Excerpts from ‘‘Workstations, Networking, Distributed Graphics, and Parallel Processing’’

Michael John Muuss

Leader, Advanced Computer Systems Team
Ballistic Research Laboratory
Aberdeen Proving Ground
Maryland 21005-5066 USA

ABSTRACT

The process of design is iterative in nature; designs are formulated, analyzed, and improved, until the goals are met. Modern graphics workstations provide a powerful platform for performing detailed design and limited analysis of solid model designs, with faster computers accessed via network links for full resolution analysis. A broad spectrum of analysis tools exist, and most express their output in graphical form. Their three main styles of graphics display will be examined, along with a look at the underlying software mechanisms to support them.

Several enabling technologies are required for analysis tools to run on one computer, with graphical display on another computer, including the network transparent framebuffer capability. The importance of portable external data representations will be reviewed, and several specific external data representations will be examined in significant detail. By carefully dividing application software into appropriate tools and connecting them with UNIX pipes, a measure of parallel processing can be achieved within one system. In a tool oriented environment with machine independent data formats, network distributed computation can be accomplished.

The next step is to make a single tool execute faster using parallel processing. Many analysis codes are implemented using the ray-tracing paradigm which is ideal for execution in parallel on tightly coupled shared-memory multiprocessors and loosely coupled ensembles of computers. Both are exploited, using different mechanisms. The strategies used for operating on shared-memory multiprocessors such as the Denelcor HEP, Alliant FX/8, and Cray X-MP will be presented, along with measured performance data.

The strategies used for dividing the work among network connected loosely coupled processors (each of which may themselves be a parallel processor) are presented, including details of the dispatching algorithm, and the design of the distribution protocol. The performance issues of this type of parallel processing will be presented, including a set of measured speeds on a variety of hardware.

January 28, 1991

Excerpts from
''Workstations, Networking, Distributed Graphics, and Parallel Processing''

Michael John Muuss

Leader, Advanced Computer Systems Team
Ballistic Research Laboratory
Aberdeen Proving Ground
Maryland 21005-5066 USA

1. (Early Chapers Omitted)

2. SHARED-MEMORY PARALLEL PROCESSING

In the preceding section, we have seen how different processors can be harnessed to achieve a single goal. The discussion so far has focused on using multiple processors (a) within a single, multi-CPU system, through the use of UNIX pipes, and (b) by distributing different tools in a pipeline to different machines on the network. This section extends the investigation into parallel processing one level further, to harness the power of multiple processor machines to make a single application run faster. For the purposes of this discussion, the application to be parallelized will be a ray-tracing program, but the techniques developed here are quite general.

2.1. The Need for Speed

Images created using ray-tracing have a reputation for consuming large quantities of computer time. For complex models, 10 to 20 hours of processor time to render a single frame on a DEC VAX-11/780 class machine is not uncommon. Using the ray-tracing paradigm for engineering analysis Muuss Understanding Solid Models often requires many times more processing than rendering a view of the model. Examples of such engineering analyses include the predictive calculation of radar cross-sections, heat flow, and bi-static laser reflectivity. For models of real-world geometry, running these analyses approaches the limits of practical execution times, even with modern SuperComputers. There are three main strategies that are being employed to attempt to decrease the amount of elapsed time it takes to ray-trace a particular scene:

1) Advances in algorithms for ray-tracing. Newer techniques in partitioning space Kaplan Space-Tracing and in taking advantage of ray-to-ray coherence Arvo Ray Classification promise to continue to yield algorithms that do fewer and fewer ray/object intersections which do not contribute to the final results. Significant work remains to be done in this area, and an order of magnitude performance gain remains to be realized. However, there is a limit to the gains that can be made in this area.

2) Acquiring faster processors. A trivial method for decreasing the elapsed time to run a program is to purchase a faster computer. However, even the fastest general-purpose computers such as the Cray X-MP and Cray-2 do not execute fast enough to permit practical analysis of all real-world models in appropriate detail. Furthermore, the speed of light provides an upper bound on the fastest computer that can be built out of modern integrated circuits; this is already a significant factor in the Cray X-MP and Cray-2 processors, which operate with 8.5 ns and 4.3 ns clock periods respectively.

3) Using multiple processors to solve a single problem. By engaging the resources of multiple processors to work on a single problem, the speed-of-light limit can be circumvented. However, the price is

that explicit attention must be paid to the distribution of data to the various processors, synchronization of the computations, and collection of the results.

Parallel processing is still a relatively young art, and presently there is only limited support available for the automatic parallelization of existing code, with newer vendors like Alliant leading the crowd. For now, there are few general techniques for taking programs intended for serial operation on a single processor, and automatically adapting them for operation on multiple processors. Ohr Minisupercomputers Speed The **Worm** program developed at Xerox PARC Shoch Worm Distributed Computation is one of the earliest known network image-rendering applications. More recently at Xerox PARC, Frank Crow has attempted to distribute the rendering of a single image across multiple processors, Crow Distributed Execution Work in Progress but discovered that communication overhead and synchronization problems limited parallelism to about 30% of the available processing power. A good summary of work to date has been collected by Peterson. Peterson Distributed Computation

Ray-tracing analysis of a model has the very nice property that the computations for each ray/model intersection are entirely independent of other ray/model intersection calculations. Therefore, it is easy to see how the calculations for each ray can be performed by separate, independent processors. The underlying assumption is that each processor has read-only access to the entire model database. While it would be possible to partition the ray-tracing algorithm in such a way as to require only a portion of the model database being resident in each processor, this would significantly increase the complexity of the implementation as well as the amount of synchronization and control traffic needed. Such a partitioning has therefore not yet been seriously attempted.

It is the purpose of the research reported in the rest of this paper to explore the performance limits of parallel operation of ray-tracing algorithms where available processor memory is not a limitation. While it is not expected that this research will result in a general purpose technique for distributing arbitrary programs across multiple processors, the issues of the control and distribution of work and providing reliable results in a potentially unreliable system are quite general. The techniques used here are likely to be applicable to a large set of other applications.

2.2. Raytracing Background

The origins of modern ray-tracing come from work at MAGI under contract to BRL, initiated in the early 1960s. The initial results were reported by MAGI geometric description technique MAGI in 1967. Extensions to the early developments were undertaken by a DoD Joint Technical Coordinating Group effort, resulting in publications in 1970 MAGIC User Manual and 1971. MAGIC Analyst Manual A detailed presentation of the fundamental analysis and implementation of the ray-tracing algorithm can be found in these two documents. Also see Appel shading machine renderings solids

More recently, interest in ray-tracing developed in the academic community, with Kay's Kay Ray Tracing 1979 thesis in 1979 being a notable early work. One of the central papers in the ray-tracing literature is the work of Whitted. Whitted Improved Illumination Model Model sampling techniques can be improved to provide substantially more realistic images by using the ''Distributed Ray Tracing'' strategy. Cook Carpenter Distributed Ray Tracing For an excellent, concise discussion of ray-tracing, consult pages 363-381 of Rogers. Rogers Procedural Elements

There are several implementation strategies for interrogating the model by computing ray/geometry intersections. The traditional approach has been batch-oriented, with the user defining a set of ''viewing angles'', turning loose a big batch job to compute all the ray intersections, and then post-processing all the ray data into some meaningful form. However, the major drawback of this approach is that the application has no dynamic control over ray paths, making another batch run necessary for each level of reflection, etc.

In order to be successful, applications need: (1) dynamic control of ray paths, to naturally implement reflection, refraction, and fragmentation into multiple subsidiary rays, and (2) the ability to fire rays in arbitrary directions from arbitrary points. Nearly all non-batch ray-tracing implementations have a specific closely coupled application (typically a model of illumination), which allows efficient and effective control of the ray paths. However, the most flexible approach is to implement the ray-tracing capability as a general-purpose library, to make the functionality available to any application as needed, and this is the approach taken in the BRL CAD Package. Muuss CAD Package Release 1.21 The ray-tracing library is called **librt**, while the ray-tracing application of interest here (an optical spectrum lighting model) is called **rt**.

2.3. The Structure of librt

In order to give all applications dynamic control over the ray paths, and to allow the rays to be fired in arbitrary directions from arbitrary points, BRL has implemented its third generation ray-tracing capability as a set of library routines. **librt** exists to allow application programs to intersect rays with model geometry. There are four parts to the interface: three preparation routines and the actual ray-tracing routine. The first routine which must be called is rt_dirbuild(), which opens the database file, and builds the in-core database table of contents. The second routine to be called is rt_gettree(), which adds a database sub-tree to the active model space. rt_gettree() can be called multiple times to load different parts of the database into the active model space. The third routine is rt_prep(), which computes the space partitioning data structures and does other initialization chores. Calling this routine is optional, as it will be called by rt_shootray() if needed. rt_prep() is provided as a separate routine to allow independent timing of the preparation and ray-tracing phases of applications.

To compute the intersection of a ray with the geometry in the active model space, the application must call rt_shootray() once for each ray. Ray-path selection for perspective, reflection, refraction, etc, is entirely determined by the application program. The only parameter to the rt_shootray() is a **librt** ''application'' structure, which contains five major elements: the vector a_ray.r_pt which is the starting point of the ray to be fired, the vector a_ray.r_dir which is the unit-length direction vector of the ray, the pointer *a_hit() which is the address of an application-provided routine to call when the ray intersects the model geometry, the pointer *a_miss() which is the address of an application-provided routine to call when the ray does not hit any geometry, the flag a_onehit which is set non-zero to stop ray-tracing as soon as the ray has intersected at least one piece of geometry (useful for lighting models), plus various locations for each application to store state (recursion level, colors, etc). Note that the integer returned from the application-provided a_hit()/a_miss() routine is the formal return of the function rt_shootray(). The rt_shootray() function is prepared for full recursion so that the a_hit()/a_miss() routines can themselves fire additional rays by calling rt_shootray() recursively before deciding their own return value.

In addition, the function rt_shootray() is serially and concurrently reentrant, using only registers, local variables allocated on the stack, and dynamic memory allocated with rt_malloc(). The rt_malloc() function serializes calls to **malloc**(3). By having the ray-tracing library fully prepared to run in parallel with other instances of itself in the same address space, applications may take full advantage of parallel hardware capabilities, where such capabilities exist.

2.4. A Sample Ray-Tracing Program

A simple application program that fires one ray at a model and prints the result is included below, to demonstrate the simplicity of the interface to **librt**.

```
#include <brlcad/raytrace.h>
struct application ap;
main() {
    rt_dirbuild("model.g");
    rt_gettree("car");
    rt_prep();
    ap.a_point = [ 100, 0, 0 ];
    ap.a_dir = [ -1, 0, 0 ];
    ap.a_hit = &hit_geom;
    ap.a_miss = &miss_geom;
    ap.a_onehit = 1;
    rt_shootray( &ap );
}
hit_geom(app, part)
struct application *app;
```

```
struct partition *part;
{
    printf("Hit %s", part->pt_forw->pt_regionp->reg_name);
}
miss_geom(){
    printf("Missed");
}
```

2.5. Normal Operation: Serial Execution

When running the **rt** program on a serial processor, the code of interest is the top of the subroutine hierarchy. The function main() first calls get_args() to parse any command line options, then calls rt_dirbuild() to acquaint **librt** with the model database, and view_init() to initialize the application (in this case a lighting model, which may call mlib_init() to initialize the material-property library). Finally, rt_gettree() is called repeatedly to load the model treetops. For each frame to be produced, the viewing parameters are processed, and do_frame() is called.

Within do_frame(), per-frame initialization is handled by calling rt_prep(), mlib_setup(), grid_setup(), and view_2init(). Then, do_run() is called with the linear pixel indices of the start and end locations in the image; typically these values are zero and width*length-1, except for the ensemble computer case. In the non-parallel cases, the do_run() routine initializes the global variables cur_pixel and last_pixel, and calls worker(). At the end of the frame, view_end() is called to handle any final output, and print some statistics.

The worker() routine obtains the index of the next pixel that needs to be computed by incrementing cur_pixel, and calls rt_shootray() to interrogate the model geometry. view_pixel() is called to output the results for that pixel. worker() loops, computing one pixel at a time, until *cur_pixel* > *last_pixel*, after which it returns.

When rt_shootray() hits some geometry, it calls the a_hit() routine listed in the application structure to determine the final color of the pixel. In this case, colorview() is called. colorview() uses view_shade() to do the actual computation. Depending on the properties of the material hit and the stack of shaders that are being used, various material-specific renderers may be called, followed by a call to rr_render() if reflection or refraction is needed. Any of these routines may spawn multiple rays, and/or recurse on colorview().

2.6. Parallel Operation on Shared-Memory Machines

By capitalizing on the serial and concurrent reentrancy of the **librt** routines, it is very easy to take advantage of shared memory machines where it is possible to initiate multiple ‘‘streams of execution’’ or ‘‘threads’’ within the address space of a single process. In order to be able to ensure that global variables are only manipulated by one instruction stream at a time, all such shared modifications are enclosed in critical sections. For each type of processor, it is necessary to implement the routines RES_ACQUIRE() and RES_RELEASE() to provide system-wide semaphore operations. When a processor acquires a resource, and any other processors need that same resource, they will wait until it is released, at which time exactly one of the waiting processors will then acquire the resource.

In order to minimize contention between processors over the critical sections of code, all critical sections are kept as short as possible: typically only a few lines of code. Furthermore, there are different semaphores for each type of resource accessed in critical sections. *res_syscall* is used to interlock all UNIX system calls and some library routines, such as **write**(2), **malloc**(3), **printf**(3), etc. *res_worker* is used by the function worker() to serialize access to the variable cur_pixel, which contains the index of the next pixel to be computed. *res_results* is used by the function view_pixel to serialize access to the result buffer. This is necessary because few processors have hardware multi-processor interlocking on byte operations within the same word. *res_model* is used by the **libspl** spline library routines to serialize operations which cause the model to be further refined during the raytracing process, so that data structures remain consistent.

Application of the usual client-server model of computing would suggest that one stream of execution would be dedicated to dispatching the next task, while the rest of the streams of execution would be used for ray-tracing computations. However, in this case, the dispatching operation is trivial and a ‘‘self-dispatching’’ algorithm is used, with a critical section used to protect the shared variable cur_pixel. The real purpose of the function do_run() is to perform whatever machine-specific operation is required to initiate *npsw* streams of execution within the address space of the **rt** program, and then to have each stream call the function worker(), each with appropriate local stack space.

Each worker() function will loop until no more pixels remain, taking the next available pixel index. For each pass through the loop, RES_ACQUIRE(res_worker) will be used to acquire the semaphore, after which the index of the next pixel to be computed, *cur_pixel*, will be acquired and incremented, and before the semaphore is released, ie,

```
worker() {
        while(1)  {
                RES_ACQUIRE( &rt_g.res_worker );
                my_index = cur_pixel++;
                RES_RELEASE( &rt_g.res_worker );
                if( my_index > last_pixel )
                        break;
                a.a_x = my_index%width;
                a.a_y = my_index/width;
                ...compute ray parameters...
                rt_shootray( &a );
        }
}
```

On the Denelcor HEP H-1000 each word of memory has a full/empty tag bit in addition to 64 data bits. RES_ACQUIRE is implemented using the Daread() primitive, which uses the hardware capability to wait until the semaphore word is full, then read it, and mark it as empty. RES_RELEASE is implemented using the Daset() primitive, which marks the word as full. do_run() starts additional streams of execution using the Dcreate(worker) primitive, which creates another stream which immediately calls the worker() function.

On the Alliant FX/8, RES_ACQUIRE is implemented using the hardware instruction test-and-set (TAS) which tests a location for being zero. As an atomic operation, if the location is zero, it sets it non-zero and sets the condition codes appropriately. RES_ACQUIRE embeds this test-and-set instruction in a polling loop to wait for acquisition of the resource. RES_RELEASE just zeros the semaphore word. Parallel execution is achieved by using the hardware capability to spread a loop across multiple processors, so a simple loop from 0 to 7 which calls worker() is executed in hardware concurrent mode. Each concurrent instance of worker() is given a separate stack area in the ‘‘cactus stack’’.

On the Cray X-MP and Cray-2, the Cray multi-tasking library is used. RES_ACQUIRE maps into LOCKON, and RES_RELEASE maps into LOCKOFF, while do_run() just calls TSKSTART(worker) to obtain extra workers.

2.7. Performance Measurements

An important part of the BRL CAD Package is a set of five benchmark model databases and associated viewing parameters, which permit the relative performance of different computers and configurations to be made using a significant production program as the basis of comparison. For the purposes of this paper, just the "Moss" database will be used for comparison. Since this benchmark generates pixels the fastest, it will place the greatest demands on any parallel processing scheme. The benchmark image is computed at 512x512 resolution.

The relative performance figures for running **rt** in the parallel mode with Release 1.20 of the BRL CAD Package are presented below. The Alliant FX/8 machine was brl-vector.arpa, configured with 8 Computational Elements (CEs), 6 68012 Interactive Processors (IPs), 32 Mbytes of main memory, and was running Concentrix 2.0, a port of 4.2 BSD UNIX. The Cray X-MP/48 machine was brl-patton.arpa, serial number 213, with 4 processors, 8 Mwords of main memory, with a clock period of 8.5 ns, and UNICOS 2.0, a port of System V UNIX. Unfortunately, no comprehensive results are available for the Denelcor

HEP, the only other parallel computer known to have run this code.

Parallel **rt** Speedup -vs- # of Processors				
# Processors	FX/8	(eff)	X-MP/48	(eff)
1	1.00	100%	1.00	100%
2	1.84	92.0%	1.99	99.5%
3	2.79	93.0%	2.96	98.7%
4	3.68	92.0%	3.86	96.5%
5	4.80	96.0%		
6	5.70	95.0%		
7	6.50	92.9%		
8	7.46	93.3%		

The multiple-processor performance of **rt** increases nearly linearly for shared memory machines with small collections of processors. The slight speedup of the Alliant when the fifth processor is added comes from the fact that the first four processors share one cache memory, while the second four share a second cache memory. To date, **rt** holds the record for the best achieved speedup for parallel processing on both the Cray X-MP/48 and the Alliant. Measurements on the HEP, before it was dismantled, indicated that near-linear improvements continued through 128 streams of execution. This performance is due to the fact that the critical sections are very small, typically just a few lines of code, and that they account for an insignificant portion of the computation time. When **rt** is run in parallel and the number of processors is increased, the limit to overall performance will be determined by the total bandwidth of the shared memory, and by memory conflicts over popular regions of code and data.

3. DISTRIBUTED COMPUTATION ON LOOSELY-COUPLED ENSEMBLES OF PROCESSORS

The basic assumption of this design is that network bandwidth is modest, so that the number of bytes and packets of overhead should not exceed the number of bytes and packets of results. The natural implementation would be to provide a remote procedure call (RPC) interface to rt_shootray(), so that when additional subsidiary rays are needed, more processors could potentially be utilized. However, measurements of this approach on VAX, Gould, and Alliant computers indicates that the system-call and communications overhead is comparable to the processing time for one ray/model intersection calculation. This much overhead rules out the RPC-per-ray interface for practical implementations. On some tightly coupled ensemble computers, there might be little penalty for such an approach, but in general, some larger unit of work must be exchanged.

It was not the intention of the author to develop another protocol for remote file access, so the issue of distributing the model database to the **rtsrv** server machines is handled outside of the context of the **remrt** and **rtsrv** software. In decreasing order of preference, the methods for model database distribution that are currently used are Sun NFS, Berkeley **rdist**, Berkeley **rcp**, and ordinary DARPA **ftp**. Note that the binary databases need to be converted to a portable format before they are transmitted across the network, because **rtsrv** runs on a wide variety of processor types. Except for the model databases and the executable code of the **rtsrv** server process itself, no file storage is used on any of the server machines.

3.1. Distribution of Work

The approach used in **remrt** involves a single dispatcher process, which communicates with an arbitrary number of server processes. Work is assigned in groups of scanlines. As each server finishes a scanline, the results are sent back to the dispatcher, where they are stored. Completed scanlines are removed from the list of scanlines to be done and from the list of scanlines currently assigned to that server. Different servers may be working on entirely different frames. Before a server is assigned scanlines from a new frame, it is sent a new set of options and viewpoint information.

The underlying communications layer used is the package (PKG) protocol, provided by the **libpkg** library, so that all communications are known to be reliable, and communication disruptions are noticed. Whenever the dispatcher is notified by the **libpkg** routines that contact with a server has been lost, all unfinished scanlines assigned to that server will be requeued at the head of the ‘‘work to do’’ queue, so that it will be assigned to the very next available server, allowing tardy scanlines to be finished quickly.

3.2. Distribution Protocol

When a server process **rtsrv** is started, the host name of the machine running the dispatcher process is given as a command line argument. The server process can be started from a command in the dispatcher **remrt**, which uses **system**(3) to run the **rsh** program, or directly via some other mechanism. This avoids the need to register the **rtsrv** program as a system network daemon and transfers issues of access control, permissions, and accounting onto other, more appropriate tools. Initially, the **rtsrv** server initiates a PKG connection to the dispatcher process and then enters a loop reading commands from the dispatcher. Some commands generate no response at all, some generate one response message, and some generate multiple response messages. However, note that the server does not expect to receive any additional messages from the dispatcher until after it has finished processing a request, so that requests do not have to be buffered in the server. While this simplifies the code, it has some performance implications, which are discussed later.

In the first stage, the message received must be of type MSG_START, with string parameters specifying the pathname of the model database and the names of the desired treetops. If all goes well, the server responds with a MSG_START message, otherwise diagnostics are returned as string parameters to a MSG_PRINT message and the server exits.

In the second stage, the message received must be of type MSG_OPTIONS or MSG_MATRIX. MSG_OPTIONS specifies the image size and shape, hypersampling, stereo viewing, perspective -vs- ortho view, and control of randomization effects (the ‘‘benchmark’’ flag), using the familiar UNIX command line option format. MSG_MATRIX contains the 16 ASCII floating point numbers for the 4x4 homogeneous transformation matrix which represents the desired view.

In the third stage, the server waits for messages of type MSG_LINES, which specify the starting and ending scanline to be processed. As each scanline is completed, it is immediately sent back to the dispatcher process to minimize the amount of computation that could be lost in case of server failure or communications outage. Each scanline is returned in a message of type MSG_PIXELS. The first two bytes of that message contain the scanline number in network order 16-bit binary. Following that is the 3*width bytes of RGB data that represents the scanline. When all the scanlines specified in the MSG_LINES command are processed, the server again waits for another message, either another MSG_LINES command or a MSG_OPTIONS/MSG_MATRIX command to specify a new view.

At any time, a MSG_RESTART message can be received by the server, which indicates that it should close all it’s files and immediately re-**exec**(2) itself, either to prepare for processing an entirely new model, or as an error recovery aid. A MSG_LOGLVL message can be received at any time, to enable and disable the issuing of MSG_PRINT output. A MSG_END message suggests that the server should commit suicide, courteously.

3.3. Dispatching Algorithm

The dispatching (scheduling) algorithm revolves around two main lists, the first being a list of currently connected servers and the second being a list of frames still to be done. For each unfinished frame, a list of scanlines remaining to be done is maintained. For each server, a list of the currently assigned scanlines is kept. Whenever a server returns a scanline, it is removed from the list of scanlines assigned to that server, stored in the output image, and also in the optional attached framebuffer. (It can be quite entertaining to watch the scanlines racing up the screen, especially when using processors of significantly different speeds). If the arrival of this scanline completes a frame, then the frame is written to disk on the dispatcher machine, timing data is computed, and that frame is removed from the list of work to be done.

When a server finishes the last scanline of its assignment and more work remains to be done, the list of unfinished frames is searched and the next available increment of work is assigned. Work is assigned in blocks of consecutive scanlines, up to a per-server maximum assignment size. The block of scanlines is recorded as the server’s new assignment and is removed from the list of work to be done.

3.4. Reliability Issues

If the **libpkg** communications layer looses contact with a server machine, or if **remrt** is commanded to drop a server, then the scanlines remaining in the assignment are requeued at the head of the list of scanlines remaining for that frame. They are placed at the head of the list so that the first available server will finish the tardy work, even if it had gone ahead to work on a subsequent frame.

Presently, adding and dropping server machines is a manual (or script driven) operation. It would be desirable to develop a separate machine-independent network mechanism that **remrt** could use to inquire about the current loading and availability of server machines, but this has not been done. This would permit periodic status requests to be made and automatic reacquisition of eligible server machines could be attempted. Peterson's Distrib Peterson Distributed Computation System incorporates this as a built-in part of the distributed computing framework, but it seems that using an independent transaction-based facility such as Pistritto's Host Monitoring Protocol (HMP) facility Natalie Muuss BRL VAX UNIX Manual would be a more general solution.

If the dispatcher fails, all frames that have not been completed are lost; on restart, execution resumes at the beginning of the first uncompleted frame. By carefully choosing a machine that has excellent reliability to run the dispatcher on, the issue of dispatcher failure can be largely avoided. However, typically no more than two frames will be lost, minimizing the impact. For frames that take extremely long times to compute, it would be reasonable extend the dispatcher to snapshot the work queues and partially assembled frames in a disk file, to permit operation to resume from the last ‘‘checkpoint’’.

3.5. Distributed remrt Performance

Ten identical Sun-3/50 systems were used to test the performance of **remrt**. All had 68881 floating point units and 4 Mbytes of memory, and all were in normal timesharing mode, unused except for running the tests and the slight overhead imposed by /etc/update, **rwhod**, etc. To provide a baseline performance figure for comparison, the benchmark image was computed in the normal way using **rt**, to avoid any overhead which might be introduced by **remrt**. The elapsed time to execute the ray-tracing portion of the benchmark was 2639 seconds; the preparation phase was not included, but amounted to only a few seconds.

remrt Speedup -vs- # of Processors

	Ratios		Elapsed Seconds			
# CPUs	Theory	Sun-3/50	Theory	Sun-3/50	Total Speedup	Efficiency
1	1.0000	1.0072	2639.0	2658	0.993	99.3%
2	0.5000	0.5119	1319.5	1351	1.953	97.7%
3	0.3333	0.3357	879.6	886	2.979	99.3%
4	0.2500	0.2524	659.7	666	3.949	98.7%
5	0.2000	0.2027	527.8	535	4.916	98.3%
6	0.1666	0.1686	429.8	445	5.910	98.5%
7	0.1429	0.1470	377.0	388	6.778	96.8%
8	0.1250	0.1266	329.9	334	7.874	98.4%
9	0.1111	0.1133	293.2	299	8.796	97.7%
10	0.1000	0.1019	263.9	269	9.777	97.8%

The ‘‘speedup’’ figure of 0.993 for 1 CPU shows the loss of performance of 0.7% introduced by the overhead of the **remrt** to **rtsrv** communications, versus the non-distributed **rt** performance figure. The primary result of note is that the speedup of the **remrt** network distributed application is very close to the theoretical maximum speedup, with a total efficiency of 97.8% for the ten Sun case! The very slight loss of performance noticed (2.23%) is due mostly to ‘‘new assignment latency’’, discussed further below. Even so, it is worth noting that the speedup achieved by adding processors with **remrt** was even better than the performance achieved by adding processors in parallel mode with **rt**. This effect is due mostly to the lack of memory and semaphore contention between the **remrt** machines.

Unfortunately, time did not permit configuring and testing multiple Alliants running **rtsrv** in full parallel mode, although such operation is supported by **rtsrv**.

When **remrt** is actually being used for producing images, many different types of processors can be used together. The aggregate performance of all the available machines on a campus network is truly awesome, especially when a Cray or two is included! Even in this case, the network bandwidth required does not exceed the capacity of an Ethernet (yet). The bandwidth requirements are sufficiently small that it is practical to run many **rtsrv** processes distributed over the ARPANET/MILNET. On one such occasion in early 1986, 13 Gould PN9080 machines were used all over the east coast to finish some images for a publication deadline.

3.6. Performance Issues

The policy of making work assignments in terms of multiple adjacent scanlines reduces the processing requirements of the dispatcher and also improves the efficiency of the servers. As a server finishes a scanline, it can give the scanline to the local operating system to send to the dispatcher machine, while the server continues with the computation, allowing the transmission to be overlapped with more computation. When gateways and wide-area networks are involved (with their accompanying increase in latency and packet loss), this is an important consideration. In the current implementation, assignments are always blocks of three scanlines because there is no general way for the **rtsrv** process to know what kind of machine it is running on and how fast it is likely to go. Clearly, it would be worthwhile to assign larger blocks of scanlines to the faster processors so as to minimize idle time and control traffic overhead. Seemingly the best way to determine this would be to measure the rate of scanline completion and dynamically adjust the allocation size. This is not currently implemented.

By increasing the scanline block assignment size for the faster processors, the amount of time the server spends waiting for a new assignment (termed ‘‘new assignment latency’’) will be diminished, but not eliminated. Because the current design assumes that the server will not receive another request until the previous request has been fully processed, no easy solution exists. Extending the server implementation to buffer at least one additional request would permit this limitation to be overcome, and the dispatcher would then have the option of sending a second assignment before the first one had completed, to always keep the server ‘‘pipeline’’ full. For the case of very large numbers of servers, this pipelining will be important to keep delays in the dispatcher from affecting performance. In the case of very fast servers, pipelining will be important in achieving maximum server utilization, by overcoming network and dispatcher delays.

To obtain an advantage from the pipeline effect of the multiple scanline work assignments, it is important that the network implementations in both the servers and the dispatcher have adequate buffering to hold an entire scanline (typically 3K bytes). For the dispatcher, it is a good idea to increase the default TCP receive space (and thus the receive window size) from 4K bytes to 16K bytes. For the server machines, it is a good idea to increase the default TCP transmit space from 4K bytes to 16K bytes. This can be accomplished by modifying the file /sys/netinet/tcp_usrreq.c to read:

```
int   tcp_sendspace = 1024*16;
int   tcp_recvspace = 1024*16;
```

or to make suitable modifications to the binary image of the kernel using *adb*(1):

```
adb -w -k /vmunix
tcp_sendspace?W 0x4000
tcp_recvspace?W 0x4000
```

The dispatcher process must maintain an active network connection to each of the server machines. In all systems there is some limit to the number of open files that a single process may use (symbol NOFILE); in 4.3 BSD UNIX, the limit is 64 open files. For the current implementation, this places an upper bound on the number of servers that can be used. As many campus networks have more than 64 machines available at night, it would be nice if this limit could be eased. One approach is to increase the limit on the dispatcher machine. Another approach is to implement a special ‘‘relay server’’ to act as a fan-in/fan-out mechanism, although the additional latency could get to be an issue. A third approach is to partition the problem at a higher level. For example, having the east campus do the beginning of a movie, and the west campus do the end would reduce the open file problem. Additionally, if gateways are involved, partitioning the problem may be kinder to the campus network.

3.7. Conclusions

Parallel computing is good.

This paper has shown how it is possible to implement good graphics interfaces within the confines of a single uniprocessor machine. With the adoption of a ‘‘software tools’’ approach when providing data handling capabilities, it was shown how to transparently take advantage of multi-processor machines, and thus realize a speed advantage. Furthermore, the careful selection of file formats permitted realizing a further speed advantage in the accomplishment of a single task by utilizing multiple systems located across a network.

Carying the theme of increased speed further, multi-processor sytems were examined as a vehicle for making single image generation tools operate faster. An important result was to note that when operation in a shared memory parallel environment was an initial design goal, the implementation of concurrently reentrant code did not significantly increase the complexity of the software. Having code with such properties allows direct utilization of nearly any shared memory multiprocessor with a minimum of system-specific support, namely the RES_ACQUIRE and RES_RELEASE semaphore operations, and some mechanism for starting multiple streams of execution within the same address space.

Finally, collections of processors connected only by conventional speed network links are considered as the final environment for making a single tool operate faster using multiprocessing. It was shown that network distributed computing need not be inefficient or difficult. The protocol and dispatching mechanism described in the preceding sections has been shown to be very effective at taking the computationally intensive task of generating ray-traced images and distributing it across multiple processors connected only by a communications network. There are a significant number of other application programs that could directly utilize the techniques and control software implemented in **remrt** to achieve network distributed operation. However, the development and operation of this type of program is still a research effort; the technology is not properly packaged for widespread, everyday use. Furthermore, it is clear that the techniques used in **remrt** are not sufficiently general to be applied to all scientific problems. In particular, problems where each ‘‘cell’’ has dependencies on some or all of the neighboring cells will require different techniques.

Massive proliferation of computers is a trend that is likely to continue through the 1980s into the 1990s and beyond. Developing software to utilize significant numbers of network connected processors is the coming challenge. This paper has presented a strategy that meets this challenge, and provides a simple, powerful, and efficient method for distributing a significant family of scientific analysis codes across multiple computers.

Acknowledgements

The author would like to thank Dr. Paul Deitz for providing his unflagging support and encouragement, Phil Dykstra, Paul Stay, Gary Moss, Chuck Kennedy, and Bob Reschly for the long hours as we designed and built this software, and Dr. Dave Rogers for once again persuading me to write it all down.

The following strings which have been included in this paper are known to enjoy protection as trademarks; the trademark ownership is acknowledged in the table below.

Trademark	Trademark Owner
Cray	Cray Research, Inc
Ethernet	Xerox Corporation
FX/8	Alliant Computer Systems Corporation
IBM 370	International Business Machines Corporation
Macintosh	Apple Computer, Inc
MacPaint	Apple Computer, Inc
NFS	Sun Microsystems, Inc
PowerNode	Gould, Inc
ProNet	Proteon, Inc
Sun Workstation	Sun Microsystems, Inc
SunView	Sun Microsystems, Inc
UNIX	AT&T Bell Laboratories
UNICOS	Cray Research, Inc

VAX	Digital Equipment Corporation
VaxStation	Digital Equipment Corporation
X Window System	Massachusetts Institute of Technology