NASA Contractor Report 201637

ICASE Report No. 96-75

ICASE

BEYOND THE RENDERER: SOFTWARE ARCHITECTURE FOR PARALLEL GRAPHICS AND VISUALIZATION

Thomas W. Crockett

NASA Contract No. NAS1-19480
December 1996

Institute for Computer Applications in Science and Engineering
NASA Langley Research Center
Hampton, VA 23681-0001

Operated by Universities Space Research Association

National Aeronautics and Space Administration

Langley Research Center
Hampton, Virginia 23681-0001

Beyond the Renderer:
Software Architecture for Parallel Graphics and Visualization

Thomas W. Crockett

Institute for Computer Applications in Science and Engineering
M.S. 403, NASA Langley Research Center
Hampton, VA 23681
USA

Abstract

As numerous implementations have demonstrated, software-based parallel rendering is an effective way to obtain the needed computational power for a variety of challenging applications in computer graphics and scientific visualization. To fully realize their potential, however, parallel renderers need to be integrated into a complete environment for generating, manipulating, and delivering visual data.

We examine the structure and components of such an environment, including the programming and user interfaces, rendering engines, and image delivery systems. We consider some of the constraints imposed by real-world applications and discuss the problems and issues involved in bringing parallel rendering out of the lab and into production.

This work was supported by the National Aeronautics and Space Administration under Contract No. NAS1-19480 while the author was in residence at the Institute for Computer Applications in Science and Engineering (ICASE), M/S 403, NASA Langley Research Center, Hampton, VA 23681-0001, USA.

Email: tom@icase.edu *World Wide Web:* http://www.icase.edu/~tom/

1 Introduction

Synthesizing high-quality images from abstract geometric or numerical representations of a scene is a computationally demanding task. As the power and availability of general-purpose parallel computer systems have grown, the computer graphics community has become increasingly interested in exploiting them to support sophisticated rendering methods and complex scenes. In numerous projects over the last several years, all of the common rendering techniques (polygon, volume, and terrain rendering, ray tracing, and radiosity methods) have been mapped onto parallel architectures, ranging from tightly-coupled symmetric multiprocessors and data-parallel SIMD arrays to distributed-memory message-passing systems and loosely-coupled networks of workstations. While the performance and efficiency of these implementations have varied widely, there have been enough successes to conclude that carefully-constructed parallel renderers can be an effective way to obtain the needed processing power for demanding applications in computer graphics and data visualization.

Although the rendering process contains ample parallelism at several different levels, the issues involved in developing efficient parallel renderers are complex [8][18], and most of the effort to date has focused on the rendering algorithms themselves and their interactions with specific architectural platforms. The question of integrating parallel renderers into the broader computing environment has often been neglected, and in some cases explicitly ignored. The purpose of this paper is to examine the role of parallel renderers within a broader, application-oriented context, and to discuss some of the issues involved in moving parallel rendering from a research curiosity to a production tool. The focus is on software-based renderers running on general-purpose parallel platforms—we do not consider special-purpose architectures designed specifically to support rendering.

We briefly review some of the applications for which software-based parallel rendering is and is not appropriate, and then examine the overall software architecture needed to support parallel rendering applications. The role of each component within the system is explored in some detail, emphasizing the impact that each has upon the others. We conclude with some thoughts on the challenges and opportunities which await parallel graphics and visualization researchers as we move forward from the current state-of-the-art.

2 Applications of Software-Based Parallel Rendering

Hardware-based rendering engines for workstation-class systems are relatively affordable and provide impressive performance which continues to grow at a dramatic rate. This raises an obvious question: Why bother with software-based renderers on complex architectures when off-the-shelf hardware solutions are readily available? There are several answers.

While dedicated rendering hardware can be applied to many problems in computer graphics, it lacks the flexibility of software-based renderers. Hardware rendering engines usually provide direct support for a restricted class of rendering methods (e.g., polygon rendering), lighting models, and image resolutions. Alternate rendering techniques such as ray tracing, radiosity, and volume rendering often run at much less than interactive rates on these systems, making them obvious candidates for software-based parallel solutions. Software-based renderers can be easily modified to incorporate alternative techniques for illumination, shading, interpolation, composition, etc. They can also support arbitrary classes of geometric primitives, e.g., spheres and parametric surfaces. By exploiting parallelism, software-based renderers may be able to regain some of the performance which they have sacrificed in favor of flexibility.

Since many parallel rendering algorithms exploit pixel-level parallelism by partitioning the image across multiple processors, they are especially well-suited for graphics applications which require high resolution or large amounts of data at each pixel. By adding additional processors, more memory (as well as computing power) can be added to support larger images, multiple views, supersampling, transparency, etc.

Visualization and graphics applications involving very large datasets are also candidates for parallel rendering. Large-scale scientific applications, particularly those involving time-dependent phenomena, can generate results which range from hundreds of megabytes to hundreds of gigabytes in size. These datasets may be too large to process effectively with anything less than a supercomputer-class system, or too cumbersome to move across the network for postprocessing elsewhere. In such cases it may be more practical to perform the visualization and graphics operations in parallel on the system where the data originates, transmitting images, rather than the raw data, back to the user.

With appropriate interfaces, parallel renderers enable parallel application programs to produce live visual output at runtime. This visual feedback is especially helpful with large, complex problems, where it can be employed for debugging or to monitor the progress of executing jobs. Visual output can also play an important role in interactive steering applications, where the user adjusts the execution parameters at runtime to explore larger design spaces or to reach a solution more quickly.

2.1 Limitations

For the majority of graphics applications, software-based parallel rendering is not an appropriate choice. Workstation-class systems provide more than enough power except for the largest problems or the most expensive rendering methods. It also seems unlikely that software renderers running on general-purpose parallel systems will ever be able to compete with commercial hardware rendering engines on a price/performance basis. By tuning the architectural design for a constrained set of graphics operations, hardware engines eliminate redundant components (such as power supplies, circuit boards, and I/O

subsystems), and provide dedicated high-speed data paths among the processing elements and to the frame buffer.

In addition, applications requiring highly interactive response are probably not well-suited to software-based rendering. While the computational demands of virtual reality and real-time simulation (particularly for large datasets or complex scenes) make parallel processing an attractive option, they have stringent requirements on frame rates ($\geq$ 10 fps) and latencies ($\leq$ 10 ms) which are difficult to achieve with software solutions on general-purpose systems. At the very least, hardware support for image output (e.g., [17]) appears to be essential in providing smooth interactive operation with current architectures. This situation may change in coming years as the performance of processors and I/O interfaces continues to improve.

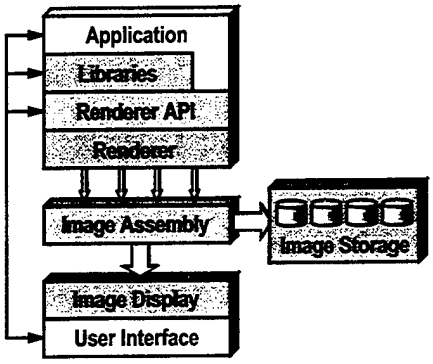

Figure 1. Software architecture for a typical parallel rendering application.

3 Software Architecture for Parallel Rendering

To become useful tools, parallel renderers must interface with other elements in a larger computing environment. Figure 1 shows the principal software components in a typical parallel rendering application. Although not explicitly shown, we assume that most if not all of these will call upon an underlying operating system for services such as memory and process management, I/O, and communications. We discuss each component in some detail in the following sections.

3.1 Application

The application layer defines the overall task to be accomplished. For assistance, it relies upon system and domain-specific libraries, and may invoke the renderer directly for low-level graphics operations. In a purely graphical application, this layer may be simply a thin veneer over the renderer. In other cases, such as numerical simulations or virtual reality, the renderer may be only a small piece in a much larger puzzle.

An important property of the application is its parallel programming paradigm, which in turn may be heavily influenced by the architecture on which it resides. For example, an application could be written in a loosely-coupled MIMD fashion, in which different components are working on significantly different aspects of the problem, while sharing global information and exchanging intermediate results. Or it could be written in a tightly-coupled SIMD mode, performing fine-grained parallel operations in lock-step on regular data structures. A popular programming style for scientific applications is SPMD (Single

Program Multiple Data), in which all of the processors execute the same program, following roughly the same path through the code, with occasional synchronization and communication points.

The programming paradigm adopted by an application has a major influence on all of the software layers below it, which must be able to interface with the application on its terms. The lower level components must either adopt the same paradigm, or provide appropriate interfaces between the application's structure and their own internal strategies.

Another crucial property of the application is its memory reference model. A message-passing application which partitions its data structures across distributed memories may need very different library interfaces than one which relies on global access to shared data.

3.2 Libraries

Most applications depend on a variety of libraries for support services ranging from I/O to numerical solutions. In our context, we are interested in libraries which provide a higher-level interface to the low-level operations supported by the renderer. For example, an application which computes values on a 3D grid may prefer to invoke isosurface routines rather than triangle-drawing primitives. Visualization techniques, geometry modeling, and user-interface operations are among the likely candidates for implementation at the library level. As an intermediary between the application and the renderer, the library layer also provides an opportunity to match the application's programming paradigm and data structures to those used by the renderer.

An important consideration for parallel libraries, and for all of the software layers beneath them, is the ability to run with acceptable performance in a multi-algorithm environment. Many applications are composed of several different computations, or phases, each requiring different memory reference patterns and communication topologies. The library developer must assume that the application programmer or end user will map processes onto processors and adjust communication parameters to suit the application's needs. The communication patterns, task structure, and synchronization requirements of the application may differ markedly from those of parallel visualization algorithms or of the renderer. Thus supporting software layers should not depend on algorithms which require particular mappings of processes onto processors or specific physical communication topologies. This presents a challenge for parallel algorithm developers, since many existing techniques exploit fixed interconnection topologies to obtain good performance. However, the development of robust, topology-independent algorithms also aids cross-platform portability, which is particularly important given the rapid obsolescence of systems and continuing evolution of parallel architectures.

3.3 Renderer API

The renderer's application programming interface (API) is a low-level library which provides access to the graphical operations implemented by the renderer. The API (perhaps with help from the library layer) is responsible for matching the application's programming paradigm, memory access model, and data structures to the underlying parallel algorithms employed by the renderer. A poorly-designed API will inflict additional burdens upon the application and library layers, making the renderer difficult or inconvenient to use.

The design of the API has a fundamental impact on the design of the renderer. To ensure that the API layer does not become a significant source of memory, communication, or computational overheads, the renderer must directly and efficiently support the operations defined by the API. Thus ease-of-use and efficiency considerations suggest that the API layer should be relatively thin, and that the renderer should be designed to accommodate the programming paradigm and data structuring conventions of the prevailing applications.

3.4 Renderer

Many of the parallel rendering algorithms and implementations reported in the literature have assumed that the renderer itself is the driving application. In this scenario, the application layer needs to do little more than read in a scene description, arrange it in memory to suit the rendering algorithm, and dispose of the resulting image. Often the focus has been strictly on the renderer, and I/O and display times have been ignored in reporting the results.

This narrow view of the rendering application has far-reaching algorithmic design consequences. The renderer is free to assume that the entire resources of the system are available for its use, including memory, processing power, and I/O. Thus it is common to read about rendering algorithms which assume that sufficient memory is available to buffer all of the intermediate results at various steps in the rendering pipeline, or to replicate the entire image memory on every processor.

In reality, many of the applications which reside on parallel systems have very demanding resource requirements of their own (if they didn't, they wouldn't need to be on a parallel computer in the first place). To be an effective tool in this environment, a parallel renderer must be modest in its own demands, and this imposes additional constraints on algorithm design [5]. For example, a polygon rendering pipeline must actually be implemented in pipelined fashion, rather than as a series of sequential stages, so that intermediate results can be consumed as they are generated. Hence an application-friendly parallel renderer is likely to exploit both functional parallelism and data parallelism [8]. Likewise, renderers which partition their image data structures can better accommodate memory-intensive techniques such as supersampled antialiasing and transparency with less impact on applications.

3.5 Image Assembly

Since much of the parallelism available in the rendering process occurs at the pixel level, most parallel renderers try to distribute screen-space computations across the available processors. On distributed-memory architectures at least, this implies that a series of partial images must be assembled or composited to produce the final result. The details of this process depend upon the structure of the rendering algorithms, the intended destination of the completed images, and the hardware and software architecture of the underlying system. We find it useful to distinguish between two cases, *internal assembly* and *external assembly* [8].

With internal assembly, the final image is formed in its entirety somewhere within the memory of the parallel system, and is then routed to its destination, which may be a display device, a file, or a remote workstation. Internal assembly implies that sufficient memory must be allocated in one place to accommodate the full image. If every processor allocates this space, then memory consumption may be excessive. If only one processor allocates space for the image, then memory consumption will not be uniform across processors, leading to potential complications for memory-hungry applications. One alternative is to single out a processor to be responsible for image assembly, and to offload other tasks from it, thereby bringing its resource requirements more in line with those of the other processors. Another alternative is to use an auxiliary processor, such as a service or I/O node, to perform this function. In either case, an element of heterogeneity is introduced into the software environment, which often translates into additional complexity for the application designer or end user. For these reasons, external image assembly may be preferable.

With external assembly, the components which make up an image are routed to a remote location (typically their final destination) before being combined into a whole. For example, different segments of an image may be sent from each processor to an addressable frame buffer, in which case the complete image is formed only within the frame buffer's memory system. Or, image components may be written in encoded form to a file, in which case assembly occurs only when the file is decoded for playback.

With either internal or external assembly, image components must be retrieved efficiently from multiple processors and merged into an output stream. To sustain interactive or animation rates for full-screen displays, bandwidths on the order of 100 MB/s may be required. While the internal communication networks of current-generation systems may be able to support these data rates, software overheads introduce additional delays, particularly when large numbers of processors need to combine their results into a single output stream. Merging algorithms must be carefully designed to exploit parallelism in order to provide scalability and to reduce the impact of serial bottlenecks.

3.6 Image Transport, Display, and Storage

Because of the large volume of data involved, getting image streams out of a parallel system and onto a display or storage device is an important consideration. The problem is easiest to address at the hardware level, where devices such as HIPPI frame buffers provide high-bandwidth interfaces to video displays. To provide additional parallelism in the image assembly and display process, several systems (including the CM-2 and nCUBE) have incorporated multi-ported frame buffers [2][17]. The renderer must then assemble the image data into several partial streams (typically one per port) with appropriate global synchronization.

3.6.1 Remote image display

Although directly-attached display devices offer the best performance, they suffer from a number of drawbacks for parallel rendering applications. For one thing, they are a single-user resource, even though most parallel systems support the execution of multiple jobs concurrently. Perhaps more importantly, large-scale parallel systems are scarce commodities, typically serving a large and geographically dispersed user community. For the majority of users, access to a directly-connected display device will be either inconvenient or infeasible. What is needed is a way to transmit the rendered images to the user's desktop at rates which will support interaction and avoid excessive I/O delays for the application. Given the bandwidth requirements outlined above, and the typical performance of congested long-haul networks, this is a challenging problem indeed. However, for local area networks the problem is more manageable, and we have had some success in delivering image streams from parallel systems to desktop workstations [5][7].

Existing networks at most sites provide peak bandwidths on the order of 1–10 MB/s using Ethernet, Fast Ethernet, FDDI, or similar technologies. Sluggish network interfaces, software overheads, and contention with other traffic can easily reduce this by 30-75%. Since this is far short of the 100 MB/s needed for animation, some compromises are clearly in order. Order-of-magnitude reductions in bandwidth requirements can be achieved by resorting to lower-resolution images (640 x 512 *vs.* 1280 x 1024) and reduced color precision (8 bits *vs.* 24). In some applications, lower frame rates (1–10 fps) may also be acceptable, reducing bandwidth demands even further. By combining these strategies, it is possible to reach a level at which Ethernet is a tolerable, if not wholly satisfactory, medium for delivering output streams from parallel renderers.

3.6.2 Image compression

For some applications these compromises are not acceptable, and in other cases the available network bandwidth may still be insufficient. To accommodate these situations, we must resort to data compression techniques, perhaps in combination with some of the other strategies. Although many

methods are available for compressing images and video [3][4], with new developments appearing almost daily, little if any of this work has been done with parallel rendering applications in mind.

The parallel rendering environment imposes some additional requirements and opportunities which are not present in most image compression applications. We have already noted that the image data is likely to be partitioned among multiple processors. This raises the possibility that the compression phase can be performed in parallel on different segments of the image. As the number of processors grows, so does the available parallelism, but the amount of image data per processor decreases, reducing the length of the input string. To complicate matters further, load balancing considerations often favor decompositions which scatter the image data across processors, thereby avoiding hotspots due to local variations in image complexity [8]. Unfortunately, this also limits the ability of compression algorithms to exploit spatial coherence. On the other hand, the image streams generated in many parallel rendering applications do not vary radically from one frame to the next, so the opportunity exists to exploit temporal coherence. On the receiving end, we expect to have a PC- or workstation-class system available to perform the decompression, typically employing a single processor. Thus the computing power available to decompress the data may be one to two orders of magnitude less than that available for compressing it.

With these considerations in mind, an ideal compression scheme for parallel rendering would:

- compress with reasonable speed,
- parallelize well, with minimal interprocessor communication,
- exhibit good compression with relatively short input strings,
- accept arbitrary orderings of the input data,
- exploit temporal coherence, and
- decompress very rapidly.

For computer graphics applications in which image quality is paramount, or for visualization applications in which accuracy is essential, we also insist on lossless compression schemes. In other cases, particularly when bandwidth is low, lossy methods may be acceptable, or even required. We are aware of very little work that has been done on real-time image compression and transmission methods for parallel rendering, making this a fruitful area for future research. Some simple techniques designed for use on local area networks are described in [5] and [9].

Although network-based image transmission lacks the responsiveness of graphics workstations, we are optimistic about the prospects for the future. All of the components involved (processors, network interfaces, network infrastructure, real-time protocols, compression algorithms, etc.) are improving, and we expect that remote graphical interaction with parallel applications will be quite practical in a few years.

3.6.3 Image storage

In some applications, real-time interaction is not a primary concern. For example, sophisticated photorealistic rendering techniques may take many minutes to generate an image, even with parallel methods. Large scientific applications may also take minutes or hours to compute a single iteration or timestep, and the realities of supercomputer scheduling may force large jobs to run in batch queues with unpredictable and inconvenient starting times. In these situations, it is more appropriate to route the image stream to secondary storage for later perusal. For interactive applications, it may be desirable to save a copy of the visual output for future reference, or to take snapshots of particularly interesting frames.

Even with compression, long animation sequences can become quite large, particularly with high quality images. A long simulation could reasonably generate several gigabytes of image data. Particularly for batch applications, we may be able to tolerate increased compression and (possibly) decompression times in exchange for more compact output files. If the image files can be structured appropriately, it may also be possible to take advantage of parallel I/O operations to reduce write times. This is in contrast to networked transmission, where the image data typically needs to be serialized and written to a single socket descriptor.

These differences suggest that the image assembly algorithms, compression methods, and data formats for file storage may need to be different than those for direct display or remote transmission. Thus our parallel rendering systems should be designed in a flexible manner which will accommodate multiple output strategies.

It should be apparent from the forgoing discussion that image handling considerations can impact the design of the renderer itself. For example, the optimum image partitioning strategy from a rendering standpoint could lead to extra communication in the image assembly phase, or reduced effectiveness of the compression algorithms. Thus it is essential to bear in mind the overall performance of the system and the feedback relationships between each of the components.

3.7 User Interface

The ability to interact with a scene and its contents is essential in many computer graphics applications. Static views which are determined *a priori* may be necessary for batch-mode applications, but invariably there are features of interest which are obscured or are difficult to interpret properly with a fixed viewpoint and static lighting. Providing the user with the ability to modify what he is seeing greatly enhances the power of computer graphics. This is especially true for parallel rendering applications, which often involve large, complex, and dynamic scenes.

The user interface should be tightly-coupled to the display software, since the most effective interaction mechanisms are often those which involve direct manipulation of the imagery. With directly-attached hardware displays, the user interface software will need to be integrated into the parallel environment, perhaps residing on a single processor which is dedicated to that task. With remote displays, it is more appropriate, and generally more convenient, to host the user interface code on the receiving workstation, along with the display code. At a minimum, the user interface component is responsible for handling device events and communicating them back to the application. 2-D GUI interfaces are probably best implemented on the workstation side where they can take advantage of existing libraries and operating system support. 3-D GUIs, in which interface objects appear in the scene, may need to be implemented on the rendering side. This poses some interesting issues in modeling and interacting with interface elements which, for load balancing and other reasons, may need to be distributed across multiple processors.

User interaction requests can occur at several different levels in the software hierarchy. For example, changes in viewing parameters and light sources will generally be handled through the renderer's API, while manipulation of geometric models and visualization parameters might be implemented within the library layer. For purposes of debugging and interactive steering, the user interface may communicate directly with the application in order to interrogate and modify its variables and data structures.

As we pointed out in Section 2.1, highly responsive interaction is difficult to achieve with software-based solutions, and placing the display and user interface on a remote desktop only exacerbates the problem. This implies that interaction mechanisms should be designed to cope with high latency and sluggish image updates. To prevent the user from getting too far ahead of the application, event streams may need to be collapsed in order to avoid the overhead of rendering and transmitting images that will be seriously out of sync by the time they arrive.

Another interesting situation arises in dealing with applications which require lengthy computations in order to generate a new image. For example, it may be natural for a numerical simulation to invoke the renderer at the end of an iteration or a time step to display the current state of the computation. If the time between updates exceeds a few seconds, interactivity is effectively lost. There are two principal options for dealing with this situation. One possibility is for the renderer to run as a different process (or at least a different thread), either sharing processors with the application, or perhaps running in its own dedicated pool of processors. The renderer can then respond to user interface requests asynchronously with respect to the application. Unfortunately, the application's data structures are likely to be in an inconsistent state during the midst of an iteration, forcing the renderer to maintain its own copy of scene-related data. This copy must then be updated periodically in coordination with the application.

An alternate strategy, which avoids data copying and separate processes, is to have the user indicate when he wants to interact with the application. The application pauses once it reaches a consistent state, switching control to the renderer which can then operate on stable data using the full processing power of

the system. When the user finishes an interaction sequence, he directs the application to resume its operation. Since neither of these strategies is entirely satisfactory, the choice should be made depending on the application's requirements and user preferences.

Designing a parallel renderer is a complex process which requires careful balancing of competing requirements and numerous tradeoffs. Unfortunately, adopting a system-level view of the process only complicates matters by introducing additional considerations and imposing new constraints. We have identified several of these issues in the above discussion, but there are no doubt others which we have neglected. Finding acceptable compromises will be the key to developing parallel rendering systems which become useful components in the parallel computing toolbox.

4 Challenges and Opportunities for Parallel Rendering Systems

We now turn our attention to several areas which we think will present both challenges and opportunities for parallel rendering research in the next few years. Of particular interest are issues of portability, scalability, and ease-of-use.

4.1 Towards Portability

Portability is a major concern in the development of full-featured systems to support parallel graphics and visualization applications. The amount of code required to supply the needed capabilities is too large to reimplement each time a new architecture comes along. On parallel systems, portability implies much more than just having the code compile cleanly on a new system. Performance considerations often result in algorithms which have architectural assumptions deeply ingrained within them. Transferring the code to a different platform may require substantial re-engineering, or perhaps even reformulation of the problem.

4.1.1 Programming paradigms

One approach to portability is to adopt a lowest-common-denominator programming paradigm. A potential candidate is the data-parallel programming model, which has proven itself to be useful on a variety of architectures. Although many of the concepts originated in the SIMD world, data-parallel algorithms can often be implemented at different granularities to suit the characteristics of the target architecture, and numerous parallel renderers have been implemented using this paradigm.

However, the rendering process involves many irregularities [8] which lend themselves more naturally to SPMD or MIMD implementations, and data-parallel rendering algorithms have achieved their success primarily on SIMD architectures with low communication overheads. Both SPMD and SIMD programming models are common in scientific and engineering applications, although the SPMD

paradigm appears to offer more flexibility for computations involving complex grids and irregular data structures. Despite the advantages of a unified programming model, it is questionable whether portability considerations will outweigh the need to adapt the renderer to the prevailing paradigm on a given architecture.

4.1.2 Access to memory

Another impediment to portability is the diversity of memory access models. At one extreme is global shared memory, typically found on symmetric multiprocessors. At the other is message-passing, used by many distributed-memory architectures as well as networks of workstations. It is generally agreed that message-passing programs are more complex and require more lines of code than their shared-memory counterparts. Message-passing also involves considerable software overheads, even with assistance from dedicated co-processors. Several recent parallel architectures provide hardware support for global addressing of physically distributed memory. The resulting reductions in communication overhead enable algorithmic approaches which are not practical in message-passing environments, and this advantage is apparent in parallel rendering applications [19].

Does this mean message passing is dead? Not necessarily. Locality is still important, and message passing makes it apparent and provides explicit control [15]. For portability purposes, message passing also serves as a lowest common denominator, since it can be implemented with relative ease in shared memory environments.

4.1.3 Parallel programming languages

An alternate avenue to portability is the use of high-level parallel programming languages. Parallel languages provide a common programming paradigm and memory access model across architectures by shifting (some of) the burden of architecture-specific adaptations to the compiler. The most widely accepted example is High Performance Fortran (HPF) [12], which is now supported by a number of vendors. The initial version of the language was intended primarily for data-parallel applications, but current developments are intended to broaden its applicability to irregular problems. Similar efforts involving parallel derivatives of C++ may be more suitable for computer graphics applications, which have traditionally been written in C and C++ to take advantage of more sophisticated data structures. Nonetheless, the availability of HPF on an assortment of architectures presents an opportunity to assess the use of parallel languages to enhance the portability of rendering applications.

4.2 Parallel Rendering on Teraflops and Petaflops Architectures

Emerging supercomputer architectures will require higher levels of parallelism and offer higher performance than anything previously encountered. The first general-purpose teraflops systems (procured for the U.S. Department of Energy's Accelerated Strategic Computing Initiative [1]) will

employ on the order of 10^3–10^4 high-performance commercial microprocessors, in relatively conventional architectural arrangements. To date, most parallel rendering implementations on similar architectures have not scaled well beyond 100–200 processors [6][10][11][13]. We conjecture that this "scalability barrier" arises from inherent properties of the rendering problem.

First of all, typical rendering applications are limited to image resolutions of about 1 megapixel or less. Given that the computations performed at a single pixel represent a relatively fine-grained task, we must aggregate many of them to obtain efficient performance. As we increase the number of processors, the image size usually stays fixed, so task granularity decreases. Since a significant portion of the available parallelism occurs at the pixel level, Amdahl's law implies that a fixed image resolution will limit the ultimate scalability of parallel renderers.

To compound the problem, rendering is inherently communication-intensive. The projection from object-space to image-space results in communication graphs which are dense, scene- and view-dependent, and dynamic (due to user interaction and changing scene content). As the number of processors increases, so do the number of communication operations required at each processor, while the amount of data transferred with each operation decreases. Although store-and-forward aggregation schemes can improve performance by reducing the communication complexity [10][11], they merely substitute less expensive data copying for more expensive data communication, so the inherent overheads persist, although with reduced severity.

We conclude from this that parallel rendering algorithms have inherent limits on scalability which depend more on workstation display technology than on the complexity of the scenes being rendered. Whether these limits will come into play in a given application depends in part on architectural parameters (such as communication overheads and the number of processors), and in part on characteristics of the application (such as the percentage of computation due to screen-space operations).

What are the implications of this for parallel rendering on future architectures? Current projections are that petaflops-class systems will require from 10^5 to 10^6 parallel threads to keep them busy [16]. Unless display resolutions increase dramatically over the next ten years, we will have image-space tasks consisting of at most a few pixels. Will communication systems improve to the point that we can effectively partition our images at this level of granularity? The growing gap between processor speeds and memory access times suggests that this won't be the case.

Does this mean that software-based parallel rendering will not be a viable option, and that we should turn instead to specialized rendering engines which are closely coupled to large parallel systems? Or will we simply run the rendering operations on a subset of the available processors, letting the others go idle? Should we even care about processor utilization, as long as frame rates are high enough to support real-time interaction?

On the other hand, the ability to perform a trillion or quadrillion operations per second may change the way we think about rendering. Perhaps volume rendering, global illumination methods, procedural

textures, 36-bit color, 16x supersampling, and HDTV resolution will all become routine. What new rendering and visualization methods will this massive computing power enable, and how will they benefit the large-scale scientific and engineering applications which will run on these platforms? Finding the answers to these and other questions will be the objective of many new explorations in parallel rendering during the next decade.

4.3 APIs for Parallel Rendering

At the present time, there is no standard API for parallel rendering, and only a handful of parallel renderers provide library interfaces for application programs. To gain wide acceptance, a standard library interface for parallel graphics needs to be developed, in much the same way that MPI [14] has unified disparate message-passing implementations. Whether a single graphics API can support a sufficiently wide range of parallel applications is an open question.

Portability between sequential and parallel platforms is also desirable, and this argues for extending or adapting existing graphics APIs. OpenGL is clearly the *de facto* standard for serial architectures, but it is not obvious that it provides an appropriate basis for parallel rendering. There are two main problem areas: immediate mode rendering, and constraints on rendering order. Very few parallel renderers provide true immediate mode rendering for individual geometric primitives. With dedicated hardware, dropping a polygon description into the head of a pipeline and waiting for the pixels to pop out at the other end is a natural operation. In the parallel environment, however, the overheads involved for synchronization, communication, and image updating make operations at this fine level of granularity very inefficient. Instead, parallel renderers generally accept an entire scene description (or similarly coarse-grained objects) as their input, and often employ considerable internal buffering of intermediate results to obtain satisfactory performance.

OpenGL also insists on maintaining the order of rendered primitives, which is useful for "painting" objects on top of one another. In the parallel environment, the concept of time is much fuzzier, requiring explicit synchronization operations in order to impose global order (except on SIMD systems, where instruction-level synchronization is provided by the hardware). Defining and enforcing global order at the level of individual primitives is such a cumbersome proposition that parallel renderers have almost universally abandoned it, relying instead on other techniques (such as multi-pass rendering) to provide order in those rare cases when an application demands it.

We may also need to extend existing APIs to accommodate new rendering methods. Volume rendering, in particular, is becoming an increasingly important and accepted tool, in part because parallel implementations are making it more practical. Thus it seems useful to support both surface rendering and volume rendering within a single API, or a coordinated set of APIs. While the rendering algorithms for each of these techniques might be very different, there is a good chance that they could share much of the

back-end image handling and display infrastructure, perhaps allowing for volume-rendered and polygon-rendered geometry to co-exist within a single frame.

4.4 High-level Support for Parallel Graphics and Visualization

Although computer graphics is a powerful tool, it remains underutilized by most application programmers. This is due in part to the specialized knowledge which is needed to set up things like viewing parameters and modeling transformations, but it also reflects the relatively tedious level of detail at which most graphics APIs operate. What is needed is a more intuitive and simpler way to interface with the renderer, using operations on application-level data structures such as arrays or grids. When displaying an isosurface becomes as easy as calling "printf", computer graphics will come into its own as a tool for application developers.

Going one step further, we can imagine the insertion of compiler directives which would indicate that a particular data structure, say a 3D grid, is to be displayed visually at runtime. By enabling an appropriate compiler option or runtime switch, a GUI interface could pop up with a variety of visualization options, ranging from cutting planes and isosurfaces to streamlines and volume rendering. While none of this is specific to the parallel environment, the power of a parallel system can make high-level visualization operations more responsive. Compilers for languages such as HPF also have fairly detailed knowledge of architectural parameters and data flow within an application, and it may be possible to exploit this information to generate efficient code for graphical operations.

5 Conclusion

With some effort, software-based rendering techniques can be successfully used on current parallel architectures to generate visual output from application programs. Improvements in usability and performance will derive from a broader, system-level view of the rendering process and its various components. Important challenges remain, particularly in the areas of scalability, portability, ease-of-use, and image handling. Meeting these challenges will allow software-based parallel renderers to evolve from research curiosities into viable tools for production computing.

Acknowledgments

The author would like to thank Kwan-Liu Ma and Piyush Mehrotra for sharing their insights on several of the topics covered in this paper.

References

[1] *ASCI.* http://www.llnl.gov/asci/, Lawrence Livermore National Laboratory.

[2] R. E. Benner, Parallel graphics algorithms on a 1024-processor hypercube. *Proceedings of the 4th Conference on Hypercubes, Concurrent Computers, and Applications*, Vol. I, 133-140, 1989.

[3] V. Bhaskaran and K. Konstantinides, *Image and Video Compression Standards: Algorithms and Architectures.* Kluwer Academic Publishers, Boston, 1995.

[4] R. J. Clarke, *Digital Compression of Still Images and Video.* Academic Press, London, 1995.

[5] T. W. Crockett. Design considerations for parallel graphics libraries. *Proceedings of the Intel Supercomputer Users Group,* 1994 Annual North America Users Conference, San Diego, CA, 3-14, June 1994.

[6] T. W. Crockett and T. Orloff, Parallel polygon rendering for message-passing architectures. *IEEE Parallel and Distributed Technology, 2*(2), 17-28, Summer 1994.

[7] T. W. Crockett, R. G. Wilmoth, and P. J. Crossno. *Runtime Visualization for Parallel Applications.* Supercomputing '94 exhibit, http://www.icase.edu/hilites/tom/RuntimeVis.html, Nov. 1994.

[8] T. Crockett. Parallel rendering. In *Encyclopedia of Computer Science and Technology*, Vol. 34, Supp. 19, A. Kent and J. G. Williams, eds., Marcel Dekker, 335–371, 1996.

[9] T. W. Crockett. *The YMF Image Format.* http://www.icase.edu/~tom/YMF/YMF.html, 1996.

[10] D. A. Ellsworth. A new algorithm for interactive graphics on multicomputers. *IEEE Computer Graphics and Applications, 14*(4), 33–40, July 1994.

[11] T.-Y. Lee, C. S. Raghavendra, and J. N. Nicholas. Image composition schemes for sort-last polygon rendering on 2-D mesh multicomputers. *IEEE Transactions on Visualization and Computer Graphics, 2*(3), 202-217, Sept. 1996.

[12] D. B. Loveman. High performance fortran. *IEEE Parallel and Distributed Technology, 1*(1), 25-42, Feb. 1993.

[13] U. Neumann. Communication costs for parallel volume-rendering algorithms. *IEEE Computer Graphics and Applications, 14*(4), 49-58, July 1994.

[14] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra. *MPI: The Complete Reference.* MIT Press, Cambridge, Mass., Nov. 1995.

[15] T. A. Ngo and L. Snyder. On the influence of programming models on shared memory computer performance. *Proceedings Scalable High Performance Computing Conference SHPCC-92*, IEEE Computer Society Press, 284-291, Apr. 1992.

[16] T. Sterling. Draft findings. *Petaflops Architecture Workshop*, http://www.aero.hq.nasa.gov/hpcc/petaflops/petasoft96/paws.outcome/findings.html, Oxnard, California, Apr. 1996.

[17] G. Stoll, B. Wei, D. Clark, E. W. Felten, and K. Li. Evaluating multi-port frame buffer designs for a mesh-connected multicomputer. *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, ACM SIGARCH, 96-105, June 1995.

[18] S. R. Whitman, A survey of parallel algorithms for graphics and visualization. In *High Performance Computing for Computer Graphics and Visualisation,* M. Chen, P. Townsend, and J. A. Vince, eds., Springer-Verlag, London, 1996.

[19] S. Whitman, private communication, Mar. 1996.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1996	Contractor Report

4. TITLE AND SUBTITLE
BEYOND THE RENDERER: SOFTWARE ARCHITECTURE FOR PARALLEL GRAPHICS AND VISUALIZATION

5. FUNDING NUMBERS
C NAS1-19480
WU 505-90-52-01

6. AUTHOR(S)
Thomas W. Crockett

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Institute for Computer Applications in Science and Engineering
Mail Stop 403, NASA Langley Research Center
Hampton, VA 23681-0001

8. PERFORMING ORGANIZATION REPORT NUMBER
ICASE Report No. 96-75

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Langley Research Center
Hampton, VA 23681-0001

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
NASA CR-201637
ICASE Report No. 96-75

11. SUPPLEMENTARY NOTES
Langley Technical Monitor: Dennis M. Bushnell
Final Report
Proceedings of the First Eurographics Workshop on Parallel Graphics and Visualization.

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified–Unlimited

Subject Category 60, 61

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

As numerous implementations have demonstrated, software-based parallel rendering is an effective way to obtain the needed computational power for a variety of challenging applications in computer graphics and scientific visualization. To fully realize their potential, however, parallel renderers need to be integrated into a complete environment for generating, manipulating, and delivering visual data.

We examine the structure and components of such an environment, including the programming and user interfaces, rendering engines, and image delivery systems. We consider some of the constraints imposed by real-world applications and discuss the problems and issues involved in bringing parallel rendering out of the lab and into production.

14. SUBJECT TERMS	15. NUMBER OF PAGES
computer graphics; visualization; parallel rendering; system software; parallel computing	19
	16. PRICE CODE A03

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified		