Hiding the Disk and Network Latency of Out-of-Core Visualization

David Ellsworth*
AMTI / NASA Ames Research Center

Abstract

This paper describes an algorithm that improves the performance of application-controlled demand paging for out-of-core visualization by hiding the latency of reading data from both local disks or disks on remote servers. The performance improvements come from better overlapping the computation with the page reading process, and by performing multiple page reads in parallel. The paper includes measurements that show that the new multithreaded paging algorithm decreases the time needed to compute visualizations by one third when using one processor and reading data from local disk. The time needed when using one processor and reading data from remote disk decreased by two thirds. Visualization runs using data from remote disk actually ran faster than ones using data from local disk because the remote runs were able to make use of the remote server's high performance disk array.

1 Introduction

Simulations run on large parallel systems produce large data sets having hundreds of megabytes to terabytes of data. These data sets cannot usually be visualized on the system that produced them because it is reserved for simulation runs. Instead, the visualization is typically performed on workstations. The workstations usually do not have sufficient memory to load the entire data set, which means that out-of-core visualization techniques must be used. These techniques calculate the visualization with only a fraction of the data set resident in memory. In addition, many data sets are so large that they only fit on central file servers. Since most file servers do not have significant extra CPU and memory capacity, remote out-of-core visualization is required.

One method for performing out-of-core visualization is application-controlled demand paging [1]. This is similar to the demand paging used in virtual memory systems, but it is built into the application instead of the operating system. Demand paging takes advantage of the fact that most visualization calculations only touch a small fraction of the data set. The main advantage of application-controlled paging over operating system demand paging is that it allows each page of data to hold a 3D cube of data, which experiments have shown to reduce the amount of data required to compute a visualization by about half [1]. Another advantage is that it reduces the swap and address space required for the application.

However, the original implementation of out-of-core visualization using demand paging did not try to perform computation and disk access at the same time. While the operating system's disk caching and read ahead did overlap disk access and computation, the amount of overlap was small. In addition, the original implementation only had one disk request outstanding at a time. This meant that the operating system could not optimize use of the disk by reordering the requests to reduce seek time, or by issuing concurrent requests to different drives in RAID disk subsystems. Finally, overlapping computation and disk access is even more important when the disk is accessed across the network since the network adds latency.

This paper increases the amount of overlap of computation and disk accesses by dividing the visualization into a number of tasks, and then running the tasks using a pool of worker threads. A scheduler initially runs one thread per processor. When a thread needs to read data from disk, it is blocked, and the scheduler allows another thread to run. The blocked thread is restarted after the data has been read and a processor becomes available. A separate pool of reader threads request data pages from the operating system and wait for the requests to complete. If the data set is on local disk, the reader threads run as part of the application; if the data set is on a remote server, the threads run on that server.

This new multithreaded demand paging algorithm has several advantages other than its increased performance. First, while a visualization algorithm must be modified to take advantage of the overlapped disk access and computation, the modification is useful in itself: the algorithm must be parallelized. The number of tasks generated may need to be higher than with in-core visualizations, but number of tasks is usually easy to change since the number should be varied with the number of processors. A second advantage of the new algorithm is that it is not tied to a particular visualization algorithm; instead, it can be used to accelerate a number of algorithms.

The new algorithm's last advantage is that it is compatible with time-critical visualization [2], which is where the time to compute a visualization is limited to guarantee a specified frame rate. With time-critical visualization, each visualization object stops its computation after its time budget has been exceeded. To do this, each object must have a fairly accurate estimate of the CPU time used. Some operating systems, such as Unix, record the amount of CPU time that a thread uses, but the granularity of the CPU time is too coarse for interactive visualization applications. Instead, because the algorithm only schedules one thread per processor, and because most systems have a high-resolution real time clock, the amount of elapsed wall-clock time should be an acceptable estimate of the elapsed CPU time. We hope to extend our implementation to support time-critical visualization in the future.

2 Related Work

In addition to demand paging algorithms, out-of-core visualization algorithms include streaming algorithms and indexing algorithms. Streaming algorithms read the entire data set, but read it in small pieces that will fit into memory. Once one piece has been brought into memory, the computation is run over that portion of the data. Further pieces are read and processed until the visualization has been computed for the entire data set. Law *et al.* [3] describes a general streaming architecture. The UFAT batch visualization program [4] also performs streaming on time varying data sets. When the visualization only accesses a small fraction of the data set, a streaming algorithm can be slower than a demand paging algorithm if the streaming algorithm does not avoid reading all of the data.

The second type of out-of-core visualization algorithms is indexing algorithms. Many indexing algorithms have been described for isosurface computation [5, 6, 7]. These algorithms precompute an

*NASA Ames Research Center, Mail Stop T27A-2, Moffett Field, CA 94035 (ellswort@nas.nasa.gov)

index that identifies the portion of the data that is necessary to compute the requested visualization. For isosurface computation, the index identifies which cells contain portions of the isosurface. One disadvantage of many index algorithms is that their index is specific to the visualization algorithm.

Some non-visualization algorithms have similarities to this work. Flight simulation [8] and walkthrough algorithms [9] store their geometry on disk, and only keep the geometry which is inside the viewing frustum resident in memory. These algorithms can hide the disk latency by prefetching the geometry that will soon move inside the viewing frustum. The prefetching is possible because the viewer's expected position can be computed by extrapolating the user's position using his or her last view positions. Prefetching is harder with visualization computations because it is often impossible to predict how the algorithm will traverse the data set.

3 Application-Controlled Demand Paging

The basic idea of demand paging for visualization starts with logically breaking the data set into fixed size pages. When a file is opened, only enough header information is read to set up data structures for tracking the pages that have been loaded into memory. During execution, when a data value is needed, the page number is computed. If the page is in memory, the requested value is returned. Otherwise, memory is allocated for the page, the page is read, and the requested value returned. Because the implementation uses a fixed-size memory pool for page storage, allocating a page when the pool is full involves reallocating, or *stealing*, the memory used by another page.

The technique used for dividing the data set into page impacts the performance. This paper's implementation uses a *cubed* page format for paging structured grid files (unstructured grids are not supported). The original 3D arrays of data are broken into a series of pages, each page containing an 8x8x8 cube, or 2 KB, of the original data. Using a cube of data instead of the original array order reduces the numbers of page that must be read because, if the original data was simply broken into page without changing the layout, each page would contain a plane or slab of data. For most directions of traversal, a larger fraction of the data in a page is used when traversing a cube of data instead of a plane of data. The page size should be the best compromise between having large pages, which decreases the cost of reading each byte, and smaller pages, which retrieve a smaller amount of unnecessary data. The 8x8x8 page size had the best performance in experiments described in the earlier demand paging paper [1].

The cubed page format requires that files be converted to a new file format before the visualization process. For the three data sets used as examples in Section 6, their converted files would require an additional 19 to 30% of storage if partially-filled pages were padded to the full page size when written to disk. Because 19% of a large file is still large, partially filled pages are not padded on disk. These pages are expanded to full size when they are loaded into memory to allow the run-time data access code to be simpler and faster.

When a new page must be read when the memory pool is full, an existing memory block must be stolen. The paging module allocates a block that has not been used recently by associating a *referenced* bit with every page in memory. The referenced bit is set when a page is referenced. When a page must be stolen, the in-memory pages are scanned, looking for one with a cleared referenced bit. The referenced bit of a page is cleared as it is examined during the scanning, which means that a page is reallocated if it has not been accessed after two scanning passes have completed. This algorithm is similar to ones used for virtual memory page replacement in operating systems [10].

3.1 Field Encapsulation Library

The paging system is part of the Field Encapsulation Library [11]. This library encapsulates the management of field data for different grids, such as regular, structured curvilinear, multiblock, and unstructured grids. It provides a grid-independent interface by placing all the grids types in a C++ class hierarchy and using polymorphism to direct requests to the correct functions at run time. Because paged grids and fields are also defined in this class hierarchy, visualization algorithms do not need to be modified to use paged files.

FEL retrieves data from the paging system either vertex at a time or a 2x2x2 group of vertices at a time. Each request can be for all or part of the data (coordinates, solution data) stored at the vertex or cell vertices. Being able to retrieve multiple values with one function call reduces the cost of translating the i, j, k lattice coordinates to page number and offset. One consequence of this interface is that a single retrieval request can cause a number of pages to be read if multiple values are requested or a 2x2x2 request falls on a page boundary.

4 Multi-Threaded Demand Paging

The multi-threaded demand paging algorithm halts a computation when it requires a page that is not resident and attempts to run another computation while the page is being read. This is done by using a simple, high-level multitasking library called the Abstract Multitasking Library (AML).

An application uses AML to compute a visualization by creating a number of tasks. Typically, each task represents a complete or partial visualization object, such as a single streamline or a single grid surface. Each task is a C++ object that holds enough information to identify the work to be done, and has a method that is called to do the computation. For example, the task object might hold pointers to a streamline's seed point and the velocity field within which the point will be integrated, and define a function that calls an existing streamline integration function.

Once the application creates the tasks, it places them in an AML task group, which is a simple list of tasks. Then, the application uses AML to initialize a pool of worker threads, and tells AML to use the pool to compute the visualizations in the task group. At the start of the computation, AML first assigns a task from the task groups to each thread, and starts the threads running. One thread is started for each processor that will be used. Each thread then works independently on its assigned task until it finishes the task or finds that the it needs a page of data that is not memory resident. If a thread finishes the task, it uses AML to find another task in the task group to compute. If a thread needs a page of data, it requests that the page be read by a reader thread; this process is described below. After placing the read request in the queue, the thread sees if another thread is waiting to get use of a processor. If so, the first thread wakes up the other thread before going to sleep. If there is no waiting thread, the first thread will check to see if there are remaining tasks in the task group as well as an idle thread in the thread pool. If so, the first thread wakes up the idle thread before going to sleep. The idle thread then starts work on the next task in the task group.

The algorithm just described is a thread scheduler; it is similar to the ones built into operating systems. The scheduler attempts to keep one thread running on each processor by only having one thread per processor in a runnable state; that is, one thread that is not blocked. This means that a thread is not always immediately restarted after a page is read for it. The thread is immediately restarted if the scheduler sees that there are fewer running threads than processors. However, if every processor has a thread to run,

the now-ready-to-run thread is placed in a queue to wait until a processor becomes available.

The scheduler does not use any special operating system functions to manage its pool of worker threads. Instead, it uses standard interprocess communication mechanisms such as condition variables. If a thread is to be blocked, it waits on a condition variable, which causes it to stop execution. The scheduling mechanism does assume that, if only one thread per processor is not blocked, the operating system is smart enough to run each of the threads on a separate processor. Our experience is that this works reasonably well on Irix systems if the `sproc` threading library is used. The pthreads multitasking library gives lower performance. A possible explanation for the low performance is that, if the pthreads package does its own thread scheduling outside the kernel (as is typical), the pthreads scheduler interacts unfavorably with the AML scheduler.

4.1 Parallelizing Demand Paging

The parallel paging algorithm has a few differences from the serial paging algorithm described above. The changes fall in three categories:

Page access. Each access to a page to retrieve data must be done atomically. Otherwise, one thread could verify that a page was present, a second thread could steal that page's storage and read a new page into it, and then the first thread could retrieve the new page's data by mistake. The problem is eliminated by serializing access to the page with a mutual exclusion lock. However, allocating one lock per page in the file is impractical since there may be tens of millions of pages mapped at once. Instead, the implementation uses one lock for a group of 48 to 80 pages, depending on the type of file.

Reading a page. When a thread finds that a page is not present, instead of finding memory for the page and reading the page from disk, it instead finds the memory and puts a request into a queue for the page to be read. If the thread needs more than one page, it allocates memory and puts a request into the read queue for each page. Then, the thread waits on a condition variable for the reads to be completed.

A separate pool of reader threads takes requests from the read queue, reads the page, and unpacks the page if necessary. When a reader thread finishes a read, it checks whether all of the worker's requests have been completed. If so, the worker thread is restarted if the scheduler indicates that a processor is available, and the thread is marked as "ready to run" otherwise.

Corner cases. The page allocation code needs to be modified to insure that only one thread allocates a page at a time using a lock for scanning the page table, and also the per-page locks. Also, the page reading code must handle having more than one worker request the page at the same time. This is handled by keeping a list of pages that are being read, and checking the list before adding a request to the read queue.

4.2 Remote Demand Paging

Remote demand paging could be performed using a distributed file system, such as the Network File System (NFS). However, they do not provide the same performance that a specialized paging server, as shown below. One reason for the lower performance of NFS is that it only sends blocks that are aligned on regular block boundaries. Because paged files have arbitrarily sized pages, the protocol will return more data than are necessary. In addition, some NFS implementations may require more context switches and copying of data compared to what can be achieved with a specialized client and server.

The remote paging server is a simple application that communicates with the local paging library using a TCP socket. The server supports three primary operations: Open, which opens a file and returns a file handle; Read, which reads and returns data given a file handle, an offset, and size; and Close, which closes the file specified by a file handle. The server does not support writing.

When a worker thread discovers that a page from a remote paged file is not memory resident, it puts the request in the read queue, and also sends a read request to the remote server. The remote server application has a pool of reader threads that constantly take incoming read requests from the socket, perform the reads, and return the requested data via the socket. The reader threads serialize reading and writing to and from the socket using a pair of semaphores. A single local reader thread waits for results coming from the remote server, and matches the returned data to a request in its read queue. Then, the thread reads the data from the socket and unpacks the page if necessary. The final step is to wake up the requesting worker thread if a processor is available. Because the responses can come back in any order, each request and response has a sequence number identifying it.

To allow reasonable performance, the TCP socket must have the TCP_NODELAY option enabled. If the option is not enabled, the performance on Irix systems is much lower. This happens because the TCP protocol code will hold on to a read request message for a while due to a desire to combine multiple small messages into a single large one.

5 Implementation

The local and remote demand paging algorithms just described have been implemented a batch visualization program called `batchvis`. This program uses FEL and the VisTech [12] visualization library. This program allows the user to compute a set of visualizations for each time step in the visualization. Currently, the program supports particle tracing (streamlines, streaklines, and pathlines) as well as the extraction of surfaces of the grid. Additional visualization methods will be supported in the future. The visualizations can be optionally colored by using one of several standard functions.

A non-threaded version of FEL and `batchvis` can be created using compile time flags that replace the threaded portions of the code with the older, non-threaded versions. The serial version of the remote paging code is similar to the parallel version, but only allows synchronous requests to the server. The experiments described below give timings with this version to show the improvements due to the new algorithms. The threaded version of `batchvis` uses SGI `sproc`-style threads instead of the pthreads package because the former gives better performance.

6 Experimental Methodology

We evaluated the multi-threaded demand paging algorithm's performance by measuring the time required to compute a visualization for several different configurations. The performance was measured for different data sets, different locations of the data (local or remote), and for different algorithm parameters.

The experiments were run under the following conditions. The runs were made on the three systems described in Table 1; all ran the Irix 6.5 operating system. We used two remote servers due to disk space limitations. The systems were connected by an 800 Mbit/second HIPPI TCP network. While HIPPI networks are fairly exotic, the performance should be similar on the more common Gigabit Ethernet since the remote protocol does not use HIPPI's large packet capability. All of the runs set the size of the memory pool used to hold data pages to 200 MB.

The remote server's large memory and processor configurations were largely unused during the runs since very little processing was

How Used	System Type	Number of Processors	Memory Size	Data Storage
Computation, local storage	Onyx	4	1 GB	4 striped disks
Remote SSLV, F18 storage	Onyx2	8	3.5 GB	RAID Array
Remote Harrier storage	Onyx	8	5 GB	RAID Array

Table 1: Systems used for experiments.

Data Set	Number of Time Steps	Grid Size (MB)	Solution Size (MB)	Total Size (GB)	Percent Read
SSLV	1	254.0	317.5	0.56	7.9
F18	150	26.9	33.6	4.96	1.7
Harrier	1600	54.9	68.6	107.4	7.7

Table 2: Data set statistics.

necessary, and because all of the machines had their operating system's file cache flushed before each run. The cache was flushed by running a program that allocated as much memory as possible, which takes memory away from the file cache, and then reading a different file in random order. In addition, five copies of the SSLV and F18 data sets were placed on the remote server. Consecutive runs rotated through the data set copies.

We used the three data sets shown in Figures 5 to 7 for the performance timings. Table 2 contains statistics about the data sets. The data sets are:

SSLV. This data set is of the Space Shuttle Launch Vehicle flying at mach 1.25. This steady simulation was computed in order to have a more accurate simulation of the shuttle aerodynamics compared to earlier simulations, and enabled more accurate engineering analyses. The visualization contains several streamlines showing the airflow between the external tank, the solid rocket booster, and the orbiter. The streamlines are colored by the local density value.

F18. This data set shows the F18 flying at a 30-degree angle of attack. The simulation was performed to analyze the interaction of the vortex formed over the leading-edge extension with the vertical stabilizer. The visualization injects particles into the center of the vortex, and colors them according to the local density value.

Harrier. The Harrier data set shows the Harrier flying slowly 30 feet above the ground. The simulation is part of research into the cause of oscillations seen when the jet is flying at this level. The visualization shows particles injected into the jet exhausts, which shows the structure of the ground vortices created by the exhaust. The particles were injected every third time step to reduce the computation requirements, and are colored according to the local pressure. Because the visualization takes over an hour to compute, only a few performance runs for the Harrier are shown below.

Different sets of runs explored the following variables:

Data set access. Runs accessing a local copy of the data show the performance of the local demand paging algorithm. Different runs compared the performance of accessing remote data using the custom paging protocol and the standard NFS protocol.

Number of processors. Some runs show the basic performance of the algorithms, when they are run on a single processor. Other runs used all four of the system's processors, which shows the amount of speedup possible. It would be unreasonable to expect linear speedups because the disk and network performance did not change. The single processor runs used the Irix `runon` command to restrict all threads to a single processor.

Number of reader and worker threads. Different runs show how the amount of computation and disk access concurrency affects performance.

Data	Local Disk	Remote NFS	Remote Server
SSLV	30.3	38.1	46.6
F18	146	164	163

Table 3: Non-threaded times, in seconds.

Data Access	Number Worker Threads	Number of Reader Threads 1	4	16
Remote NFS	4	176	178	171
	16	169	169	162
Remote server	4	118	116	89.1
	16	109	103	77.9

Table 4: Harrier timings, in minutes.

7 Results

The results are shown in Tables 3 to 6. Each row in the SSLV and F18 tables shows values for a constant number of worker threads per processor (WTPP). This means that the timings in the left half of each row are for either 1, 4, or 8 worker threads, and the right half times are for 4, 16, or 32 worker threads. All of the timings are from single run, which means run-to-run variations are expected. The F18 timings are also shown in chart form in Figures 1 to 4. The curves for the SSLV timings are similar, and are omitted to save space. Much of the discussion below refers to the best run for a configuration; this happens to be the runs using the largest number of threads tried: 16 reader threads and 8 worker threads per processor.

Overall, the results show that, for both remote and local paging, the multithreaded paging approach is substantially faster than the previous serial paging approach, even when the multithreaded paging is run on a single processor. The visualization computation times decrease by about 30% for local paging. For remote paging, the times decrease by 50 to 66%—a two to three times speedup. Using the multithreaded remote paging server to compute a visualization was also substantially faster than using NFS: the custom paging server took between 40 to 64% less time. The remainder of this section examines the results in more detail.

Local data performance. Using the threading library decreased the run time by about one third when the data are on local disk. The best single-processor threaded time for the SSLV was 20.3 seconds, while the serial implementation took 30.3 seconds. With the F18, the time decreased from 146 seconds for the serial version's run to 94.4 seconds for the best single-processor threaded run.

Using more processors did not speed up the fastest SSLV run: the best single-processor time was 20.3 seconds, while the best four-processor time was 19.8 seconds. The difference between the run times is below the margin of error. The F18 did run faster with four processors: the best time decreased from 94.4 to 76 seconds. It is not surprising to see no or a small speedup since the disk can be the limiting factor. The decrease in time for the F18 does show that these multithreading techniques will make good use of multiprocessor systems if there is sufficient disk bandwidth.

Varying parameters with local data. Figures 1 and 2 show the general trends when the number of worker and reader threads is varied. Increasing the number of worker or reader threads usually increases the performance if you allow for run-to-run variations. Using eight instead of four worker threads per processor does not

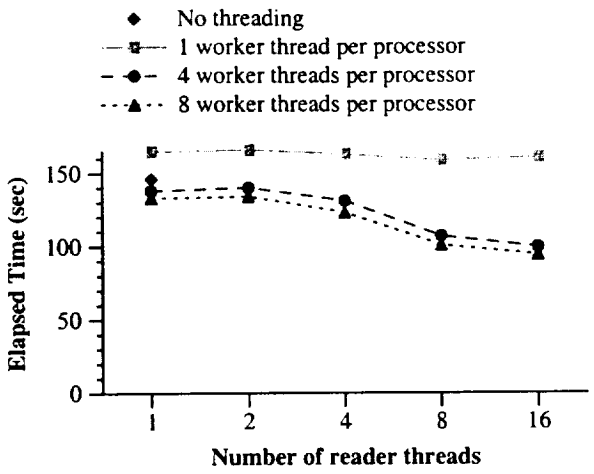

Figure 1: F18 timings using data from local disk and one processor.

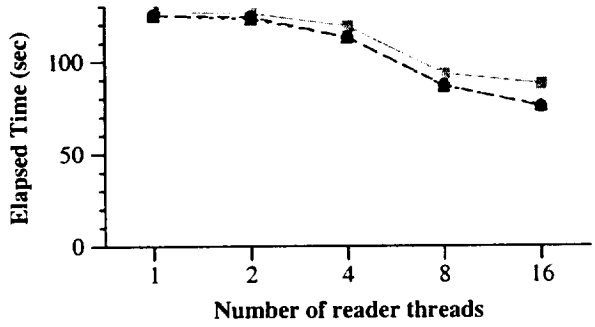

Figure 2: F18 timings using data from local disk and four processors.

result in a large speed increase. Adding more reader threads beyond the 16 threads tested should increase the performance, but 16 threads looks to be near the point of diminishing returns.

The main exception to the above trend is seen when a single processor and a single worker thread are used: adding reader threads does not increase performance. This is not surprising because the pages that the single worker thread requests will usually be adjacent, which means that the disk caching done by the operating system allows the one reader thread to give good performance.

Remote data versus local data. One surprising result with the remote timings is that the fastest remote runs were faster than the corresponding fastest local data runs. For example, the fastest local timing for the F18 was 94.4 seconds, while the fastest remote time is 76 seconds. This can be explained by the higher performance disk subsystem on the remote server: it has a RAID array with dozens of disk drives, while the local disk subsystem has only four striped disks. This speedup will likely be seen in production usage of threaded demand paging because central file servers usually have a better storage system than a personal workstation.

Remote access speedup. The remote single-processor multithreaded paging times were two to three times faster than the original serial paging code. The best single-processor multithreaded time for the SSLV was 15.8 seconds, which is nearly three times faster than the 46.6 seconds needed for the serial paging version. The corresponding F18 times are 82 and 163 seconds.

NFS performance with threading. The fastest runs that used NFS with multiple reader threads were only somewhat faster than the serial NFS times. The SSLV serial run took 46.6 seconds, which decreased to 43.3 seconds with threading and a single processor,

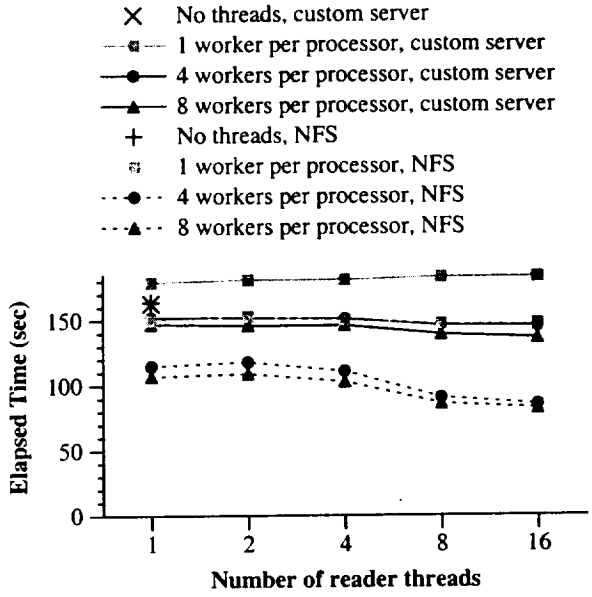

Figure 3: F18 timings using data from remote disk and one processor.

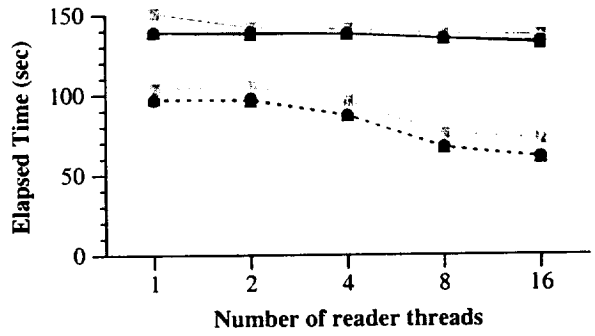

Figure 4: F18 timings using data from remote disk and four processors.

and 41.4 seconds with four processors. The respective times for the F18, 164, 136.4, and 132 seconds, show a moderate performance increase. Figures 3 and 4 show that adding additional reader threads does not change the NFS performance appreciably, and that adding worker threads only results in a small speed increase.

NFS versus remote server. Accessing remote data using the demand paging server was much faster than using NFS. With the SSLV data set, the fastest one-processor time was 43.3 seconds when using NFS, and 15.8 seconds with the demand paging server, a speedup of nearly three. The fastest one-processor remote server F18 run took 82 seconds, 40% less time than the fastest one-processor NFS run, which took 136 seconds. Finally, the fastest Harrier run using the threaded server (77.9 minutes) took less than half the time required when using NFS (162 minutes).

The performance increase was larger when using four processors because the NFS four-processor runs took about the same time as the single-processor runs. The fastest four-processor demand paged server run with the F18 was 60.7 seconds, while the corresponding single-processor run was 82.2 seconds. This shows that accessing data remote data with the threaded server scales better than accessing data with NFS.

Data Access	Num. WTPP	1 Processor					4 Processors				
		1 RT	2 RT	4 RT	8 RT	16 RT	1 RT	2 RT	4 RT	8 RT	16 RT
Local Disk	1	33.4	34.1	33.8	33.7	34.3	33.8	33.5	25.9	21.9	21.1
	4	34.3	34.8	26.7	22.6	21.7	36.2	34.9	26.2	21.2	19.0
	8	35.1	34.9	26.2	22.6	20.3	34.7	34.2	25.8	22.9	19.8
Remote via NFS	1	46.6	47.1	48.1	48.4	48.5	40.9	45.5	45.3	44.4	43.5
	4	44.6	45.5	45.9	44.1	43.3	45.8	47.4	44.6	43.8	42.3
	8	45.0	40.7	44.5	42.9	42.6	43.9	45.2	44.4	43.2	41.4
Remote via Server	1	34.5	36.9	35.0	35.0	35.3	30.5	30.0	21.7	18.1	17.0
	4	31.4	30.6	22.4	18.5	17.4	28.7	27.9	19.6	16.1	14.5
	8	30.3	30.1	21.7	17.7	15.8	29.4	25.2	18.6	15.7	14.0

Table 5: SSLV timings, in seconds. Key: WTPP = worker threads per processor, RT = number of reader threads.

Data Access	Num. WTPP	1 Processor					4 Processors				
		1 RT	2 RT	4 RT	8 RT	16 RT	1 RT	2 RT	4 RT	8 RT	16 RT
Local Disk	1	165	166	163	159	161	127	126	119	93.2	87.8
	4	138	140	131	107	99.6	125	124	113	86.8	75.4
	8	133	134	123	101	94.4	125	123	113	86.4	76.0
Remote via NFS	1	179	181	181	183	183	151	142	141	138	137
	4	152	152	151	146	145	139	139	138	135	133
	8	147	146	146	139	136	139	138	138	135	132
Remote via Server	1	150	153	150	147	147	104	105	96.3	75.7	72.5
	4	115	118	111	90.6	85.2	96.6	97.6	87.2	67.4	61.1
	8	107	109	103	86.0	82.2	97.7	96.4	87.2	67.0	60.7

Table 6: F18 timings, in seconds. Key: WTPP = worker threads per processor, RT = number of reader threads.

8 Summary and Future Work

This paper has described an approach that improves the performance of application-controlled demand paging for out-of-core visualization by better overlapping the computation with the page reading process. It does this by using a pool of worker threads that perform the visualization computation, and a separate pool of reader threads to perform the page reads. A scheduling module manages the worker threads so that only one worker runs per processor. Measurements show that the multithreaded paging algorithm decreases the time needed to compute visualizations by one third when using one processor and reading data from local disk. The time needed when using one processor and reading data from remote disk decreased by two thirds, in part due to the high performance of the remote server's disks. By using the new remote paging algorithm instead of using NFS for remote paging, the computation time decreased time by 40 to 63%, depending on the data set.

The performance increases described in this paper make out-of-core visualization using local and remote demand paging more attractive. In addition, the fact that remote demand paging can be faster than local paging could, in some settings, even replace out-of-core demand paging from local disk with paging from remote disk.

One direction of future work would be run experiments with larger worker and reader thread pools to determine whether there is a point at which adding threads decreases the performance. A second direction would be to implement an interactive time-critical visualization system in order to gauge the effectiveness of the time-critical support built into the new multithreaded paging algorithm. A third direction would be to evaluate the performance of remote demand paging over a wide area network instead of over a local area network.

Acknowledgments

This work was supported by NASA contract DTTS59-99-D-00437/A61812D. I am grateful to Raynaldo Gomez and Fred Martin for providing the SSLV data; Ken Gee and Scott Murman for the F18 data; and Jasim Ahmad, Neal Chaderjian, Scott Murman, and Shishir Pandya for the Harrier data. I would also like to thank Pat Moran for his last-minute editing assistance, and Tim Sandstrom for his help with the VisTech library.

References

[1] Michael B. Cox and David Ellsworth. Application-controlled demand paging for out-of-core visualization. In Roni Yagel and Hans Hagen, editors, *IEEE Visualization '97*, pages 235–244. IEEE, October 1997.

[2] Stephen T. Bryson and Sandra Johan. Time management, simultaneity and time-critical computation in interactive unsteady visualization environments. In *IEEE Visualization '96*. IEEE, October 1996. ISBN 0-89791-864-9.

[3] C. Charles Law, William J. Schroeder, Kenneth M. Martin, and Joshua Temkin. A Multi-Threaded streaming pipeline architecture for large structured data sets. In David Ebert, Markus Gross, and Bernd Hamman, editors, *IEEE Visualization '99*, pages 225–232. IEEE, October 1999.

[4] David Lane. UFAT: A particle tracer for time-dependent flow fields. In *IEEE Visualization '94*, pages 225–232. IEEE, October 1994.

[5] Philip M. Sutton and Charles D. Hansen. Isosurface extraction in time-varying fields using a temporal branch-on-need tree (T-BON). *IEEE Visualization '99*, pages 147–154, October 1999.

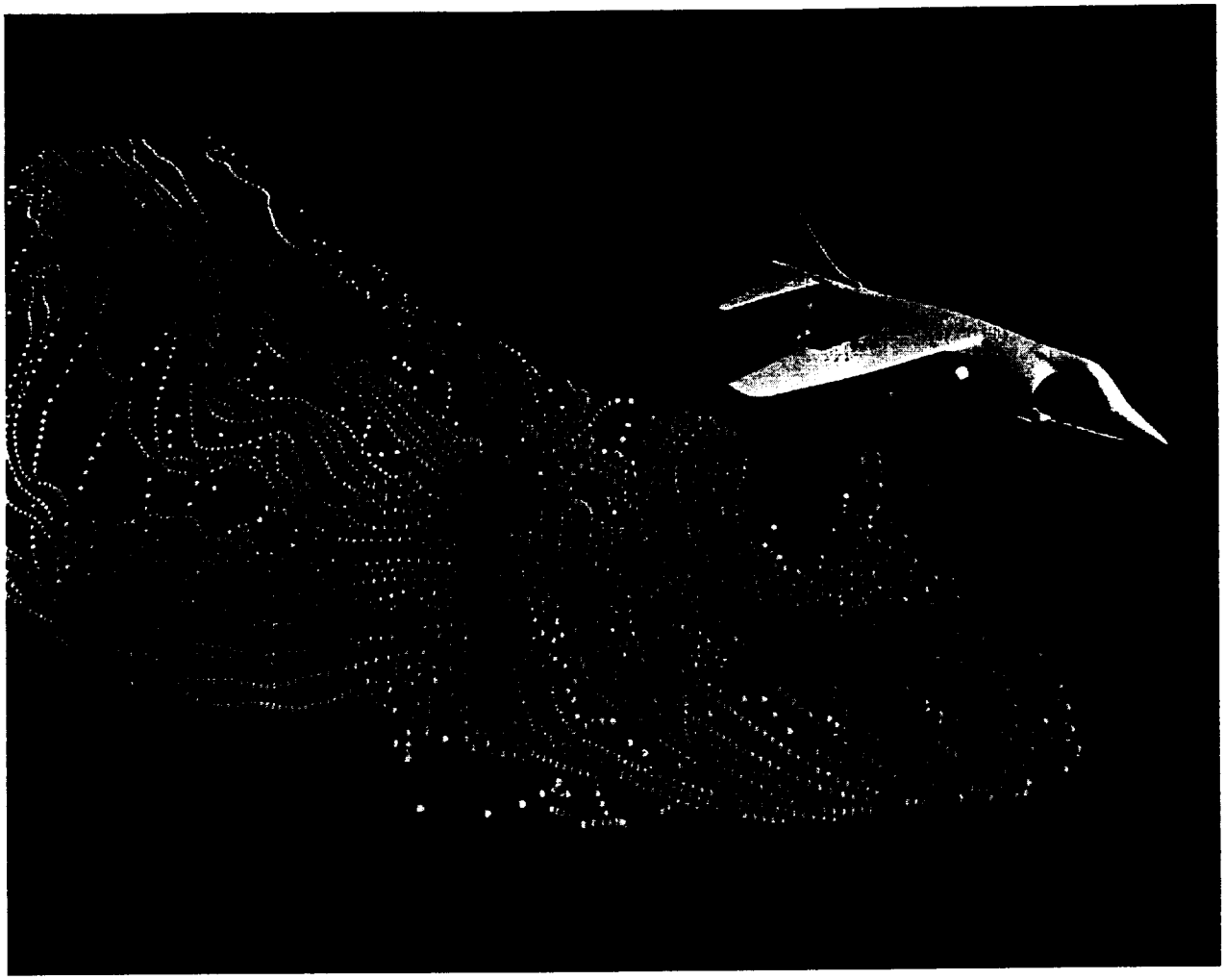

Figure 5: Visualization of the Harrier data set.

[6] Yi-Jen Chiang and Cláudio T. Silva. I/O optimal isosurface extraction. In Roni Yagel and Hans Hagen, editors, *IEEE Visualization '97*, pages 293–300. IEEE, November 1997.

[7] Yi-Jen Chiang, Cláudio T. Silva, and William J. Schroeder. Interactive out-of-core isosurface extraction. *IEEE Visualization '98*, pages 167–174, October 1998. ISBN 0-8186-9176-X.

[8] B. J. Schachter. Computer image generation for flight simulation. *IEEE Computer Graphics and Applications*, 1:29–68, October 1981.

[9] Thomas A. Funkhouser. Database management for interactive display of large architectural models. In Wayne A. Davis and Richard Bartels, editors, *Graphics Interface '96*, pages 1–8. Canadian Information Processing Society, Canadian Human-Computer Communications Society, May 1996. ISBN 0-9695338-5-3.

[10] Abraham Silberschatz and Peter Baer Galvin. *Operating System Concepts*. John Wiley and Sons, 5th edition, 1998.

[11] Patrick Moran, Chris Henze, and David Ellsworth. The FEL 2.2 user guide. Technical Report NAS-00-002, NAS Systems Division, NASA Ames Research Center, January 2000.

[12] Han-Wei Shen, Tim Sandstrom, David Kenwright, and Ling-Jen Chiang. *VisTech Library User and Programmer Guide*. National Aeronautics and Space Administration, 1999.

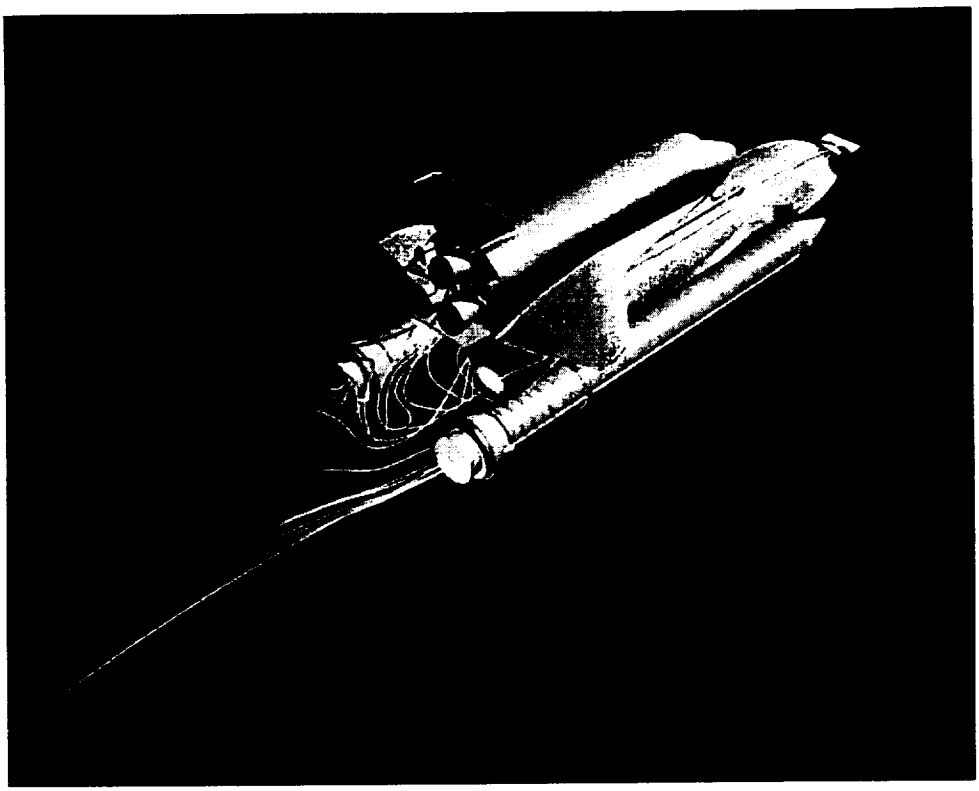

Figure 6: Visualization of the SSLV data set.

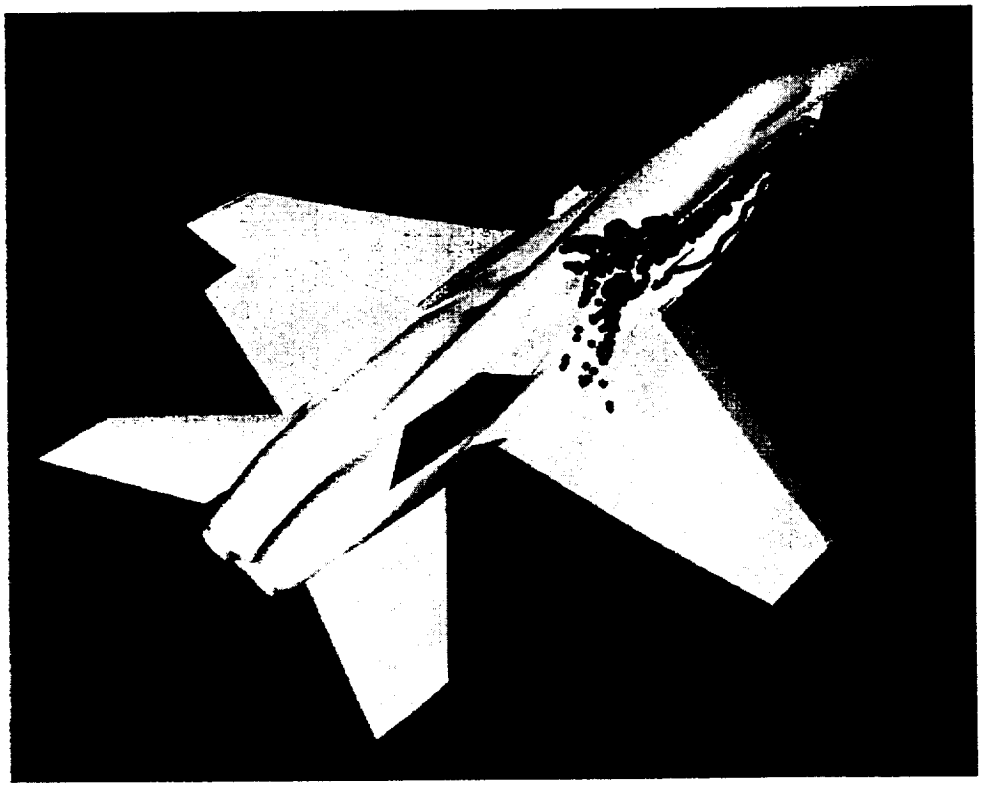

Figure 7: Visualization of the F18 data set.