nota interne B4-46

IST. EL. INF.
BIBLIOTECA
Posiz. ARCHIVIO
B4-46

CNUCE

PARALLEL RENDERING OF VOLUMETRIC DATASET ON DISTRIBUTED MEMORY ARCHITECTURES

Internal Report C91/25

Dec.19, 1991

C. Montani, R. Perego,
R. Scopigno, M. Tartaglia

Parallel Rendering of Volumetric Dataset on Distributed Memory Architectures *

C. Montani *, *R. Perego, R. Scopigno, M. Tartaglia*

Istituto CNUCE - Consiglio Nazionale delle Ricerche

Via S. Maria, 36 – 56100 Pisa, Italy

* Istituto Elaborazione dell'Informazione - Consiglio Nazionale delle Ricerche

Via S. Maria, 40 – 56100 Pisa, Italy

Abstract

A parallel solution to the visualization and rendering of volume data is presented. Based on the ray tracing (RT) visualization technique, the system works on a distributed memory parallel architecture. The volumetric dataset is internally represented by the STICKS coding scheme, a data structure which exploits the one-dimensional coherence of the data, and is distributed between the nodes using a slice-partitioning technique. The similarity between the chosen data partition, the communication pattern of the visualization processes and the topology of the physical system architecture is one of the keypoints of the proposal and improves both software design and efficiency. The partitioning technique used and the network interconnection topology reduce the communication overhead and allow an efficient implementation of a static load balancing technique based on the pre-rendering of a low resolution image. Details related to the practical issues involved in the parallelization of volumetric RT are reported.

*This work was partially funded by the Progetto Finalizzato "Sistemi Informatici e Calcolo Parallelo", Sub-projects "Calcolo Scientifico per Grandi Sistemi" and "Processori Dedicati per la Sintesi di Immagini", of the Consiglio Nazionale delle Ricerche.

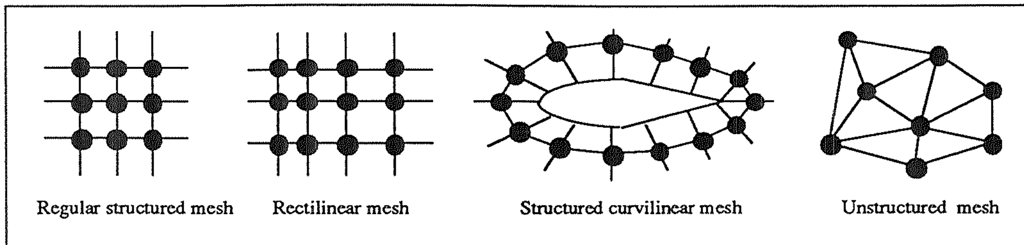

Figure 1: Volumetric dataset taxonomy.

1 Introduction

Applications requiring data visualization exist in many disparate fields of research, and a large subset of these applications makes frequent use of sampled scalar/vector fields of three spatial dimensions, also known as *volume data*.
The growth of 3D scanner devices (CT, PET, MRI, Ultrasounds) which input the properties of real 3D objects directly into machine readable format and the increasingly urgent need to understand natural phenomena by analysing large empirical data sets obtained by measures or simulations (several applications can be found, for example, in geophysics, astrophysics, meteorology, chemistry) has made Volume Visualization a topical subject [5]. Volume Visualization provides the user with representation data structures and tools whichcan render a 3D grid of data in a more comprehensive way than presenting them in tabular formats or as a sequence of 2D images.
Basic issues in Volume Visualization system design are: (a) choosing the most suitable representation scheme; (b) choosing the most suitable visualization approach; (c) using techniques/architectures to speed up the visualization process.

Taxonomy and representation schemes

The wide range of different volumetric dataset can be classified following the taxonomy given in Fig. 1; the graphical sketch gives, for the sake of simplicity, a 2D representation of the geometrical shape of different classes of volumetric data. In this paper the authors will always refer to dataset defined on *regular structured mesh*, in brief *voxel dataset*.

Voxel dataset can be simply represented by set of 2D matrices or, directly with 3D arrays. The use of data structures allowing a compression of the spatial occupancy of the data can be extremely useful, because the space required for high-resolution voxel dataset representation is huge when a direct array storing is applied. Using a space-saving representation such as the

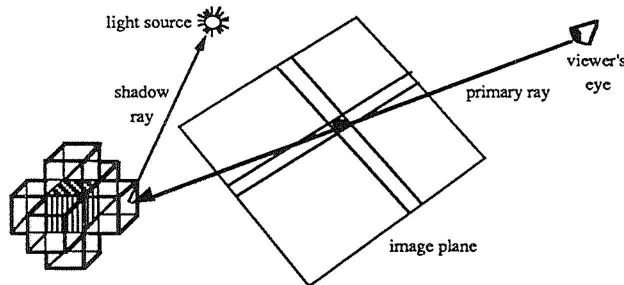

Figure 2: Ray tracing.

Octree [11] or the STICKS [12] may lead to significative savings in space. Using such a data compression scheme can even make the visualization process less complex.

Volume rendering

Two antithetic approaches are possible to render volumetric dataset. The first is based on the computation of a surface-model approximation of some of the iso-value surfaces contained in the voxel dataset [10] and the use of the standard surface rendering techniques [4] to visualize these surfaces. The second approach, known as *direct volume rendering*, directly visualizes the dataset without the need of an explicit voxel_to_boundary conversion. The latter approach is more appropriate with high-resolution voxel dataset, both in terms of results and efficiency [20]. Direct volume rendering techniques can be futher subdivided into two other classes: *projective* algorithms [14] [21] [19] and *ray tracing* (RT) algorithms [2] [9] [17].

The parallelization of a *volumetric direct RT* algorithm is presented in this paper; the original sequential version of the same RT algorithm was described in [12].
RT is a consolidated visualization technique [6], which is also frequently used to render volumetric data [2] [9] [17]. Ray tracing determines the visibility and the shaded color of the surfaces in the scene by tracing imaginary rays from the viewer's eye to the objects in the scene. Once the viewer's eye position (*vep*) and the image plane window have been defined, then a ray is fired from the *vep* and through p, for each pixel p in the image window. The closest ray-object intersection indentifies the visible surface, and a recursive ray generation and tracing process is applied to handle shadows, reflection and refraction (Figure 2).
The computational complexity of volumetric RT is far lower than in classical RT of surface models, because much less "realism" is required in the visualization of volume dataset; for example, specular reflection effects and shadow computation do not generally need to be correctly

simulated. On the other hand, transparency effects are important for the analysis of inner constituents, and so the associated secondary rays usually have to be traced.
The number of rays generated is therefore lower than that required by surface-based ray tracers, and can be extimated linear to the number of primary rays.
Moreover, tracing a ray is generally simpler in volumetric RT than in surface RT. The first requires the ray rasterization and the access to the $\mathcal{O}(n)$ pierced voxels only, with n^3 the resolution of the voxel space; the latter computes $\mathcal{O}(p)$ ray-object intersections, with p the number of elementary primitives (e.g. polygons) in the scene and, usually, $p >> n$.

Towards interactive visualization

Visualizing volume dataset requires expensive computation, due to the complexity of the algorithm and the dataset. The practical every day use of visualization as an analysis tool requires interactive dialog between the user and the visualization system; therefore, image synthesis time is a critical issue.

A number of specialized and/or parallel architectures designed for volumetric rendering have been proposed; some of them are described in the survey by Kaufman et al. [8].

- **Special-purpose architectures**: among these is the *CUBE* architecture, which is characterized by an original organization of the memory and which uses a multiprocessing engine; CUBE allows parallel access and processing of beam of voxels with a raycasting-based rendering approach. The *INSIGHT* system represents voxel dataset by using a data compression scheme (the octree scheme) and uses a specialized processor to apply a Back_to_Front traversal of the octree nodes. Both these architectures and others, such as the *PARCUM* system, the *VOXEL Processor* and the $3DP^4$ architecture have not yet been developed in full scale, only low resolution prototypes or software simulation are currently available.

- **Implementations on general-purpose multiprocessors**: the *CARVUPP* system [22] generates shaded images of a single iso-valued surface from a volumetric dataset by using a Front_to_Back projective approach; the system was developed with an INMOS Transputer network and is part of a medical imaging workstation. Both ray tracing and projection approaches were parallelized by Challinger [3] on a shared memory multiprocessor (a BBN TC2000). A projective approach for rendering of orthogonal views was implemented by Schroeder and Salem [15] on a data-parallel computer (a Thinking Machine CM-2).

A parallel implementation of a ray tracing algorithm is proposed in this paper. The algorithm works on STICKS-represented voxel dataset, and runs on a low cost Transputer-based multicomputer.

The paper is organized as follows:

Section 2: Presentation and assesment of the parallelization and data partitioning strategies.

Section 3: Detailed description of the proposed solutions to parallel ray tracing of voxel dataset.

Section 4: Results and concluding remarks.

2 Parallel Ray Tracing of Volumetric Dataset

Numerous parallel systems have been proposed to speed up surface-based RT [6]. Much less studied is the particular application of parallel RT to voxel visualization. In our project we concentrated on the parallel design of a RT algorithm to visualize classified voxel dataset on distributed-memory MIMD architectures. These architectures allow efficient parallel implementations of RT and are commercially available at a good price/performance ratio.

2.1 Parallelizing strategies

According to Badouel, Bouatouch and Priol [1], the *parallelizing strategies* for RT can be classified, by focusing on the kind of data transmitted on the processor interconnection network, as follows:

- **parallelism without dataflow**: the whole scene data is replicated on each processor and the RT code distributed on the processors is a standard sequential RT implementation; a partition of the image space identifies the subset of the image that each processor will synthesize independently; implementations are straightforward and scalable, but the memory requirements are often prohibitive.
- **parallelism with ray dataflow**: the scene data is partitioned and distributed to the processors. Each processor searches for intersections by testing each ray with the objects of the scene contained in its local partition. Each non-resolved ray is transmitted to "adjacent" processors for further ray-object intersection computations.
- **parallelism with object dataflow**: a mixture of both the previous strategies, in which a partition of the whole image is assigned to each processor. The processor computes locally

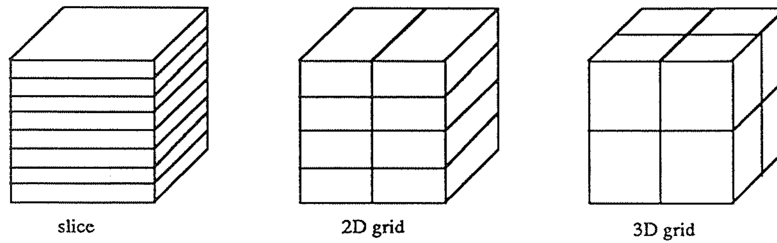

Figure 3: Partitioning strategies for a voxel dataset.

all the ray-object intersections and light computations required to solve each assigned ray but, analogously to the second strategy, the scene data is partitioned among the processors. An emulation of shared memory, based on the transmission at the request of object descriptions, is therefore required.

Our choice of a parallelism *with ray dataflow* is based on the following considerations:

- the huge amount of memory necessary to represent high-resolution voxel dataset, even if the dataset has been previously classified and compressed, makes partitioning and distributing the data a must. This is due to the limitation on local RAM space and the lack of virtual memory management common to most multicomputers.
- voxel dataset division into regular and distinct partitions is simple and does not involve all the problems common to boundary or CSG scene partition (e.g. elementary primitive spanning into more than one partition).
- the quest for visual realism is not as pressing as in high-quality rendering. Users of volume rendering applications are generally more concerned with insight into the represented phenomena than with visual realism. This means fewer number of rays generated in the image synthesis process and fewer ray transmissions on the inter-processor communication network. This leads to lower communication overheads and simpler load balancing.

2.2 Data partitioning strategies

Partitioning a voxel dataset is extremely simple, as a result of the homogeneous and regular spatial distribution of the data. Some of the different partitioning strategies are sketched in Figure 3: monodimensional partitions (*slice partition*) with the dataset divided into slices by using a set of cutting planes which are othogonal to one of the axes; bidimensional partitions,

using a 2D grid of othogonal cutting planes; 3D partitions, generated using a 3D grid of othogonal cutting planes or a recursive subdivision in octants. All of these strategies can be used to produce regular or irregular partitions of the voxel dataset.
During the RT process, an inter-processor communication will be required for each ray that exits out of a partition allocated on one node and enters in an adjacent one. Each partition criterion implies different communication patterns between the RT processes which manage each voxel dataset subset. A single bidirectional channel connects each couple of processes which manage adjacent partitions, thus leading to a logical ring interconnection scheme for the slice partition criterion and 2D or 3D mesh interconnections for 2D or 3D grid partition criteria, respectively. All the above criteria provide a simple and regular communication pattern.
The topology of the underlying architecture generally influences the partition criteria. Due to the high amount of messages interspersed with computation, an effective design of a parallel *ray-dataflow* RT algorithm must preserve communication locality by mapping the processes holding adjacent partitions on neighbour nodes in the hardware topology. A 3D grid partition strategy, for instance, is certainly not optimal if implemented on architectures with a lower dimensional topology, as in Transputer-based systems.

Adopting a *slice partition* strategy was also justified in terms of ease of implementation, management and load balancing, as the following points show:

- partitioning the dataset into slices and transmitting them to the processing nodes is straightforward;
- the simple partition criterion also allows an extremely simple management of dataset partition modifications due to load balancing needs. The nodes adjacency relation does not change after shifting cutting planes;
- only two links per processor are required to build a physical ring. The other links provided on each node (e.g. two more links on Transputers) can therefore be used to manage communications with supervisor nodes, such as host or display nodes.

The scalability of RT implementation based on the slice partition criterion is lower than those based on 2D or 3D grids. The relative simplicity of tracing voxel dataset makes the trade-off between computation and interprocessor communication extremely heavy as the mean length of the ray sections traced by each node is reduced to a few tenths of voxels. The effective scalability of the proposed system is unlikely to be reached by simply increasing the number of partitions, with the resolution of the data unchanged. On the other hand, when the dataset

resolution increases, a proportional increase in the exploited parallelism can be reached by a finer partitioning of the dataset space. The key is the identifying of the optimal partition width (measured in voxel unit).

In Section 3 an alternative approach to increase the scalability of the slice partitioning RT is introduced. This is based on the use of a number of clusters, each of which works on a ray dataflow approach, cooperating to the synthesis of a single image under an image partition criterion.

2.3 The STICKS representation scheme

The data structure used to represent the voxel dataset is the *STICKS* scheme [12]. This data-compression scheme is based on searching for one-dimensional coherence: all the adjacent voxels with an identical scalar value and which belong to the same *track* (i.e. with the same x and y coordinates) are represented by a single record (the *stick*). All of the sticks which belong to a track are connected into a linked list; a pointer to each non-void track is stored in the *Sticks Holder Array*, a n^2 array of pointer cells with n the resolution of the voxel dataset. The Sticks Holder Array is implemented by using a collection of smaller pointer arrays (called *blocks*) instead of a single large n^2 array. Each of these arrays contains 32x32 track pointers. The STICKS scheme is therefore partitioned by selecting each subset of blocks that completely covers the space partition associated with each node.

Good compression on classified voxel data set and its simple management are the main characteristics of the STICKS representation scheme.

3 A Transputer-based Volumetric Visualization System

As mentioned above, the RT algorithm was parallelized by using a ray dataflow approach, supported by a slice partition of the STICKS scheme representing the voxel data set. A simple and low cost Transputer network [18], the *TranStick system*, was designed with this approach.

3.1 The hardware architecture

The architecture of the TranStick system is shown in Fig. 4. The system does not require custom components and its physical topology allows an optimal mapping of the set of commu-

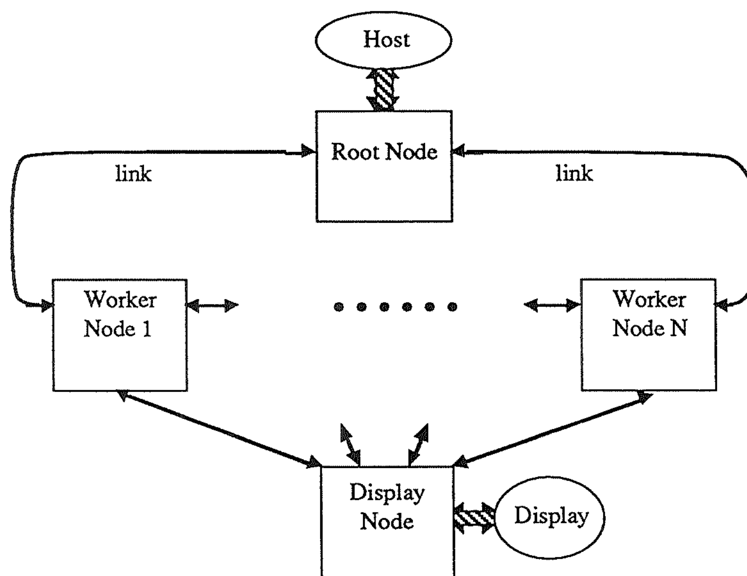

Figure 4: TranStick Architecture.

nicating processes which make up the ray tracing algorithm.

TranStick was implemented on a parallel architecture consisting of a bidirectional ring of Transputers. The current implementation consist of a Root node which interfaces the system with a personal computer (Host) supporting the file system and the user interfaces, a number of Worker nodes running the RT algorithm on their data partition, and a Display node connected to the Worker nodes. Each node is an INMOS T800 Transputer with 2 MBytes of memory. On the Display node there are also 2 MBytes of Video RAM and an INMOS G300 color Video Controller supporting a high resolution graphic display.

The fact that there are only four communication links on the Display node does not limit the system scalability. Having more than four worker nodes on the ring merely entails routing the messages from the unconnected Worker nodes towards the Display node.

3.2 The software architecture

The RT parallel algorithm is designed following a *geometric parallelism* model [7].

The process running on the **Root node** only has I/O functionalities. It manages four main activities: (a) input file scanning, (b) transmission of the STICKS-represented dataset to Workers, (c) start_tracing communications, (d) collecting results (images) and timings.

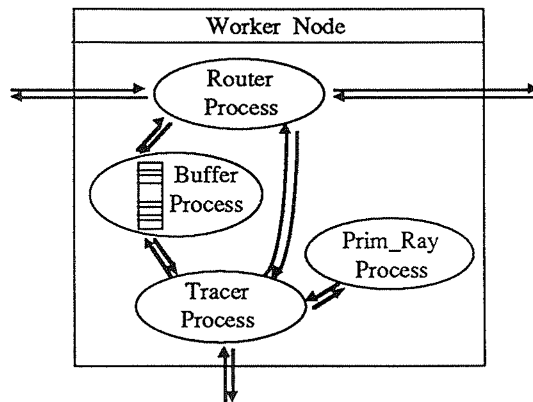

Figure 5: Processes and communication channel on the Worker Node.

(a) *Input file scanning:* the Root process reads in the input file, which contains the selection of one or more datasets plus the related visualization specifications. The user specifies, for each selected dataset, one or more set of visualization parameters (i.e. projection and view settings, threshold surface coefficients, etc.). It also specifies whether each computed image has to be saved on the file system.

(b) *STICKS loading:* the Root node transmits the selected STICKS-represented dataset to the Worker nodes. The limits of the partitions are computed and communicated to each Worker. Then, the Root starts transmitting blocks of the STICKS scheme to the Worker nodes on the communication ring. The receiving process (*Router*) on each Worker node checks if each received block is contained in the assigned space partition. If this holds, the block is loaded into the local memory of the Worker node. The same block is then transmitted to the next node in the ring, because a block could be partially contained in more than one partition. The loading time for a 256^3 voxel dataset is a few seconds (15 .. 30 sec).

(c) *Start_tracing:* for each image to be traced the Root transmit to the Workers the visualization parameters specified by the user within a *start_tracing* message.

(d) *Images and timings collection:* the Workers trace the image, send the evaluated pixels to the Display node and, if required by the user, to the Root to save the computed image on the file system. At the end of the image synthesis process, each Worker sends monitoring and statistical data back to the Root.

The same code runs on each of the **Worker** nodes and, after dataset loading, a slice of the volume data is stored on each local memory; Worker processes and communication channels are sketched in the schema of Fig. 5.
Each Worker starts to trace the primary rays whose first intersection with the bounding volume of the dataset is contained in the Worker's partition. Ray tracing consists of a simple incremental scan conversion of the ray, in order to identify all of the voxels crossed by the ray. For each traversed voxel, the algorithm searches for a stick covering that voxel in the STICKS scheme. If the algorithm finds a stick pierced by the ray, shading computation is applied. Depending on the transparency coefficient associated with the pierced stick, ray tracing is either stopped, or secondary rays are generated and other intersections are searched for and shaded.
On the other hand, if the ray exits from the space partition assigned to the Worker without hitting any stick, two different cases can occur. If the ray enters into the partition assigned to an adjacent Worker, the current worker stops tracing and transmits the ray description to the adjacent one. The adjacent Worker will continue tracing the ray in its partition. Otherwise, if the ray exits from the bounding volume of the dataset, the background color is assigned to the pixel.

The **Router process** manages the communications between adjacent processors. During ray tracing, it receives ray descriptors from adjacent nodes and sends them to the *Buffer* process. Moreover, the *Router* receives ray descriptors from the local *Tracer* process and sends them to the adjacent node chosen by the local *Tracer*.

The **Buffer process** manages a queue of ray descriptors, by sending ray descriptors to the *Tracer* process on request.

The **Prim_Ray process** manages the generation of primary rays on each Worker node. It intersects each ray with the bounding volume (*bvol*) of the dataset. The ray is then transmitted to the *Tracer* process, but only if the first ray-bvol intersection is located on the frontier of the partition allocated on the node. Otherwise this primary ray is rejected.
The drawback of this *distributed* primary ray generation policy is that *all* the primary rays are generated and checked on *all* Worker nodes. An alternative policy is a *centralized* one, where primary rays are genereted and tested for intersection on the Root node. Each ray is then transmitted to the Worker node whose space partition is traversed first by the ray. This policy leaves the workers active on effective tracing only. On the other hand, there is a significant reduction in communication overheads obtained using the *distributed* policy. Moreover, the distributed policy makes it possible to generate primary rays on each Worker only when the

local *Tracer* process is unloaded (i.e. when no ray descriptors are pending on the *Buffer* process); this significantly effects both load balancing and deadlock prevention. Following these considerations, a *distributed* policy was adopted for the TranStick system.

The **Tracer process** receives ray descriptors r_i either from the *Buffer* process or from the *Prim_Ray* process. Higher priority is given to the messages from the *Buffer* process, in order to reduce the number of requests pending from other worker nodes.
The *Tracer* process on a Worker W_i starts tracing the received primary ray r_1 in the voxel space partition allocated to W_i. As soon as the ray intersects a non-void voxel (i.e. a voxel associated with a threshold surface) the first intersection p_1 is indentified. If the voxel in p_1 is not completely opaque, a secondary ray r_2 is generated and traced by the *Tracer* process. If it is possible to completely process r_2 in W_i space, the color contribute of r_2 is composed with the contribution of r_1 and the resulting pixel value is transmitted to the Display node. Otherwise, if r_2 exits from the space partition of W_i and enters those of an adjacent W_j, the ray r_2 has to be transmitted to the adjacent node W_j for further tracing. At this point, the *Tracer* process in W_i has to wait for a ray descriptor, relative to r_2 and containing the computed color contribution due to r_2, to return from W_j. In order to prevent active waiting, the *Tracer* process in W_i saves the current state of the computation on the descriptor of the primary ray r_1 (i.e. the computed intersection p_1, the generated ray r_2, the approximated surface normal in p_1, etc.) and starts tracing the next ray received from the *Buffer* or from the *Prim_Ray* processes. The *Tracer* process will only terminate the computation of the pixel color associated with the primary ray r_1 when it receives a ray descriptor containing the resolved r_2 and its color contribution from the *Buffer* process.
Shadow rays are managed analogously to transparency rays: if they cannot be completely traced in the current partition they are transmitted to the adjacent node and the *Tracer* process save the current tracing status in order to restore it as soon as the resolved ray descriptor will be received.
The *Tracer* process can therefore manage a list of partially processed primary rays descriptors, stored in the Worker's local memory. Each of these contains: the primary ray direction and starting point and, if computed or generated, the resolved intersection p_1, the secondary rays and their respective intersections, etc. All this status information will result in a list of descriptors, one for each non-resolved primary ray.

Once all rays associated with a primary ray have been traced, the pixel color can be computed on the basis of the threshold surfaces intersected, their optical characteristics and the

normal vectors approximated in the intersection points p_i.
The *normal vector* is approximated, by using an *object-space gradient* method [16] with the gradient computed from the value of a set of voxels at a distance of 1 or 2 steps from p_i. If p_i is on the boundary of the partition, then some of the required neighbouring voxels could be on the adjacent node's local memory. In order to prevent the management of these requests for external voxel values, and the associated overheads, we allocate a dataset slice larger than the associated partition (i.e. g voxel planes larger on both sides, with g the maximum width of the 3D interval needed for gradient computation) on each Worker local memory.

3.3 Termination algorithm

The particular topology of the TranStick architecture makes the termination algorithm extremely simple. The **Display** processor is requested to give termination control. It increments a counter for each pixel received and sends a *termination message* to the connected workers upon receiving m^2 pixels, with m the resolution of the image. On receiving the termination message, each worker propagates the termination message to the root processor with the local processing times (for statistics and evaluation).

3.4 Load balancing

Dynamic load balancing methods generally involve a high degree of message interchanging in order to synchronize and move data between the nodes. In some cases, the strategies are complex, thus requiring partition modification and the dynamic reallocation of the data. Furthermore, it is difficoult to manage the frequency of load redistribution and the behaviour of the system under multiple partition redistribution.

The technique we implemented is a static technique similar to that proposed by Priol et alt. [13], based on a data redistribution depending on the chosen view point and, as a consequence, on the workload of each processor. As described in the preceeding sections the initial data partitioning is uniform. The Root node individuates a subset of the primary rays, regularly distributed on the image plane (for example a regular grid of 16x16 rays), and sends these test rays to the appropriate processors. On the basis of the total time t_i spent by each processor Wi to trace these test rays, the Root node defines a new subdivision of the dataset, with size[i] of each new partition computed as:

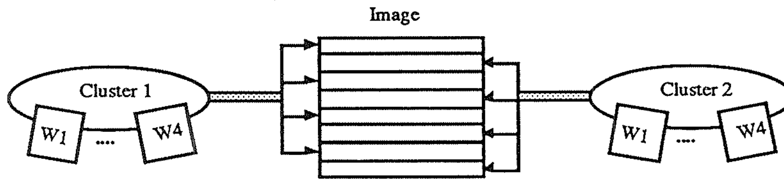

Figure 6: A more scalable parallel system based on a hybrid approach.

$size[i] = size[i] + (t_{mean} - t_i)/t_{single_plane}$

where t_{mean} is the mean processing time on the Worker nodes and $t_{single_plane} = \sum(t_i)/n$, with n the resolution of the dataset on the Y axes.
In order to achieve the new distribution, each worker node transfers data to (and/or acquires from) its neighbour nodes; this data transmission process is performed in parallel by the Workers. The technique is effective, because the redistribution time is much lower than the image synthesis time; the good speedups obtained are reported in Table 1.

3.5 Scalability

The efficiency of some parallel surface-based RT decreases rapidly when the number of processors increases. This is generally due to the increase in communication overhead and to the number of replicated ray-object intersections. The latter is related to the same objects lying in more than one partition, which becomes more probable as the number of partitions increases [13]. The repeated intersection of the ray with the duplicated objects causes some wasted computation cycles.
The latter cause of overhead does not apply to parallel volume rendering. Each elementary object, the voxel, is only contained in one partition and therefore no repeated ray-voxel intersections are evaluated even if the data are more finely partitioned.
But, due to communication overhead, the scalability of a *slice-based* RT of voxel data is low (as shown in Section 2.2). However, ray tracers based on different partitioning strategies (e.g. 2D or 3D grids) prevent the exploitation of high parallelism (e.g. 10^2 or more processors).
In our opinion, high parallel solution of voxel RT can be developed by applying a *ray dataflow* approach within a *without dataflow* approach. The solution can be designed as a set of cooperating clusters, each consisting of a number of nodes. The image space can be partitioned by assigning a subset of pixels rows to each *cluster*. Each *cluster* is designed as a set of cooperating nodes, working under a *ray dataflow* policy. A solution analogous to that presented in this

Head dataset	$W_1elap.$	$W_1act.$	$W_2elap.$	$W_2act.$	$W_3elap.$	$W_3act.$	$W_4elap.$	$W_4act.$
$2W, FV$	00:15	00:11	00:15	00:12				
$4W, FV$	00:06	00:04	00:07	00:05	00:09	00:06	00:08	00:06
$2W, 45^0V, NoBal$	01:46	00:38	01:46	01:36				
$2W, 45^0V, Bal$	01:21	01:08	01:20	01:11				
$4W, 45^0V, NoBal$	01:20	00:25	01:20	00:19	01:20	00:31	01:20	01:11
$4W, 45^0V, Bal$	00:54	00:34	00:54	00:41	00:52	00:39	00:51	00:39

Table 1: Processing times, in mm:ss, required to visualize the *Head* dataset (*2W*: 2 Worker nodes, *4W*: 4 Worker nodes; *FV*: frontal view, 45^0V: lateral view; *NoBal*: with no static balancing, *Bal*: with static balancing); the reported times are the *elapsed* times ($W_ielap.$) and the *active* times ($W_iact.$), where the latter equals the elapsed time minus the overhead time spent waiting for ray communications.

paper can be used to RT on each cluster the assigned image rows (Figure 6).
We are now designing a similar hybrid approach on a nCUBE 6400, a highly parallel architecture based on the hypercube network.

4 Results and Conclusions

A proposal for a cost-effective distributed-memory parallel system to render volumetric datasets has been presented. The methodologies and parallelization strategies followed in the design of the system have been pointed out and justified. By using a ray tracing approach, the system renders volumetric dataset coded by the STICKS representation scheme. It adopts a *ray dataflow* approach based on a *slice partitioning* of the dataset. The detailed description of both hardware and software architecture of the system has been given and their similarity has been highlighted. This similarity allows to minimize the communication overhead and to implement simple and efficient policies for termination detection and static load balancing.

Some results showing the effective parallelism exploitation of the TranStick system are reported in Table 1, which contains the timings relative to the synthesis of images with resolution 400x400 from a 256x256x109 magnetic resonance scan of a human head.
The images come from different views, the first with view direction orthogonal to the XY plane

SOD dataset	$2Workers$	$4Workers$	$eff.$	$8Workers$	$eff.$
$10^0V, Bal$	07:28	03:52	0.96	02:16	0.82
$45^0V, Bal$	07:30	03:54	0.96	02:18	0.81

Table 2: Elapsed times, in mm:ss, required to visualize the *SOD* dataset with a different number of Workers (10^0V: lateral view, rotation of 10^0 on the Y axis; 45^0V: lateral view, rotation of 45^0 on the Y axis; *Bal*: with static balancing); *eff.* is the relative efficiency factor ($eff. = \frac{T_{nW}}{(n/2)*T_{2W}}$).

(frontal view, *FV* in the tables) and the second with direction forming a 45 degree angle with the Z and X axes and othogonal to the Y axis.
Table 2 reports the timings of the same algorithm on a 388x388x464 dataset, obtained by the interpolation of a 97x97x116 dataset which represent the electron density map of an enzyme (SOD, Super Oxide dismutase).
On both table single Worker runs are not reported, because neither dataset completely fits into a single node local memory.

Acknowledgements

The magnetic resonance scan dataset of Table 1 was taken on the Siemens Magnetom and which was provided courtesy of Siemens Medical Systems Inc., Iselin, NY. The electron density map used in Table 2 was provided by Duncan Mc Ree, Scripps Clinic, La Jolla (CA). We also thanks the University of North Carolina at Chapel Hill for having collected and diffused the above and other datasets.

References

[1] D. Badouel, K. Bouatouch, and T. Priol. Ray tracing on distributed memory parallel computers: strategies for distributing computations and data. In *Parallel Algorithms and Architectures for 3D Image Generation - ACM SIGGRAPH '90 Course Note no.28*, pages 185–198, July 1990.

[2] J.A. Challinger. *Object Oriented Rendering of Volumetric and Geometric Primitives.* PhD thesis, University of California, Santa Cruz, CA, 1990.

[3] J.A. Challinger. Parallel volume rendering on a shared-memory multiprocessor. Technical

Report CRL 91-23, University of California, Santa Cruz, CA, July 1991.

[4] J. Foley, A. van Dam, S. Feiner, and J. Hugues. *Computer Graphics: Principles and Practice - Second Edition.* Addison Wesley, 1990.

[5] K.A. Frenkel. Volume rendering. *Comm. ACM*, 32(4):426–435, April 1989.

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

[7] A.J.G. Hey. Experiment in MIMD Parallelism. In *Proc. of Int. Conf. PARLE '89*, pages 28–42, Eindhoven, The Netherlands, June 1989. LNCS 366 Spinger-Verlag.

[8] A. Kaufman, R. Bakalash, D. Cohen, and R. Yagel. A survey of architectures for volume rendering. *IEEE Engineering in Medicine and Biology*, pages 18–23, Dec. 1990.

[9] M. Levoy. A hybrid raytracer for rendering polygon and volume data. *IEEE C. G.& A.*, 10(2):33–40, March 1990.

[10] W. Lorensen and H. Cline. Marching cubes: a high resolution 3D surface construction algorithm. *ACM Computer Graphics*, 21(4):163–170, 1987.

[11] D. Meagher. Geometric modeling using octree encoding. *Computer Graphics and Image Processing*, 19(2):129–147, 1982.

[12] C. Montani and R. Scopigno. Rendering volumetric data using the sticks representation scheme. *ACM Computer Graphics*, 24(5):87–94, November 1990.

[13] T. Priol and K. Bouatouch. Static load balancing for a parallel ray tracing on a mimd hypercube. *Visual Computer*, 5(1/2):109–119, March 1989.

[14] R. A. Reynolds, D. Gordon, and L.S. Chen. A dynamic screen technique for shaded graphics display of slice-represented objects. *Computer Graphics and Image Processing*, (38):275–298, 1987.

[15] P. Schroder and J. Salem. Fast rotation of volume data on data parallel architectures. In *State of the Art in Volume Visualization - ACM SIGGRAPH '91 Course Note no.8*, pages V:14–21, August 1991.

[16] U. Tiede, K. Hoene, M. Bomans, and et alt. Investigation of medical 3d-rendering algorithms. *IEEE C. G.& A.*, 10(2):41–53, March 1990.

[17] C. Upson and M. Keeler. V-buffer: Visible volume rendering. *ACM Computer Graphics*, 22(4):59–64, August 1988.

[18] P. Van Renterghem. Transputer for industrial applications. *Concurrency: Practice and Experience*, 1(2):135–169, December 1989.

[19] L. Westover. Footprint evaluation for volume rendering. *ACM Computer Graphics*, 24(4):367–376, July 1990.

[20] J. Wilhelms. Decisions in volume rendering. In *State of the Art in Volume Visualization - ACM SIGGRAPH '91 Course Note no.8*, pages 1–11, July 1991.

[21] J. Wilhelms and A. Van Gelder. A coherent projection approach for direct volume rendering. *ACM Computer Graphics*, 25(5):275–284, July 1991.

[22] F.E. Yazdy, J. Tyrrel, and M. Riley. Carvupp: Computer assisted radiological visualization using parallel processing. In K. H. Hohne et al., editor, *NATO ASI Series - 3D Imaging in Medicine*, Vol.F60, pages 363–375. Springer-Verlag, 1990.