Distributed Computation for Computer Animation

John W. Peterson
Computer Science Deptartment
University of Utah

Abstract

Computer animation is a very computationally intensive task. Recent developments in image synthesis, such as shadows, reflections and motion blur enhance the quality of computer animation, but also dramatically increase the amount of CPU time needed to do it. Fortunately, the computations involved with computer animation are easily decomposed into smaller tasks, such as rendering single frames or parts of a frame. This makes the problem an ideal candidate for "coarse-grain" parallel implementation.

In order to provide the necessary cycles, unused idle time on personal workstations is used to provide a single large parallel computing resource. A survey of several schemes for coordinating this type of resource is presented, along with a detailed examination of a Unix based system currently in use at the University of Utah.

1 Introduction

As large networks of computers become commonplace, it has become interesting to consider them as a single computational resource, rather than individual machines. Recent advances in networking software such as distributed filesystems and remote procedure calls make using networks much more transparent to application programs. For some applications, the ability to use multiple machines in parallel is limited (e.g., complex simulations where the next iteration depends on data from the previous one). Other applications, such as computer animation, are ideally suited to parallel execution. This paper examines this type of application on large networks.

The type of parallelism explored in this paper is assumed to be course grained, with individual computations lasting minutes or hours instead of fractions of a second. Another assumption is that the computing resources are developed and maintained for general purpose use. In other words, we wish to take advantage of an existing resource, rather than develop one specifically for the task.

This paper gives a brief discussion of the scale of the resources we are discussing, and then presents an informal survey of existing systems for using computational power on networks as a whole. Finally, a system developed at the University of Utah for computer animation is examined in detail.

2 The Resources

The resources available on a network of workstations are dependent on two factors: how much the workstations are used by their dedicated users, and the power of the individual machines.

2.1 CPU usage

In addition to the obvious periods of idle time (night, weekends, etc.) a typical workstation CPU is usually not fully utilized even during the day. A workstation often spends its time performing relatively simple tasks, such as editing, reading mail and terminal emulation. Statistics gathered indicate a typical workstation CPU spends approximately 90–95% of its time in the idle loop[1].

Unfortunately, not all this idle time is directly available. If the CPU's idle loop is replaced with a major application, distracting side effects occur even if the application is running at a low priority. For example, if the workstation user is interacting with a Lisp interpreter, having another large application in the background may dramatically increase the paging activity and slow down the interaction.

2.2 CPU power

The availability of advanced microprocessors like the 68020/68881 chip set have blurred the distinction between mainframe and workstation computing power. Some recent benchmarks conducted at BRL [6] give the approximate comparisons:

> 68020 based workstation $\approx$ 1 Vax 780
>
> 4 processor Cray XMP/48 $\approx$ 90 Vax 780's.

So if you can get efficient parallelism:

> 90 workstations $\approx$ 4 processor Cray XMP/48

The economics of this comparison are interesting, since a Cray goes for something like \$10–\$15 Million vs. \$3-\$5 Million for a large workstation network.

3 Getting Parallelism The Hard Way

There are many examples of animation done by manually starting the computation on a number of machines. Among the best known are:

- Jim Blinn's animation of DNA molecules for the PBS *Cosmos* series. Blinn and his collegues wandered all over NASA's Jet Propulsion Laboratory after 5:00 pm looking for unused PDP-11's. When one was found, a tape was loaded on the machine and it was left to crunch away on part of the sequence for the evening. The results were collected on magnetic tape in the morning.[10]

- The short film *The Adventures of Andre and Wally Bee* produced by Lucasfilm was done on a larger geographic scale. Portions of the film were computed on a Cray in Minnesota and on ten Vaxes at MIT's project Athena. Data was shipped out and results were collected on tape. The final results were composited at Lucasfilm's facilities in California.

[1] Usage statistics were taken on the Sun and Apollo workstations in the CS department. The HP workstations don't appear to keep track of this information.

- Apollo Computer's film *Quest - A Long Ray's Journey Into Light* was computed on a few hundred workstations at Apollo. Although the machines were connected with a local area network, in the rush to complete the film little software was written for coordinating the computation. Instead, a person (given the screen credit 'Node Hunter and Gigabyte Master') typed the necessary commands into individual nodes. Since that project, more advanced software has been developed for starting the computations [1].[2]

4 Systems for Distributed Computation

4.1 The Xerox "Worm" Programs

One of the earliest examples of a system for performing distributed computation on a local area network is the Worm developed at Xerox PARC[9]. The worm worked as a layer on top of which applications were built. The program was executed on several machines at a time, each machine a *segment* of the worm. The worm worked at a relatively low level compared to more modern systems.

After a worm was initially started, it operated by attempting to fill out the rest of its segments. It would work through the network incrementally, probing nodes to find out if they were idle. When an idle node was found, the worm would continue to boot itself on the idle machines until it had filled out its segments. The segments of the worm communicated with each other using a limited broadcast, or *multicast* protocol.

Because the worm operated at such a low level (there was very little operating system support beneath it) controlling it was a major problem. If the worm encountered a serious problem it could crash the workstation it was running on. If a worm became corrupted as it moved from machine to machine, the corrupted segment might run, but would spawn new segments that would crash. Since the original worm thought it needed to fill out the rest of its segments, it would continue trying to boot until all of the machines on the net had crashed (the paper describes a situation where this actually happened).

One of the applications for the worm described by the paper is a multi-machine animation system. The worm was modified so one machine could serve as central control node. This in turn spawned a series of smaller worms that located the worker nodes. The master node (itself not part of a worm) would send out the basic scene description to the worker nodes, and would later collect the results.

4.2 Recent distributed computation systems

The Xerox Process Server Recently, Xerox has developed a system knows as the Process Server [4] for workstations running their Cedar environment. This system is designed to make excess cycles on a workstation available to other users on the net in a relatively transparent fashion. The system uses remote procedure calls and transparent access to file servers for communication between nodes. Three types of entities are provided by the system: *Clients*, the workstations that request services; *Servers*, the machines allowing computations to be run on them; and a *Controller* which processes the client's request and assigns a server to it.

[2]The films *Quest* and *The Adventures of Andre and Wally B.* appear in the anthology *Animation Celebration*, released by Expanded Entertainment in 1986.

When a user makes a request for work, the parameters (commands, arguments, etc.) are passed to the Client process on the user's workstation. The Client then contacts the Controller, and if the request seems valid, the Controller selects a server machine (based on which one appears least loaded) and returns the machine's identifier to the Client. The Client then contacts the Server. The Server fetches the files it needs, and starts executing the command. During execution, the Server uses the Client for file operations and answering questions about the user's environment. If an error occurs, the Server brings up an error window on the Client's node. If the Server aborts the computation (because the load was too high) or crashes, the Client must ask the Controller to assign it a new Server and restart the computation.

The system is implemented using a Remote Procedure Call (RPC) protocol. It is designed to be relatively transparent to the applications executed by it, with few changes needed to the source. It is intended for relatively large granularity computing (compilation, typesetting, image generation, etc.). Because the system uses specific, lightweight protocols, the Process Server runs with relatively little overhead.

Apollo's Network Computing System Apollo Computer recently developed a system called the Network Computing System (NCS) for sharing resources (including computation) across large networks of heterogeneous machines[3]. It provides a Remote Procedure Call interface, a network data representation definition, an interface compiler and support for replicated databases. The remote procedure call interface supports several scalar formats. These are automatically converted for different hosts (to compensate for differences in floating point, byte ordering, etc.). A Network Interface Definition Language automatically constructs the RPC networking interface from user-defined stubs. It also provides methods for passing more complex data structures over the network, such as trees.

NCS provides a Location Broker, a service that allows objects on the network to be found by type, interface or combinations of these characteristics. They are identified by Universal Unique Identifiers that are guaranteed to be unique across the network. Although NCS supports remote computation, it currently doesn't provide for automatically selecting hosts for the remote computation on the basis of load. This is planned as a future extension; nodes will be able to query a "compute slot allocator" to access a replicated database of candidate nodes.

Remote Unix Remote Unix (RU) is a system developed by Michael Litzkow at the University of Wisconsin [5]. This system is designed to allow a single process to operate for a very long time, migrating from machine to machine as various workstations are used or become idle. A unique feature of RU is that processes can be completely checkpointed as they execute, including the status of open files. This takes place when a user logs into a workstation. When the RU spooler finds another machine to restart the computation on, it resumes the checkpointed computation without loss of work. This facility also gives RU a large degree of fault tolerance, since if a machine crashes the process can always be restarted from the last checkpoint file.

The control system contains two components, a central *resource manager* for gathering information about all the available machines, and a *local scheduler* to make decisions affecting a particular workstation. The resource manager periodically polls the schedulers to determine which workstations are accepting RU jobs and what jobs are waiting to run. When it finds an "idle" workstation, it sends a message to the waiting job granting permission to execute on the idle machine.

RU has been in use at Wisconsin on a large network consisting of several larger Vaxes (11/750's, 11/780's) and about 100 MicroVax workstations. In one case a single job was able to accumulate 60 CPU days over a three month period. Although the system supports parallel execution by queuing several jobs at the same time, no statistics have been gathered on this mode of operation.

5 Some Example Systems

In this section we present some examples of systems actually used for animation or similar purposes. Because these systems are usually built around existing environments informally, there are not many published examples of them. In order to provide more examples, a poll was conducted on the Usenet and Arpanet networks requesting information about these types of systems. Most of these examples are from this poll.

(A note about notation: In the descriptions below, the word *dispatcher* means the machine responsible for controlling the computation. The computations are executed on *worker* nodes.)

5.1 Apollo/MBX based system

Part of the animation system described in [7] contained a method for using a large number of Apollo workstations. It was based on Apollo's MBX ("Mailbox") system routines. These routines allow inter-process communication between multiple workstations via filesystem objects known as mailboxes. After the dispatcher process opens a mailbox, the workers can open connections to the dispatcher process via this mailbox. Since all the Apollos on the network share the same filesystem, they can all open connections to this mailbox.

This system required the ray tracer to be modified to call the routines:

Init Opened the inital connection to the dispatcher's mailbox.

Send_status Sends a progress report (e.g., the current scanline number) to the dispatcher.

Test_login Asked the node if anybody had logged into it. If this routine returned true (somebody had logged in) the program is expected to call the next routine:

Shutdown Informs the dispatcher this node is no longer available, closes the MBX connection, and exits the program.

Finished Informs the dispatcher this node is finished with its task and can start on another.

The dispatcher would first open the MBX file, and start the worker processes on the remote nodes. A simple protocol was used for giving each worker a unique ID to identify itself, since the dispatcher received all of its input on a single channel. The dispatcher listened to messages generated by Send_status, Shutdown and Finished, and updates its record of the work done. If Finished was called, that workitem would be removed from the queue, if Shutdown was called it would be re-queued. Every transaction was recorded in a logfile.

As implemented, the system could tolerate worker failures but not dispatcher failures. Although never implemented, some ideas were planned for increasing the robustness. This included assigning one of the workers to be the "copilot". The copilot would receive a copy of the

dispatcher's state every time it performed a Send_status call. If the copilot tried to perform a Send_status and failed (i.e, the MBX channel was no longer open) it would spawn off a local copy of the dispatcher. This new dispatcher would re-open the MBX channel. If other workers tried to perform a Send_status and failed, they could try to close and re-open the MBX file to establish a connection to the new dispatcher.

Although the basic system was used to generate a few stills, it was never used for large scale work, mainly because the bulk of the computing resources were on other (non-Apollo) systems. The system was dropped, eventually replaced with one that could take advantage of any Unix host.

5.2 Locus based system at UCLA

At UCLA, Matthew Merzbacher developed a scheme for generating frames with a collection of ten Vaxes running the Locus operating system. Locus provides a common shared filesystem across several machines in a Unix environment. This allowed all of the Vaxes to access a single directory where all of the data and results were kept. The files were named after the worker machines and given suffixes indicating the state of the system (e.g.: athena.i, athena.r, athena.d).

After the dispatcher started, it created **.i** files named for each of the worker machines, containing the data for the rendering program, and spawned rendering processes on all of the workers. The worker process polled the main directory, waiting for a **.i** file with its name on it. When one was found, the worker created the **.r** file to indicate it was running. When it finished the job, the worker removed the **.r** file and created a **.d** file, indicating it was done.

The dispatcher polled the directory looking for the **.d** files. When one was found it removed the **.i** and **.d** files (in that order, to prevent the job from running twice) and places a new **.i** file in the directory. If the load on a machine was too high or if it was past 7 am, no new jobs would be started. Logs were kept of how many jobs per night were completed. Each job corresponded to a frame of the movie and required approximately ten minutes of Vax 11/750 time.

If the dispatcher failed, there was a backup dispatcher waiting to take over (which in turn would spawn a new backup). A new dispatcher was automatically started each evening with the Unix *at* utility. It could detect if a job had failed the previous night, because the directory would contain a **.i** file without a corresponding **.d** file.

5.3 TCP based system at BRL

At the Army Ballistics Research Laboratory, Mike Muuss developed a system for taking advantage of idle time on large mainframes and supercomputers. The ray tracing program was modified so it could operate remotely, receiving and writing information over a TCP connection. The work is dispatched to the worker machines in small portions of a frame (e.g., three scanlines) and collected by the dispatcher after each scanline is finished. Each worker has a private copy of the database, but the information specific to the frame being rendered (viewpoint, positions of objects, etc.) is transmitted directly to the worker via point-to-point TCP connections. Machines are selected manually, and can be added and dropped from the pool of workers on the fly. The rendering process runs at a very low priority on the worker mainframes.

If a worker fails, the job running on it is automatically re-queued on the next available machine. However, the dispatcher writes out the data collected from the workers after every

frame, so the entire frame is lost if it fails. The system is usually run with mainframes or supercomputers, in one instance 13 Gould 9000 series machines "all over the east coast" were used.

5.4 Lisp Machine based system at the MIT Media Lab

At the MIT Media Lab, Steve Strassmann developed a system for using up idle time on Lisp Machines. When each host boots, it creates a copy of the idle time "server" daemon. This daemon remains dormant until it detects the machine is idle. Then the daemon wakes up and reads a job specification file from a central host. It picks a job to run and executes it. Synchronization and the division of labor are specified by the application the daemon is executing, not by the daemon itself.

If a job is interrupted by an error, it quits and the daemon goes back to the central job specification file to see what to do next. An arbitrary "clean up" procedure can be associated with a job, and is executed whenever the job exits (write to a log file, etc.). The job specification also allows for a maximum execution time for a job (kill it after N minutes of CPU time), uniqueness (only one copy of the job is run at a time) or logging (start and stop times are logged in a central file). If the central job description file is unavailable, the daemon goes to sleep until it can re-open it.

The system is not tied to any particular task. Applications have included running diagnostics on a connection machine, and 'frivolous console animations'.[3]

5.5 File based system at NYIT

While at NYIT, Paul Heckbert developed a scheme for soaking up the idle time on eight Vaxes there. Because the systems went down at least once a day for backups and loads across machines were uneven, the system had to be fault tolerant, de-centralized and able to deal with loaded and unloaded machines.

The boot script for each of the vaxes was modified to start a daemon responsible for running the computation. This daemon would read a "job/log" file containing a list of shell commands to run and the status of each. For example, a job/log file for computing five frames of animation might look like this:

```
done(vaxb, vaxg)        gen.sh 0
done(vaxc)              gen.sh 1
done(vaxa)              gen.sh 2
running(vaxa)           gen.sh 3
-                       gen.sh 4
```

This file means that frames zero through two are done, frame zero was computed on two machines (vaxb and vaxg), frame three is being computed on vaxa, and frame four hasn't started yet.

The daemon read this file and picked a job to run based on the following priorities:

[3] This is much like the applications for the Xerox PARC "Worm" program.

1. If a job is listed as running on the machine reading the file, then it must have crashed, so resume work on that job. The ray tracing program was written so it could resume computation in mid-job to minimize lost work.

2. Run an unstarted job, if any are left.

3. Run a running job. This is useful in the case of a another machine crashing or slowing down due to a heavy load.

Each machine decides which jobs to run, there is no single master machine. Just before a machine started up a job, it would update the status in the job/log file and then copy it over the network to the other machines in the pool. The shell script started by the daemon saved its output in a common directory. This was inspected once a day and the results transferred to a big disk.

The system was used for three large jobs:

- An animated sequence of 120 ray-traced frames. It took 84 CPU-days over a period of 19 days on seven Vaxes (six Vax 780's and one Vax 750);

- An eight CPU-day ray-traced image of a morphine molecule (computed at a resolution of 2048x2048);

- Computing all amicable number chains up to 200,000,000 (a number theory problem).

On the larger jobs, the system was able to use 65% to 75% of the available CPU time. It was able to recover from machine crashes and shutdowns, and ran around the clock on seven machines for several weeks. The only significant problem was the job/log files becoming inconsistent on the various machines, probably because there was no locking scheme for the job/log files.

5.6 Systems at Xerox PARC

While at Xerox PARC Steve Schiller developed a system for using approximately 100 workstations to compute an animated sequence of fractal images. In that system, one of the workstations served as the main dispatcher for the computation. It made a remote procedure call to a worker machine, giving it the parameters for computing a particular frame. When the worker finished computing the frame it would inform the dispatcher that it was finished. It was up to the dispatcher to actually retrieve the frame. The dispatcher could also ask a worker if it was busy, and if so, when it expected to finish the frame. If somebody logged into the machine, the computation stopped, and the work was re-queued on the next available machine (code was implemented to re-start partial frames, but became a source of trouble and was dropped). The computation times ranged from five minutes to one hour per frame, depending on the complexity of an image (twenty minutes was the average).

The scheduling of the computations was complicated by disk space constraints on the dispatcher. The frames had to be recorded by the camera in the correct order and then removed to make room for new frames. However, the workers might not have finished them in the proper sequence. Since the dispatcher had only twenty frames worth of available disk space, there would occasionally be times when the dispatcher could not retrieve a frame because it didn't

have enough disk space. (Of course, enough space must be available for the frame the camera is waiting on).

In order to help avoid this problem, the dispatcher kept track of all of its outstanding frame requests and when they were made. When it wasn't busy with anything else it checked the machine working on the frame the camera was waiting on. If that machine was unreachable (crashed, net problems, somebody logged into it, etc.) or was taking a suspiciously long time, then the dispatcher re-assigned the frame to the next free machine. A log was kept of when each frame was retrieved, how long it took to complete it and the name of the machine that worked on it. This was useful for pinpointing slow or unreliable workers.

Once the kinks were worked out, the system was fairly reliable. Schiller estimates about nine out of ten 48 hour runs were without incident. The system achieved approximately 80% parallelism during operation.

5.7 Work with finer-grained parallelism

More recently at PARC, Frank Crow has experimented with using groups of workstations to compute single images, rather than animation. The distribution is done with the Xerox Compute server (described above). Instead of decomposing the problem by dividing the image up (as most approaches presented above), Crow rendered individual objects in the scene on different processors. These objects must be linearly separable (see [2]), so the method is restricted to '2.5D graphics'. The motivation for this method is that it is easier to predict the time to render a given object rather than the time to compute a slice of an image.

The system was initially tested on images with a small number of linearly separable shapes. These were sorted by depth on the "home" (dispatcher) machine and sent to other worker machines for rendering. Each worker would render the pixels in the bounding rectangle of the shape, and return this image along with a coverage mask for the shape[8]. Finally, the images were composited together on the dispatcher machine.

The improvements gained by distributing the work this way were not substantial. Some statistics of the system's operation were gathered, such as which processor got what job, how long it took, and how much time was spent compositing the images together. This revealed three important things: 1) Some shapes took much longer to render than others; 2) The processes were unevenly distributed to the processors; and 3) The final compositing phase was taking long enough to prevent dramatic speedups on complex images. The process distribution itself was also a source of overhead, as data files had to be shipped out and images collected.

Some steps were taken to improve the benefits of distributing the work. Since the disparity between rendering times for different objects was much larger than expected, some heuristics were developed for estimating the cost of rendering an object, and allowing it to be rendered in several strips. The compositor was also substantially optimized, reducing a major bottleneck. With these improvements in place, Crow was able to improve the parallelism to over 30%. Crow describes the system as "Work in progress" — it has no doubt improved substantially since the paper was written.

6 The Distrib System

At the University of Utah Computer Science Department a system called *Distrib* (developed by Rod Bogart, Glenn McMinn and the author) is currently in use for distributing animation computations over a large network of workstations. The computing environment used by Distrib consists of a large network of workstations, including Apollos, Suns, and a large number of Hewlett Packard Series 9000/300 machines. All of these machines are accessible over the Ethernet using the TCP/IP protocols, and all run some variant of Unix. A Vax 11/785 with a large amount of disk space serves as the central dispatcher machine where Distrib runs.

6.1 Operation

Distrib reads as input two files, one containing a list of jobs to execute and the other a list of machines to execute them on. The *job* file specifies for each job the input and output data files to use, the script to execute on the remote machine, and the parameters for that job (scanlines to render, frame number, rendering options, etc.). The *machines* file describes where the files (programs, texture maps, data files, etc.) live on each machine, and also specifies any restrictions on the use of a given machine. Machines can be set up in three ways:

Unrestricted Distrib uses the machine without reguard to time of day or if somebody is logged in. This mode is used for lightly used machines, where the additional rendering job doesn't cause a major impact. (Users of the machine are always able to kill the job if it does get in the way).

Unoccupied Distrib only uses the machine if nobody is logged in or running a "screensaver" program.

Night only Like unoccupied, except that if Distrib finds the machine in use it won't even check back until the evening (or weekend).

As explained below, it's possible to change these restrictions while Distrib is running.

When Distrib starts it reads in the job and worker machine description files. For each worker, Distrib copies the appropriate data files to the worker, using the Unix *rcp* program. It then uses the *rexec* routine to start the computation on the worker. *Rexec* returns a socket file descriptor that listens to the remote process on the worker machine. Once all of the hosts are started, Distrib listens to all of the *rexec* connections simultaneously with the *select* system call.

When the *select* call returns, (indicating activity on one or more of the sockets) Distrib looks at the messages returned by the worker machines. If the message indicates successful completion of the job, Distrib collects the results from worker (verifying the transfer) and cleans up the data area on the worker. If the worker machine was specified as restricted, Distrib makes sure the machine is still available (i.e., nobody has logged in) before starting another job on it.

If the message from the worker's socket indicates failure (e.g., the process is terminated by somebody logging in, the job stops unexpectedly with an error, or the socket simply closes because a worker crashes) Distrib acts according to the machine's restriction. If the machine is "unrestricted", Distrib marks it as "down", and waits an hour before trying to re-use it. If a machine is marked as "restricted", Distrib marks it as "occupied" and waits until it is free before trying to use it again. In any case, the aborted job is re-queued on the next free machine. Distrib maintains an extensive log of all of this activity.

6.2 Problems encountered

Several interesting problems were encountered in the process of getting Distrib to run reliably. In the original version an *rsh* process was forked to start the remote process instead of using *rexec*. Instead of returning a socket, this returned a process ID and the *wait* system call was used to detect when jobs were finished. While simple to implement, this uncovered a number of problems. Most noticeable was that because each *rsh* created two processes, the Distrib program quickly exceeded the Unix limit of the number of processes a user is allowed to have when a large number of workers were used.

In the original versions of Distrib, a job's input data was copied *from* the dispatcher by each of the individual workers. Distrib would spawn the all workers simultaneously, and they would all start asking the dispatcher for data at the same time. This flooded the dispatcher with I/O requests, and often some of the requests would fail because system limits were exceeded. Distrib now copies the necessary files *to* the workstation before starting the job on it. This serializes the I/O, and prevents the dispatcher from being swamped with file requests.

Because TCP/IP is a "reliable" protocol, connections will not time out once they are initiated. In one case, a worker crashed as the data files were being copied to it, and Distrib became hung waiting for the transfer to complete, preventing it from starting work on other machines. It now sets up a "watchdog" timer before sending or retrieving files from workers. If the transfer doesn't complete before the timer runs out, Distrib receives a signal and the job is aborted. The worker is marked as "down" and the job is re-queued.

6.3 Interaction with Distrib

A Distrib run may last for several days. During this period of time, it's useful to be able to interact with Distrib to inquire about the status of the jobs or to make minor adjustments to its state. To accommodate this, Distrib listens to a "command" socket in addition to the *rexec* sockets. When a connection is made to this socket (usually with a utility like telnet) the user can interact with Distrib and find out exactly what the status of the computation is. This is usually much quicker than trying to get the same information from Distrib's log files, which become quite large during a long run.

Another use for the command socket is to change the state of the machines Distrib is controlling. For example, an unrestricted machine can be changed to restricted if Distrib was interfering with its normal use, or a recently re-booted machine can be changed from down to up.

6.4 What if Distrib dies?

The advantage to using the *rexec* connection is Distrib knows exactly when a workstation finishes (or aborts). There is no periodic polling needed to get a worker's status. A disadvantage to this approach is that most of Distrib's state is in the form of open *rexec* sockets. If the machine Distrib is running on goes down, there is no way to recover this information. When the dispatcher dies, the usual approach is to wait an hour or so for most of the workers to finish their jobs.[4] Scripts are then run to collect any finished work, kill any remaining jobs, and clean

[4] i.e, go out for pizza...

up the worker data directories. A new job file is made by subtracting the finished work from the original job file, and Distrib is re-started.

This problem could be solved by making the system running on the worker end more intelligent. The worker would have two processes, one to do the rendering and the other to talk to Distrib. This second "supervisor" process could detect if Distrib went away, and listen for a new Distrib if it did. When Distrib is restarted, it would contact all of the supervisor processes, determine their state, and pick up the computations based on this information. Fortunately, the Vax Distrib is usually run on has proved quite reliable, so motivation to implement this scheme has been low.

6.5 Some Results

Some example Distrib runs include:

- A high resolution still of a butterfly. The image was computed at 1024x1024 pixel resolution. The jobs consisted of ray-tracing 32 horizontal strips of the image. It took three hours elapsed time using 13 idle HP Series 9000/320 machines. The total CPU time used was 27 hours, so the computation achieved about 70% efficient parallelism.
- A simple animated station logo (approximately two seconds worth). It took 24 hours elapsed time using 30 workstations. The total CPU time used was 24 days, 10 hours (about 80% efficient).
- Another run for producing twelve seconds of animation took 64 hours of elapsed time. It ran on 60 workstations (some of them un-available during the day). The total CPU time was three months, 3 days and 20 hours (2252 hours), about 57% efficient.

In most of these cases, production deadlines were met that would not have been possible without a facility like Distrib.

7 Conclusions

There are some consistent features in the systems presented above. Almost all of them provide facilities for logging the activity performed. Since the computation involved often extends over hours or days, there is no other way to supervise the work. Log files are often the only way to debug the system when it's actually in use.

Fault tolerance is an important issue. Even if the average reliability of a machine is good (say, only one shutdown or failure a month), this decreases rapidly as you use more machines (e.g., 30 machines gives you one failure a day). Without at least some facility for dealing with worker failures, a distributed computation system often grinds to a halt.

The computing resources offered on a typical large workstation network are substantial — often equivalent to a single supercomputer. Since computer animation is an easily decomposable and large-grained problem, it makes an ideal problem for solving with distributed computation.

8 Acknowledgements

I would like to thank the many people who responded to the Usenet survey and took the time to write up their experiences, Jules Bloomenthal at Xerox PARC for providing information about recent work there, and Jay Lepreau for pointing out recent work with Unix. Glenn Mcminn and Robert Mecklenburg gave the paper a good critical reading.

Peter Ford, Mark Bradakis, and "Charlie Root" provided us with valuable assistance while getting Distrib running.

We would also like to thank the Hewlett Packard corporation for their generous gift of HP workstations. These systems allowed us to work on a very large scale.

This work was supported in part by DARPA (DAAK1184K0017) and the National Science Foundation (MCS-8121750). All opinions, findings, conclusions or recommendations expressed in this document are those of the author and do not necessarily reflect the views of the sponsoring agencies.

References

[1] F. C. Crow. *Experiences in Distributed Execution: A Report on Work in Progress.* Tutorial Course Notes: Advanced Image Synthesis, ACM-SIGGRAPH, August 1986.

[2] F. C. Crow. A more flexible image generation environment. *Computer Graphics*, 18(3), July 1984.

[3] T. H. Dineen, P. J. Leach, N. W. Mishkin, J. N. Pato, and G. L. Wyant. The network computing architecture and system. In *Proc. 1987 Summer Usenix Conferance*, Usenix, June 1987.

[4] R. Hagmann. Process sever: sharing processing power in a workstation environment. In *6th Intl. Conf. on Distributed Computing*, IEEE, Cambridge, MA, May 1986.

[5] M. J. Litzkow. Remote unix — turning idle workstations into cycle servers. In *Proc. Summer Usenix Conferance*, Usenix, Phoenix, AZ, June 1987.

[6] Michael Muuss. *Solid Modeling System and Ray-Tracing Benchmark.* Distribution Release Notes, U.S. Army Ballistics Research Lab, December 1986.

[7] John W. Peterson. *A System For High Quality Image Synthesis.* CS Project Memo 86-01, Unversity of Utah, June 1984.

[8] Thomas Porter and Tom Duff. Compositing digital images. *Computer Graphics*, 18(3):253, July 1984. Proceedings of SIGGRAPH 84.

[9] John F. Shoch and Jon A. Hupp. The worm programs - early experience with a distributed computation. *Communications of the ACM*, 25(3):172, March 1982.

[10] Turner Whitted. *The Hacker's Guide to Making Pretty Pictures.* Tutorial course notes – Image Rendering Tricks, ACM-SIGGRAPH, August 1986.