = M_{ji} = \alpha_{ij} = \alpha_{ji} = \frac{1}{2}$$
$$M_{ii} = M_{jj} = 1 - \alpha_{ij} = \frac{1}{2}$$

All the other elements corresponding to i and j-rows are zero.

No exchange is realised elsewhere and all the entries of the matrix except for the elements of diagonal are zero:

$$\forall k \neq i \text{ and } k \neq j,\ \forall l \neq k,\ M_{kl} \ = \ 0.$$

The global load of the system is preserved:

$$\forall k,\ if\ k \neq i\ and\ k \neq j\ then\ \alpha_{kk} = 1.$$

The reader shall notice when an overloaded processor receives a request from an idle processor, it gives the half of its load to the idle one. In fact, we have shown in [Kraje96a] that the server initited strategy is more efficient when half of the load is exchanged rather than only one task is exchanged between the overloaded and the idle processors.

The partner selection whe choose is very simple: an idle processor looks for the first processor which is able to share its load. This strategy is fast, the number of exchanged messages is small and this is very important when the parallel machine uses a large number of processors. We justify this choice by the fact that Eager *et al.* have shown that the best partner selection cannot sensitively improve the efficiency of the dynamic load balancing strategy since the number of exchanged messages is very large [Eager86].

Example: we give here an example to show how the matrix of work distribution is built in the case of server initiated algorithm with annealing partner selection and with the hypothesis that the overloaded processor gives the half of its tasks *i.e.* $\forall i,\ j \neq i,\ \alpha_{ij} = \frac{1}{2}$ or 0. The matrix associated to the following work distribution $w(k) = \begin{pmatrix} 5 \\ 0 \\ 2 \\ 8 \end{pmatrix}$ is:

$$M(k) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \frac{1}{2} & \frac{1}{2} & 0 \\ 0 & \frac{1}{2} & \frac{1}{2} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

The work distribution at time $k+1$ is:
$w(k+1) = M(k) \times w(k)$, and we obtain:
$w(k+1) = \begin{pmatrix} 5 \\ 1 \\ 1 \\ 8 \end{pmatrix}$.

Proposition 1
the server initiated algorithm based on annealing partner selection converges to the uniform work distribution vector.

Proof:
Straightforward, it suffices to show that $q(k),\ k = 1, ..., n$ define a decreasing and minor sequence.

To analyze this strategy in details, all the indicators detailed previously described, are computed for a ray tracing application in section 4.

3.5 A dynamic centralized algorithm

Client-server strategies are easy to implement and are, in a large number of cases, efficient. Therefore, centralized strategies are widely used by the scientific community [Eckst94, Hamdi97].

Principles of this method:

- the server has the set of tasks in its spool. It sends a task to a processor when it makes a request ;
- The other processors are all clients and request the tasks to perform, as soon as they are idle ;
- the program ends when there is no more task in the server spool.

This kind of algorithms is very efficient. The key advantage of client-server scheme is that it makes a good load distribution without any hypothesis on tasks (execution time, memory resource, ...). Nevertheless, if the granularity is too fine or the number of processors is too high, the server can become a bottleneck.

The use of client-server strategies is, in general, very simple. The update policy of relevant information is easy since it is performed only on one processor. The efficiency of this method is good because the load balancing overhead remains low since the client processors are not penalized by the dynamic management of the load. But, when the number of processors grows in a sensitive way, the server can become a bottleneck. In particular, the update policy of the information distributed over all the processors can be very expensive.

The client-server and server-initiated strategies are both shown to be very closed experimentally. In the next section, we compare the results derived from the analytical model with respect to the experimental results. For this strategy, similarly to the previous one, all the indicators detailed in previous section are computed in section 4.

Table 1: Theoretical and experimental result performances for Sphereflake

Strategy	Sphereflake					
	theoretical		experimental		error	
	32 procs.	64 procs.	32 procs.	64 procs.	32 procs.	64 procs.
No load balancing (sec.)	183.17	93.96	183.17	93.96	0	0
speedup	23.51	45.82	23.51	45.82	0	0
efficiency	73	72	73	72	0	0
Server initiated algo (sec.)	138.36	71.41	139.07	71.61	0.005	0.002
speedup	31.12	60.29	30.96	60.12	0.005	0.003
efficiency	97	94	97	94	0	0
T_p	132.38	64.77	130.96	65.38	0.01	0.009
$N_{load-bal}$	125	153	121	156	0.032	0.02
$N_{ask-proc}$	1573	5653	1391	5257	0.115	0.07
$N_{com-task}$	148	112	127	107	0.14	0.05
Client-server algo (sec.)	139.82	69.75	143.47	71.11	0.026	0.019
speedup	30.79	61.73	30.01	60.55	0.025	0.02
efficiency	96	96	94	95	0.02	0.01

4 COMPARISON BETWEEN MODEL AND EXPERIMENTAL RESULTS

In this section, validation results for the matrix iterative load balancing model applied to the MIMD ray tracing algorithm are presented. These studies contrast model derived performance data with experimentally obtained data.

We have experimented the server initiated and client-server load balancing algorithms presented in this paper, on a Connection Machine (CM-5). All the tests have been made with different scenes which are composed of 50 to 450 primitives. We show here the results obtained for two particular scenes. We have used two scenes proposed by E. Haines which are mount and sphereflake [Haine87]. The sphereflake's scene is made up of 1 polygon and 91 balls. The mount's scene is made up of 32 polygons and 4 balls. Our parallel ray tracing algorithm can use a Constructive Solid Geometry (CSG) model and a Boundary representation (B-rep) to modelize the scene. These scenes have been rendered with a 512×512 resolution. For those tests, the maximum tree depth is 5 (where the eye ray is of depth 1). We present this particular experimental setting, in order to give concrete elements about the experiments; however the qualitative results of our experiments were observed in a variety of other scenes.

Our algorithm uses a regular grid to accelerate the rendering time of an image [Amana87] and before processing the intersection point between an object and a ray, the ray is tested against the bounding box of the object [Roth82]. First, the sequential computational time for the two considered scenes named *Sphereflake* and *Mount* are respectively 4305,5 sec and 3750.8 sec.

Second, we must add that in order to evaluate theoretical performances the time computations of tasks $ftask(i)$ are fixed (and given by the time measured on one processor) for our two particular scenes. We would relaxe this hypothesis in a future work where a very large sample of randomly generated scenes should be considered and strengthened by a statistical study.

Tables 1 and 2 respectively show both the derived model performances and the derived experimental performances for *Sphereflake* and *Mount* scenes.

We observe that experimental and theoretical result performances are very closed. The gap between model and measured values was found to be less than 7 % for execution time, speedup and efficiency. However, other indicators such as the number of load balancing steps, communications and asked processors are approached with about 20 % error.

Figure 1 displays for Sphereflake and Mount scenes, three curves of theoretical CPU time representing no load balancing , server initiated and client-server behaviors among 8 through 128 processors. We mainly observe that theoretical performances of client-server and server-initiated strategies are very closed and both improve performances widely. Notice that these results are also experimentally observed.

Table 2: Theoretical and experimental result performances for Mount

Strategy	Mount					
	theoretical		experimental		error	
	32 procs.	64 procs.	32 procs.	64 procs.	32 procs.	64 procs.
No load balancing (sec.)	246.45	124.88	246.45	124.88	0	0
speedup	15.22	30.04	15.22	30.04	0	0
efficiency	48	47	48	47	0	0
server initiated (sec.)	124.01	63.97	126.35	68.51	0.018	0.07
speedup	30.25	58.63	29.69	54.75	0.018	0.066
efficiency	95	92	93	86	0.021	0.065
T_p	113.57	54.70	113.59	55.66	0.0002	0.017
$N_{load-bal}$	286	373	240	338	0.16	0.096
$N_{ask-proc}$	2223	7437	1551	5982	0.30	0.20
$N_{com-task}$	566	537	454	440	0.19	0.18
Client-server (sec.)	125.92	65.47	130.34	66.87	0.035	0.021
speedup	29.79	57.29	28.78	56.09	0.033	0.02
efficiency	93	90	90	88	0.032	0.022

5 CONCLUSION

This paper presents a performance tool based on the matrix iterative model that helps in the elaboration and performance evaluation of dynamic load balancing algorithms for MIMD applications. Such a performance tool helps parallel application developers in exploring various load balancing strategies without implementing them. In addition, the best load balancing strategy can be *a priori* derived only from sequential data. The model and performance measures are general and then can be used for a large range of applications and load balancing strategies. We illustrated here our model by studying a MIMD parallel ray tracing algorithm. We mainly compared the result performances of two dynamic load balancing schemes. The model predictions were compared against an experimental system consisting of two scenes running on a Connection Machine (CM-5) of 128 processors. The gap between model and measured values amounts to less than 7 % for execution time, speedup and efficiency. However, other indicators that the number of load balancing, communications are approached with about 20 % error. In a future work, we plan to improve this result by proposing some refinements of the matrix model.

Our results show that receiver initiated algorithms improve performances compared to no load balancing and give better efficiency both on theoretical and experimental point of view. In order to provide a more valuable assessment of our model, we will propose a statistical study using the gaussian and log-gaussian distribution to fixe the parameter $ftask(i)$ (the execution time of the ith task).

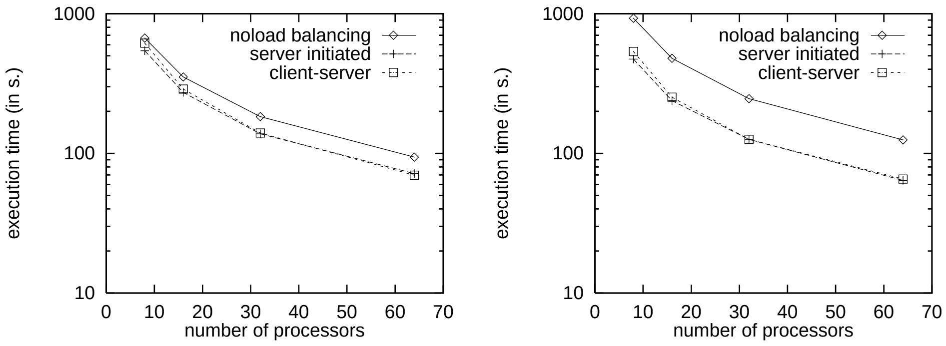

Figure 1: Comparing execution times for Sphereflake and Mount with respect to various LB strategies

ACKNOWLEDGEMENTS

All parallel computations were carried out using the CM-5 of the C.N.C.P.S.T. (Centre National de Calcul Parallèle en Sciences de la Terre).

REFERENCES

[Amana87] Amanatides, J, Woo, A: *A Fast Voxel Traversal Algorithm for Ray Tracing*, Proceedings of EUROGRAPHICS 87, Elsevier Science Publ., pp. 65–77, 1987.

[Aykan94] Aykanat, C, Isler, V, Özgüç, B: *Efficient parallel spatial subdivision algorithm for object-based parallel ray tracing*, CAD, vol. 26, pp. 883–890, Dec. 1994.

[Badou94] Badouel, D, Bouatouch, K, Priol, T, Arnaldi, B: *Distributing data and control for ray tracing in parallel*, IEEE Computer Graphics and Applications, pp. 69–77, July 1994.

[Cyben89] Cybenko, G: *Dynamic load balancing for distributed memory multiprocessors*, Journal of Parallel and Distributed Computing, vol. 7, pp. 279–301, 1989.

[Eager86] Eager, D. L, Lazowska, E. D, Zahorjan, J: *Adaptative Load Sharing in Homogeneous Distributed Systems*, IEEE transactions on software engineering, vol. SE-12, n. 5, pp. 662–675, may. 1986.

[Eckst94] Eckstein, J: *Parallel branch-and-bound for mixed integer programming*, SIAM News, vol. 27, pp. 12–15, 1994.

[Frank96] Franklin, M. A, Govindan, V: *A general matrix iterative model for dynamic load balancing*, Parallel computing, vol. 22, pp. 969–989, Oct. 1996.

[Glasn84] Glassner, A. S: *Space subdivision for fast ray tracing*, in IEEE Computer graphics and applications, pp. 15–22, oct. 1984.

[Green90] Green, S. A, Paddon, D. J: *A Highly Flexible Multiprocessor Solution for Ray Tracing*, The Visual Computer, vol. 6, pp. 62–73, 1990.

[Glasn89] Glassner, A. S: *An Introduction to Ray Tracing*, Academic Press, 1989.

[Hamdi97] Hamdi, M, Lee, C. K: *Dynamic load-balancing of image processing applications on clusters of workstations*, Parallel computing, vol. 22, pp. 1477–1492, Jan. 1997.

[Haine87] Haines, E: *A proposal for standard graphics environments*, IEEE CG&A, vol. 7, pp. 3–5, Nov. 1987.

[Kobay93] Kobayashi, H, Kubota, H, Horiguchi, S, Nakamura, T: *Load balancing based on load coherence between continuous images for an object-space parallel ray-tracing system*, IEICE Transactions on Information and Systems, vol. V, pp. 1490–1499, 1993.

[Kraje96a] Krajecki, M, Habbas, Z, Herrmann, F, Gardan, Y: *Distributed Load Balancing Strategies for Parallel Ray-tracing*, 9th International Conference on Parallel and Distributed Computing Systems, Dijon (France), pp. 50–55, sep. 1996.

[Kraje96b] Krajecki, M, Habbas, Z, Herrmann, F, Gardan, Y: *équilibre de charge pour le lancer de rayons: étude théorique et expérimentale*, Calculateurs parallèles, vol. 8, no. 1, pp. 69–87, 1996.

[Priol89] Priol, T, Bouatouch, K: *Ray tracing on parallel computers: a performance study*, Tech. Rep. 451, IRISA Rennes, Jan. 1989.

[Lefer93] Lefer, W: *Parallel Strategies for the Ray Tracing, IEEE Visualization '93 Parallel Symposium*, San Jose (USA), oct. 1993.

[Roth82] Roth, S. D: *Ray casting for modeling solids, Computer Graphics and Image Processing*, vol. 18, n. 2, pp. 109–144, feb. 1982.

[Sekha95] Sekharan, C. N, Goel V, Sridhar, R: *Load balancing methods for ray tracing and binary tree computing using PVM*, vol. 21, pp. 1963–1978, 1995.

[Whitt80] Whitted, T: *An improved illumination model for shaded display*, in Commun. ACM, pp. 343–349, 1980.