Performance Experiments with the StormView Distributed Parallel Volume Renderer

Jo Asplin and Dag Johansen

Department of Computer Science
University of Tromsø, Norway

Abstract

Volume rendering is a useful but cpu-intensive method for visualizing large scalar fields. The time to render a single image may be reduced by parallel processing. This paper reports on performance experiments with the StormView volume renderer, which is parallelized on a set of 57 MIPS / 17 MFLOPS workstations connected by a 10 Mbps Ethernet. For certain user patterns, we show that our parallelization exhibits substantial speedups. We compare the performance of a dynamic and a static load balancing algorithm.

1 Introduction

Many sciences need to interpret large 3D (three-dimensional) data sets. One example is weather predictions produced by an atmosperic model where the state of the atmosphere is represented as a grid of numbers. Another example is electron density fields studied in chemistry. Obviously, a scientist does not want to deal directly with a list of perhaps 10^6 numbers, so some form of visualization is desirable. Volume rendering is a popular method of visualizing 3D data sets [5]. It consumes substantial time- and space-resources. Parallel processing is one of the most common ways to support volume rendering [6, 8, 11].

The speedup achieved by a parallel algorithm is often related to load balancing. The goal of load balancing is to maximize the fraction of time that each processor is busy working on the problem and minimize the fraction spent communicating or being idle. Load balancing methods are often classified as dynamic or static. Whereas dynamic methods uses communication to adjust the work assignment as the computation progresses, static methods do not change the initial assignment. Dynamic methods often prove to be superior if the problem exhibits a high computation to communication ratio, and the general load variation of the execution environment is difficult to predict.

The output of a volume rendering algorithm is typically described by 1 MB or so of image data (assuming a screen-resolution of 1000x1000 pixels). In a parallel execution, this data has to be transferred from the worker processes to a single controller process responsible for assembling and displaying the image. At one extreme, the workers could postpone the transfer of image data until after the computation. At the other extreme, the image data could be transferred in small fragments during the course of the computation. Whether one strategy performs better than the other, depends on the computation to communication ratio of the overall computation.

Another issue of relevance to the performance of a distributed parallel volume renderer is the user pattern with respect to the number of images produced per data set. At one extreme, a single data set is visualized throughout a session. At

the other extreme, a new data set has to be loaded before the generation of each new image.

In this paper, we report on StormView, a parallel volume renderer running on a local area network of workstations shared between several users. We have implemented well-known static and dynamic load balancing methods, each of which may either scatter or concentrate the transfer of image-parts. This allowed us to study the performance of the four algorithms experimentally in a multiuser network. In addition, in order to evaluate the performance characteristics of the two user pattern extremes, we also measured the time taken to transfer the data set to the parallel processes.

The rest of the paper is organized as follows. In section 2 we describe the functionality of the StormView volume renderer. Section 3 presents the general parallel interaction model employed in StormView, along with design choices related to this model. Section 4 gives a motivation for load balancing, and discusses the methods chosen. In Section 5 we describe the experiments. The results are discussed in Section 6 before we summarize the paper in Section 7.

2 The StormView Volume Renderer

StormView is a system that volume renders large scalar fields [1]. In this paper, meteorological data sets is used. StormView has been constructed as part of the StormCast project which applies distributed computing to the meteorology and environmental (pollution) domains [4].

2.1 Input to StormView

The input to StormView consists of a *grid object* (GO) and a *rendering-specification object* (RO). The grid object represents a 3D scalar field with one scalar value at each grid point. The grid is rectilinear, meaning that the grid points are axis-aligned, but not necessarily evenly spaced. A concrete example of a grid object is the 121x97x18 grid output from the Norwegian Meteorological Institute's atmospheric model LAM50S covering most of Europe. Scalar values suitable for volume rendering include horizontal wind speed and relative humidity.

The rendering-specification object contains parameters to control a single rendering of a grid object. These include 3D viewpoint, light source, atmospheric attenuation, and functions for mapping scalar values to color and opacity. Mapping functions control what regions in the data set are rendered, the degree of transparency, and the coloring. As an example, a meteorologist might want to make regions having wind speed between 30 and 40 meters per second appear as red. If, however, the red-colored regions are semi-transparent, the total color (the one that is eventually mapped onto the screen) includes contributions from whatever lies behind these regions.

2.2 Output from StormView

The output from StormView is an *image object (IO)*. This is a matrix of RGBα-tuples, where R, G and B are intensities for the red, green and blue color-components, and α is opacity. The opacity is stored along with the color in order to be able to blend the image with a background image, such as a landscape or a grid reference frame. Figure 1 shows the result of blending two image objects with a background reference. The images shows renderings of the same data set (horizontal vindspeed). While the left one gives an iso-surface effect, the right one shows a smoother color variation.

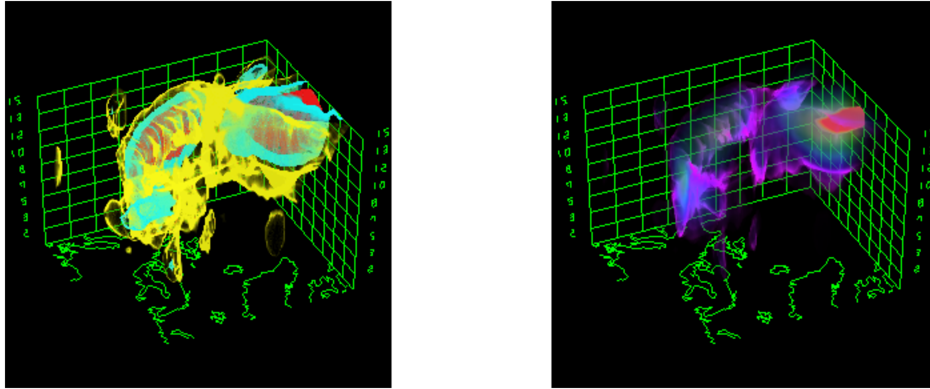

Figure 1: *Volume rendered horizontal wind speed.*

2.3 The Volume Rendering Algorithm

The volume rendering algorithm employed in StormView is the ray casting method described in [12]. To determine the color of a particular pixel, a ray is sent through it into 3D grid-space. Opacity and color is integrated numerically along the ray while it cuts its way through a sequence of grid-cells (see Figure 2). The integrands are evaluated on the basis of the mapping functions contained in the rendering-specification object and trilinear interpolation of the scalar values contained in the grid object. The integral is approximated using the composite trapezoid rule for each cell the ray passes through. Consequently, the exact positions where a ray enters and leaves a cell need to be calculated. The integration continues until either the accumulated opacity reaches a maximum value or the ray reaches the back side of the grid. At this point, the corresponding RGBα-tuple in the image object is calculated from the integrals.

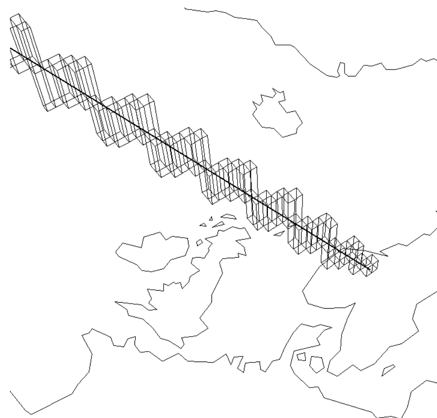

Figure 2: *A ray passing through a sequence of grid cells*

3 Parallelizing StormView

Response time is the resource that is optimized in our work. Hence, the main design goal of StormView has been to exploit the parallel system's potential for response time reduction. No particular efforts have been devoted to reduce the space optimizations.

3.1 A Sequential Approach

Consider implementing StormView as a single process rendering the volume in a strict sequential fashion. Table 1 summarizes wall-clock timing of the individual rays during rendering of a 121x97x18 horisontal wind speed field on a 57 MIPS / 17 MFLOPS HP-720 workstation.

	Rays hitting grid (35276)	Rays missing grid (4724)
Min	0.000741	0.000074
Max	1.587320	0.001451
Median	0.045469	0.000077
Mean	0.052072	0.000079
Std.dev	0.034094	0.000033

Table 1: *Sequential performance (in seconds).*

Rays that miss the grid requires considerably less computation than those that hit it (that is, rays within the 2D projection of the grid), so these two classes of rays are measured separately. The data in Table 1 indicate that if all rays passing through a 500x500 image hit the grid, the rendering may take more than 3.5 hours on a single workstation. This is too long for most practical use.

3.2 Introducing Parallelism

The parallelization of StormView is based on two decisions. First, we decided to decompose the problem at the pixel (ray) level. The smallest unit of work is the computation of the RGBα value for a single pixel. A *task* represents the computation of a set of pixels, given information to locate each.

Second, we decided that each participating process should have access to an entire grid object in its primary memory. Any process can now compute an arbitrary ray, although the total amount of memory devoted to the grid object will be high.

A consequence of these two decisions is that a process need not interact with any other process during its computation of a task. The result is the high computation to communication ratio necessary to exploit the parallel processing power inherent with high-perfomance computing nodes and low-performance network.

A convenient way to organize such a system is to have a set of *workers* perform ray casting in parallel, and a *controller* distribute the tasks and collect the results. Figure 3 illustrates the parallel architecture as well as the flow of data. Note, the graphical user interface process is separated from the rest of the system. The experiments presented in this paper are concerned solely with the interaction between the controller and the workers.

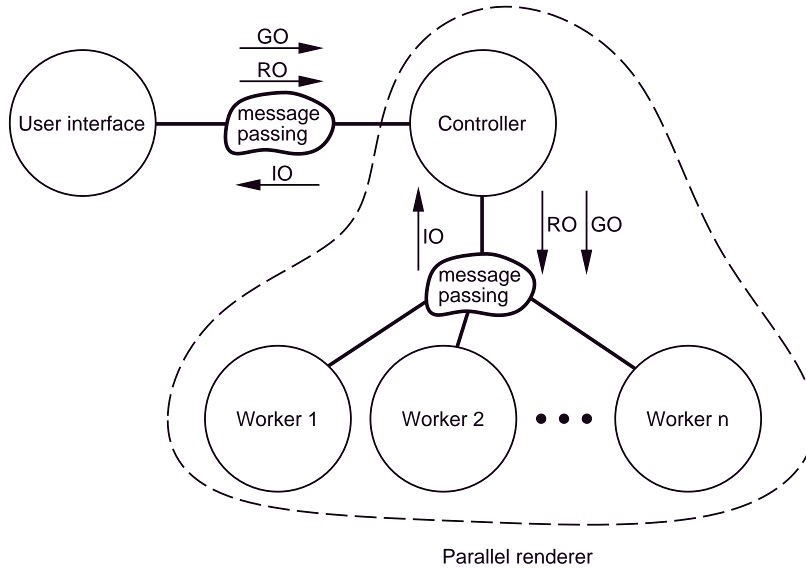

Figure 3: *Parallel architecture.*

4 Load Balancing

Load balancing is an important property of any efficient parallel algorithm. We have experimented with both static and dynamic load balancing schemes. In this section we describe the two schemes and the rationale behind them.

4.1 Static Task Distribution

0 In static task distribution (also known as *pre-scheduling*), the pixels that will be handled by each individual worker are decided in advance [9]. This division of responsibility is not altered once the computation starts. Each worker is assigned excactly one task per image.

The simplest approach to static task distribution is to partition the image into equally large rectangular regions. Worker 1 is responsible for region 1, worker 2 for region 2 and so on. Unfortunately, the method does not compensate for the variation in image complexity. Different amounts of work are required to render different regions of the image. This variation has several causes:

- the length of the part of the ray where work is being done may vary from zero to the length of the grid diagonal depending on:
  - **- opacity** - the ray may reach the visibility limit, which in turn depends on scalar values and mapping functions.
  - **- viewpoint** - the ray may pass through a narrow region of the grid, or even miss it completely.
- the number of iterations in the numerical integration depends on the complexity of the respective integrands, which in turn depends on scalar values and mapping functions.

Although the total effect of these factors is difficult to predict in advance, the data set should have a certain degree of continuity. Hence, there is a high probability that neighbouring rays require the same amount of work. We exploit this *image coherence* by making sure that each worker is assigned pixels from all over the image, a method often called *scattered decomposition* [10, 13]. For the rest of this paper, the term *static load balancing* will refer to the algorithm that employs scattered decomposition.

4.2 Dynamic Task Distribution

The static algorithm has one serious shortcoming. It ignores the variations in the individual worker efficiency. *Dynamic task distribution* adapts to this kind of variation. We chose the method of *demand driven 0computation*, also known as *self-scheduling* [3, 9]. Initially, the controller keeps a set of tasks corresponding to small subimages. Workers then issue task-requests to the controller, which, in turn, assigns tasks until the set is empty. The crux of this scheme is that a slow worker sends fewer requests and receives less work than a fast worker.

4.3 Tasks vs. Sub-images

In order to achieve good performance, one may need to consider the subdivision of the image into *sub-images*. A sub-image is a group of rendered pixels (RGBα-tuples) that a worker may transfer to the controller using a single call to the available send-primitive. Note that sub-images constitute the major part of the communication bandwidth in the algorithm. Therefore, it is useful to contemplate different ways of transfering this data from the workers to the controller. We want to test whether many small messages scattered over time will yield a better performance than a few large messages postponed to the end of the computation.

We have implemented two versions of the dynamic and the static algorithms. The two versions differ in that they either *scatter* or *concentrate* the transfer of sub-images from workers to the controller. We denote them D_S, S_S, D_C, and S_C respectively (D_S meaning *D*ynamic/*S*catter and so on). In D_S and S_C there is a one-to-one correspondence between tasks and sub-images. Once a complete task is computed, the corresponding sub-image is sent to the controller. D_C works as D_S except that in a worker, all the finished sub-images are accumulated into one single sub-image, which is transferred after all tasks have been computed. S_S differs from S_C in that a worker transfers several sub-images to the controller during the computation of the task. Figure 4 illustrates the four algorithms.

4.4 A Process Group Implementation

Our implementation uses the ISIS distributed toolkit v. 3.0.7 as platform [2]. This choice was motivated mainly by the location transparent naming of process groups, the presence of a reliable multicast primitive, and the ability to marshall messages in a convenient way.

Figures 5 and 6 show the pseudo code for the ISIS implementation of S_C and D_S. This code uses the (slightly renamed) ISIS-abstractions *mcast*, *reply*, and *thread*. The two first are primitives for multicast-communication between a process and a process group. Threads play an important part in the delivery of a multicast message. The ISIS runtime system delivers a message to the application program by instanciating and executing a thread with the message as parameter. Several threads may execute logically in parallel within one process. A thread executes non-preemptively until the control is explicitely handed over to the ISIS runtime

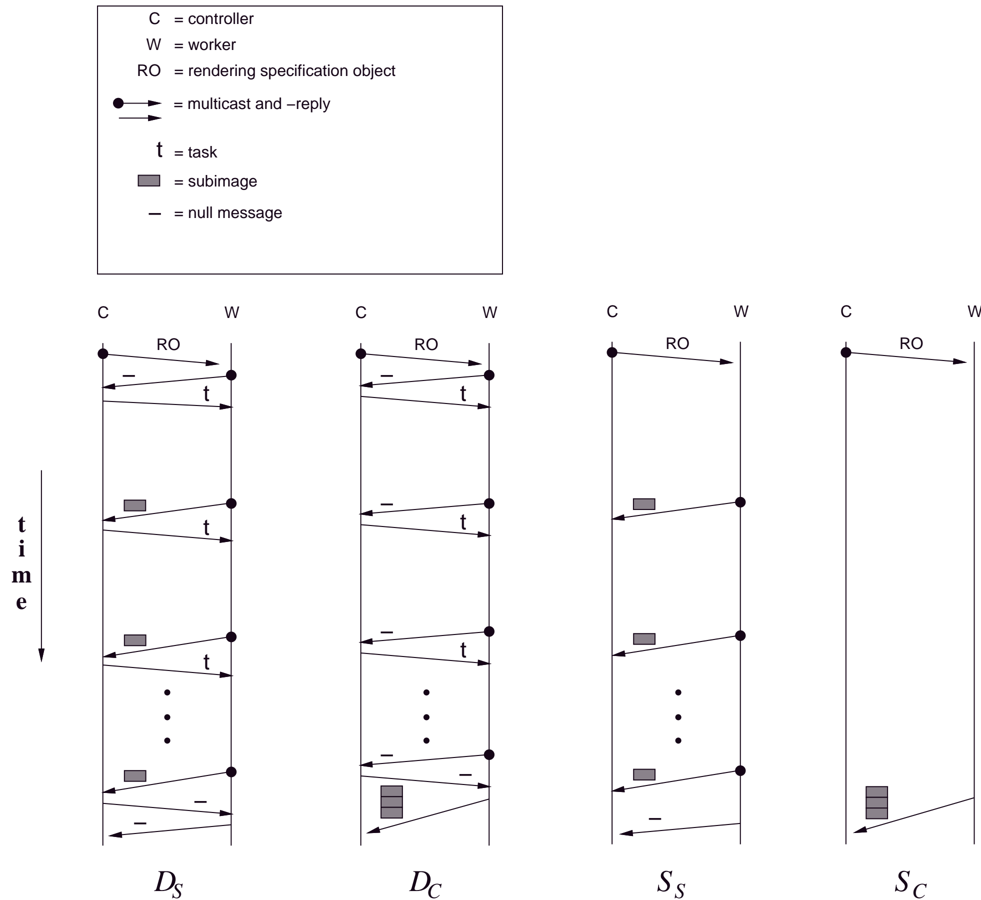

Figure 4: *The four load balancing algorithms (for simplicity, only one worker is shown).*

system. This occurs, for instance, at the termination of the thread, or in a call to a communication procedure.

For simplicity, we have not included code for transmission of the grid object. We assume that a current grid object is already allocated at each worker.

The two programs are both synchronous in the sense that every multicast waits for a reply. The static algorithm has a trivial communication structure involving only a single multicast from the controller to the workers. Note that the *render*-routine ensures that a given worker is assigned pixels scattered all across the image. The dynamic algorithm is more complex. Essentially, an initial multicast from the controller to the workers encloses a series of multicasts from the workers to the controller. The multicasts (actually unicasts) from the workers are a way of sending sub-images to the controller and new tasks to the workers. Note how this resembles the technique of piggybacking in network protocols.

Controller:

```
IO = mcast(RO, W);
```

Worker i:

```
thread(RO)
{
    reply(render(RO, i));
}
```

Operations and variables:

i	:	worker identifier
RO	:	rendering-specification object
IO	:	image object
W	:	group of workers
render(RO, i)	:	computes RGBα-tuples for pixels assigned to worker i with respect to RO
mcast(m, g)	:	multicasts message m to process group g
reply(m)	:	replies message m to the process issuing the multicast

Figure 5: *ISIS pseudo code for S_C (static task distribution with concentrated sub-image transfer).*

5 Performance Measurements

A set of experiments enabled us to compare the performances of the two load balancing algorithms and the two ways of transfering sub-images from the workers to the controller. We also measured the time for transfering a new grid object to the workers.

5.1 Experimental Environment

The experiments were run on a cluster of HP-720 workstations connected by a 10 Mbps Ethernet. Each workstation contains a 50 MHz PA-RISC 7100 CPU, 32 MB of RAM, and runs version 9.03 of HP-UX. The theoretical performance of such a workstation is 57 MIPS and 17 MFLOPS.

All experiments use the same input. The grid object represents horizontal wind speed from the LAM50S atmospheric model. The rendering-specification object includes a bird's-eye view point where all rays hit the grid and no ray traveled through more than 18 grid cells. Moreover, the light-source is turned off and the data set is rendered with a sharp iso-contour at a wind speed of 40 m/s. There are 54549 pixels in the image.

Task size is a critical factor in the performance of the dynamic algorithms. If load balancing was the only consideration, then task size should be as small as possible. Unfortunately, the smaller the task size, the more communication is required. We found 250 pixels to give a reasonable tradeoff between load balancing and communication. This is also a satisfiable size of a sub-image in the version of the static algorithm that scatters the sub-images over time (S_S).

Controller:

```
k = 0;
mcast(RO, W);

thread(I_h)
{
  if (h != -1)
      IO += I_h;

  j = k++;
  reply(T_j);
}
```

Worker i:

```
thread(RO)
{
   T_j = mcast(I_-1, C);

   while (j < q)
       T_j = mcast(render(RO, T_j), C);

   reply();
}
```

Operations and variables:

i	:	worker identifier
RO	:	rendering-specification object
IO	:	image object
W	:	group of workers
C	:	group of controller (a single-member group)
q	:	the number of tasks
T_j	:	task j
render(RO, T_j)	:	computes RGBα-tuples for pixels defined by task T_j with respect to rendering-specification object RO
mcast(m, g)	:	multicasts message m to process group g
reply(m)	:	replies message m to the process issuing the multicast
k	:	task identifier
h	:	subimage identifier
j	:	task and subimage identifier
r	:	the number of stop-signals sent

Figure 6: *ISIS pseudo code for D_S (dynamic task distribution with scattered sub-image transfer).*

5.2 A Scenario Favoring the Dynamic Algorithm

It is worth pointing out that there is one particular scenario in which the dynamic algorithm always outperforms the static one. This is when the speed of the individual workers differs substantially. When another user decides to run a computing intensive process at one of the worker nodes, the worker efficiency decreases. Clearly, this decrease is undesirable, since the the overall response time of the static algorithm depends on the slowest worker. When employing the dynamic algorithm under similar conditions, the loaded worker would be assigned a smaller amount of work, the rest of it being distributed fairly among the others.

5.3 Performance of Load Balancing

To evaluate the performance of the four versions of the load balancing algorithms, we measure total response time of each of the algorithms (a), with a different number of workers (n). In a single experiment, each combination of (a, n) is repeated 15 times. The following loop-structure is used:

for $sample := \{1, 2, \cdots, 15\}$
  for $n := \{15, 20, 25\}$
    for $a := \{D_S, D_C, S_C, S_S\}$
      $<$ run algorithm a with n workers $>$

Experiments were conducted at different hours. The results from one experiment during the day and one during the night are summarized in figures 7 and 8. These figures plots the 99.9 % confidence intervals for the mean of the total response time.

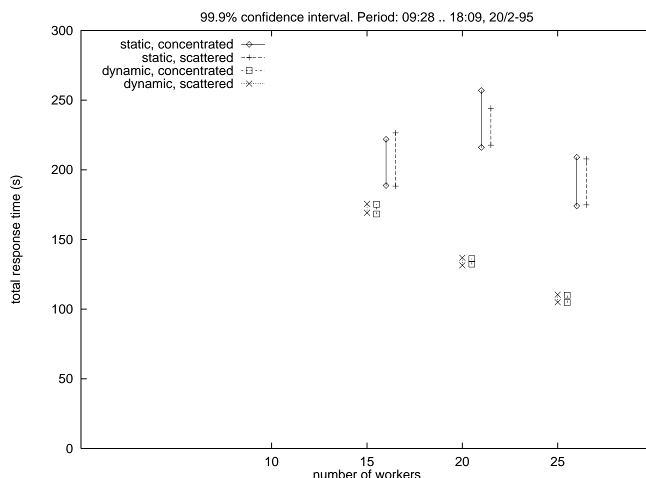

Figure 7: *99.9 % confidence intervals, day.*

From the measurements we make three major observations for our problem: First, the dynamic algorithms always perform better than the static ones. However, in certain situations, the algorithms exhibits a similar performance. Second, the dynamic algorithms are significantly more stable. Third, there is no significant difference between scattering and concentrating the transfer of sub-images.

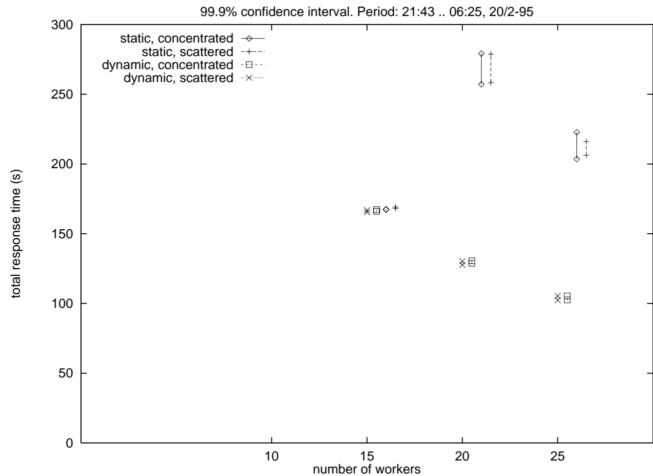

Figure 8: *99.9 % confidence intervals, night.*

5.4 Expense of Grid Distribution

The experiments presented so far are concerned with the rendering of a single image, assuming the grid object is always distributed to the workers before the computation begins. In certain situations the user needs to load new grids relatively often, such as in the rendering of a time series with a different grid per image. It is therefore worthwhile to measure the grid distribution time.

We measured this time in an experiment using the following loop structure:

for *sample* $:= \{1, 2, \cdots, 30\}$
    < distribute the grid to 27 workers >

The results from the experiment is presented in Figure 9.

6 Discussion

We now use the experimental results as basis for discussing load balancing method, sub-image transfer, and user pattern. Without extensive data collection it is impossible to give very confident statements about the load variation. We believe the experiments give reasonably representative indications of the underlying behaviour of the algorithms when executed in the current environment.

6.1 Load Balancing Method

The experiments clearly demonstrates the superiority of dynamic load balancing for this particular system and input. This result is primarily accounted for by the dynamic algorithm's ability of adapting to imbalance in both image complexity and worker efficiency.

Note also that even when the static and dynamic performance approaches each other, the dynamic algorithm is still better (see Figure 8). This suggests that the computation to communication ratio is sufficiently high for the extra synchronization overhead inherent in the dynamic algorithm to be negligible. As noted before, the size of the computation fraction is affected by viewpoint, mapping functions,

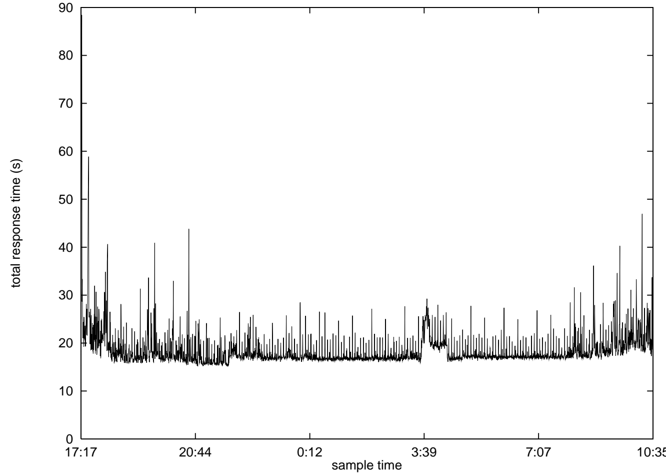

Figure 9: *Time profile, multicast of a 422532 bytes grid object to 27 workers.*

grid size and scalar values. However, situations in which the computation fraction is small also seem to yield less interesting images. The interesting images seem to include a combination of large semi-transparent regions (meaning less performance gain from early ray-termination) and sharp iso-contours (meaning more iterations in the numeric integration due to a more complex integrand).

From Figure 7 we make an important observation. Note how the response times of the two static algorithms make a jump upwards when the number of workers is increased from 15 to 20. This demonstrates how sensitive the static algorithms are to the inclusion of inefficient workers.

6.2 Method of Transfering Sub-images

The experiments reveals no difference between scattering and concentrating transfer of sub-images. This result is not surprising when comparing the order of magnitude of the total response time of the computation and the network bandwidth respectively. The network bandwidth is around 1 MB per second. Suppose that the computation of a 54549 pixel image is parallelized over 25 workers which finishes computation simultaneously. The worst possible scenario with respect to bandwidth exploitation arises when the transfer of sub-images is concentrated at the end. One RGBα-tuple is represented by 5 bytes, so this takes $(5 * 54649)/10^6 \approx 0.3$ seconds to transfer. Table 1 shows that the computation itself requires around $(0.05 * 54549)/25 \approx 109$ seconds. The transfer time is clearly negligible.

6.3 User Pattern

During a session, a user of StormView is repeatedly faced with two main choices: loading a new grid or rendering the current one. At one extreme, a single grid is loaded, followed by a sequence of renderings. This situation arises, for example, when the user wants to render a grid from different viewpoints and with different color- and transparency mappings. At the other extreme, a new grid is loaded before each rendering. This would be the case when rendering a time series.

The two extremes may be represented algorithmically as follows:

```
User pattern 1:        User pattern 2:

load grid              loop
loop                     load grid
  render grid            render grid
end loop               end loop
```

In the second user pattern, full replication of the grid object may have a bigger influence on the performance. As Figure 9 shows, the grid transfer time is not ignorable compared to the rendering time. To see how this situation might be improved, keep in mind that the current communication protocol (ISIS v. 3.0.7) implements multicast by invoking a send operation to each individual process in the destination group. Consequently, we could choose either or both of the following strategies:

- reduce the amount of data transferred in send operations to individual processes by abandoning full replication.
- reduce the number of send operations to individual processes by using a communication protocol that allows hardware multicast.

The first strategy might imply a reduced performance during the rendering phase. One possibility is to partition the grid into regions of responsibility and assign one region to each worker [6]. Another is caching [7]. These methods lead to extra communication and, possibly, unnecessary computation (for instance, it might be difficult to cut off rays early). On the second hand, our load balancing experiments shows that the computation fraction may still be high enough for these effects to be negligible.

7 Conclusion and Future Work

In a workstation environment, dynamic load balancing is the superior choice for ray casting volume rendering algorithms in which complete rays may be computed without the need for synchronization. This has two reasons. First, dynamic load balancing adapts better to the kind of worker efficiency imbalance often occuring in such an environment. Second, ray casting volume rendering is parallelizable at a sufficiently coarse grained level for the extra communication present in the dynamic algorithm to be negligible.

The possible gain of scattering the transfer of sub-images over time is negligible. This is also due to the high computation to communication ratio.

If the user pattern implies loading a new grid per every image to be rendered, it would be desirable to employ hardware multicast and also avoiding full replication of the grid.

Future work includes investigating how a faster network technology such as ATM (Asynchronous Transfer Mode) might influence the relative difference in performance between dynamic and static load balancing. Furthermore, it would be interesting to identify what issues of fault tolerance might be relevant for this type of application, and what performance tradeoffs will occur in a fault tolerant version.

Acknowledgements

We thank Tage Stabell Kulø and Fred B. Schneider for comments on early drafts of the paper.

References

[1] Jo Asplin. Distribuert Parallell Volumvisualisering av Skalarfelt fra en Atmosfæremodell. Master's thesis, Seksjon for Informatikk, Universitetet i Tromsø, March 1994.

[2] Kenneth P. Birman. The Process Group Approach to Reliable Distributed Computing. *Communications of the ACM*, 36(12):36–53, December 1993.

[3] Stuart Green. *Parallel Processing for Computer Graphics.* Pitman, 1991.

[4] Dag Johansen. StormCast: Yet Another Excercise in Distributed Computing. In F. Brazier and D. Johansen, editors, *Distributed Open Systems*, pages 152–174. IEEE Computer Society Press, 1993.

[5] Arie Kaufman. Introduction to Volume Visualization. In Arie Kaufman, editor, *Volume Visualization*, pages 1–18. IEEE Computer Society Press, 1991.

[6] Kwan-Liu Ma and James S. Painter. Parallel Volume Visualization on Workstations. *Computers and Graphics*, 17(1):31–37, 1993.

[7] Paul Mackerras and Brian Corrie. Exploiting Data Coherence to Improve Parallel Volume Rendering. *IEEE Parallel & Distributed Technology*, 2(2):8–16, 1994.

[8] C. Montani, R. Perego, and R. Scopigno. Parallel Rendering of Volumetric Data Sets on Distributed-Memory Architectures. *Concurrency: Practice and Experience*, 5(2):153–167, April 1993.

[9] Michael J. Quinn. *Designing Efficient Algorithms for Parallel Computers.* McGraw-Hill, 1987.

[10] John Salmon and Jeff Goldsmith. A Hypercube Ray-Tracer. In G. C. Fox, editor, *Proceedings of the Third Conference on Hypercube Computers and Applications*, 1988.

[11] Jaswinder Pal Singh et al. Parallel Visualization Algorithms: Performance and Architectural Implications. *IEEE Computer*, 27(7):45–55, July 1994.

[12] Craig Upson and Michael Keeler. V-BUFFER: Visible Volume Rendering. *Computer Graphics*, 22(4):59–64, August 1988.

[13] B. W. Weide. Analytical Models to Explain Anomalous Behaviour of Parallel Algorithms. In *Proceedings of the 1981 International Conference on Parallel Processing*, pages 183–187, New York, August 1981. IEEE.