Accelerating Demand Paging for Local and Remote Out-of-Core Visualization

David Ellsworth

NAS Technical Report NAS-01-004
June 2001

ellswort@nas.nasa.gov
Advanced Management Technology, Inc.
NASA Ames Research Center, M/S T27A-2
Moffett Field, CA, 94035, USA

Abstract

This paper describes a new algorithm that improves the performance of application-controlled demand paging for the out-of-core visualization of data sets that are on either 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 new algorithm can be applied to many different visualization algorithms since application-controlled demand paging is not specific to any visualization algorithm. 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 up to 60%. Visualization runs using data from remote disk ran about as fast as ones using data from local disk because the remote runs were able to make use of the remote server's high performance disk array.

Accelerating Demand Paging for Local and Remote Out-of-Core Visualization

David Ellsworth*
Advanced Management Technology, Inc.
NASA Ames Research Center

Abstract

This paper describes a new algorithm that improves the performance of application-controlled demand paging for the out-of-core visualization of data sets that are on either 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 new algorithm can be applied to many different visualization algorithms since application-controlled demand paging is not specific to any visualization algorithm. 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 up to 60%. Visualization runs using data from remote disk ran about as fast as ones using data from local disk because the remote runs were able to make use of the remote server's high performance disk array.

Keywords: out-of-core visualization, demand paging, large data visualization, computational fluid dynamics.

1 Introduction

Simulations run on large parallel systems produce large data sets having hundreds of megabytes to terabytes of data. The researchers producing these data sets prefer to visualize them using their personal workstations. High-end PC workstations currently have the compute and graphics power to perform these visualizations. However, these workstations do not have sufficient memory to completely load large data sets, 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 can 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. The availability of reasonably-priced Gigabit Ethernet equipment means that network bandwidth is not an issue for remote out-of-core visualization over local area networks.

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 many visualization calculations only touch a small fraction of the data set. For example, streamline calculations only use the data surrounding the streamlines.

The demand paging algorithm divides the data set into fixed-size blocks, or pages. When a data value is requested, the paging system loads the page if it is not resident, and the page is cached in a memory pool. This means that, if the portion of the data set that is currently needed (the working set) is smaller than the memory pool and also has been loaded into memory, the performance is nearly the same as if the entire data set has been loaded into memory. A user examining one region of a data set will often be able to load the working set into memory and take advantage of this improved performance.

A locally-written interactive visualization tool has successfully used demand paging to allow interactive visualization of 5 to 10 GB data sets on systems with 500 MB to 1 GB of memory. Data sets on remote systems can be visualized interactively using the Network File System (NFS) to retrieve data pages.

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, a visualization algorithm must only be parallelized for it to take advantage of the overlapped disk access and computation. This modification is useful in itself, and may have already been performed. This is an advantage over out-of-core techniques that require the visualization algorithm to be modified for that specific technique [2, 3, 4]. A second advantage is that demand-paging techniques are not tied to a particular visualization algorithm; instead, they can be used to accelerate a number of visualization algorithms. Data structures proposed to enable out-of-core visualization of specific visualization techniques [2, 3, 4]. could be adapted to use demand paging and also be accelerated using the techniques described in this paper.

The new algorithm was designed so that it is compatible with time-critical visualization [5], 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

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

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 out-of-core visualization 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 by reading it in pieces that are small enough to 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.* [6] describes a general architecture that streams data through arbitrary visualization pipelines. The UFAT batch visualization program [7] also performs streaming on time varying data sets. When the visualization only accesses a small fraction of the data set, streaming algorithms that do not avoid reading all of the data can be slower than a demand paging algorithm.

The second type of out-of-core visualization algorithms is indexing algorithms. Many indexing algorithms have been described for isosurface computation [3, 4, 8, 9, 10]. These algorithms precompute an 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 out-of-core algorithms have similarities to this work. Flight simulation [11] and walk-through algorithms [12] 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 from the last few viewing positions. These algorithms cannot be used for computing visualizations because there is no concept of a viewing frustum during the visualization computation [13].

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 the data structures which track the pages that have been loaded into memory. When a data value is needed during a visualization computation, the associated page number is first 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 pages impacts the performance. This paper's implementation uses a *cubed* page format for paging structured grid files (unstructured grids are not currently 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 number of pages that must be read because, if the original data was simply broken into pages 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. Experiments show that using cubes instead of planes of data reduces the amount of data that must be read by about half. 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 described 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 to locate 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 was adapted from similar ones used for virtual memory page replacement in operating systems [14].

3.1 Field Encapsulation Library

The paging system is part of the Field Encapsulation Library [15]. 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 perform out-of-core visualization. Instead, they simply access data as if the entire data set was loaded into memory, and the demand paging system loads data as required.

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. Otherwise, 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 the checks succeed, the first thread wakes up the idle thread before going to sleep. The previously 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. However, we have not yet fully explored all of the pthreads scheduling options.

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 or hundreds of millions of pages mapped at once, and `sproc` locks use about 150 bytes each. Instead, the implementation uses one lock for a group of 48 to 80 pages, depending on the type of file. Limited experiments indicate that the number of pages per lock is not a critical parameter: doubling the number of pages per lock lowers the performance by about 3%.

Reading a page. When a thread finds that a page is not present, it finds memory for the page, and then, instead of reading the page from disk, it puts a request for the page to be read into a queue. 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 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 both a lock for scanning the page table and the per-page-group 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, NFS provides lower performance than 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 is 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 the 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 in both interactive and batch visualization applications. While most users will use the interactive application, the timing runs described below used a batch visualization application

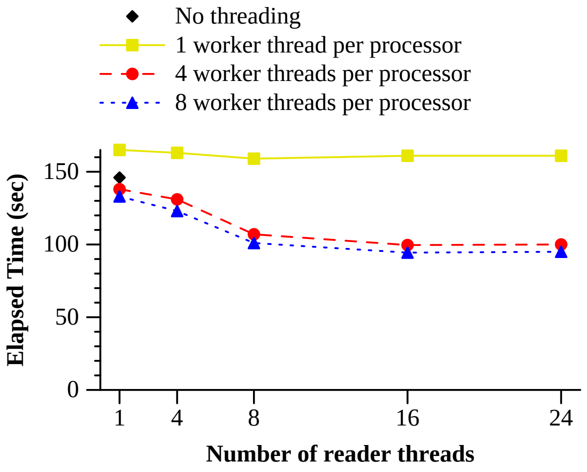

Figure 1: F18 timings using data from local disk and one processor.

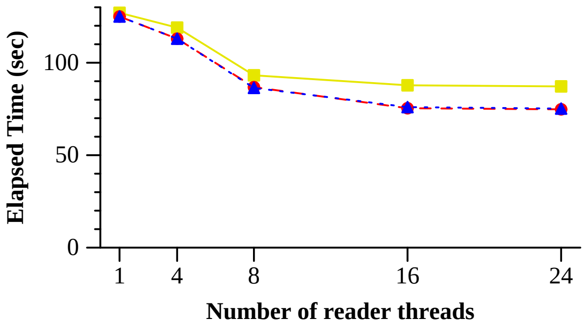

Figure 2: F18 timings using data from local disk and four processors.

called `batchvis`. We used the batch application because it allowed performance data to be recorded for the same visualization under a number of different conditions. However, because the data set is traversed only once, this is a worst-case scenario for out-of-core visualization using demand paging: there is no chance for the data set's working set to be entirely loaded into the cache of data pages.

The `batchvis` application uses FEL and the VisTech [16] visualization library. This application allows the user to compute a set of visualizations for each time step in the visualization. The program currently 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 of the field.

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.

Because the current remote paging server is a prototype, it does not implement security. We expect that adding security would not be difficult since the server only uses a single TCP/IP socket for communication.

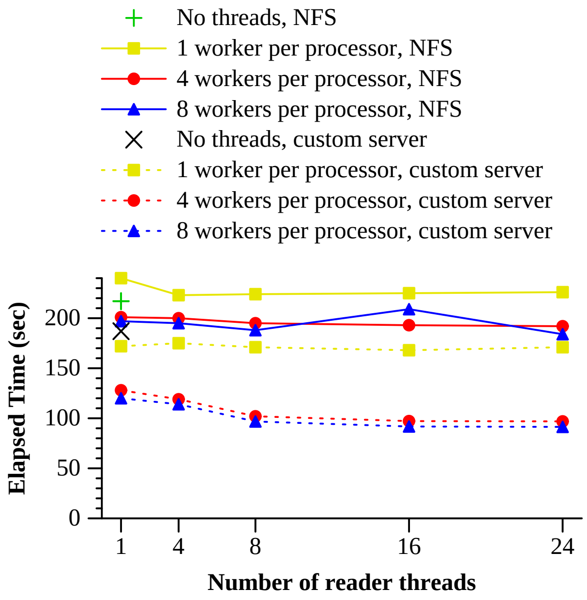

Figure 3: F18 timings using data from remote disk and one processor.

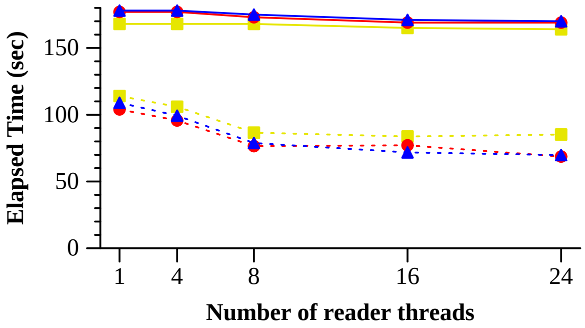

Figure 4: F18 timings using data from remote disk and four processors.

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 on older systems that have approximately the same performance that can be achieved with a modern fully configured high-end PC system. The visualizations were computed on an SGI Onyx with 4 196 Mhz R10000 processors and 1 GB of memory. Local data resided on a 4-disk striped disk array. These disks are fairly old, which means that their performance is low: 12.5 MB/sec for large, sequential reads using direct I/O. Remote data was served by an SGI Onyx with 8 196 Mhz R10000 processors and 5 GB of memory. The remote data was stored on a older RAID disk array that has a peak sequential performance of 25 MB/sec. This large system was used as a file server because it was the only system with sufficient disk space that could be dedicated to running performance experiments.

The remote server's large memory and processor configurations were largely unused during the runs since very little processing was necessary, and because all of the machines had their operating sys-

Data Set	Number of Time Steps	Grid Size (MB)	Solution Size (MB)	Total Size (GB)	Amount Read (MB)
SSLV	1	254.6	318.3	0.56	45.8
F18	150	27.0	33.7	4.97	99.6
Harrier	1600	55.1	68.9	107.7	8745

Table 1: Data set statistics. The sizes include both the data and file headers.

tem'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, large file in random order. In addition, multiple copies of the SSLV and F18 data sets were placed on the remote server. Consecutive runs rotated through the data set copies. All of the runs used a 200 MB memory pool to hold data pages.

The two systems were connected by an 800 Mbit/sec 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. The price of Gigabit Ethernet has decreased to the point where it can be deployed to individual researcher's workstations.

We used the three data sets shown in Figures 8 to 10 for the performance timings. Table 1 contains statistics about the data sets. The data sets are:

SSLV. This data set is 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. The F18 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 workstation used for the local runs did not have sufficient disk space to hold the Harrier, only runs using remote data access are shown below. Also, because the visualization takes over an hour to compute, fewer performance runs were measured with the Harrier.

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.

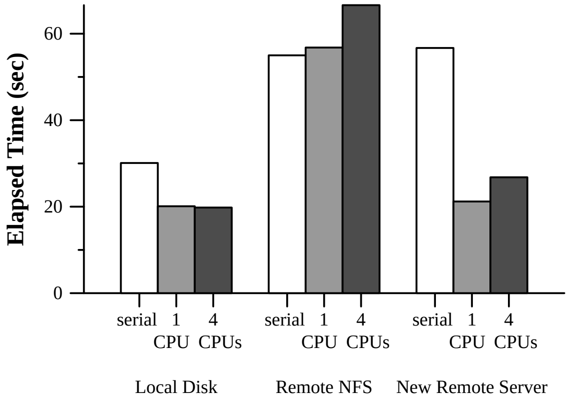

Figure 5: SSLV timings summary. The 1- and 4-CPU values are for 16 reader threads and 8 worker threads per processor.

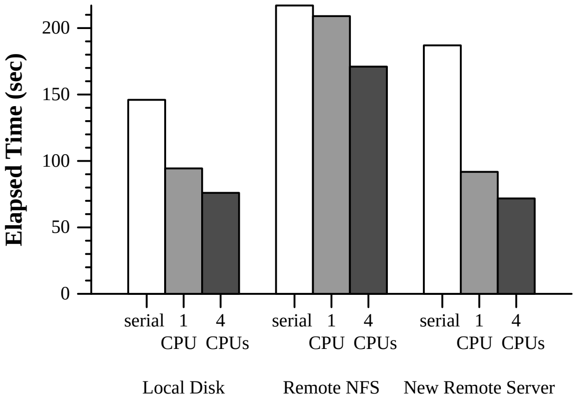

Figure 6: F18 timings summary. The 1- and 4-CPU values are for 16 reader threads and 8 worker threads per processor.

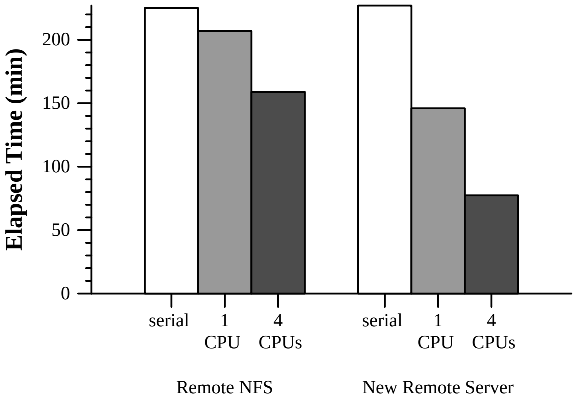

Figure 7: Harrier timings summary. The 1- and 4-CPU values are for 16 reader threads and 8 worker threads per processor.

7 Results

The detailed results are shown in Tables 2 to 4. All of the timings are from single run, which means run-to-run variations are expected.

Figures 1 to 4 show the general performance trends with the F18 when the number of reader and worker threads are varied.

The general trends for the other two data sets are similar. These

Data Access	Serial	1 Processor						4 Processors					
		Num. WT	Number of Reader Threads					Num. WT	Number of Reader Threads				
			1	4	8	16	24		1	4	8	16	24
Local	30.1	1	33.8	34.2	34.4	34.2	34.6	4	33.3	25.5	21.8	21.1	20.9
		4	34.5	26.6	22.7	21.5	21.6	16	35.0	25.8	21.9	19.5	18.7
		8	34.9	26.3	22.5	20.1	19.7	32	35.2	25.6	21.6	19.8	18.8
Remote via NFS	55.0	1	68.7	72.3	68.6	60.3	73.1	4	63.8	63.3	58.3	66.1	64.9
		4	59.5	69.3	66.4	64.8	67.8	16	69.7	58.5	71.5	63.2	61.6
		8	72.9	65.0	61.5	56.8	67.4	32	71.4	68.8	71.5	66.6	55.2
Remote via Server	56.7	1	41.6	43.2	43.4	43.6	45.9	4	44.5	35.3	30.7	29.2	29.6
		4	41.1	32.3	27.1	25.2	25.4	16	42.8	33.7	29.0	26.5	25.1
		8	35.4	27.4	23.3	21.2	20.9	32	42.8	34.1	29.2	26.8	26.3

Table 2: SSLV timings, in seconds. Key: WT = worker threads.

Data Access	Serial	1 Processor						4 Processors					
		Num. WT	Number of Reader Threads					Num. WT	Number of Reader Threads				
			1	4	8	16	24		1	4	8	16	24
Local	146	1	165	163	159	161	161	4	127	119	93.2	87.8	87.2
		4	138	131	107	99.6	100	16	125	113	86.8	75.4	74.8
		8	133	123	101	94.4	95.0	32	125	113	86.4	76.0	75.2
Remote via NFS	217	1	240	223	224	225	226	4	168	168	168	165	164
		4	201	200	195	193	192	16	177	177	173	169	169
		8	197	195	188	209	184	32	178	178	175	171	170
Remote via Server	187	1	172	175	171	168	171	4	114	106	86.6	83.7	85.2
		4	128	119	102	97.2	96.9	16	104	95.7	76.5	77.1	68.7
		8	120	114	96.8	91.8	91.4	32	109	99.3	78.8	71.8	69.8

Table 3: F18 timings, in seconds. Key: WT = worker threads.

Data Access	Serial	1 Processor				4 Processors			
		Num. WT	Num. Reader Threads			Num. WT	Num. Reader Threads		
			1	8	16		1	8	16
Remote via NFS	225	1	264	267	275	4	174	181	170
		4	231	234	226	16	165	171	159
		8	219	223	207	32	166	173	159
Remote via Server	227	1	225	233	229	4	123	116	90.7
		4	187	181	150	16	108	103	78.5
		8	173	167	146	32	109	105	77.4

Table 4: Harrier timings, in minutes. Key: WT = worker threads.

charts have curves for constant numbers of worker threads per processor. This means that the curves in the 1-processor charts are for 1, 4, or 8 worker threads, and the curves in the 4-processor charts are for 4, 16, or 32 worker threads.

Increasing the number of worker or reader threads generally increases the performance when the data set is on local disk or accessed via the custom server. This result shows that increasing the amount of concurrency that is available to the new multithreaded paging library increases the performance up to a point. However, when remote data are accessed using NFS, increasing the number of reader threads does not increase performance. Increasing the number of worker threads appears to slightly increase performance.

The charts show that 8 worker threads per processor is only slightly faster than using 4 worker threads. Increasing the number of worker threads further is unlikely to increase performance. Using 24 reader threads instead of 16 reader threads does not always increase the performance (see Tables 2 to 4). Overall, the best algorithm parameters are 8 worker threads per processor and 16 reader threads. These parameters give good performance and minimize the total number of threads. Since a real implementation would have to use fixed parameters, the following discussions will only consider timings with these parameters.

Figures 5 to 7 have timings with these parameters that allow the different algorithms to be compared. The comparison results fall into three categories:

Local data performance. The new threading library substantially decreased the execution time when the data set resides on local disk. If only one processor is used, the time taken is a third less than the time needed by the serial implementation. The run time decreases when four processors are used with the F18 to nearly half the serial time. This shows that these multithreading techniques will make good use of multiprocessor systems if there is sufficient disk bandwidth. The time does not decrease with the SSLV because that visualization only requires 3 CPU-seconds of computation—the bulk of the time is spent waiting for data.

Remote data performance. The runs that accessed remote data using the demand paging server were substantially faster than any of the serial runs with the demand paging server or any of the runs that retrieved data using NFS. When one CPU was used, the paging server runs took between 35% (for the Harrier) and 60% (for the SSLV) less time than the fastest 1-CPU NFS or serial runs. Runs using the demand paging server and 4 CPUs were even faster with the Harrier and F18. The 4-CPU SSLV run was slightly slower since there was no need for the additional processors, and using additional processors has increased overhead.

Remote data versus local data. One surprising result with the remote timings is that the remote runs using the demand paging server were about as fast as the corresponding local data runs. This can be explained by the higher performance disk subsystem on the remote server: it has a RAID array with multiple 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.

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 between 35% and 60%, in part due to the high performance of the remote server's disks. Finally, the new remote paging algorithm was substantially faster than using NFS for remote paging.

The performance increases described in this paper make out-of-core visualization using local and remote demand paging more attractive. The increased speed will allow researchers to visualize even larger data sets using the workstations on their desk instead of having to go down the hall to a large shared visualization system. Furthermore, the increased performance of remote demand paging will allow researchers to more quickly visualize data sets on their personal workstations that are too large to be stored on their workstation's disk.

One direction of future work would be run experiments using 100 Mbit/sec Fast Ethernet instead of HIPPI. The runs shown here read data at an average rate that could be handled with Fast Ethernet, at most 2.2 MB/sec. However, the peak rate is undoubtedly higher. 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. Finally, over the next few weeks researchers in our division will be exploring the limits of out-of-core remote visualization by using these new techniques to visualize a one terabyte data set on personal workstations.

Acknowledgments

This work was supported by NASA contract DTTS59-99-D-00437/A61812D. I would like to thank 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 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] Shyh-Kuang Ueng, Christopher Sikorski, and Kwan-Liu Ma. Out-of-Core Streamline Visualization on Large Unstructured Meshes. *IEEE Transactions on Visualization and Computer Graphics*, 3(4):370–380, December 1997.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

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

[8] T. Itoh and K. Koyamada. Automatic isosurface propagation using an extrema graph and sorted boundart cell lists. *IEEE Transactions on Visualization and Computer Graphics*, 1(4):319–327, December 1995.

[9] 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.

[10] H.-W. Shen. Isosurface extraction in time-varying fields using a temporal hierarchical index tree. In *Proceedings of Visualization '98*, pages 159–166. IEEE Computer Society Press, Los Alamitos, CA, 1998.

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

[12] 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.

[13] Michael Cox. Large data management for interactive visualization design. In *SIGGRAPH '99 System Designs for Visualizing Large-Scale Scientific Data course notes*, pages 5–29. August 1999.

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

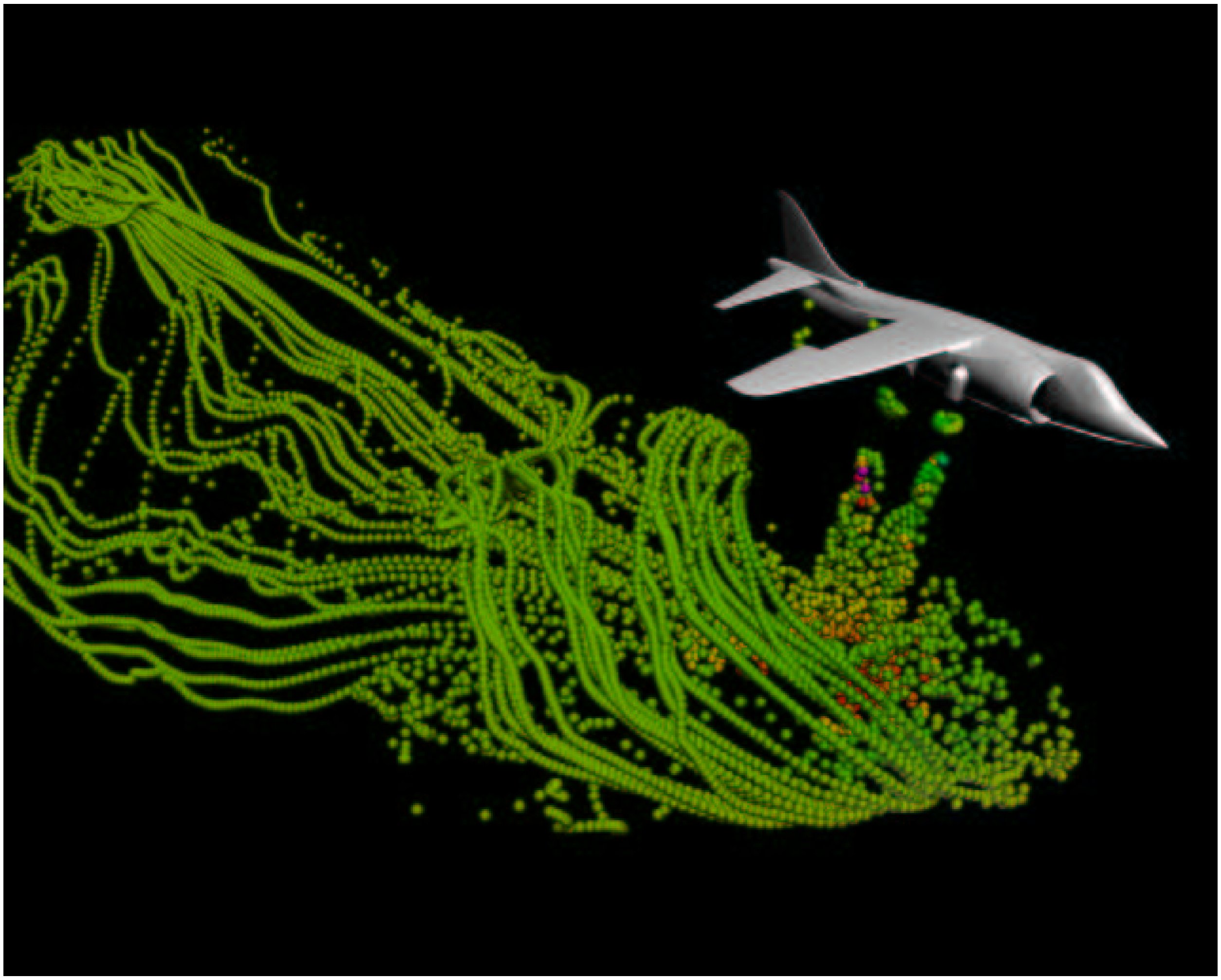

Figure 8: Visualization of the Harrier data set.

[15] 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.

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

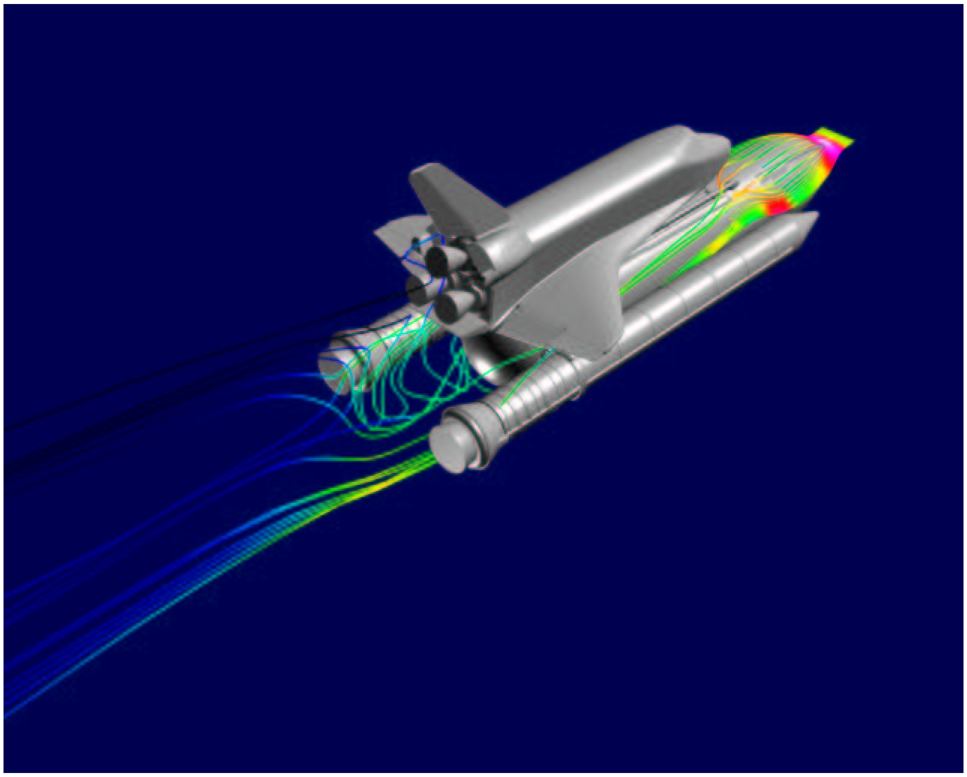

Figure 9: Visualization of the SSLV data set.

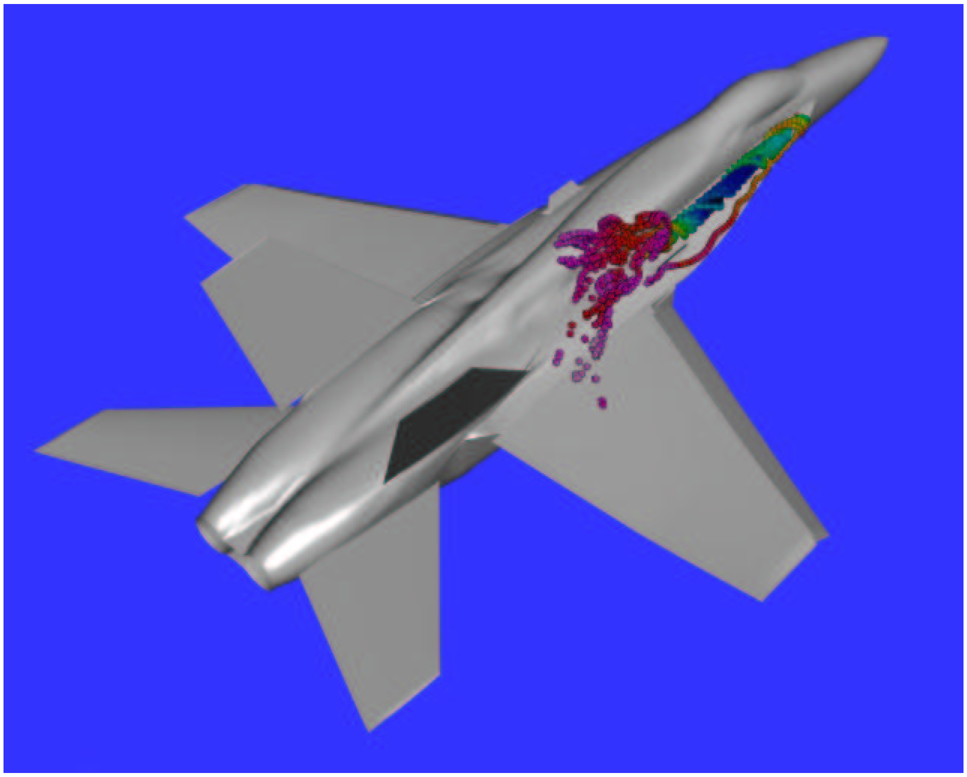

Figure 10: Visualization of the F18 data set.