A High Population, Fault Tolerant Parallel Raytracer

Abstract

Raytracing algorithms provide an ideal situation for parallelization, and are used widely in the movie industry to generate photorealistic two-dimensional images of three-dimensional artificial worlds. However, parallelizing any process can introduce the possibility of various types of failure and potentially decrease the overall performance if it does not take into account distribution overhead. In this paper, we present a hierarchical master-slave design pattern that allows for fault tolerance at the application level, avoids network partitioning, and reduces excess network communication congestion in the event of a failure. In addition, our design allows for scalability with minimal data redundancy and maximizes the utilization of each client involved in the raytracing rendering process. Our results show that our three-layer system can survive any type or number of client failures, and any non-concurrent server failures, while maintaining a near linear increase in performance with the addition of each new processing client.

1 DESCRIPTION

Raytracers are extremely computationally intensive and ideal for distribution, because each projected ray used to generate a pixel in the final image requires a set of independent calculations and the results from the calculations for one ray are almost never related to the calculations for a neighboring ray. Therefore, each pixel in the scene has the possibility of being composed completely in parallel, given that the scene data is shared among the computer hardware participating in the rendering calculation.

A simple implementation of a parallel raytracer would involve a single server managing the distribution of pixels of a scene, and a number of clients responsible for rendering some small portion of the scene. This is a typical application of the master-slave design pattern [7]. In order to effectively handle any large number of clients and to handle various failures in the network, we designed our raytracer in a tiered manner, with a master server, slave servers, and clients. The master server manages setup of the network and rough distribution of work to slave servers, which manage finer distribution to clients that perform the actual work of rendering pixels. This structure reduces network communication in the event of a failure and reduces the overhead involved in managing a very large number of rendering clients. In effect, our design reduces the “branching factor” of our network, and therefore reduces the load on each machine involved in the rendering process.

2 RELATED WORK

Raytracing is considered to be an “embarrassingly parallel” task, and therefore distributing the work of rendering pixels to more than one machine is not new to the field of computer graphics [3, 5, 6]. Our work focuses on the distribution of the rendering instead of the rendering itself; therefore the details of these previous systems are not relevant to this paper. The concept of a hierarchical master-slave design and its benefits to the field of distributed computing is also not new [2]. Kindberg, Sahiner, and Paker designed a hierarchical master-slave system using the Equus QoS manager system for general distributed applications [4]. However, this system employed unidirectional communication of processing requests down the hierarchy, and separate “collater” processes collected the output of all workers to compose the final results. In addition, there was no discussion of fault tolerance within the hierarchy. Aida, Natsume, and Futakata also demonstrated the performance increase and network overhead reduction when applying the hierarchical master-slave design pattern to solving an optimization problem with a parallel branch and bound algorithm [1]. Our contribution to this body of work is the design and implementation of application-level fault tolerance to the hierarchical master-slave design pattern. We present a raytracer that utilizes this design and demonstrate its suitability for high performance and high population, distributed applications.

3 DESIGN

Our parallel raytracer is based upon a variation of the master-slave design pattern. A single master server acts as a master to a number of slave server machines. Each of these slave servers, in turn, has a number of clients. The master server handles the drawing of the graphical user interface, ensures proper distribution of the scene data, separates the image to be rendered into batches of lines for each server, and is responsible for evenly distributing newly connected clients among slave servers in the distributed system. The slave servers, on the other hand, manage their own subset of clients and are responsible for a batch
of work to be spread amongst these clients. One of these slave servers acts as the next-of-kin, which takes over as master in the event that the master server fails. Finally, the rendering clients are the workers of the group. After receiving scene data from the master, they are assigned to a server and sequentially render single lines of the final image. Data is passed back to their respective slave server, which eventually returns it in a batch to the master for final composition.

Rendering of the graphical user interface consists of drawing a window containing all the currently rendered lines. The master is the only process that draws the scene to the screen, and only the master and the next-of-kin server have a copy of this rendered image data. This allows for sufficient redundancy in the event of a master server failure, without excess network overhead involved in sharing the image data with every process in the group.

The scene data for the raytracer is a collection of objects that describe geometric figures with varying locations, sizes and other properties in the scene to be rendered. The master server is responsible for sharing the scene data with all slave servers, who later pass it along to their respective clients. We hard-coded a complex set of objects into the scene generator and used the resulting scene object in all performance evaluations.

The master server is also responsible for separating up the image into portions to be sent to each slave server. We chose the finest granularity of image distribution to be lines, and the batches sent to each slave server to be groups of lines. The master keeps track of this distribution, and ensures that all lines are eventually rendered. If any slave server fails, the batch of lines assigned to the lost server is not rendered, and will be properly reassigned when all other lines have finished.

A client connecting to the raytracer initially connects to a master server, and is either assigned to be a slave server or is redirected to an existing slave server for rendering. The master manages the spawning of new slave servers and the distribution of clients to each of these slaves. This management process involves an initial client connection and a query for any available client spots from slave servers. If an open spot exists, then the client is redirected. Otherwise, the client is commanded to become a server and ready itself for accepting new client connections.

The master server assigns one of the slave servers as a next-of-kin. This next-of-kin will take over as master if the original master fails. Whenever the assignment of a next-of-kin takes place, all other servers are notified of this assignment and are aware of the network address of this failsafe master. This ensures that all slave servers are able to properly handle the death of a master server and switch over to a new master.

3.2 Server

Each slave server is responsible for managing its own group of clients, distribution of a batch of lines to be rendered, and assuming the next-of-kin role, if appropriate. Slave servers are in charge of clients that have been redirected to them by the master. If a client fails, then the slave server handles
the failure, ensures that line assigned to the lost client is properly reassigned to an active client, and continues rendering the batch of lines as normal.

Upon first connection to a slave server, clients receive scene data if they do not already have it. They are then continually assigned single lines for rendering. When the rendering of a group of lines has completed, the slave server sends a block of image data back to the master server, which is later displayed on the master server’s screen. If a slave server has been assigned as next-of-kin, it shares all the currently rendered image data with the original master. If the original master fails, then the next-of-kin server immediately becomes master and beings accepting connections.

3.3 Client

The client contains an instance of the engine that performs raytracing computations and renders a single line of the given scene at a time. The features of the raytracer engine used in the clients include various types of texturing on objects, reflection, refraction, Boolean operations between objects, and anti-aliasing. The scene object used for performance testing in this paper uses all of the above features of the engine. The recursive tracing and intersection algorithms used to render the scene are well-known and the specifics of the design of the engine are out of the scope of this paper [6].

4 IMPLEMENTATION

The raytracer is executed by initializing the master server and starting several client processes. Initially there are no slave server processes, and the master server determines the number of slave servers to instantiate. The master delegates the slave server role to the first connecting client; therefore clients must be able to become a slave server. Additionally, slave servers must be able to assume the role of master server if the master fails. All processes are started from the Driver class, which allows clients to assume the role of slave server and slave servers to assume the role of master server. Slave servers are unable to revert to the client role, which may eventually lead to an unbalanced client-to-server ratio. The driver will start a master server process if the master flag is specified. If the client flag is specified, then a client process is instantiated and connects to the given host.

4.1 Master

When the master process is started, it immediately constructs a server socket running on a predetermined port. A constant is used for the port, which results in a simple recovery process for master failures. In the event of a master failure, a slave server assumes that the next-of-kin will become the master and open a socket on the predetermined port. All processes initially connect to the master server and are assigned to their respective roles in the raytracer topology. A client is assigned a slave server role if all connected servers are full, which occurs when a predefined number of clients are connected to each slave server. Initially the project had intended to adjust this value dynamically to allow for scalability. However, experimental results concluded that a static value of 10 produces the best performance.

The master server must be able to accept client and slave server connections. A client connects to the master to establish the initial connection and disconnects from the master once it is assigned to a slave server or told to become a slave server. A client also connects to the master if the client’s server fails or the server redirects the client. When a client connects, the master is responsible for assigning the client to a slave server. The master iterates through the server list and queries each slave server for a client opening. If a slave server response specifies an opening, then the client is redirected to the slave server. If no slave servers have openings, then the client is assigned to become a server. The master server does not keep track of which servers have openings, because clients may be dropped at any time. This implementation introduces a possible bottleneck and requires unnecessary messages, but results in higher fault tolerance.

A slave server connects to the master server to establish the initial connection and is not disconnected until rendering has completed. A server may also become disconnected in the event of a failure, which results in the slave server connecting to a new master server. When a slave server connects, the master server adds the server to the list of known servers and spawns a new thread to manage socket communication. If this is the first connecting slave server, then it is assigned the role of next-of-kin. Additionally, the next-of-kin is sent a copy of the rendered image data. This redundancy allows the next-of-kin to continue rendering with minimal data loss in the event of a master server failure.

After connecting, the slave server sends a message to the master specifying if the slave server has the scene data. A slave server will already have the scene data if it was started as a client process and later redirected to a slave server role. The master responds with the host name of the next-of-kin and the scene data if requested. The server is sent batches of lines to render until rendering has completed. When the server finishes a batch of lines, it responds with the rendered image data. This data is forwarded from the master to the next-of-kin to allow for redundancy. When rendering completes, the master sends the slave server a message specifying there are no further lines to render. The slave server forwards the message to all of its clients then terminates.

In addition to creating the network topology, the master server is responsible for assigning batches of lines to render and synchronizing the rendered image data. The master server only keeps an index
into the image and a list of the completed lines, therefore the master server does not know which lines are currently being rendered. Each time a batch of lines is requested, the master server starts from the current index. The master server adds uncompleted lines to the batch until the batch is full, or all lines have been considered. This technique results in multiple servers assigned the same line to render, but this condition only occurs while rendering the final batch of lines. This may result in slave servers concurrently rendering the same line. This race condition can be ignored, because the servers will generate duplicate data. This method may result in computational redundancy, but is worth the tradeoff for greater fault tolerance.

4.2 Slave Server

After connecting to the master server, a slave server constructs a server socket running on a free port. The server socket only accepts client connections. When a client connects, the slave server checks for a client opening. If the slave server is currently full, then the client is redirected to the master server. Otherwise the client is added to the list of clients and a thread is spawned for socket communication with the client.

4.3 Fault Tolerance

Recovery from client failures is the simplest case, because there is at most one process communicating with the client. The client is in one of three possible states during a failure: connected to the master, connected to a slave server, or redirecting. No slave server or master server error handling is necessary if a client fails while in the redirecting state, because there are no open sockets to the client. If a client failure occurs while connected to the master server, the master server catches the exception and attempts to close the socket. If a client fails while connected to a slave server, the slave server catches the exception, attempts to close the socket, removes the client from the list of clients, and reassigns the line being rendered by the client. After a failure, a client will attempt to reconnect to the master and the next-of-kin. If both of these attempts fail, then the client will terminate.

Handling a slave server failure is more complicated than a client failure, because many clients may be connected to the slave server during the failure. When a slave server fails, the master server will catch the socket exception and remove the slave server from the list of servers. Additionally, each of the slave server’s clients will catch a socket exception and redirect to the master server. The master server will assign one of the clients the slave server role if there are not enough client slots. The remaining slave servers are unaffected by the failure, because they have no knowledge of the server. If a server assigned next-of-kin fails, then the master server must perform additional error handling. The master server picks the next slave server in the list of servers as the next-of-kin and informs the remaining slave servers. If there are no remaining servers, then the next slave server to connect is assigned the role of next-of-kin.

If a master server failure occurs, the topology of the network must be reconstructed. When a master server fails, the next-of-kin disconnects from its clients and assumes the role of master server. The remaining slave servers will detect the failure and attempt 5 connections to the new master server. Several attempts are required, because the server may detect the failure before the next-of-kin and the next-of-kin server takes time to transform to the role of master server. The disconnected clients will attempt to reconnect to the previous master server and then attempt to connect to the new master server. The clients connected to the remaining slave servers will be unaffected by the failure and will keep rendering while the slave server is connecting to the new master server. This implementation allows the raytracer to continue rendering during the error recovery phase.

4.4 Network Partitioning

The three-layer implementation allows the raytracer to effectively handle network partitioning of any single machine. The raytracer can handle partitioning, because every process can redirect to the next-of-kin in the event of a failure. If a client is partitioned from the network, then the system will handle the partitioning as a client failure. The client will attempt to redirect to the master and next-of-kin and terminate if it fails. If a slave server is partitioned from the network, an exception will be thrown once the slave server sends a message to the master server. The slave server catches the exception and disconnects from all clients, which will be redirected to the master. If the next-of-kin is partitioned, the master server will detect the failure and assign a new next-of-kin. The previous next-of-kin will assume the role of master server, but no servers or clients will connect. If the master server is partitioned, the result is the same as a master server failure.

5 RESULTS

The first experiment analyzed the scalability of the raytracer. Four different client configurations were tested and the results are displayed in figure 2. The raytracer was run with five clients per server and a batch size of 30. The graph displays a near linear speedup between the number of clients and run time. However, the increase in performance per additional client decreases after 30 clients. The raytracer should be able to scale to about 100 clients, adding additional clients would increase run time due to network overhead.

The approximate number of messages sent while rendering a 1000 by 1000 pixel image is displayed in figure 3. The number of messages was calculated as double the sum of the total number of
messages sent by the master server and the number of messages sent by each slave server to each client. This function was used to approximate total messages, because the servers communicate with the master server in a master-slave pattern and the clients communicate with the server in a master-server pattern. A non-linear relationship between clients and total messages resulted, due to the three-layer design.

Two additional experiments were performed that varied the number of clients per server and the number of lines per batch. The run times for different client to server ratios with a batch size of 30 are displayed in figure 4. A ratio of 10 clients per server produced the best results for the three-layer design. The results show that smaller batch sizes resulted in faster run times. This speedup occurs, because there is less redundancy while rendering the last portion of the image.

The fault tolerance of the system was tested by initially starting 19 processes, shutting down all except one of the processes, and restarting 6 more processes. A ratio of five clients per server was used; therefore the initial topology was a single master server, three slave servers, and fifteen client processes. Random processes were closed until a single machine remained, which assumed the role of master server. There was a minimum of fifteen seconds between each server failure. Clients were disconnected alone or in groups of two. The remaining process successfully assumed the role of master server and continued rendering from the previous master’s last image update. Six additional processes were instantiated, which formed a topology with a single master server, a single slave server, and five client processes. The raytracer successfully passed all fault tolerance tests.

6 CONCLUSIONS

The results of our work show that our hierarchical master-slave design is not only a feasible pattern for implementation, but effective in maintaining speed increases across large numbers of clients, limiting excess network congestion, and resisting group failure even when faced with many types of process losses. The tiered design distributes not only the individual task of rendering the scene, but also the management of this distribution. Since this client management is separated among each slave server, no single machine is overwhelmed by requests, and the total network overhead involved in a rendering operation is reduced. Finally, the multi-level design allows for an application-level approach to effective fault tolerance that is able to contain the occurrence of failures to a specific portion of the hierarchy and can therefore withstand any number of simultaneous client failures and any type of non-concurrent server failure.

We achieved a near linear increase in performance in our tests, demonstrating that the overhead involved with the fault tolerance and work distribution was minimal in comparison to the increase in overall rendering speed. Even with this initial implementation, we observe that the approximate ideal number of rendering clients in our particular system approaches 50 to 100, which would likely improve with optimization of the code, batch size, and server-to-client ratio. Our tests also revealed that a single-tier design, as simulated in our 20 clients per server test run, is markedly slower than our three-tier design in total rendering speed. Our initial assumption about the benefit of this design over the simpler arrangement of single server and multiple clients is therefore strongly supported. It can be predicted, from this data, that as a system like this grows larger and larger, it reaches a point where the
addition of a new tier of data management would greatly benefit the overall efficiency of rendering progress. However, a dynamically tiered system is left as potential future work, as described in the following section.

Overall, the design and its implementation were a success, and we achieved the increase in performance and resilience to failure that we expected during the onset of the project. Raytracing provides an ideal problem in the field of distributed computing, and demonstrates that a tiered design with failsafe mechanisms is an essential part of dealing with the massive numbers of clients and scene complexity associated with raytracing.

7 FUTURE WORK

This parallel raytracer focused on implementing a modified master-slave design that could scale to handle a large number of clients and withstand any type of non-concurrent server failure or simultaneous client failure in the system. Now that this pattern has been determined to be feasible and beneficial in the context of a parallel raytracer, there is much room for improvement in the areas of scene distribution, load balancing, and simultaneous server failure handling. If used in a professional setting, such as a visual effects studio, the scene data used by a raytracer can grow to be very large in size. Because this data must be spread among many servers and clients, it would be useful to implement a type of scene partitioning to spatially separate various objects in the scene and send only the parts needed by the server or clients. Server and clients could then request scene data on the fly and cache it as needed.

Load balancing also plays an important part in parallel raytracing. Due to the fact that certain rays in a scene may involve more complex calculations, it is useful to be able to take a complex section of image data and dynamically split it up among more clients than what is assigned by default. The current implementation of our raytracer performs very coarse, static load balancing, basing its image separation on lines. This may contain many pixels and never changes granularity based on previous performance metrics. While this simple division of pixels, in most cases, results in an even separation of work, there may be cases when particularly slow clients should receive smaller number of pixels.

In addition, future work could involve generating the network topology based on the rendering performance and network analysis of the machines used in the distribution system. This would allow for assignment of the optimum roles for each process in the group. For example, the master and next-of-kin should ideally be physically dislocated from each other, to avoid simultaneous failure. Network metrics could determine the most suitable computers for server roles, while the fastest machines would assume client roles. In addition, the number of tiers in the system could vary as the size of the total workload increases, reducing the workload on the master server. Adaptive master-slave work scheduling systems have been studied extensively in previous work [4, 8]. Dynamic topology would effectively reduce the branching factor of connections in the system. However, the complexity of this arrangement may create a larger opportunity for failures to occur in the midst of network rearrangement, and therefore present a challenge in maintaining the current level of fault tolerance.

The last and most difficult problem in our system is handling multiple server failures. While our system can handle an unlimited number of simultaneous client failures, it cannot handle simultaneous loss of both the master and the next-of-kin. One possible way to deal with this total loss of master servers would be to implement multicast messages, allowing all machines to elect a new master in the event of this type of leader-based failure.

8 REFERENCES


