MARTI—A Multiprocessor Architecture for Ray Tracing Images

M-P. Hébert, M. D. J. McNeill, B. Shah, R. L. Grimsdale and P. F. Lister

ABSTRACT Multiprocessor systems are well suited to ray tracing, since each ray can be traced independently. However, the large databases required to model complex scenes create problems of data access. In this paper we propose a multiprocessor architecture for ray tracing which removes the need for duplication of the database at processor level. The database is held on a group processor basis, and resides in shared memory. Many of these groups, or *clusters*, can be replicated to form a highly parallel multiprocessing system. Results of a software simulation of the architecture are promising, indicating that a large number of processors per cluster is possible.

1 Introduction

Ray tracing has emerged as one of the most photorealistic methods of rendering, particularly when combined with a sophisticated lighting model. In the ray tracing model primary rays are cast from the eye through each pixel into the scene, where intersection tests are performed on objects. Once the intersection between a ray and the closest object has been found, further rays are cast to the lights in the scene and in the reflection and refraction directions, where more intersection tests are performed [17]. Thus several secondary rays are cast for each primary ray. Since high quality graphics monitors typically display around one million pixels, it is easy to see that many millions of rays are generated during a rendering, particularly when improvement techniques such as antialiasing are included. Scenes can contain thousands of object primitives, so a brute force technique where every ray is intersected with every object is computationally too expensive. It is well known that most of the computational power is required for the intersection tests [17], and therefore speed-up techniques which remove the need to intersect every ray with every object have been developed [6,7,10]. The rendering time is dramatically reduced, although the algorithm still requires many minutes or hours of conventional processing time.

The solution to the problem of unacceptably long rendering times has been to parallelize the process. Ray tracing is well suited to parallelization, since every ray can be traced independently, although there are many problems introduced by such a solution—handling of the database, inter-processor communication and efficient use of system resources. Algorithms have been proposed based on architectures such as transputers [8], where the database is distributed across the network of processors, and on more dedicated hardware in which the database is duplicated in each processor [13]. In this paper we propose a scheme where the database is held on a group processor basis, minimising inter-processor communications but avoiding the need for a fully distributed system with its associated overheads.

delivered by
EUROGRAPHICS
DIGITAL LIBRARY
www.eg.org
diglib.eg.org

The structuring of data in object space is a favourite speed-up technique, usually employed as a pre-processing step, since this can be performed in a view-independent way. In this paper we present a method by which the data structure can be built as an integral part of the rendering process, which not only encourages a more complete analysis of the algorithm, but raises issues of changing scenes and efficient use of a multiprocessor system.

In sections 2 and 3 data structures exploiting coherency and antialiasing issues are discussed. In section 5 we propose the architecture, and section 6 describes the simulation technique used in its evaluation. Results are presented in section 7.

2 Octree Structure

2.1 Ray Tracing Octrees

Spatial subdivision techniques, such as octrees, are well known for reducing the computational expense of ray tracing algorithms. It is well known that a ray can access the cells of a grid [6] more quickly than those of an adaptive subdivision structure (such as an octree [7] or a Binary Space Partition (BSP) tree [10]). However, in a grid cell, more intersections between rays and objects may be computed as objects may be unevenly distributed among the cells. Moreover, areas with a high density of objects are usually the 'interesting' parts of the scene and thus many rays will intersect these heavily loaded cells. If the grid is finely subdivided in order to diminish the computational cost, then more memory is required than for an adaptive subdivision structure. For instance, an octree usually has a depth greater than 10. A grid of 8^{10} cells requires at least four gigabytes of memory. This is a problem in a multiprocessor architecture, where a shared resource such as memory can be considered to have a lower effective bandwidth than a local, privately owned resource.

The BSP defined by Kaplan [10] is very similar to the octree. Instead of dividing a parallelepiped in eight child nodes, the subdivision is performed in three successive stages. On the one hand, it could be argued that a BSP needs less leaf nodes than an octree for achieving the same efficiency. On the other hand, the number of branch nodes increases in a BSP. Empirical data show that no one structure is best suited for all scenes, and that the number of nodes is similar for all structures in most sample scenes.

Our choice for the octree structure was motivated by the possibility of using the HERO algorithm [1]. Compared with other octree traversal methods, HERO decreases the number of floating point operations and the number of manipulations of pointers addressing nodes. In order to reach the next hit voxel from the current one, a common ancestor technique [14] is traditionally used. HERO avoids all ascents to the common ancestor by using recursion. Moving to a parent node only requires a value to be popped from a stack. Recursion also enables the intersection between rays and the boundary planes of nodes to be computed only once per ray and per plane. The coordinates of the boundary planes can be determined by mid-point subdivision. Precision problems do not occur as the octree depth is much smaller than computer word lengths. Thus, these coordinates do not need to be stored; this diminishes the size of the octree structure dramatically.

2.2 Building Octrees

Most nodes belong to the lower levels of the octree. However, these nodes are rarely accessed during the ray tracing process (see Figure 1). For the Utah teapot database

modelled with 9120 polygons, the four deepest levels of the octree contains 56.5% of the nodes but only 2.2% of accesses concern this nodes. The remaining 97.8% of accesses occur in the upper part of the octree which contains 43.5% of the nodes. This suggests that the lower levels of the octree should not be built. However, if these lowest levels are not built, the cost in terms of floating point operations increases fourteenfold for the teapot image, due to the additional computed intersections. We have therefore investigated a dynamic building of the octree. During a preprocess the top levels of the octree are built. Due to the adaptive nature of the octree structure, lowest level nodes, if required, can be easily added to the tree during the ray tracing process. Only those nodes which are useful are created. The size of the data structure is therefore reduced, yet the number of arithmetic operations does not increase. Dynamic octree building is particularly efficient for very large data bases.

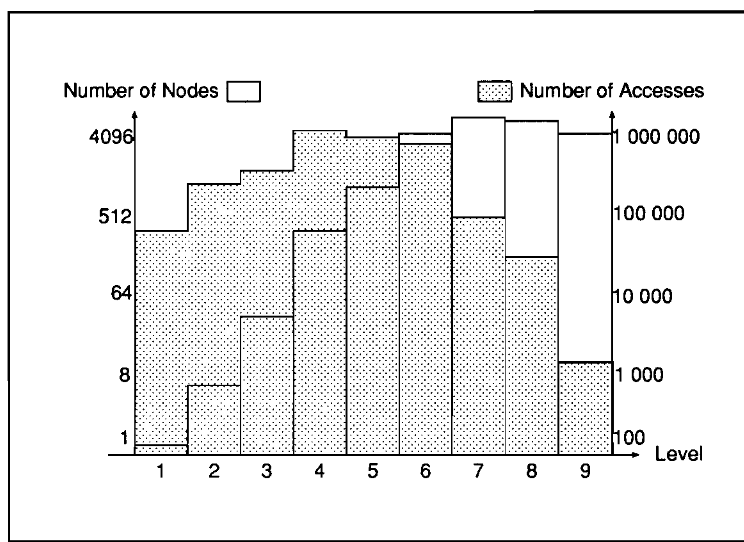

Fig. 1. Number of accesses and number of nodes per level of octree

Typically a data structure such as an octree is built for a particular scene and stored on disc. This file is then accessed by the renderer and pixel values are either stored on disc or displayed on the screen. Since the data structure can be used for many viewing positions the build time is negligible. For many scenes, when rendering is performed by a single processor, the build time for the octree is less then 10% of the total rendering time. However, when multiprocessor systems are considered, this build time can become significant in the rendering process due to several factors. Firstly, when hundreds or thousands of processors are employed to render an image, the percentage of time taken to build the data structure increases. Secondly, once rendering times are reduced to an acceptably small time—and we consider this to be within the bounds of current hardware performance—application developers and users will want to move away from the idea of a static scene and develop instances of moving objects and lights. This may perhaps lead to the data structure being re-built for each frame. It is therefore sensible to use the available processors to build the data structure. Subtrees of an octree can be built in parallel and then ordered. A similar algorithm to static building is used for building and ordering subtrees dynamically.

3 Antialiasing

Being a point sampling method, ray tracing is prone to aliasing. According to the Shannon theorem, aliasing can only be reduced by either filtering out the high frequency components of the image or by increasing the sampling density. The simplest way to antialias an image is to supersample the whole image or to pass it through a low pass filter. However, a lot of computational effort is wasted in regions of low frequency pattern changes and the reduction in aliasing at high frequency pattern changes is not significant. Hence adaptive supersampling methods need to be used. Antialiasing in ray tracing can be performed in image, object or ray space.

3.1 Antialiasing Algorithms

Image Space Antialiasing

Most antialiasing algorithms are based on image space. Whitted [17] introduced the idea of adaptive oversampling, whereby rays are traced through the four corners of the pixel. If the intensity values vary more then some threshold value, the pixel is subdivided. The contribution of each subpixel is weighted by its area, and the final pixel intensity is obtained by the summation of the subpixel values. However, whatever the oversampling rate, regular aliasing defects will appear. To avoid this problem, Mitchell [12] and Dippé and Wold [5] suggest algorithms based on non-uniform sampling, which distorts the aliasing effects, thus making them less conspicuous to the eye. Mitchell uses multistage box filters, whereby different regions of the image are sampled at different densities. The eye perception model is used to evaluate the threshold values. These values form the criteria to determine the sample densities. Dippé and Wold implement an adaptive stochastic sampling algorithm, which uses the Poisson function and jittering to determine the error estimates, error bounds and the sampling rates.

Object Space Antialiasing

Algorithms based on object space antialiasing use information available in the object space to prefilter the image before being sampled. Amanatides [3] introduced a cone tracing algorithm which traces cones into the environment, and antialiases the image by filtering across the cross-section of the cones in object space. However, the cost of tracing cones greatly outweighs the advantages in image quality.

Thomas [16] introduced the method of edge detection. Edges are detected by observing how the ray passes through covers, built around the surfaces. Only rays which pass near a surface edge are filtered using a low pass filter.

Ray Space Antialiasing

Akimoto [2] present a method based on adaptive undersampling. Representative pixels in a region are raytraced and their ray trees stored. Intermediate pixel intensities are obtained by interpolating between these pixels. This method, however, is expensive in terms of memory requirements and for very complex scenes the regions have to be pixel wide to ensure small objects are not missed.

3.2 Edge Detection Algorithm

The algorithm used at Sussex has been developed with multiprocessing systems in mind, and incurs no additional communication overheads. When a ray passes close to an edge, the corresponding pixel is supersampled. Since we use polygonal databases only, proximity

of a ray from an edge is detected as part of the intersection algorithm. This requires only a single additional comparison. If the edge and the intersection point do not lie in the same node, then pointers to the node containing the edge and the following leaf nodes which the ray crosses up to the intersection point are stored. The pixel identifier and the leaf node list are passed to the antialiasing routine, which traces the subpixel rays through the nodes in the list, thereby reducing the overhead of the subpixel rays crossing the octree. Results have shown that on average only 25% of pixels need to be antialiased.

4 Texturing

An important feature of any renderer is the ability to provide textured objects. Although one-, two- and three-dimensional mapping takes place, a widely used technique in ray tracing is to map a two-dimensional texture onto a three dimensional object. Texture can be applied procedurally, such as wood effects, where there is little implication for the architecture, or by color mapping, where the intersection point of the ray and the surface is used to index into a stored color map.

4.1 Distribution of Texture Data

The inclusion of texturing on a large scale—tens or hundreds of color maps requiring many megabytes of storage space—presents the problem of data access. Texture information can be stored with the object/surface, with each object containing only the texture-id and the address of the entry into the color map. When a textured surface is hit by a ray, plane constants can be worked out and stored with the surface for future use. Once the intersection point of the ray and the surface is found, the tracing processor calculates the index into the color map before requesting the required entry.

4.2 Antialiasing in Texturing

Antialiasing textured surfaces is necessary when a surface lies at an oblique angle to the observer, since one pixel may cover perhaps hundreds or thousands of texture pixels, or texels. The problem is how to approximate the color of the area covered by the pixel. Antialiasing therefore needs to be performed not just near an edge, but in every case where a ray intersects a textured surface.

The following algorithm was developed to work in conjunction with the antialiasing algorithm discussed in section 3. When a ray hits a surface and the intersection point is not near the edge of the surface, additional rays can be cast in order to approximate the color by supersampling, followed by filtering. Note that these rays need only to be intersected with the surface and do not need to be traced through space, since the intersection is not near the surface edge. For the case where the ray intersects the surface near an edge then these additional rays need to be traced through the stored voxels as outlined in section 3. Once the parameters are worked out for the particular intersection point on the surface the appropriate index can be calculated and the color requested.

5 Description of the Architecture

5.1 Architecture Overview

Although searching through an octree structure reduces the rendering time dramatically, ray tracing is still a computationally expensive algorithm. Processor power must therefore

be fully exploited. Algorithms using image parallelism compute a pixel intensity fully in one processor. On the one hand, it makes maximum use of the processors by incurring a minimum of overheads. On the other hand, each processor needs to be able to access the whole database. Databases are so large that it is unrealistic to duplicate the scene data for every processor. Resources must then be shared among processors. MARTI—a Multiprocessor Architecture for Ray Tracing Images—is based on observations of the ray tracing algorithms.

MARTI is made up of independent units, called *clusters*, which consist of a number of general purpose, fast floating point, programmable microprocessors tracing rays using image parallelism (see Figure 2). Using microprocessors as opposed to specialized hardware enables the implementation of various types of primitives and lighting models, in addition to the evolution of the algorithm. Processors are grouped in such a way that each cluster is seen as only one unit. An interface node deals with all interfaces between MARTI and external devices. All clusters, the host processor and the display are linked through an interconnection network.

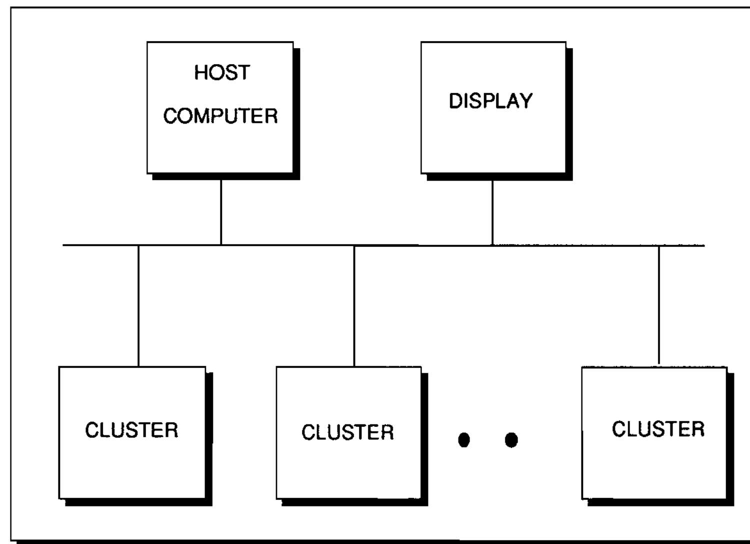

Fig. 2. Architecture overview

5.2 Cluster Architecture

Sharing Memory

The choice of powerful processors with local memories diminishes inter-processor traffic. A shared memory allows a few processors to access the database with very short delays, due to the high locality of reference for ray tracing algorithms, and straightforward memory management. The database is stored in the *cluster memory*, which is shared among all processors of the cluster. A bus links the cluster memory to the processing nodes.

Master/Slave Operation

A master node supervises the cluster. It waits for interrupts, controls job execution and interfaces the cluster to the interconnection network. When a cluster is idle, it can request

a task from the job queue of the host processor. For instance, during the ray tracing process, the location and size of a tile on the screen is fetched, and the intensity of every pixel of the tile is computed by the cluster. By managing a cluster job queue, the master node knows when a cluster has finished its task and can send an 'end of process' message to the interface node. Although no deadlocks occur in our implementation, the master node can prevent such an event from occurring in another software implementation.

The control of the master node avoids disturbing a processing node which does not request any service, whenever an external device or another node interrupts. It also enables the internal management of the cluster to be transparent to the rest of the architecture.

Processing Nodes

Every processing node includes a processor and local memory. Processors are computationally powerful in order to reduce management overheads. As ray tracing requires a lot of floating point operations, they must include a fast floating point arithmetic unit. Programmable processors allow the rendering of any type of scene models and also algorithm development.

Programs are duplicated in every local memory. Data requested by a processor are copied from the cluster memory to its local memory. Due to the high locality of reference of ray tracing algorithms, most data requests will be local; no delays will occur for accessing cluster memory (see section 7 for results).

Figure 3 sums up the cluster architecture. Each cluster has a shared memory, a master node and several processing nodes made up of a processor and local memory.

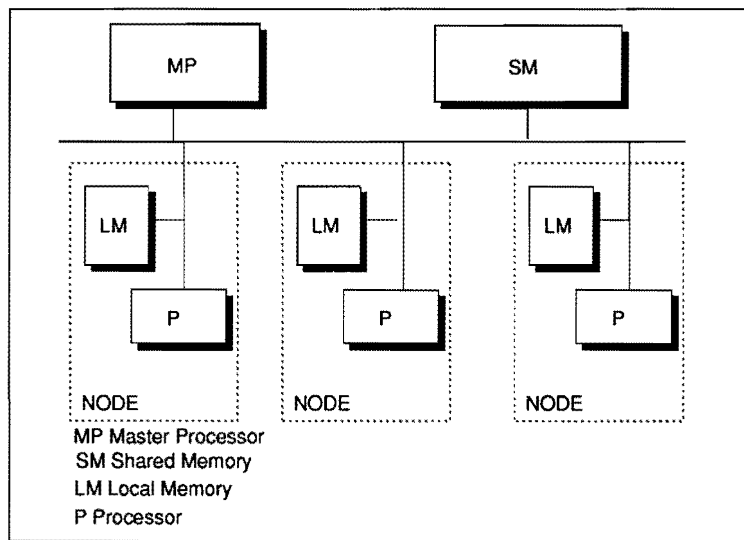

Fig. 3. Cluster Architecture

5.3 Ray Tracing with MARTI

Initialization and Preprocessing

The host computer sends programs and data to MARTI. The clusters can then start to build the octree. The master node distributes disjoint subtrees to the processing nodes requiring nodes. The processing nodes build the subtrees and write them to shared memory.

Ray Tracing

In each cluster, the master nodes requests patches of tiles from the host and makes tiles available to processing nodes. Each processing node requires scene data to compute pixel intensities. If these data are not held in the local memory, then shared memory is accessed and the data are copied locally. Once a processing node has achieved its task, it writes pixel intensities to the shared memory, and fetches a new job from the job queue. The master node requests a new task from the host processor when the job queue is empty.

During the calculation of a pixel intensity, a voxel pointing to a large number of objects may be reached. In that case, dynamic octree building is performed. The ray tracing process then continues normally. There is no data consistency problem as the data are computed and never modified. The worst case is when the same computation is performed several times by different processors.

Fragmentation of objects into several nodes of an octree implies redundant ray-object intersections. A mailbox [4] can be efficiently used since a ray is fully processed in one processor. As a cluster computes only a part of the image, dynamic octree building is an efficient data management technique for a cluster.

Aliasing

The intensity of a pixel is computed fully inside a single processor. Hence there is no communication overhead associated with the antialiasing routine. It is also reasonable to assume that the whole ray path is stored in the local memory in the first pass and therefore there is no need for shared memory accesses for the subpixel rays.

Pixels which need to be antialiased require longer to trace, but since tiles are read by a processor on request, no additional load imbalance problems will arise due to antialiasing.

6 Simulation

6.1 Multiprocessing in a Unix Environment

The BSD socket abstraction provides the means whereby processes can communicate with each other within a communications domain. Sockets are derived from BSD 4.2 and will also be included in future releases of AT&T Unix[1] System V. Apollo DOMAIN/OS, incorporating BSD4.3, SYSV.3 and AEGIS environments allow the use of sockets in both its BSD and System V environments. Sockets are implemented on top of the reliable transmission control protocol (TCP), which provides a robust environment for interprocess communication.

The basic building block for interprocess communication is the socket, an endpoint for communication. Each socket has a type and one or more associated processes. Sockets exist within communication domains—abstractions introduced to bundle common properties of

[1] Unix is a registered trademark of AT&T in the USA and other countries

processes communicating through sockets. Each socket is identified uniquely by a socket address. This is a structure that specifies the socket's *Address Family*, *Network Address* and *Port Number*.

Standard routines are provided by Unix for mapping host names to network addresses, network names to network numbers, protocol names to protocol numbers and service names to service numbers, and the appropriate protocol to use in communicating with the server process. With these support routines, an application program rarely has to deal directly with addresses. Thus, services can be developed in a largely network-independent fashion.

6.2 Motivation

Simulating the various architectural considerations benefits from a distributed multiprocessor tool for several reasons.

- The use of the socket abstraction allows a robust and transparent way of logging of all inter-process communication. Processes can represent individual processor action, groups of processors, buses, or indeed any action on data the programmer desires. Data flow into and out of the process is again under programmer control; data can be input to and output from the process via file, standard input or socket—another process. The ability of a process to poll its connected sockets for incoming data—listen for input on all its connections—allows processes to emulate the action of a bus. A library of routines can be built to implement the inter-process connections in a generic way, so making the use of sockets transparent to the user, and allowing simulations to be built easily and quickly, without the need to re-compile code for different connection topologies.

- The ability to run several processes in parallel. Although it is not necessary to simulate the architecture wholly, executing two or more processes synchronized, in parallel, and logging their data flow overcomes the drawback of simulation using sequential algorithms, namely execution time and the unrepresentative nature of the simulation.

- The processor bound nature of the ray tracing algorithm. The simulation requires isolating the various parts of the code conforming to different processors in the architecture, running sections of the code in parallel and also transferring possibly large amounts of data between processes. To do this sequentially on a single machine would increase the already long time needed for the algorithm by an unacceptable amount. Implementing transparent support for the Internet communications protocols allows the extra power of networked processors to be utilized.

- The ability to run all or part of the simulation on a specialized architecture. Since the use of the Internet communications protocols allows processes to be executed on hosts other than Apollo Domain workstations, the simulation can include all or part of the algorithm executing on a specialized type of architecture, e.g. a shared memory machine such as the Sequent Symmetry.

Since programming with sockets allows generic access to various protocol suites, the extra time involved in the construction of a distributed simulation tool was considered worthwhile. With the portability afforded by the C language [11] and the socket abstraction the simulation has access to a wide range of equipment, as well as indicating how well the ray tracing algorithm performs in a distributed environment.

The use of the socket abstraction in this way provides a representative method of profiling parallel processes. Although the initial coding involved is not insubstantial, it is an easy concept to grasp, and the transparency and portability afforded by such an abstraction are good justifications for proceeding along this path.

A framework, allowing simulation of the proposed architecture was therefore written using C in the Unix environment. Details of how the simulation was implemented are beyond the scope of this paper. The Distributed Ray Tracer is currently in use tracing images at Sussex.

6.3 Simulation of Cluster Level

A commonly used paradigm in constructing distributed applications is the client/server model. Processes can act as either *clients*, *servers*, or both. Client applications are the controlling processes, which make requests for service from server processes.

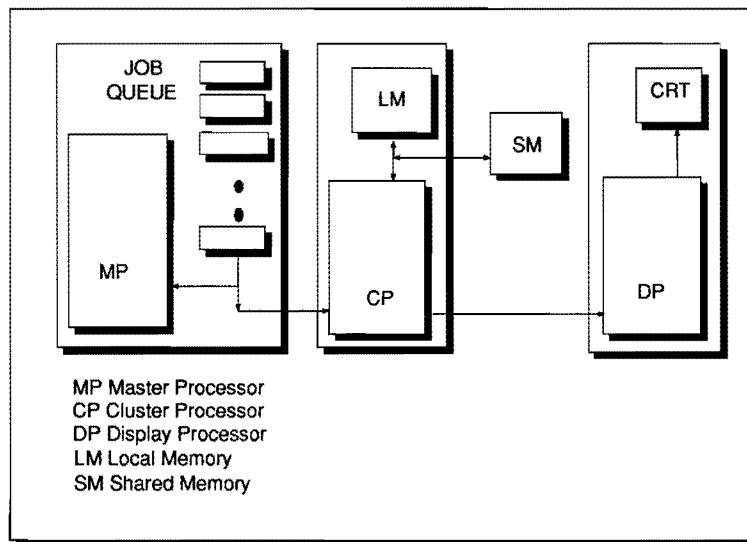

Fig. 4. Functionality of the Cluster Architecture

The client/server model is well suited to many areas of the architecture simulation. Consider the cluster level. The controlling process, the *client*, is the master processor, which after initialization sends tiles to the processing nodes. The processing nodes act as *server* processes, providing the service of tracing rays through pixels. They also act as client processes, however, since they require the service of the local frame buffer to accept rgb data. Since software processes are under programmer control, local memory can be modelled in software, and requests for data on the cluster bus—data not held locally—can be logged. Individual *processes*, then, can represent a master node, processing node, display processor, or cluster bus (see Figure 4).

The simulation is initialized by a managing process, which reads a user-defined data file containing details of the required configuration of the simulation. This managing process starts each process, and listens for signals from the process to indicate their status. If a process should become corrupted, the manager can kill the process and redistribute the task to another process. When a server process is initialized, it uses Unix system calls

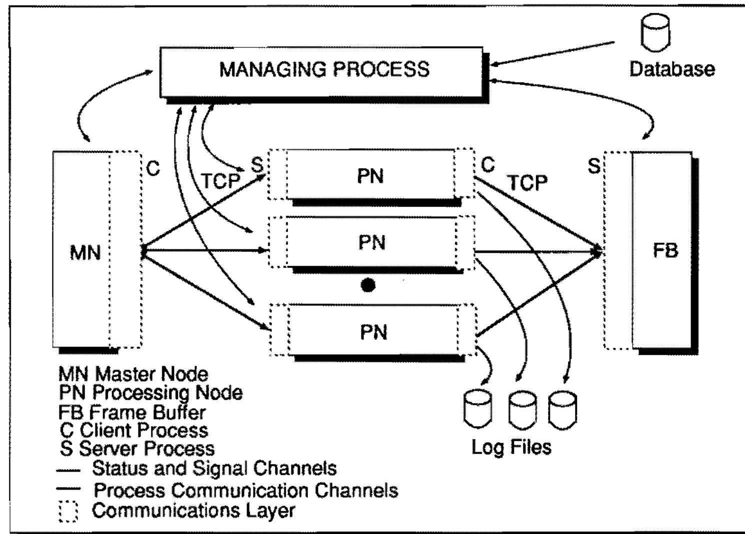

Fig. 5. Simulation Architecture

to establish a local address at which to offer its service. Once a suitable address and port number have been established, the server returns information regarding its whereabouts to the managing process. When a client process is initialized, it establishes a connection to the server (see Figure 5). The address and port number of the server is forwarded to the client at startup by the managing process. It is unrealistic to expect user programs to know proper values for the local address and local port, since a server may reside on multiple networks and the set of allocated port numbers is not directly accessible to the user. By using system calls to choose a valid address and port number, server processes rely on the local system to provide a valid communications endpoint and the programmer needs no detailed knowledge about the (possibly remote) environment.

The simulation has enabled us to investigate at process level the behaviour of the ray tracing paradigm in a multiprocessor architecture. The flexibility of the system has allowed for investigation of different software and hardware models. In particular several areas of interest have been closely modelled for different scenes which have suggested specific hardware and software solutions.

7 Performance

Fast access to shared resources and load balance are typical issues in multiprocessor architecture.

Load Balance

Since pixel parallelism is intrinsically load balanced, a good balance is achieved by processing nodes and clusters. The master node only manages the cluster job queue and sends messages to the host and display node. Its task is negligible compared with that of processing nodes, thus it is able to cope with many processors per cluster. Sending tile instead of pixel intensities to the display node diminishes the management task of the display node, so no bottleneck should occur here.

Accessing Shared Resources

Within a cluster, all processors share the cluster bus and the shared memory. The communication on the cluster bus was determined by simulating a cluster, as explained in section 6.3. Results have been collected for two scenes: one is the gears picture [9] with 1170 polygons, the other is the Utah teapot modelled with 9120 triangles. The tiles are of 5×5 pixels. A cluster computes 2500 tiles. No antialiasing was implemented for these tests; however, as subsamples are computed in the same processing node, antialiasing does not increase the amount of communications. Results for four and nine processors in a cluster are shown in Figures 6 and 7.

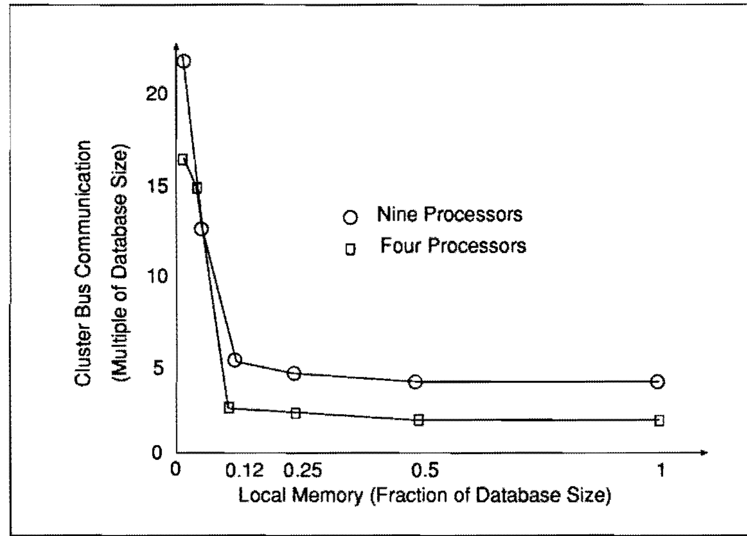

Fig. 6. Cluster bus communication for the gears database

The size of the local memories greatly influences the amount of communication on the bus, when it falls under one eighth of the size of the database. Over this size, the communication on the bus does not increase; thus there is no need for duplication of the database at a processor level. The number of accesses to the shared memory versus size of local memory (see Figure 6) behaves as the number of cache misses versus the size of cache memory [15]. This is an expected result as the local memories copy data as a cache memory does.

Note than in the worst case—the teapot picture with processing nodes having a local memory for data of only 3.125% of the size of the data base (about 87 Kbytes)—the communication on the cluster bus is 4.7% of the communication between a unique processor and memory computing the same tiles. In this simulation, the communications due to messages are about 200 Kbytes. There are no contention problems on the cluster bus even with small local memories. Therefore it is possible to add many more processors; the exact number is technology dependent.

Table 1 shows the ratio of the shared memory access to the local memory access when ray tracing the gears database and the teapot database with nine processors.

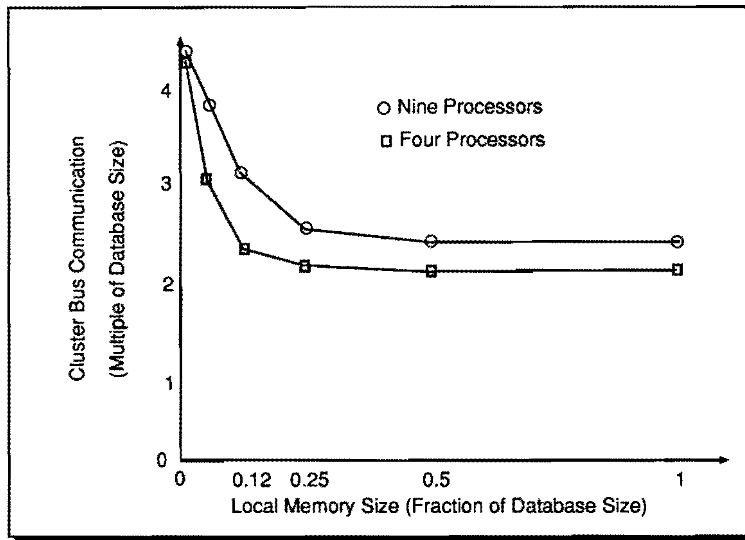

Fig. 7. Cluster bus communication for the teapot database

Table 1. Ratio of Shared Memory Accesses to Local Memory Accesses

Local Memory Size (Database Unit)	Gears Database	Teapot Database
1/4	0.005	0.027
1/8	0.006	0.035
1/16	0.015	0.038
1/32	0.027	0.047

Although only scene data are considered, we can say that the locality of reference is significantly high to allow many processors per cluster. For instance, with local memory size equal to one thirty-second of the size of the database, our results predict that thirty-seven processors per cluster will be acceptable, when only considering object and octree dataflow.

This is due to the fact that MARTI benefits from:

- image coherence by using image parallelism,
- space coherence as the number of accesses to data diminishes dramatically by use of the octree structure and the HERO algorithm. Only data of those objects and octree nodes which lie near the ray paths are accessed. Thus less data are required for every ray and data are not overwritten in the local memory from one pixel computation to the next.

If the database is duplicated in every cluster, there is no communication between clusters. The only communication on the interconnection network connecting clusters,

display node and host is due to the distribution of tiles and of the writing of tile intensity to the the cluster node.

8 Conclusion

The architecture proposed, in particular the allocation of memory to the clusters, offers certain advantages for the ray tracing paradigm. The use of image space parallelism allows for inherent load balancing. The use of space and image coherence maintains low traffic on buses. The development of a general purpose hardware system combined with a integrated software model encourages maximum use of resources. The generality of the hardware is such as not to preclude the use of other algorithms. Results have shown that the locally available database to a group of processing nodes provides effective support for the ray tracing algorithm, while remaining scalable.

Future research will concentrate on increasing the communication bandwidth between clusters, in order to ray trace scenes without duplicating the data in every cluster. By benefiting from the high coherence of ray tracing, an architecture with an interconnection network of low complexity should be effective.

We also propose a trade-off between the efficiency of the space subdivision technique and its size. The dynamic octree building should be particularly efficient for animated ray tracing.

Acknowledgements

The authors wish to thank Andrew D. Nimmo, Steven R. Evans, Martin White and Graham J. Dunnett for their contribution to this work. This project is supported by the U.K. Science and Engineering Research Council.

References

[1] M. Agate, R.L. Grimsdale, and P.F. Lister.: The HERO algorithm for ray-tracing octrees. In *Advances in Computer Graphics Hardware IV*. Springer-Verlag Berlin Heidelberg New York, 1991.

[2] T.K. Akimoto, K. Mase, A. Hashimoto, and Y. Suenaga.: Pixel selected ray tracing. In *Proceedings of the Eurographics 89*, pages 39–50, 1989.

[3] J. Amanatides.: Ray tracing with cones. *Computer Graphics*, 18(3):129–135, July 1984. SIGGRAPH'84 (Minneapolis, Minnesota, July 23-27, 1984).

[4] B. Arnaldi, T. Priol, and K. Bouatouch.: A new space subdivision method for ray tracing CSG modelled scenes. *The Visual Computer*, 3:98–108, 1987.

[5] M.A.Z. Dippé and E.H. Wold.: Antialiasing through stochastic sampling. *Computer Graphics*, 19(3):69–78, July 1985. SIGGRAPH'85 (San Francisco, California, July 22-26, 1985).

[6] A. Fujimoto, T. A. Tanaka, and K. Iwata.: ARTS: Accelerated Ray Tracing System. *IEEE Computer Graphics and Applications*, pages 16–26, April 1986.

[7] A.S. Glassner.: Space subdivision for fast ray tracing. *IEEE Computer Graphics and Applications*, 4(10):15–22, October 1984.

[8] S. Green, D. Paddon, and E. Lewis.: A parallel algorithm and tree-based computer architecture for ray-tracing computer graphics. In PM Dew and TR Heywood RA Earnshaw, editors, *Parallel Processing for Computer Vision and Display*, 1989.

[9] E. Haines.: A proposal for standard graphics environments. *IEEE Computer Graphics*, 7(11):3–5, November 1987.

[10] M.R. Kaplan.: Space tracing, a constant time ray tracer. In *SIGGRAPH'85 tutorial on the uses of spatial coherence in ray tracing*, July 1985, San Francisco, CA.

[11] B. Kernighan and D. Ritchie.: *The C Programming Language*. Prentice-Hall Software Series. Prentice-Hall, 1988.

[12] D.P. Mitchell.: Generating antialiased images at low sampling densities. *Computer Graphics*, 21(4):65–69, July 1987. SIGGRAPH'87 (Anaheim,California, July 27-31, 1987).

[13] T. Naruse and M. Yoshida.: SIGHT - a dedicated computer graphics machine. *Computer Graphics Forum*, 6(4):327–334, 1987.

[14] H.Q. Samet.: *Applications of Spatial Data Structures: Computer Graphics, Image Processing and GIS*. Computer Series. Addison Wesley, 1989.

[15] H.S. Stone.: *High-Performance Computer Architecture*. Electrical and Computer Engineering. Addison-Wesley Publishing Company, second edition, 1990.

[16] D. Thomas, A. Netravali, and D. Fox.: Anti-aliased ray tracing with covers. *Computer Graphics Forum*, 8:325–336, 1989.

[17] T. Whitted.: An improved illumination model for shaded display. *Communications of the ACM*, 23(6):343–349, June 1980.