Conference: PDPTA’04

Paper ID #: PDP2207

Fault-Tolerance of Parallel Volume Rendering on Cluster of PCs

SERGIO GUEDES
ELECTRONIC COMPUTING CENTER - NCE/UFRJ
FEDERAL UNIVERSITY OF RIO DE JANEIRO, BRAZIL

CRISTIANA BENTES
DEPARTMENT OF SYSTEMS ENGINEERING
STATE UNIVERSITY OF RIO DE JANEIRO, BRAZIL

GABRIEL PEREIRA DA SILVA
ELECTRONIC COMPUTING CENTER - NCE/UFRJ
FEDERAL UNIVERSITY OF RIO DE JANEIRO, BRAZIL

RICARDO FARIAS
COPPE/UFRJ
FEDERAL UNIVERSITY OF RIO DE JANEIRO, BRAZIL

Abstract. *In this paper we address a very important issue in parallel rendering systems, reliability. Distributed systems, such as clusters of PCs, are low-cost alternatives for running parallel rendering systems. However, distributed systems are usually not reliable, machines can fail during the rendering process, resulting in incomplete final images. Therefore, our goal is to take advantage of specific features of the parallel rendering applications, like tile-based computation, to include mechanisms to dynamically detect machine failure and automatically process tasks retrieval, with low overhead and no extra hardware. We developed three different parallel rendering systems, all based on the Parallel ZSweep algorithm[5], to provide fault-tolerance in different ways. Our experimental results show that the three systems present a small overhead to detect the failures, and when a failure occurs, the redistribution of the work does not degrade the system performance. We conclude that it is possible to provide fault-tolerance at low-cost in a cluster of PCs.*

1 Introduction

Volume Rendering of large-scale 3D data sets is a well-known computationally expensive problem. Even the most efficient rendering algorithm takes a substantial amount of time to run on a single processor. Out of the main approaches for undertaking this problem, parallel processing is widely recognized as the most effective one to provide real-time execution.

Several parallel volume rendering algorithms were proposed in the literature, e.g. [1, 7, 8, 13, 14], and they achieve quite good performance, running on parallel machines like SGI Power Challenge, IBM SP2, or SGI Origin 2000. These machines, however, are very expensive. On the other hand, the decreasing cost and high availability of commodity PCs and network technologies make cluster of PCs a low-cost alternative for running parallel rendering [16, 19, 15]. Cluster of PCs make parallel large-scale volume rendering a reality for scientists who cannot afford expensive parallel machines.

Nevertheless, such distributed systems are usually not reliable, which means that one or more processing nodes can fail during the rendering process. In a parallel rendering process, a node failure may damage the image, or even cause the whole process to halt. Therefore, parallel rendering systems strongly require additional mechanisms to detect and recover from node failures. In fact, guaranteeing the robustness of the parallel rendering is as important as guaranteeing its high performance, since there is no use for an efficient algorithm that either generates incorrect images or does not generate any result at all.

Traditionally, fault-tolerance has been provided through dedicated hardware [10], redundant software [11], or general and adaptive software infrastructure [9]. However, when we focus on a single type of fault, processing node failure, in an interactive parallel rendering environment, these traditional solutions are not well suited. Redundant hardware is quite expensive for a cluster of PCs. Replicating software services has impact on the performance of interactive rendering. Adaptive software infrastructures, like Chameleon, can provide multiple level of fault-tolerant execution of off-the-shelf applications, but, on the other hand, include a significant overhead when a fault occurs (from 20% to 60%).

So, we claim here that interactive parallel rendering on a cluster of PCs requires inexpensive and low-overhead fault-tolerance mechanisms. Our idea is to take advantage of the specific features of this application, like the tile-based segmentation of the computation, to include simple mechanisms to dynamically detect a machine failure and readily retrieve from it.

Particularly, we investigate the use of heuristic and heartbeat detection mechanisms and a very simple tile redistribution scheme. The use of a tile-based work distribution avoids the need of doing checkpoints, or restart the application to recover from a node failure. The main issue we are interested in is analyzing the overhead generated by these mechanisms. Having more information allows a more accurate failure detection mechanism, but, on the other hand, include extra messages on the network. Our evaluation of this overhead is based on the implementation of the fault-tolerance mechanisms on a real parallel rendering environment. We developed three different fault-tolerance parallel rendering systems on top of the Parallel ZSweep algorithm and evaluate their performance on a cluster of PCs.

Our experimental results show that the mechanisms implemented include only a small overhead to detect the failures, and when a failure occurs, the redistribution of the work does not degrade the system performance. From our studies, we conclude that it is possible to provide a fault-tolerance in a cluster of PCs at low cost.

The remainder of this paper is organized as follows. Section 2 describes the mechanisms we used to provide a fault-tolerant system. Section 3 presents the three fault-tolerant systems we implemented on top of Parallel ZSweep parallel rendering algorithm. In section 4 we present the results of our most important experiments. Finally, in section 5, we present our conclusions and proposals for future work.

2 Fault-Tolerant Parallel Rendering

Our fault-tolerance mechanisms are based on some specific characteristics of parallel rendering application, but they can be applied to a broad range of parallel rendering algorithms. Our mechanisms, however, were developed to treat only processing node failures. We define a processing node failure as the shutdown or crash of one of the cluster processing nodes, regardless of the cause (hardware reboot, software reboot or node freeze). Network faults are not considered in this first study. Therefore, we assume here that the network is reliable and a processing node failure does not disconnect any part of the network.

2.1 Parallel Rendering Application

Volume rendering is a visualization technique for interpretation of large amounts of 3D data, enabling the viewer to fully reveal the internal structure of the data. Commonly, the parallelization of volume rendering algorithms follows two different approaches: object-space parallel algorithms and image-space parallel algorithms. In object space algorithms, the parallel tasks are formed by the partition of the computation on the geometric description of the object. In image space algorithms, the tasks are formed by the partition of the computation of the 2D image space, which means that each task is responsible for computing a set of individual pixel values. Our fault-tolerance mechanisms were developed to the class of image space parallel algorithms, e.g. [3, 5, 15, 19], where the image plane is statically divided into small rectangular *tiles* that can be computed independently by the processing nodes of the parallel architecture.

2.2 Fault-tolerance Mechanisms

A fault-tolerant parallel rendering system guarantees that the new image is perfectly generated even if one or more nodes fail during the rendering process. This is accomplished by solving two main problems:

- How to detect a node failure;
- How to redistribute all work previously assigned to the faulting processor;

Because of the common tile-based work distribution of image-space parallel rendering algorithms, we can apply simple mechanisms to detect a node failure and to redistribute the work.

We use two different failure detection mechanisms: *heuristic detection* and *heartbeat* [12]. Heuristic detection assumes a node failure heuristically when the tiles assigned to that node are not computed after a Δt period of time. Heartbeat works like a heart pulsing. Each node sends periodically a message indicating that it is alive.

The redistribution of the work follows a single rule: all the tiles assigned to the faulting node are sent to another node. The dynamic strategies used to rebalance the load after this redistribution are out of the scope of this paper.

2.3 Implementation

We consider here two possible implementations of the detection and redistribution mechanisms:

- failure manager implementation
- distributed implementation

When there is a failure manager, it keeps track of the other nodes behavior, freeing the nodes from the task of detecting and recovering from failures. The centralization of the fault-tolerance activities in the manager simplifies the failure detection and recovering mechanisms. On the other hand, is not possible to provide fault-tolerance on the system when the failure manager itself fails. Therefore, the failure manager is useful on systems that have a server with special features like having no-breaks for which the probability of occurring a failure is considered minimum, almost zero. This ”special server” can run the failure manager.

2.3.1 Failure Manager Implementation

The failure manager gives each node a state: *running* or *broken*. At first, all nodes are *running* and the manager is aware of the distribution of the tiles among the nodes.

Heuristic Detection: the failure manager have the role of collecting the tiles computed from the nodes, so it waits for the tiles computed and establishes a time-out for this wait. If a node p does not respond after Δt time, the manager considers that p has failed and changes its state to *broken*. Obviously, this is not a precise failure detection mechanism, but, on the other hand, it avoids the communication overhead incurred on ping-pong messages between the manager and the nodes to monitor their work. In case of an incorrect failure detection, i.e. p is still computing the tile when its time-out ends, p answer will arrive at the manager later, and its state is restored to *running*.

Heartbeat Detection: each processor sends periodically a message to the manager indicating that it is alive. If the manager does not receive the heart-beat message from processor p after three intervals of time, it changes p state to *broken*.

Work Redistribution: when the manager is aware of node p failure, it verifies which tiles were sent to p, and send them to a *running* node.

2.3.2 Distributed Implementation

When there is no failure manager, the failure handling is distributed among the nodes. We did not use the heuristic detection mechanism in this implementation because, in this case, every node would have to receive a message for each tile computed, which would generate too much traffic in the network. So, the distributed implementation uses only the heartbeat detection mechanism.

Heartbeat Detection: each node sends its heartbeat messages to every other node, not only to a manager. This way, every node can detect a failure. When a node does not receive the heartbeat message from processor p after three intervals of time, it considers that p has failed.

Work Redistribution: when a node first detect a failure in p, it sends a failure message to the others. As the initial tile distribution is known by every node, and the processors ids represent a total order of processors, the processor with the less id number is elected to compute the tiles assigned to p.

3 Methodology

We evaluate the fault-tolerance mechanisms described developing three different fault-tolerant parallel rendering systems based on the Parallel ZSweep rendering algorithm [5]. We decided to use Parallel ZSweep in our experiments because it is a simple and very efficient parallel rendering algorithm.

Parallel ZSweep is a parallelization of the ZSweep algorithm [4]. ZSweep is based on sweeping the dataset with a plane parallel to the viewing plane, in the order of increasing z (i.e., front-to-back). It computes ray intersections with the faces of the cells, each time a vertex is found during a z-sweep, the faces incident on it are marked, and the ray intersections for the pixels that overlap with the faces are computed, and inserted on intersection lists. In order to avoid having the lists get arbitrarily large, ZSweep employs a scheme for early compositing.

The parallelization of ZSweep is based on breaking the screen into tiles and putting the tile ids on a *task queue*. Each tile represents a computational unit of work and, from a tile id, a processor knows which part of the image it has to render. So, there are two categories of parallel processes in Parallel ZSweep: the *master*; and the *slaves*. The master is responsible for managing the task queue; and the slaves are responsible for the rendering work which consists of applying ZSweep to the subset of the vertices which have faces projecting inside the tile.

Following the task queue programming model, the master creates the task queue, and continuously sends a tile id to an idle slave, until all the tiles have been rendered. Each idle slave that receives a tile id from the master, renders the subimage corresponding to that tile, and sends this subimage back to the master. The master, then, paste the tiles rendered by the slaves to form the final image. For more details on Parallel ZSweep, see [5].

On top of Parallel ZSweep algorithm we developed three fault-tolerant parallel rendering systems: *Heuristic Failure Detection* system (*HFD*), *HeartBeat Propagation* system (*HBP*), and *Master Reconfiguration* system (*MR*). The first two systems, HFD and HBP, use a failure manager, and MR provides failure handling on a distributed way.

3.1 Heuristic Failure Detection System

On our first system, HFD, the Parallel ZSweep master process assumes the failure manager tasks and uses the heuristic detection mechanism. The master distributes one tile for each *running* processor, and waits for the processor's answer (i.e., the message containing the subimage rendered) to send it another tile. If this wait time expires the time-out established, the master assumes that the node has failed.

The redistribution of the work sent to a broken processor involves putting the uncomputed tile back in the task queue. If processor p expires Δt computing t_j, this tile is moved to the end of the task queue. This means that t_j was not rendered and will be sent later to another processor. Moving t_j to the end of the task queue, postpones t_j rendering the most, giving a chance for p to finish its process (when p is actually still running). If there is a case of a high-density tile t_k that makes more than half of the nodes to be considered *failed* (because they could not compute t_k in Δ), the master stops resending this tile, restores the nodes state to *running*, doubles Δt, and waits for some node to finish t_k computation. This mechanism was implemented based on the assumption that the probability of more than half of the nodes fail in the same time interval is too low.

3.2 Heart-Beat Propagation System

Our second system, HBP, also considers the existence of a failure manager. The failure manager runs on the same node as the master process, but as another thread that implements the heartbeat detection mechanism.

When the manager is aware of processor p failure, it verifies which tiles were sent to p, and reinsert them back to the end of the task queue.

3.3 Master Reconfiguration System

The main difference between MR and the other two systems is that, in MR, the failure handling is distributed among the nodes, and to provide tolerance on master failures, the master can be reconfigured to run on any available processor.

The MR failure detection mechanism is also based on the distributed heartbeat mechanism. To implement this mechanism, we create two threads for the master and the slaves, one for the regular master and slaves computing and other to manage the heartbeat detection mechanism.

MR uses exactly the same implementation as HBP for the redistribution of the work sent to a failed processor, if the failed processor runs a slave process (i.e., the master acts as a failure manager and re-insert the work in the task queue). When the failure occurs on the master, however, the system has to be reconfigured to adopt another processor as the master. This is accomplished in the following way. First, the processors have to elect another master. The processor with the less id number is elected the new master. So, everybody knows who is the next master, and can send the new rendered tiles to it. The new master, however, cannot reconstruct the image, since it doesn't have the other rendered tiles sent to the old master. Instead of making the new master gather all the rendered tiles from the slaves, we implemented another strategy: *caching*. We decided to use the memory of each processing node as a cache for the rendered image (if the node fails, the redistribution mechanism will guarantee that the tiles in its cache will be recomputed by other node). The idea here is to have the image distributed over the network, avoiding the overhead of moving the tiles to the new master, when there is a master reconfiguration. Since each tile is generated once, and is not updated afterwards, there is no need of cache coherence mechanisms.

4 Performance Evaluation

4.1 Computing Environment

Our experimental environment consists of a cluster with 16 Processors Intel Pentium III 800Mhz. The nodes are connected by Fast Ethernet 100Mbits/sec cards. All the nodes run Linux kernel 2.4.20, and the communication is handled using MPI 1.2.5 [20].

4.2 Workload

We have used the SPX2 dataset in our experiments. It is tetrahedralized versions of the well-known NASA datasets. SPX2 is a tetrahedral unstructured grid, with 150K vertices and 830K cells. We generated images of 1024×1024 divided in 16-by-16 tile.

The execution times and overheads reported here correspond to a single rendering of SPX2 (0^o), which takes about 15s to run on 16 processors. For a multiple viewing processing, a single machine failure can invalidate or even halt all the remaining process, requiring a complete reexecution.

Table 1: Execution time of PZS, HFD, HBP and MR running under 16 processors without failures.

System	*Execution Time (sec)*	*Overhead*
PZS	12.28	–
HFD	12.37	0.74%
HBP	12.30	0.19%
MR	12.59	2.54%

The recovering mechanism we propose is well suited when larger datasets are used to generate bigger images. SPX2 is only a case of study, since what really matters is the overhead introduced by our methods.

4.3 Overhead without Failures

In our first experiment, we compare the execution time of our three systems with the execution time of the original Parallel ZSweep system (PZS), when no fault is generated. Our goal is to measure the overhead our failure detection mechanisms entail in the systems.

Table 1 shows the execution time of PZS, HFD, HBP and MR running under 16 processors. As we can observe in the table, the three systems perform almost the same when there is no fault, which means that the failure detection mechanisms practically do not generate overhead. The overhead increases less than 5% the overall execution time. For HFD, the heuristic detection mechanism does not include any overhead in detecting a failure (as it only assumes a time-out), but includes the overhead of resubmitting a tile even when there is no failure (i.e., the master incorrectly considered that a processor failed). For HBP and MR, we found that heartbeat mechanism does not provide a great disturb in the overall system performance. For MR, the disturb is bigger, but represents only at most 2.5% of increasing in the execution time.

4.4 Overhead with Failures

Our second experiment evaluates the overhead generated by our systems when one machine actually fails. To accomplish this, we injected a single fault in a random time interval in a random machine[*], and compared the execution time with the fault-free execution. This single fault was inserted artificially following the methodology adopted by other fault-tolerant systems [9] and the Mean Time To Failure (MTTF) previously reported on [17] where a hardware reboot takes about 2 weeks to happen and a process crash takes about 1 month.

Table 2 shows the execution times of HFD, HBP and MR running without and with one fault, and the overhead generated when the fault occurs. The results suggest that, the redistribution mechanism does not degrade the system performance. Note that, the increase in the execution

[*]In HFD and HBP evaluation, however, the master never fails

Table 2: Execution Times of HFD, HBP, and MR running under 16 processors with and without failures, and the overhead generated.

Execution(sec)	*HFD*	*HBP*	*MR*
fault-free	12.37	12.30	12.59
fault injection	15.93	14.53	16.01
overhead	28.7%	18.1%	27.16%

time for the fault injection execution is not only caused by the fault-tolerance mechanisms, because, after the fault occurs, the rendering proceeds with less nodes and, furthermore, the load imbalance increases. In terms of our three systems, HFD performs worse than the others, because of the useless resubmission of work.

5 Conclusions

In this paper we studied a very important issue in parallel rendering systems, reliability. We showed that a parallel rendering system can use simple mechanisms to dynamically detect and recover from failures with a quite low overhead, allowing fault-tolerant interactive volume rendering on a cluster of PCs. We developed three fault-tolerant parallel rendering systems based on Parallel ZSweep algorithm: Heuristic Failure Detection (HFD), Heart-Beat Propagation (HBP); and Master Reconfiguration (MR), that differ in terms of the detection and recovery mechanisms used.

Our experimental results showed that the three systems include a small overhead to detect the failures, and when a failure occurs, the redistribution of the work does not degrade the system performance. In terms of the amount of overhead generated, when a failure actually occurs, the overall execution time increases in the average only at 24%. Our results also showed that the heartbeat messages did not have much impact on the overall system performance.

We conclude that is possible to guarantee reliability of a parallel rendering system at a low cost. This reliability is essential to platforms like cluster of PCs or Grids [2, 6]. As a future work, we plan to implement the detection and redistribution mechanisms used here on other parallel rendering algorithms. We also plan to study detection and recovery mechanisms on the presence of network faults.

References

[1] J. Challinger. Scalable parallel volume raycasting for nonrectilinear computational grids. In *ACM SIGGRAPH Symposium on Parallel Rendering*, pages 81–88, November 1993.

[2] W. Cirne, and K. Marzullo. Open Grid: A User-Centric Approach for Grid Computing. In *13th Symposium on Computer Architecture and High Performance Computing*, September 2001.

[3] W Correa, J. Klosowski, C. Silva. Out-of-Core Sort-First Parallel Rendering for Cluster-Based Tiled Displays. In *Fourth Eurographics Workshop on Parallel Graphics and Visualization*, 2002.

[4] R. Farias, J. Mitchell, and C. Silva. ZSWEEP: An effi - cient and exact projection algorithm for unstructured volume rendering. In *2000 Volume Visualization Symposium*, pages 91–99. October 2000.

[5] R. Farias, and C. Silva. Parallelizing the ZSWEEP Algorithm for Distributed-Shared Memory Architectures. In *International Workshop on Volume Graphics*, pages 91–99. October 2001.

[6] I. Foster, C. Kesselman, and S. Tuecke. The Anatomy of the Grid - Enabling Scalable Virtual Organizations. *the International Journal of Supercomputing Applications* 15(3), 2001.

[7] C. Hofsetz and K.-L. Ma. Multi-threaded rendering unstructured-grid volume data on the sgi origin 2000. In *Third Eurographics Workshop on Parallel Graphics and Visualization*, 2000.

[8] L. Hong and A. Kaufman. Accelerated ray-casting for curvilinear volumes *IEEE Visualization '98*, pages 247–254, October 1998.

[9] Z. Kalbarczyk, S. Bagchi, K. Whisnant and R. Iyer. Chameleon: A Software Infrastructure for Adaptive Fault Tolerance. *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, n. 6, June 1999.

[10] H. Kopetz et al. Distributed Fault-Tolerant Real-Time Systems: The MARS Approach. *IEEE Micro*, vol. 9, n. 1, pages 25–40, 1989.

[11] M. Little and S. Shrivastava. Using Application Specifi c Knowledge for Confi guring Object Replicas. In *Proc. Third International Conference Confi gurable Distributed Systems*. May, 1996.

[12] E. Marcus and H. Stern. *Blueprints for High Availability*. John Wiley & Sons, Inc. New York, 2000.

[13] K.-L. Ma. Parallel volume ray-casting for unstructured-grid data on distributed-memory architectures. *IEEE Parallel Rendering Symposium*, pages 23–30, October 1995.

[14] K.-L. Ma and T. Crockett. A scalable parallel cell-projection volume rendering algorithm for three-dimensional unstructured data. *IEEE Parallel Rendering Symposium*, pages 95–104, November 1997.

[15] M. Meißner and T. Hüttner and W. Blochinger and A. Weber. Parallel Direct Volume Rendering on PC Networks. *Proc. of the Intl. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA '98)*. July, 1998.

[16] S. Muraki, E. Lum, K Ma, M. Ogata and X. Liu. A PC Cluster System for Simultaneous Interactive Volumetric Modeling and Visualization *2003 IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, October, 2003.

[17] K. Nagaraja, X. Li, B. Zhang, R. Bianchini, R. P. Martin and T. D. Nguyen. Using Fault Injection and Modeling to Evaluate the Performability of Cluster-Based Services *Proceedings of the Usenix Symposium on Internet Technologies and Systems* , March, 2003.

[18] J. Nieh and M. Levoy. Volume rendering on scalable shared-memory mimd architectures. In *1992 Workshop on Volume Visualization Proceedings*, pages 17–24, October 1992.

[19] R. Samanta, J. Zheng, T. Funkhouser, K. Li and J.P Singh. Load Balancing for Multi-Projector Rendering Systems. In *SIGGRAPH/Eurographics Workshop on Graphics Hardware, 1999.*

[20] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, J. Dongarra. MPI - The Complete Reference. *The MIT Press, Cambridge, Massachusetts, Lodon, england,* 1996.