Parallel Rendering of Volumetric Dataset on Distributed Memory Architectures

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

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 solution is proposed to the problem of interactive visualization and rendering of volume data. Designed for parallel distributed memory MIMD architectures, the volume rendering system is based on the ray tracing (RT) visualization technique, the Sticks representation scheme (a data structure exploiting data coherence for the compression of classified datasets), the use of a slice-partitioning technique for the distribution of the data between the processing nodes and the consequent ray dataflow parallelizing strategy.

The system has been implemented on two different architectures: an INMOS Transputer network and a hypercube nCUBE 6400 architecture. The high number of processors of this latter machine has allowed us to exploit a second level of parallelism (parallelism on image space, or parallelism on pixels) in order to arrive at a higher degree of scalability.

In both proposals, the similarities between the chosen data partitioning strategy, the communications pattern of the visualization processes and the topology of the physical system architecture represent the key points and provides improved software design and efficiency. Moreover, the partitioning strategy used and the network interconnection topology reduce the communications overhead and allow for an efficient implementation of a static load balancing technique based on the pre-rendering of a low resolution image.

Details of the practical issues involved in the parallelization process of volumetric RT, commonly encountered problems (i.e. termination and deadlock prevention) and the sw migration process between different architectures are discussed.

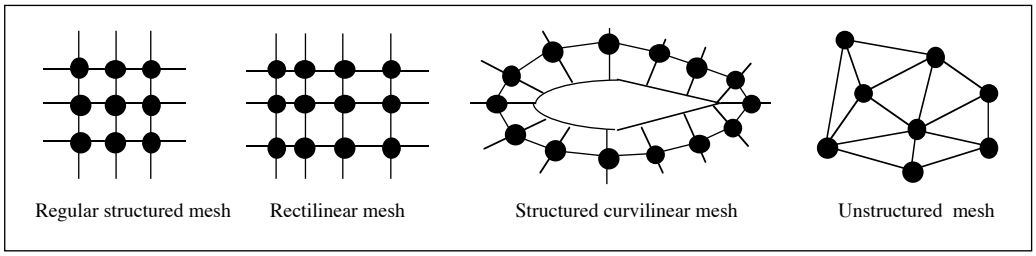

Figure 1: Volumetric dataset taxonomy.

1 Introduction

Applications requiring data visualisation 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*). *Volume Rendering* provides the user with representation data structures and tools to visualize such volume data.
The growth of 3D scanner devices (CT, PET, MRI, Ultrasounds) which directly input the properties of real 3D objects in machine readable format, and the increasingly urgent need to understand natural phenomena by analyzing large empirical data sets obtained by measures or simulations (several applications can be found, for example, in geophysics, astrophysics, meteorology, chemistry) has made Volume Rendering a subject of current interest [5].
Basic issues in Volume Rendering 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 visualisation process.

1.1 Taxonomy and representation schemes

The wide range of different volumetric datasets can be classified following the taxonomy given in figure 1; the schematical diagram presents, for the sake of simplicity, a 2D representation of the geometrical shape of different classes of volumetric data. In this paper we will refer exclusively to datasets defined on a *regular structured mesh* called, in brief, *voxel dataset*.

Voxel datasets can be simply represented by set of 2D matrices or, directly with 3D arrays. The use of data structures allowing compression of the spatial occupancy of the data can be extremely useful because of the huge amount of storage space required for high-resolution voxel dataset representation in uncompressed form. Using an appropriate representation scheme such as the Octree [12] or the Sticks [13] may lead to significative savings in space.

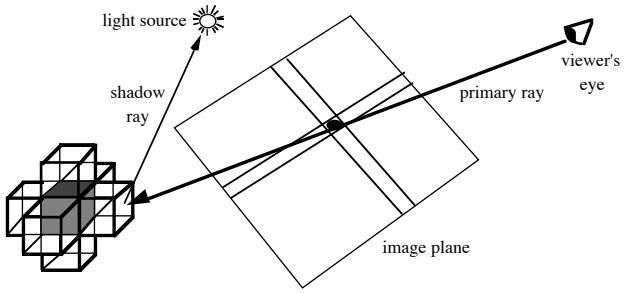

Figure 2: Ray tracing.

1.2 Volume Rendering

Two antithetic approaches are possible for rendering volumetric datasets. The first is based on the computation of a surface-model approximation of some of the iso-valued surfaces contained in the voxel dataset [11] and the use of standard surface rendering techniques [4] to visualize these surfaces. The second approach, known as *direct volume rendering*, visualizes the dataset without the need for an explicit voxel-to-boundary conversion. The latter approach is more appropriate to high-resolution voxel datasets, in terms both of results and efficiency.
Direct volume rendering techniques can be further subdivided into two classes: *projective* algorithms and *ray tracing* (RT) algorithms [20].

RT is a well–established technique for the visualisation of traditional geometrical models [7]. Ray tracing determines the visibility and the shaded color of the surfaces in a 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, a ray is fired from the *vep* through each pixel p in the image window. The closest ray-surface intersection identifies the visible surface, and a recursive ray generation and tracing process is applied to handle shadows, reflection and refraction (Figure 2).
Many RT algorithms were devised to render volume data [2] [10] [18]. Some modifications to the traditional RT model are necessary to manage the peculiarities of volume datasets. One of the most obvious is how rays are traced: computing and sorting ray-surface intersections is replaced by ray rasterization and voxel data access. Tracing a ray is therefore generally simpler in volumetric RT than in geometrical RT. The first requires the ray rasterization and 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 (when no acceleration techniques are used), with p the number of elementary primitives in the scene and, usually, $p >> n$.

The computational complexity of volumetric RT is further reduced with respect to traditional RT because much less "realism" is required in the visualization of volume datasets. Specular reflection effects and shadow computation do not generally need to be 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 total number of rays generated is therefore lower than that required by surface-based ray tracers, and can be estimated as linear with respect to the number of primary rays.

1.3 Towards interactive visualization

Image synthesis time is a critical issue, because the practical every day use of visualization as an analytical tool requires interactive dialogue between the user and the visualization system. A number of specialized and general purpose parallel architectures designed for volumetric rendering have been proposed; an extensive description can be found in the survey by Kaufman et al. [9] and Stytz et al. [16].

- **Special-purpose architectures**: among these is the *CUBE* architecture, which is characterized by an original memory organization and uses a multiprocessing engine; CUBE allows parallel accessing and processing of voxel beam with a raycasting-based rendering approach. The *INSIGHT* system represents voxel datasets by using a data representation 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 on a full scale; only low resolution prototypes or software simulation are currently available.

- **Implementations on general-purpose multiprocessors**: the *CARVUPP* system [21] 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 on 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 ray caster which works on multiprocessor workstations was proposed by Fruhauf [6]. The system is based on the transformation of the dataset from modeling space to viewing space. Rays are then casted by separate processors with ray direction always orthogonal

to the viewing plane.

1.4 Paper overview

Two parallel implementations of the same ray tracing algorithm [13] working on a STICKS-represented voxel dataset are proposed in this paper. Section 2 contains the presentation and assessment of the parallelization and data partitioning strategies.
Then, Section 3 reports a detailed description of the first implementation, devised for a small specialized Transputer network. The low cost of the system and the efficiency of the solution are the key points in this approach but low scalability is its main drawback.
Continuing from this early experience and in order to increase scalability, an evolved system was designed for a highly parallel hypercube (a nCUBE 6400 with 128 nodes), which is described in Section 4.
Finally, concluding remarks are presented in Section 5.

2 Parallel Ray Tracing of Volumetric Dataset

Numerous parallel systems have been proposed to speed up classical polygon-based RT. Much less research has been devoted to the exploitation of parallelism for volume RT. In our project we concentrated on the design of a parallel RT algorithm to visualize classified voxel datasets on distributed-memory MIMD architectures. These architectures allow efficient parallel implementations of RT and many commercial systems possessing good price/performances ratios are available nowadays.

2.1 Parallelizing strategies

According to Badouel, Bouatouch and Priol [1], the *parallelizing strategies* for classical surface-based RT on distributed memory environments can be classified by focusing on the kind of data transmitted on the processor interconnection network. Of the four approaches they classified, only two can be profitably exploited if applied to the rendering of volumetric dataset:

- **parallelism via image partition** (also known as parallelism on pixels or without dataflow): a *processor farm* [8] approach where a standard sequential RT implementa-

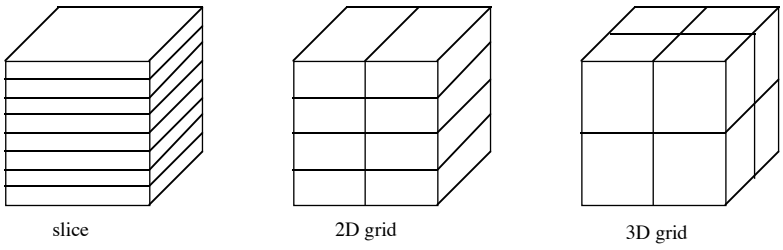

Figure 3: Partitioning strategies for a voxel dataset.

tion runs on each worker and the whole scene is replicated on the local memories of all the nodes. The computation of a subset of the image is assigned (statically or dynamically) to each worker. Since the computation of every pixel is independent of the others, implementations are straightforward and scalable, but the local memory requirements are often prohibitive.

- **parallelism with ray dataflow**: the scene data is partitioned and distributed among the nodes according to a *geometric parallelism* approach [8]. Each node rasterizes the rays inside the local partition of the dataset. A ray exiting the local partition is transmitted to the node managing the "adjacent" partition crossed by the ray for further tracing. The design of an RT code exploiting geometric parallelism is complex. The nodes communicate intensively and wholly asynchronously: deadlock prevention and termination detection have to be provided. As the cost of image synthesis on each node depends on both the viewer's eye position and the inner constituents of the assigned partition, load unbalance is a key issue and has to be carefully taken into account.

Unfortunately, because of the huge amount of memory needed to represent high–resolution voxel datasets, there is very little choice – a *with ray dataflow* design is compulsory. Even if the datasets have been previously classified and compressed, they typically require $2 - 100$ Mbytes of memory. Considering the limitations on local RAM space, the lack of virtual memory management and the limited I/O bandwidth common to most multicomputers, partitioning and distributing the data is therefore a must.

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 out

in Figure 3: monodimensional partitions (*slice partition*), bi-dimensional and three-dimensional partitions. All of these strategies can be used to produce regular or irregular partitions of the voxel dataset using regular or adaptive partition criteria.
During the RT process, inter-processor communications will be required for each ray that exits a partition allocated to one node and enters an adjacent one. Thus, each partition criterion implies different communications patterns between the RT processes which manage the voxel partition. A bi–directional channel must join each pair of processes which manage adjacent partitions, thus leading to a logical ring interconnection scheme for the slice partition criterion and 2D or 3D logical mesh interconnections for 2D or 3D grid partition criteria, respectively. All the above criteria provide simple and regular communications patterns.
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 communications locality by mapping the processes managing adjacent partitions on neighbour nodes in the hardware topology.

Our choice of a *slice partition* strategy, common to both the presented solutions, is 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 simple load balancing management based on dataset partition modifications. Shifting the cutting planes modifies the load associated to each node, while the node adjacency relations remain the same;
- the consequent communications pattern can be perfectly embedded in most topologies. 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 volumetric RT implementations based on a ray dataflow approach alone is not high. The relative simplicity of tracing voxel dataset involves a heavy trade-off between computation and interprocessor communications as the mean length of the ray sections traced by each node is reduced to a few tenths of voxels. As often happens with geometric parallelism, effective scalability under condition of constant data resolution is unlikely to be achieved by simply increasing the number of partitions. On the other hand, when the dataset resolution

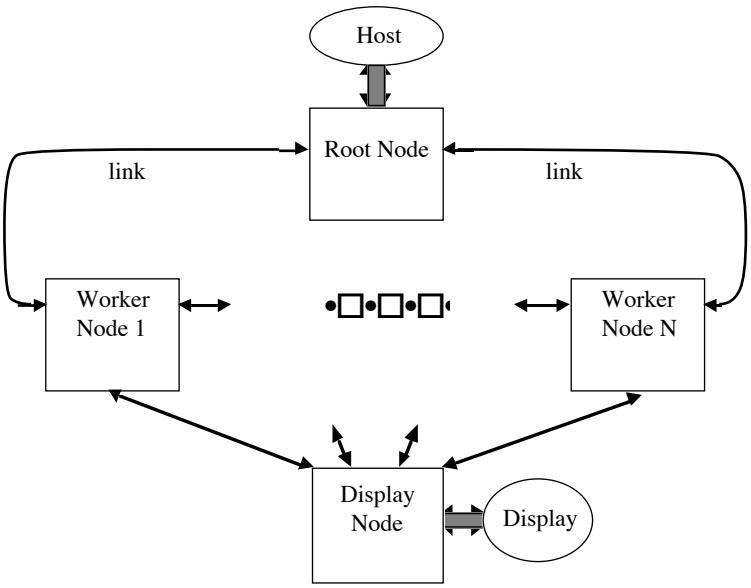

Figure 4: The architecture of the Transputer based system.

is increased, a proportional increase in the exploited parallelism can be attained by a finer partitioning of the dataset space. The key is the identification of the optimal partition width (measured in voxel unit) which produces good response times, an acceptable efficiency level and does not place prohibitive memory requirements on the nodes.
Unfortunately this choice depends strongly on the input, e.g. on the peculiar characteristics of the dataset to be rendered and on the requested visualization parameters (i.e. lights, projection and view settings).

On the basis of a slice partitioning strategy it was natural to adopt the *Sticks* scheme [13] for voxel dataset representation. This scheme is based on searching for one-dimensional coherence: all the adjacent voxels with an identical scalar value and belonging 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 belonging to a track are connected in a linked list; a pointer to each non-void track is stored in the *Sticks Holder Array*, an n^2 array of pointer cells with n the resolution of the voxel dataset. Good compression of classified voxel datasets and simple management are the main characteristics of the STICKS representation scheme.
The slice partitioning of the dataset reduces simply to the slice partitioning of the Sticks Holder Array.

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 represented dataset. A simple and low cost Transputer network [19] was designed to support this approach.

3.1 The hardware architecture

The hardware architecture of the system is shown in Fig. 4. The system does not require custom components and its physical topology allows optimal mapping of the communicating processes which make up the ray tracing algorithm.
The system consists of: (a) a Root node which interfaces the system with a personal computer (Host) supporting the file system and the user interface; (b) a number of Worker nodes running the RT algorithm on their data partition, interconnected in a bidirectional ring; (c) 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 graphics display.

3.2 The software architecture

The process running on the **Root node** has I/O functions only. It manages three main activities: (a) input stream scanning, (b) transmission of the STICKS-represented dataset to the Workers, (c) collecting results (images) and timing.

(a) *Input stream scanning:* the Root process reads the input stream, which contains the selection of one or more datasets plus the related visualization parameters (i.e. projection and view settings, threshold surface coefficients, etc.); it transmits input data to the Workers and broadcasts Start/Stop tracing messages

(b) *Sticks loading:* the Root node transmits the selected dataset to the Worker nodes. The limits of the partitions are computed and communicated to each Worker. Then, the Root starts transmitting blocks of Sticks 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 loading time for a 256^3 voxel dataset is a few seconds (15 .. 30 sec).

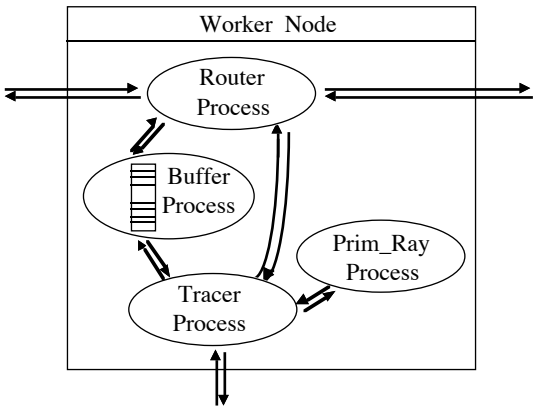

Figure 5: Processes and communications channels on the Worker Node.

(c) *Image collection and timing:* 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. At the end of the image synthesis process, each Worker sends monitoring and statistical data back to the Root.

Each **Worker** node runs the same code and stores a slice of the input dataset; Worker processes and communications channels are sketched in figure 5.
Each Worker starts to trace the primary rays whose first intersection with the bounding volume of the dataset is contained within its own partition. Ray tracing consists of a simple incremental scan conversion of the ray in order to identify all the voxels crossed by the ray. For each traversed voxel, the algorithm searches for a stick covering that voxel. 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.
If a ray exits from the space partition assigned to the Worker two different events can occur. If the ray enters 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 the bounding volume of the dataset, the background color is blended to the current ray color.

The **Router** and **Buffer** processes (see Fig. 5) manage communications between adjacent processors and a queue of ray descriptors, respectively.

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 is that *all* primary rays are generated and checked on *all* Worker nodes. We preferred to use a distributed method rather than a *centralized* one (where rays are generated and tested on the host and then transmitted to the Workers) in order to reduce communications overhead. Moreover, a distributed method allows for generating of 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 affects both load balancing and deadlock prevention.

The **Tracer process** receives ray descriptors 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 Worker W_i starts tracing a received primary ray r_1 in the voxel space partition allocated to W_i. As soon as the ray intersects a voxel associated with a threshold surface the first intersection p_1 is identified. 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 contribution of r_2 is blended with the contribution of r_1 and the resulting pixel value is transmitted to the Display node. Otherwise, if r_2 exits the W_i space partition of and enters that of an adjacent node, 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, 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 and starts tracing the next ray received from the *Buffer* or from the *Prim_Ray* processes.
Shadow rays are managed analogously: if they cannot be completely traced in the current partition they are transmitted to the adjacent node and the *Tracer* process saves the current tracing status in order to restore it as soon as the resolved ray descriptor is received. The *Tracer* process must therefore manage a list of partially processed primary ray descriptors.

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 and the approximated normal vectors in the intersection points p_i.
The *normal vector* is approximated by using an *object-space gradient* method [17] applied to the values 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 slightly larger than the associated partition to each Worker.

3.3 Load balancing and performance evaluation

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

We therefore implemented a *static* load balancing technique similar to that proposed by Priol et alt. [14]; it is based on a data redistribution which depends on the chosen view point and on the workload of each processor. The initial data partitioning is uniform and obtained through horizontal cutting planes. The Root node selects a subset of the primary rays which are regularly distributed on the image plane (for example a regular grid of 16*16 rays), and sends these test rays to the appropriate processors. On the basis of the total time t_i spent by each processor W_i to trace these test rays, the Root node defines a new subdivision of the dataset, with size[i] of each new partition computed as:

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

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 them 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 speedups obtained are reported in Table 1.

Table 1 reports the times relative to the synthesis of images with resolution 400*400 from a 256*256*109 magnetic resonance scan of a human head.
The images come from different views, the first viewing direction is orthogonal to the XY plane (frontal view, *FV* in the tables) and the second at a 45 degree angle to the Z and X axes and orthogonal to the Y axis.

Head dataset	$W_1 elap.$	$W_1 act.$	$W_2 elap.$	$W_2 act.$	$W_3 elap.$	$W_3 act.$	$W_4 elap.$	$W_4 act.$
$2W, FV$	15	11	15	12				
$4W, FV$	6	4	7	5	9	6	8	6
$2W, 45^0V, NoBal$	106	38	106	96				
$2W, 45^0V, Bal$	81	68	80	71				
$4W, 45^0V, NoBal$	80	25	80	19	80	31	80	71
$4W, 45^0V, Bal$	54	34	54	41	52	39	51	39

Table 1: Processing times, in seconds, 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 ($W_i elap.$) and the $active$ times ($W_i act.$), where the latter equals the elapsed time minus the overhead time spent waiting for ray communications.

SOD dataset	$2Workers$	$4Workers$	$eff.$	$8Workers$	$eff.$
$10^0V, Bal$	448	232	0.96	136	0.82
$45^0V, Bal$	450	234	0.96	138	0.81

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

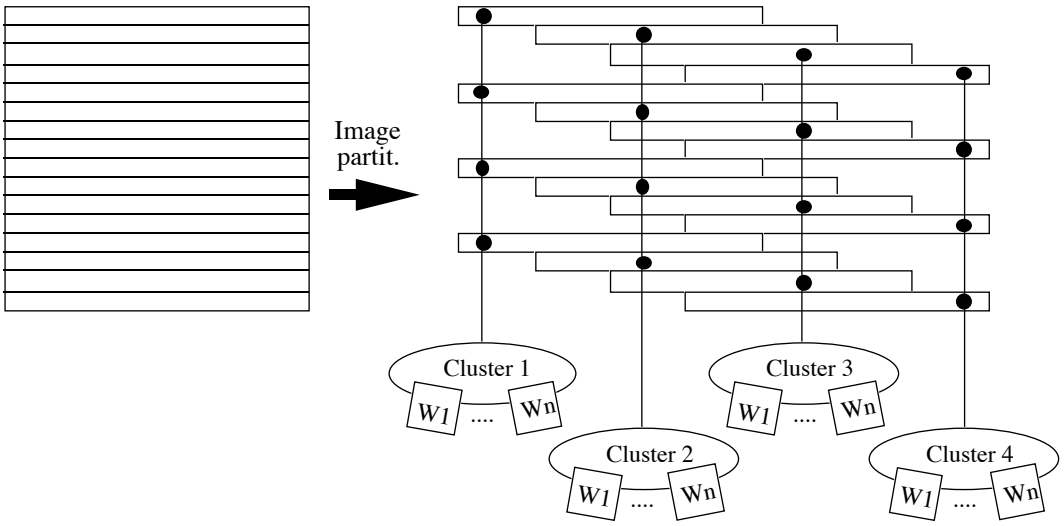

Figure 6: A hybrid approach to parallel volume rendering: a *farm* of *geometric* workers (clusters).

Table 2 reports the times of the same algorithm on a 194*194*232 dataset, obtained by the interpolation of a 97*97*116 dataset which represents the electron density map of an enzyme (SOD, Super Oxide Dismutase).
Single Worker runs are not reported on either table because neither dataset completely fits into a single node local memory.

4 A scalable hybrid solution to volume RT

The scalability of a *ray dataflow* volume ray tracer based on slice partitioning is low. As soon as the number of partitions is increased, the communications overhead increases and the mean computational cost to trace each ray inside a partition decreases, thus reducing the efficiency of the system. However, ray tracers based on different partitioning strategies (e.g. 2D or 3D grids) also prevent the exploitation of high parallelism.
To achieve a real scalability of a volumetric ray tracer we propose the adoption of a hybrid parallelization strategy *with and without dataflow*. It is based on the geometric decomposition of the workers of a processor farm which exploits parallelism at the pixel level (Figure 6). We subdivide the nodes of the system into clusters; each cluster has to compute a part of the final image. The dataset is therefore replicated on each cluster, but it is partitioned among the local memory of the cluster nodes as in the previous solution.
This solution has been implemented on a 128-node hypercube (a nCUBE 6400). In the software

design the 2^k processing nodes of the hypercube are logically divided into 2^i **clusters**, each composed of 2^{k-i} nodes. Each cluster acts as a single worker in the processor farm and computes the assigned pixels according to the *with dataflow* method described in depth in the previous section. The resulting organization leads to a set of nodes which communicate according to a multi-ring pattern. To exploit communications locality this pattern has been embedded into the hypercube topology preserving the neighbourhood relations.

Due to their architectural characteristics, some differences exist between the cluster–level code implemented on the nCUBE system with respect to the Transputer code. First of all, the messages are routed on the nCUBE by a hardware virtual cut-through unit; no buffer and router processes are therefore needed and only a *Tracer* process runs on each node. Secondly, due to the lack of a graphics board, the images synthetized on the parallel machine are sent back to the host workstation (a SUN 4/330GX).

At start–up each *Tracer* process locally computes its cluster number and the position inside the cluster with simple transformations of its node address (the gray code assigned to each node). The dataset is partitioned into slices, and each slice s_j of the volume data is replicated on the local memory of the node w_j on all the clusters. The position of a node inside a cluster univocally identifies the dataset slice assigned to it. From secondary storage each node can asynchronously load the data subset contained in the assigned space partition.

The image is partitioned and a subset of pixels is statically assigned to each cluster. The strategy chosen to partition the image between clusters is extremely simple: the image is divided into rows of pixels; row r_i, with $1 \leq i \leq N$ and $N * N$ the image resolution, is assigned for computation to the $(iDIVm)^{th}$ cluster. The resulting load is well balanced among the clusters because the time for the synthesis of rows which are adjacent or close in the image is nearly equal, if the hypothesis of $N >> m$ holds.

For the sake of optimum scalability, the system has been designed with the aim of exploiting the flexibility of hybrid parallelism. In fact, depending on both the specific characteristics of the datasets to be visualized and the requested visualization parameters, a different configuration of the system (i.e. in terms of the number of cluster and of the parallelism degree inside each) can be chosen. The user can specify the dimension k of the hypercube to be used and the number of clusters in which the hypercube has to be configured as input parameters. For example, the user can choose to synthesize the image following a pure *without dataflow* approach by selecting a unitary dimension for the 2^k clusters; or, at the other extreme, a pure *with dataflow* approach, by configuring the system as a single cluster of 2^k nodes.

SOD dataset	$1node$	$2nodes$	$4nodes$	$8nodes$	$16nodes$	$32nodes$	$64nodes$	$128nodes$
$time$	485.24	260.35	130.19	65.82	32.96	17.26	8.90	5.14
$speedup$	1.	1.86	3.73	7.37	14.72	28.11	54.52	94.40
$efficiency$	1.	0.93	0.93	0.92	0.92	0.88	0.85	0.74

Table 3: Times (in seconds), speedups and efficiency of visualization of the *SOD* dataset using an increasing number of **nCUBE** nodes.

4.1 System evaluation

Some results demonstrating the hybrid system's effective exploitation of parallelism are presented in Table 3. The table contains the image synthesis times at a resolution of 350*250 from the same dataset as Table 2.
Images are generated with a viewing direction at 45 degree angle to the Z and X axes and orthogonal to the Y axis.
Table 4 reports times and the efficiencies measured on the same dataset as Table 3, using different configurations of the hypercube nodes: the hypercube dimension is fixed, and the number of nodes in each cluster (*Cd*: cluster dimension) decreases from 32 to 1. Even if the resolution of the considered dataset is not too high to be fully replicated on each node's local memory (i.e. when a unitary dimension of each cluster is selected), the lowest rendering times with 128 nodes are still obtained when the clusters are composed of two nodes. This is because the static strategy used to distribute the pixels among the workers works well only if the coherence properties of the resulting image are exploited. In others words, the load is balanced only if the resolution of the image is much greater than the number of clusters used for its synthesis (the ratio between the image resolution and the number of clusters must be greater than or equal to 4 or 5).

5 Conclusions

We have presented two different implementations of a volumetric rendering system working on distributed memory MIMD parallel architectures. The first implementation, on a INMOS Transputer network, exploits ray dataflow parallelism based on a simple and efficient data partitioning strategy. The second, on an nCUBE architecture, is an extension of the former solution

SOD dataset	$Cd = 32$	$Cd = 16$	$Cd = 8$	$Cd = 4$	$Cd = 2$	$Cd = 1$
64nodes, $times$	36.50	24.16	17.16	11.93	10.10	8.90
$efficiency$	0.21	0.31	0.44	0.64	0.75	0.85
128nodes, $times$	18.53	12.32	8.77	6.70	5.14	6.42
$efficiency$	0.20	0.31	0.43	0.57	0.74	0.59

Table 4: Times (in seconds) and efficiency measured on the SOD dataset with decreasing number of nodes in each cluster (Cd: cluster dimension).

and attains a good level of scalability by exploiting image partition parallelism as well.
The presentation does not attempt to compare between the two architectures used in the project: consideration of the relative costs of the machines is enough to make any such comparison meaningless. The goal of this work was, on one hand, to test parallel volume rendering techniques and, on the other, to evaluate the operating environments of the different architectures. In this sense it is noteworthy that the migration from the INMOS Transputer environment to the nCUBE environment did not involve excessive cost in terms of software modifications (mainly restricted to the parts of the code dealing with communications and I/O control).
Even though we are rather far from software portability (which will hopefully increase with the diffusion and standardisation of parallel programming environments such as Express and Linda) we can assert that migration of parallel software between architectures belonging to the same class (distributed memory MIMD machines, in our case) does not pose particular difficulties.

Acknowledgements

This work was partially funded by the Progetto Finalizzato "Sistemi Informatici e Calcolo Parallelo" of the Consiglio Nazionale delle Ricerche.
The magnetic resonance scan dataset of Table 1 was taken on the Siemens Magnetom and was provided courtesy of Siemens Medical Systems Inc., Iselin, NY. The electron density map used in Tables 2,3,4 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 distributed the above and other datasets.
The authors also thank Paolo Bussetti, Luca Misericordia and Marco Tartaglia for their valuable contribution to the systems implementation.

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] T. Fruhauf. Volume rendering on a multiprocessor architecture with shared memory: A concurrent volume rendering algorithm. In 3^{rd} *EuroGraphics Workshop on Scientific Visualization*, Pisa, April 1992.

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

[8] 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 Springer-Verlag.

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

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

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

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

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

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

[15] P. Schroder and J. Salem. Fast rotation of volume data on data parallel architectures. In *Proceedings of IEEE Visualization '91*, October 1991.

[16] M.R. Stytz, G. Frieder, and O. Frieder. Three-dimensional medical imaging: Algorithms and computer systems. *ACM Computing Survey*, 23(4):421–499, December 1991.

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

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

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

[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] 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*, volume F60, pages 363–375. Springer-Verlag, 1990.