

Performance of Parallel Volume Graphics Algorithms on SGI Power Challenge

C. E. Prakash and Arie E. Kaufman

March 11, 1997

Abstract

The shared-memory multiprocessor system recently has attracted a lot of attention because of its low cost and wide availability to the graphics researchers. In this paper, we show practical parallel solutions to the following volume graphics[4] problems: volume texture modeling, volume texture rendering, and voxelization for unstructured grid rendering. All these solutions process 2D and 3D arrays and employ P processors on the shared-memory system. We present results for parallel execution time, parallel memory requirement and overheads for the three algorithms described in this work. We also present results of performance prediction based on both analytical and experimental results.

1 Introduction

Many efficient parallel graphics algorithms have been proposed in the literature for the shared-memory class of machines. These include algorithms for: volume splatting, volume resampling, shear-warp, etc. We refer to this group of parallel machines having up to 100 nodes as Class I systems.

Class II systems from 100 to 1000 processors have been developed commercially. Class III systems have more than 1000 processors. Several graphics algorithms have been developed and reported in literature for Class II and Class III systems. The emphasis in Class II and III has been to identify parallelism that could effectively utilize large number of processors. These algorithms, even though have linear and super linear speedups, are of no practical use today due to its limited availability.

The motivation of this work is to:

- demonstrate practical parallel algorithms for volume graphics on shared-memory systems.
- identify parallel algorithms that can be effectively used by a large spectrum of users who have access to two or more processors.
- exploit the execution characteristics of the powerful processor in each node that dramatically differ from the Class II and III systems.
- explore efficient partitioning schemes on Class I machines which do not have to worry about mapping of data for efficient communication.
- motivate graphics researchers to use practical parallelism in their day-to-day research.
- to provide a performance measure which could be used by graphics researchers and users to analyze existing sequential algorithms for possible parallel implementation.

Hence the emphasis of this work is to provide parallel algorithms for the relatively widespread and easy to use Class I machines. In this paper we present new results of parallel performance estimation on Shared-Memory Multiprocessor systems. The following algorithms have been implemented and the performance has been experimentally verified on a 16 CPU SGI Power Challenge:

1. Volume Texture Modeling(VTM) of Terrain
2. Volume Texture Rendering(VTR) of Terrain
3. Coherent Voxelization Algorithm(CVA) for Unstructured Grids

2 Parallel Rendering Algorithms

2.1 Volume Texture Modeling of Terrain

This is a new system for visual simulation of a volumetric terrain model. The input is the elevation data and corresponding aerial photographs or satellite images of a terrain. The output is a volumetric terrain model of the terrain. The sequential algorithm for Volume Texture Mapping is as follows:

```
VTM_sequential()
{
  for(all_scanlines_in_the_image) {
    for(each_pixel_in_scanline){
      vox_column = texture_map(elev,color);
      write column of voxels in vol_buffer
    }
  }
}
```

```

for(each voxel in intermediate volume) {
    tex_volume[voxel] = splat_kernel();
}
}

```

A voxel column is computed for each pixel in the terrain image. Each column can be computed independently. A simple yet inexpensive way of parallelization is to partition the image into scanlines and assign them to processors. Since there is no communication there is no need for mapping of adjacent scanlines to adjacent processors. The parallel texture mapping algorithm is shown below:

```

VTM_parallel()
{
    for(all_scanlines_in_this_processor) {
        for(each pixel in scanline){
            vox_column = texture_map(elev,color);
            write column of voxels in vol_buffer
        }
    }
    for(each voxel in intermediate volume) {
        tex_volume[voxel] = splat_kernel();
    }
}

```

For implementation on a shared-memory multiprocessor, the input elevation and image is loaded in shared-memory and the output volume-buffer is stored in shared-memory.

2.2 Volume Texture Rendering of Terrain

The VolVis system has been extended for parallel raycasting of a Volume Texture Model on a shared-memory system. The sequential algorithm of VTR in VolVis is as follows:

```

VTR_sequential()
{
    for(all_scanlines_in_the_image) {
        for(each pixel in scanline){
            pixel = cast_ray(current_pixel);
        }
    }
}

```

A ray is cast for each pixel in the final image. Each pixel can be computed independently. The image is partitioned into scanlines and assigned to processors. There is no communication among processors. The parallel raycasting algorithm is shown below:

```

VTR_parallel()
{
    for(all_scanlines_in_this_processor){

```

```

        for(each pixel in scanline)
            pixel = cast_ray(current_pixel);
    }
}

```

For implementation on a shared-memory multiprocessor, the input volume-buffer is loaded in shared-memory and the output image is stored in shared-memory.

2.3 Voxelization for Unstructured Grid Rendering

Voxelization has been proposed as an efficient method for rendering large unstructured grids [9][8]. The goal here is to achieve improved performance of parallel voxelization-based volume rendering of unstructured grids. The unstructured grid is decomposed into convex cells. The summary of the algorithm is as follows:

```

CVA_sequential()
{
    for ( each cell in grid) {
        for ( each face of cell){
            form_buffers(current_polygon)
            if(Front Face)
                scan_convert(to_frontbuffer)
            else if(Back Face)
                scan_convert(to_backbuffer)
        }
        voxelize_between_two_buffers()
    }
}

```

The *voxelize_between_two_buffers()* procedure interpolates the scalar values for all voxels within the cell.

In the coarse grain approach, one cell is assigned to a processor and voxelization is done on the individual cell. All other computations for the cell are done on the same processor. This helps the concurrent execution of cells in different processors because the cells do not intersect each other. The algorithm for parallel voxelization is given below:

```

CVA_parallel()
{
    for ( each cell assigned to this processor) {
        for ( each face of cell){
            form_buffers(current_polygon)
            if(Front Face)
                scan_convert(to_front_buffer)
            else if(Back Face)
                scan_convert(to_back_buffer)
        }
        voxelize_between_two_buffers()
    }
}

```

}
}

In the parallel version, all the cells in the grid are not voxelized on the same processor. Each processor voxelizes only those cells which are assigned to it.

For implementation on a shared-memory multiprocessor,

- The input unstructured grid is loaded in shared-memory.
- The output volume-buffer is stored in shared-memory.
- The frame-buffers and z-buffers for each cell are local to the processor.

2.4 Related Work

Parallel implementations on Class I systems have been discussed in [2] [10], [7], [5] and [6]. In this paper we limit our discussion to parallel implementations on small parallel machines.

2.5 Design Goals

The major issues and our goals in parallel algorithms for graphics are:

- **Data Duplication:** The input data is usually a grid which consists of voxels in a regular grid or cells in an unstructured grid. Data duplication happens when more than one copy of part or all of the grid occurs. However such a problem is not normally encountered on shared-memory systems. Our design goal is not to duplicate the data.
- **Synchronization Requirement:** Most algorithms forces the synchronization among processors. Our design goal is to avoid the need for synchronization among the processors.
- **Critical Regions:** When multiple processors compute a scene in parallel, all processors cannot write concurrently to the graphics subsystem. This forces the use of spin-locks. Our objective is to avoid busy waiting of processors on a spin-lock.
- **Idle Processors:** This situation arises when there is no proper load balancing. Our aim is to avoid additional computation required for load balancing, and to provide good load balancing without much computation for load balancing.
- **Rigid Data Partitioning/Task Allocation:** Existing methods require adjoining cells and adjacent scanplanes to reside in the same processor. Our goal is to eliminate such dependencies.

Table 1: Parallel Execution time of VTM(in seconds).

Terrain Model	No. of Processors					
	1	2	4	8	12	16
TERR1	16.73	11.12	5.90	3.13	2.19	1.82
	89.95	46.05	25.27	14.28	10.36	8.95
	389.39	196.61	100.75	54.78	39.81	36.11
TERR2	16.66	11.12	5.88	3.13	2.18	1.84
	89.71	45.91	25.13	14.32	10.34	8.83
	388.65	196.18	100.29	55.55	39.83	35.98

Table 2: Parallel Execution time of VTR(in seconds).

Texture Render	No. of Processors					
	1	2	4	8	12	16
256x256	4.74	2.71	1.54	0.89	0.73	0.61
512x512	18.02	9.94	5.53	3.07	2.43	2.30
1024x1024	79.30	42.89	22.96	13.03	10.53	9.77
256x256	8.41	4.95	2.64	1.44	1.17	1.01
512x512	27.49	17.18	9.44	5.10	3.88	3.15
1024x1024	92.38	55.99	30.95	15.43	12.44	11.11

- **Interprocessor Communication:** This is another major overhead in several parallel volume rendering algorithms. Our aim is to reduce and, if possible eliminate this overhead.

3 Implementation and Results

We have used a 16 processor MIPS R10000 (194 MHz) based Power Challenge iR graphics system with 3 GB CPU memory.

The experimental results for VTM, VTR and CVA are shown in Table 1-3.

4 Parallel Performance Estimation

Several methods have been proposed in literature for analyzing the performance of parallel algorithms. Two important schemes relevant for graphics algorithms are Green[3] and Tom [1]. The two schemes have been extended to analyze the performance and scalability of parallel volume graphics algorithms on shared address

Table 3: Parallel Execution time of Voxelization of Grids (in seconds).

Grid (Cells)	No. of Processors					
	1	2	4	8	12	16
BFIN	3.78	2.44	1.49	1.07	0.79	0.57
	14.23	10.42	7.32	4.54	3.75	2.97
	94.86	59.48	32.29	17.58	13.79	9.85
POST	5.86	4.42	2.43	2.21	2.16	2.11
	14.20	8.75	5.28	3.53	3.15	3.13
	55.45	31.88	18.39	11.77	9.37	9.10

Table 4: Estimation of parallel performance (Green’s method)

Algorithm	CPU R_1	R2	α	p_{max}	s_{max}
VTM	388.65	196.18	1.85	105	52.74
VTR	92.38	55.99	9.80	5	2.76
CVA	55.45	31.88	4.15	7	3.85

space systems. The Green’s method helps in analyzing the peak performance and the pek scalability of the algorithm The Tom’s method helps in the prediction of performance for more processors.

4.1 Model of Performance - I: Green’s Method

The objective here is to estimate the scalability of the three parallel volume graphics algorithms when more CPUs are added. Let

- p - total number of processors
- R_1 - the time for execution using 1 cpu
- α - the overhead in adding a CPU

The average rate of processing R in a system with p processors is

$$R(t) = R_1 - \alpha(p - 1)$$

Total performance of the system is

$$T(p) = pR(p)$$

To estimate the speedup s , let

- t_1 - time for single processor
- t_p - time for p processors

The speedup:

$$\begin{aligned} s(p) &= t_1/t_p \\ &= T(p)/T(1) \\ &= pR(p)/R(1) \\ &= p(R_1 - \alpha(p - 1))/R_1 \\ &= ((1 + \alpha)/R_1)p - \alpha p^2/R_1 \end{aligned}$$

For peak performance:

$$\begin{aligned} dT/dp &= 0 \\ d/dp(p(R_1 - \alpha(p - 1))) &= 0 \text{ then} \\ R_1 - 2\alpha p + \alpha &= 0 \end{aligned}$$

The peak performance p_{max} is obtained when $dT/dp = 0$ and s_{max} the corresponding s . Then p_{max} and s_{max} are expressed in terms of R_1 and α as follows:

$$\begin{aligned} p_{max} &= (1/2)(1 + R_1/\alpha) \\ s_{max} &= (1/4)(1 + R_1/\alpha)(1 + \alpha/R_1) \end{aligned}$$

The peak performance p_{max} and scalability s_{max} are estimated for all the three volume graphics algorithms and results are shown in Table 4.

4.2 Model of Performance - II: Tom’s Method

This method also analyses the scalability of a parallel algorithm by estimating the overhead involved in

parallelization due to the addition of more CPUs. This method has been used to predict the performance when more CPUs are added for parallel execution.

4.2.1 Performance Prediction

The overheads during the different stages of execution are:

- t_{trans} - overhead in input grid transformation.
- t_{split} - overhead in splitting grid into sub-grids.
- t_{init} - overhead in volume buffer initialization.
- t_{exec} - overhead in execution of cells.
- t_{store} - the overhead in storing voxels.

The total time is then expressed as:

$$T_{parallel} = Cn/p + t_{exec} + t_{store} + t_{trans} + t_{split} + t_{init}$$

where n is the number of cells, p is the number of processors, and C is a constant which is dependent on the data.

The overheads may be either due to the extra computation required due to parallelization, due to the contention for shared bus, message communication latency, synchronization requirement or due to variations in workload.

In the shared address space scheme, the computation is done by assigning the workload to processors. The different buffers reside in shared-address space. The overheads are t_{exec} and t_{store} only. The execution stage requires access of local buffers from shared memory which may lead to contention. The t_{exec} is also due to the shared access for the buffers. The other overhead arises during the storage of output from several processors into the volume buffer in shared-address space. This could be easily isolated. Let us take a scene with uniform workload. By adding an additional processor one half of the workload is computed on each processor. The time taken is more than the execution of just the first one-half by a single processor. This overhead incurred is due to t_{exec} and t_{store} . This is used to predict the performance of the system when multiple CPUs are added for computation.

$$t_{overhead} = t_{exec} + t_{store}$$

The overhead is assumed to be constant since we assume that the design of the shared-memory system provides the necessary bandwidth in the shared-bus and hence does not lead to contention.

4.2.2 Performance Prediction for VTM

For the VTM, when it is texture mapped into a volume buffer of size 570x66x695 the overhead for adding an additional CPU from the experiment is 1.855 seconds.

Table 5: Estimation of parallel performance (Tom’s method)

Grid (Cells)	No. of Processors					
	4	8	12	16	32	64
VTM	99.01	50.43	34.24	26.14	14.00	7.92
(Expt)	100.29	54.78	39.81	36.11		
VTR	32.89	21.34	17.49	15.57	12.68	11.24
(Expt)	30.95	15.43	12.44	11.11		
CVA	18.01	11.08	8.77	7.61	5.88	5.01
(Expt)	18.39	11.77	9.37	9.10		

The estimation expression assumes a constant overhead of 1.855 seconds.

In the case of VTM, when $p = 1$, for a single CPU the time taken for the Terr1 terrain data is 388.65 seconds.

$$Cn/p = 388.65$$

$$Cn = 388.65$$

For $p = 2$, the $T_{overhead}$ from experiments is computed as follows:

$$\begin{aligned} T_2 &= Cn/p + T_{overhead} \\ T_2 &= 196.18 \\ T_{overhead} &= T_2 - Cn/2 \\ &= 196.18 - 388.65/2 \\ &= 1.855 \end{aligned}$$

4.2.3 Performance Prediction for VTR

For the VTR, when it is rendered into a volume buffer of size 570x66x695 the overhead for adding an additional CPU from the experiment is 9.8 seconds. The estimation expression assumes a constant overhead of 9.8 seconds.

4.2.4 Performance Prediction for CVA

For the CVA, when it is voxelized into a volume buffer of size 1000x1000x160 the overhead for adding an additional CPU from the experiment is 4.15 seconds. The estimation expression assumes a constant overhead of 4.15 seconds.

The above assumption holds good since there is not much variation in overhead as obtained from the experiments when more processors are added. Predicted time for p processors for the VTM, VTR and CVA are shown in Table. 5.

Several data sets have been tested for the three algorithms and results are shown in Fig. 4-9.

5 Conclusion

The performance of this algorithm gets significantly degraded when multiple CPUs are used for execution. One possible reason is due to the number of memory references that arises frequently due to load and store

operations which is not a desirable aspect for super-scalar processors.

References

- [1] Thomas W. Crockett and Tobias Orloff. Parallel polygon rendering for message passing architectures. *IEEE Parallel and Distributed Technology*, 2(2):17–28, Summer 1994.
- [2] Christopher Giertsen and Johnny Petersen. Parallel volume rendering on a network of workstations. *IEEE CG&A*, 13(6):16–23, Nov 1993.
- [3] Stuart Green. *Parallel Processing for Computer Graphics*. The MIT Press, Cambridge, MA, 1991.
- [4] Arie E. Kaufman, Daniel Cohen, and Roni Yagel. Volume graphics. *IEEE Computer*, 26(7):51–64, Jul 1993.
- [5] Philippe Lacroute. Real-time volume rendering on shared memory multiprocessors using the shear-warp factorization. *Proc. of Parallel Rendering Symposium 1995, ACM Press*, pages 15–22, Oct 1995.
- [6] Philippe Lacroute. Analysis of a parallel volume rendering system based on the shear-warp factorization. *IEEE Trans. on Visualization and Graphics*, 2(3):218–231, Sep 1996.
- [7] Bruce Lucas. A scientific visualization renderer. *Proc. of Visualization '92*, pages 227–234, Oct 1992.
- [8] C. E. Prakash and S. Manohar. Shared-memory multiprocessor implementation of voxelization for volume visualization. In *HPC for Computer Graphics and Visualization, M. Chen, P. Townsend and J. A. Vince (Eds), Springer, London*, pages 135–145, 1995.
- [9] C. E. Prakash and S. Manohar. Volume rendering of unstructured grids: A voxelization approach. *Computers and Graphics*, 19(5):711–726, Sep/Oct 1995.
- [10] Peter L. Williams. Interactive direct volume rendering of curvilinear and unstructured data. *PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign*, 1992.

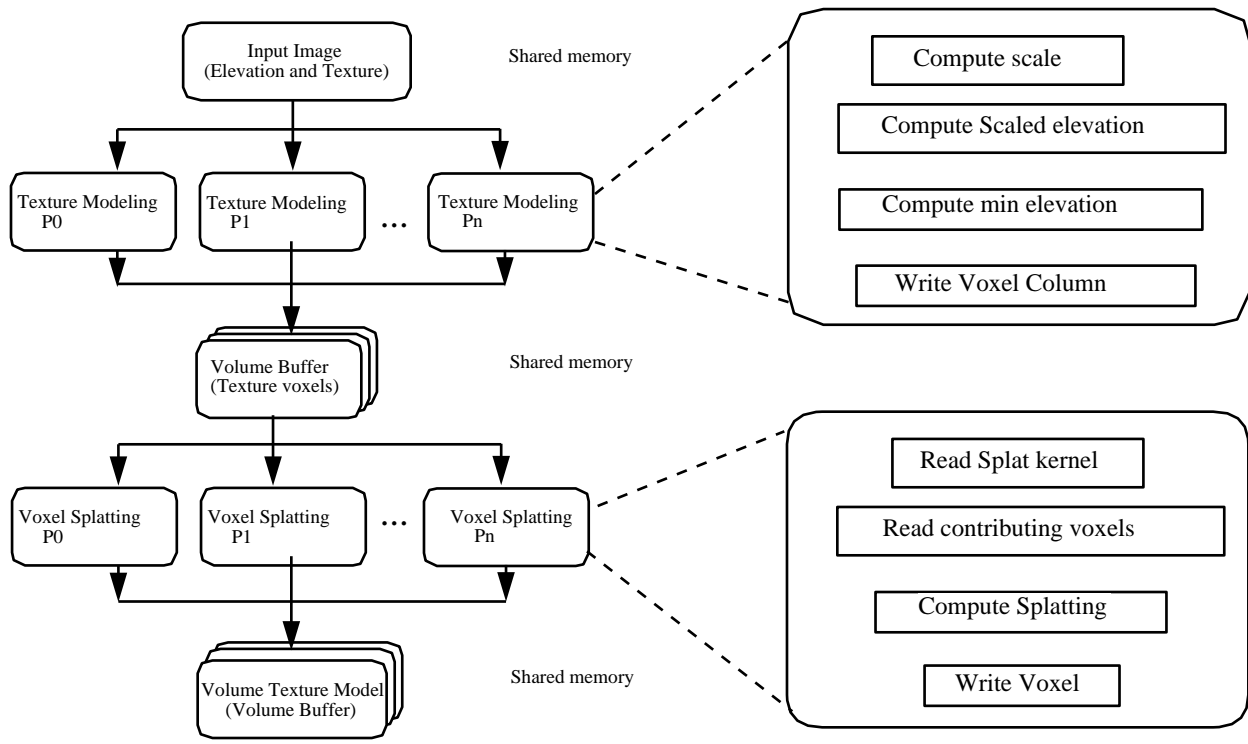


Fig. 1. Parallel Volume Texture Modeling of Terrain

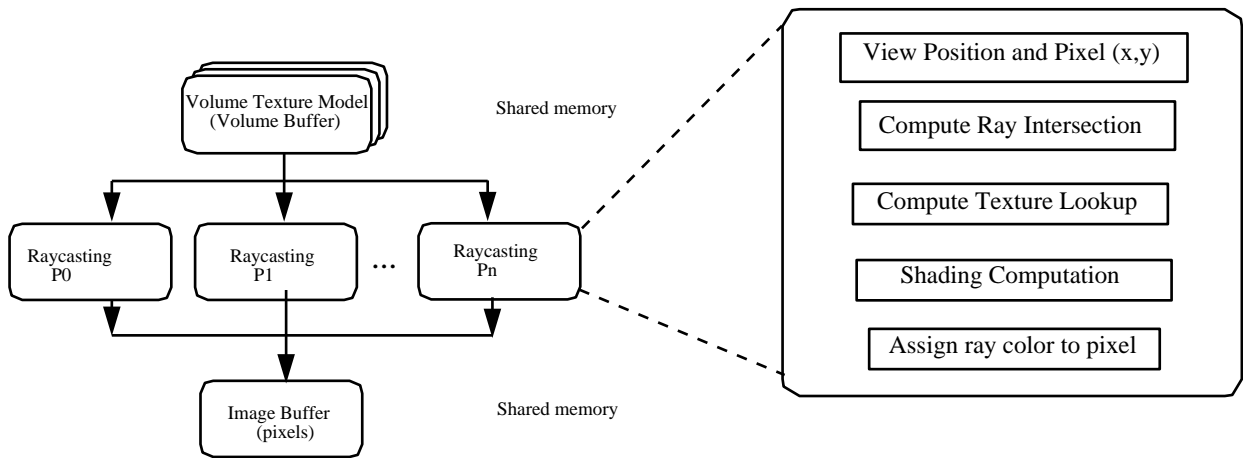


Fig. 2. Parallel Volume Texture Rendering of Terrain

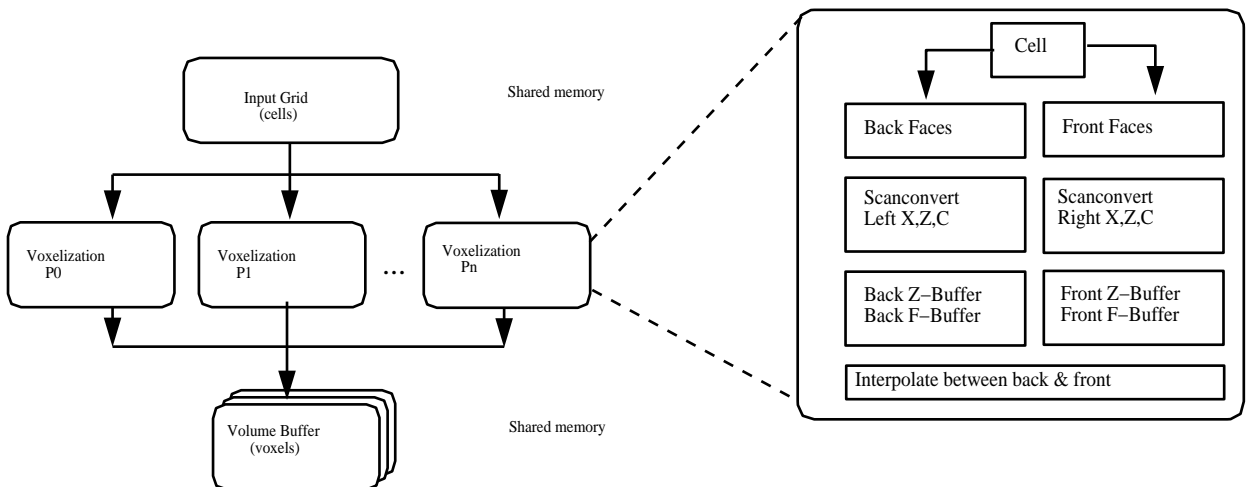


Fig. 3. Parallel Voxelization of Unstructured grids