

LUX: An Heterogeneous Function Composition Parallel Computer for Graphics

Stéphane Mancini and Renaud Pacalet

Ecole Nationale Supérieure des Télécommunications, Paris, France

Abstract. In this paper, we present an heterogeneous parallel computer dedicated to high realism computer graphics. A small network, with a reduced chip set, allows us to reduce rendering time by a very attractive factor. The low level mechanisms of the network are designed to manage the wide variety of data and algorithms used in computer graphics. Some nodes of the network may be specialized in the most time consuming parts of the algorithm and have specific data paths. Thanks to the function composition scheme, we unify both the management of specialization and of parallelism. Those mechanisms allow flexibility and easy design of programs.

1 Introduction

The graphic community is in need of very high computational power to achieve high realism pictures at interactive rates. We observe that single general purpose computers can't provide the needed performance, and general parallel computers hardly reach them, and for a high financial cost. In this paper, we present an heterogeneous parallel computer which may reach expected performances, with a reduced chip-set and this for a low financial cost.

We are particularly interested in the speed up of Ray-Tracing algorithms. They simulate the propagation of light in an environment, the scene, made of geometric primitives. The most time consuming part of such algorithms is the computation of the intersection between the rays and the scene. The computation of surface lighting characteristics can also be quite expensive. The graphic community has developed many algorithms to reduce the rendering time, essentially by reducing the number of ray-object intersections, but it's still quite high.

We propose to design specialized data paths to speed-up the most time consuming parts of the algorithm. Those specialized units are interconnected through a dedicated network to form an heterogeneous parallel computer.

In the second part of this paper, we briefly recall the Ray-Tracing algorithm and the way we want to accelerate it. In part 3, we show how we manage specialization and scheduling with the function composition scheme. In the two next parts (4 and 5), we precise the architecture and the load balancing strategy. At last, before the conclusion, we show the tool developed to help us to make architectural choices.

2 Motivation

2.1 Ray Tracing

The Ray-Tracing algorithm simulates the propagation of light in a scene. More exactly, we follow the inverse path of light to determine which object is seen by the observer at a given pixel of the picture. A primary ray is sent, originating at the observer and passing through the considered pixel, and we compute its intersection with the scene to determine which object is visible. Once the intersection is found, we determine the amount of light at this point by sending secondary rays. We send rays to each light source to compute the intensity of direct lighting and we send refracted and reflected rays, depending on the local illumination law of the surface, to determine the amount of indirect lighting. The Ray-Tracing algorithm is recursive, and we stop the recursion when we reach a given criteria (recursion depth, contribution threshold, ...).

This basic algorithm is improved in several ways to deal with effects like indirect lighting through glasses and other realistic effects but the main point is that they're built over the computation of the intersection between a ray and the scene.

The Ray-Tracing algorithm needs very high computational power. Indeed, we need to compute the intersection of several millions of rays with millions of geometric primitives. That's why the graphic community has developed many algorithms to reduce the number of intersections to compute but the rendering times are still quite high. To accelerate it again, we need to increase computational power.

2.2 Parallelization

The Ray-Tracing algorithm can easily be parallelized. For example, the computation of two pixels is independent, and the intersection of a ray with two objects can be done in parallel. In the literature, we find two main strategies, implemented on general purpose parallel computers:

- “Control parallelism”: part of the picture are mapped to nodes which exchange objects
- “Data parallelism”: the scene is split over nodes which exchange rays

The choice between such solutions is often empirical but some ([2]) proposed a mix strategy where rays and objects move at the same time. The main advantage of general parallel computers is that programmers can reuse existing graphic software, especially in the first strategy. But the speed-up is at best linear with the number of nodes and mainly depends on the amount of memory of each node and on the buses bandwidth.

2.3 Specialization

Mainly all previous specialized solutions ([4], [3], [1]) failed because they didn't offer enough flexibility, which is essential in computer graphics. For example, ([3]) hard-wired Constructive Solid Geometry (CSG) but didn't provide algorithmic acceleration. [1] tried to design very specialized processors but that time technology didn't allow their realization. Also, the bottleneck would have been the host computer because their processors didn't accelerate all the parts of the algorithm at the same rate.


```

I_Intersect_Ray_Triangle(ray,triangle,destination)
segment=intersect_triangle(ray,triangle)
Set_Argument(destination,segment)

I_Intersect_Ray_CSG_Add(ray,node,destination)
T_add=Create_Task(Add_segment,nil,nil,destination)
T_left=Create_Task(Intersect,ray,node-left,(T_add,0))
T_right=Create_Task(Intersect,ray,node-right,(T_add,1))

```

Fig. 2. Sample pseudo code

3.2 Tasks

Our system provides mechanisms to manage the stack created by the recursive run on the scene tree. As we don't want to centralize the stack, we give a task informations on the task to compose with.

To call a function, we create a task which is a data structure composed of two elements: an identifier of the function it uses and an array of arguments. We will note:

$$F = (f, (\arg_0, \arg_1, \dots, \arg_n))$$

f is the function associated with the task F . Arguments are pointers on the data needed by the function. A task is said sleeping until all the arguments are set, otherwise the task is said activated and, then, may be processed. Task composition is achieved by indicating a function the argument of the task to compose with. When a task produces a result, it sets the corresponding consumer task argument to the result's address. The consumer task may then be activated and move over the network to be processed somewhere.

Figure 2 illustrates this mechanism with a program sample. It shows the instantiation's pseudo-code of the ray/primitive or ray/CSG object intersection. The function `Create_Task` returns a pointer on the created task and the notation (\underline{Task}, n) points on the n th argument of the task. The function `Set_Argument` creates a low level task which sets the argument of a task to a pointer on the specified list. We note that the two tasks $\underline{T_left}$ and $\underline{T_right}$ can be processed in parallel because they are both activated. The two segments "left" and "right" are destroyed by the task $\underline{T_add}$.

3.3 Task Migration

The management of specialization is distributed over the nodes. When a node receives a tasks, it decides or to send it to a distant node or to process it locally. The diversity of load balancing strategies and migration law is coded in the function's genus. We have the types: "Sedentary", "Universal", "Specialized" and "Follower". A "Sedentary" task can only be executed on a specific node. At opposite, an "Universal" task can be processed everywhere. The management of specialized hardware is done through "Specialized" tasks which migrate according to the type of their arguments. And, at last, a "Follower" task migrates until it reach its first argument. The type of a task can have a great impact on the performances of the system.

3.4 Data Structures

We type data with three fields: (Class, Species, Type), called a genus. The data structures are unified in lists which are coded linearly, like we would write them by hand. Tasks are also coded within lists. A task is coded in a list of three elements: a function, the list of arguments and a data list. An argument can be a pointer on a list or a reference to a sub list of the data list. This late one allows us to move task and data together. We distinguish system tasks from applicative tasks with the function's genus.

4 Architecture

4.1 General

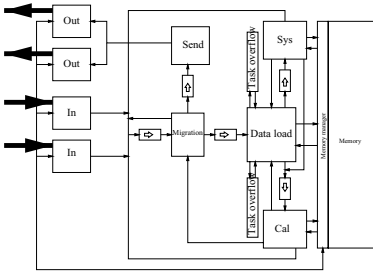


Fig. 3. Node architecture

LUX is an heterogeneous MIMD parallel computer and nodes of the network are specialized in parts of the rendering process. Nodes are interconnected through a service network and a data network. The service network interconnects all the processors through a ring and is dedicated to system communications like load balancing or specialization. The data network is a hierarchical network and is dedicated to local communications. In order to unify communications, processors only exchange tasks; even simple data are enclosed in tasks. The topology of

the data network is free and we can build sub networks which can be considered by the rest of the network as single nodes.

Memory is distributed over the nodes of the network and addressing is unified through pointers; A pointer refers to a processor and to the local address of the data. Consequently, we have to transfer data and store them during computation of tasks. To do so, the private memory of each node is split between local data and cached data. This cache is made of copies of lists originally located on other processors. We note that the function composition scheme doesn't need the management of cache coherency. A data propagates on the tree and is destroyed when it has run everywhere it's needed. So, a list can be created or destroyed but not modified, except tasks. This policy may increase data transfers and memory resources but allows simple and fast mechanisms. Memory resources of processors is free and the larger they are the more we reduce communications. Nevertheless, high memory quantities, like mass memory, can be managed by specific processors to deal with large databases.

4.2 Node Architecture

The main functionality of a node is the task pipeline execution which steps are "Migration" step, "Load" step and "Compute" step.

At the "Migration" step, the node determines the target node depending on the migration law of the task and on the load balancing strategy. A task may continue the pipeline or be sent to another node.

Next, at the “Load” step of the pipeline, the node loads distant data. Cached lists are stored in cache cells. When there’s a cache miss, the task is removed from the task pipeline and the node creates tasks to load the data. When the cell is loaded, the tasks which use it continue the pipeline. If there aren’t enough available cells, the task is put on the bottom of a stack for a further try. We note that cache cells are locked until the end of the computation of tasks which use them.

The task function is computed at the third step. To simplify the design of processors, we suggest to standardize the units which manage all the system mechanisms and adapt the applicative unit, like shown on figure 3.

We introduce FIFOs between each stage of the task pipeline to smooth access on the different units. To prevent interlocking, we use FIFOs of virtual infinite capacity. Indeed, if the stacks were of finite size, when they would be full, the node couldn’t receive neither create any active task which may solve the situation. So, to prevent interlocking, when one stack is full, a part of its content is saved in external memory. The saved content of the stack is restored when the stack is above a threshold. So, stacks are managed both by a local unit and by a distant manager. This way, we can consider that the stacks are of potential very large size and we prevent interlocking efficiently.

5 Load Balancing Strategy

5.1 The Algorithm

The resolution of the function migration law of a task may give us several target processors where to compute the task. We have to make a local choice in order to minimize the global computing time. The difficulty is that the different processors may have various computing powers. We choosed a policy based on the dynamic evaluation of the computing time of tasks depending on specialization and target processor.

For each possible processor, we estimate the time it has to run and the target processor is the one with minimum total time. The estimated run time of each processor is given by the following equation:

$$T = \sum_i n_{I_i} * T_{I_i}$$

Where n_{I_i} is the number of instantiation i of the tasks in the pipeline and T_{I_i} the evaluation of the computing time of the instantiation. In practice, we only take into account the time of the applicative tasks and the run time of a node is incrementally estimated when a task passes the migration step and when the computing of a task ends.

So, each processor needs to have knowledge about the state of others. Those informations are propagated through the service network at low rate. To reduce communications, we only send the estimated run time when its difference with the previous one is superior to a given threshold.

To take into account transfers on the buses, we attribute a factor of correction to the evaluation of the run time of each distant node. This policy allows us both to avoid ping-pong effects while a node discharge its tasks to a node with same specialization and to maintain locality of computations on subnetworks.

5.2 Qualitative Study

We have to ensure us that this algorithm is stable and allows a good load balancing of the nodes. Here, we don't show a mathematical proof but give some qualitative results. Instability would come from the delay between the moment we take a decision and the moment we have knowledge of its results. We send tasks to the wrong node until we get an evaluation of the estimated run time. One of the determinant factor is the time of the travel of the task from its originating node to the target node. The more the buses are loaded, the more we make wrong decisions. The delay due to the communication of the time after its estimation, at the end of the processing of the task, also has influence until the estimated time of the instantiation is stable. To reduce the delay, we can try to predict the computing time of distant nodes. To do so, we modify the evaluation of the computing time at the moment we make the decision. When we set the target node of a task, we add to the distant node run time the task's distant time.

The presence of many nodes with same specialization on the network has a drawback: we increase the communications on the buses. Indeed, a data used by many tasks may be loaded on each node cache cell and travel through the bus each time. On the other hand, the Ray-Tracing algorithm produces many tasks which use the same rays in a short time. The wrong choice of the target node, previously disputed, may send all those tasks to the same node and save some bandwidth.

At this point, we understand that it's quite difficult to give an a priori estimation of the performances of our system. That's why we provide a simulation tool to help us to make architectural decisions.

6 Simulation Framework







6.1 Simulator Engine

We've developed a simulation framework of the system to help us to make choices about network architecture, specialization, computational power, load balancing strategies and various parameters value. The simulator engine of the framework is written in C and the configuration tool in Tcl/Tk. We choosed C rather than VHDL, which would allow a fine description from top level to gates, because VHDL would have been too slow. The C code is divided in three parts:

- The core simulation engine
- Configuration and instantiation of the network
- The rendering program, collection of function's instantiations

The simulator is an event driven simulator and we attribute instantiations a time model. This model allows to simulate the different computational powers of each processor, depending on its supposed specialization. Each action is timed and the simulator freezes the different units until they finished their job. So, with probes set at various places, we can have a fine analysis of task migration and data flows.

Table 1. Architecture and scene influence

Architecture	Bus width	Scene	Picture size	Time	Chip load	Bus load
	32	scene ₁	50*50	27 ms	$T = 1$	$B = 0.23$
			100*100	118 ms	$T = 1$	$B = 0.1$
	64	scene ₂	100*100	41 ms	$T = 0.9$	$B = 0.7$
			100*100	41 ms	$T = 0.9$	$B = 0.4$
	32	scene ₂	100*100	41 ms	$T = 0.9$	$B_r = 0.5$ $B_l = 0.3$
	32	Id.	Id.	41 ms	$T = 0.9$	$B_r = 0.5$ $B_l = 0.1$ $B_u = 0.2$
	32	Id.	Id.	44 ms	$T = (0.4, 0.5)$	$B = 0.9$
	64	Id.	Id.	30 ms	$T = 0.6$	$B = 0.6$
	32	Id.	Id.	30 ms	$T = 0.6$	$B_r = 0.6$ $B_l = 0.4$
	32	Id.	Id.	30 ms	$T = 0.6$	$B_r = 0.6$ $B_l = 0.2$ $B_u = 0.3$

6.2 Program Design

The first step to write a program is to list the different functions involved and attribute them an identifier. The second step is the design of each function's instantiations. Some instantiations may create sub tasks to be executed and that for, they create a task with the function field set to the corresponding identifier. The system will automatically manage task's activation, like shown with the sample code on figure 2. The third step is the compilation and installation of the different instantiations on their processors. Finally, we establish a local link between each function and its instantiation, depending on the migration law. We note that, as we need to have an overall view of the different kind of processors to identify the functions, the steps declaration, compilation and establishment of link are done by a supervisor.

Finally, each processor communicates its specialization to others through the service network. An important point is that the design of the program is independent from the data network. Task migration, data access and load balancing is automatically managed by low-level mechanisms.

7 Results

7.1 Simulation Parameters

In this section, we give some results of simulation to show the influence of various parameters like scene description, network architecture and program design.

The simulations are done with a scene which contains a ground plane, a tower and a space station and is lighted with a punctual light source. The tower is made of boxes CSG added and subtracted. The space station is a mesh of about 10000 triangles split in an octree. In scene₁, the octree has a maximum subdivision depth of 6, with a maximum of 20 triangles per node. In scene₂ the maximum depth is 8 with 10 triangles per nodes. The nodes of the network are:

T Ray \cap triangle	P Ray \cap polyhedra	C CSG
Tr Transformation	F Filter	R Rendering

The T node computes the intersection between a ray and a triangle in 4 clock cycles at a 100 MHz frequency, which is an extreme computational power. To simplify the paper, all the system is synchronous, at 100 MHz, and all the buses have the same bandwidth, specified on the table of results. The computing power of all other processors is not critical but they have side effects. The load balancing is the one in its basic version. On all the simulations, for the clarity of the paper, we only give measures for the triangle processor and buses (the buses subscripts are: l=left, r=right, u=up and b = bottom).

7.2 Analysis

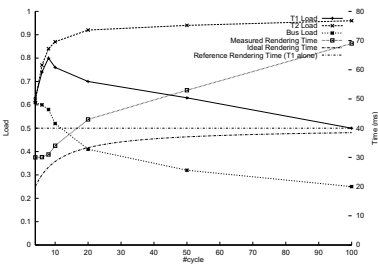


Fig. 4. Load balance

is the triangle processor. Their behavior is different when we add a second triangle processor because the bottleneck becomes the bus bandwidth. To reduce the rendering time, we have to reduce the bus load. To do that, we can increase the bus bandwidth or split the bus.

We observe that the balance between the two triangle processors is quite good: we have less that 1% of difference of efficiency when they have the same computational power. The figure 4 shows the evolution of the balance between two processors of same specialization but with different computing power. We set the T_1 processor to the highest computing power (one intersection in 4 clock cycles) and we decrease the computational power of T_2 . We observe that when the two processors have similar computational power, they behave the same. When the difference is too important, the slowest processor becomes the bottleneck. This behavior comes from bad evaluations of the computing time of tasks which leads to wrong choices. If the wrong decision of the migration step leads to a migration, the choice can be corrected by the target node. But, if

The table 1 shows us the influence of the relation between computation power and bus bandwidth. The results obtained with the first architecture shows us that the bottleneck is the T node when we do the computation with the scene scene₁ which contains a few complex primitives. In scene₂, we have many small primitives and the bottleneck becomes the bus which as to transfer the primitives and the results of intersections.

The three first architectures have approximately the same performances because the bottleneck

it leads to local execution of the task, once it's in the pipeline of the slowest processor, it stays in it and becomes the bottleneck.

Simulations show us that we don't significantly increase the network performances by modifying simulation parameters. So, we have to improve the network mechanisms. A solution would be to have a better model of the computing time which would take into account the complexity of the arguments of the functions. We could insert a "refining" step in the pipeline, after the migration step and before the load of operands, which would give a better approximation of the computing time of the task. We are currently investigating another solution which performs regular correction of the load balance. In case of known unbalance, we remove tasks from the pipeline and send them back to the migration unit which would make better choices for a better balance. This late approach seems to be very promising.

8 Conclusion

We have presented a new kind of parallel computer dedicated to a class of application. Rather than to parallelize Ray-Tracing in screen or object space, we chose to parallelize on nodes and leaves of the scene tree. This allows us to have specialized processors dedicated on nodes or leaves and to preserve flexibility. The load balancing strategies are automatically managed by the system and the user can have a fine control on them by tuning the data flows and program design. The user may also add its own strategies with specific system tasks. However, our strategy allows us to reduce the time to ray trace pictures by a very attractive factor, like shown by simulations. We note that although initially designed for Ray-Tracing, our architecture could be used for a wide variety of algorithms.

References

- [1] Kadi Bouatouch, Yannick Saouter, and Jean Charles Candela. A VLSI chip for ray tracing bicubic patches. In *Proc. of Eurographics '89*, pages 107–124. W. Hansmann and F. R. A. Hopgood and W. Strasser, 1989.
- [2] Alan Heirich and James Arvo. A competitive analysis of load balancing strategies for parallel ray tracing. *Journal of Supercomputing*, 12(1 and 2):57–68, 1998.
- [3] Gherson Kedem and Jonh L. Ellis. The raycasting machine. In *Proc. of the IEEE International Conference on Computer Design: VLSI in Computers ICCD '84*, pages 533–538. IEEE Computer Society Press, 1984.
- [4] Tadashi Naruse, Masaharu Yoshida, Tokiichiro Takahashi, and Seiichiro Naito. Sight — a dedicated computer graphics machine. *Computer Graphic Forum*, 6(4):327–334, December 1987.