A Distributed Graphics Library System

Frank Adelstein
Golden Richard III
Loren Schwiebert
Rick Parent
Mukesh Singhal

Department of Computer and Information Science
2036 Neil Avenue Mall
The Ohio State University
Columbus, OH 43210-1277

Phone: (614)292-4634
FAX: (614)292-2911

Abstract

We present a set of library routines that allow easily parallelized graphics rendering routines that require no communication between each parallel task, such as ray-tracing, to be run efficiently in an environment of distributed workstations. The presentation of the paper focuses on the problems encountered in implementing a distributed system under Unix and proposes solutions to each problem. Specifically, we discuss the challenges involved in overcoming the limits of communicating with a large number of processes in Unix and in providing fault tolerance when using sockets. Technical aspects of the implementation and some additional problems that were encountered are discussed. Finally, we compare the rendering times for a complex image with a renderer using the library and show that the library routines are able to exploit much of the existing parallelism. The library is presented using a graphics application, though the concepts are generic enough to be of use in designing any distributed system under Unix.

Key Words: Ray tracing, distributed computing, distributed graphics, fault tolerance, library routines.

Introduction

Motivations

Many applications in computer graphics, especially those for generating high quality images, require a tremendous amount of computing power. Ray tracing [1] is a good example of a technique where computation time and image quality are closely related. Generating high-quality ray-traced images can take days or weeks of CPU time on typical single processor workstations. Algorithms without data dependencies, such as ray tracing, can be easily parallelized since no communication or synchronization is required between regions of the image computed by different processors.

The primary computing environment in the Department of Computer and Information Science at The Ohio State University consists of over 200 Sun SLC workstations. There are over two dozen file servers, set up so that there is a consistent view of files from all machines. This kind of distributed environment makes available a tremendous amount of raw CPU power, well suited for rendering high quality images very quickly.

Objectives

The goal was the development of a set of generic library routines that could be used by existing renderers. In order to facilitate the incorporation of these routines into existing code, it was desirable to minimize the modifications and limitations the library placed on the renderer. Therefore, the library routines are responsible for acquiring remote machines, initializing all processes on those machines, handling all data communication and scheduling the remote processes.

Additionally, the routines must handle hardware and software failures cleanly. The probability of machine failure increases with the number of machines, so the lack of fault tolerance would severely limit the usefulness of the routines in even the most stable environment. It is desirable to limit the amount of work recomputed due to a failure. The failure of process should be determined quickly regardless of the cause, the overhead of failure detection should be minimized and the work should be reassigned in a way that maintains good utilization of the remaining processes. Successful resolution of these issues minimizes the effect process failure has on the execution time.

Finally, the library routines should provide good performance by exploiting as much parallelism as possible. This requires an effort to minimize the communication to computation ratio. There is also a need to divide the work equally between the processes and to provide scalability. Careful algorithm design and coding of the library routines minimize the overhead and improve the performance.

Rather than write a simplistic renderer to test the library, a publicly available, mature ray tracer was used. The renderer chosen was rayshade [2]. This simplified the debugging process and provided insights into adapting an existing program to use the library. Because the ray tracer is publicly available, there are also publicly available input data files, with known rendering times on a variety of platforms. This information allows the relative performance of the library to be measured.

Advantages

The main advantage of utilizing a large number of machines during image generation is the speedup. The speedup is not ideal, since there are inherently sequential parts [3] and there are synchronization and communication overheads. Also, the computation cannot be equally divided between the processors, since the time to compute a pixel varies.

High quality images can be generated in a single evening that would require weeks to generate on a single machine. Techniques such as anti-aliasing, complex lighting and complex scene layouts that would normally be avoided due to the high computing cost can be employed.

Rapid prototyping is also possible. A full size image can be quickly generated and used for object, light and camera layout. Unless physically modeled, the layout process is usually iterative, and low quality low resolution, or "preview" modes are used by the renderer. While this is often sufficient, it is useful to be able to generate higher quality images during this phase.

The use of distributed workstations is also cost effective. Using multiple workstations, an image can be generated in less time than using a single supercomputer, such as a Cray Y-MP. A supercomputer can be far more expensive than hundreds of workstations, so the distributed approach is a more cost effective solution for the type of applications under consideration.

More reliable generation of images is possible. For complex images, it is not uncommon for rendering to require several CPU days on a Sun SLC workstation. While downtime is minimal in a stable environment, it is unrealistic to expect machines to consistently remain up for a week or more at a time. In addition, while idle machines are available during non-peak times, it is difficult to reserve a machine for extended use or do interactive work when a CPU intensive application is running in the background.

Our system handles machine failure in a robust fashion. With the exception of the central machine on which the system starts, virtually all machines can fail and the system continues to run, reassigning the unfinished work to the remaining machines.

Related Work

Several other researchers have used distributed processors to accelerate ray tracing or other rendering techniques. In [4], Carter and Teague explore the design of a distributed ray tracer. In [5], the same authors consider how to distribute the data for the ray tracer to avoid storing the entire database at each processor. Meyers [6] discusses ray tracing on a network of Macintoshes. Other distributed ray tracing references can be found in [7]. The Condor system [8] provides fault-tolerant process execution in a Unix environment, but is intended for long-running, single-threaded applications. Linda [9] provides an attractive system for creating parallel applications in a variety of languages and on a number of different parallel computers. Although the implementation of distributed processing discussed in this paper is not unique, there is little discussion in the literature of how to overcome the inherent Unix limitations in order to support fault-tolerant, distributed applications. The focus of this paper is a discussion of these limitations and the presentation of possible solutions.

Technical Challenges

Process Limits

The most significant technical challenge encountered was the process limits imposed by Unix. Specifically, file descriptors per process and number of processes per user are both limited. A unique file descriptor is required for each child process in order to support interprocess communication. For example, if the limit is 50 file descriptors per process, a single process can communicate with at most 50 processes. This limits the number of machines that a naive version of the library could support.

The other limit is the number of processes a user can have on a machine at a single time. At least 300 clients can be specified on a Sun SparcStation 2 without problems. However, this could be a problem on slower machines or machines with less memory. A number of processes are required to initiate a process on a remote machine via the rsh(1) command, but they are not necessarily required for the duration of the process and can be released. Some mechanism is required to prevent slow machines from attempting to start too many processes at one time.

Network Timeouts

Another problem discovered after the initial implementation occurs when machines crash or are hard-booted. If a process is killed, the kernel tends to clean up and remote socket communication is terminated in a fashion that is detectable at the other end. However, no clean up is performed if the operating system crashes or a machine is stopped due to a hardware failure or human intervention. The TCP communication level does provide a method of periodically sending data to a socket to test if it is alive and to force a shutdown of the socket if no answer is received within the timeout period. However, the timeout period is not standard. Suns use two hours for their timeout, which was longer than desired.

RLE Format

Another problem encountered involved the run length encoding (RLE) format as implemented by the Utah Raster Toolkit [10]. RLE is a standard format for graphic images and what the rayshade raytracer produces as output. The problem is that for a given scanline, there is no guarantee that any output is produced. The RLE output routines wait for the next scanline before outputting the current scanline. If the next scanline is the same as the current one, both are not output; instead a single line is output with a count indicating this line is to be repeated. This can be generalized to runs of three or more identical scanlines. While this is an efficient method for compressing images, it becomes difficult to determine when the renderer has finished computing a scanline. It also requires the computation of scanlines in sequential order.

Library Implementation

Library Structure

The system is centralized, with all aspects coordinated by the machine that started the computation. As shown in Figure 1, there are five distinct components to the system. Each is a separate process that is responsible for a different aspect of the system. Note that these processes do not all run on the same machine. The master and all submasters run on a single machine; the slave processes usually run on different machines.

The library uses several input files. Since many command line options exist, default values can be specified in the default options file. The default values can be overridden with values specified on the command line. The available remote machines are specified in an input file referred to as the machine list file. The input to the renderer is referred to as the client's input file.

Master

The master process runs on the central machine. It is responsible for processing the command line options, the default options file, the machine list file and the client's input file. It initiates the submaster processes and passes the client's input file, the scanlines the submaster computes and the remote machine list to each submaster. Once all the submasters terminate, the master constructs the final image. The master handles any signals from the user by attempting to force termination of the submasters.

Any error in the master process is a fatal error for the system. There is little communication between the master and submaster level. Information is passed down to the submaster as parameters and passed up to the master by the exit code of the submaster process. The submaster's output is saved in a file with a unique name specified by the master.

Submaster

The submaster process is designed to handle the built-in Unix limitation on the number of file descriptors, typically about 50, a single process may have open concurrently. If fewer machines are used, then the submaster level is not necessary. Multiple submasters are needed to handle a larger number of machines.

The submaster sets up the socket and port for interprocess communication with the slave process. The actual communication occurs within the dialog subroutine described below. The submaster starts the slaves on the remote machines via the rsh(1) command

and accepts connections from the remote slave processes. Machines that do not connect in a predefined time period are not used. Results computed by the clients are written to the output file. The submaster insures clean termination of the remote clients.

Slave

A slave process is started on each remote machine that is used. The slave is a short-lived process that is responsible for connecting to its submaster and forking off a child process that runs the renderer. It then detaches from its controlling tty and sets its child's standard input, standard output and standard error file descriptors to the socket that is connected to its submaster. The slave also forks off another child that becomes the watchdog process described below.

Once everything is set up and the two child processes are running, the slave process exits. This terminates the rsh connection to the submaster and allows approximately three processes and three file descriptors on the central machine to be released. From this point on, any diagnostic information must be sent via the message protocol described later, because any normal output, such as from a printf() statement, is misinterpreted as a message header by the submaster. The reception of an illegal message header is considered a fatal error by the system. All of the information the slave needs for establishing a connection, such as the hostname and socket port, as well as the name of the renderer and its command line arguments are sent via rsh(1) command line arguments.

The slave process has several options that allow the user to dynamically limit which machines the library uses. The load limit option causes the slave to check the current load on the machine via the uptime(1) command and to not run if the load is above the specified limit. The console priority option causes the slave to check the ownership of the machines console. It runs only if it is owned by the user who started the master process or by root, which implies no one is logged in. Note that this option does not check for users that are remotely logged in. The nice option causes the slave process to invoke the nice(3) system call, which reduces the CPU priority of the renderer and watchdog processes.

Renderer

The renderer is started by the slave process. There is one renderer started for each entry in the original machine list file. Multiple entries of the same machine result in multiple renderers started on that machine. Standard input, standard output and standard error are all directed transparently to the submaster via sockets, so it looks like normal I/O to the renderer. If there is a problem on the central machine and its submaster dies, then the renderer dies with a broken pipe signal very quickly.

Watchdog

The watchdog process was implemented once it was discovered that a socket connection timeout cannot be detected in a timely, consistent fashion. A socket option called **SO_KEEPALIVE** is built into the TCP communication level, but it is implemented differently on every platform. On Suns, although it is documented that dead connections time out after five minutes, the timeout is actually two hours. BSD Unix is set for eight minutes [11].

Approximately every minute, the watchdog process checks if the renderer process still exists and sends a message to the submaster. If the renderer no longer exists, then the watchdog process terminates. The renderer's process ID (PID) and the submaster host and port are passed to the watchdog as command line arguments. The PID is used to track the renderer; the host and port are used to connect to the submaster.

Using the Library

The distributed graphics library is designed so that a programmer can take a simple stub program and link in the object module to produce the distributed renderer. The stub program has four callback routines that can be registered, similar to the X Window System's Athena Widgets [12]. The callbacks are for: initialization, data reading, scanline rendering and termination routines.

Minor changes to the layout of the renderer may be required. The renderer *must* be split into the four previously mentioned parts, each a separately callable function. The initialization and exit routines, either or both of which can be omitted if not needed, take no arguments. The data reading and scanline rendering routines are passed a single argument, which represents the size of the data being sent, as the first message. The remaining messages can be read by calling the provided library routine.

For the data reading routine, the next message provides the command line arguments. Following this message is the input data for the renderer. This is **not** in the message format and is intended to be read as if it were originating from standard input. The parser routine must recognize that the end of the data is delimited by a special token, which is automatically added by the library when the data is transmitted to the renderer. For the scanline rendering routine, the next message contains the beginning and ending scanline range.

The callback routines return zero upon successful execution. A non-zero return value indicates an error. The stub program automatically sends an appropriate message back to the submaster indicating success or failure. If no callback routine is registered, the stub program automatically sends a success message.

Dialog Subroutine

The dialog subroutine is the main loop of the submaster process. It is essentially a state machine, tracking the states of the remote processes. State transitions occur when messages arrive from the remote processes. The next state is determined by the current state and the type of message. Actions that need to be performed are handled here, such as sending the data or scanline assignments.

The dialog routine handles the assignment of scanlines to the processors. This is done by maintaining a queue of idle processors and a queue of unassigned scanlines. Additionally, there is a third queue containing the completed scanlines. The dialog routine assigns a range of scanlines from the scanline queue to each idle processor. The size of the range is determined by runtime parameters.

The dialog routine also periodically calls a timeout routine and terminates the connection of any process that has not responded recently. The unfinished portion of the scanline range assigned to that client is returned to the scanline queue.

Messages and Formats

There are eight different message types currently defined, three for the client and five for the submaster. The data structure for the message, called PacketHead, consists of the message type and a field for the size of the data segment. The data segment is a free form field used by some of the messages. The value of the size field is zero when the data segment is unused. The message types are defined as follows:

```
typedef enum messages_t {
  /* client messages */
  Ack,                          /* generic positive acknowledgement */
  Nack,                         /* generic negative acknowledgement */
  Result,                       /* data that's returned by client   */
  /* submaster messages */
  Hello,                        /* establish connection             */
  Bye,                          /* end connection                   */
  Data,                         /* passing in arbitrary data        */
  Run,                          /* run program, given parameters    */
  Noop,                         /* no operation (ignored by client) */
  } Messages;

typedef struct packet_t {
  Messages  type;
  int       size;
} PacketHead;
```

The client can send the following messages: **Ack**, **Nack**, and **Result**. The **Result** and **Nack** messages use the optional data field. For the **Result** message, this field contains the raw pixel data for the scanline that the client has just computed. For the **Nack** message, this field contains the text of the error message that the client generated. The **Ack** message is used to indicate the success of operations. Socket failures are treated as **Nack** receptions. A library routine is provided to send the result message.

The following messages are defined for the submaster: **Hello**, **Bye**, **Data**, **Run** and **Noop**. The **Noop** message does nothing and was used during early communication testing phases. The **Hello** message tells the client to run its initialization code. The **Bye** message tells the client to run its termination code.

The **Data** message tells the client to run its data reading routine. The submaster then sends the data directly to the client through the socket. To the client, it appears that the data is coming from standard input. The reason that the submaster sends the data is twofold. First, two hundred or so machines all simultaneously accessing the same data file can put a heavy load on the file server. Second, there may be unsupported features specified in the input file which need to be removed. This is done by preprocessing the file once by the master before passing it to the submasters.

The **Run** message tells the client to run the scanline rendering routine. When a scanline is done, the client sends back one or more **Result** messages and the submaster updates the client's assignment field. The client sends an **Ack** when it finishes its assigned scanline range, indicating that it is now ready to be assigned more scanlines.

The Submaster's State Table

There is an entry in a state table for every remote client that the submaster is able to successfully start. The table is created when the remote clients connect to the submaster's well-known socket. The submaster waits until either all of its remote clients have connected or a timeout has occurred. The timeout is currently set to 20 seconds after the last connection occurred. Each entry of the state table is defined as follows:

```
typedef struct sockarray_t {
  int socket;                   /* socket connection to the client    */
  long lastalive;               /* last time it sent an alive msg.    */
  client_state status;          /* current state of the client        */
  struct sockaddr_in saddr;     /* address of client                  */
  int assignment;               /* current scanline being rendered    */
  int firstline;                /* first line in current assignment   */
  int lastline;                 /* last line in current assignment    */
} Sockarray;
```

The possible states of the client are the following: *Created*, *InitWait*, *DataWait*, *Ready*, *Running*, *Answering*, *Failed*. The state transition diagram is shown in Figure 2.

The *Created* state is the initial state of the process. A process is in the *InitWait* state from the time a **Hello** message is received until an **Ack** is sent. Once an **Ack** is received by the submaster, the data is transmitted and the process enters the *DataWait* state until the process sends an **Ack**. Once an **Ack** is received by the submaster, the process is placed in the *Ready* state. A process is put on the Ready queue when it enters the *Ready* state.

Processes that have been assigned work move into the *Running* state and are removed from the Ready queue. A running process that sends a **Result** message is in the *Answering* state. It may send multiple **Result** messages, depending on the number of scanlines that it is assigned. When a process that is in the *Answering* state sends an **Ack**, it is placed back in the *Ready* state and is moved back onto the Ready queue.

A process that sends a **Nack** enters the *Failed* state. If it is in the Ready queue, it is removed. Whatever assigned work it has not finished is returned to the scanline queue. Once a process enters the *Failed* state, it remains there.

If an illegal message type is received, it is considered a fatal error, and the entire system terminates after attempting to shut down all of the remote clients by sending **Bye** messages to them.

Statistics Logging and Debugging Information

There are several different debugging levels available, determined by a runtime parameter. Higher levels produce more verbose information on the progress of the program, such as clients used, scanlines assigned, failed clients and messages received.

In addition, each submaster creates a file called stats.out.XXX, where XXX is the submaster's index number. Each time a message is sent by the submaster or is received from a client, an entry is added to the statistics log. Each entry consists of the message type, the client ID, the old state of the client and the time it occurred. Scripts are used to convert the information in these logs into a more human readable format and to report on the clients that currently have assignments.

Meeting the Goals

Limiting Modifications Needed To Use Library

The master requires an interface to the renderer which conforms to the specifications given below. A custom input parser must be written for the master, using the function **read_client_input()**, which modifies the client's input file to support the distributed implementation. The function must also parse out information, such as the screen resolution, that is needed by the master program. Since this routine serves as an interface between the library and the renderer, the routine must be written for each renderer.

The renderer must satisfy the following five constraints to use the library:

- *four separate functions* The renderer must be divided into four segments, each callable as a separate function. The segments are: initialization, data reading, scanline rendering and termination. If the renderer requires no initialization or termination code, a null function can be used.

- *output raw pixel data to stdout* The renderer must be able to generate raw pixel data, as opposed to RLE format, and write it to standard output.

- *read data from stdin* The renderer must be configurable to accept all data from standard input.

- *no diagnostics to stdout or stderr* The renderer must not print any messages to standard error or standard output. Routines are provided in the library which allow a failure message to be sent to the master for display.

- *have some explicit end-of-input token in the grammar* The renderer's input language must include a token to mark the end of input. The library uses the literal *endfile*. This token should cause the renderer to terminate parsing.

Some users might find the constraint on renderer diagnostics too restrictive. In the current implementation stdout and stderr share the same socket. An alternative would be to provide a separate socket for stderr. This would allow diagnostic and informational messages to be separated from the output data. An additional socket would be required for each client, which further limits the number of clients per submaster. However, if the user does not need to distinguish which client sends a message, then all clients could share a single stderr socket. This method requires that the client be modified to send all diagnostic and informational messages to stderr.

While the library requires the literal *endfile*, a minor change would allow it to be user-specified. This constraint disallows the use of an actual end of file to terminate the input because this would also close stdin. Stdin is used to send messages to the renderer and therefore must remain open for the duration of the renderer's execution. Providing a separate channel for the input file would require another socket for each client.

Process Failure

One of the earliest design decisions in this project was that it should be able to handle process failures due to either internal errors or external intervention. To support fault tolerance, every remote process has a state associated with it in the submaster. When a process dies or is killed, its socket connection is terminated. The socket appears to be readable by the select(2) system call, however, any attempt to read from that socket results in zero bytes being read. Any read that returns zero bytes causes that process to be placed in the *Failed* state.

In addition, every process that is currently working on a scanline has the scanline range recorded by the submaster. This is updated every time a scanline is returned. Upon failure, the unfinished scanlines are returned to the scanline queue. The process is removed from use and the lost work is limited to the scanline being computed at the time of failure, which is then recomputed by another process.

Distributing Workload In An Efficient Manner

Each submaster is a separate process that does not communicate with the other submasters. This was done both for simplicity of design and implementation and to minimize communication overhead. Once work has been assigned to a submaster, only that submaster's remote processes can do that work. Since scanlines cannot be dynamically moved to another submaster, situations can arise where one submaster has many scanlines left to compute while one or more submasters are completely idle. Partition problems tend to be intractable by nature, so two different heuristics for dividing the scanlines among the submasters are used. Another option is to use a single submaster, however, this limits the number of machines that can be used to the number of file descriptors available to a single process.

The first technique divides the screen into equal parts and assigns one part to each submaster. A submaster could then assign a range of scanlines to each process, which is useful for algorithms that make use of scanline coherence. However, it may not be an optimal way to divide the computation. In general, "interesting" areas of a picture tend to be centered, while background or less computationally intense parts tend to be near the ends. The submasters that get the ends usually finish long before the submasters that get the regions near the center.

An alternative method was devised to achieve a better load balance. The screen is divided such that submaster n_i, of N submasters, computes the following scanlines:

$$n_i \equiv scanline \ (mod \ N)$$

In other words, each submaster is assigned every Nth line of the screen offset by its submaster number. This causes the submasters to get scanlines from the entire picture, which tends to more evenly distribute the workload.

Results

Figure 3 shows the time of runs with a varying number of machines on the same image. Figure 4 shows the same data without the single machine case. The datafile used for the test runs is aq17.ray [13]. The image resolution used for the tests was 1000×1000 pixels. Only machines with low loads at runtime were used in an effort to obtain valid timing results. However, there is no way to restrict other users from using those machines once execution has begun. This caused the discrepancies in the data where the execution time increases with more machines.

Ideally, the time to render an image should decrease linearly with the number of machines used. However, there is an overhead cost associated with starting up each process and this cost grows linearly with respect to the number of machines used. Figures 3 and 4 clearly show a substantial, although not linear, decrease in the execution time as the number of machines used increases. This demonstrates the inherent parallelism of the application and the efficiency of the implementation.

The most dramatic decrease in execution time occurred when the number of machines was small. As machines are added there is less work for each machine, so the incremental improvement is less. As expected, once a large number of machines are utilized, such as on the order of 150, additional machines cause only a slight reduction in execution time.

For the display devices used, only about 1000 scanlines can be shown. If 200 machines are used, each machine computes an average of five scanlines. The theoretical maximum number of machines that could be used is about 1000, since any additional machines would be idle. However, the overhead of adding new machines eventually exceeds the benefit gained by adding them. The complexity of the scene determines where this point occurs with longer computation times allowing the use of more machines. For the picture used, this point probably occurs at around 200 machines.

Conclusion

A fault tolerant library for implementing parallel applications with no data dependencies in a distributed workstation environment has been presented. The library includes features that reduce the impact the distributed computations have on other users as well as handling machine failures. The structure of the implementation, including the process hierarchy and communication protocol, has been described. Major technical challenges were described along with solutions. Test results show that the implementation provides a significant speedup by utilizing multiple machines. Although the library is presented using a graphics program, non-graphical applications could also be supported.

References

[1] Turner Whitted. An Improved Illumination Model for Shaded Display. *Communications of the ACM*, 23(6):343–349, June 1980.

[2] Craig Kolb and Rod Bogart. *Rayshade 4.0*, 1991. From the documentation included in Release 4.0 of Rayshade. Available via anonymous ftp from weedeater.math.yale.edu as pub/rayshade.4.0.

[3] G. M. Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *Proc. AFIPS Spring Joint Computer Conf.*, pages 483–485, Atlantic City, NJ, April 1967.

[4] Michael B. Carter and Keith A. Teague. The Hypercube Ray Tracer. In *Proc. 5th Distributed Memory Computing Conference*, April 1990.

[5] Michael B. Carter and Keith A. Teague. Distributed Object Database Ray Tracing on the Intel ipSC/2 Hyercube. In *Proc. 5th Distributed Memory Computing Conference*, April 1990.

[6] Tim Myers. Ray Tracing on the Macintosh II via LAN Parallel Processing. *Netware*, October 1988.

[7] N. Magnenat-Thalmann and D. Thalmann. *New Trends in Computer Graphics - Proceedings of CG International '88.* Springer-Verlag, 1988.

[8] Michael Litzkow and Marvin Solomon. Supporting Checkpointing and Process Migration Outside the Unix Kernel. In *USENIX Winter Conference*, San Francisco, CA, January 1992.

[9] Nicholas Carriero and David Gelernter. Linda in Context. *Communications of the ACM*, 32(4):444–458, April 1989.

[10] University of Utah. *Utah Raster Toolkit*, June 1990. From the version 3.0 source code. Available via anonymous ftp from cs.utah.edu as pub/urt-*.

[11] Regents of the University of California. *BSD 4.3*, 1986. From the BSD Unix Version 4.3 source code.

[12] Chris D. Peterson and MIT X Consortium. *Athena Widget Set – C Language Interface X Window System.* MIT X Consortium, 1989. From the documentation included in the X11 Release 4 distribution.

[13] Alan Kilian and Jerome A. Farm. *aq17.ray.* Cray Research, Inc., 1991. The rayshade 4.0 data file for the aquarium image. Available via anonymous ftp from ftp.ee.lbl.gov as RAY/aq.tar.Z.

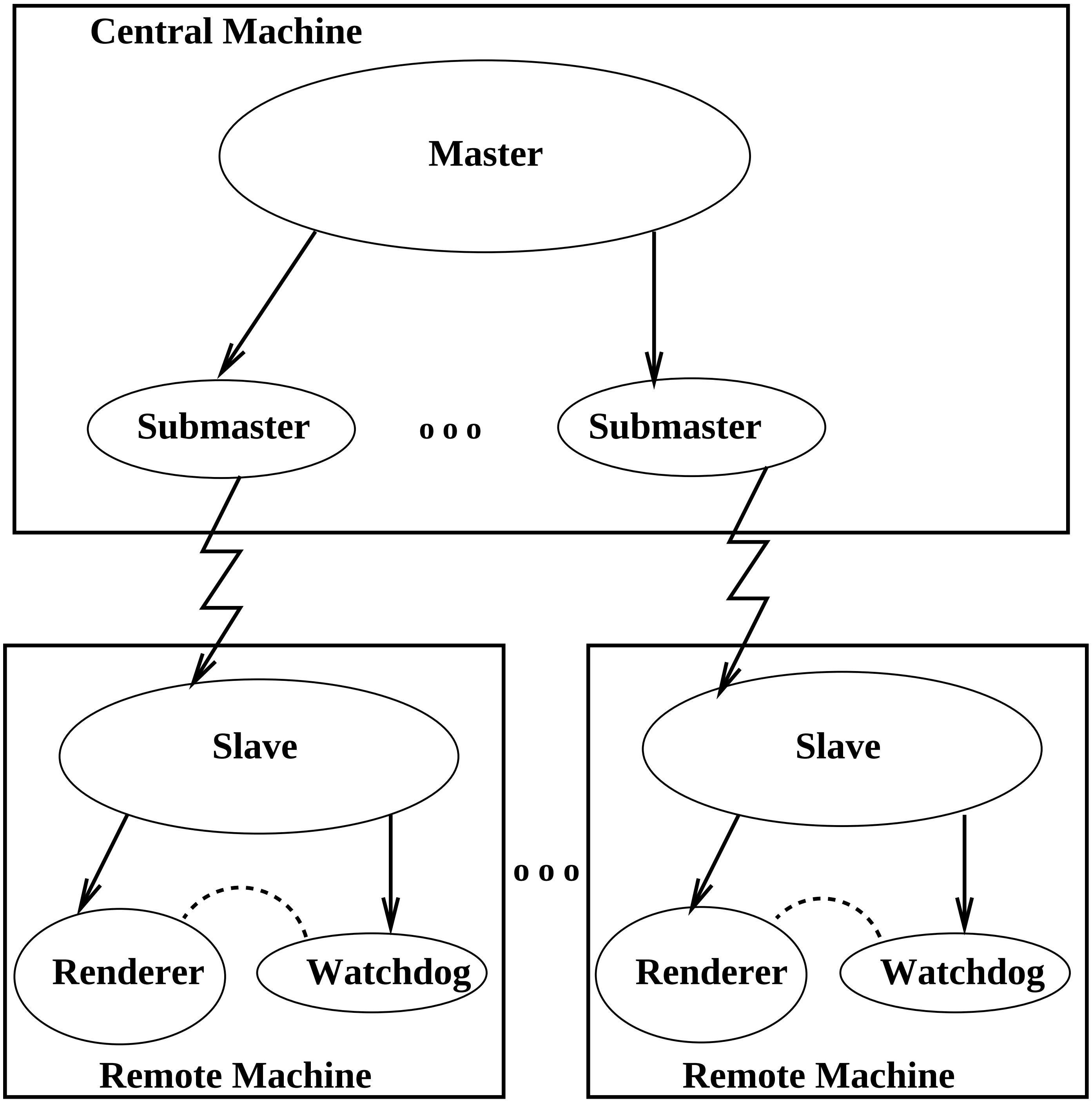

Figure 1: Process Hierarchy

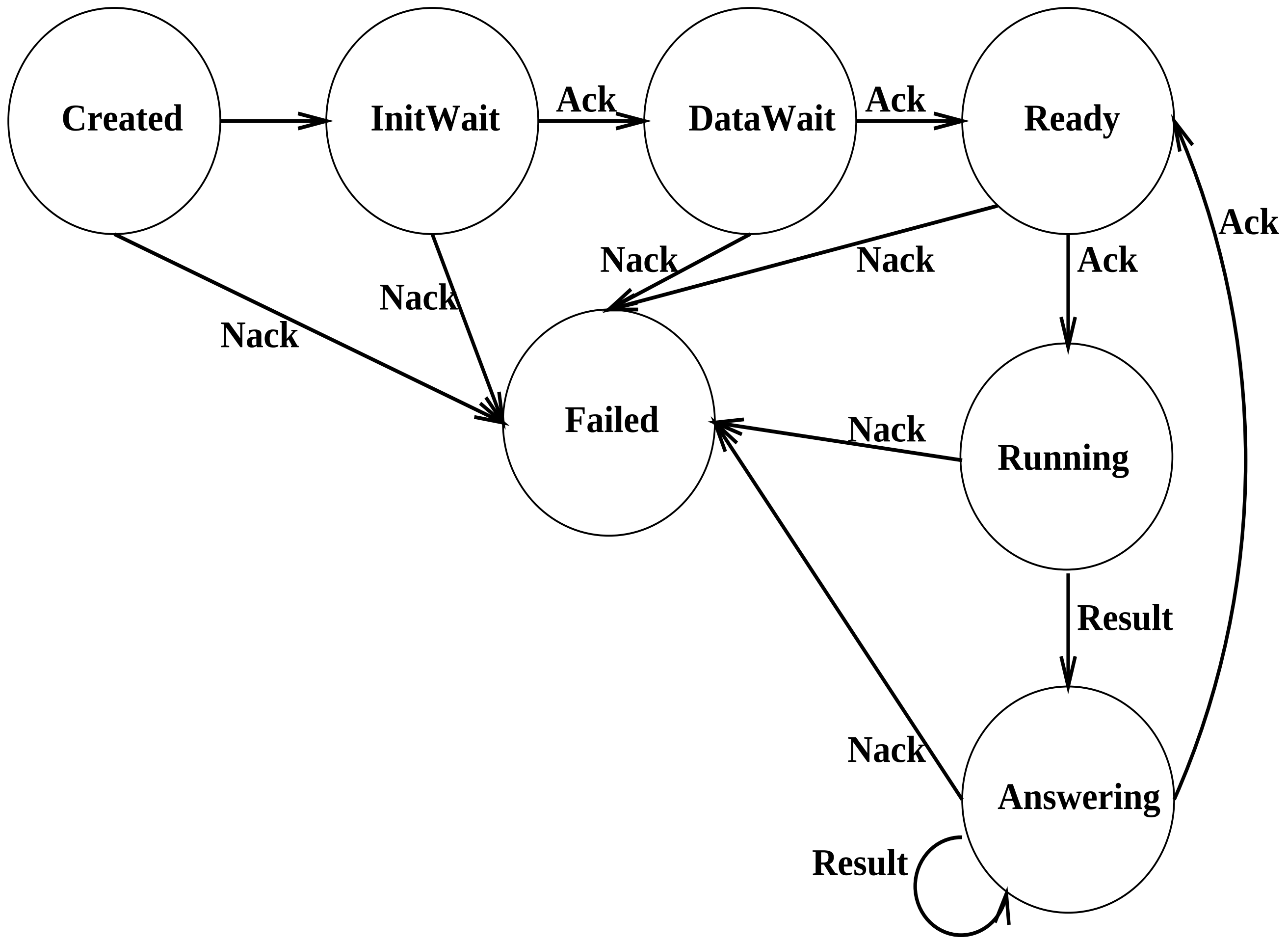

Figure 2: Process State Transitions

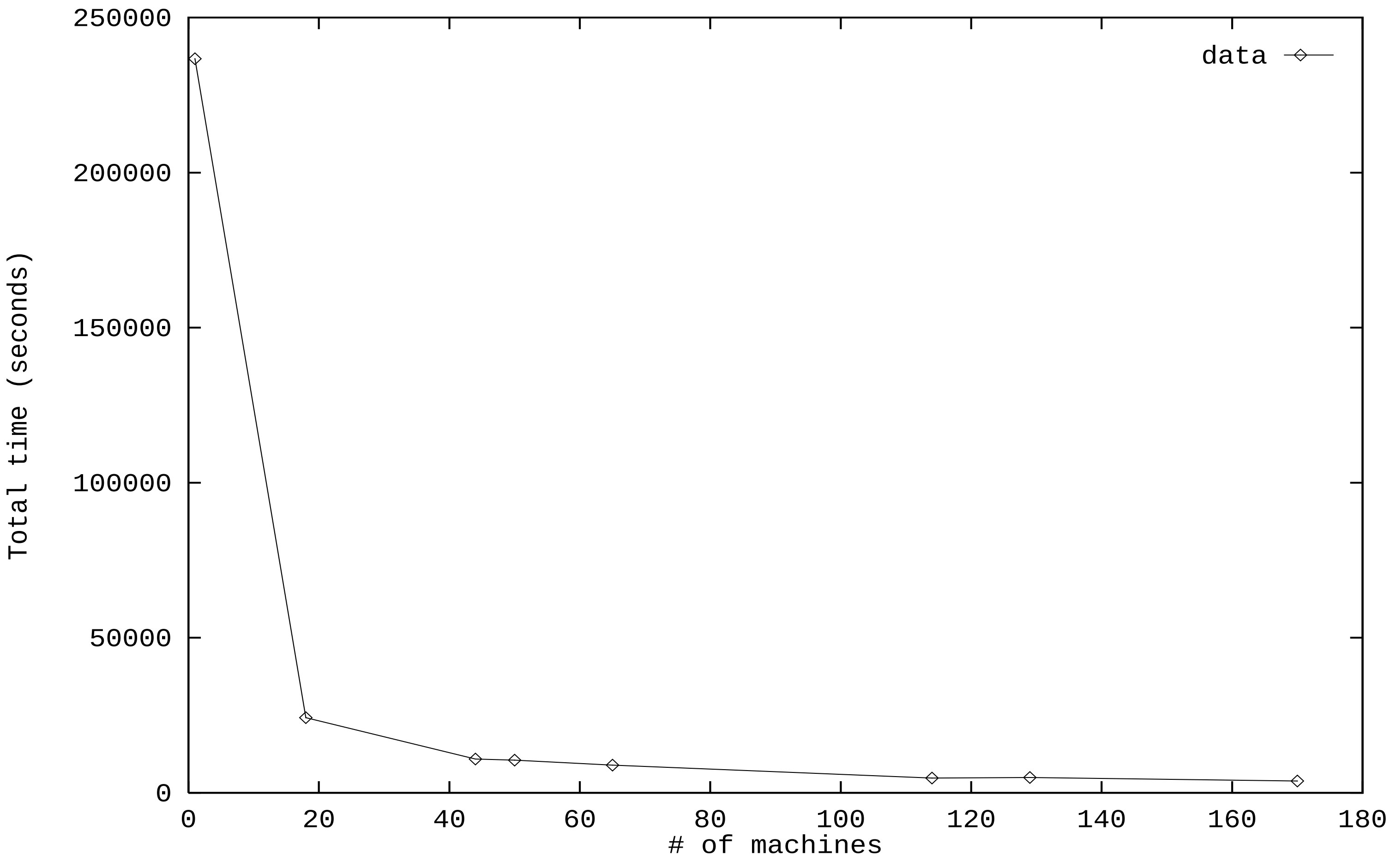

Figure 3: Results of Machines vs. Time to Render

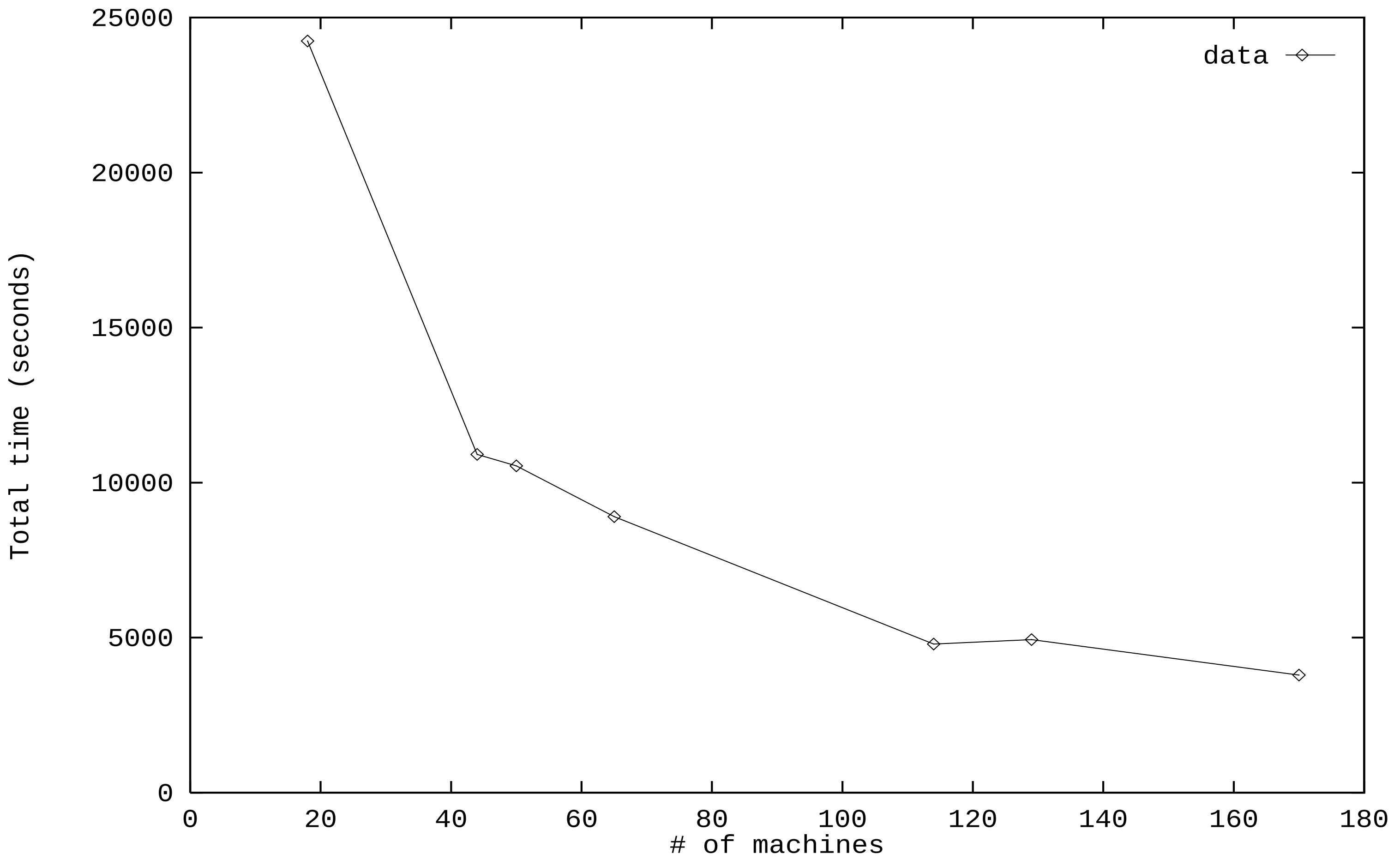

Figure 4: Results of Machines vs. Time to Render (excludes single machine run)