Design Considerations for Parallel Graphics Libraries

Thomas W. Crockett

Institute for Computer Applications in Science and Engineering

NASA Langley Research Center

Abstract

Applications which run on parallel supercomputers are often characterized by massive datasets. Converting these vast collections of numbers to visual form has proven to be a powerful aid to comprehension. For a variety of reasons, it may be desirable to provide this visual feedback at runtime. One way to accomplish this is to exploit the available parallelism to perform graphics operations in place. In order to do this, we need appropriate parallel rendering algorithms and library interfaces. This paper provides a tutorial introduction to some of the issues which arise in designing parallel graphics libraries and their underlying rendering algorithms. The focus is on polygon rendering for distributed memory message-passing systems. We illustrate our discussion with examples from PGL, a parallel graphics library which has been developed on the Intel family of parallel systems.

This work was supported by the National Aeronautics and Space Administration under Contract Nos. NAS1-18605 and NAS1-19480 while the author was in residence at the Institute for Computer Applications in Science and Engineering (ICASE), M/S 132C, NASA Langley Research Center, Hampton, VA 23681-0001.

E-mail: tom@icase.edu

1. Introduction

Massively parallel supercomputers are becoming important tools for solving large-scale computational problems. For many applications, the ability of these systems to provide large, scalable memory capacities is at least as important as their raw computational power. The output datasets produced by these applications may range in size from hundreds of megabytes to hundreds of gigabytes. The data may also be in complex form, with sophisticated data structures and numerous interrelated variables.

Understanding these vast collections of numbers poses a challenge to the scientists and engineers who need to relate them to the problem under investigation. Conversion of abstract data into visual representations has proven to be a powerful aid to comprehension. The traditional approach to this process has been to run the simulation on the parallel machine, dump the results to disk, and then ship the data across the network to a specialized graphics workstation for post-processing and display. For large datasets, and particularly for time-dependent simulations, this approach may be impractical, due to limitations on file system capacities and I/O rates, network bandwidth, and workstation memory and processing capacities. Thus the researcher is forced to select some subset of the results for inspection, for example, by limiting the analysis to certain timesteps or reducing the resolution. This raises the possibility that unexpected or fine-grain phenomena may be overlooked.

An alternate approach is to exploit the power of the parallel supercomputer to perform the graphics and visualization operations in place. In the following sections, we explain why this approach is useful, and examine some of the issues which arise in designing graphics software on which parallel visualization applications can be based. For the sake of concreteness, we limit our discussion to the problem of parallel polygon rendering on distributed-memory message-passing architectures, as exemplified by the Intel family of parallel systems.

2. Why Parallel Rendering Is Useful

The term *rendering* refers to the computational process of generating an image from an abstract description of a scene. *Parallel rendering* refers to the exploitation of parallelism in performing the rendering computations. We will describe the rendering process in more detail when we examine strategies for parallelizing it in a subsequent section. But first we need to understand why parallel rendering is useful in the context of parallel applications. The benefits can be grouped into four main categories: data reduction, live visual feedback, processing power, and flexibility. We will examine each of these individually.

2.1 Data reduction

By performing visualization and graphics operations in place on the data, we may be able to reduce the size of the output stream substantially. With current workstation technology, useful image resolutions for visualization range from about 512 x 512 (roughly one-quarter of the display) up to 1280 x 1024 (full-screen on a large monitor). Images may be in a color-mapped or pseudo-color format, or in full color. To achieve reasonable quality, color-mapped images usually require at least 8 bits per pixel, resulting in a selectable palette of 256 colors. Full-color images usually use three color channels with 8 bits of color resolution in each channel, which results in a color palette containing over 16 million colors. Thus a typical image requires between 256 KB and 4 MB of data to represent. While this may seem like a lot, it is often a few orders of magnitude smaller than the output dataset produced by a large simulation, and image compression techniques can reduce this significantly.

With image sizes of a few hundred kilobytes or below, we have reached a level at which current networking technology (Ethernet and FDDI, for example) can deliver images at rates ranging from one to a few frames per second. At this speed, interactive work becomes practical.

By rendering on the parallel machine, we also eliminate the need for an expensive graphics workstation to drive the display device. Essentially any workstation-class machine with a color monitor and a reasonable network connection is sufficient, bringing visualization services to the desktop of the ordinary user.

2.2 Live visual feedback

Given appropriate parallel rendering algorithms and application programming interfaces, plus support for network image transmission, we can incorporate graphics directly into parallel applications to produce runtime visual output. This capability is useful for several purposes, including debugging, execution monitoring, and interactive steering. In debugging, for example, a visual representation of the data can help to identify variables which are not within expected ranges, or illustrate portions of the computational domain in which processors are not behaving as expected.

In execution monitoring, the user can obtain periodic visual snapshots of a computation to follow its progress or see how certain phenomena develop. Visual monitoring can also provide an early indication that things are going awry, allowing a run to be terminated early, with commensurate savings in computer charges and faster turnaround time.

The next step beyond monitoring is interactive steering, in which the user intervenes during the computation to modify simulation parameters or explore alternate scenarios. A parallel renderer does not

provide this capability directly, of course, but is one of the building blocks which can help to make interactive steering practical on parallel machines.

2.3 Processing power

A graphics library, parallel or otherwise, does not implement visualization operations directly, but provides a set of general-purpose routines which visualization tasks can use for modelling and rendering purposes. Some visualization methods are computationally expensive in their own right, especially when applied to large datasets. These techniques can make good use of parallelism to reduce computation times. One such example is described in [4].

The availability of parallel graphics libraries allows image generation to be incorporated directly into the parallel visualization task, as is normally the case for workstation-based visualization. This is useful not only for runtime visualization, but also for resource-intensive post-processing and data exploration tasks, in which the power or storage capacity of the parallel system may be needed to perform the visualization computations. Figure 1 shows the relationships between various software components in an application with integrated visualization.

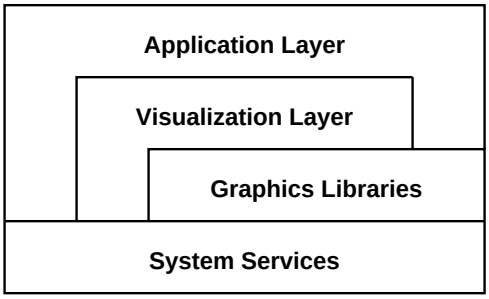

Figure 1: Relationship of software components in an application with integrated visualization.

2.4 Flexibility

Graphics workstations provide high levels of rendering performance by implementing common graphics operations in hardware. While this approach is quite effective, it has certain limitations. Usually the graphics operations can be performed only on a small set of predefined geometric primitives, with a limited set of illumination options, and at a maximum resolution which is determined by the size of the frame buffer memory. When specialized functionality is required, it must be performed in software or approximated using the built-in capabilities, with corresponding reductions in performance or image quality.

Software-based parallel rendering provides more flexibility, allowing additional geometric primitives, non-standard lighting models, high-resolution imaging, etc. The use of parallelism compensates for the absence of specialized rendering hardware to deliver performance that can be quite good.

3. Scope

In the remainder of this paper, we will concentrate on the problem of designing parallel graphics libraries which support parallel *polygon rendering* on distributed-memory message-passing (DMMP) architectures. Polygon rendering is the process of generating 2D images from a collection of 3D objects whose surfaces are defined by planar polygonal facets. Figure 2 illustrates the essentials of the polygon rendering problem. This is the type of rendering which is commonly supported in hardware by graphics workstations. It is useful for a wide variety of tasks, especially when features such as transparency and texture mapping are added. Polygon rendering is distinct from other techniques such as ray-tracing, radiosity, and volume rendering, which also have important applications. For a broader view of the parallel rendering field, see [5] [8] [9] [13] [14].

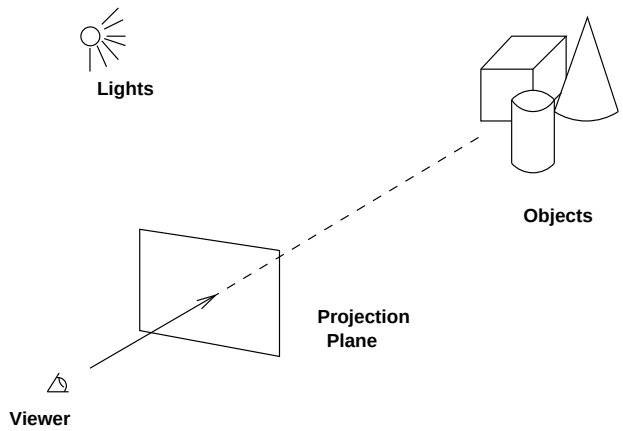

Figure 2: A scene is described by a collection of objects with associated light sources and viewing parameters.

Although our focus is on polygon rendering, most of the issues we discuss generalize to other types of geometric primitives (points, lines, patches, etc.), and a parallel polygon renderer could easily support additional primitives without changing the basic structure of the algorithms.

We concentrate on message-passing systems because they are the predominant class of scalable parallel system in use today. Again, we can generalize our discussion to any system with distributed memory which distinguishes between local and remote memory references. We will see, however, that the relative

cost of a remote memory reference is an important parameter which can influence a number of design decisions.

4. Design Issues for Parallel Graphics Libraries

The design space for a parallel graphics library and its underlying rendering algorithms is large, so it is important to understand the issues involved to help guide our decisions. Design considerations include, but are not necessarily limited to,

- finding and exploiting parallelism
- programming paradigm and library interface
- memory constraints
- data distribution
- communication
- load balancing
- performance scalability
- image output and display

We will examine each of these areas in detail.

4.1 Finding and exploiting parallelism

We would like for a parallel graphics library to exploit parallelism in its own internal computations so that rendering is fast and efficient, and does not become a bottleneck to the rest of the application. Thus we need to understand where parallelism can be found in the rendering process. There is a well-established functional pipeline for the polygon rendering process, as shown in Figure 3. This pipelined approach has been used to great advantage by graphics workstation vendors who have essentially mapped the pipeline into hardware. Geometric primitives are fed into the pipe one at a time, and a series of operations is performed which ultimately results in corresponding pixel values being set in the frame buffer memory.

However, the parallelism in the pipelined approach is limited by the number of operations in the pipe. For massively parallel systems, we need to exploit *data parallelism*, perhaps in combination with pipelining, in order to keep the processors busy. With the data parallel approach, operations are performed on multiple data items simultaneously, with the potential for much higher levels of parallelism. This strategy has been

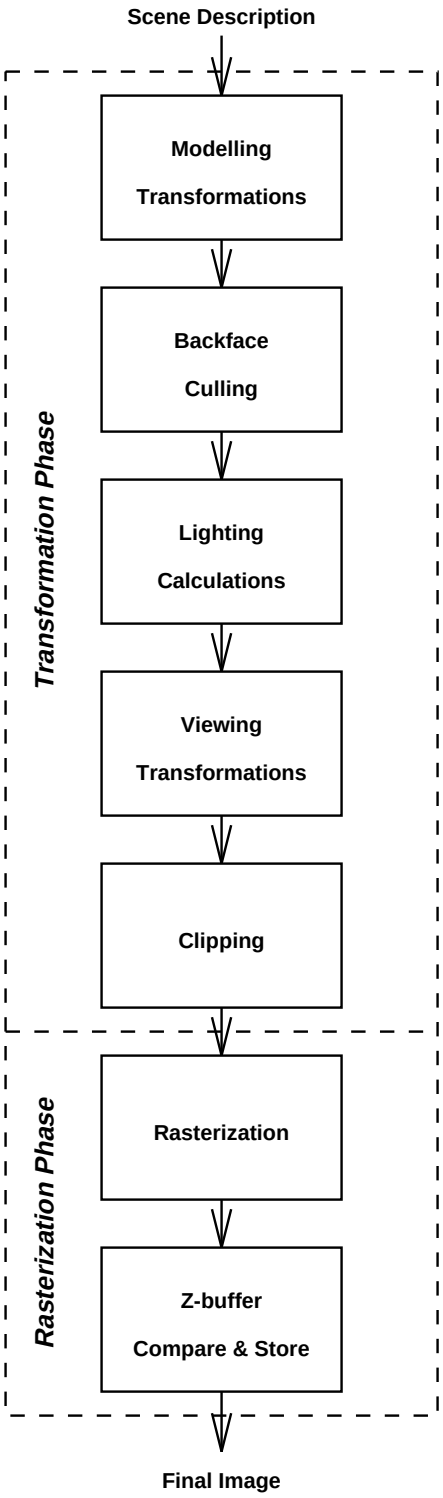

Figure 3: A typical polygon rendering pipeline.

adopted by a number of high performance systems, including Silicon Graphics' RealityEngine [1] and the University of North Carolina's Pixel-Planes 5 [7] and PixelFlow systems [10].

There are two main types of data parallelism which can be found in the rendering process. The first type is called *object parallelism*, because it refers to the operations which are performed on the geometric primitives which comprise objects in the scene. These operations constitute the first few stages of the rendering pipeline, including modelling and viewing transformations, backface culling, lighting calculations, and clipping. Collectively, we refer to this portion of the rendering pipeline as the *transformation phase*. The operations in this phase can be performed independently on each geometric primitive. Since complex scenes may contain hundreds of thousands or even millions of polygons, the transformation phase provides ample opportunities for data parallelism.

The second type of parallelism occurs in the later stages of the rendering pipeline, where the individual pixel values are computed for each transformed primitive. This process is known as *rasterization*, and we call this section of the rendering pipeline the *rasterization phase*. In principle, the value at each pixel can be computed independently based on the information at polygon vertices, resulting in massive *image parallelism*. In practice, pixel values are computed in groups, either over an entire primitive, or some subset of the primitive. This allows more efficient incremental interpolation algorithms to be used, but complex scenes may still contain millions of independent rasterization tasks.

So far we have not addressed the problem of hidden surface elimination. A variety of techniques are available, but for simplicity we will assume a z-buffer algorithm, in which a depth component (z coordinate) is associated with each pixel. Pixels which lie behind other pixels along the line of sight are discarded in the final step of the rasterization phase. Due to its simplicity, z-buffering has been very popular in hardware renderers, even though it requires a few bytes of additional memory at each pixel. For parallel rendering, z-buffering is attractive since it exploits image parallelism very naturally.

4.2 Programming paradigm and library interface

Now that we have determined where parallelism exists in the rendering process, we must consider how parallel applications behave and how they will invoke graphics operations. DMMP systems are very flexible with respect to the parallel programming paradigms which they support. At one extreme, they can support truly MIMD applications, in which processors may perform completely different functions. At the other extreme, they allow data-parallel applications to run in a pseudo-SIMD mode, with every processor performing the same set of operations on its portion of a distributed data structure. In between is the SPMD (Single-Program Multiple-Data) model in which processors execute the same program, but with some flexibility in local control flow.

For reasons that will become apparent, parallel rendering algorithms require significant amounts of communication and at least some internal synchronization. Thus a purely MIMD application interface is unworkable. However, parallel polygon renderers have been developed using both SIMD [12] and SPMD [3] [6] approaches, so it should be possible to construct a library interface using either of these models. On a DMMP system, many applications are written in an SPMD style, so this choice is preferred. It does not conflict with purely data-parallel applications, but does allow processors some independence for operations such as data modelling. The rendering and display operations are then considered to be synchronization points in the code, and must be invoked more or less simultaneously by every processor.

4.3 Memory constraints

As we have already discussed, many applications of interest are memory-intensive. Thus it is important for the graphics library and its underlying rendering algorithms to avoid consuming the resources that the application needs to use. A good parallel graphics library should have modest, uniform, and predictable memory requirements so that the application programmer can plan accordingly. This constraint is noticeably absent in much of the literature on parallel rendering algorithms, where the assumption is often made that the renderer exists *in vacuo* and that large amounts of memory are available for its use.

Memory considerations also argue against *retained mode* approaches in which the graphics library maintains an internal copy of the object data. Rather, the library should be structured to give the application programmer considerable control over memory management.

4.4 Data distribution

The rendering process requires two large data structures:

- the list of objects to be rendered
- the memory buffer in which the image is constructed

The size of the object database depends on the application. Since we are targeting a DMMP environment, we can assume that an application will distribute its large data structures in a roughly uniform fashion among the processors. Since these same data structures either constitute the objects to be rendered or contain the information from which objects will be generated, partitioning of the object database is a natural side-effect of parallelizing the application.

Image memory stores information about individual pixels, including as a minimum color and depth values. Simple renderers require about 8 bytes per pixel, or roughly 8 MB for an image with 1024 x 1024

resolution. The addition of features such as transparency can increase this significantly. On most DMMP systems this is too high a price to pay on a per processor basis, so we will need to partition the image memory, rather than replicating it. A number of strategies are possible. Perhaps the simplest is to partition the image memory into p equal-sized segments, where p is the number of processors. Figure 4 shows several common approaches. A subdivision into square rather than rectangular regions is theoretically more efficient, since it tends to minimize the number of regions intersected by a primitive, which in turn reduces communication requirements and interpolation overheads. The adaptive partitioning strategy of Figure 4(d) has been used effectively in shared memory environments in conjunction with dynamic load balancing schemes [15].

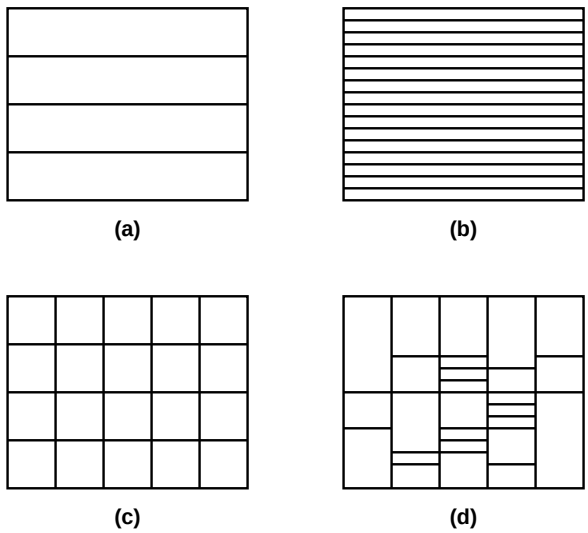

Figure 4: Image partitioning strategies: (a) scanline strips, (b) interleaved scanlines, (c) square blocks, (d) adaptive decomposition. Vertical partitionings are also possible.

By distributing both the object data structures and image memory, we provide *data scalability*. Increases in scene complexity or image resolution can be accommodated simply by adding additional processors.

4.5 Communication

During the rendering process, three-dimensional primitives in user-defined *world coordinates* are projected into a two-dimensional screen space. The mapping between these two spaces is a function of the viewing parameters and the positions of the objects, both of which are subject to change from frame to frame. Given that the object data and image memory are both distributed among the processors, an inherent sorting step must take place at some point during the rendering pipeline. For complex scenes, this sort results in a massive amount of communication as geometric primitives (or pieces of primitives) are

moved to the processors which contain the corresponding image pixels. Performing this communication efficiently is one of the central issues for parallel rendering algorithms on DMMP systems. We will discuss this problem in more detail in Section 4.7.

Molnar *et al.* have developed a taxonomy for parallel rendering algorithms based on where the communication occurs within the rendering pipeline [11]. They propose three classes of algorithms which they call *sort-first*, *sort-middle*, and *sort-last*. In a sort-first algorithm, primitives are mapped to their final destinations early in the transformation phase. Sort-middle algorithms insert communication between the transformation and rasterization phases, while sort-last algorithms send pixels to their final destination late in the rasterization phase. Molnar's analysis shows that the preferred approach depends on a variety of architectural parameters and application considerations. Recent work on DMMP systems has tended toward sort-middle approaches [3] [6], which tend to strike a balance between computational overhead and communication volume. With sufficiently high bandwidth, sort-last approaches become more interesting, since they avoid the computational overheads which occur when primitives must be split prior to rasterization.

4.6 Load balancing

Another major issue is load balancing. To achieve maximum performance, processors need to have roughly the same amount of work to do. If primitives are uniformly distributed among processors and they map uniformly into the image space, then the rendering process is almost perfectly parallel, although overheads are introduced due to primitive splitting and communication [3]. Unfortunately, real scenes are not this well-behaved. In many applications, the area of interest may be localized within the data space, and depending on the data partitioning scheme in use, this may correspond to a subset of the processors. In this case the objects to be rendered will not be evenly distributed. If the load imbalance is severe enough, it may be worthwhile to redistribute objects, either before or during the rendering process.

The distribution of primitives within the image space is often localized as well. An example is shown in Figure 5. In this image, the vast majority of primitives are located on the leading and trailing edges of the wing and engine pod, and to a lesser extent on the fuselage. Hence the bulk of the rasterization operations map to a small percentage of the image space. Processors which are responsible for these pixels will have an inordinate amount of work to do relative to others which may be assigned to empty space. Perhaps the best static approach to this problem is to assign scanlines to processors in an interleaved fashion, as in Figure 4(b). At least for modest numbers of processors, this will distribute the load reasonably well, with no additional overhead due to load balancing.

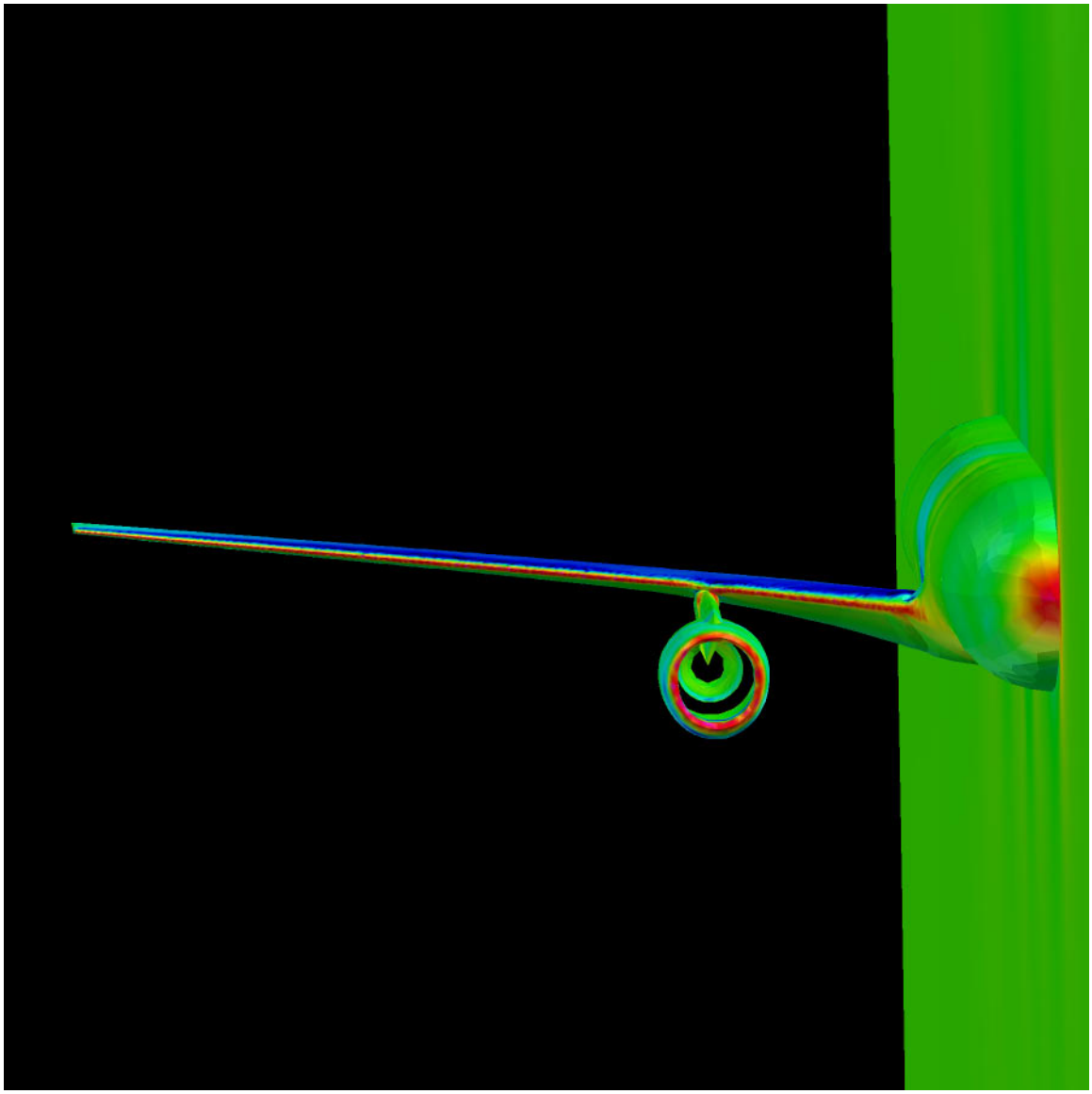

Figure 5: Example of a scene in which geometric primitives are highly localized. The image contains 31,271 triangles, mostly along the wing and engine pod.

For large numbers of processors or more pathological cases, dynamic load balancing schemes may be in order. While these can be devised fairly easily for shared memory systems [2] [14] [15], higher communication costs and the need to maintain global state make this problem more challenging in a DMMP environment, and little work has been done in this area. Ellsworth [6] uses a predictive approach to the problem on Intel's Touchstone Delta system. He exploits a property known as *frame coherence*, the tendency of one frame in a sequence to be similar to immediately preceding frames. With this technique, the distribution of primitives within the previous frame is used to predict the rasterization loads in the

current frame, and image segments are reassigned based on the expected load. While the results are promising, his load reassignment algorithm is serial, and becomes a significant bottleneck with large numbers of processors.

In the context of a graphics library, some dynamic load balancing schemes can pose a problem with respect to memory utilization. If significant portions of the object data or image buffers are reassigned, the previously stated goal of modest, uniform, and predictable memory utilization may be compromised. This is a classic example of the trade-offs which arise in designing parallel renderers. In this case, we could place limits on the amount of data which could be reassigned, with a possible performance penalty for situations in which the load imbalance is high.

An alternate view is to treat processors as simply “smart memory”. In this approach, data is statically assigned, and processors are responsible for operations on their local data. With large numbers of processors, performance will be generally high (but see the discussion of latency in the next section), even though utilization may not be particularly good.

4.7 Performance scalability

Ideally, we would like for rendering performance to scale with the number of processors. Unfortunately, this is a major problem for parallel renderers on DMMP systems. Communication costs and load imbalance both contribute to this problem, but communication is more serious, since the high costs of remote memory references also contribute to overheads in dynamic load balancing schemes.

Given a uniform load, Crockett and Orloff show that the software overhead, or *latency*, involved in sending and receiving messages poses fundamental limits on the scalability of parallel rendering algorithms which distribute both object and image data [3]. This occurs because the communication patterns which arise are all-to-many or all-to-all, implying that the average processor will have to communicate with a substantial proportion of the other processors. Hence the number of messages handled by each processor (and thus the message overhead) increases with the number of processors in the system.

Despite this fundamental restriction, there are some things that can be done to improve the situation. Data items can be buffered into longer messages before sending them, which helps to amortize the message latencies. For a given scene, we can define a quantity called the *useful message length.* This is the average number of data items which a processor will send to each of the processors with which it communicates. Increasing message sizes beyond this value will have diminishing returns, since few messages will actually contain that much data. For fixed scene complexity, the useful message length decreases with the square of the number of processors, so this technique scales very poorly.

Crockett and Orloff also show that network contention can cause delays as message sizes approach their useful limits, so that in bandwidth-limited networks, the optimum message size is less than the expected useful length. Figure 6 illustrates this phenomenon for the highly uniform random triangle scene shown in Figure 7. For a given numbers of processors *p*, the message length is varied from 1 to the useful maximum. On the iPSC/860, longer messages result in high volumes of data being injected into the network within a relatively short time, which in turn results in contention delays and degraded performance. On the Paragon, where the network bandwidth is roughly an order of magnitude higher than the load which can be generated by a single processor, contention delays are not apparent.

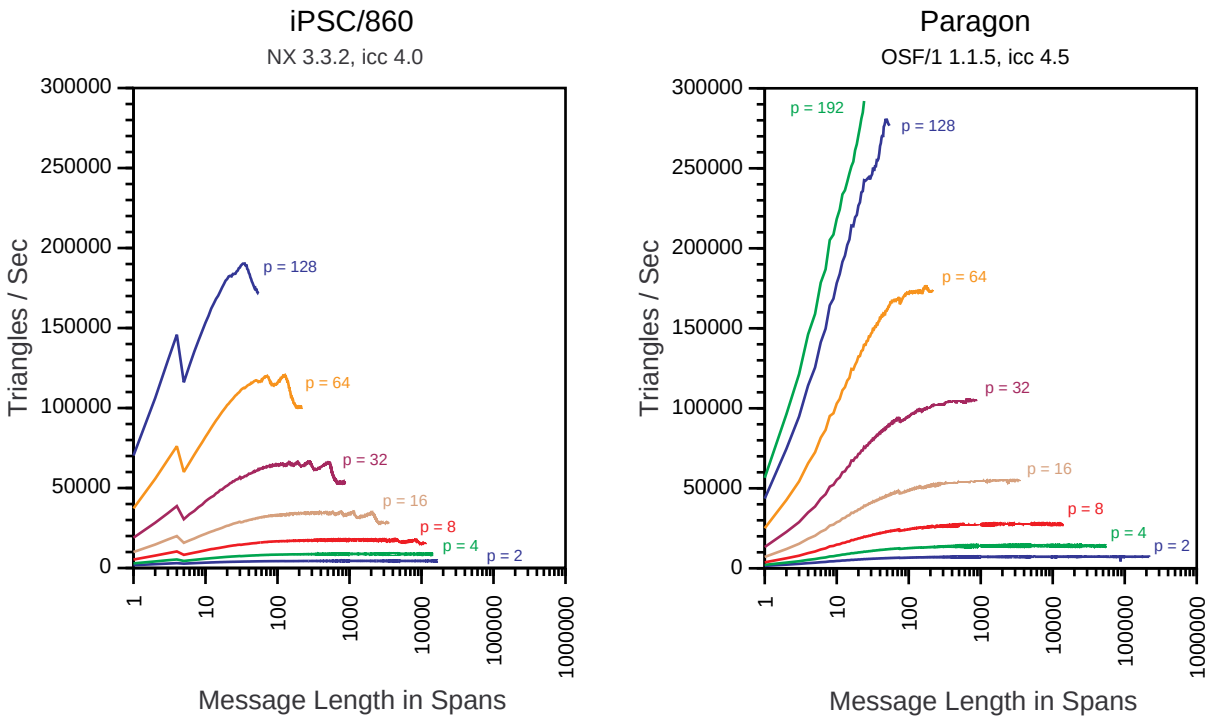

Figure 6: Rendering rates vs. message lengths for 100,000 random triangles. For a given number of processors *p*, message lengths range from 1 to the expected useful maximum. On the iPSC/860, for *p* < 16, maximum message lengths were limited by processor memory capacities.

Another way to combat message overhead is to reduce the algorithmic complexity of the message counts. Ellsworth uses a "two-step" communication protocol to reduce the number of messages which are required [6]. His store-and-forward approach inserts an intermediate communication step in which data from multiple source processors are aggregated for routing to the final destination. As an added benefit, the useful message lengths also increase, providing better latency amortization. Results from Intel's Touchstone Delta show substantial performance improvements with large numbers of processors.

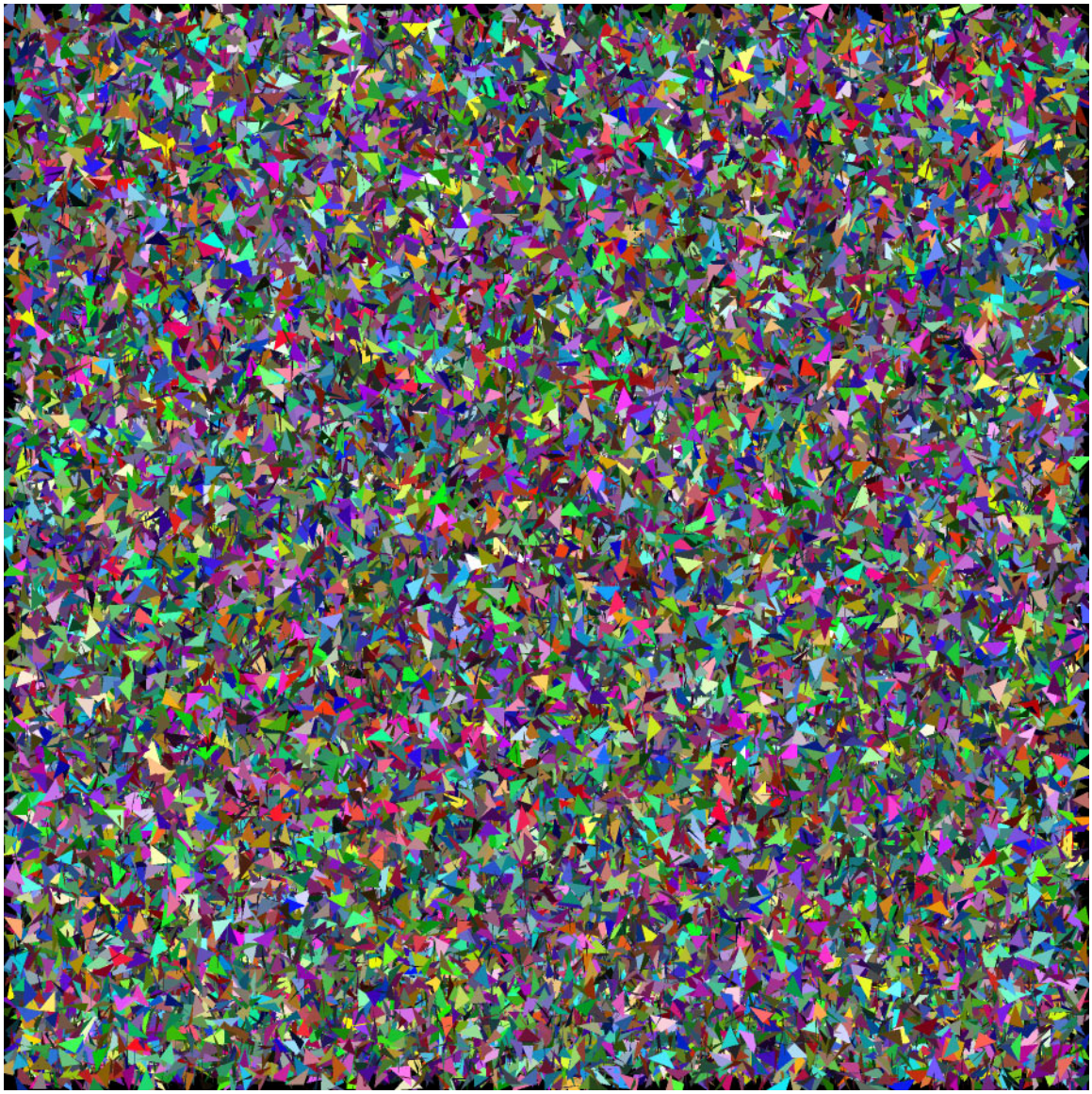

Figure 7: Test scene composed of 100,000 isosceles right triangles randomly oriented within a cubic volume. This scene statistically approximates a uniform scene and is useful for verifying our analytical performance models.

In situations where network bandwidth is a limiting factor or processors have varying workloads, it may be desirable to decouple the transformation and rasterization phases, allowing each to proceed asynchronously with respect to the other. Crockett and Orloff describe a technique for doing this which uses a combination of buffering and asynchronous message passing protocols to allow a high degree of overlap between computation and communication [3]. This method should be well-suited to systems which provide independent mechanisms for message delivery, such as the message co-processors on the Intel Paragon.

4.8 Image output and display

While distributing image memory among processors may be desirable from the standpoint of maximizing image parallelism and assuring data scalability, it poses a problem when the results need to be displayed. Portions of the image must be collected from every processor and assembled into a whole. Since a complete image may be several megabytes in size, we would prefer to have the final assembly occur someplace external to the graphics library, in order to minimize the impact on the driving application. We would also prefer to have the image segments written in parallel to the display device, to avoid a potentially serious sequential bottleneck.

The problem of image assembly is not especially severe. If the image is being saved to a file, each processor can write its own segment of the image. Address tags can be inserted into the image format so that the order of the segments is not important, and the image can be reconstructed at display time. With appropriate support for parallel file I/O, the output step may perform satisfactorily.

If the image is going to a dedicated display device, for example a HiPPI frame buffer, then the problem may also be tractable. To illustrate the requirements, consider this demanding scenario: Assume that we have a high-performance parallel renderer which is capable of generating full-frame (1280 x 1024) images in 24-bit color at animation rates, i.e., 30 frames per second. In order to display the image data at these rates (assuming no compression), we need a display bandwidth of 120 MB/s. On paper, the internal bandwidth of recent high-performance systems such as the Cray T3D and Intel Paragon are sufficient to support this. The problem is to orchestrate data flow from the processors to the display device so that these rates can be achieved in practice. If image segments are sent to the display device as messages, latencies may become the limiting factor with large numbers of processors. Aggregation techniques can be used to reduce the number of messages, but the increased memory requirements could conflict with our goal of modest memory consumption within the graphics library. We also need an external interface to the display capable of sustaining the internal data rates, and the display memory must be randomly addressable to avoid serializing the image stream.

If the images are destined for a remote workstation, a serial data stream is required and the bandwidth problem becomes more severe. Image compression techniques need to be applied to achieve reasonable display rates. Again, image assembly is not particularly a problem, since typical workstations can perform this task at satisfactory frame rates.

In the next section, we will give a brief overview of a parallel graphics library being developed by the author, and describe how it addresses some of the design issues raised above.

5. PGL: A Parallel Graphics Library for DMMP Systems

PGL is a prototype parallel graphics library for distributed memory message passing systems. It is intended both as a testbed for research on parallel rendering algorithms, as well as a semi-production graphics library for parallel scientific and engineering applications. PGL is written to be portable to a variety of message-passing systems, but to date it has been tested only on Intel iPSC/860 and Paragon XP/S platforms. In the following paragraphs, we will examine PGL in the context of the design considerations discussed above.

PGL is based on an improved version of the rendering algorithm described in [3]. The PGL renderer exploits both object and image parallelism, with the sort step occurring in the middle of the rasterization stage of the rendering pipeline. The unit of communication is a *span*, which is defined as that portion of a 2D primitive which intersects a single scanline (Figure 8). Each span consists of two endpoints with associated color and depth information. A significant advantage of sorting at the span level is that computational overheads incurred by splitting geometric primitives are completely eliminated, since spans are produced as a normal part of the rasterization process. The disadvantage is that the volume of data to be communicated is higher than if intact primitives were used. On high-bandwidth machines (such as the Paragon), this message traffic is manageable. Furthermore, our rendering algorithm permits a high degree of overlap between computation and communication, so message transfers can proceed in the background.

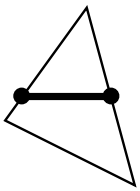

Figure 8: A *span* is that portion of a 2D geometric primitive which intersects a single scanline.

The PGL application interface represents a compromise between retained mode and immediate mode rendering strategies. Efficient operation is obtained by first defining aggregate data objects, and then rendering them all at once. To reduce memory requirements, PGL does not make its own copy of the data, but instead uses pointers to the application's data structures.

Objects are defined as collections of geometric primitives, with a number of attributes (color, normals, material properties, etc.) which can be specified on a per-object or per-primitive basis. In order to render an object it must first be instantiated. Objects may be instantiated as many times as needed. The instantiation mechanism allows for inheritance of geometry and attributes, with the ability to override these properties

on a per-instance basis. An instance may have nested sub-instances, and modelling transformations may be specified at each level.

PGL assumes an SPMD programming paradigm. Modelling operations may be performed semi-independently on each processor, while rendering and display operations require the participation of every processor. Operations which set global state, such as viewing parameters or lighting specifications, must also be performed cooperatively.

The current version of PGL makes no attempt to perform dynamic load balancing. Instead, it uses an interleaved partitioning of scanlines to spread the rendering load for high-density areas of the screen across multiple processors (the "smart memory" philosophy of Section 4.6). This approach meshes well with the span-based sorting step, and strictly follows the rule that memory requirements must be uniform and predictable. By always assigning complete scanlines to processors, PGL guarantees that spans will never need to be split, and the full benefit of incremental interpolation algorithms can be achieved.

For PGL, performance scalability remains an open issue. The current design does not use Ellsworth's two-step sending optimization, but instead is anticipating (perhaps prematurely) the day when low-latency remote memory references will make direct communication competitive with the store-and-forward approach. The problem with direct communication in a high-latency environment is readily seen in Figure 6. With large numbers of processors, the number of spans to be communicated from one processor to any other is relatively small. Consequently, message latencies cannot be amortized completely, and performance suffers. For this particular scene, a single Paragon processor can render about 4,200 triangles/second. With 192 processors, performance peaks at about 292,000 triangles/second, for a parallel efficiency of 36%. With more complex scenes, rendering rates in excess of half a million polygons per second have been achieved, with parallel efficiencies reaching 59%.

Image output in PGL has been designed primarily to support network transmission to remote workstations, although output to files is also provided. For remote image display, we assume that the network will be the bottleneck, so we want to reduce the volume of output to a reasonable level. PGL accomplishes this by compressing images using a combination of run-length encoding and frame-to-frame differencing. This strategy exploits both spatial coherence within an image as well as temporal coherence between images. Although more aggressive compression techniques are possible, this approach has several advantages:

- the algorithm is lossless
- compression is fast, and can be performed independently and in parallel on each scanline, with no

communication required

- decompression is very fast, approaching animation rates on typical workstations

The principal disadvantage is that a copy of the previous image must be retained to compute the frame-to-frame differences. Because image memory is evenly distributed, the per-processor memory penalty is generally acceptable with moderate to large numbers of processors.

For transmission across the network, compressed image segments must be sent to a designated processor which then writes them to a socket connection. This is a potential bottleneck with large numbers of processors, but for the moment, the speed of the network and network interfaces is the limiting factor.

Our experience has shown that we can expect display rates of between 0.5 and 2 frames per second on Ethernet and FDDI networks in the presence of normal traffic. While faster display rates would be preferred, this is adequate for simple interactive work. Of course, performance in any given situation depends on image content and resolution along with network bandwidth and traffic.

5.1 Initial experiences with PGL

Although it is still under development, we have used PGL in two demonstration programs. One is a simple interactive geometry viewer which allows a user to load object descriptions into memory and manipulate them. The interactive interface resides on the user's workstation and sends commands to the renderer which runs on the parallel machine. Figure 5 was rendered using this program.

The second demonstration program is a parallel Direct Simulation Monte Carlo (DSMC) application developed by Richard G. Wilmoth at NASA Langley. This code is being used to investigate the behavior of rarefied flowfields, such as the plumes from space shuttle thrusters when they are fired in space. These simulations typically run for thousands of time steps with hundreds of thousands of particles. We added PGL calls to the DSMC code to produce live visual output at each timestep of the simulation. Figure 9 shows an example of the output. Table 1 gives execution times for the first 100 timesteps of this particular problem. The number of particles being simulated starts at 0 and increases over time until a steady state is reached. At the 100th timestep, we are rendering 176,712 particles. The problem was run on Langley's Paragon XP/S using 32 processors. The display time includes network transmission via FDDI to a Sun SPARCstation-10, as well as output to an animation file on the Paragon's Parallel File System. Image resolution was 800 x 640.

We have also found PGL to be very useful for high resolution rendering. We routinely use it to produce publication-quality images on our film recorders and dye sublimation printers at resolutions up to 4096 x

Operation	*Time in seconds*
Modelling	2.52
Rendering	88.02
Display	78.57
Total (simulation + graphics)	365.67

Table 1: Cumulative execution time for graphics operations in the DSMC simulation.

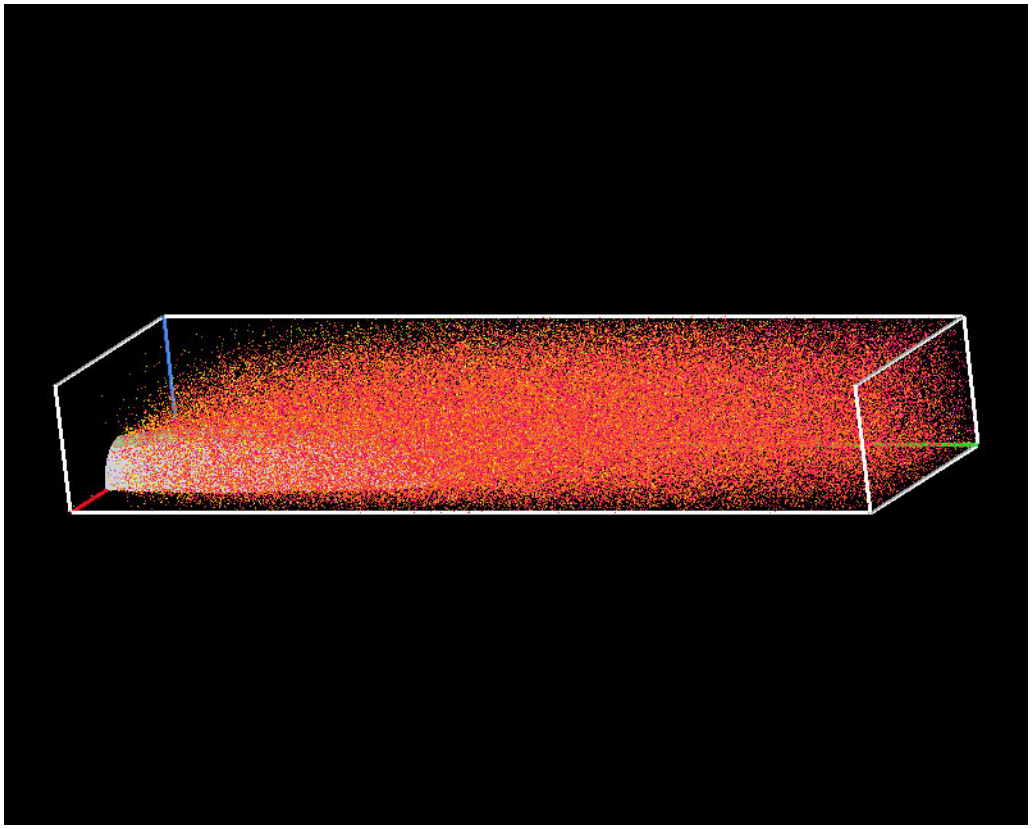

Figure 9: This simulated plume from a reaction control jet was rendered using PGL. It contains 176,712 particles.

4096. At these resolutions, details emerge which are often invisible on workstation monitors. Given a sufficiently large machine and a suitable output device, PGL is capable of rendering at gigapixel (32K x 32K) resolution. Eight gigabytes of memory would be needed for the image and z buffers.

6. Summary

Applications which run on large-scale parallel supercomputers often generate massive datasets. Graphics and visualization operations on these datasets can be performed in place by exploiting the available

parallelism. Not only does this approach cut down on network traffic, it also allows live visual feedback from executing applications.

To realize these benefits, we require appropriate parallel graphics libraries and underlying rendering algorithms. This is an active area of research. Many issues arise in designing both the library interface and the rendering algorithms. These are driven by the properties of the intended applications as well as the target architectures. Successful designs must achieve balanced trade-offs between conflicting goals. Message latency, load balancing, and image output are areas which require special attention in order to achieve scalable performance.

We are developing a parallel graphics library called PGL which is intended specifically for distributed memory message passing environments. We have described how PGL addresses many of the design issues which arise. While PGL does not solve all of the outstanding problems, it has proven to be viable for the small number of applications on which it has been tested. We expect ongoing research efforts by ourselves and others to yield further improvements.

Acknowledgments

This work was conducted using Intel iPSC/860 and Paragon XP/S computers operated by NASA's Numerical Aerodynamic Simulation Program and the Information Systems Division of Langley Research Center. The grid and solution data used to create Figure 5 was provided by Dimitri Mavriplis.

References

[1] Akeley, Kurt. RealityEngine graphics. *Computer Graphics Proceedings*, Annual Conference Series, 1993, ACM SIGGRAPH, Aug. 1993, pp. 109-116.

[2] Crockett, T., and Orloff, T. A Parallel Rendering Algorithm for MIMD Architectures. ICASE Report 91-3 (NASA CR 187571), ICASE, June 1991.

[3] Crockett, T., and Orloff, T. A MIMD rendering algorithm for distributed memory architectures. *Proceedings 1993 Parallel Rendering Symposium*, IEEE Computer Society Press, Oct. 1993, pp. 35-42, 108.

[4] Crossno, P., Cline, D., and Jortner, J. A heterogeneous graphics procedure for visualization of massively parallel solutions. FED-Vol. 156, *CFD Algorithms and Applications for Parallel Processors*, ASME Fluids Engineering Conference, Washington, D.C., June 1993, pp. 65-70.

[5] Dew, P. M., Earnshaw, R. A., and Heywood, T. R., eds. *Parallel Processing for Computer Vision and Display*, Addison-Wesley, 1989.

[6] Ellsworth, D. A multicomputer polygon rendering algorithm for interactive applications. *Proceedings 1993 Parallel Rendering Symposium*, IEEE Computer Society Press, Oct. 1993, pp. 43-48, 109.

[7] Fuchs, H., Poulton, J., Eyles, J., Greer, T., Goldfeather, J., Ellsworth, D., Molnar, S., Turk, G., Tebbs, B., and Israel, L. Pixel-Planes 5: a heterogeneous multiprocessor system using processor-enhanced memories. *Computer Graphics 23*, 3, ACM SIGGRAPH, July 1989, pp. 79-88.

[8] *IEEE Computer Graphics and Applications 14*, 4, special issue on parallel rendering, July 1994.

[9] *IEEE Parallel and Distributed Technology 2*, 2, Summer 1994.

[10] Molnar, S., Eyles, J., Poulton, J. PixelFlow: high-speed rendering using image composition. *Computer Graphics 26*, 2, ACM SIGGRAPH, July 1992, pp. 231-240.

[11] Molnar, S., Cox, M., Ellsworth, D., and Fuchs, H. A sorting classification of parallel rendering. *IEEE Computer Graphics and Applications 14*, 4, July 1994, to appear.

[12] Ortega, F., Hansen, C., and Ahrens, J. Fast data parallel polygon rendering. *Proceedings Supercomputing '93*, IEEE Computer Society Press, Nov. 1993, pp. 709-718.

[13] *Proceedings 1993 Parallel Rendering Symposium*, ACM Press and IEEE Computer Society Press, October 1993.

[14] Whitman, S. *Multiprocessor Methods for Computer Graphics Rendering*, Jones and Bartlett, 1992.

[15] Whitman, S. A task adaptive parallel graphics renderer. *Proceedings 1993 Parallel Rendering Symposium*, IEEE Computer Society Press, Oct. 1993, pp. 27-34, 107.