REPORT

Computer Science Technical Report: **97-28** January 1997

On the Design and Performance of the PARFUM Parallel Fault Tolerant Volume Renderer

Jo Asplin and Sindre Mehus

INSTITUTE OF MATHEMATICAL AND PHYSICAL SCIENCES

Department of Computer Science

University of Tromsø, N-9037 TROMSØ, Norway, Telephone +47 77 64 40 41, Telefax +47 77 64 45 80

Abstract

Volume rendering is an important and CPU-intensive technique for visualizing large scalar fields. In essence, a volume renderer performs two activites on behalf of the user: loading a new data set, and rendering the current one. At one level, the performance of an individual activity is important. At another level, the erformance of the session as a whole, in particular switching from one activity to the next, should be addressed. In this paper we present PARFUM, a parallel volume renderer based on a controller/worker model in a network of workstations. PARFUM has three essential properties that increase the performance of a parallel volume renderer. First, dynamic load balancing is employed during a rendering activity. Second, workers may enter or fail without affecting the correctness of a session. Third, a user may easily abort the current activity in favour of a new one. These properties may more easily be achieved by accepting (rather than fighting) the inherent asynchrony in a distributed system. As a consequence, PARFUM attempts to minimize causal dependencies in the interaction between the user and the controller as well as between the controller and the workers. We evaluate two implementations based on the TCP and UDP transport protocols respectively.

Keywords: Parallel processing, fault tolerance, dynamic load balancing, transport protocols, volume rendering.

Contents

1 Introduction

Examples of large volumetric data sets can be found in a broad set of disciplines ranging from quantum chemistry to geophysics. Such data sets typically consist of in the order of 10^6 or 10^7 values, which makes visualizing them efficiently a serious challenge. Traditionally, only restricted parts of the data set (such as cut-planes or iso-surfaces) have been rendered in one image using surface rendering techniques. In the recent years, volume rendering has gained popularity as a complementary visualization technique. By sacrificing some precision, the entire volumetric data set may be visualized as a semi-transparent "fog" in one image. Due to heavy resource requirements, volume rendering is a natural target for parallel processing. A network of workstations forms an attractive platform for many parallel computations. In particular, for computations with a potential high computation-to-communication ratio, such as volume rendering, the comparatively low communication speed of such a platform may be insignificant. Two inherent properties of a network of workstations are *individual failure* (a single node failure won't necessarily bring the entire system down) and *asynchrony* (one cannot distinguish between a slow process and one that has failed).

Several authors have reported successful exploitation of a distributed computing environment for parallel volume rendering [6, 2, 3]. Fault tolerance and dynamic load balancing for distributed parallel computations in general is discussed in [1]. In this paper, we present PARFUM, a fault tolerant and dynamically load balanced parallel volume renderer executing in a distributed environment. PARFUM is highly flexible with respect to the failure and asynchrony properties of the underlying system. Any participant of the parallel computation may enter or leave the scene without explicitly invoking initialization or clean up protocols. PARFUM adresses the overall efficiency of a complete volume rendering session. In particular, the inherent asynchrony of the user is recognized. The user is allowed to change her mind at any time during a session. For example, she does not have to wait for an uninteresting rendering to complete before issuing a new one.

In Section 2, we present the main design of PARFUM, first describing how processes may join and leave the computation at any time, and second how dynamic load balancing and fault tolerance is achieved. In Section 3, we discuss how PARFUM may be implemented in either a TCP- or a UDP-style interaction paradigm. Performance experiments are presented in Section 4, after which the paper is concluded in Section 5.

2 The PARFUM Volume Renderer

2.1 Volume rendering

The PARFUM [1] volume renderer is based on a traditional raycasting algorithm for rendering large volumetric data sets [8, 5].

The input to PARFUM 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. 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

[1]PARallel Fault tolerant volUMe renderer

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) will include contributions from the scalar values behind these regions.

The output from PARFUM 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 blend the image with a background image, such as a landscape or a grid reference frame.

The basic raycasting function takes as input a grid- and rendering specification object, as well as a coordinate of the image. The corresponding RGBα-tuple is generated by sampling and accumulating color and opacity at evenly spaced points along the part of the ray that intersects the grid. Starting from the point closest to the image, this tracking continues until the ray leaves the grid or the accumulated opacity reaches a given limit.

A basic pre-condition for the raycasting function is that the entire grid is allocated in local memory. Hence, the grid has to be fully replicated in a parallel execution.

2.2 Controller/worker model

PARFUM adopts a traditional controller/worker model (also known as the master/slave model) for parallel execution. Typically, the controller executes on the local host, whereas the workers execute on a set of remote hosts (one worker per host). The controller keeps a set of tasks each of which may be computed by any worker. Since a ray may be computed in its entirety without the need for communication, the problem is subdivided at the image level. The controller partitions the image into equally sized tasks corresponding to subregions (groups of pixels). Having received the current grid, workers may then compute tasks in parallel. Communication occurs between a worker and the controller only, not between two workers.

2.3 A user session

From the user's point of view, a typical volume rendering session might look like this: A grid is given to the controller, which in turn distributes it to the workers. Then, a rendering of the current grid is initiated by giving the controller a rendering specification. As the computation progresses, the incoming results (subimages with all their RGBα-values computed) are assembled into a complete image. After this, the user computes another image from the current grid or loads a new one.

The normal (or expected) way to use PARFUM is to wait for an ongoing activity to complete before starting a new one. However, situations might occur in which a user would like to abort the current activity and substitute a new one. This is typically the case if a rendering specification is found to be inappropriate after a rendering is started. In this case, the controller and workers immediately forget the current activity and start the next one.

PARFUM can also be used in batch mode. That is, a user can write a script consisting of grid loading and rendering instructions which is given as input to the controller. This provides a convenient way to generate series of rendered images, e. g. time series.

Eager invitation

A design decision in PARFUM is that a worker does not have to know the location of the controller, but rather be executed on an arbitrary host among a set of registered hosts (hosts on which a worker may potentially execute).

This behavior is achieved by employing *eager invitation* for including new workers in the session. That is, at regular intervals, the controller takes the initiative to invite new workers to join the session. The price to pay for this method is that the controller needs to know the set of registered worker-hosts. Furthermore, the controller must spend time on unsuccessful invitations.

2.4 Dynamic load balancing

Load balancing is essential for any parallel computation. In the case of PARFUM, there are two phenomena which may lead to load imbalance: variation in image complexity (different regions of the image take different times to compute), and variation in worker efficiency (different workers exhibit different computing speeds). In general both variations are difficult to predict.

In order to keep all workers busy during the computation, we employ a traditional *demand-driven* or *self-scheduling* method of distributing tasks [4, 7]. Basically, as soon as a worker completes a task, it returns the result together with a new task-request. Obviously, slow workers sends fewer requests and hence are assigned less work than fast workers.

In general, the number of tasks is independent of the number of workers. Obviously, there should be more tasks than workers, but one has to be aware of the tradeoff involved. Many tasks implies much communication, which in turn may have a negative effect on parallel speedup. On the other hand, few tasks may inhibit the load balancing properties.

2.5 Fault tolerance

In a distributed environment where processes may fail independently, failure masking becomes a relevant issue. Consider for example a scenario where a user inputs a script of grid loading and rendering commands to the controller in the evening. The next morning the user expects to find the complete set of images on the disk. It would be desireable if the failure of single workers did not terminate the session as a whole. By adding fault tolerance to PARFUM, we effectively reduce the risk of such a session failure. Later we show that the performance penalty induced by this extra functionality is negligable.

In the context of PARFUM, we define a worker to *fail* if it halts or is permanently unable to communicate with the controller. Notice that this does not include e. g. Byzantine failures. Assuming an asynchrounous execution-environment, there is no way to distinguish between a slow and a failed worker. The fault tolerance property of PARFUM can be stated as follows:

> Let W denote the non-empty, finite set of workers that will be started at some point before or during a session. For a session to complete, it is sufficient that at least one worker does not fail (at least one worker will eventually be active). Hence, PARFUM masks $|W| - 1$ faulty workers.

Currently, PARFUM does not mask any kind of controller failure.

Eager scheduling

The fault tolerance property follows partly from a simple enhancement of demand-driven scheduling called *eager scheduling* [1]. An outstanding task (whose result has not been returned yet) may be assigned to other workers. By assumption, at least one worker will eventually receive the task, compute the result, and return it successfully to the controller. The fault tolerance also depends on the *eager* *invitation* mentioned earlier. By assumption, at least one worker will eventually be available to compute remaining tasks.

From the controller's point of view, tasks are partitioned into two groups: *finished* (results returned) and *unfinished* (no results returned). In order to reduce duplicate work, upon a task-request, the controller should hand out the unfinished task that has previously been handed out the least number of times. We implement this strategy in a simple and efficient way by keeping all unfinished tasks in a circular list. Upon a task-request, if the piggybacked result exists in the list (i.e. it has not been returned previously), it is removed. Then, the next task is handed out in a circular fashion. Notice that both list-operations are $O(1)$. Figure 1 shows how the list is changed after processing a task-request containing the first result of task 14. Notice that task 33 and 14 are handed out and removed respectively.

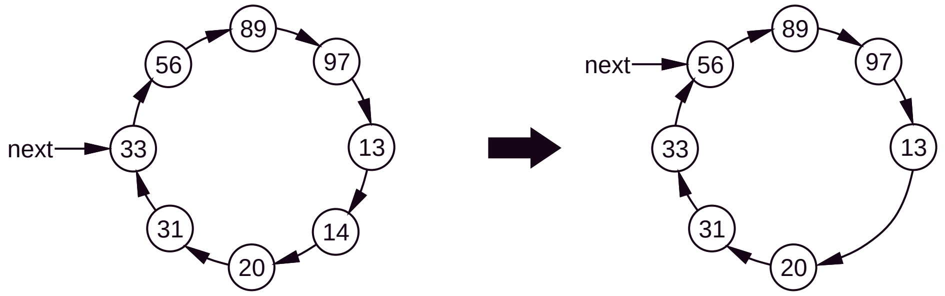

Figure 1: *The circular list of unfinished tasks*

3 Transport service

PARFUM has been implemented in two versions differing solely in the type of transport service employed. The first version uses TCP which offers a connection oriented, reliable stream service. The second is based on the connectionless, unreliable datagram service UDP. These transport services were chosen partly because of their widespread use and availability, and partly because of the possibility of executing a session across wide area networks such as the Internet.

3.1 TCP vs. UDP

In order to evaluate the use of TCP and UDP in the context of PARFUM, we now look more closely at some major differences between the two protocols.

Reliability A major advantage of TCP is reliability. The stream of bytes received is identical to the one that was sent. UDP only guarantees correctness of datagrams actually received.

Data transfer cost TCP transfers data by using an efficient flow control algorithm.

Application-level fragmentation When transfering large data sets through UDP, these have to be fragmented and re-assembled within the application itself due to the limited size of the datagrams.

Channel establishment cost In TCP the channel between the controller and a given worker is realized through a connection established by a potentially expensive three-way handshake protocol. In UDP, these channels exist implicitely and do not require explicit connections.

Resource allocation In contrast to UDP, TCP must allocate an amount of resources proportional to the number of channels. This might reduce the scalabiliy properties of the TCP-version. As an example, many operating systems impose an upper bound on the number of open socket-descriptors of a process.

Message abstraction Being datagram-based, UDP more naturally supports a communication pattern involving small messages.

The following table summarizes the relevant pros and cons of TCP and UDP:

	TCP	UDP
reliability	+	−
data transfer cost	+	−
application-level fragmentation	+	−
channel establishment cost	−	+
resource allocation	−	+
message abstraction	−	+

3.2 The TCP version

The controller loops forever waiting for one of three events to occur. First, the user may decide to load a new grid or render the current one. In that case, the controller immediately sends the grid or rendering specification to the currently connected workers. Second, a task-request may arrive. If it refers to the current grid and rendering specification, it is handled according to the eager scheduling method described in section 2.5. Otherwise it is ignored. Third, a timer goes off, forcing the controller to invite new workers as described in section 2.3. The current grid and rendering specification is sent to every new worker.

Any single *send* or *receive* operation returning an error is interpreted as a failure of the worker, and results in a disconnection.

The worker first accepts an invitation from the controller. Then, it loops forever prepared to receive a grid, a rendering specification, or a task. The reception of a rendering specification results in the worker sending its initial task-request. Upon receiving a task, the worker computes it and piggybacks the result on a new task-request.

3.3 The UDP version

In the UDP version, the controller has the same event-driven structure as the TCP version. Due to the unreliable, packet-based nature of UDP, grid-loading is handled differently. The controller divides the grid into fragments and leaves it to the workers to ask for their missing fragments employing the same demand-driven scheme used for distributing tasks.

In order to deal with the complexity of unreliability and asynchrony, it is convenient to model the worker as a finite state machine as illustrated in Figure 2. The worker can be in one of three states: idle, grid-loading, or rendering. State transitions are driven by messages from the controller. A work-request is used to notify the workers of the current grid and rendering specifications (if any). New grids and rendering specifications are assigned strictly increasing identifiers by the

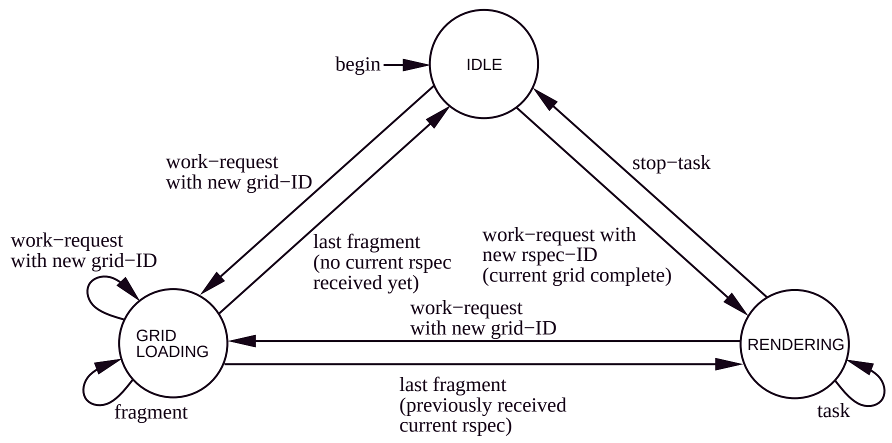
begin
IDLE
work−request with new grid−ID
stop−task
work−request with new grid−ID
last fragment (no current rspec received yet)
work−request with new rspec−ID (current grid complete)
work−request with new grid−ID
GRID LOADING
RENDERING
fragment
last fragment (previously received current rspec)
task

Figure 2: *Finite state machine of the worker*

controller. By including these identifiers in all messages, the controller and workers can easily filter out irrelevant messages, such as outdated ones.

4 Performance

It is well known that ray casting volume rendering with full grid replication has a high potential of speedup. This is due to flexible decomposition and a potentially high computation-to-communication ratio. We have conducted experiments with both the TCP and UDP version in order to evaluate how well PARFUM exploits this potential.

4.1 Description of experiments

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 the electron density of a AuH (gold hydride) molecule, and consists of $60 * 60 * 60 = 216,000$ scalar values.

The rendered image contains $300,000$ pixels and is shown in Figure 3.

Task size is a critical performance factor. If load balancing was the only consideration, then the task size should be as small as possible. Unfortunately, the smaller the task size, the more communication is required. We found 250 pixels to be a reasonable tradeoff.

We measured total response time at the controller with a varying number of workers, each combination repeated seven times. All experiments were run during the night when the overall system load was low. The TCP and UDP versions were tested on two seperate nights.

4.2 Results

Figure 4 shows the results of the experiments. Observe that the speedup is close to linear for both versions of the algorithm. From this, we conclude that no significant overhead is generated by neither the fault tolerance functionality nor specific

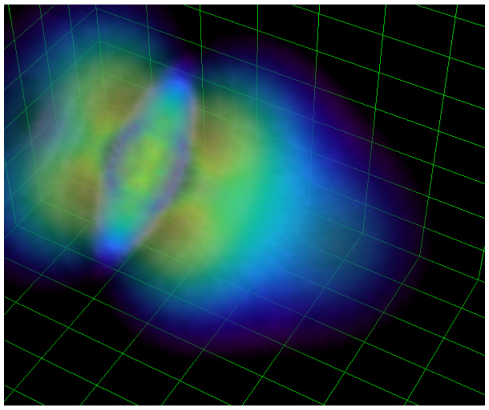

Figure 3: *The rendered image*

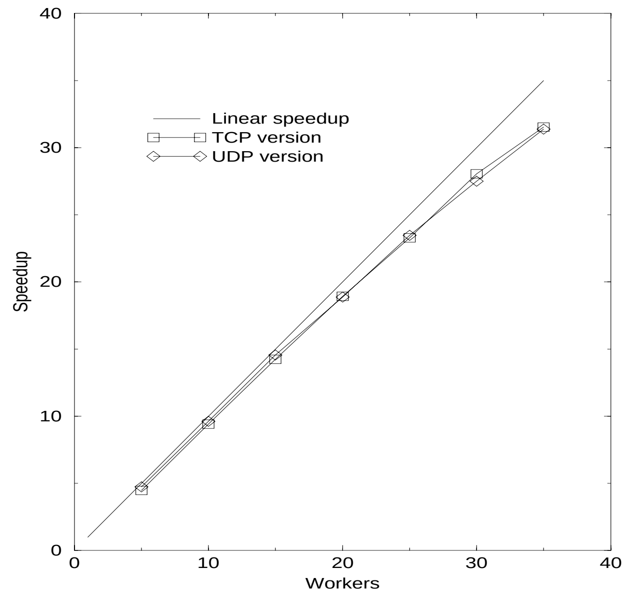

Figure 4: *Speedup of the TCP and UDP versions*

properties of the transport protocols. The low overhead of the fault tolerance functionality is due to the fact that all essential state is maintained by the controller which is assumed never to fail. Hence, no expensive disk operations or communication are required to back up the state. The state contained within a single worker at all times is minimized to a single result. If the worker fails, its current result will simply be re-computed by another worker at a later point.

5 Conclusion

We have presented the PARFUM, a parallel volume renderer based upon the controller/worker paradigm. PARFUM adresses the efficiency of a volume rendering session as a whole. The two most important properties are fault tolerance and dynamic load balancing. Under the assumption that the controller never fails, we have shown that fault tolerance does not induce any significant performance penalty. Furthermore, we have demonstrated that an implementation may employ either the TCP or UDP transport protocol. Neither induces significant overhead. From the end-users perspective, PARFUM is flexible to use. In particular, processes may enter and leave the computation in any order, and the current activity may be substituted by a new one at any time.

References

[1] Arash Baratloo et al. CALYPSO: A Novel Software System for Fault-Tolerant Parallel Processing on Distributed Platforms. In *Proceedings of the 4th IEEE International Symposium on High Performance Distributed Computing*, August 1995.

[2] T. Todd Elvins. Volume Rendering on a Distributed Memory Parallel Computer. In *Proceedings of the IEEE Visualization '92 Conference*, pages 93–98, 1992.

[3] Vinod Anupam et al. Distributed and Collaborative Visualization. *IEEE Computer*, 27(7):37–43, July 1994.

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

[5] Mark Levoy. Display of Surfaces from Volume Data. *Computer Graphics and Applications*, 8(5):29–37, May 1988.

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

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

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