Rendering and Visualization in Parallel Environments

Dirk Bartz, Bengt-Olaf Schneider, and Claudio Silva

WSI/GRIS, University of Tübingen (Email: bartz@gris.uni-tuebingen.de),
IBM T.J. Watson Research Center (Email: bosch@us.ibm.com, csilva@watson.ibm.com)

Abstract

The continuing commoditization of the computer market has precipitated a qualitative change. Increasingly powerful processors, large memories, big harddisk, high-speed networks, and fast 3D rendering hardware are now affordable without a large capital outlay. A new class of computers, dubbed Personal Workstations, has joined the traditional technical workstation as a platform for 3D modeling and rendering. In this tutorial, attendees will learn how to understand and leverage both technical and personal workstations as components of parallel rendering systems.

The goal of the tutorial is twofold: Attendees will thoroughly understand the important characteristics workstations architectures. We will present an overview of different workstation architectures, with special emphasis on current technical and personal workstations, addressing both single-processors as well as SMP architectures. We will also introduce important methods of programming in parallel environment with special attention how such techniques apply to developing parallel renderers.

Attendees will learn about different approaches to implement parallel renderers. The tutorial will cover parallel polygon rendering and parallel volume rendering. We will explain the underlying concepts of workload characterization, workload partitioning, and static, dynamic, and adaptive load balancing. We will then apply these concepts to characterize various parallelization strategies reported in the literature for polygon and volume rendering. We abstract from the actual implementation of these strategies and instead focus on a comparison of their benefits and drawbacks. Case studies will provide additional material to explain the use of these techniques.

The tutorial will be structured into two main sections: We will first discuss the fundamentals of parallel programming and parallel machine architectures. Topics include message passing vs. shared memory, thread programming, a review of different SMP architectures, clustering techniques, PC architectures for personal workstations, and graphics hardware architectures. The second section builds on this foundation to describe key concepts and particular algorithms for parallel polygon rendering and parallel volume rendering.

For updates and additional information, see
http://www.gris.uni-tuebingen.de/~bartz/eg99tut

© The Eurographics Association and Blackwell Publishers 1999. Published by Blackwell Publishers, 108 Cowley Road, Oxford OX4 1JF, UK and 350 Main Street, Malden, MA 02148, USA.

Preliminary Tutorial Schedule

Part One: Foundations

Introduction (Bartz/5 minutes)

Personal Workstations (Schneider/45 minutes)

Technical Workstations (Bartz/45 minutes)

Parallel Programming (Bartz/70 minutes)

Part Two: Rendering

Parallel Polygonal Rendering (Schneider/45 minutes)

Parallel Volume Rendering (Silva/45 minutes)

Part Three: Case Study

The PVR System (Silva/30 minutes)

Q+A (10 minutes)

© The Eurographics Association and Blackwell Publishers 1999.

PART ONE

Foundations

1. Personal Workstations

1.1. Introduction

The advent of powerful processors and robust operating systems for PCs has sparked the creation of a new type of compute platform, the Personal Workstation (PWS). Several vendors, including Compaq, HP, and IBM, sell systems that are targeted at market segments and applications that till only a few years ago were almost exclusively the domain of UNIX-based technical workstations [105]. Such applications include mechanical and electrical CAD, engineering simulation and analysis, financial analysis, and digital content creation (DCC). PWSs are rapidly adopting many features from UNIX workstations, such as high-performance subsystems for graphics, memory, and storage, as well as support for fast and reliable networking. This development creates the opportunity to leverage the lower cost of PWSs to attack problems that were traditionally in the domain of high-end workstations and supercomputers. We will start with an overview of the state of the technology in PWSs and their utility for building parallel rendering systems. Then we will discuss how to improve parallel rendering performance by enhancing PWS subsystems like disks or network connections.

1.2. Personal Workstations

1.3. Architecture

In accordance with the intended application set, PWSs constitute the high-end of the PC system space. Figure 1 shows the architecture of a typical Personal Workstation.

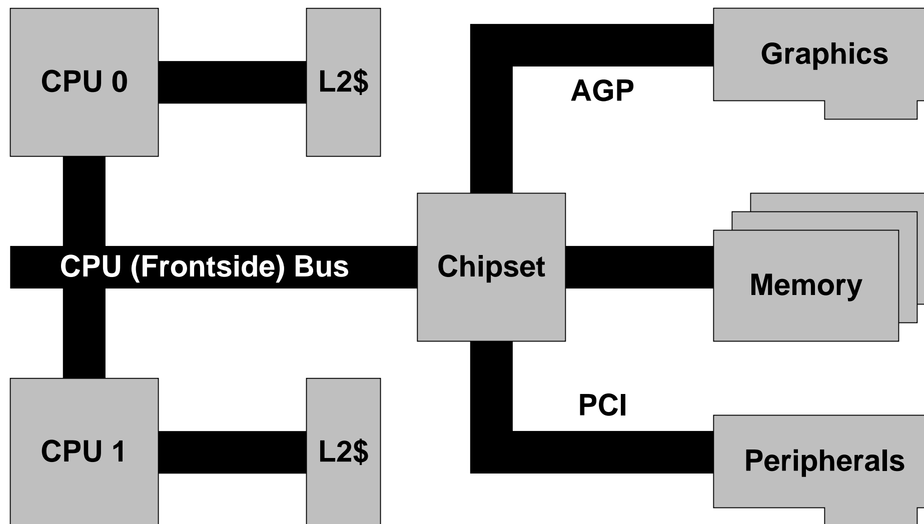

Figure 1: *Architecture of a PWS.*

The system contains one or two Pentium II processors, large L2 caches (up to 512 kBytes) and main memory (32 MBytes up to several GBytes). If configured with multiple CPUs, the system acts as a symmetric multiprocessor (SMP) with shared memory. As is well known, shared memory architectures have only limited scalability due to finite access bandwidth to memory. Current PWSs only support dual-processor configurations.

The chipset connects the main processor(s) with other essential subsystems, including memory and peripherals. Among the techniques employed to improve the bandwidth for memory accesses are parallel paths into memory [2] and faster memory technologies, e.g. Synchronous DRAM (SDRAM) [62]. Intel has announced that its next generation processor will use Rambus (RDRAM) technology to increase the available memory bandwidth.

The graphics adapter is given a special role among the peripherals due to the high bandwidth demands created by 3D graphics.

© The Eurographics Association and Blackwell Publishers 1999.

Integer performance:	650 MIPS
Floating point performance:	250 MFLOPS
Memory bandwidth:	150 MBytes/sec
Disk bandwidth:	13 MBytes/sec

Table 1: *Approximate peak performance data for a Personal Workstation.*

Token Ring 16 Mbit/sec:	14-15 Mbit/sec
Ethernet 10 Mbit/sec:	7-8 Mbit/sec
Ethernet 100 Mbit/sec:	90 Mbit/sec
Ethernet 1 Gbit/sec:	120 Mbit/sec

Table 2: *Peak bandwidth between Personal Workstations for different LAN technologies.*

The Accelerated Graphics Port (AGP) [3] provides a high-bandwidth path from the graphics adapter into main memory. The AGP extends the basic PCI bus protocol with higher clock rate and special transfer modes that are aimed at supporting the storage of textures and possibly z-buffers in main memory, thus reducing the requirements for dedicated graphics memory.

The graphics adapter itself supports at least the OpenGL functionality for triangle setup, rasterization, fragment processing [13] as well as the standard set of 2D functions supported by Windows. Currently, most low-end and mid-range graphics adapters rely on the CPU to perform the geometric processing functions, i.e. tessellation of higher-order primitives, vertex transformations, lighting and clipping. However, a new class of high-end PC graphics adapters is emerging that implement the geometry pipeline in hardware. Hardware-supported geometry operations are important because rasterizers reach performance levels (several million triangles/sec and several 10 million pixels/sec) that cannot be matched by the system processor(s). Also, geometry accelerators can usually provide acceleration more economically than the CPU, i.e. lower $/MFlops, while freeing the CPU for running applications. However, geometry accelerators will only deliver significant improvements to application performance if the application workload contains a large portion of graphics operations. Many applications (and application-level benchmarks) contain only short bursts of graphics-intensive operations.

Balancing the system architecture requires fast disk, e.g. 10,000 rpm SCSI disk drives, and networking subsystems, e.g. 100 Mbit/sec or 1Gbit/sec Ethernet.

1.3.1. Parallel Configurations

For the purposes of parallel rendering we will be considering two forms of parallelism: tightly coupled processors in a SMP configuration (as shown in Figure 1) and a cluster of workstations connected over networks. While in a single-processor machine CPU performance is often the most important factor in determining rendering performance, parallel configurations add specific constraints to the performance of parallel rendering algorithms. For SMP workstations, the performance is affected by memory and disk bandwidth. For workstation clusters, the disk and network bandwidth are the most important parameters influencing the rendering performance. The next section provides concrete values for these parameters.

1.3.2. Performance

To illustrate the performance that can be expected from a PWS we provide approximate performance data in Table 1.

These data were measured with an in-house tool on a preproduction workstation configured with a Pentium II Xeon processor running at 450 MHz, 512 KBytes of L2 cache, Intel 440GX chipset, 256 MBytes of 100 MHz SDRAM system memory and a 9.1 GByte Ultra-2 SCSI disk. The system ran Windows NT 4.0 with Service Pack 4. Note that many factors affect the actual performance of workstations, amongst them BIOS level, memory architecture and core logic chipset.

We have also conducted measurements of networking performance using various local area network technologies (Table 2). These measurements consisted of transferring large data packets and used the TCP/IP stack that is part of Windows NT 4.0. Note that the observed bandwidth for Gigabit-Ethernet is far below the expected value. A likely source for this shortfall is inefficiencies in the implementation of the TCP/IP stack and the resulting high CPU loads. It is well known that such inefficiencies can result in severe performance degradations [33] and we expect that a better TCP/IP stack would raise the transfer rate.

© The Eurographics Association and Blackwell Publishers 1999.

1.3.3. PWS Market Trends

So far we have reviewed technical characteristics of PWSs. When selecting a workstation platform technical issues are but one factor.

The developments in the PWS market reflect the PWS's dual-inheritance from Unix workstations and PCs.

As the NT workstation markets matures the price gap between the best performing systems and the systems with best price-performance appears to be closing. This is a known trend know from the desktop PC market which has turned into a commodity market. The PWS market is assuming characteristics of a commodity market with surprising speed, i.e. most products are very similar and have to compete through pricing, marketing and support offerings.

At the same time, PWSs remain different from desktop PCs – and are similar to Unix workstations – in that application performance (in contrast to servicability and manageability) is the primary design and deployment objective. Most purchasing decisions are heavily based on the results in standard and customer-specific application benchmarks.

A particularly interesting question is whether PWSs offer inherently better price-performance than traditional Unix workstations. Over the period that both workstation types participated in the market (1996-1999), NT workstations as a whole have consistently delivered better price-performance than Unix workstations for standard benchmarks. Only recently (mid 1998) Unix workstation are beginning to reach the same price-performance levels. It is unclear whether this constitutes a reversal of the earlier trend or whether the gap will be restored when Intel delivers its next generation processors. Another explanation for the narrowing of this gap is that NT workstations are starting to include high-performance subsystems that are required for balanced systems (see below).

1.4. Building Parallel Renderers from Personal Workstations

Parallel rendering algorithms can be implemented on a variety of platforms. The capabilities of the target platform influence the choice of rendering algorithms. For instance the availability of hardware acceleration for certain rendering operations affects both performance and scalability of the rendering algorithm.

Several approaches to implementing parallel polygon rendering on PWSs with graphics accelerators have been investigated in [107]. It should be noted that this analysis does not consider pure software implementations of the rendering pipeline; rasterization was assumed to be performed by a graphics adapter.

This is in contrast to software-only graphics pipelines. Such approaches lead to more scaleable rendering systems, even though both absolute performance and price-performance are likely to be worse than the hardware-accelerated implementation. In [128] parallel software renderers have shown close to linear speedup up to 100 processors in a BBN Butterfly TC2000 even though the absolute performance (up to 100,000 polygons/sec) does not match the performance available from graphics workstations of equal or lower cost. However, software renderers offer more flexibility in the choice of rendering algorithms, e.g. advanced lighting models, and the option to integrate application and renderer more tightly.

Following the conclusions from [107] we will now look at the various subsystems in a PWS that may become a bottleneck for parallel rendering. In part, PWSs have inherited these bottlenecks from their desktop PC ancestors. For example, both memory and disk subsystems are less sophisticated than those of traditional workstations. We will also discuss the merit of possible improvements to various subsystems with respect to parallel rendering performance.

Applications and Geometry Pipeline. As pointed out above, CPU portion of the overall rendering time scales well with the number of processors. Therefore, it is desirable to parallelize rendering solutions with a large computational component. Advance rendering algorithms such as advanced lighting algorithms or ray-tracing will lead to implementations that scale to larger numbers of processors.

Processor. Contrary to initial intuition, the performance of CPU and rasterizer does not significantly influence the overall rendering performance. Therefore, parallel rendering does not benefit from enhancements to the CPU, such as by higher clock frequency, more internal pipelines or special instructions to accelerate certain portions of the geometry pipeline. However as stated earlier, faster CPUs may benefit the applicationŠs performance.

Memory Subsystem. Currently, memory bandwidth does not limit rendering performance as much as disk and network performance. We expect that memory subsystems will keep increasing their performance over time and retain their relative performance compared to disks and networks. Therefore, more sophisticated memory subsystems, like [2], will not improve parallel rendering performance.

Disk Subsystem. The disk subsystem offers ample opportunity for improvements over the standard IDE or SCSI found in todayŠs PWSs. Faster disk subsystems, e.g. SSA [1] or RAID 0 (disk striping), can be used to alleviate this problem.

© The Eurographics Association and Blackwell Publishers 1999.

Graphics Subsystem. In workstation clusters the use of graphics accelerators with geometry accelerators can be beneficial. For applications with mostly static scenes, e.g. walkthroughs or assembly inspections, the use of retained data structures like display lists can reduce the bandwidth demands on system memory as geometry and lighting calculations are performed locally on the adapter. In SMP machines or for single-frame rendering faster graphics hardware will not provide large rendering speed-ups.

Network. In clusters, a slow network interconnect can become the dominant bottleneck. Increasing the network bandwidth by an order of magnitude will alleviate that problem. As stated above, current shortcomings of the protocol implementations prevent full realization of the benefits of Gigabit-Ethernet under Windows NT. Alternative technologies, like Myrinet [39] promise higher sustained bandwidth than Ethernet. However, these technologies are either not available under Windows NT or have not yet been developed into a product. Prototype implementations under Unix (Linux) have demonstrated the advantages of such networks.

1.5. Conclusion

As Personal Workstations are emerging as an alternative to traditional workstations for technical applications they are frequently considered as building blocks for affordable parallel rendering.

Even though PWS are used for parallel rendering in at least one commercial rendering package [57], its actual implementation is hampered by the lack of efficient networking technologies and insufficient disk performance. Improving these subsystems is possible but will result in more expensive systems, eliminating some of the perceived cost advantage of PWS over traditional workstation.

2. Technical Workstations

In this Section, we discuss general aspects of parallel environments. Although our tutorial covers PCs and workstations, we will focus in this Section only on workstation environments. However, most of the information on software aspects (message passing, process communication, and threads) is applicable to all UNIX environments (e.g. Linux).

The following Sections will discuss the different parallel approaches, architectures, and programming models for parallel environments.

2.1. Parallel Approaches

Three basic approaches are available for parallel environments. The first approach connects different computers via a network into a cluster of workstations (or PCs). On each individual computer processes are started to perform a set of tasks, while communication is organized by exchanging messages via UNIX sockets, message passing (e.g. PVM), or - more recently - via the Internet. We call this type a loose coupled system, sometimes referred as a distributed processing system.

The second approach consists of a single computer, which contains multiple processing elements (PE which actually are processors). These processing elements are communicating via message passing on an internal high-speed interconnect, or via memory. This type is called a tight coupled system. In contrast to the first approach, communication is faster, usually more reliable, and - in the case of a shared memory system - much easier to handle. However, depending of the interconnection system, the number of processing elements is limited.

The third basic approach is a fusion of the first two approaches. We generally can combine tight or loose coupled systems into a hyprid coupled system. However, in most cases we will loose the advantages of a tight coupled system.

2.2. Taxonomy

Flynn developed a taxonomy to classify the parallel aspects of the different (more or less) parallel systems. However, this taxonomy actually only applies to tight coupled systems.

Flynn distinguishes two basic features of a system, the instruction stream (I) - which is code execution - and the data stream (D) - which is the data flow. These features are divided into a single (S) or multiple stream (M). In a single instruction stream, only one instruction can be individually performed by a set of processors, while a multiple instruction stream can perform different instructions at the same time. If we have a single data stream, only this data can be computed or modified at the same time. With a multiple data stream, more than one data element can be processed.

Overall, we have four different types of parallel processing:

© The Eurographics Association and Blackwell Publishers 1999.

- **SISD** - is the standard workstation/PC type. A single instruction stream of a single processor is performing a task on a single data stream.
- **SIMD** - is the massively-parallel, or array computer type. The same instruction stream is performed on different data. Although a number of problems can easily mapped to this architecture (e.g. matrix operations), some problems are difficult to solve with SIMD systems.
  Usually, these systems cost hundreds of thousands of US$ one of the reasons these machines are not covered by this tutorial.
- **MISD** - is not a useful system. If multiple instructions are executed on a single data stream, it will end up in a big mess. Consequently, there are no computer systems using the MISD scheme.
- **MIMD** - is the standard type of a parallel computer. Multiple instruction streams perform their task on their individual data stream.

2.3. Memory Models

Many aspects of parallel programming depend on the memory architecture of a system, and many problems arise from a choosen memory architecture. The basic question is if the memory is assigned to the processor level, or if the memory is assigned on system level. This information is important for the distribution of a problem to the system. If all memory - except caches - is accessible from each part of the system - memory is assigned on system level, we are talking of a shared memory system. In case the individual processing elements can only access their own private memory - memory is assigned on processor level, we are talking of a distruted memory system. Shared memory systems are further divided into UMA (Uniform Memory Access) systems (not interchangeable with Uniform Memory Architecture), and into NUMA (Non-Uniform Memory Access) systems.

2.3.1. Distributed Memory Systems

In distributed memory systems, the memory is assigned to each individual processor. At the beginning of the processing, the system is distributing the tasks and the data through the network to processing elements. These processing elements receive the data and their task and start to process the data. At some point, the processors need to communicate with other processors, in order to exchange results, to synchronize for periphery devices, and so forth. Finally, the computed results are sent back to the appropriate receiver and the processing element waits for a new task. Workstation clusters fit into this category, because each computer has its individual memory, which is not accessible from its partner workstations within the cluster. Furthermore, each workstation can distribute data via the network.

Overall, it is important to note that communication in a distributed memory system is expensive. Therefore, it should be reduced to a minimum.

2.3.2. Shared Memory Systems

UMA systems contain all memory† in a more or less monolithic block. All processors of the system access this memory via the same interconnect, which can be a crossbar or a bus (Figure 2). In contrast, NUMA systems are combined of two or more UMA levels which are connected via another interconnect (Figure 3). This interconnect can be slower than the interconnect on the UMA level. However, communication from one UMA sub-system to another UMA sub-system travels through more than one interconnection stage and therefore, takes more time than communication within one UMA sub-system.

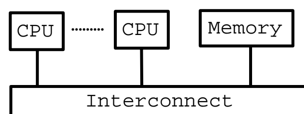

Figure 2: *Uniform Memory Access*

If UMA systems have a better communication, why should we use NUMA systems? The answer is that the possibilities to extend UMA systems are limited. At some point the complexity of the interconnect will rise into infinity, or the interconnect will not be powerfull enough to provide sufficient performance. Therefore, a hierarchy of UMA sub-systems was introduced.

† We are talking of main memory. Processor registers, caches, or harddiscs are not considered as main memory.

© The Eurographics Association and Blackwell Publishers 1999.

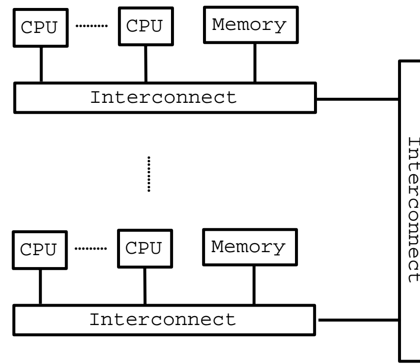

Figure 3: *Non-Uniform Memory Access*

2.4. Programming Models

So far, we jave introduced different approaches of parallelization (loose coupled or distributed processing, tight-coupled processing, and hyprid models of loose- or tight-coupled processing) and different memory access architectures. In this Section, we add two different paradigms for the programming of parallel environments.

2.4.1. Message-Passing

This programming paradigm connects processing entities to perform a joined task. As a matter of principle, each processing entity is an individual process running on a computer. However, different processes can run on the very same computer, especially, if this computer is a multi-processor system. The underlying interconnection topology is transparent from the users point of view. Therefore, it does not make a difference in programming, if the parallel program which communicates using a message-passing library runs on a cluster of workstations, on a distributed memory system(e.g. the Intel Paragon), or on a shared-memory system (e.g. the HP Convex/SPP).

For the general process of using a message-passing system for concurrent programming it is essential to manually split the problem to be solved into different more or less independant sub-tasks. These sub-tasks and their data are distributed via the interconnect to the individual processes. During processing, intermediary results are sent using the explicit communication scheme of message-passing.

Considering the high costs using the network, communication must be reduced to a minimum. Therefore, the data must be explicitly partitioned. Finally, the terminal results of the processing entities are collected by a parent process which returns the result to the user.

Their are several message-passing libraries around. However, most applications are based on two standards, which are explained in Section 3.3 and Section 3.2; the PVM3 library (Parallel Virtual Machine) and the MPI standard (Message Passing Interface).

2.4.2. Threading

A more recent parallel programming paradigm is the thread model. A thread is a control flow entity in a process. Typically, a sequential process consists of one thread; more than one thread enable a concurrent (parallel) control flow. While the process provides the environment for one or more threads - creating a common address space, a synchronization and execution context - the individual threads only build a private stack and program counters. The different threads of a single process communicate via synchronization mechanisms and via the shared memory.

Sometimes the concept of light-weight processes (LWP) is used as a synonym for threads. However, a LWP actually is a physical scheduling entity of the operating system, in a way the physical incarnation of the logical concept of a thread.

In contrast to message passing, threading is only possible on multi-processor systems[‡] Moreover, multi-processor systems need a shared-memory architecture, in order to provide the same virtual address space.

‡ There are some thread models which run on distributed memory systems, or even on workstation clusters. However, there is usually no access to a shared memory, thus limiting communication severely.

© The Eurographics Association and Blackwell Publishers 1999.

Besides easy communication and data exchange using the shared memory, switching between different threads is much cheaper/faster than switching between individual processes. This is due to the shared address space, which is not changed during a thread switch.

Basically, there are three different kinds of implementations for threads. There is a user thread model, a kernel thread model, and a mixed model. The user thread model is usually a very early implementation of a thread package. All thread management is handled by the thread library; the UNIX kernel only knows the process, which might contain more than one thread. This results in the situation that only one thread of a process is executed at any particular time. If you are using threads on a single processor workstation, or your threads are not compute-bound, this is not a problem. However, on a multi-processor system, we do not really get a concurrent execution of multiple threads of one process. On the other hand, this implementation model does not require a modification of the operating system kernel. Furthermore, the management of the threads does not require any kernel overhead. In Pthread terminology, this model is called all-to-one-scheduling.

In contrast to user threads, each kernel thread (on Solaris systems a kernel thread is called a light-weight process, on SGI systems a sproc) is known to the operating system kernel. Consequently, each kernel thread is individually schedulable. This results in a real concurrent execution on a multi-processor, which is especially important for compute-bound threads. However, allocation and management of a kernel thread introduces significant overhead to the kernel, which eventually might lead to a bad scaling behaviour. Pthread terminology denotes this model to be one-to-one-scheduling.

As usual, the best solution is probably a mixed model of user and kernel threads. The threads are first scheduled by the thread library (user thread scheduling). Thereafter, the threads scheduled by the library are scheduled as kernel threads. Threads that are not compute-bound (e.g. performing I/O) are preempted by the scheduling mechanism of the library, while only compute-bound threads are scheduled by the kernel, thus enabling high-performance concurrent execution. In Pthread terminology, this model is called the many-to-one or some-to-one scheduling.

2.5. Example Architectures

In Table 3, we present a rough and incomplete overview of different SMP technical workstations. Please note that all price information is selected from the different web sites of the different companies. Therefore, discounts and sales tax are not included. Prices are in US$ 1.8/US$ 1).rapidly and is probably already outdated when you read these tutorial notes. It is just meant to give a rough picture of a small variety of mid-range systems.

More or less, we always tried to configure a standard system with a four GB harddisc, no graphics subsystem (if possible), no monitor, and 128 MB of main memory. (This may appear not enough memory, but who buys memory from the system vendor anyway?).

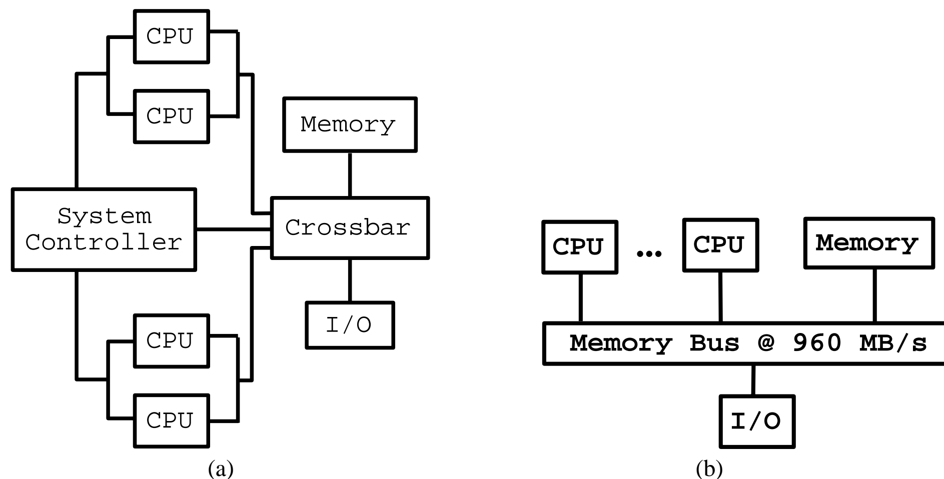

Figure 4: *(a) Basic Sun Enterprise 450 architecture; (b) Basic HP D-class/J-class architecture.*

© The Eurographics Association and Blackwell Publishers 1999.

Sun Enterprise 450

Figure 4a gives an overview of the Sun Ultra Enterprise 450 architecture. Up to four processors are connected via a crossbar to the UMA memory system and to the I/O system. The processors are managed via the system controler. On Sun workstations/servers, pthreads are available as mixed model implementation (Solaris 2.5 and above). OpenMP is not yet available on Sun systems. However, Sun has endorsed OpenMP an will soon provide the necessary compilers.

Hewlett-Packard K/D/J-class architecture

In Figure 4b, the basic architecture of K-class, D-class and J-class architecture of Hewlett-Packard is shown. Up to two processors for D/J-class systems, and up to six processors for the K-class systems are connected via the memory bus to the UMA memory system and the I/O system . Similar to this architecture, the K-class servers can connect up to four processors. On HP-UX 10.30, pthreads are available as a kernel model. Older versions implement a user model. Hewlett-Packard uses third party OpenMP tools (KAI). This toolset is scheduled for 6/99.

Silicon Graphics Octane architecture

The processor boards of the SGI Octane architecture contain up to two processors and the UMA memory system. These boards are connected via a crossbar with the Graphics system and the I/O system (Figure 5a). Pthreads are available for IRIX 6.3 and above, where pthreads are available as patch set for IRIX 6.2. On all implementations, a mixed model is used. The MIPSpro compiler version 7.3 (scheduled for 2Q/99) will support C/C++ OpenMP. Fortran is already supported by available compilers.

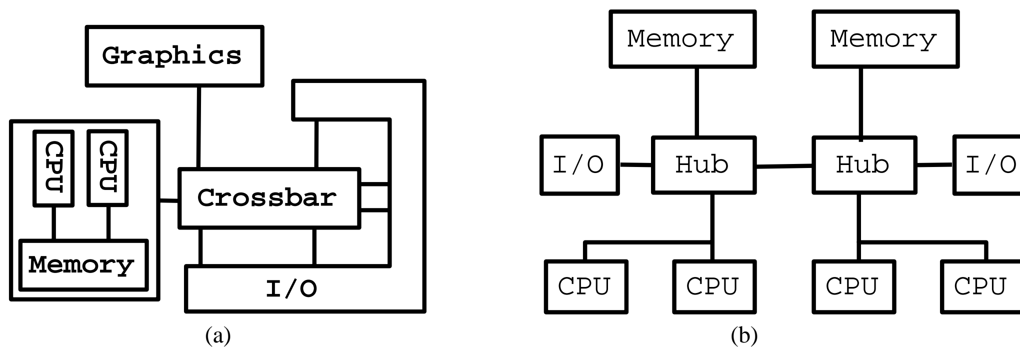

Figure 5: *(a) Basic SGI Octane architecture; (b) Basic SGI Origin 200 architecture.*

Silicon Graphics Origin200 architecture

In contrast to the SGI Octane, no crossbar is used for the Origin200 architecture. The single tower configuration (up to two processors) connects the processors with the UMA memory system and the I/O system via a hub interconnect. For the four processors configuration, a “Craylink” interconnect links two two processors towers system to a Non-Uniform Memory Access (NUMA) system (Figure 5b). In the case of the Origin200, a cache-coherent NUMA scheme is implemented, in order to provide a consistant memory view for all processors. Pthreads are available for IRIX 6.3 and above, where pthreads are available as patch set for IRIX 6.2. On all implementations, a mixed model is used. The MIPSpro compiler version 7.3 (scheduled for 2Q/99) will support C/C++ OpenMP. Fortran is already supported by available compilers.

3. Parallel Programming

3.1. Concurrency

There are some differences between programming of sequential processes and concurrent (parallel) processes. It is very important to realize that concurrent processes can behave completely differently, mainly because the notion of a sequence is not really available on process level of thread-parallel process, or the overall parallel process of a message-passing parallel program. However, the notion of a sequence is available on thread level, which compares to an individual process of a parallel message-passing program. We depict this level of threads or individual processes as level of processing entities.

First of all, the **order of sequential processes** is determined at all times. In parallel processes, however, it is not. There

© The Eurographics Association and Blackwell Publishers 1999.

Vendor/Model	CPU(s)	[N]UMA	Interconnect	Max. Memory
Sun/Enterprise 450	1 @300 MHz	UMA	crossbar @1.6 GB/s	4 GB
	2 @300 MHz	UMA	crossbar @1.6 GB/s	4 GB
	4 @300 MHz	UMA	crossbar @1.6 GB/s	4 GB
HP/J Class J2240	2 @236 MHz	UMA	bus @960 MB/s	1 GB
HP/D Class D390	1 @240 MHz	UMA	bus @960 MB/s	3 GB
	2 @240 MHz	UMA	bus @960 MB/s	3 GB
HP/K Class K580	1 @240 MHz	UMA	bus @960 MB/s	8 GB
(1GB of memory)	2 @240 MHz	UMA	bus @960 MB/s	8 GB
	4 @240 MHz	UMA	bus @960 MB/s	8 GB
	6 @240 MHz	UMA	bus @960 MB/s	8 GB
SGI/Octane SE	1 @250 MHz	UMA	crossbar @1.6 GB/s	2 GB
	2 @250 MHz	UMA	crossbar @1.6 GB/s	2 GB
SGI/Origin 200	1 @180 MHz	UMA	hub @1.28 GB/s	2 GB
	2 @180 MHz	UMA	hub @1.28 GB/s	2 GB
	4 @180 MHz	NUMA	hub/craylink @1.28 GB/s	4 GB

Table 3: *Systems overview.*

are usually no statements which control the actual order processing entities are scheduled. Consequently, we cannot tell which entity will be executed before an other entity.

Second - **critical sections**. A sequential process does not need to make sure that data which is not completely changed might be already read in another part of the process, because a sequential process only performs one statement at a time. This is different with concurrent processes, where different processing entities might perform different statements at virtually the same time. Therefore, we need to protect those areas, which might cause inconsistent states, because the modifying thread is interrupted by a reading thread. These areas are called critical sections. The protection can be achieved by synchronizing the processing entities at the beginning of these critical sections.

Third - **error handling**. Another difference is error handling. While UNIX calls usually return an useful value, if execution was successful, a potential error code is returned to the general error variable `errno`. This is not possible using threads, because a second thread could overwrite the error code of a previous thread. Therefore, most pthread calls return directly an error code, which can be analyzed or printed onto the screen. Alternatively, the string library function

```
char* strerror(int errno);
```

returns an explicit text string according to the parameter `errno`.

This problem does not really affect message-passing processes, because the processing entities are individual processes with a “private” `errno`. However, most message-passing calls return an error code.

A. Message Passing

In this part of the tutorial, we briefly introduce two message-passing libraries. First we discuss the Message-Passing Interface library - MPI[42, 43], followed by the Parallel Virtual Machine library - PVM[45, 10]. A comparison of these libraries can be found in an article by G. Geist et al.[46]. All these papers can be found on the web, either at netlib, or at the respective homepages of the libraries.

© The Eurographics Association and Blackwell Publishers 1999.

3.2. Message Passing Interface - MPI

MPI 1 (1994) (and later MPI 2 (1997)) is designed as a communication API for multi-processor computers. Usually, the functionality of MPI is implemented using a communication library of the vendor of the machine. Naturally, this vendor library is not portable to other machines. Therefore, MPI adds an abstraction level between the user and this vendor library, in order to guarantee the portability of the program code of the user.

Although MPI does work on heterogeneous workstation clusters, its focus is on high-performance communication on large multi-processors[46]. This results in a rich variety of communication mechanisms. However, the MPI API lacks dynamic resource management, which is necessary for fault tolerant applications.

In the following Sections, we introduce the main components of MPI. Furthermore, we briefly explain some MPI functions, which are used in the PVR system, which is presented in the re-print section of these tutorial notes.

3.2.1. Process Topology and Session Management

To tell the truth, their is no real session management in MPI. Each process of a MPI application is started independent from the others. At some point, the individual processes are exchanging messages, or are synchronized at a barrier. Finally, they shut-down, thus terminating the application. The distribution of the individual processes to the different processing entities (e.g. processors of a multi-processor) is handled by the underlying vendor library.

- **int MPI_Init(int *argc, char ***argv);** - inializes process for MPI.
- **int MPI_Finalize(void);** - releases process from MPI.

Furthermore, the user can specify the process topology within a group (see Section 3.2.2). Besides creating a convenient name space, the specification can be used by the runtime system to optimize communication along the physical interconnection between the nodes[42].

3.2.2. Grouping Mechanisms

A special feature of MPI is support for implementing parallel libraries. Many functions are provided to encapsulate communication within parallel libraries. These functions define a group scope for communication, synchronization, and other related operations of a library. This is done by introducing the concepts of communicators, contexts, and groups.

Communicators are the containers of all communication operations within MPI. They consist of participants (members of groups) and a communication context. Communication is either between members of one group (intra-communication), or between members of different groups (inter-communication). While the first kind of communication provides point-to-point communication and collective communication (such as broadcasts), the second kind only allows point-to-point communication. After initializing MPI for a process, two communicators are predefined. The MPI_COMM_WORLD communicator includes all processes which can communicate with the local process (including the local process). In contrast, the MPI_COMM_SELF communicator only includes the local process.

A group defines the participants of communication or synchronization operations. They define a unique order on their members, thus associating a rank (identifier of member within the group) to each member process. The predefined group MPI_GROUP_EMPTY defines an empty group.

The following functions provide information on a group or its members.

- **int MPI_Comm_size(MPI_Comm com, int* nprocess);** - returns the number of participating processes of communicator com.
- **int MPI_Comm_rank(MPI_Comm com, int* rank);** - returns rank of calling process.

A context defines the “universe” of a communicator. For intra-communicators, they guarantee that point-to-point communication does not interfere with collective communication. For inter-communicators, a context only insulates point-to-point communication, because collective operations are not defined.

3.2.3. Communication

There are two different communication methods. Group members can be either communicate pair-wise, or they can communicate with all members of the group. The first method is called point-to-point communication, the second method is called collective communication. Furthermore, a communication operation can be blocking (it waits until the operation is done) or non-blocking (it does not wait).

© The Eurographics Association and Blackwell Publishers 1999.

Point-To-Point Communication

This class of communication operation defines communication between two processes. These processes can be either members of the same group (intra-communication), or they are members of two different groups (inter-communication). However, we only describe systems with one group (all processes). Therefore, we only use intra-communication.

Usually, a message is attached to a message envelope. This envelope identifies the message and consist of the source or destination rank (process identifier), the message tag, and the communicator.

For blocking communication, the following functions are available:

- **int MPI_Send(void *buf, int n, MPI_Datatype dt, int dest, int tg, MPI_Comm com);** - sends the buffer buf, containing n items of datatype dt to process dest of communicator com. The message has the tag tg.
- **int MPI_Recv(void *buf, int n, MPI_Datatype dt, int source, int tg, MPI_Comm com);** - receives the message tagged with tg from process source of communicator com. The used buffer buf consist of n items of the datatype dt.

These functions are specifying the *standard* blocking communication mode, where MPI decides if the message is buffered. If the message is buffered by MPI, the send call returns without waiting for the receive post. If the message is not buffered, send waits until the message is successfully received by the respective receive call. Besides this *standard mode*, there are *buffered*, *synchronous*, and *ready modes*. More information on these modes can be found in the MPI specification papers[42, 43].

For non-blocking communication MPI_Isend and MPI_Irecv are provided for *intermediate* (I) communication. For *buffered*, *synchronous*, or *ready* communication modes, please refer to the MPI papers. After calling these functions, the buffers are send (or set while receiving). However, they should not be modified until the message is completely received.

- **int MPI_Isend(void *buf, int n, MPI_Datatype dt, int dest, int tg, MPI_Comm com, MPI_Request* req);** - sends the buffer buf, contain n items of datatype dt to process dest of communicator com. The message has the tag tg.
- **int MPI_Irecv(void *buf, int n, MPI_Datatype dt, int source, int tg, MPI_Comm com, MPI_Request* req);** - receives the message tagged with tg from process source of communicator com. The used buffer buf consist of n items of the datatype dt.

In addition to the blocking send and receive, the request handle req is returned. This handle is associated with a communication request object - which is allocated by these calls - and can be used to query this request using MPI_Wait.

- **int MPI_Wait(MPI_Request* req, MPI_Status *stat);** - waits until operation req is completed.

The last call we describe for point-to-point communication is MPI_Probe. This call checks incoming messages if they match the specified message envelope (source rank, message tag, communicator), without actually receiving the message.

- **int MPI_Iprobe(int source, int tg, MPI_Comm com, int* flag, MPI_Status* stat);** - checks incoming messages. The result of the query is stored in flag.

If flag is set true, the specified message is pending. If the specified message is not detected, flag is set to false. The source argument of MPI_Iprobe may be MPI_ANY_SOURCE, thus accepting messages from all processes. Similarly, the message tag can be specified as MPI_ANY_TAG. Depending on the result of MPI_Iprobe, receive buffers can be allocated and source ranks and message tags set.

Collective Communication

Collective Communication is only possible within a group. This implements a communication behavior between all members of the group, not only two members as in point-to-point communication.

We concentrate on two functions:

- **int MPI_Barrier(MPI_Comm com);** - blocks calling process until all members of the group associated with communicator com are blocked at this barrier.
- **int MPI_Bcast(void *buf, int n, MPI_Datatype dt, int root, MPI_Comm com);** - broadcasts message buf of n items of datatype dt from root to all group members of communicator com, including itself.

While the first call synchronizes all processes of the group of communicator com, the second call broadcasts a message from group member root to all processes. A broadcast is received by the members of the group by calling MPI_Bcast with the same parameters as the broadcasting process, including root and com. Please note that collective operations should be executed in the same order in all processes. If this order between sending and receiving broadcasts is changed, a deadlock might occur. Similarly, the order of collective/point-to-point operation should be the same too.

© The Eurographics Association and Blackwell Publishers 1999.

3.3. Parallel Virtual Machine - PVM

While MPI was designed for message-passing on multi-processors, PVM was originally intended for message-passing within a heterogeneous network of workstations. In order to guarantee interoperability between independent computers, the concept of a virtual machine was introduced. While MPI supports only portability (a MPI-based application can be compiled on any system) but not interoperability, PVM processes can even communicate with processes build on completely different machines. Furthermore, processes can be started or terminated dynamically from a master process[§], thus enabling dynamic resource management and fault tolerant applications.

Generally, a parallel application using PVM3 is split into a master process and several slave processes. While the slaves do the actual work of the task, the master distributes data and sub-tasks to the individual slave processes. Finally, the master synchronizes with all slaves at a barrier, which marks the end of the parallel processing.

Before starting the parallel sessions, all designated machines of the cluster need to be announced in a *hostfile*. Furthermore, PVM demons must run on these machines. After running of the parallel sessions, all PVM demons (virtual machines) are shut down.

After this initialization, the master starts its execution by logging on to the running parallel virtual machine (PVM demon). Thereafter, it determines the available hardware configuration (number of available machines (nodes), ...), allocates the name space for the slaves, and starts these slaves by assigning a sub-task (program executable). After checking if all slaves are started properly, data is distributed (and sometimes collected) to the slaves.

At the end of the parallel computation, results are collected from the slaves. After a final synchronization at a common barrier, all slaves and the master log off from the virtual machine.

Next, we briefly introduce some commands for the process control. Furthermore, we introduce commands for distributing and receiving data. For details, please refer to the PVM book[45].

PVM Process Control

- **int pvm_mytid(void);** - logs process on to virtual machine.
- **int pvm_exit(void);** - logs process off from virtual machine.
- **int pvm_config(int* nproc, ...)** - determines number of available nodes (processes), data formats, and additional host information.
- **int pvm_spawn(char *task, ...)** - starts the executable task on a machine of the cluster.
- **int pvm_joingroup(char *groupname);** - calling process joins a group. All members of this group can synchronize at a barrier.
- **int pvm_lvgroup(char *groupname);** - leaving the specified group.
- **int pvm_barrier(char *groupname);** - wait for all group members at this barrier.
- **int pvm_kill(int tid)** - kill slave process with identifier tid.

PVM Communication

- **int pvm_initsend(int opt)** - initializes sending of a message.
- **int pvm_pkint(int* data, int size, ..);** - encodes data of type int[¶] for sending.
- **int pvm_send(int tid, int tag, ..);** - sends data asynchronous (does not wait for an answer) to process tid with specified tag.
- **int pvm_bcast(char* group, int tag);** - broadcasts data asynchronously to all group members.
- **int pvm_mcast(int* tids, int n, int tag);** - broadcasts data synchronously to *n* processes listed in *tids*.
- **int pvm_nrecv(int tid, int tag);** - non-blocking (does not wait if message has not arrived yet) receiving of message.
- **int pvm_recv(int tid, int tag);** - blocking receiving of message *tag*.
- **int pvm_upkint(int* data, int size, ..);** - decodes received data of type int.

There is only one active message buffer at a time. This determines the order of initialization, coding, and sending of the message.

§ Functions to start or terminate processes are integrated in MPI 2.0.

¶ There are commands for other data types, such as byte, double, as well.

© The Eurographics Association and Blackwell Publishers 1999.

B. Thread Programming

As already pointed out in Section 2.4.2, shared-memory can be used for fast communication between the processing entities. Two different approaches are available to provide a higher-level and a lower-level of programming of parallel applications. The lower-level of parallel programming is available using one of the various thread models. Based on a thread model, a higher-level of programming is provided by OpenMP. In the next Sections, we will outline OpenMP and pthreads as basic parallel programming approaches with a coarse grain/high-level and a fine grain/low-level parallel control.

3.4. OpenMP

OpenMP is a vendor initiated specification to enable basic loop-based parallelism in Fortran (77 and up), C, and C++. It basically consists of compiler directives, library routines, and environment variables. More information on the history of OpenMP can be found in the OpenMP FAQ ‖ list [15].

Currently, OpenMP is available for a variety of platforms. While Fortran compiler already support OpenMP compiler directives, the C and C++ support is expected to be available this year. However third-party vendors partially provide the necessary tools. Check the OpenMP website for any details (www.openmp.org).

In this tutorial, we can only give an overview of OpenMP. Information on how to use OpenMP with C or C++ can be found in the OpenMP Tutorial at SuperComputer 1998 conference [30] and in the "OpenMP Application Programming Interface" (API) [14]. Both documents can be found at www.openmp.org.

3.4.1. Execution Model

An OpenMP parallel process starts with a master thread executing the sequential parts of the process (sequential region). Once, the master thread arrives at a parallel construct, it spawns a *team of threads* which process the data associated with the parallel construct in parallel. How the workload is distributed to the different threads of the team and how many threads can be in the team is usually determined by the compiler. However, these numbers can be modified in a controlled way by calling specific library functions. Finally, an implicit barrier at the end of the parallel constructs usually synchronizes all threads, before the master thread continues processing the sequential parts of the program.

3.4.2. Parallel Programming with OpenMP

Several constructs are available for parallel programming. These constructs enable parallel programming, synchronization, a specified concurrent memory view, and some control on how many threads can be used in a team. A subset of these constructs is presented in this Section. For more details, see the OpenMP C/C++ API [14].

Compiler Directives

- `#pragma omp parallel [<clauses>] {...}` specifies a parallel region in which a team of threads is active. The clauses declare specific objects to be shared or to be private.
- `#pragma omp parallel for {...}` constructs enable loop parallel execution of the subsequent for loop. The workload of this loop is distributed to a team of threads based on the for-loop iteration variable. This construct is actually a shortcut of a OpenMP parallel construct containing an OpenMP for construct.
- A sequence of `#pragma omp section {...}` constructs – embraced by a `#pragma omp parallel` construct – specifies parallel sections which are executed by the individual threads of the team.
- `#pragma omp single {...}` specifies a statement or block of statements which is only executed once in the parallel region.
- `#pragma omp master {...}` specifies a statement or block of statements which is only executed by the master thread of the team.
- `#pragma critical [(name)] {...}` specifies a critical section – named with an optional `name` – of a parallel region. The associated statement(s) of the critical sections with the same name are only executed sequentially.
- `#pragma omp atomic <statement>` ensures an atomic assignment in an expression statement.
- `#pragma omp barrier` synchronizes all threads of the team at this barrier. Each thread which arrives at this barrier waits until all threads of the team have arrived at the barrier.

‖ Frequently-Asked-Question

© The Eurographics Association and Blackwell Publishers 1999.

- `#pragma omp flush [(<list>)]` ensures the same memory view to the objects specified in `list`. If this list is empty, all accessible shared objects are flushed.

Other compiler directives or clauses in combination with the compiler directives introduced above can be used to define a specific memory handling, scheduling, and other features.

Library Functions

In addition to the various compiler directives, library functions can be used to affect the number of threads in a team or to determine specific information on the current parallel system. To use these functions, the header file `omp.h` must be included.

- `omp_set_num_threads(int num_threads);` and `int omp_get_num_threads(void);` are available to set or to determine the number of threads in a team in the closest enclosing parallel region.
- `int omp_get_num_procs(void)` determines the number of available processors that can be assigned to the process.
- `int omp_in_parallel(void)` returns a non-zero integer if called in a parallel region. Otherwise, it returns zero.
- `void omp_init_lock(omp_lock_t *lock}, void omp_destroy_lock(omp_lock_t *lock), void omp_set_lock(omp_lock_t *lock), void omp_unset_lock(omp_lock_t *lock),` and `void omp_test_lock(omp_lock_t *lock)` specify the use of a locking mechanism similar to a mutex.

Due to the currently limited availability of C/C++ Compilers with OpenMP support, we cannot really provide a deep treatment of OpenMP at press deadline. However this might change in the final version of the tutorial notes. Check http://www.gris.uni-tuebingen.de/~bartz/sig99course for recent updates.

4. Pthread Programming

There are quite a number of thread models around, like the mthread package[121] of the University of Erlangen-Nürnberg, the dots package[12] of the University of Tübingen, the Compiler-Parallel-Support package of HP/Convex. There are NT-threads, Solaris-threads, and last but not least there is the IEEE POSIX thread standard (pthreads). In this tutorial, we will focus only on pthreads. Furthermore, all the examples are tested on SGI's implementation of pthreads (available for IRIX 6.x and up).

The pthread standard defines an "Application Programming Interface" (API), as specified by POSIX standard 1003.l, or more specific: ISO/IEC 9945-1:1996 (ANSI/IEEE Std 1003.1, 1996 Edition). However, this standard does not define a particular implementation of this standard. Therefore, many definitions are opaque to the user, e.g. thread mapping, data types, etc...

The following text only gives a more or less brief introduction into pthread programming. Advanced features like real-time scheduling or attribute objects are only briefly mentioned or even completely ignored. For a more complete introduction into those topics, please refer to the books [16, 90, 68, 61, 93] listed in Section 6.13.0.3.

4.1. Controlling Pthreads

In this part, we discuss the life cycle of a pthread. The life cycle starts with the creation of the pthread, its work, and the end of its existence.

To start the life of a pthread, we need to execute the `pthread_create` command:

```
int pthread_create( pthread_t *pthread_id,const pthread_attr_t* ptr,
                    void* (*thread_routine) (void *),void  *arg);
```

where

- `pthread_id` is the returned identifier of the created pthread,
- `pthread_attr_t` is the passed attribute structure. If NULL is passed, the default attributes are used.
- `thread_routine` is the name of the function which is called by the created pthread, and
- `arg` is a pointer to the parameter structures for this pthread.

If this function returns error code 0, it was successful. If an error was encountered, the return code specifies the encountered problem.

If a pthreads needs to know its identity, this identity can be established using the call

```
pthread_t pthread_self(void);
```

© The Eurographics Association and Blackwell Publishers 1999.

where the pthread identifier of the current pthread is returned. However, the pthread identifier of another pthread is only known by its caller. If this information is not passed to the particular pthread, this pthread does not know the identifier of the other pthread.

Similar to the last call,

```
int pthread_equal(pthread_t t1,pthread_t t2);
```

determines if two pthread identifiers are referring to the same pthread. If `t1` is equal `t2` a nonzero value will be returned ("True"); if they are not equal, zero will be returned ("False").

The life of a pthread usually terminates with a

```
int pthread_exit(void *ret_val);
```

call. Although the pthread is terminated, the resources used by this pthread are still occupied, until the pthread is detached. Using the command

```
int pthread_detach(pthread_tpthread_id);
```

explicitly detaches a pthread, telling the operating system that it can reclaim the resources as soon as the pthread terminates.

If a pthread A needs to wait for termination of pthread B, the command

```
int pthread_join(pthread_t pthreadB_id, void **ret_val);
```

can be used. As soon as pthread B terminates, it joins pthread A, which is waiting at the `pthread_join` command. If pthread B is returning a result using the pointer `ret_val`, this pointer is accessible via `ret_val` of the `pthread_join` command. If `ret_val` is set to NULL, no return value will be available. `pthread_join` implicitly detaches the specified pthread.

An example for pthread creation can be found as listing 1 in Section 4.4.1.

4.2. Pthread Synchronization

One of the most important topics in thread programming is synchronization. Different resources (e.g. variables, fields, etc.) are shared by different threads. Therefore, the access to these resources needs to be protected. Usually, this protection for MUTual EXclusion is done by a mutex. However, other synchronization mechanisms are known, such as conditions and barriers.

4.2.1. Mutex Synchronization

A mutex protects a critical section in a program. Considering a scenario, where rendering information is stored in a special data structure - e.g. a FIFO queue -, and two threads try to read information from that data structure, obviously, the access to this data structure is a critical section and the access must be limited to one thread at the time. Therefore, the data structure must be protected by a mutex.

Initialization

```
phtread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

int pthread_mutex_init(pthread_mutex_t *mutex,phtread_mutexattr_t *attr);

int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

After memory allocation of the mutex structure, it must be initialized. For static allocation, we can simply assign the preprocessor macro `PTHREAD_MUTEX_INITIALIZER` to the mutex.

In most cases, however, we dynamically allocate a mutex. For these cases, we can use `pthread_mutex_init` to initialize the allocated mutex structure. The second parameter of this command is used to specify a mutex attribute object. This attribute object is not frequently used. Therefore, we pass `NULL`.

If no pthread is locking the mutex, we can destroy it using `pthread_mutex_destroy` before releasing the mutex structure memory. If the mutex is statically allocated and initialized, the explicit destruction of the mutex is not necessary.

© The Eurographics Association and Blackwell Publishers 1999.

Using a Mutex

```
int pthread_mutex_lock(pthread_mutex_t *mutex);

int pthread_mutex_trylock(pthread_mutex_t *mutex);

int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Before entering a critical section in a parallel program, we need to lock the associated mutex using `pthread_mutex_lock`. If the mutex is already locked, the current pthread will be blocked, until the mutex is unlocked by the other pthread. The behavior if a pthread tries to lock a mutex which is already locked by the very same pthread is not defined. Either an error code will be returned, or this pthread will end up in a deadlock.

In case you do not want to wait on an already locked mutex, you can use `pthread_mutex_trylock`. This call returns `EBUSY` in case that the specified mutex is already locked by another pthread. At the end of a critical section you need to unlock the locked mutex using `pthread_mutex_unlock`.

An example for pthread mutexes can be found as listing 2 in Section 4.4.2.

Semaphores

Semaphores is a concept which is more or less a generalization of a mutex. While a mutex only is a binary representation of the state of a resource, a semaphore can be used as a counter ("counting semaphores"). Although the pthread standard does not specify semaphores, the POSIX semaphores can be used.

4.2.2. Condition Synchronization

While mutexes protect a critical section of a program, conditions are used to send messages on the state of shared data. Considering the classic user/producer problem, the producer signals a condition to the users that it has produced data which can be digested by the users.

Dave Butenhof[16] says that

"**Condition variables are for signaling, not for mutual exclusion.**"

Initializing

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

int pthread_cond_init(pthread_cond_t  *cond,pthread_condattr_t *condattr);

int pthread_cond_destroy(pthread_cond_t *cond);
```

Similar to the mutex initialization, static and dynamic allocated condition structures need to be initialized using the respective commands. For our use, we always pass NULL to the `condattr` parameter. Further discussion of the attribute features can be found in Butenhof's book[16].

After use, the condition structures need to be destroyed before releasing the associated memory.

Using conditions

```
int pthread_cond_wait(pthread_cond_t  *cond, pthread_mutex_t *mutex);

int pthread_cond_timedwait(pthread_cond_t  *cond,pthread_mutex_t *mutex,struct timespec *exp);

int pthread_cond_signal(pthread_cond_t  *cond);

int pthread_cond_broadcast(pthread_cond_t  *cond);
```

Note that conditions are always associated with a mutex, where pthreads waiting on the same condition must use the very same mutex. It is not possible to combine two mutexes with one condition, while it is possible to combine two (or more) conditions with one mutex.

© The Eurographics Association and Blackwell Publishers 1999.

Before entering the wait stage using `pthread_cond_wait` or `pthread_cond_timedwait`, the associated mutex must be locked. This mutex is automatically unlocked while waiting on that condition and re-locked before leaving the wait stage. Similar, a signaling pthread needs to lock the mutex before signaling the waiting pthread (see listing 3, Section 4.4.3).

If you consider a waiting pthread A and a signaling pthread B, A will lock the associated mutex mA before entering the wait stage of condition cA. Immediately before blocking pthread A, the system unlocks mutex mA. Later, pthread B locks mutex mA in order to signal pthread A the condition cA. The signal is received by pthread A, which tries to lock mutex mA. After unlocking mutex mA by pthread B, pthread A locks mutex mA and returns from the `pthread_cond_wait` to the user's code. Thereafter, the user unlocks mutex mA.

Another important note is that pthreads might wake up without getting the proper signal for various reasons. Therefore, we need to use a shared predicate which is set if there is a proper wake-up signal. If this predicate is not set, the waiting pthread will wait again until it receives the proper signal.

In some situations it is useful to limit the waiting time by a timeout. In these cases, the maximum waiting time can be specified by the `exp` parameter of the `pthread_cond_timedwait` command. It will return with the value `ETIMEDOUT` if the pthread does not receive the expected signal within the timeout limit.

The pthread mechanism for waking-up pthreads waiting at a condition is `pthread_cond_signal` and `pthread_cond_broadcast`. While the first one only wakes up the first pthread waiting at that condition, the latter wakes up all pthreads waiting at that condition.

Please note, if no pthread is waiting at a condition, this condition will simply die away. Furthermore, if a pthread starts waiting at this condition shortly after the wake-up signal/broadcast, it remains waiting for a signal which possibly never arrives.

An example for pthread conditions can be found as listing 3 in Section 4.4.3.

4.2.3. Barrier Synchronization

The last presented synchronization concept is the barrier synchronization. Unfortunately, this concept is not part of the current pthread standard (1996), but it is on the draft list for the next version.

Generally, a barrier synchronization stops threads at this barrier, until the specified number of threads arrive. Thereafter, all threads proceed. There are different suggestions how to implement barriers in the current pthread standard. We will present two examples of an implementation. The first one implements a barrier synchronization at the end of the life cycle of the threads by joining them in a cascade (see listing 2 in Section 4.4.2). However, this method is not suited for a barrier synchronization which is not at the end of the life cycle of the pthreads, but in the middle of the working program. In addition, it has some structural limitations, because each pthreads in the cascade needs to know its successor's pthread identifier.

The second example is from Dave Butenhof book on POSIX threads[16]. In this example, every pthread which waits at a barrier is decrementing the waiting pthread counter and checks if more pthreads are expected to wait at this barrier. If no further pthread is expected to wait, it broadcasts the other waiting pthreads that the appropriate number of pthreads arrived at the barrier. If the number of waiting pthreads is not reached, this pthreads starts waiting for the broadcast. This implementation of a barrier can be found as listing 4, Section 4.4.4.

4.3. Additional Topics

4.3.1. Concurrent Memory Visibility

As mentioned earlier, programming concurrent (parallel) systems is quite different from programming sequential systems. This is especially true for the view of the memory we are using within our parallel program.

Modern processors are buffering data into caches of different sizes and different levels. If more than one processor is working for one program, different caches are storing information. Therefore, the information visible by one processors (in its cache) might be not the same as visible to another processor (in its cache or the main memory). This problem becomes even worse if NUMA memory architectures are used, because checking for changes in different caches and different memory hierarchies is much more difficult.

The pthread standard defines situations when the memory view of the different threads (possibly running on different processors) is equal, providing that the memory has not changed after these commands.

- After starting pthreads (`pthread_create`), the started pthreads have the same memory view as their parent.

© The Eurographics Association and Blackwell Publishers 1999.

- After explicitly (`pthread_mutex_unlock`) or implicitly (conditions) unlocking mutexes, the pthreads which are blocked at this mutex have the same memory view as the unlocking pthread.
- Furthermore, the memory view of terminated pthreads (canceled pthreads, exited pthreads, or simply returning from their thread function) is the same as of the pthread which joins the terminating pthreads.
- Finally, each pthread which is waked-up by a signaling or broadcasting pthread has the same memory view as the signaling or broadcasting pthread.

Apart from these situations, the same memory view can not be guaranteed. Although you might never encounter this problems on a particular system (it might be cache-coherent), you can never be sure.

4.3.2. Cancelation

```
int pthread_cancel(pthread_t pthread_id);

int pthread_setcancelstate(int state, int* ostate);

int pthread_setcanceltype(int type, int* otype);

void pthread_testcancel(void);
```

Usually, a thread is executing a particular part of the program until the task is done and the thread is either returning to its parent thread (main thread), or exits. However, there are situations where the task of the thread becomes dispensable. In those cases, it is useful to cancel this thread.

In general, we need the pthread identifier of the pthread to be canceled. Without this identifier, we cannot cancel the pthread. To cancel a pthread, we call `pthread_cancel(pthread_id);`.

There are three different cancellation modes the user can choose from. First, there is the DISABLED mode, where the cancel state is set to PTHREAD_CANCEL_DISABLE (the value of the cancel type will be ignored). In this mode no cancellation is possible. It becomes meaningful to prevent data corruption, while the pthread is changing data. In this cases, the pthread disables cancellation until it has finished the modification. Thereafter, it enables cancellation again. Cancel requests issued while the cancellation is disabled, are queued until the cancellation state is enabled again.

If the cancellation state is set to PTHREAD_CANCEL_ENABLE, we can choose from two cancellation types; PTHREAD_CANCEL_DEFERRED (the default) or PTHREAD_CANCEL_ASYNCHRONOUS. The second type indicates that the respective pthread should be canceled at any time from now. This might cause data corruption, deadlocks - pthreads which are locked at a mutex locked by the canceled pthread -, and so forth. This is really an emergency kind of cancellation. Better is the first cancellation type, which asks the pthread to stop at the next cancellation point. At implicit cancellation points like `pthread_cond_wait`, `pthread_cond_timedwait`, or `pthread_join`, the pthread cancel immediately after executing these commands. However, an explicit cancellation point can be set using `pthread_testcancel`. If a cancel request is pending, the pthread returns the value PTHREAD_CANCELED to a pthread which waits to join this pthread. If no cancel request is pending, the `pthread_testcancel` command immediately returns. Besides these implicit or explicit cancellation points, there are library calls or system calls which are implicit cancellation points. Generally, these calls can introduce some blocking behavior and are therefore good candidates for cancellation. Please refer to one of the pthread books for a list of these calls.

Please note, enabling cancellation is not a cancellation point. Therefore, you need to explicitly set a cancellation point after enabling cancellation.

Another feature of cancellation is the specification of an cleaning-up handler for the pthread to be canceled. This cleaning-up handler can close files, release memory, repair data modifications, and so forth. Please refer to Butenhofs book[16] for more information on cleaning-up canceled pthreads.

4.3.3. Hints

In this Section, we provide some tips and hints on common problems and usage of pthreads on some systems.

Debugging

- **Thread races.** Never count on an execution order of pthreads. Generally, we can not assume a certain executing order of pthreads. The standard does not completely control the actual scheduling of the physical system. Furthermore, after creation of a pthread, you cannot count that this pthread will start before another pthread created after the first pthread.

© The Eurographics Association and Blackwell Publishers 1999.

- **Avoid potential deadlock situations.** Well, this sounds obvious. However, there are many unavoidable situations which are potential deadlock situations. If you use mutex hierarchies (lock one mutex after successfully locking a first mutex), you need to consider a back-off strategy in case that the second mutex locking will block the pthread, which keeps the first mutex.
- **Priority inversion.** If you use real-time priority scheduling (see Section 4.3.4), your scheduling strategy (FIFO) might schedule a pthread to run which tries to lock a mutex, locked by a pthread preempted by the first pthread. Mutual exclusion and scheduling performing a kind of contradictory execution which can cause a deadlock.
- **Sharing stacks.** Pthread attributes (Section 4.3.4) enable the user to share stack memory. If the size of this stack is too small for these pthreads, you will encounter some strange effects.

Performance

- **Mutexes** are not for free. You should always carefully decide if you use a "big mutex" protecting one big piece of code, or a number of mutexes protecting more fine granularly critical sections.
- **Concurrency Level** Pthread implementations on UNIX98 conform systems (i.e., IRIX 6.5) provide `int pthread_setconcurrency(int npthreads);` to advice the kernel how many kernel execution entities (kernel threads) should be allocated should be allocated for the pthread parallel process. However, some UNIX systems do not require this call, because user mode and kernel mode scheduler are cooporating close enough (i.e., Digital UNIX). `int pthread_getconcurrency(void)` returns the current level of concurreny.
- **Pthreads on IRIX.** The current implementation of pthreads on SGI workstations maps the pthreads on sproc light-weight processes of the operating systems. Furthermore, the system/pthread library decides if it starts an additional sproc for an additional pthread.
  In my experience, this does not work very well. Therefore, you can tell the operating system/pthread library that your pthreads are compute-bound, by setting the environment variable `PT_ITC` (setenv PT_ITC). This usually results in starting enough sprocs for all processors. More recent implementations of pthreads on IRIX (6.5) use `pthread_setconcurrency(<npthreads>);` to advice the system how many kernel execution entities (kernel threads) should be allocated (see above).
- **Solaris threads.** On Solaris 2.5, threads are not time-sliced. Therefore, we need to set the concurrency level to the number of started threads, in order to obtain a concurrent program execution. The respective command is `thr_setconcurrency(<nthreads>);` (see pthread concurrency level above).

I found a nice quote in Dave Butenhof's book for those who are getting frustrated while debugging a concurrent program:

"**Wisdom comes from experience, and experience comes from lack of wisdom.**"

4.3.4. Further topics

In this tutorial, we do not provide material on all pthread topics. In my experience, I have never needed features like one-time initialization, real-time scheduling, thread-specific data, thread attributes, and so forth. However, there are situations where you might need this features. A discussion of these additional features can be found in Butenhofs book[16].

4.4. Example Code

This part contains example code for pthread programming. Please note that we denote the thread which starts all other threads as main thread. Naturally, this thread is considered as a pthread too. Furthermore, we use the term pthread for all threads started by the main thread using the command pthread_create.

4.4.1. Initializing Pthreads

The pthread program listed listing 1, starts five pthreads and passes a couple of values to the pthreads. Finally, the main pthread collects the pthread started first.

14, number of pthreads started (including the main thread).

16-19, type definition of parameter record passed to the started pthreads.

21-38, thread function which is executed by the started pthreads. The own pthread identifier is look up (27). After a short loop (28), the thread function tests if the current pthread is the main thread (30). All pthreads created by the main thread are terminated and return their number identifier (35).

48-63, PTHREADS - 1 pthreads are started and a parameter record containing two parameters are passed to the pthreads; the current loop value (like a number identifier for the pthreads) and the pthread identifier of the main thread.

© The Eurographics Association and Blackwell Publishers 1999.

65-72, the four pthreads (PTHREADS - 2) started last are detached from the main thread. After their termination, their resources are released to the operating system. If the main thread terminates before these pthreads, they terminate immediately (without completing their thread function).

74, the main thread executes the thread function (like the other pthreads).

75-76, the main thread joins with the first pthread started (and implicitly detaches this pthread, 75). The return value of the started pthread is returned in resp (75) and casted (76).

4.4.2. Mutex Example

The pthread program listed listing 2, starts five pthreads and passes a couple of values to the pthreads. Each started pthread tries to lock the mutex allocated and initialized by the main thread.

20-25, type definition of parameter passed to the pthreads.

27-63, thread function executed by the started pthreads. Each started pthread locks the shared mutex 100 times (32-557). After locking (36), it performs a loop (43) and unlocks the mutex (44-49). Finally, each started pthread terminates using pthread_exit (60). In contrast to the mutex locking of the main pthread (105), the pthreads are using trylock and count the unsuccessful tries (38-41).

72-83, a common mutex is allocated (72-76) and initialized (78-83).

85-103, five pthreads are started and the respective data is passed to them (96).

105-117, the mutex is locked by the pthread (105-110). Please note that this mutex is not necessarily locked by the main thread first. The pthread standard does not specify a scheduling/execution order (see 4.3.3 thread races). After successful locking, the main thread executes a loop (111) and unlocks the mutex (112-117).

119-129, the main thread joins with all started pthread in the order they were created. If a pthread created later terminates before an earlier pthread, it is not joined until all pthreads created earlier were joined. This is a cascade implementation of a barrier. The main thread does not proceed until all started pthreads are joined. Please note that after the presented kind of barrier synchronization, no pthreads are running anymore.

131-137, the mutex is released.

4.4.3. Condition Example

The pthread program listed listing 3, starts two pthreads and passes a couple of values to the pthreads. Finally, the main pthread collects the started pthreads. The pthreads are alternating processing a shared variable using conditions to signal the state of the variable to the pthread.

18-24, type definition of shared data passed to the pthreads.

26, shared data definition.

39-72, allocates and initializes mutex (43-51) and conditions (53-70).

74-113, producer pthread. After waiting for two seconds (82) in order to make sure that consumer pthread waits at the condition, the producer locks the mutex (87-90), manipulates the shared data, setting the predicate to 0, marking that it has been processed by the producer (91-92), and signals to the consumer that the shared data is ready to process (93-96). Thereafter, the producer waits until the data is consumed by the consumer pthread (98-103). Each time the producer is waked up by a signal (99), it checks if the predicate is correct. If not, it continues waiting (98,103). This is to prevent wrong wake-ups of the waiting pthread. After waiting of the pthread at this condition, the mutex is unlocked (107-110). Please note that while waiting for the signal (99), the mutex is unlocked by the system and re-locked before returning to the user code.

115-151 consumer pthread. Similar to the producer pthread, the consumer locks the mutex (124-127) and waits at the condition for the proper signal (128-133). If a wrong signal is received which waked-up the consumer, the predicate is not set properly. Therefore, we continue waiting (128, 133). After successful receiving the proper signal, the consumer consumes the shared data (135), sets the predicate (136), and signals the consumption to the producer pthread (139-142). Thereafter, it unlocks the mutex (144-147) and continues waiting (124) for new data. Please note that the mutex is locked while producer/consumer are manipulating the shared data. The mutex is released by the pthreads, while they are waiting at the condition. If the mutex is unlocked while manipulating the data, a deadlock is usually the result, because the later signal might be received by the other pthread. Due to scheduling, it is possible that the pthread just waked-up by a wrong signal, therefore misses the correct signal.

© The Eurographics Association and Blackwell Publishers 1999.

159-166, producer and consumer are started. The producer/consumer cycles is performed 200 times. (Actually, only pointers to data structures should be passed. An integer does not always fit into the memory space of a pointer.

170-177, both pthreads are collected by the main thread.

179-193, resources are released.

4.4.4. Barrier Example

This Section describes the barrier example of the book by Dave Butenhof**.

Three functions are defined; barrier_init and barrier_destroy define the initializing and destructor functions of the barrier. The function barrier_wait defines the entrance to the barrier. The pthreads at this barrier wait until a specified number of pthreads has arrived. This number is specified in barrier_init.

1-42 barrier header file barrier.h.

43-186 barrier code file barrier.c.

19-26 type definition of barrier.

72-88 barrier_init. This function initializes the barrier `barrier` (72). The number of pthreads which need to wait at this barrier is specified with `count` (72,76). In 77, cycle is initialized. This variable is used to filter wrong wake-up signals. Finally, the barrier is made valid in 86.

93-125 barrier_destroy. This function removes the barrier `barrier` (93). After checking if the barrier is valid (97), the barrier access mutex is locked (100). If any pthreads are still waiting at this barrier (108), this function is aborted (110). If no pthreads are waiting, the barrier is invalidized (113), the access mutex unlocked and released, and the condition is removed (122,123).

132-186 is the actual barrier function. The pthreads which enter this function are blocked (169) until the it receives a signal and the cycle has changed, since the pthread has entered this function. If the pthread which has just entered the barrier_wait function is the pthread all the other pthreads are waiting for, it changes the cycle (146), resets the counter (147), and broadcasts to all waiting pthreads that it has arrived (148).

** D. Butenhof, PROGRAMMING with POSIX THREADS, (page 245). (C) 1997 Addison Wesley Longman Inc., Reprinted by permission of Addison Wesley Longman.

© The Eurographics Association and Blackwell Publishers 1999.

Listing 1 - Initializing Pthreads

```
1  /*
2   * create.c
3   * starting and terminating pthreads
4   *
5   */
6
7  #include <stdio.h>
8  #include <stdlib.h>
9  #include <limits.h>
10 #include <string.h>
11
12 #include <pthread.h>
13
14 #define NO_PTHREADS 6
15
16 typedef struct {
17   pthread_t main;
18   int       pthread_no;
19 } ident_t;
20
21 void* thread_function(void* arg)
22 {
23   int       i;
24   ident_t*  info = (ident_t*) arg;
25   pthread_t self;
26
27   self = pthread_self();
28   for (i=0; i<INT_MAX/100; i++);
29
30   if (pthread_equal(self,info->main)) {
31     fprintf(stderr,"Current pthread is main thread.\n");
32   } else {
33     fprintf(stderr,"Current pthread is thread #%d.\n",
34                  info->pthread_no);
35     pthread_exit((void*) &(info->pthread_no));
36   }
37   return NULL;
38 }
39
40 int main(void)
41 {
42   int       i,rc;
43   int      *res;
44   void     *resp;
45   pthread_t ids[NO_PTHREADS+1];
46   ident_t   infos[NO_PTHREADS+1];
47
48   ids[0]= pthread_self();
49   infos[0].pthread_no = 0;
50   infos[0].main = pthread_self();
51   for (i=1; i<=NO_PTHREADS; i++) {
52     infos[i].pthread_no = i;
53     infos[i].main = pthread_self();
54     rc = pthread_create(&ids[i], NULL, thread_function,
55                                               (void*) &(infos[i]));
56     if (rc) {
57       fprintf(stderr,"ERROR - while creating pthread %d: %s\n",
58                    infos[i].pthread_no, strerror(rc));
59       exit(-1);
60     }
61     fprintf(stderr,"Main: Thread %d started.\n",
62                  infos[i].pthread_no);
63   }
64
65   for (i=2; i<=NO_PTHREADS; i++) {
66     rc = pthread_detach(ids[i]);
67     if (rc) {
68       fprintf(stderr,"ERROR - while detaching pthread %d: %s\n",
69                    infos[i].pthread_no, strerror(rc));
70       exit(-1);
71     }
72   }
73
74   thread_function((void*) &(infos[0]));
75   rc = pthread_join(ids[1], &resp);
76   res = (int*) resp;
77   if (rc) {
78     fprintf(stderr,"ERROR - while joining pthread %d: %s\n",
79                  infos[i].pthread_no, strerror(rc));
80     exit(-1);
81   }
82   fprintf(stderr,"Joined pthread result is %d\n",
83                  (int) *res);
84 }
```

© The Eurographics Association and Blackwell Publishers 1999.

Listing 2 - Mutex Example

```
1   /*
2    * mutex.c
3    * pthread mutex handling
4    *
5    */
6
7   #include <stdio.h>
8   #include <stdlib.h>
9   #include <errno.h>
10  #include <string.h>
11
12  #include <pthread.h>
13
14  #define EOK          0
15  #define TRUE         1
16  #define FALSE        0
17  #define NO_PTHREADS  6
18  #define LOCK_TIMES   100000
19
20  typedef struct {
21    int pthread_no;
22    pthread_t main;
23    int mutex_tries;
24    pthread_mutex_t* mutex;
25  } param_t;
26
27  void* thread_function(void* arg)
28  {
29    int       i,j,rc,tries,read;
30    param_t*  info = (param_t*) arg;
31
32    for (i=0; i<100; i++) {
33      tries = 0;
34      read  = TRUE;
35      while (read) {
36        rc = pthread_mutex_trylock(info->mutex);
37        switch (rc) {
38        case EBUSY:
39          tries++;
40          info->mutex_tries++;
41          break;
42        case EOK:
43          for (j=0; j<LOCK_TIMES; j++);
44          rc = pthread_mutex_unlock(info->mutex);
45          if (rc) {
46             fprintf(stderr,"ERROR - while unlocking mutex: %s\n",
47                     strerror(rc));
48             exit(-1);
49          }
50          read = FALSE;
51          break;
52        default:
53          fprintf(stderr,"ERROR - while trying to lock mutex: %s\n",
54                  strerror(rc));
55        }
56      }
57    }
58
59    if (!pthread_equal(pthread_self(),info->main)) {
60      pthread_exit(NULL);
61    }
62    return NULL;
63  }
64
65  int main(void)
66  {
67    int       i,rc;
68    pthread_t ids[NO_PTHREADS+1];
69    pthread_mutex_t *mutex;
70    param_t   infos[NO_PTHREADS+1];
71
72    mutex = (pthread_mutex_t*) calloc(1,sizeof(pthread_mutex_t));
73    if (!mutex) {
74      fprintf(stderr,"ERROR - while allocating mutex.\n");
75      exit(-1);
76    }
77
78    rc = pthread_mutex_init(mutex,NULL);
79    if (rc) {
80      fprintf(stderr,"ERROR - while init' mutex: %s\n",
81                   strerror(rc));
82      exit(-1);
83    }
84
85    rc = pthread_mutex_lock(mutex);
86    if (rc) {
87      fprintf(stderr,"ERROR - while locking mutex: %s\n",
88                   strerror(rc));
89      exit(-1);
90    }
91    ids[0]= pthread_self();
92    infos[0].pthread_no = 0;
93    infos[0].mutex_tries = 0;
94    infos[0].mutex = mutex;
95    infos[0].main = pthread_self();
96    for (i=1; i<=NO_PTHREADS; i++) {
97      infos[i].pthread_no = i;
98      infos[i].mutex_tries = 0;
99      infos[i].mutex = mutex;
100      infos[i].main = pthread_self();
101      rc = pthread_create(&ids[i], NULL, thread_function,
102                          (void*) &(infos[i]));
103      if (rc) {
104        fprintf(stderr,"ERROR - while creating pthread %d: %s\n",
105                    infos[i].pthread_no, strerror(rc));
106        exit(-1);
107      }
108      fprintf(stderr,"Thread %d started.\n",i);
109    }
110
111    for (i=0; i<100000; i++);
112    rc = pthread_mutex_unlock(mutex);
113    if (rc) {
114      fprintf(stderr,"ERROR - while unlocking mutex: %s\n",
115                  strerror(rc));
116      exit(-1);
117    }
118
119    /* Simulating a barrier */
120    for (i=1; i<=NO_PTHREADS; i++) {
121      rc = pthread_join(ids[i], NULL);
122      if (rc) {
123        fprintf(stderr,"ERROR - while joining pthread %d: %s\n",
124                    infos[i].pthread_no, strerror(rc));
125        exit(-1);
126      }
127      fprintf(stderr,"On average, pthread %d waited %d times\n",
128                  infos[i].pthread_no,infos[i].mutex_tries/LOCK_TIMES);
129    }
130
131    rc = pthread_mutex_destroy(mutex);
132    if (rc) {
133      fprintf(stderr,"ERROR - while destroying mutex: %s\n",
134                  strerror(rc));
135      exit(-1);
136    }
137    free(mutex);
138  }
```

© The Eurographics Association and Blackwell Publishers 1999.

Listing 3 - Condition Example

```
1   /*
2    * cond.c
3    * pthread condition handling
4    *
5    */
6
7   #include <stdio.h>
8   #include <stdlib.h>
9   #include <unistd.h>
10  #include <string.h>
11  #include <pthread.h>
12
13  #define NOTHING                          -1
14  #define CREATED                          0
15  #define MODIFIED           1
16  #define RES                              (void*) 0;
17
18  typedef struct {
19    int val;
20    pthread_mutex_t *mutex;
21    pthread_cond_t  *created;
22    pthread_cond_t  *consumed;
23    int              pred; /* shared predicate */
24  } buffer_t;
25
26  static buffer_t buffer;
27
28  void ferr(char *text, int rc)
29  {
30    fprintf(stderr,"%s: %s\n",text, strerror(rc));
31    exit(-1);
32  }
33
34  void message(char *text)
35  {
36    fprintf(stderr,"%s\n",text);
37  }
38
39  void init(void)
40  {
41    int rc;
42
43    buffer.mutex =
44      (pthread_mutex_t*) calloc(1,sizeof(pthread_mutex_t));
45    if (!buffer.mutex) {
46      ferr("ERROR - while allocating mutex",0);
47    }
48    rc = pthread_mutex_init(buffer.mutex, NULL);
49    if (rc) {
50      ferr("ERROR - while init' mutex",rc);
51    }
52
53    buffer.created =
54      (pthread_cond_t*) calloc(1,sizeof(pthread_cond_t));
55    if (!buffer.created) {
56      ferr("ERROR - while allocating condition created",0);
57    }
58    buffer.consumed =
59      (pthread_cond_t*) calloc(1,sizeof(pthread_cond_t));
60    if (!buffer.consumed) {
61      ferr("ERROR - while allocating condition consumed",0);
62    }
63    rc = pthread_cond_init(buffer.created, NULL);
64    if (rc) {
65      ferr("ERROR - while init' condition created",rc);
66    }
67    rc = pthread_cond_init(buffer.consumed, NULL);
68    if (rc) {
69      ferr("ERROR - while init' condition conumed",rc);
70    }
71    message("Classic producer/consumer problem is started ..");
72  }
73
74  void *producer(void *times)
75  {
76    int i,t,rc;
77
78    t = (int) times;
79    i = 1;
80
81    /* To slow down initial signal */
82    sleep(2);
83
84    message("Producer is started..");
85
86    while (i<=t) {
87      rc = pthread_mutex_lock(buffer.mutex);
88      if (rc) {
89         ferr("ERROR - while locking mutex",rc);
90      }
91      buffer.val =  i+1;
92      buffer.pred = 0;
93      rc = pthread_cond_signal(buffer.created);
94      if (rc) {
95         ferr("ERROR - while signaling created",rc);
96      }
97
98      do {
99        rc = pthread_cond_wait(buffer.consumed, buffer.mutex);
100        if (rc) {
101           ferr("ERROR - while waiting on consumed",rc);
102        }
103      } while (buffer.pred != 1);
104      fprintf(stderr,"  Modified value No. %d=%d\n",i,buffer.val);
105      i++;
106
107      rc = pthread_mutex_unlock(buffer.mutex);
108      if (rc) {
109        ferr("ERROR - while unlocking mutex",rc);
110      }
111    }
112    return NULL;
113  }
114
115  void *consumer(void* times)
116  {
117    int i,t,rc;
118
119    t = (int) times;
120    i = 1;
121    message("Consumer is started..");
122
```

© The Eurographics Association and Blackwell Publishers 1999.

```
123    while (i<=t) {
124      rc = pthread_mutex_lock(buffer.mutex);
125      if (rc) {
126        ferr("ERROR - while locking mutex",rc);
127      }
128      do {
129        rc = pthread_cond_wait(buffer.created,buffer.mutex);
130        if (rc) {
131           ferr("ERROR - while waiting on empty",rc);
132        }
133      } while (buffer.pred != 0);
134
135      buffer.val *=11;
136      buffer.pred = 1;
137      i++;
138
139      rc = pthread_cond_signal(buffer.consumed);
140      if (rc) {
141        ferr("ERROR - while  signaling consumed",rc);
142      }
143
144      rc = pthread_mutex_unlock(buffer.mutex);
145      if (rc) {
146        ferr("ERROR - while unlocking mutex",rc);
147      }
148    }
149
150    return NULL;
151  }
152
153  void main(void)
154  {
155    pthread_t pthreadA, pthreadB;
156    int     rc;
157
158    init();
159    rc = pthread_create(&pthreadA, NULL, producer, (void*) 200);
160    if (rc) {
161      ferr("ERROR - while creating pthread ",rc);
162    }
163    rc = pthread_create(&pthreadB, NULL, consumer, (void*) 200);
164    if (rc) {
165      ferr("ERROR - while  creating pthread",rc);
166    }
167
168    message("Main is waiting for pthreads ...");
169
170    rc = pthread_join(pthreadA,NULL);
171    if (rc) {
172      ferr("ERROR - while joining pthread",rc);
173    }
174    rc = pthread_join(pthreadB,NULL);
175    if (rc) {
176      ferr("ERROR - while joining pthread",rc);
177    }
178
179    rc = pthread_mutex_destroy(buffer.mutex);
180    if (rc) {
181      ferr("ERROR - while destorying mutex",rc);
182    }
183     rc = pthread_cond_destroy(buffer.created);
184    if (rc) {
185      ferr("ERROR - while destorying condition created",rc);
186    }
187    rc = pthread_cond_destroy(buffer.consumed);
188    if (rc) {
189      ferr("ERROR - while destorying condition consumed",rc);
190    }
191    free(buffer.mutex);
192    free(buffer.created);
193    free(buffer.consumed);
194
195    message("Done.");
196  }
197
```

© The Eurographics Association and Blackwell Publishers 1999.

Listing 4 - Barrier Example

D. Butenhof, PROGRAMMING WITH POSIX THREADS, (page 245). (c) 1997 Addison-Wesley-Longman Inc., Reprinted by permission of Addison-Wesley-Longman.

```
1   /*
2    * barrier.h
3    *
4    * This header file describes the "barrier" synchronization
5    * construct. The type barrier_t describes the full state of the
6    * barrier including the POSIX 1003.1c synchronization objects
7    * necessary.
8    *
9    * A barrier causes threads to wait until a set of threads has
10   * all "reached" the barrier. The number of threads required is
11   * set when the barrier is initialized, and cannot be changed
12   * except by reinitializing.
13   */
14  #include <pthread.h>
15
16  /*
17   * Structure describing a barrier.
18   */
19  typedef struct barrier_tag {
20      pthread_mutex_t     mutex;          /* Control access to barrier */
21      pthread_cond_t      cv;             /* wait for barrier */
22      int                 valid;          /* set when valid */
23      int                 threshold;      /* number of threads required */
24      int                 counter;        /* current number of threads */
25      int                 cycle;          /* alternate wait cycles (0 or 1)
*/
26  } barrier_t;
27
28  #define BARRIER_VALID   0xdbcafe
29
30  /*
31   * Support static initialization of barriers
32   */
33  #define BARRIER_INITIALIZER(cnt) \
34      {PTHREAD_MUTEX_INITIALIZER, PTHREAD_COND_INITIALIZER, \
35      BARRIER_VALID, cnt, cnt, 0}
36
37  /*
38   * Define barrier functions
39   */
40  extern int barrier_init (barrier_t *barrier, int count);
41  extern int barrier_destroy (barrier_t *barrier);
42  extern int barrier_wait (barrier_t *barrier);
43  /*
44   * barrier.c
45   *
46   * This file implements the "barrier" synchronization construct.
47   *
48   * A barrier causes threads to wait until a set of threads has
49   * all "reached" the barrier. The number of threads required is
50   * set when the barrier is initialized, and cannot be changed
51   * except by reinitializing.
52   *
53   * The barrier_init() and barrier_destroy() functions,
54   * respectively, allow you to initialize and destroy the
55   * barrier.
56   *
57   * The barrier_wait() function allows a thread to wait for a
58   * barrier to be completed. One thread (the one that happens to
59   * arrive last) will return from barrier_wait() with the status
60   * -1 on success -- others will return with 0. The special
61   * status makes it easy for the calling code to cause one thread
62   * to do something in a serial region before entering another
63   * parallel section of code.
64   */
65  #include <pthread.h>
66  #include "errors.h"
67  #include "barrier.h"
68
69  /*
70   * Initialize a barrier for use.
71   */
72  int barrier_init (barrier_t *barrier, int count)
73  {
74      int status;
75
76      barrier->threshold = barrier->counter = count;
77      barrier->cycle = 0;
78      status = pthread_mutex_init (&barrier->mutex, NULL);
79      if (status != 0)
80          return status;
81      status = pthread_cond_init (&barrier->cv, NULL);
82      if (status != 0) {
83          pthread_mutex_destroy (&barrier->mutex);
84          return status;
85      }
86      barrier->valid = BARRIER_VALID;
87      return 0;
88  }
89
90  /*
91   * Destroy a barrier when done using it.
92   */
93  int barrier_destroy (barrier_t *barrier)
94  {
95      int status, status2;
96
97      if (barrier->valid != BARRIER_VALID)
98          return EINVAL;
99
100      status = pthread_mutex_lock (&barrier->mutex);
101      if (status != 0)
102          return status;
103
104      /*
105       * Check whether any threads are known to be waiting; report
106       * "BUSY" if so.
107       */
108      if (barrier->counter != barrier->threshold) {
109          pthread_mutex_unlock (&barrier->mutex);
110          return EBUSY;
111      }
112
113      barrier->valid = 0;
114      status = pthread_mutex_unlock (&barrier->mutex);
115      if (status != 0)
116          return status;
117
118      /*
119       * If unable to destroy either 1003.1c synchronization
120       * object, return the error status.
121       */
122      status = pthread_mutex_destroy (&barrier->mutex);
123      status2 = pthread_cond_destroy (&barrier->cv);
124      return (status == 0 ? status : status2);
125  }
126
```

© The Eurographics Association and Blackwell Publishers 1999.

D. Butenhof, PROGRAMMING WITH POSIX THREADS, (page 245). (c) 1997 Addison-Wesley-Longman Inc., Reprinted by permission of Addison-Wesley-Longman.

```
127  /*
128   * Wait for all members of a barrier to reach the barrier. When
129   * the count (of remaining members) reaches 0, broadcast to wake
130   * all threads waiting.
131   */
132  int barrier_wait (barrier_t *barrier)
133  {
134      int status, cancel, tmp, cycle;
135
136      if (barrier->valid != BARRIER_VALID)
137          return EINVAL;
138
139      status = pthread_mutex_lock (&barrier->mutex);
140      if (status != 0)
141          return status;
142
143      cycle = barrier->cycle;   /* Remember which cycle we're on */
144
145      if (--barrier->counter == 0) {
146          barrier->cycle = !barrier->cycle;
147          barrier->counter = barrier->threshold;
148          status = pthread_cond_broadcast (&barrier->cv);
149          /*
150           * The last thread into the barrier will return status
151           * -1 rather than 0, so that it can be used to perform
152           * some special serial code following the barrier.
153           */
154          if (status == 0)
155              status = -1;
156      } else {
157          /*
158           * Wait with cancellation disabled, because barrier_wait
159           * should not be a cancellation point.
160           */
161          pthread_setcancelstate (PTHREAD_CANCEL_DISABLE, &cancel);
162
163          /*
164           * Wait until the barrier's cycle changes, which means
165           * that it has been broadcast, and we don't want to wait
166           * anymore.
167           */
168          while (cycle == barrier->cycle) {
169              status = pthread_cond_wait (
170                      &barrier->cv, &barrier->mutex);
171              if (status != 0) break;
172          }
173
174          pthread_setcancelstate (cancel, &tmp);
175      }
176      /*
177       * Ignore an error in unlocking. It shouldn't happen, and
178       * reporting it here would be misleading -- the barrier wait
179       * completed, after all, whereas returning, for example,
180       * EINVAL would imply the wait had failed. The next attempt
181       * to use the barrier *will* return an error, or hang, due
182       * to whatever happened to the mutex.
183       */
184      pthread_mutex_unlock (&barrier->mutex);
185      return status;          /* error, -1 for waker, or 0 */
186  }
```

4.5. OpenMP versus Pthreads

OpenMP and pthreads provide tools for the parallel programming based on shared memory and the thread model. While OpenMP provides a somewhat higher-level of programming and a coarser grain of parallel control, it is easier to use than pthreads. On other hand, pthreads provide more flexibility and a better parallel control for fine grain parallel problems. Moreover, pthreads are not necessarily focusing on parallel computer systems; pthreads are already worthwhile to consider for multi-threaded application on single-processor systems. Finally, OpenMP is frequently implemented on top of a thread implementation of the specific systems. Therefore, its performance is depending on used thread implementation.

PART TWO

Rendering

5. Parallel Polygonal Rendering

5.1. Introduction

Many datasets in design, modeling, and scientific visualization are built from polygons and often from simple triangles. Frequently these datasets are very big (several millions to several tens of millions of triangles) as they describe a polygonal approximation to an underlying true surface. The size of these datasets often exceeds the processing and rendering capabilities of technical workstations. Parallel algorithms and parallel computers have often been seen as a solution for interactive rendering of such datasets.

Parallel rendering algorithms have been developed in different domains of computer graphics. Developers of graphics hardware have long recognized the need to partition the graphics pipeline amongst several processors in order to achieve fast rendering performance. These efforts resulted in highly specialized architectures that were optimized for particular algorithms and workloads.

As supercomputers became more powerful and less expensive it was a natural step to use them to render and display the results of the computations they were running. This had the advantage of saving time and bandwidth because the data did not need to be transferred from the supercomputer to a dedicated rendering engine. Rendering on supercomputers often does not constitute the most cost-effective solution, e.g. measured as dollars per rendering performance. However, there is no dedicated rendering hardware and all graphics algorithms are implemented in software, thus offering more flexibility in the choice of algorithms and supported rendering features.

This paper will describe and discuss different solutions to the problem of efficient rendering of polygonal models on parallel computers. We will start out with a description of the background of the problem, in particular the polygon rendering process, different rendering scenarios, and issues related to the architecture of the target platform. Then we will discuss ways to classify different parallel rendering algorithms which will provide insights into the properties of different strategies. Finally, we will describe different approaches to achieve load-balancing in parallel rendering systems.

For further study the reader is referred to the papers in the bibliography, in particular [127], and the proceedings of the Parallel Rendering Symposiums and the Eurographics Rendering Workshops.

5.2. Background

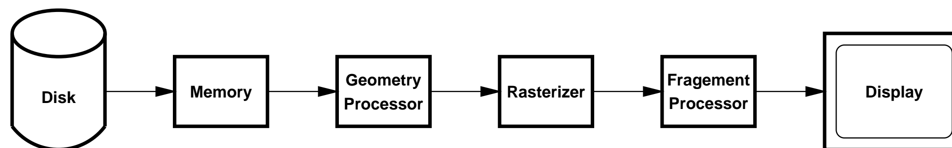

Figure 6: *Simplified model of the rendering pipeline.*

5.2.1. Rendering Pipeline

In this paper we will only consider rendering of polygonal models using the standard rendering pipeline, i.e. we will not discuss ray-tracing or volume rendering. Figure 6 shows the prinicipal steps in rendering of a polygonal model. The description of the

© The Eurographics Association and Blackwell Publishers 1999.

model is stored on disk in some file format such as VRML. Before commencing the actual rendering process, the model must be loaded from disk into main memory and converted into an internal representation suitable for rendering. All further processing steps are then memory-to-memory operations. It should be noted that the order of primitives on disk and in the in-memory representation is arbitrary and is usually determined by the application. In particular, the order of primitives in the should not be relied upon when trying to load-balance parallel processors.

Geometry processing forms the first stage of the rendering pipeline. It includes the steps of transforming objects from their intrinsic coordinate system, e.g. model coordinates, into device coordinates, lighting, computation of texture coordinates, and clipping against the view frustum. Except for clipping, all operations in this stage are performed on vertex information. (Clipping operates on entire polygons which is, in particular on SIMD computers, often disrupting the data flow. The steps in the geometry pipeline can be rearranged such that clipping is postponed until the very end when vertices are reassembled into triangles for rasterization [94, 108]. Geometry processing needs mostly floating point operations to implement the matrix multiplications required to transform vertices and to support lighting calculations. Depending on the number of lights and the complexity of the lighting model geometry processing requires between several hundred and a few thousand floating point operations per vertex.

Rasterization converts primitives (typically triangles) described as screen-space vertices into pixels. The resulting pixels are then subjected to various *fragment processing* operations, such as texture mapping, z-buffering, alpha-blending etc. The final pixel values are written into the frame buffer from where they are scanned out onto the display. Most graphics systems implement rasterization and fragement processing as a unit. One notable exception is the PixelFlow system [36].

Rasterization and fragment processing are use predominantly fixed-point or integer computations. Depending on the complexity of the fragment processing operations, between 5 and up to 50 integer computations per pixel and per triangle are required. Because rasterization is algorithmically simple yet requires such a huge number of operations it is often implemented in hardware.

More details on the computational requirements for the different stages in the rendering pipeline can be found for instance in [37]pp. 866-873.

Finally, the complete image is either sent to the screen for display or written to disk. In many parallel rendering algorithms this step forms a performance bottleneck as partial images stored on different processors have to be merged in one central location (the screen or a disk). (Although this step should be included when measuring the end-to-end performance of a parallel rendering system, some researchers explicitly exclude this step due to shortcomings of their particular target platform [31].)

5.2.2. Single-frame vs. multi-frame rendering

Rendering polygonal models can be driven by several needs. If the model is only used once for the generation of a still image, the entire rendering process outlined above has to be performed. The creation of animation sequences requires rendering of the same model for different values of time and consequently varying values for time-dependent rendering parameters, e.g. view position, object location, or light source intensities. Even though multi-frame rendering could be handled as repeated single-frame rendering, it offers the opportunity to exploit inter-frame coherence. For example, access to the scene database can be amortized over several frames and only the actual rendering steps (geometry processing and rasterization) must be performed for every frame. Other ways to take advantage of inter-frame coherence will be discussed below.

5.2.3. Target Architectures

Parallel polygon rendering has been demonstrated on a large variety of platforms ranging from small multi-processor system using off-the-shelve microprocessors over multi-million dollar supercomputers to graphics workstations built with special-purpose hardware.

In spite of the many differences between those computers, their rendering performance depends on a few key architectural features:

Disk bandwidth determines how fast the model can be loaded from file into memory. For off-line rendering, i.e. storing the image on file instead of displaying it on screen, disk performance also affects how fast the final image can be written back. For large models, disk access time takes up an appreciable portion of the total rendering time and calls for high-performance disk subsystems like disk striping.

Inter-processor communication is required both to exchange or transfer model data between processors and to synchronize the operation of the parallel processors. The former calls for a connection providing high bandwidth for large data packages while the latter requires low latency for small data transfers. Often these two needs result in conflicting technical requirements. The physical interconnection can be formed by a bus, shared memory, dedicated interconnection networks, or by a standard

© The Eurographics Association and Blackwell Publishers 1999.

networking infrastructure like Ethernet. A mismatch between rendering algorithm and communication infrastructure will lead to saturated networks, low graphics performance, and underutilized CPUs. It should be noted that advertised peak bandwidth of a network technology is often based on raw hardware performance and may differ by as much as an order of magnitude from the bandwidth attainable by an application in practice.

Memory bandwidth determines how fast the processor can operate on model and pixel information once that information has been loaded from disk or over the network. The effective bandwidth is determined by the entire memory subsystem, including main memory and the various levels of caching.

Compute power. Both floating point and integer operations must be matched to the algorithm. As note above, geometry processing requires mostly floating point operations while rasterization uses mostly integer operations. The core rendering routines contain only few branches. Note that the available compute power affects only the computational portions of the algorithm. Often computation is outweighed by communication, which leads to the (often surprising and disappointing) effect that increases in compute power have little effect on the overall rendering performance [107].

5.3. Algorithm Classification

For many years the classification of parallel rendering algorithms and architectures has proven to be an elusive goal. We will discuss several such classifications to gain some insight into the design space and possible solutions.

5.3.1. Pipelining vs. Parallelism

Irrespective of the problem domain, parallization strategies can be distinguished by how the problem is mapped onto the parallel processors.

For *pipelining* the problem is decomposed into individual steps that are mapped onto processors. Data travel through the processors and are transformed by each stage in the pipeline. For many problems, like the rendering pipeline (sic!), such a partitioning is very natural. However, pipelining usually offers only a limited amount of parallelism. Furthermore, it is often difficult to achieve good load-balancing amongst the processors in the pipeline as the different functions in the pipeline vary in computational complexity.

To overcome such constraints pipelining is often augmented by *replicating* some or all pipeline stages. Data are distributed amongst those processor and worked on in parallel. If the algorithms executed by each of the processors are identical, the processors can perform their operation in lockstep, thus forming a SIMD (single-instruction, multiple-data) engine. If the algorithms contain too many data dependencies thus making SIMD operation inefficient, MIMD (multiple-instruction, multiple-data) architectures are more useful. SIMD implementations are usually more efficient as the processors can share instructions and require very little interprocessor communication or synchronization.

5.3.2. Object Partitioning vs. Image Partitioning

One of the earliest attempts at classifying partitioning strategies for parallel rendering algorithms took into consideration whether the data objects distributed amongst parallel processors belonged into object space, e.g. polygons, edges, or vertices, or into image space, i.e. collections of pixels such as portions of the screen, scanlines or individual pixels [4]. Object-space partitioning is commonly used for the geometry processing portion of the rendering pipeline, as its operation is intrinsically based on objects. Most parallelization strategies for rasterizers employ image-space partitioning [34, 126, 28, 7, 6, 86] A few architectures apply object-space partitioning in the rasterizer [124, 38, 106].

5.3.3. Sorting Classification

Based on the observation that rendering can be viewed as a sorting process of objects into pixels [35], different parallel rendering algorithms can be distinguished by where in the rendering pipeline the sorting occurs [40]. Considering the two main steps in rendering, i.e. geometry processing and rasterization, there are three principal locations for the sorting step: Early during geometry processing (*sort-first*), between geometry processing and rasterization (*sort-middle*), and after rasterization (*sort-last*).

Figure 7) illustrates the three approaches. In the following discussion we will follow [40] in refering to a pair of geometry processor and a rasterizer as a renderer.

Sort-middle architectures form the most natural implementation of the rendering pipeline. Many parallel rendering systems, both software and hardware, use this approach, e.g. [7, 34, 6, 27, 25, 128, 31]. They assign primitives to geometry processors that implement the entire geometry pipeline. The transformed primitives are then sent to rasterizers that are each serving a portion of

© The Eurographics Association and Blackwell Publishers 1999.

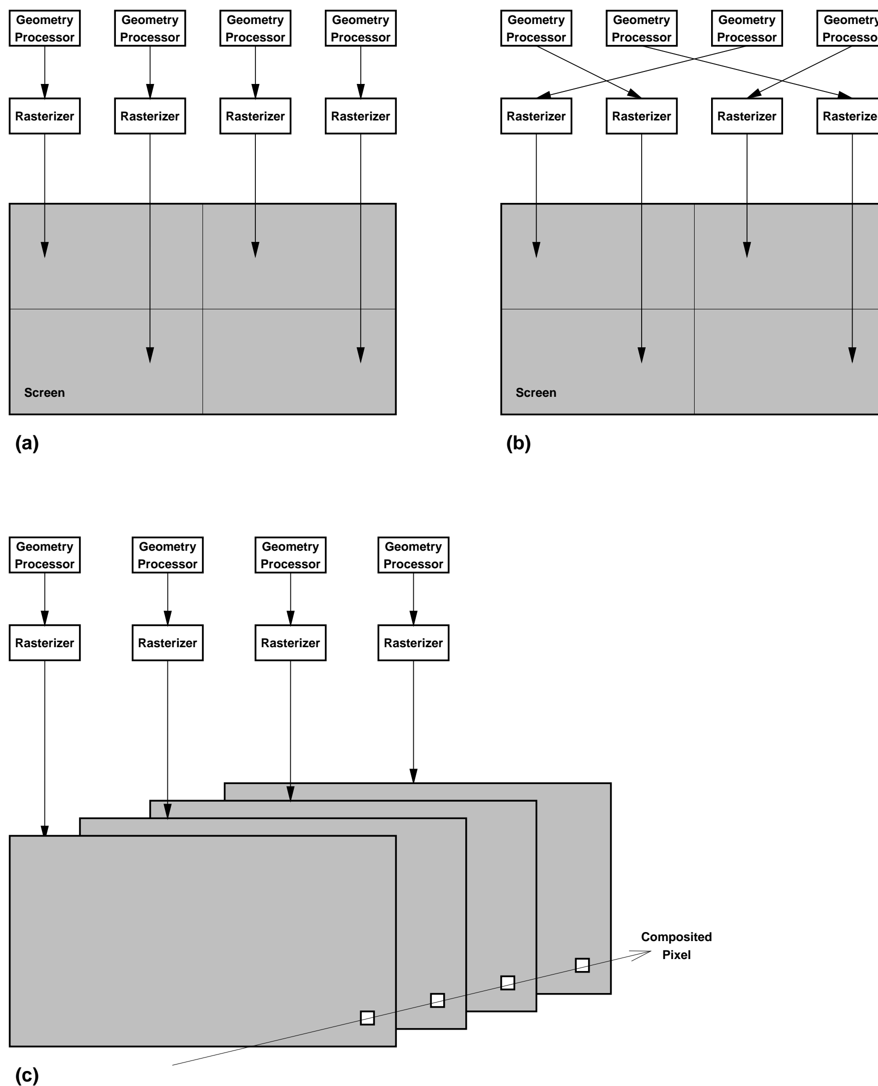

Figure 7: *Classification of parallel rendering methods according to the location of the sorting step. (a) sort-first (b) sort-middle (c) sort-last.*

the entire screen. One drawback is the potential for poor load-balancing among the rasterizers due to uneven distribution of objects across the screen. Another problem of this approach is the redistribution of primitives after the geometry stage which requires a many-to-many communication between the processors. A hierarchical multi-step method to reduce the complexity of this global sort is described in [31].

Sort-last assigns primitives to renderers that generate a full-screen image of all assigned primitives. After all primitives have been processed, the resulting images are merged/composited into the final image. Since all processors handle all pixels this approach offers good load-balancing properties. However compositing the pixels of the partial images consumes large amounts of bandwidth and requires support by dedicated hardware, e.g. [36]. Further, with sort-last implementations it is difficult to support anti-aliasing, as objects covering the same pixel may be handled by different processors and will only meet during the final compositing step. Possible solutions, like oversampling or A-buffers [19], increase the bandwidth requirements during the compositing step even further.

Sort-first architectures quickly determine for each primitive to which screen region(s) it will contribute. The primitive is then assigned to those renderers that are responsible for those screen regions. Currently, no actual rendering systems are based on this

© The Eurographics Association and Blackwell Publishers 1999.

approach even though there are some indications that it may prove advantageous for large models and high-resolution images [87]. [87] claims that sort-first has to redistribute fewer objects between frames than sort-middle. Similar to sort-middle, it is prone to suffer from load-imbalances unless the workload is levelled using an adaptive scheme that resizes the screen regions each processor is responsible for.

5.4. Load Balancing

As with the parallel implementation of any algorithm the performance depends critically on balancing the load between the parallel processors. There are workload-related and design-related factors affecting the load balancing. We will first discuss workload issues and then describe various approaches to design for good load-balance.

Before the discussion of load balancing strategies, we will define terms used throughout the rest of the discussion.

Tasks are the basic units of work that can be assigned to a processor, e.g. objects, primitives, scanlines, pixels etc.

Granularity quantifies the minimum number of tasks that are assigned to a processor, e.g. 10 scanlines per processor or 128x128 pixel regions.

Coherence describes the similarity between neighboring elements like consecutive frames or neighboring scanlines. Coherence is exploited frequently in incremental calculations, e.g. during scan conversion. Parallelization may destroy coherence, if neighboring elements are distributed to different processors.

Load balance describes how well tasks are distributed across different processor with respect to keeping all processors busy for all (or most) of the time. Surprisingly, there is no commonly agreed upon definition of load balance in the literature. Here, we define load balance based on the time between when the first and when the last work task finish.

$$LB = 1 - \frac{T - T_f}{T} \tag{1}$$

where T is the total processing time and T_f is the time when the fastest processor finishes.

5.4.1. Workload Characterization

Several properties of the model are important for analyzing performance and load-balancing of a given parallel rendering architecture. Clipping and object tesselation affect load-balancing during geometry processing, while spatial object distribution and primitive size mostly affect the balance amongst parallel rasterizers.

Clipping. Objects clipped by the screen boundaries incur more work than objects that are trivially accepted or rejected. It is difficult to predict whether an object will be clipped and load-imbalances can result as a consequence of one processor receiving a disproportionate number of objects requiring clipping. There are techniques that can reduce the number of objects that require clipping by enabling rasterizers to deal with objects outside of the view frustum [97, 32]. This reduces the adverse affects of clipping on load-balancing to negligible amounts.

Tesselation. Some rendering APIs use higher order primitives, like NURBS, that are tesselated by the rendering subsystem. The degree of tesselation, i.e. the number of triangles per object, determines the amount of data expansion occuring during the rendering process. The degree of tesselation is often view-dependent and hence hard to predict a priori. The variable degree of tesselation leads to load imbalances as one processor's objects may expand into more primitives than objects handled by another processor. Tesselation also affects how many objects need to be considered during the sorting step. In sort-first architectures, primitives are sorted before the tesselation, thus saving communication bandwidth compared to sort-middle architectures.

Primitive distribution. In systems using image-space partitioning, the spatial distribution of objects across the screen decides how many objects must be processed by each processor. Usually, objects are not distributed uniformly, e.g. more ojects may be located in the center of the screen than along the periphery. This creates potential imbalances in the amount of work assigned to each processor. Below we will discuss different approaces to deal with this problem.

Primitive size. The performance of most rasterization algorithms increases for smaller primitives. (Simply put: It takes less time to generate fewer pixels.) The mix of large and small primitives therefore determines the workload for the rasterizer. Several experiments have shown (see e.g. [22]) that many scenes contain a large number of small objects and a few large objects. The primitive size also affects the overlap factor, i.e. the number of screen regions affected by an object. The overlap factor afffects the performance of image-space partitioning schemes like sort-first and sort-middle algorithms.

© The Eurographics Association and Blackwell Publishers 1999.

5.4.2. Designing for Load-Balancing

Several design techniques are used to compensate for load-imbalances incured by different workloads. They can be distinguished as *static*, *dynamic* and *adaptive*.

Static load balancing uses a fixed assignment of tasks to processors. Although, a low (i.e. no) overhead is incurred for determining this assignment, load imbalances can occur if the duration of tasks is variable. Dynamic load balancing techniques determine the on the fly which processor will receive the next task. Adaptive load balancing determines an assignment of tasks to processors based on estimated cost for each task, thereby trying to assign equal workload to each processor.

We will now look at several concrete techniques to load balance graphics tasks in multi-processor systems.

On-demand assignment

is a dynamic method that relies on the fact that there are many more tasks (objects or pixels) than there are processors. New work is assigned to the first available, idle processor. Except during initialization and for the last few tasks, every processor will be busy all the time. The maximum load imbalance is bounded by the difference in processing time between the smallest (shortest processing time) and largest (longest processing time) task. The ratio of the number of tasks and the number of processors is called the *granularity ratio*. Selecting the granularity ratio requires a compromise between good load balancing (high granularity ratio) and overhead for instance due to large overlap factor (low granularity ratio). The optimal granularity ratio depends on the model, typical values range from about 4 to 32.

An example for dynamic load-balancing through the use of on-demand assignment of tasks is the Pixel-planes 5 system [34]. In Pixel-planes 5, the tasks are 80 128x128 pixel regions that are assigned to the next available rasterizer module. The dynamic distribution of tasks also allows for easy upgrade of the system with more processors.

Care must be taken when applying this technique to geometry processing: Some graphics APIs (like OpenGL) require that operations are performed in the exact order in which they were specified, e.g. objects are not allowed to “pass each other” on their way through the pipeline. MIMD geometry engines using on-demand assignment of objects could violate that assumption and must therefore take special steps to ensure temporal ordering, e.g. by labeling objects with time stamps.

Interleaving

is a static technique which is frequently used in rasterizers to decrease the sensitivity to uneven spatial object distributions. In general, the screen is subdivided into regions, e.g. pixels, scanlines, sets of scanlines, sets of pixel columns, or rectangular blocks. The shape and the size of these regions determines the overlap factor. For a given region size, square regions minimize the overlap factor [40]. Among n processor, each processor is responsible for every n-th screen region. The value n is known as the interleave factor. Since clustering of objects usually occurs in larger screen regions and since every object typically covers several pixels, this technique will eliminate most load-imbalances stemming from non-uniform distribution of objects. Interleaving makes it harder to exploit spatial coherence as neighboring pixels (or scanlines) are assigned to different processors. Therefore, the interleave factor, i.e. the distance between pixels/scanlines assigned to the same processor, must be chosen carefully. Several groups have explored various aspects of interleaving for parallel rasterization, e.g. [56].

Adaptive scheduling

tries to achieve balanced loading of all processors by assigning different number of tasks depending on task size. For geometry processing this might mean to assign fewer objects to processors that are receiving objects that will be tesselated very finely. In image-space schemes this means that processors are assigned smaller pixel sets in regions with many objects, thus equalizing the number of objects assigned to each processor.

Adaptive scheduling can be performed either dyamically or statically. Dynamic adaptation is achieved by monitoring the load-balance and if necessary splitting tasks to off-load busy processors. Such a scheme is described in [128]: Screen regions are initially assigned statically to processors. If the system becomes unbalanced, idle processors grab a share of the tasks of the busy processors.

Statically adaptive schemes attempt to statically assign rendering tasks such that the resulting work is distributed evenly amongst all processors. Such schemes are either predictive or reactive. Predictive schemes estimate the actual workload for the current frame based on certain model properties. Reactive schemes exploit inter-frame coherence and determine the partioning of the next frame based on the workload for the current frame, e.g. [31, 101].

Numereous rendering algorithms using adaptive load-balancing have been described. Most these methods operate in two

© The Eurographics Association and Blackwell Publishers 1999.

steps: First, the workload is estimated by counting primitives per screen regions. Then, either the screen is subdivied to create regions with approximately equal workload or different number of fixed-sized regions (tasks) are assigned to the processors.

One of the earliest such methods was described by Roble [101]. The number of primitives in each screen region are counted. Then, low-load regions are combined to form regions with a higher workload. Regions with high workload are split in half. Since the algorithm does not provide any control over the location of splits, it has the potential of low effectiveness in load-balancing.

Whelan [126] published a similar method that determines the workload by inspecting the location of primitive centroids. High-workload regions are split using a median-cut algorithm, thereby improving the load-balancing behavior over Roble's algorithm. The median-cut approach incurs a sizeable overhead for sorting primitives. The use of centroids instead of actual primitives introduces errors because the actual size of the primitives is not taken into account.

Whitman [127] measures workload by counting the number of primitives overlapping a region. The screen is then subdivided using a quad-tree to create a set of regions with equal workload. The prinicipal problem with the algorithm is that work may be overestimated due to double-counting of primitives.

Mueller [87] improves over the shortcomings of Whelan's method. The Mesh-based Adaptive Hierarchical Decomposition (MAHD) is based on a regular, fine mesh overlaid over the screen. For each mesh cell, primitives are counting in inverse proportion to their size. This approach is experimentally justified and avoids the double-counting problems of Whitman's method. The mesh cells are then aggregated into larger clusters by using a summed-area table for the workload distribution across the screen. The summed-area table is more efficient than the media-cut algorithm in Whelan's method.

Later, Whitman [128] describes an dynamically adaptive scheme. Initially, the screen is subdivided regularly to a predetermined granularity ratio (here: 2). During the actual processing run, processors that complete their tasks early, "steal" work from busier processors by splitting their work region. In order to avoid instability and/or extra overhead, processors steal from the processor with most work left. Also, splitting only occurs if the remaining work exceeds a set threshold. Otherwise, the busy processor finishes his work uninterrupted.

Finally, Ellsworth [31] describes a reactive adaptive load-balancing method. The method is based on a fixed grid overlaid over the screen. Between the frames of an animation, the algorithm counts the number of primitives overlapping each grid cell and uses this count to estimate the workload per cell. Then, cells are assigned to processors for the upcoming frame. The assignment is a multiple-bin-packing algorithm: Regions are first sorted by descending polygon counts; the regions are assigned in this order to the processor with the lightest workload.

Frame-parallel rendering

is a straight-forward method to use parallel processors for rendering. Each processor works independently on one frame of an animation sequence. If there is little variation between consecutive frames, frames can be assigned statically to processors as all processor tend complete their respective frame(s) in approximately the same time. If processing time varies between frames, it is also possible to assign frames dynamically (on-demand assignment). In either case, the processors are working on independent frames and no communication between processors is required after the initial distribution of the model. Unfortunately, this approach is only viable for rendering of animation sequence. It is not suitable for interactive rendering as it typically introduces large latencies between the time a frame is issued by the application and when it appears on the screen.

The *application programming interface (API)* impacts how efficiently the strategies outlined above can be implemented. Immediate-mode APIs like OpenGL or Direct3D do not have access to the entire model and hence do not allow global optimizations. Retained-mode APIs like Phigs, Performer, OpenGL Optimizer, Java3D and Fahrenheit maintain an internal representation of the entire model which supports partitioning of the model for load-balancing.

5.4.3. Data Distribution and Scheduling

In distributed memory architectures, e.g. clusters of workstations or message-passing computers, object data must be sent explicitely to the processors. For small data sets, one can simply send the full data set to every processor and each processor is then instructed which objects to use. This approach fails however for large models either because there is not enough storage to replicate the model at every processor and/or the time to transfer the model is prohibitive due to the bandwidth limitations of the network.

Therefore, most implementations replicate only small data structures like graphics state, e.g. current transformation matrices, light source data, etc., and distribute the storage for large data structures, primarily the object descriptions and the frame buffer.

For system using static assignment of rendering tasks object data have to be distributed only during the initialization phase of the algorithm. This makes it easy to partition the algorithm into separate phases that can be scheduled consecutively.

© The Eurographics Association and Blackwell Publishers 1999.

For dynamic schemes data must be distributed during the entire process. Therefore processors cannot continuously work rendering objects but must instead divide their available cycles between rendering and communicating with other processors. Such an implementation is described in [25]: The system implements a sort-middle architecture where each processor works concurrently on geometry processing and rasterization, i.e. producing and consuming polygons. The advantage is that only a small amount of memory must be allocated for polygons to be transferred between processors. Determining the balance between polygon transformation (generation) and polygon rasterization (consuming) is not obvious. However, [25] states that the overall system performance is fairly insensitive to that choice.

5.5. Summary

Parallel rendering of polygonal datasets faces several challenges most importantly load-balancing. Polygon rendering proceeds in two main steps: geometry processing and rasterization. Both steps have unique computational and communication requirements.

For geometry processing load balancing is usually achieved using on-demand assignment of objects to idle processors. For rasterization, interleaving of pixels or scanlines mostly eliminates load-balancing problems at the expense of less inter-pixel or inter-scanline conherence for each processor. Adaptive load-balancing schemes estimate or measure the workload and divide the screen into regions that will create approximatly equal workload.

6. Parallel Volume Rendering

Volume rendering [60] is a powerful computer graphics technique for the visualization of large quantities of 3D data. It is specially well suited for three dimentional scalar [65, 29, 122, 103] and vector fields [23, 78]. Fundamentally, it works by mapping quantities in the dataset (such as color, transparency) to properties of a cloud-like material. Images are generated by modelling the interaction of light with the cloudy materials [132, 80, 79]. Because of the type of data being rendered and the complexity of the lighting models, the accuracy of the volume representation and of the calculation of the volume rendering integrals [11, 59, 58] are of major concern and have received considerable interest from researchers in the field.

A popular alternative method to (direct) volume rendering is isosurface extraction, where given a certain value of interest $\lambda \in R$, and some scalar function $f : R^3 \rightarrow R$, a polygonal representation for the implicit surface $f(x,y,z) = \lambda$ is generated. There are several methods to generate isosurfaces [70, 81, 51, 92], the most popular being the marching cubes method [70]. Isosurfaces have a clear advantage over volume rendering when it comes to interactivity. Once the models have been polygonized (and simplified [110] – marching cubes usually generate lots of redundant triangles), hardware supported graphics workstation can be used to speed up the rendering. Isosurfaces have several disadvantages, such as lack of fine detail and flexibility during rendering (specially for handling multiple transparent surfaces), and its binary decision process where surfaces are either inside or outside a given voxel tends to create artifacts in the data (there is also an *ambiguity* problem, that has been addressed by later papers like [92]).

6.1. Volumetric Data

Volumetric data comes in a variety of formats, the most common being (we are using the taxonomy introduced in [120]) cartesian or regular data. Cartesian data is typically a 3D matrix composed of voxels (a *voxel* can be defined in two different ways, either as the datum in the intersection of each three coordinate aligned lines, or as the small cube, either definition is correct as long as used consistently), while the regular data has the same representation but can also have a scaling matrix associated with it.

Irregular data comes in a large variety, including curvilinear data, that is data defined in a *warped* regular grid, or in general, one can be given scattered (or unstructured) data, where no explicitly connectivity is defined. In general, scattered data can be composed of tetrahedra, hexahedra, prisms, etc. An important special case is tetrahedral grids. They have several advantages, including easy interpolation, simple representation (specially for connectivity information), and the fact that any other grid can be interpolated to a tetrahedral one (with the possible introduction of Steiner points). Among their disadvantages is the fact that the size of the datasets tend to grow as cells are decomposed into tetrahedra. In the case of curvilinear grids, an accurate decomposition will make the cell complex contain five times as many cells. More details on irregular grids are postponed until Section 6.7.

6.2. Interpolation Issues

In order to generate the cloud-like properties from the volumetric data, one has to make some assumptions about the underlying data. This is necessary because the rendering methods typically assume the ability to compute values as a continuous function,

© The Eurographics Association and Blackwell Publishers 1999.

and (for methods that use normal-based shading) at times, even derivatives of such functions anywhere in space. On the other hand, data is given only at discrete locations in space usually with no explicit derivatives. In order to correctly interpolate the data, for the case of regular sampled data, it is generally assumed the original data has been sampled at a high enough frequency (or has been low-pass filtered) to avoid aliasing artifacts [50]. Several interpolation filters can be used, the most common by far is to compute the value of a function $f(x,y,z)$ by trilinearly interpolating the eight closest points. Higher order interpolation methods have also been studied [18, 76], but the computational cost is too high for pratical use.

In the case of irregular grids, the interpolation is more complicated. Even finding the cell that contains the sample point is not as simple or efficient as in the regular case [88, 99]. Also, interpolation becomes much more complicated for cells that are not tetrahedra (for tetrahedra a single linear function can be made to *fit* on the four vertices). For curvilinear grids, trilinear interpolation becomes dependant on the underlying coordinate frame and even on the cell orientation [130, 48]. Wilhelms et al. [130] proposes using inverse distance weighted interpolation as a solution to this problem. Another solution would be to use higher order interpolation. In general, it is wise to ask the creator of the dataset for a suitable fitting function.

6.3. Optical Models for Volume Rendering

Volume rendering works by modelling volume as cloud cells composed of semi-transparent material which emits its own light, partially transmits light from other cells and absorbs some incoming light [131, 77, 79]. Because of the importance of a clear understanding of such a model to rendering both, regular and irregular grids, the actual inner workings of one such mechanism is studied here. Our discussion closely follows the one in [131].

We assume each volume cells (differentially) emits light of a certain color $E_\lambda(x,y,z)$, for each color channel λ (red, green and blue), and absorbs some light that comes from behind (we are assuming no multiple scattering of light by particles – our model is the simplest “useful” model – for a more complete treatment see [77]).

Correctly defining opacity for cells of general size is slightly tricky. We define the *differential opacity* at some depth z to be $\Omega(z)$. Computing $T(z)$, the fraction of light transmitted through depth 0 to z (assuming no emission of light inside the material), is simple, we just need to notice that the amount of transmitted light at $z+\Delta z$ is just the amount of light at z minus the attenuation $\Omega(z)$ over a distance of Δz:

$$T(z+\Delta z) = T(z) - \Omega(z)T(z)\Delta z \tag{2}$$

what (after making a division by Δz and taking limits) implies

$$\frac{dT(z+\Delta z)}{dz} = -\Omega(z)T(z) \tag{3}$$

The solution to this linear equation of the first order [21] with boundary condition $T(0) = 1$ is:

$$T(z) = e^{-\int_0^z \Omega(u)du} \tag{4}$$

The accumulated opacity over a ray from front-to-back inside a cell of depth d is $(1 - T(d))$. An important special case is when the cell has constant differential opacity Ω, in this case $T(z) = e^{-\Omega z}$. Before we continue, we can now solve the question of defining *differential opacity* Ω from the *unity* opacity (usually user defined and saved in a transfer function table). A simple formula can express Ω in terms of O:

$$\Omega = \log(\frac{1}{1-O}) \tag{5}$$

If the model allows for the emission of light inside the material, a similar calculation can be used to calculate the intensity I_λ for each color channel inside a cell. In this case using an initial intensity $I_\lambda(0) = 0$, the final system and solutions are as follows:

$$\frac{dI_\lambda(z)}{dz} = -\Omega(z)I_\lambda(z) + E_\lambda(z) \tag{6}$$

© The Eurographics Association and Blackwell Publishers 1999.

$$I_\lambda(z) = T(z)\int_0^z \frac{E_\lambda(v)}{T(v)}dv \tag{7}$$

Specializing the solution for constant color and opacity cells (as done above) we get the simple solution:

$$I_\lambda(z) = \frac{E}{\Omega}(1 - e^{-\Omega z}) \tag{8}$$

Usually, for computational efficiency, the exponential in the previous equation is approximated by its first terms in the Taylor series. [131, 77, 79] describe in detail analytical solutions under different assumptions about the behavior of the opacity and emitted colors inside the cells, extensions to more complex light behavior and the several tradeoffs of approximating the exponentials with linear functions.

The previous equations show how to calculate the continuous color and opacity intensity, usually this calculation is done once for every cell, and the results from each cell are *composited* in a later step. Compositing operators were first introduced in [98], and are widely used. The most used operator in volume visualization is the **over** operator, its operation is basically to add the brightness of the current cell to the attenuated brightness of the one behind, and *in the case of front-to-back compositing* update the opacities of the cells. The equations for the **over** operator are:

$$C_o = C_a + C_b(1 - O_a) \tag{9}$$

$$O_o = O_a + O_b(1 - O_a) \tag{10}$$

It is important to note, that in these equations the colors are saved pre-multiplied by the opacities (i.e., the actual color is C_o/O_o), this saves one multiplication per compositing operation.

6.4. Ray Tracing

A popular method to generate images from volume data is to use *ray tracing* or *ray casting* [49, 65]. Ray casting works by casting (at least) one ray per image pixel into volume space, point sampling the scene with some lighting model (like the one just presented) and compositing the samples as described in the previous section. This method is very flexible and extremely easy to implement. There are several extensions of basic ray casting to include higher order illumination effects, like discrete ray tracing [133], and volumetric ray tracing [119]. Both of these techniques take into account global illumination effects incorporating more accurate approximations of the more general rendering equation [58].

Because of its size, volumetric ray casting (and ray tracing) is very expensive. Several optimizations have been applied to ray tracing [66, 67, 26]. One of the most effective optimizations are the *presence acceleration* techniques, that exploit the fact volumetric data is relatively sparse [66, 67, 26, 135, 134]. Levoy [66] introduced the idea by using an octree [104] to skip over empty space. His idea was further optimized by Danskin and Hanrahan [26] to not only skip over empty space, but also to speed up sampling calculations over uniform regions of the volume. Another important acceleration techniques include *adaptive image sampling* and *early ray termination*. Recently, Lacroute and Levoy [63] have introduced a hybrid method that combines some of the previous optimizations in a very efficient class of volume rendering algorithms.

PARC – Hardware-Based Presence Acceleration

Avila, Sobierajski and Kaufman [9, 118] introduced the idea of exploiting the graphics hardware on workstations to speed up volume rendering. First, they introduce PARC (Polygon Assited Ray Casting) [9], a technique that uses the Z-buffer [41] to find the closest and farthest possibly contributing cells. Later, a revised technique [118] is proposed that (still using the Z-buffer) can produce a better approximation of the set of contributing cells.

Their algorithm consists of first creating a polygonal representation of the set of contributing cells (based on axis aligned quadrilaterals) from a *coarse* volume (see Figure 8). The coarse volume is calculated by grouping neighboring voxels together, creating *supervoxels*. Each supervoxel is then tested for the presence of *interesting* voxels (i.e., voxels that belong to the range

© The Eurographics Association and Blackwell Publishers 1999.

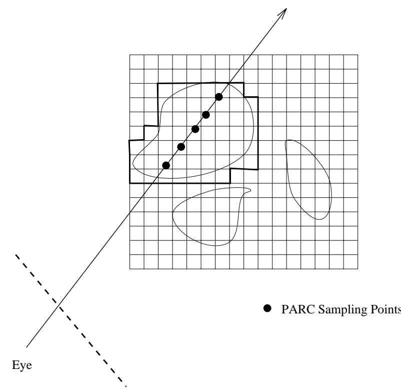

Figure 8: *Polygon Assisted Ray Casting.*

of voxels mapped to non-zero intensities and opacities by the transfer functions). All six external faces of supervoxels are then marked based on its possible visibility (the second method seems to need to project all the faces).

In order to perform the actual rendering, in the first method (called *Depth Buffer PARC*), all the visible quadrilaterals are transformed and scan-converted twice. Once for finding the first non-empty front voxel, and again to determine the final integration location. In the second method (called *Color Buffer PARC*), a sweep along the closest major axis is generated by coloring the PARC cubes with power of two numbers (so they do not interfere with each other), what leaves a footprint of the intervals (t_i, t_{i+1}) that can be used to better sample the regions having interesting voxels. This can be quite a savings, given that volumes are quite sparse (most of the time, only 5-10% of a volume contains any lighting and shading information for a given set of transfer functions).

6.5. Splatting or Projection

Ray casting, described in Section 6.4, works from the image space to the object space (volume dataset). Another way of achieving volume rendering is to reconstruct the image from the object space to the image space, by computing for every element in the dataset its contribution to the image. Several such techniques have been developed [29, 125].

Westover's PhD dissertation describes the *Splatting* technique. In splatting, the final image is generated by computing for each voxel in the volume dataset its contribution to the final image. The algorithm works by virtually "throwing" the voxels onto the image plane. In this process every voxel in the object space leaves a *footprint* in the image space that will represent the object. The computation is processed by virtually "peeling" the object space in slices, and by accumulating the result in the image plane.

Formally the process consists of reconstructing the signal that represents the original object, sampling it and computing the image from the resampled signal. This reconstruction is done in steps, one voxel at a time. For each voxel, the algorithm calculates its contribution to the final image, its footprint, and then it accumulates that footprint in the image plane buffer. The computation can take place in back-to-front or front-to-back order. The footprint is in fact the reconstruction kernel and its computation is key to the accuracy of the algorithm. Westover [125] proves that the footprint does not depend on the spatial position of voxel itself (for parallel projections), thus he is able to use a lookup table to approximate the footprint. During computation the algorithm just need to multiply the footprint with the color of the voxel, instead of having to perform a more expensive operation.

Although projection methods have been used for both regular and irregular grids, they are more popular for irregular grids. In this case, projection can be sped up by using the graphics hardware (Z-buffer and texture mapping) [112].

6.6. Parallel Volume Rendering of Regular Grids

Here, we present a high performance parallel volume rendering engine for our PVR system. Our research has introduced two contributions to parallel volume rendering: *content-based load balancing* and *pipelined compositing*. Content-based load balancing (Section 6.6.2) introduces a method to achieve better load balancing in distributed memory MIMD machines. Pipelined compositing (Section 6.6.3) proposes a component dataflow for implementing the *Parallel Ray Casting* pipeline.

© The Eurographics Association and Blackwell Publishers 1999.

The major goal of the research presented is to develop algorithms and code for volume rendering extremely large datasets at reasonable speed with an aim on achieving real-time rendering on the next generation of high-performance parallel hardware. The sizes of volumetric data we are primarily interested are in the approximate range of 512-by-512-by-512 to 2048-by-2048-by-2048 voxels. Our primary hardware focus is on distributed-memory MIMD machines, such as the Intel Paragon and the Thinking Machines CM-5.

A large number of parallel algorithms for volume rendering have been proposed. Schroeder and Salem [109] have proposed a shear based technique for the CM-2 that could render 128^3 volumes at multiple frames a second, using a low quality filter. The main drawback of their technique is low image quality. Their algorithm had to redistribute and resample the dataset for each view change. Montani et al. [85] developed a distributed memory ray tracer for the nCUBE, that used a hybrid image-based load balancing and context sensitive volume distribution. An interesting feature of their algorithm is the use of clusters to generate higher drawing rates at the expense of data replication. However, their rendering times are well over interactive times. Using a different volume distribution strategy but still a static data distribution, Ma et al. [71] have achieved better frame rates on a CM-5. In their approach the dataset is distributed in a K-d tree fashion and the compositing is done in a tree structure. Others [55, 17, 89] have used similar load balancing schemes using static data distribution, for either image compositing or ray dataflow compositing. Nieh and Levoy [91] have parallelized an efficient volume ray caster citeLevoy:1990:ERT and achieved very impressive performance on a shared memory DASH machine.

6.6.1. Performance Considerations

In analyzing the performance of parallel algorithms, there are many considerations related to the machine limitations, like for instance, communication network latency and throughput [89]. *Latency* can be measured as the time it takes a message to leave the source processor and be received at the destination end. *Throughput* is the amount of data that can be sent on the connection per unit time. These numbers are particularly important for algorithms in distributed memory architectures. They can change the behavior of a given algorithm enough to make it completely impractical.

Throughput is not a big issue for methods based on volume ray casting that perform static data distribution with ray dataflow as most of the communication is amortized over time [85, 55, 17]. On the other hand, methods that perform compositing at the end of rendering or that have communication scheduled as an implicit synchronization phase have a higher chance of experiencing throughput problems. The reason for this is that communication is scheduled all at the same time, usually exceeding the machines architectural limits. One should try to avoid synchronized phases as much as possible.

Latency is always a major concern, any algorithm that requires communication pays a price for using the network. The start up time for message communication is usually long compared to CPU speeds. For instance, in the iPSC/860 it takes at least 200μs to complete a round trip message between two processors. Latency hiding is an important issue in most algorithms, if an algorithm often blocks waiting for data on other processors to continue its execution, it is very likely this algorithm will perform badly. The classic ways to hide latency is to use pipelining or pre-fetching [53].

Even though latency and throughput are very important issues in the design and implementation of a parallel algorithm, the most important issue by far is *load balancing*. No parallel algorithm can perform well without a good load balancing scheme.

Again, it is extremely important that the algorithm has as few inherently sequential parts as possible if at all. Amadahl's law [53] shows how speed up depends on the parallelism available in your particular algorithm and that *any*, however small, sequential part will eventually limit the speed up of your algorithm.

Given all the constraints above, it is clear that to obtain good load balancing one wants an algorithm that:

- Needs low throughput and spreads communication well over the course of execution.
- Hides the latency, possibly by pipelining the operations and working on more than one image over time.
- Never causes processors to idle and/or wait for others without doing *useful work*.

A subtle point in our requirements is in the last phrase, how do we classify *useful work* ? We define useful work as the number of instructions I_{opt} executed by the best sequential algorithm available to volume render a dataset. Thus, when a given parallel implementation uses a suboptimal algorithm, it ends up using a much larger number of instructions than theoretically necessary as each processor executes more instructions than $\frac{I_{opt}}{P}$ (P denotes the number of processors). Clearly, one needs to compare with the best sequential algorithm as this is the actual speed up the user gets by using the parallel algorithm instead of the sequential one.

The last point on useful work is usually neglected in papers on parallel volume rendering and we believe this is a serious flaw in some previous approaches to the problem. In particular, it is widely known that given a transfer function and some segmentation bounds, the amount of useful information in a volume is only a fraction of its total size. Based on this fact, we can

© The Eurographics Association and Blackwell Publishers 1999.

claim that algorithms that use static data distribution based only on spatial considerations are presenting "efficiency" numbers that can be inaccurate, maybe by a large margin.

To avoid the pitfalls of normal static data distribution, we present in the next section a new way to achieve realistic load balancing. Our load balancing scheme, does not scale linearly, but achieves very fast rendering times while minimizing the "work" done by the processors.

6.6.2. Content-Based Load Balancing

This section explains our approach to load balancing, which is able to achieve accurate load balancing even when using presence acceleration optimizations. The original idea of our load balancing technique came from the PARC [9] acceleration technique. We notice that the amount of "work" performed by a presence accelerated ray caster is roughly directly proportional to the number of *full supervoxels* contained in the volume.

We use the number of full supervoxels a given processor is assigned as the measure of how much work is performed by that particular processor. Let P denote the number of processors, and c_i the number of full supervoxels processor i has. In order to achieve a good load balancing (by our metric) we need a scheme that *minimizes* the following function for a partition $X = (c_1, c_2, \ldots)$:

$$f(X) = \max_{i \neq j} \left| c_i - c_j \right|, \forall i, j \leq P \tag{11}$$

Equation 11 is very general and applies to any partition of the dataset D into disjoint pieces D_i. In our work we have tried to solve this optimization problem in a very restricted context. We have assumed that each D_i is convex. (We show later that this assumption makes it possible to create a *fixed* depth sorting network for the partial rays independently calculated each disjoint region.) Furthermore, we only work with two very simple subdivisions: slabs and a special case of a BSP-tree.

Before we go any further, it is interesting to study the behavior of our load balancing scheme in the very simple case of a slab subdivision of the volume D. Slabs (see Figure 9) are consecutive slices of the dataset aligned on two major axes. Given a volume D, with s *superslices* and p processors with the restriction that each processor gets contigous slices, the problem of calculating the "best" load balancing partition for p processors consists of enumerating all the $(s-1)(s-2)\ldots(s-p+1)$ ways of partitioning D, and choosing the one that *minimizes* Equation 11.

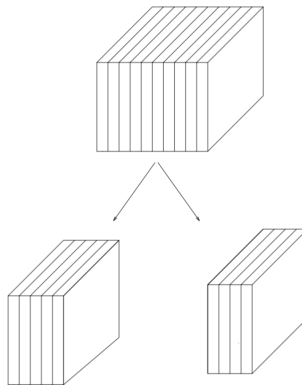

Figure 9: *During slab-based load balancing, each processor gets a range of continuous data set slabs. The number of full supervoxels determines the exact partition ratio.*

The problem of computing the optimal (as defined by our heuristic choice) load balance partition indices can be solved naively as follows. We can compute all the possible partitions of the integer n, where n is the number of slabs, into P numbers, where P is the number of processors (it is actually a bit different, as we need to consider addition non-associative). For example, if $n = 5$, and $P = 3$, then $1 + 1 + 3$ represents the solution that gives the first slab to the first processor, the second slab to the second processor and the remaining three slabs to the third processor. Enumerating all possible partitioning to get the optimal one is a feasible solution but can be very computationally expensive for large n and P. We use a slightly different algorithm for the computations that follows, we choose the permutation with the smallest square difference from the average.

© The Eurographics Association and Blackwell Publishers 1999.

In order to show how our approach works in practice, let us work out the example of using our load balancing example to divide the *neghip* dataset (the negative potential of a high-potential iron protein of 66^3 resolution) for four processors. Here we assume the number of superslices to be 16, and the number of supervoxels to be 64 (equivalent to a level 4 PARC decomposition). Using a voxel threshold of 10-200 (out of a range up to 255), we get the following 16 supervoxel count for each slab, out of the 1570 total full supervoxels:

12, 28, 61, 138, 149, 154, 139, 104, 106, 139, 156, 151, 129, 62, 29, 13

A naive approach load balancing scheme (such as the ones used in other parallel volume renderers) would assign regions of equal volume to each processor resulting in the following partition:

$$12 + 28 + 61 + 138 = 239$$
$$149 + 154 + 139 + 104 = 546$$
$$106 + 139 + 156 + 151 = 552$$
$$129 + 62 + 29 + 13 = 233$$

Here processors 2 and 3 have twice as much "work" as processors 1 and 4. Using our metric, we get:

$$12 + 28 + 61 + 138 + 149 = 388$$
$$154 + 139 + 104 = 397$$
$$106 + 139 + 156 = 401$$
$$151 + 129 + 62 + 29 + 13 = 384$$

One can see that some configurations will yield better load balancing than others but this is a limitation of the particular space subdivision one chooses to implement, the more complex the subdivision one allows, the better load balancing but the harder it is to implement a suitable load balancing scheme and the associated ray caster. Figure 10 plots the examples just described for the naive approach. Figure 11 shows how well our load balancing scheme works for a broader set of processor arrangements.

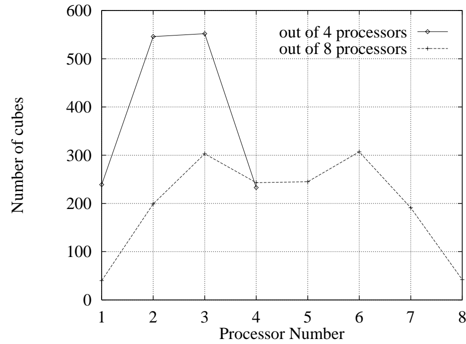

Figure 10: *The graph shows the number of cubes per processor under naive load balancing.*

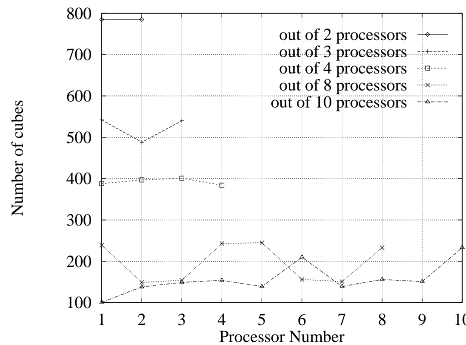

Figure 11: *Load balancing measures for our algorithm. The graph shows the number of cubes the processor receives in our algorithm.*

© The Eurographics Association and Blackwell Publishers 1999.

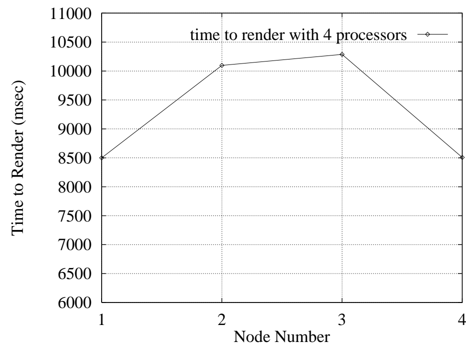

Figure 12: *Naive load balancing on the Paragon. The graph shows the actual rendering times for 4 processors using the naive load balancing.*

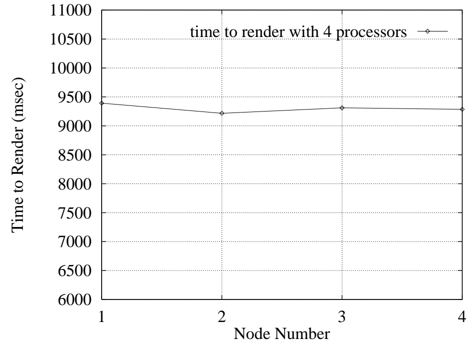

Figure 13: *Our load balancing on the Paragon. The graph shows the actual rendering times for 4 processors using our load balancing.*

These shortcomings of slabs let us to an alternative space decomposition structure previously used by Ma et al. [71, 72], the *Binary Space Partition* (BSP) tree, originally introduced by Fuchs et al. [44].

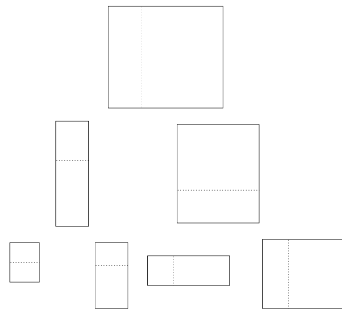

Figure 14: *An example of the partition scheme we used for load balancing. The bottom represents a possible decomposition for 8 nodes. Notice that a cut can be made several times over the same axis to optimize the shape of the decomposition.*

6.6.3. The *Parallel Ray Casting* Rendering Pipeline

Compositing Cluster

The compositing nodes are responsible for regrouping all the sub-rays back together in a consistent manner, in order to keep image correctness. This is only possible because composition is an associative operation, so if we have to sub-ray samples

© The Eurographics Association and Blackwell Publishers 1999.

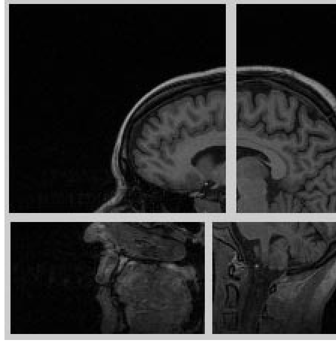

Figure 15: *A cut through the partition acomplished using our load balancing scheme on an MRI head. It is easy to see that if a regular partition scheme were used instead, as the number of processors increase, large number of processors would get just empty voxels to render.*

where one ends where the other starts, it is possible to combine their samples into one sub-ray recursively until we have a value that constitutes the full ray contribution to a pixel.

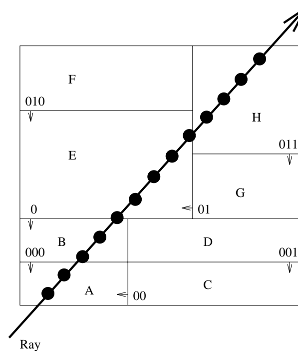

Figure 16: *Data partitioning shown in two dimensions. The dataset is partitioned into 8 pieces (marked A . . . H) in a canonical hierarchical manner by the 7 lines (planes in 3D) represented by binary numbers. Once such a decomposition is performed, it is relatively easy to see how the samples get composited back into a single value.*

Ma et al. [72] use a different approach to compositing, where instead of having separate compositing nodes, the rendering nodes switch between rendering and compositing. Our method is more efficient because we can use the special structure of the sub-ray composition to yield a high performance pipeline, where multiple nodes are used to implement the complete pipeline (see Figure 17). Also, the structure of compositing requires synchronized operation (e.g., there is an explicit structure to the composition, that needs to be guaranteed for correctness purposes), and light weight computation, making it much less attractive for parallelization over a large number of processors, specially on machines with slow communication compared to CPU speeds (almost all current machines).

It is easy to see that compositing has a very different structure than rendering. Here, nodes need to synchronize at every step of the computation, making the depth of the compositing tree a hard limit on the speed of the rendering. That is, if one uses 2^m nodes for compositing, and it takes t_c time to composite two images, even without any synchronization or communication factor in, it takes at least mt_c time to get a completely composited image.

Fortunately, most of this latency can be hiden by pipelining the computation. Here, instead of sending one image at a time, we can send images continuosly into the compositing cluster, and as long as we send images at a rate lower than one for every t_c worth of time, the compositing cluster is able to composite those at full speed, and after mt_c times, the latency is fully hiden from the computation. As can be seen for our discussion, this latency hiding process is very sensitive to the rate of images coming in the pipeline. One needs to try to avoid “stalls” as much as possible. Also, one can not pipe more than the overall capacity of the pipeline.

© The Eurographics Association and Blackwell Publishers 1999.

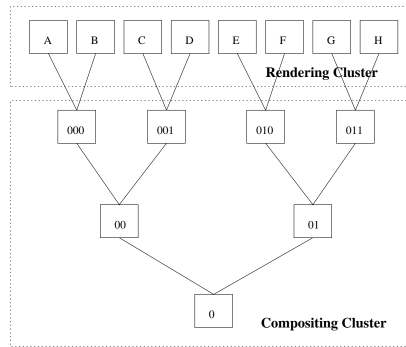

Figure 17: *The internal structure of one compositing cluster, one rendering cluster and their interconnection is shown. In PVR, the communication between the compositing and the rendering clusters is very flexible, with several rendering clusters being able to work together in the same image. This is accomplished by using a set of tokens that are handled by the first level of the compositing tree in order to guarantee consistency. Because of its tree structure, one properly synchronized compositing cluster can work on several images at once, depending on its depth. The compositing cluster shown is relative to the decomposition shown in Figure 16.*

Several implications for real-time rendering can be extracted from this simple model. Even though the latency is hiden from the computation, it is not hiden from the user, at least not totally. The main reason is the overall time that an image takes to be computed. Without network overhead, if an image takes t_r time to be rendered by the rendering cluster, the first image of a sequence takes (at least) time $t_r + mt_c$ to be received by the user. Given that people can notice even very small latencies, our latency budget for real-time volume rendering is extremely low and will definitely have to wait for the next generation of machines to be build. We present a detailed account of the timings later in this chapter.

Going back to the previous discussion, we see that as long as t_r is larger than t_c we don't have anything to worry about with respect to creating a bottleneck in the compositing end. As it turns out, t_r is much larger than t_c, even for relatively small datasets. With this in mind, an interesting question is how to allocate the compositing nodes, with respect to size and topology.

The topology is actually fixed by the corresponding BSP-tree, that is, if the first level of the tree has $n = 2^h$ images (if one image per rendering node, than n would be the number of rendering nodes), than potentially the number of compositing nodes required might be as high as $2^h - 1$. There are several reasons not to use that many compositing nodes. First, it is a waste of processors. Second, the first-image latency grows with the number of processors in the compositing tree. Fortunately, we can lower the number of nodes required in the compositing tree by a process known as virtualization. A general solution to this problem is proposed in Section 6.9.

Types of Parallelism

Due to the fact that each rendering node gets a portion of the dataset, this type of parallelism is called “object-space parallelism”. The structure of our rendering pipeline makes it possible to exploit other types of parallelism. For instance, by using more than a single rendering cluster to compute an image, we are making use of “image-space paralellism” (in PVR, it is possible to specify that each cluster compute disjoint scanlines of the same image; see [117] for the issues related to image-space parallelism). The clustering approach coupled with the inherent pipeline parallelism available in the compositing process (because of its recursive structure) gives rise to a third paralellism type, namely “time-space paralellism” or “temporal parallelism”. In the latter, we can exploit multiple clusters by concurrently calculating sub-rays for several images at once, that can be sent down the compositing pipeline concurrently. Here, it is important for the correctness of the images, that each composition step be done in lockstep, in order to avoid mixing of images. It should be clear by now that there are several advantages to our separation of nodes into our two types.

6.7. Lazy Sweep Ray Casting Algorithm

Lazy Sweep Ray Casting is a fast algorithm for rendering general irregular grids. It is based on the sweep-plane paradigm, and it is able to accelerate ray casting for rendering irregular grids, including disconnected and nonconvex (even with holes)

© The Eurographics Association and Blackwell Publishers 1999.

unstructured irregular grids with a rendering cost that decreases as the "disconnectedness" decreases. The algorithm is carefully tailored to exploit spatial coherence even if the image resolution differs substantially from the object space resolution.

Lazy Sweep Ray Casting has several desirable properties, including its generality, (depth-sorting) accuracy, low memory comsumption, speed, simplicity of implementation and portability (e.g., no hardware dependencies).

The design of our LSRC method for rendering irregular grids is based on two main goals: (1) the depth ordering of the cells should be correct along the rays corresponding to every pixel; and (2) the algorithm should be as efficient as possible, taking advantage of structure and coherence in the data. With the first goal in mind, we chose to develop a new ray casting algorithm, in order to be able to handle cycles among cells (a case causing difficulties for projection methods). To address the second goal, we use a sweep approach, as did Giertsen [48], in order to exploit both *inter-scanline* and *inter-ray* coherence. Our algorithm has the following advantages over Giertsen's:

(1) It avoids the explicit transformation and sorting phase, thereby avoiding the storage of an extra copy of the vertices;
(2) It makes no requirements or assumptions about the level of connectivity or convexity among cells of the mesh; however, it does take advantage of structure in the mesh, running faster in cases that involve meshes having convex cells and convex components;
(3) It avoids the use of a hash buffer plane, thereby allowing accurate rendering even for meshes whose cells greatly vary in size;
(4) It is able to handle parallel and perspective projection within the same framework (e.g, no explicit transformations).

6.7.1. Performing the Sweep

Our sweep method, like Giertsen's, sweeps space with a sweep-plane that is orthogonal to the viewing plane (the x-y plane), and parallel to the scanlines (i.e., parallel to the x-z plane). See Figure 18.

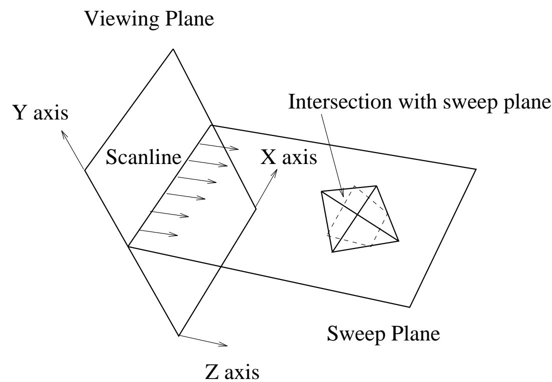

Figure 18: *A sweep-plane (perpendicular to the y-axis) used in sweeping 3-space.*

Events occur when the sweep-plane hits vertices of the mesh S. But, rather than sorting all of the vertices of S in advance, and placing them into an auxiliary data structure (thereby at least doubling the storage requirements), we maintain an event queue (priority queue) of an appropriate subset of the mesh vertices.

A vertex v is *locally extremal* (or simply *extremal*, for short) if all of the edges incident to it lie in the (closed) halfspace above or below it (in y-coordinate). A simple (linear-time) pass through the data readily identifies the extremal vertices.

We initialize the event queue with the extremal vertices, prioritized according to the magnitude of their inner product (dot product) with the vector representing the y-axis ("up") in the viewing coordinate system (i.e., according to their y-coordinates). We do *not* explicitly transform coordinates. Furthermore, at any given instant, the event queue only stores the set of extremal vertices not yet swept over, plus the vertices that are the upper endpoints of the edges currently intersected by the sweep-plane. In practice, the event queue is relatively small, usually accounting for a very small percentage of the total data size. As the sweep takes place, new vertices (non-extremal ones) will be inserted into and deleted from the event queue each time the sweep-plane hits a vertex of S.

The sweep algorithm proceeds in the usual way, processing events as they occur, as determined by the event queue and by the scanlines. We pop the event queue, obtaining the next vertex, v, to be hit, and we check whether or not the sweep-plane encounters v before it reaches the y-coordinate of the next scanline. If it does hit v first, we perform the appropriate

© The Eurographics Association and Blackwell Publishers 1999.

insertions/deletions on the event queue; these are easily determined by checking the signs of the dot products of edge vectors out of v with the vector representing the y-axis. Otherwise, the sweep-plane has encountered a scanline. And at this point, we stop the sweep and drop into a two-dimensional ray casting procedure (also based on a sweep), as described below. The algorithm terminates once the last scanline is encountered.

We remark here that, instead of doing a sort (in y) of all vertices of S at once, the algorithm is able to take advantage of the partial order information that is encoded in the mesh data structure. (In particular, if each edge is oriented in the $+y$ direction, the resulting directed graph is acyclic, defining a partial ordering of the vertices.) Further, by doing the sorting "on the fly", using the event queue, our algorithm can be run in a "lock step" mode that avoids having to sort and sweep over highly complex subdomains of the mesh. This is especially useful, as we see in the next section, if the slices that correspond to our actual scanlines are relatively simple, or the image resolution (pixel size) is large in comparison with some of the features of the dataset. (Such cases arise, for example, in some applications of scientific visualization on highly disparate datasets.)

6.7.2. Processing a Scanline

When the sweep-plane encounters a scanline, the current sweep status data structure gives us a "slice" through the mesh in which we must solve a two-dimensional ray casting problem. (See Figure 19.) Let $\mathcal{S}$ denote the polygonal (planar) subdivision at the current scanline (i.e., $\mathcal{S}$ is the subdivision obtained by intersecting the sweep-plane with the mesh S.) In time linear in the size of $\mathcal{S}$, we can recover the subdivision $\mathcal{S}$ (both its geometry and its topology), just by stepping through the sweep status structure, and utilizing the local topology of the cells in the slice. In our implementation, $\mathcal{S}$ is actually not constructed explicitly, but only given implicitly by the sweep status data structure, and then *locally* reconstructed as needed during the two-dimensional sweep (described below).

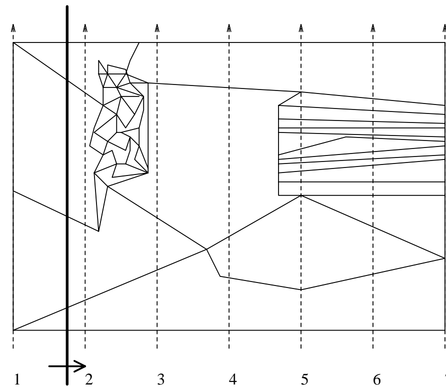

Figure 19: *Illustration of a sweep in one slice.*

The two-dimensional problem is also solved using a sweep algorithm — now we sweep the plane with a sweep-line parallel to the z axis. Events now correspond to vertices of the planar subdivision $\mathcal{S}$. At the time that we construct $\mathcal{S}$, we could identify those vertices in the slice that are locally extremal in $\mathcal{S}$ (i.e., those vertices that have edges only leftward in x or rightward incident on them.), and insert them in the initial event queue. (The actual implementation just sorts along the x-axis, since the extra memory overhead is negligible in 2D.) The *sweep-line status* is an ordered list of the edges of $\mathcal{S}$ crossed by the sweep-line. The sweep-line status is initially empty. Then, as we pass the sweep-line over $\mathcal{S}$, we update the sweep-line status and the event queue at each event when the sweep-line hits an extremal vertex, making insertions and deletions in the standard way. This is analogous to the Bentley-Ottmann sweep that is used for computing line segment intersections in the plane [99]. We also stop the sweep at each of the x-coordinates that correspond to the rays that we are casting (i.e., at the pixel coordinates along the current scanline), and output to the rendering model the sorted ordering (depth ordering) given by the current sweep-line status. We have noticed that the choice of data structure used to maintain the sweep-line status can have a dramatic impact on the performance of the algorithm.

See Silva and Mitchell [116] for details.

6.8. Parallel Rendering of Irregular Grids

Here, we present a distributed-memory MIMD machine parallelization of the LSRC method. Our parallelization is a distributed-memory parallelization, and each rendering node gets a only portion of the dataset, not the complete dataset.

© The Eurographics Association and Blackwell Publishers 1999.

The need for the parallelization of rendering algorithms for irregular-grid rendering is obvious, given the fact that irregular grids are extremely large (as compared to regular grids), and their rendering is much less efficient. The largest irregular grids currently being renderered are just breaking the 1,000,000 cell barrier, what would be equivalent to a 100-by-100-by-100 regular grid, if only data sample points are taken into account. On the other hand, such a grid requires more than 50MB of memory, when its regular counterpart only needs 1MB. Actually, regular grids of this size can be rendered by inexpensive workstations in real-time (i.e., using the Shear-Warp technique), while the irregular grids of this size would be almost out of reach just a year ago.

Actually, the sizes of irregular grids of interest of computational scientists are larger than one million cells, possibly starting at two times that range. (This is subjective data, obtained by talking to researchers at Sandia National Labs during the summer of 1996). Given that it takes us about 150 seconds to render a 500,000 cell complex, and assuming linear behavior (what is not completely correct) it would takes us over 10 minutes to generate images of a 2,000,000 cell complex. What is not an unreasonable amount of time, given that Ma [74] needed over 40 minutes to render a dataset over 10 times smaller.

But our goal is to develop a method that is both faster and scalable to larger and larger dataset. The main reason for this trust is not really current dataset, but those upcoming ones, specially from the new breed of supercomputers, such as the ACSI TeraFlop machine installed at Sandia National Labs. The ACSI machine has orders of magnitude more memory than the current Intel Paragon installed there, even more *usable* memory (i.e., not taking OS and network overhead into account). This will enable the generation of extremely large grids, possibly in ranges of 10,000,000-100,000,000 cells or larger.

Part of this increase in dataset sizes can be offset by better algorithms, specially by further improvements in our rendering code by complete implementation of our optimization ideas. But our experience with irregular grids, seems to show that only more computing power can really offset the increase in dataset size.

The other main reason for the use of parallel machines comes from the pure size of the datasets. The largest workstations available to us have 1GB–3GB of memory, what is very short of the 300GB–512GB of memory in the ASCI Tflop machine. Several reasons indicate the visualization should be performed locally: the fact that very few workstations with more than a few gigabytes of memory are available; moving 300GB of data in and out at ethernet, or even ATM OC-3 speeds is clearly infeasible; disk transfer rates, even for reasonably large (and expensive) disk arrays are just too slow for this kind of data.

As all of the reasons pointed above for the use of the parallel machines that generated the dataset is not enough, we also need to note that these simulations do not generate a single static volume, but in general, time dependent data is being generated and the time steps can not, in general, be efficiently accessed (for obvious reasons).

With all of this in mind, we present our algorithm for rendering irregular grid data, in place, on distributed-memory MIMD machines.

6.8.1. Previous Work

There has been very little work on rendering irregular grid data on distributed memory architectures. Overall parallel work on rendering irregular grids has received relatively little attention. This might be due to the fact that rendering irregular grids is so much harder than regular grids, that few people ever get to the point of being able to research parallel methods for irregular grids.

Uselton has parallelize his original ray tracing work (presented in [123]) in a shared memory multiprocessor SGI, and reported that the implementation scales linearly up to 8 processors. Challinger [20] reports on a parallel algorithm for irregular grids, implemented on a shared-memory BBN-2000 Butterfly. Giertsen [47] has also parallelized his sweep algorithm on a collection of IBM RS/6000, using a master/slave scheme and total data replication in the nodes.

The most interesting work, by our perspective, is Ma [74], where a parallelization technique very similar to the one presented here is proposed. It is unfortunate that he used a sequential ray casting technique that is shown to be at least two orders of magnitude slower than the one we use. Because of this, he did not find any interesting bottlenecks of the parallelization technique.

His technique works by breaking up the original grid into multiple, disjoint cell complexes using Chaco [52], a graph-based decomposition tool developed at Sandia National Labs. Chaco-based decompositions have several interesting and important properties for parallelization of computational methods. It is unclear, the extra overhead of using Chaco has actually any influence on the rendering speed of the parallelization. Here, as in our parallel regular grid method presented in Section 6.6, we divide the nodes into two classes: rendering and compositing nodes. The rendering nodes, compute each ray of an image, creating a set of stencils (the rays may not be completely connected). After each ray is computed, they are sent to the compositing nodes for further sorting and the final accumulation. Each compositing node is assigned a set of rays to be composited.

© The Eurographics Association and Blackwell Publishers 1999.

He reports that because the rendering takes so long, the compositing phase is negligible and he has not work any further on optimizing it.

6.8.2. Parallel LSRC

Overall the our algorithm is very similar to Ma's. Continuing in the tradition of our regular grid work and the framework of our PVR system, we divide the nodes into two relevant groups, rendering and compositing nodes. Our differences between our work and Ma's are actually in the details of the rendering and compositing.

Dataset Decomposition

In order to subdivide the dataset among the nodes, we use a hierarchical decomposition method, with a similar flavor to our load balancing scheme for regular grids. Starting with the bounding box of the complete cell complex, we start making cuts in this box, taking two things into account: the aspect ratio of the cuts, and the number of vertices. At every step, we cut along the largest axis in such a way as to break the number of vertices in half, in each stage of the cutting. because cells might belong to more a single of these convex space decomposition "boxes", we assign the cell to the box that has most of it (e.g., in the number of vertices, with ties broken in some arbitrary, but consistent way).

The obvious now, is just to assign each processor to a each box. This is a way to minimize the total rendering time of the complete irregular grid. Unfortunately, it is not clear that this is the right thing, given that one might want to create a rough picture of the grid fast, then wait for more complex rendering. In the future we expect to be able to create a scattered decomposition, that will have better properties in creating approximate renderings of the grids.

With the decomposition method just proposed, each processor should have roughly the same number of primitives, each of which, approximately confined to a rectangular grid of almost bounded aspect ratio (because of the largest-axis cutting).

Rendering

The rendering performed at each node is just a variation of our sequential technique presented in Section 6.7. This is just a single significant difference, instead of generating an image, every node generates a *stencil* data-structure. Of course, all nodes work concurrenly on generating stencil scan-lines.

The stencil representation of a scan-line is just a linked-list of color and depth of cells, who have been lazily composited. That is, if two stencils shared an end point (e.g., $(\vec{a},\vec{b})$ and $(\vec{b},\vec{c})$), they are composited into a single stencil $(\vec{a},\vec{c})$, representing the whole region. In the end of a scan-line rendering computation, each node potentially has a collection of stencils. Because of the process of decomposing the dataset among the nodes, it is expected the stencil fragmentation is low. This is necessary in order to enable fast communication for compositing.

Compositing

One solution for compositing would just to copy Ma's technique, where nodes are responsible for certain scan-lines. This way the rendering nodes could just send its collection of stencils for further sorting in the compositing nodes. In our case, we try to achieve better performance by creating a tree of compositing nodes (such as the one we use for the regular case). Every compositing node is reponsible for a certain region of space (i.e., one of the original box decompositions proposed above), that belongs to a global BSP-tree.

It is the responsability of the rendering nodes to respect the BSP-tree boundaries and send the data to the correct compositing nodes, possibly breaking stencils that are span across boundaries.

Once the data of each scan-line is received in the compositing nodes, the final depth sorting can be efficiently performed by merging the stencils into a complete image. An efficient pipeline scheme can be implemented on a scan-line by scan-line basis, with similar good properties as the one implemented image-by-image for the regular grid case.

6.9. General BSP-tree Compositing

A simple way of parallelizing rendering algorithms is to do it at the object-space level: *i.e.*, divide the task of rendering different objects among different rendering processors, and then compose the full images together. A large class of rendering algorithms (although not all), in particular scan-line algorithms, can be parallelized using this strategy. Such parallel rendering architectures, where renderers operate independently until the visibility stage, are called *sort-last* (SL) architectures [84]. A fundamental

© The Eurographics Association and Blackwell Publishers 1999.

advantage of SL architecture is the overall simplicity, since it is possible to parallelize a large class of existing rendering algorithms without major modifications. Also, such architectures are less prone to load imbalance, and can be made linearly scalable by using more renderers [82, 83]. One shortcoming of SL architectures is that very high bandwidth might be necessary, since a large number of pixels have to be communicated between the rendering and compositing processors. Despite the potential high bandwidth requirements, sort-last has been one of the most used, and successful parallelization strategies for both volume rendering and polygon rendering, as shown by the several works published in the area [24, 129, 64, 72].

Here we present a general purpose, optimal compositing machinery that can be used as a black box for efficiently parallelizing a large class of sort-last rendering algorithms. We consider sort-last rendering pipelines that are based on separating the rendering processors from the compositing processors, similar to what was proposed previously by Molnar [82]. The techniques described in this paper optimize overall performance ans scalability without sacrificing generality or the ease of adaptability to different renderers. Following Molnar, we propose to use a scan-line approach to image composition, and to execute the operations in a pipeline as to achieve the highest possible frame rate. In fact, our framework inherits most of the salient advantages of Molnar's technique. The two fundamental differences between our pipeline and Molnar's are:

(1) instead a fixed network of Z-buffer compositors, our approach uses a user-programmable BSP-tree based composition tree;
(2) we use general purpose processors and networks, instead of Molnar's special purpose Z-comparators arranged in a tree.

In our approach, hidden-surface elimination is not performed by Z-buffer alone, but instead by executing a BSP-tree model. This way, we are able to offer extra flexibility, and instead of only providing parallelization of simple depth-buffer scan-line algorithms, we are able to provide a general framework that adds support for true transparency, and general depth-sort scan-line algorithms. In trying to extend the results of Molnar to general purposes parallel machines, we must deal with a processor allocation problem. The basic problem is how to minimize the amount of processing power devoted to the compositing back-end and still provide performance guarantees (*i.e.*, frame rate guarantees) for the user. We propose a solution to this problem in the paper.

In our framework the user defines a BSP-tree, in which the leaves correspond to renderers (the renderers perform user-defined rendering functions). Also, the user defines a data structure for each pixel, and a compositing function, that will be applied to each pixel by the internal nodes of the BSP-tree previously defined. Given a pool of processors to be used for the execution of the compositing tree, and a minimum required frame rate, our processor allocation algorithm partitions the compositing operations among processors. The partition is chosen so as to minimize the number of processors without violating the frame-rate needs. During rendering, the user just needs to provide a viewpoint (actually, for optimum performance, a sequence of viewpoints, since our algorithm exploits pipelining). Upon *execution* of the compositing tree, messages are sent to the renderers specifying where to send their images, so no prior knowledge of the actual compositing order is necessary on the (user) rendering nodes side. For each viewpoint provided, a *complete* image will be generated, and stored at the processor that was allocated the root of the compositing tree. The system is fully pipelined, and if no stalls are generated by the renderers, our system guarantees a frame rate at which the user can collect the full images from the root processor.

6.9.1. Optimal Partitioning of the Compositing Tree

We can view the BSP tree as an expression tree, with compositing being the only operation. In our model of compositing clusters, evaluation of the compositing expression is mapped on to a *tree* of *compositing processes* such that each process evaluates exactly one sub-expression. See Figure 20 for an illustration of such a mapping. The actual ordering of compositing under a BSP-tree depends not only on the position of the nodes, but also on the viewing direction. So, during the execution phase, a specific ordering has to be obeyed. Fortunately, given any partition of the tree, each subtree can still be executed independently. Intuitively, correctness is achieved by having the nodes "fire up" in a on-demand fashion.

Such a decompositing is based on a model of the cost of the subtrees. For details on this, and the partitioning algorithm, shown in Figure 21, see Ramakrishnan and Silva [100].

Practical Considerations

Algorithm *partition* provides a simple way of given a BSP-tree, and a performance requirement, given in terms of the frame rate, how to divide up the tree in such a way as to optimize the use of processors. Several issues, including machine architecture bottlenecks, such as synchronization, interconnection bandwidth, mapping the actual execution to a specific architecture (*e.g.*, a mesh-connected MIMD machine) were left out of the previous discussion. We now describe how Algorithm *partition* can be readily adapted to account for some of the above issues in practice.

Compositing Granularity: Note that there is nothing in the model that requires that full images be composited and transfered

© The Eurographics Association and Blackwell Publishers 1999.

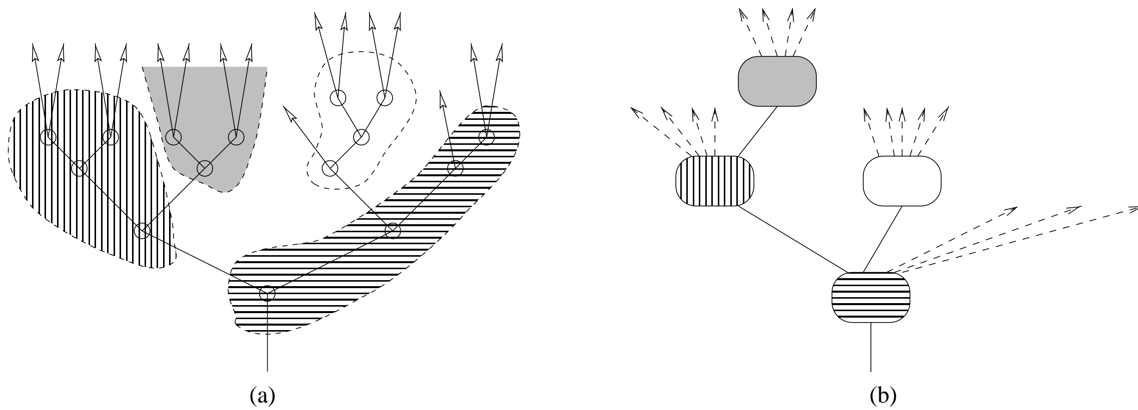

Figure 20: *(a) A BSP tree, showing a grouping of compositing operations and (b) the corresponding tree of compositing processes. Each compositing process can be mapped to a different physical node in the parallel machine.*

```
Algorithm partition(u)
     /* The algorithm marks the beginning of partitions in
        the subtree of G rooted at u. If more vertices,
        can be added to the root partition, the algorithm
        returns the size of the root partition.
        Otherwise, the algorithm returns 0. */
1.   if (arity(u) = 2) then    /* u is a binary vertex */
2.        w1 := partition(left_child(u));
3.        w2 := partition(right_child(u));
4.        w := w1 + w2 + 1;
5.        if (w > K) then
6.             if (w1 ≤ w2) then
7.                  Mark right_child(u) as start of new partition
8.                  w := w1 + 1;
9.             else
10.                 Mark left_child(u) as start of new partition
11.                 w := w2 + 1;
12.  else if (arity(u) = 1) then    /* u is a unary vertex */
13.       w := partition(child(u)) + 1;
14.  else    /* u is a leaf */
15.       w := 1;
16.  if (w = K) then
17.       Mark u as a start of new partition
18.       return(0);
19.  else
20.       return(w);
```

Figure 21: *Algorithm partition*

one at a time. Actually, one should take into consideration when determining the unit size of work, and communication, hardware constraints such as memory limitations, and bandwidth requirements. So, for instance, instead of messages being a full image, it might be better to send a pre-defined number of scan-lines. Notice that in order for images of arbitrary large size to be able to be computed, the rendering algorithm must also be able to generate the images in scan-line order.

Communication Bandwidth: Of course, in order to achieve the desired frame rate, enough bandwidth for distributing the images during composition is stritly necessary. Given p processors, each performing k compositing operations, the overall aggregate bandwidth required is proportional to $p(k+2)$. It should be clear that as k_{max} increases, the actual bandwidth re-

© The Eurographics Association and Blackwell Publishers 1999.

quirement actually decreases (both for the case of a SL-full, as well as a SL-sparse architecture) since as k_{max} increases the number of processors required decreases. This decrease in bandwidth is due to the fact that compositing computation are performed locally, inside each composite processor, instead of being sent over the network. If one processor performs exactly k_{max} compositing operations, it needs $k_{max}+2$ units of bandwidth, as opposed to $3k_{max}$ when using one processor per compositing operation— a bandwidth savings of almost a factor of three!

Another interesting consideration related to bandwidth is the fact that our messages tend to be large, implying that our method operates on the best range of the message size versus communication bandwidth curve. For instance, for messages smaller than 100 bytes the Intel Paragon running SUNMOS achieve less than 1 MB/sec bandwidth, while for large messages (i.e., 1MB or larger), it is able to achieve over 160MB/sec. (This is very close to 175MB/sec, which is the peak hardware network performance of the machine.) As will be seen in Section 6.9.2, our tree execution method is able to completely hide the communication latency, while still using large messages for its communication.

Latency and Subtree Topology: As will be seen in Section 6.9.2, the whole process is pipelined, with a request-based paradigm. This greatly reduces the overhead of any possible synchronization. Actually, given enough compositing processors, the overall time is only dependant on the performance of the rendering processors. Also, note that the actual *shape* of the subtree that a given processor gets is irrelevant, since the execution of the tree is completely pipelined.

Architectural Topology Mapping: We do not provide any mechanism for optimizing the mapping from our tree topology to the actual processors in a given architecture. With recent advancements in network technology, it is much less likely that the use of particular communication patterns improve the performance of parallel algorithms substantially. In new architectures, the point-to-point bandwidth in access of 100–400 MB/sec are not uncommon, while in the old days of the Intel Delta, it was merely on the order of 20 MB/sec. Also, network switches, with complex routing schemes, are less likely to make neighbor communication necessary. (Actually, the current trend is not to try to exploit such patterns since new fault-handling and adaptive routers usually make such tricks useless.)

Limitations of Analytical Cost Model: Even though we can support both SL-full and SL-sparse architecture, our model does not make any distinction of the work that a given compositing processor is performing based on the depth of its compositing nodes. This is one of the limitations of our analytical formulation. However, the experimental results indicate that this limitation does not seem have any impact on the use of our partitioning technique in practice. Actually, frame-to-frame differences might diminish the concrete advantage of techniques that try to optimize for this fact.

6.9.2. Optimal Evaluation

In the previous section, we described techniques to partition the set of compositing operations and allocate one processor to each partition, such that the various costs of the compositing pipeline can be minimized. We now describe efficient techniques for performing the compositing operations within each processor.

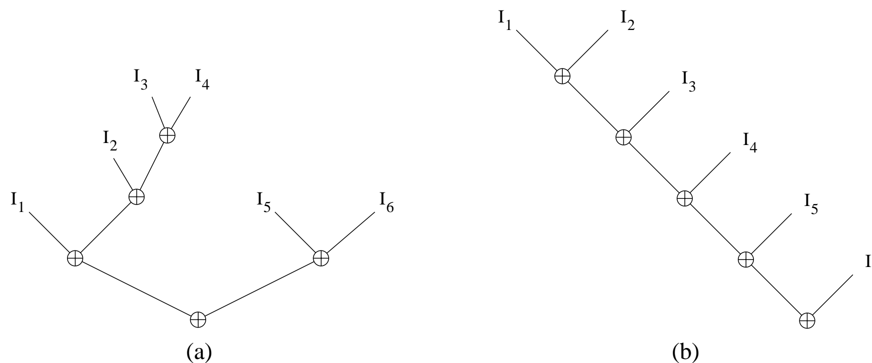

Figure 22: *(a) A compositing tree and (b) its corresponding associative tree.*

Space-Optimal Sequential Evaluation of Compositing Trees

Storage is the most critical resource for evaluating a compositing tree. We need 4MB of memory to store an image of size 512×512, assuming 4-bytes each for RGB and α values per pixel. Naive evaluation of a compositing tree with N nodes may require intermediate storage for up to N images.

© The Eurographics Association and Blackwell Publishers 1999.

We now describe techniques, adapted from register allocation techniques used in programming language compilation, to minimize the total intermediate storage required. Figure 22a shows a compositing tree for compositing images I_1 through I_6. We can consider the tree as representing the expression

$$(I_1 \oplus (I_2 \oplus (I_3 \oplus I_4))) \oplus (I_5 \oplus I_6) \tag{12}$$

where $\oplus$ is the compositing operator. Since images I_1 through I_6 are obtained from remote processors, we need to copy these images locally into intermediate buffers before applying the compositing operator. The problem now is to sequence these operations and reuse intermediate buffers such that the total number of buffers needed for evaluating the tree is minimized.

We encounter a very similar problem in a compiler, while generating code for expressions. Consider a machine instruction (such as integer addition) that operates only on pairs of registers. Before this operation can be performed on operands stored in the main memory, the operands must be loaded into registers. We now describe how techniques to generate optimal code for expressions can be adapted to minimize intermediate storage requirements of a compositing process. The number of registers needed to evaluate an expression tree can be minimized, using a simple tree traversal algorithm [5]pages 561–562. Using this algorithm, the compositing tree in Figure 22a can be evaluated using 3 buffers. In general, $O(\log N)$ buffers are needed to evaluate a compositing tree of size N. However, by exploiting the algebraic properties of the operations, we can further reduce the number of buffers needed— to $O(1)$. Since $\oplus$ is associative, evaluating expression (12) is equivalent to evaluating the expression:

$$((((I_1 \oplus I_2) \oplus I_3) \oplus I_4) \oplus I_5) \oplus I_6 \tag{13}$$

The above expression is represented by the compositing tree in Figure 22b, called an *associative tree* [111]. The associative tree can be evaluated using only 2 buffers.

Again, for full details, we refer the reader to the full paper [100].

6.9.3. Implementation

In this section, we sketch the implementation of our compositing pipeline. We implemented our compositing back-end in the PVR system [115]. PVR is a high-performance volume rendering system, and it is freely available for research purposes. Our main reason for choosing PVR was that it already supported the notion of separate rendering and compositing clusters, as explained in [113]Chapter 3. The basic operation is very simple. Initially, before image computation begins, all compositing nodes receive a BSP-tree defining the compositing operations based on the object space partitioning chosen by the user. Each compositing node, in parallel, computes its portion of the compositing tree, and generates a view-independent data structure for its part. Image calculation starts when all nodes receive a sequence of viewpoints.

The rendering nodes, simply run the following simple loop:

```
For each (viewpoint v)
  ComputeImage(v);
  p = WaitForToken();
  SendImage(p);
```

Notice that the rendering nodes do not need any explicit knowledge of parallelism; in fact, each node does not even need to know, *a priori*, where its computed image is to be sent. Basically, the object space partitioning and the BST-tree takes care of all the details of parallelization.

The operation of the compositing nodes is a bit more complicated. First, (for each view) each compositing processor computes (in parallel, using its portion of the compositing tree) an array with indices of the compositing operations assigned to it as a sequence of processor numbers from which it needs to fetch and compose images. The actual exectution is basically an implementation of the prefecthing scheme proposed here, with each `read_request` being turned into a `PVR_MSG_TOKEN` message, where the value of the token carries its processor id. So, the basic operation of the compositing node is:

```
For each (viewpoint v)
  CompositeImages(v);
  p = WaitForToken();
  SendImage(p);
```

Notice that there is no explicit synchronization point in the algorithm. All the communication happens bottom-up, with requests being sent as early as possible (in PVR, tokens are sent asynchronously, and in most cases, the rendering nodes do not wait for the tokens), and speed is determined by the slowest processor in the overall execution, effectively pipelining the computation. Also, one can use as many (or as few) nodes one wants for the compositing tree. That is, the user can determine

© The Eurographics Association and Blackwell Publishers 1999.

the rendering performance for a given configuration, and based on the time to composite two images it is straightforward simple to scale our compositing back-end for his particular application.

© The Eurographics Association and Blackwell Publishers 1999.

PART THREE

Case Studies

Cláudio T. Silva* Arie E. Kaufman* Constantine Pavlakos‡

*State University of New York at Stony Brook ‡Sandia National Laboratories

6.10. Introduction

Volume rendering [60] is a powerful computer graphics technique for the visualization of large quantities of 3D data. It is specially well suited for the visualization of three dimentional scalar and vector fields. Fundamentally, it works by modelling the volume as cloudy-like cells composed of semi-transparent material which emits their own light, partially transmits light from other cells and absorbs some incoming light. (See sidebar *Volume Rendering* for details).

In order to allow researchers and engineers make effective use of volume rendering to study complex physical and abstract structures, a coherent, powerful, easy to use visualization tool is needed. Furthermore, such a tool should allow for *interactively* visualization, ideally with support for user-defined "computational steering".

There are several issues and challenges in developing such visualization tools. (1) So much as the latest volume rendering acceleration techniques running on top-of-the-line workstations, it still takes a few seconds to a few minutes to volume render images. This is clearly far from interactive. With the advent of larger parallel machines, better scanners and instrumentation, larger and larger datasets (typically from 32MB to 512MB, but with sizes as high as 16GB) are being generated, some of which would not even fit in memory of a workstation class machine. (2) Even if rendering time is not a major concern, big datasets may be expensive to hold in storage, and extremely slow to transfer to typical workstations over network links.

These issues lead to the question of whether the visualization should be performed directly on the parallel machines which is used to generate the simulation data or sent over to a high performance graphics workstation for post-processing. First, if the visualization software was integrated in the simulation software, there would be no need for extra storage and visualization could be an active part of the simulation. Second, large parallel machines can render these datasets faster than workstations can, possibly in real-time or at least achieving interactive frame-rates. Finally, the integration of simulation and visualization in one tool (when possible) is highly desirable, because it allows users to interactively "steer" the simulation, and possibly terminate (or modify parameters in the) simulations instead of performing painfully long simulations on extremely expensive machines, with high cost of storage and transmission, only to find out at post-processing that the simulations are wrong or uninteresting.

In this paper we introduce the PVR system, currently being developed under a collaboration between Sandia National Laboratories and the State University of New York at Stony Brook. PVR is a component approach to building a distributed volume visualization system. On its topmost level it provides a flexible and high performance client/server volume rendering architecture to the user with a unique load balancing scheme which provides a continuum of cost/performance parameters that can be used to optimize rendering speed. The original goals of PVR were to achieve a level of portability and performance for rendering beyond that of other available systems and to provide a platform that can be used for further development.

But PVR is more than a rendering system, its components were specially designed to be user-extensible, in order to allow for user defined computational steering (that is, the user can easily add his own computational code to PVR and just link in our rendering library). Using PVR, it is much simpler to build portable, high performance, complex distributed visualization systems (Figure 23).

The rest of the paper introduces the PVR client/server architecture and its components, with emphasis on its application to volume rendering. Details on how to achieve computational steering with PVR are scattered across this paper.

6.11. The PVR System

It is well known that system complexity always limits the reliability of large software projects. Distributed systems exacerbate this problem with the introduction of asynchronous and non-local communication. With all of this in mind, we use a component approach to developing PVR. PVR attempts to provide just enough functionality in the basic system to allow for the development of large and complex computational steering and visualization applications. It is based on a client/server architecture, where there are coupled rendering/computing servers on one side, and on the other the user acts as a client (from his workstation).

The PVR client/server architecture is implemented in two main components: the *pvrsh*, which runs in the user workstation,

© The Eurographics Association and Blackwell Publishers 1999.

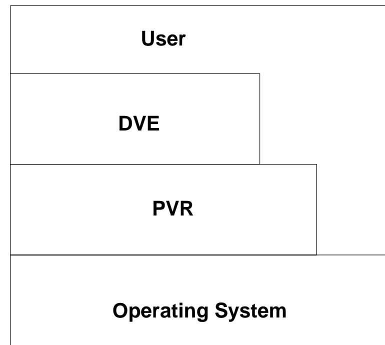

Figure 23: *The relationship of Distributed Visualization Environment (DVE) systems and PVR.*

and the *PVR renderer*, running in parallel machines. The renderer is implemented as a library and it allows for easy integration of user defined code that can share the same processors as the rendering. Communication across aplications writen with PVR, are performed using the PVR protocol, and in our implementation communication is handled by separate UNIX processes (see Figure 24).

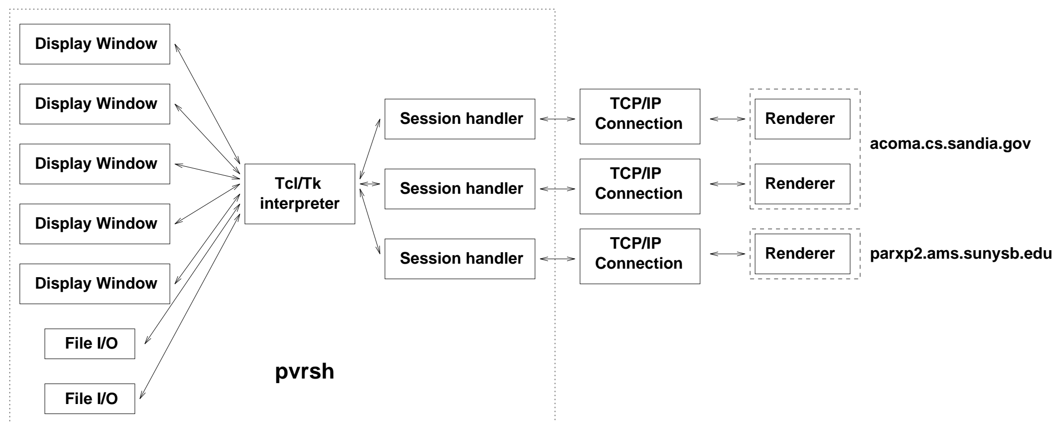

Figure 24: *PVR Architecture. The overall structure of the system is shown with emphasis on the* pvrsh*. The Tcl/Tk core acts as glue for all the client components. Everything with the exception of the* renderers *run on the user's workstation. The* renderers *run remotely on the parallel machines.*

6.11.1. The *pvrsh*

The *pvrsh* is an augmented Tcl/Tk shell. It provides a single new object to the user, the *PVR session*. We chose to use Tcl/Tk [95] as the system glue. Tcl (*Tool command language*) is a script language designed to be used as a generic language in application programs. It is easily extendable with new user commands (in C or Tcl) and coupled with the graphical environment Tk, it is a powerful graphical user interface system. The use of the Tcl/Tk, which are well-designed, debugged application language and graphical environment contribute to reducing the overall system complexity.

The *PVR session* is an object (such as the Tk objets). It contains attributes and corresponding methods (used to change those attributes). One of the most important attributes is the one that *binds* a session to a particular parallel machine. Figure 24 contains three sessions, two on acoma.cs.sandia.gov (a large Intel Paragon XP/S with over 1840 nodes running SUNMOS [75] installed at Sandia) and one on parxp2.ams.sunysb.edu (a small Intel Paragon with 110 nodes running Intel's version of OSF/1 installed at Stony Brook). The system is designed to handle multiple sessions using the same protocol with machines running different operating systems.

As part of its attributes a session specifies the number of nodes it needs, and the parameters that are passed to those nodes.

© The Eurographics Association and Blackwell Publishers 1999.

Several pieces of informations are *interactively* exchanged between the *pvrsh* and the *PVR renderer*, such as rendering configuration information, rendering commands, sequences of images, performance and debuging information.

There is a high amount of flexibility in the specification of the rendering. Not only simple rendering elements, such as changing transformation matrices, transfer functions, image sizes, datasets can be specified, but there are commands to specify (in a high level format) the complete parallel rendering pipeline (see sidebar *Parallel Volume Rendering* for details). With these parameters in hand, the *pvrsh* can be used to specify almost arbitrary scalable rendering configurations (see Section 6.11.4).

The *pvrsh* is implemented as a single process (used to make ports easier) in about 5,000 lines of C code. We have augmented the Tcl/Tk interpreter with TCP/IP connection capabilities (some versions of Tcl/Tk have this build in). Due to the need of several concurrent sessions, all the communication is performed asynchronous. We use `Tk_CreateFileHandler()` routine to arbitrate between input from the different sessions (a UNIX `select` call and polling could be used instead, but would make the code harder to understand and overall more complex). Sessions work as interrupt driven commands, responding to requests one at a time (every session can receive events from two sources at the same time, the user keyboard and the remote machine). Locking and disabling interrupt are needed to ensure consistency inside critical sessions.

The overall structure of the code allows for user augmentation of a session's functionality either by external or internal means. *External* augmentation is the one that can be performed without re-compilation (such as the one used by the user interface to show the images as they are received asynchronously from the remote parallel server). *Internal* augmentation requires changes to the source code. The source code is structured to allow for simple addition of new functionality. Only a single file needs to be changed to add a new session method (if it changes the *Resource Database* [113], two files need to be changed). New commands are added using Tcl conventions (see Part 3 of Ousterhout's Tcl/Tk book [95]).

Every PVR message is sent either as a single fixed length message, or as two messages (the first is used to specify the size of the second). This is used to make redirection easier and to achieve optimal performance under different configurations. Look up tables are set up with actions to be taken on the arrival of each message type. This setup makes additions to the PVR protocol very simple.

6.11.2. The *PVR renderer*

The *PVR renderer* is the piece of PVR that runs remotely on the parallel machine (see Figure 24). It is composed of several components, the most complex being the renderer itself. In order to start up multiple the parallel processes at the remote machine, we use *pvrd*, the PVR daemon. This daemon runs in the parallel machine. It waits on a well-known port for connection requests. Once a request for opening a new session is made it *forks* a handling process that will be responsible for allocating processors and communicating with the session on the client. In the remote machine, the handling process allocates the computing nodes, and runs the renderer code on them. The connection process is illustrated in Figure 25. One *pvrd* can allocate several processes, once it is killed, it kills all its children before exiting.

The renderer is the code that actually runs on the parallel nodes. The overall structure of the code resembles an SIMD machine [54], where there are high-level commands and low-level commands. There is one *master* node (similar to the microcontroller on the CM-2 machines), and several *slave* nodes. The functions of the slaves are completely dependent on the master. The master receives commands from the *pvrsh*, translates them, and takes the necessary actions, including changing the state of the slaves and sending them a detailed set of instructions.

For flexibility and performance, the method of sending instructions to the nodes are through *action tables* (like SIMD microcode). In order to ask the node to perform some action, the master broadcast the address of the function to be executed. Upon receiving that instruction, the slaves execute that particular function. With this method, it is very simple to add new functionality, because any new added functionality can be performed locally, without the need to change global files. Also, every function can be optimized independently, with its own communication protocol. One shortcoming of this communication method (as in SIMD machines) is that one has to be careful with non-uniform execution, specially because the Intel NX communication library (both OSF and SUNMOS have support for NX) has limited functionality for handling nodes as groups (e.g., in setting up barriers with NX it is impossible to select a group from the totality of the allocated nodes).

The master intrinsically divides the nodes into *clusters*. Each cluster has specialized computational task, and multiple clusters can cooperate in groups to achieve a large task. All that is necessary for cluster configuration is that the basic functions be specified in user-defined libraries that are linked in a single binary. During runtime, the user can use the master to reconfigure his clusters accordingly to his immediate goal. The *pvrsh* can be used to *interactively* send such commands. As an example of the use of such scheme, see Figure 26, where the rendering configuration for PVR's high performance volume renderer is specified.

In order to achieve user-defined computational steering, one can use this clustering paradigm. It will usually be necessary to

© The Eurographics Association and Blackwell Publishers 1999.

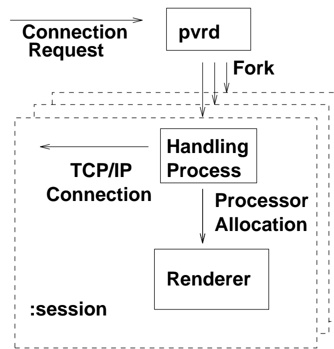

Figure 25: *In order to allocate nodes, the* pvrsh *sends a command to the* pvrd*, which in turns creates a special communication handling process and allocates a partition on the parallel machine.*

add one's functionality to the action tables (e.g., linking the computational code with PVR dispatching code), and also add extra options to the *pvrsh* (usually through the `set` command) for modifying the relevant parameters interactively.

PVR's volume rendering code was the inspiration for this overall code organization and is a very good application to demonstrates its features. Because in this paper our focus is on describing PVR, and not on the actual volume rendering code, we only sketch the implementation to give insight on how to add your own code (for possible computational steering) to PVR and to give you enough information for effective use of PVR's rendering facilities.

6.11.3. Volume Rendering Pipeline

The PVR rendering pipeline is composed of three types of nodes (besides the master, of course). There are the *rendering nodes*, *compositing nodes*, and *collector nodes* (usually just one), (see Figure 26). This specialization is necessary for optimal rendering performance and flexibility. All the clusters work in a simple dataflow mode, where data moves from top to bottom in pipeline fashion. Every cluster has its own fan-in and fan-out number and type of messages. The master configures (and re-configures) the overall dataflow using a set of user-defined and automatic load balancing parameters.

At the top level are the rendering clusters. The nodes in a rendering clusters are responsible for the resampling and shading a given volume dataset. In general the input is a view matrix, and the output is a set of sub-images, each of which is a related to a node in the compositing binary tree. The master can use multiple rendering clusters working on the same image, but disjoint scanlines in order to speed up rendering. Once the subimages are computed, they are passed down in the pipeline to the compositing clusters.

The compositing clusters are organized in a binary tree structure, matching that of the corresponding compositing tree. The number of compositing nodes can actually be different, as we can use *virtualization* to fake more processors than allocated. Images are pipelined down the tree, with every iteration combining the results of compositing until finally all the pixels are a complete depth-ordered sequence. Those pixels are converted to RGB format and sent to the collector node(s) (at this time, we just use a single collector node).

The collector node receives RGB images from the compositing nodes, compresses them using a simple run-length encoding scheme (very fast compression is necessary). Finally the images are either sent over to the *pvrsh* for user viewing (or saving), or locally cached on the disk (it can also be specified that images are trashed for performance analysis purposes).

The previous discussion is over simplistic. There are several performance issues, related to CPU speed, synchronization and memory usage that have not been discussed. For more complete details, we refer the interest reader to Cláudio Silva's Ph.D. dissertation [113].

6.11.4. Rendering with PVR

Figure 27 shows a simple PVR program. Several important features of PVR are demonstrated. In particular, the seamsless integration with Tcl/Tk, the flexible load balancing scheme, and the interactive specification of parameters. The `set` command can have several options (in the Figure 27 they are usually specified in multiple lines, but all can be specified in a single line). For instance, `-imagesz` specifies the size of the images that are outputed by the system.

© The Eurographics Association and Blackwell Publishers 1999.

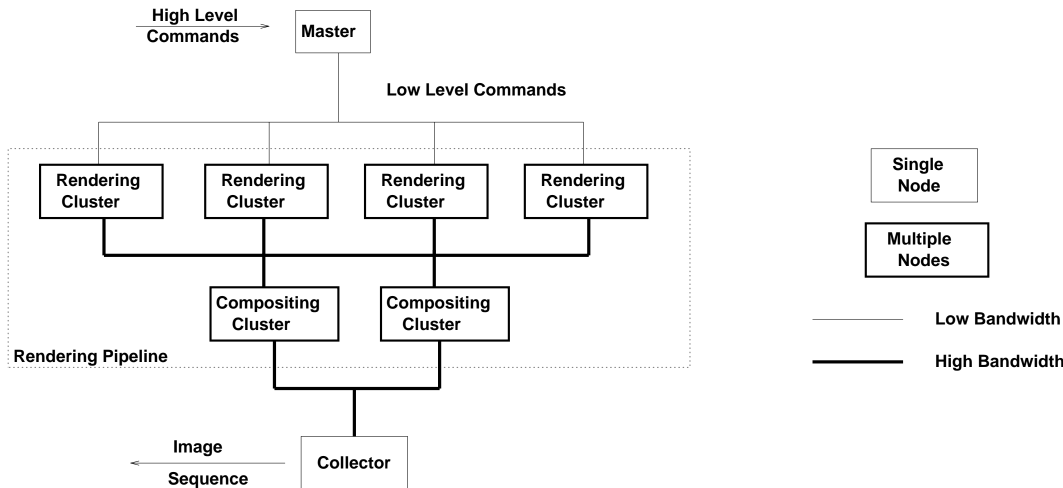

Figure 26: *The host receives high level commands that are translated into virtual* microcode *by the action tables. For rendering, the high level commands are for the generation of animations by rotations and translations, that are translated into simple transformation matrices commands. The rendering clusters perform rendering in parallel. The collector receives and groups images back together and sends an ordered image sequence to the client application.*

The `-cluster` and `-group` options are unique to PVR and its flexible load balacing scheme. With both of these options, the relative sizes of the rendering and compositing clusters can be specified together with the image calculation allocation. Several scalability strategies can be used, for instance, a rendering cluster needs to be large enough to hold the entire dataset and at least a copy of the image to be calculated. By increasing the size of the cluster, the amount of memory per node decreases. By *grouping* clusters (using `-group`), the number of scanlines a given cluster is responsible for decreases, lowering both the memory and the computational cost, thus speeding up image calculation. The same commands can be used to configure compositing clusters. The scalability parameters for compositing clusters is very different than for rendering clusters, because of the different nature of the task. Compositing nodes need memory to hold two copies of the images, what can be quite large (our current parallel machines only have between 16MB to 32MB RAM, this might not be a problem in the near future), and also compositing has very high synchronization cost that grows as the number of nodes grow. Currently, the only need for multiple compositing clusters is due to the need of more memory for large images (such as 1024-by-1024).

6.11.5. DVEs

DVEs can be easily developed by making use of the client/server metaphor. DVE developed using Tcl/Tk are very portable, as Tcl/Tk has ports for almost all the operating systems available, and TCP/IP (our communication protocol) is virtually universal.

Figure 28 shows a simple prototype GUI developed at Sandia. The complete interface is written in Tcl/Tk. The user is able to specify all the necessary rendering parameters in the right window (including image size, transfer function, etc.) and the load balancing parameters in the left window. This simple interface only uses a single session at this time, but more funcionality is being added to the system.

Using the prototype GUI, users are able to add their own funcionality to the system as needed. This flexibility not only makes the system more usable, because redundant *bells and whistles* can be discarted, but also new functionality can be easily added. The use of a portable and well-documented windows interface (e.g., Tk) is imperative. Not only users avoid having to learn yet another programming language and graphical toolkit, but the use of Tk saved us a lot of implementation and documentation cost (Tcl/Tk is widely used and highly documented). Another important feature of Tcl/Tk for the development of prototypes is that it is freely available, enabling us to do the same for PVR.

6.12. Discussion

6.12.1. Related Work

The Shastra project at Purdue has developed tools for distributed and collaborative visualization [8]. Their system implements parallel volume visualization with a mix of image space and object space load balancing, but few details of the scheme are given.

© The Eurographics Association and Blackwell Publishers 1999.

```
toplevel .rgb                              ; Tcl/Tk stuff – creates necessary windows
photo .rgb.p
pack .rgb.p
toplevel .c
canvas .c.c
pack .c.c
source stat.tcl                            ; External command specified in stat.tcl
                                           ; it will place images that get to the session handler in the
                                           ; specified window, and draw a small performance graph
pvr_session :brain                         ; creates a session called "brain"
:brain image window .rgb.p                 ; specifies the window that receives
                                           ; the images
:brain image callback imgCallback          ; specifies the external command
:brain image dir ./                        ; where to place images
:brain open acoma.cs.sandia.gov            ; opens a connection with acoma
                                           ; using the default number of nodes (100)
                                           ; the defaults are in .pvrsh
                                           ; if this command succeeds, we are connected
:brain set -dataset brain.slc              ; specifies the dataset
:brain set -cluster r,16 -group 0,0,1,1    ; 4 rendering clusters of 16 nodes
                                           ; divided into 2 groups, nodes in a group
                                           ; share the same image calculation
:brain set -cluster c -group 0,0           ; 2 compositing clusters of 15 nodes
                                           ; each, this allows for the calculation of very
                                           ; large images, as each cluster will handle half
                                           ; of pixels coming from the rendering nodes
:brain set -imagesz 512,512                ; specifies the image size
:brain render rotation 0,1,0 15,59:60      ; specifies the rendering of
                                           ; 45 images, starting from one quarter rotation
                                           ; along the y axis
:brain set -imagesz 256,256                ; specifies the image size
                                           ; for this image size, one compositing cluster is enough
:brain set -cluster c -group 0
                                           ; re-use the nodes for rendering
:brain set cluster r,16 -group 0,1,2,3,4   ; 5 rendering clusters of 16 nodes
:brain render rotation 1,1,1 0,359:360
```

Figure 27: *A set of PVR rendering commands. The commands can be put in a file and executed in batch, or can be typed interactively on the keyboard (or mixed). Tcl/Tk code (such as "stat.tcl") can be written to take care of portions of the actions.*

They report using up to 4 processors for computation, what makes it hard to evaluate the systems usability in a massively parallel environment. Rowlan et al. [102] describes a distributed volume rendering system implemented on the IBM SP-1. Their system has several of the same characteristics as ours. They also separate rendering and compositing nodes to increase performance and provide a Front-End GUI. Another cousin of our system is DISCOVER [69], developed at National Cheng-Kung University (Taiwan). Their system was customly developed for medical imaging applications and provides mecanisms for the use of remote processor pools.

6.12.2. Visualization Servers

One use of our parallel renderer is as a visualization server for parallel processes [96]. The basic idea is to pre-allocate a set of nodes that can be time-shared by multiple users for visualizing their data. Because of our novel use of pipelining, this can be achieved fairly efficient and with minimal overheard.

Actually, our system architecture is also suitable for time-varying data. When rendering time-varying data, we add a permanent *caching clusters* to the pipeline in Figure 26, that is responsible for distributing the volume data to the rendering nodes

© The Eurographics Association and Blackwell Publishers 1999.

efficiently. The caching cluster is used to hide I/O latency from disk (or other sources). This way the user can visualize his data for as long as a new version comes along. Handling data that changes too rapidly (e.g., faster than we can move it and render) is not possible as it would require large amounts of buffering.

6.12.3. Performance and Results

The current version of PVR is about 25,000 lines of C and Tcl/Tk code. It has been used at Brookhaven National Labs, Sandia National Labs and Stony Brook for the visualization of large datasets. We have demonstrated the capability of rendering a 500,000,000 bytes dataset (the CT visible human data from the National Institute of Health) in approximately 5 seconds/frame. Actually, PVR was demoed in the Sandia booth during Supercomputing '95. Our plans are to use render the full RGB visible human (14,000,000,000 bytes) by the end of the year. (Parallel I/O will be a must, currently the 500MB visible human takes 15 minutes of disk I/O).

6.12.4. Further Development

The idea of developing PVR started out of frustration trying to use network of workstations and the Paragon as rendering engines for VolVis. It was always clear that a pure distributed approach to building rendering environments would be much more powerful than special rendering tools with parallel capabilities. Even though we have completed a *usable* and efficient system, there is still a long wish list, in both the research and development front.

We are currently working on making the system stable enough for large scale availability. With that in mind we are currently working on creating a complete DVE (using VolVis as a starting point) on top of PVR. One of the challenges is how to integrate *resource allocation* and *admission control* in our DVE.

Some functionality is missing from PVR and needs to be incorporated. The most important is probably the support for multiple data sets in a session. This would make the load balancing scheme much more complicated, and simple heuristics might not generate well balanced decomposition schemes. If the volumes are allowed to overlap (as in VolVis), the problem is even harder, and the solution seem to require having special *composite* nodes that perform the sorting at the end of the pipeline. It might be necessary to have a reconfiguration phase each time a new volume is introduced in the picture. It is not clear yet how this can be done efficiently.

A simpler change is to add support in the PVR renderer for non-homogeneous processors. One just needs to change the load balancing scheme slightly by normalizing the number of PARC sub-cubes per processor with their relative performance parameters. Finally, we hope to look in the future for ways to perform real-time manipulation rendering, where the user can just move a mouse and see the picture changing in real-time with minimal lag. For this, we suspect the work done in eliminating virtual reality lag may help.

6.13. Conclusions

In this article we have introduced the PVR system. Here are some of the key features in our system:

- *Transparency* - PVR hides most of the hardware dependencies from the DVEs and the user.
- *Performance* - PVR provides a high speed pipelined ray caster with a unique load balancing scheme and mechanisms to fine tune performance for any given machine configuration.
- *Scalability* - All the algorithms used in the system were carefully choosen to be gracefully scalable. Not only with respect to the machine size, but special care was taken to allow for grown in dataset size and image size.
- *Extensibility* - The PVR architecture can be easily extended, making it easy for the DVE to add new functionality. Also, it is fairly easy for the user to add new funcionality to the PVR shell and its corresponding kernel, allowing for user defined "computational steering" coupled with visualization.

PVR introduces a new level of interactivity to high performance visualization. Larger DVEs can be built on top of PVR, and yet be portable across several architectures. These DVEs that use PVR are given the opportunity to make effective use of available processing power (upto to a few hundred processors), giving a range of cost/performance to end users. This is particularly important in the scientific research community, since most often the question is not *how fast, but how much*. PVR provides a strong foundation for building cost effective DVEs.

As far as the user interface is concerned, PVR introduces a much simpler way to create it. No longer one has to spend time coding in X/MOTIF (or Windows) to create the desired user interface. The Tcl/Tk combination is much simpler, gives more flexibility, and is closely as powerful as the other alternatives. Tcl/Tk is becoming as popular as UNIX shell programming. Different sites should be able to easily create and modify their own systems.

© The Eurographics Association and Blackwell Publishers 1999.

Acknowledgments

We would like to thank Maurice Fan Lok, who as a M.S. student at Stony Brook made fundamental contributions to PVR by co-writing the first version during 1994/95. Brian Wylie deserves special thanks for continual support of the project and for the development of the user interface. We thank Tzi-cker Chiueh, Pat Crossno, Steve Dawson, Juliana Freire, Ron Peierls, and Amitabh Varshney for useful discussions about the PVR system and for help with this paper. The port of PVR to SUNMOS was only possible due to the help of Kevin McCurley, Rolf Riesen, Lance Shuler from Sandia and Edward J. Barragy from Intel. The MRI data set is courtesy of Siemens, Princeton, NJ. C. Silva is partially supported by CNPq-Brazil under a PhD fellowship and by Sandia National Labs. A. Kaufman is partially supported by the National Science Foundation under grants CCR-9205047 and DCA 9303181 and by the Department of Energy under the PICS grant.

Electronic Information

Current information on PVR (including images, animations, related publications, etc.) is kept in http://www.ams.sunysb.edu/~csilva and http://www.cs.sandia.gov/VIS.

Appendix: Parallel Volume Rendering

There are basically three different orthogonal types of parallelism that can be exploited with volume rendering. In the PVR system we exploit all of them:

6.13.0.1. Image Space Parallelism. In image space parallelism parts of a single image are divided into multiple processors for concurrent image generation. For volume rendering (and in general for computer graphics) this is the simplest form of parallelism, as no subdivision of the volume itself need to be performed and the volume never needs to be updated. This is usually the type of parallelism implemented on shared memory machines [117]. The main shortcoming of this method (for distributed memory machines) is that large datasets can not be rendered using this type of parallelism alone.

6.13.0.2. Object Space Parallelism. In object space parallelism parts of the volume are divided into multiple processors, each computes a sub-image, that is later regrouped. The main shortcomings are the higher need for communication and synchronization among the processors (parallel machines still have slow communications with respect to processor speed). Ma et al. [73] describes an efficient algorithm for re-grouping the images back together to form a single correct image. In statically partitioning the volume dataset, one has to be careful to give every processor the same amount of work (see [114].

6.13.0.3. Time Space Parallelism. In time space parallelism, multiple images are computed at the same time. This exploits the fact that the rendering process can be easily divided into multiple disjoint phases.

Acknowledgments

Numerous individuals have contributed to all parts of the material presented here.

We want to thank the members of the Personal Workstation team at IBM Research: Randy Moulic, Nick Dono, Dan Dumarot, Cliff Pickover, Del Smith, and Dave Stevenson.

Furthermore, we would like to thank Michael Doggett and Michael Meißner of the Computer Graphics Lab at the University of Tübingen for proof readings and good suggestions to improve the text on parallel architectures and parallel programming.

For the parts on parallel volume rendering, Claudio is deeply indebted to A.E. Kaufman, J.S.B. Mitchell, C. Pavlakos, C.R. Ramakrishnan, who co-authored the original research he is briefly reporting in this tutorial. Most of this research was supported by CNPq-Brazil under a Ph.D. fellowship, by Sandia National Labs and the Department of Energy, and by the National Science Foundation (NSF) postdoctoral grant CDA-9626370.

© The Eurographics Association and Blackwell Publishers 1999.

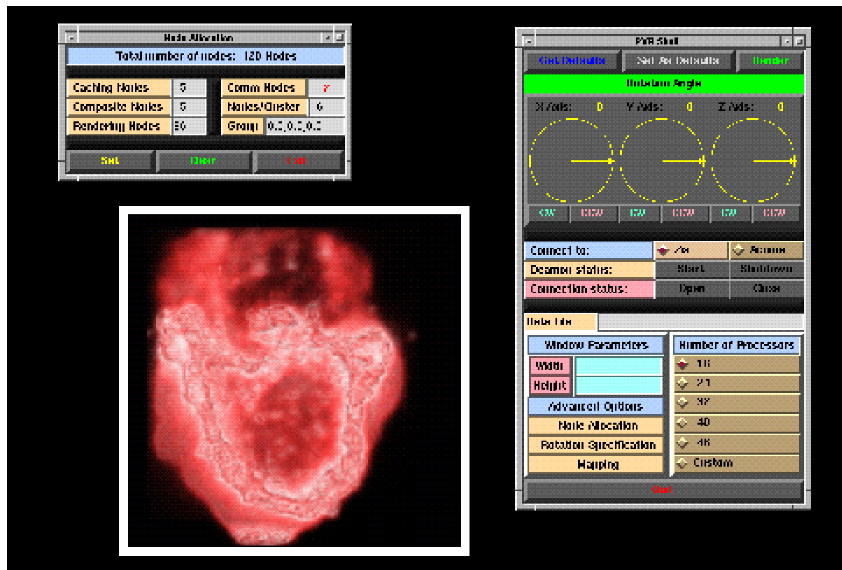

Figure 28: *A snapshot of Brian Wylie's user interface developed at Sandia National Labs. There are three windows. On the right is the main interface window, where the user can specify general rotations. On the left, the cluster configuration window. And on the bottom an image of a cell calculated with PVR.*

© The Eurographics Association and Blackwell Publishers 1999.

Appendix

Appendix A: Literature and Internet Resources on Parallel Programming

Books

- **D. Butenhof - Programming with POSIX Threads**, Addison-Wesley, 1997.
  The pthread part of this tutorial is based on this book. Besides being a good introduction into threading, it offers many details and knowledge of how to use threads.
- **S. Kleiman, D. Shah, B. Smaalders - Programming with threads**, Prentice Hall, 1995. Covers POSIX threads.
- **B. Lewis, D. Berg - Threads Primer: A guide to multithreaded programming**, Prentice Hall, 1995. Covers UI, POSIX, OS/2, and WIN32 threads.
- **B. Nichols, D. Buttlar, J. Farrel - Pthread programming**, O'Reilly, 1996. Covers POSIX threads.
- **S. Norton, M. Depasquale, M. Dipasquale - Thread Time: The Multithreaded Programming Guide**, Prentice Hall, 1997. Covers POSIX threads.
- **A. Geist, A. Beguelin, J. Dongarra, W. Jian, R. Machek, and V. Sunderam - PVM: Parallel Virtual Machine**, MIT Press, 1994. Covers PVM3.

Webpages

- **www.openmp.org** - OpenMP home page
- **www.lambdacs.com/newsgroup/FAQ.html** - thread FAQ list
- **www.best.com/ bos/threads-faq/** - thread FAQ list
- **cseng.awl.com/bookdetail.qry?ISBN=0-201-63392-2&ptype=1482** - Additional information on Dave Butenhof book (e.g. source code)
- **liinwww.ira.uka.de/bibliography/Os/threads.html** - thread bibliography
- **science.nas.nasa.gov/Software/NPB/** - results of parallel benchmarks
- **www.sun.com/workshop/threads/** - SUN's thread pages
- **www.mit.edu:8001/afs/sipb/user/proven/XMOSAIC/ pthreads.html** - pthread pages at MIT
- **www.netlib.org/{pvm3,mpi}** - a source for documents and packages for MPI and PVM.
- **elib.zib.de** - also a source for documents, packages, and more.
- **www.erc.msstate.edu/mpi/** - MPI home page at Mississippi State.
- **www.mcs.anl.gov/mpi/index.html** - MPI home page at Argonne National Lab.
- **www.epm.ornl.gov/pvm/pvm_home.html** - PVM home page at Oak Ridge National Lab.

Newsgroups

- **comp.parallel** - general newsgroup on parallel stuff
- **comp.parallel.pvm** - newsgroup on PVM
- **comp.parallel.mpi** - newsgroup on MPI
- **comp.programming.threads** - newsgroup on threading
- **comp.sys.sgi.bugs** - newsgroups for threading problems on SGI workstations
- **comp.sys.sgi.{graphics, hardware, misc}** - if the previous newsgroup does not help....

© The Eurographics Association and Blackwell Publishers 1999.

Appendix B: Tiny Thread Glossary

Barrier - All participating execution entities are synchronizing at a particular point within the parallel application. This point is called a barrier.

Cache-coherent - Modern processors use caches to speed-up memory access. On multi-processor systems this can result in different views of memory content for the individual threads. If a system is cache-coherent, special communication protocols ensure the same memory view. This system is called cache-coherent.

Concurrency - Parallel execution of programs. This parallel execution can be either time-sliced (on single processor machines), or really parallel on multi-processors.

Condition - A signaling mechanism to indicate the state of a shared resource.

Kernel thread - A kernel is a exectution entity which is scheduled by the operating system kernel (one-to-one mapping).

Light-weight process - Physical scheduling entity of an operating system. On some systems, scheduled threads are mapped on light-weight-processes for execution. On Sun/Solaris systems, the kernel threads are called light-weight processes.

Message-Passing - Execution entities communicate by sending message exchanged via a interconnection network.

Mixed-model scheduling - Scheduling model inbetween user and kernel threads. Some scheduling tasks are performed by the thread library, some by the operating system kernel (many-to-few mapping).

Mutex - Synchronization mechanism for mutal exclusion of critical sections in a parallel program.

NUMA - Non-Uniform Memory Access - Main memory is distributed to the different hierarchy stages. Therefore, the memory access times are varying, depending on the processor and the physical location of the memory address.

Oversubscribing - More threads than processors are started. This is only efficiently possible with mixed-model scheduling or with user threads.

Preemption - A process, or a thread is disabled from execution (preempted), because the scheduling algorithm decided that another process/thread is more important than the current.

Process - An execution entity, containing a whole execution context (private address space, program counter, etc.)

Pthread - Thread standard.

Recycle thread - After performing its task, a new task is assigned to the thread; the thread is recycled.

Scheduling - Decicsion which execution entity can use particular resources (e.g. periphery devices, processors, etc).

Semaphore - Synchronization mechanism similar to mutexes. In contrast to binary mutexes, semaphores can have more than two states (they are “counting”).

Shared-memory Paradigm - Execution entities communicate via memory which is accessible from all entities.

Synchronization - If a shared resources is needed by different threads, their access must be handled consistantly. The threads need to agree on an order of access to the resource.

Thread - Control flow entity within a process. Threads of the same process share parts of the execution context (such as address space). Therefore, context switching, creation and destruction of a thread is much faster than for a process.

UMA - Uniform Memory Access - Access times to main memory are the same for all processors in a system.

User thread - Execution entity of a thread library. The library itself is scheduling the user thread (many-to-one mapping).

References

1. Serial storage architecture: A technology overview, version 3.0. San Jose, August 1995.
2. Highly-parallel system architecture vs. the intel 440lx architecture in the workstation market, October 1997. http://www.compaq.com/support/techpub/whitepaper/ecg0491097.html.
3. Accelerated graphics port interface specification, May 1998. http://www.intel.com/pc-supp/platform/agfxport.
4. G. Abram and H. Fuchs. Vlsi architectures for computer graphics. In G. Enderle, editor, *Advances in Computer Graphics I*, pages 6–21, Berlin, Heidelberg, New York, Tokyo, 1986. Springer-Verlag.

© The Eurographics Association and Blackwell Publishers 1999.

5. A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers — Principles, Techniques, and Tools*. Addison Wesley, 1988.
6. K. Akeley. Realityengine graphics. In *Computer Graphics (Proc. Siggraph)*, pages 109–116, August 1993.
7. K. Akeley and T. Jermoluk. High-performance polygon rendering. *Computer Graphics (Proc. Siggraph)*, 22(4):239–246, August 1988.
8. V. Anupam, C. Bajaj, D. Schikoer, and M. Schikore. Distributed and collaborative visualization. *IEEE Computer*, 27(7):37–43, 1994.
9. R. Avila, L. Sobierajski, and A. Kaufman. Towards a comprehensive volume visualization system. In *IEEE Visualization '92*, pages 13–20. IEEE CS Press, 1992.
10. F. Bellosa, M. Gente, and C. Koppe. Vorlesung Programmierung Paralleler Systeme. Technical Report IMMD-IV, Computer Science Department, University of Erlangen-Nuremberg, 1995.
11. J. F. Blinn. Light reflection functions for simulation of clouds and dusty surfaces. In *Computer Graphics (SIGGRAPH '82 Proceedings)*, pages 21–29, July 1982.
12. W. Blochinger, W. Küchlin, and A. Weber. The Distributed Object-Oriented Threads System dots. In *Fifth International Symposium on Solving Irregularly Structured Problems in Parallel (IRREGULAR '98)*, volume LNCS 1457 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.
13. Architectural Review Board. *OpenGL Reference Manual*. Addison-Wesley, 1992.
14. OpenMP Architecture Review Board. OpenMP API. www.openmp.org/index.cgi?specs.
15. OpenMP Architecture Review Board. OpenMP FAQ. www.openmp.org/index.cgi?faq.
16. D. Butenhof. *Programming with POSIX Threads*. Addison Wesley, Reading, Mass., 1st edition, 1997.
17. E. Camahort and I. Chakravarty. Integrating volume data analysis and rendering on distributed memory architectures. In *1993 Parallel Rendering Symposium Proceedings*, pages 89–96. ACM Press, October 1993.
18. Ingrid Carlbom. Optimal filter design for volume reconstruction and visualization. In *IEEE Visualization '93*, pages 54–61, 1993.
19. L. Carpenter. The a-buffer, an antialiased hidden surface method. *Computer Graphics (Proc. Siggraph)*, 18(3):125–138, 1985.
20. J. Challinger. Scalable parallel volume raycasting for nonrectilinear computational grids. In *1993 Parallel Rendering Symposium Proceedings*, pages 81–88, 1993.
21. Earl Coddington. *An Introduction to Ordinary Differential Equations*. Prentice-Hall, 1961.
22. M. Cox and P. Hanrahan. Depth complexity in object-parallel graphics architectures. In *Proc. 7th Eurographics Workshop on Graphics Hardware*, pages 204–222, Cambridge (UK), 1992.
23. Roger Crawfis and Nelson Max. Direct volume visualization of three-dimensional vector fields. *1992 Workshop on Volume Visualization*, pages 55–60, 1992.
24. T. W. Crockett. Parallel rendering. In A. Kent and J. G. Williams, editors, *Encyclopedia of Computer Science and Technology*, volume 34, Supp. 19, A., pages 335–371. Marcel Dekker, 1996. Also available as ICASE Report No. 95-31 (NASA CR-195080), 1996.
25. T.W. Crockett and T. Orloff. A parallel rendering algorithm for mimd architectures. Technical Report ICASE-Report 91-3, Institute for Computer Science and Engineering, NASA Langley Research Center, 1991.
26. John Danskin and Pat Hanrahan. Fast algorithms for volume ray tracing. *1992 Workshop on Volume Visualization*, pages 91–98, 1992.
27. M. Deering and S.R. Nelson. Leo: A system for cost effective 3d shaded graphics. In *Computer Graphics (Proc. Siggraph)*, pages 101–108, August 1993.
28. S. Demetrescu. High-speed image rasterization using scan line access memories. In H. Fuchs, editor, *Proc. Chapel Hill Conference on VLSI*, pages 221–243, 1985.
29. Robert A. Drebin, Loren Carpenter, and Pat Hanrahan. Volume rendering. In John Dill, editor, *Computer Graphics (SIGGRAPH '88 Proceedings)*, volume 22, pages 65–74, August 1988.

© The Eurographics Association and Blackwell Publishers 1999.

30. R. Eigenmann, B. Kuhn, T. Mattson, and R. Menon. Introduction to OpenMP. In *SuperComputing 1998*, 1998.
31. D. Ellsworth. A new algorithm for interactive graphics on multicomputers. *IEEE Computer Graphics & Applications*, pages 33–40, July 1994.
32. A. Barkans et al. Guardband clipping method and apparatus for 3d graphics display system. U.S. Patent 4,888,712. Issued Dec 19, 1989.
33. D. Clark et al. An analysis of tcp processing overhead. *IEEE Communications Magazine*, pages 23–29, June 1989.
34. H. Fuchs et al. Pixel-planes 5: A heterogeneous multiprocessor graphics system using processor-enhanced memories. *Computer Graphics (Proc. Siggraph)*, 23(3):79–88, July 1989.
35. I. Sutherland et al. A characterization of ten hidden surface algorithms. *ACM Computing Surveys*, 6(1):1–55, March 1974.
36. J. Eyles et al. Pixelflow: The realization. In *Proc. 1997 Siggraph/Eurographic Workshop on Graphics Hardware*, pages 57–68, New York, 1997. ACM Press.
37. J. Foley et al. *Computer Graphics: Principles and Practice*. Addison Wesley, 2nd edition, 1990.
38. M. Deering et al. The triangle processor and normal vector shader: A vlsi system for high-performance graphics. *Computer Graphics (Proc. Siggraph)*, 12(2):21–30, August 1988.
39. N. Boden et al. Myrinet: A gigabit-per-second local-area network. *IEEE Micro*, pages 29–36, February 1995.
40. S. Molnar et al. A sorting classification of parallel rendering. *IEEE Computer Graphics & Applications*, pages 23–32, July 1994.
41. James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics, Principles and Practice, Second Edition*. Addison-Wesley, Reading, Massachusetts, 1990. Overview of research to date.
42. MPI Forum. Mpi: A message-passing interface standard. *International Journal of Supercomputer Applications*, 8(3/4):165–416, 1994.
43. MPI Forum. MPI-2: Extensions to the Message-Passing Interface. Technical Report MPI 7/18/97, Message-Passing Interface Forum, 1997.
44. H. Fuchs, Z. M. Kedem, and B. F. Naylor. On visible surface generation by a priori tree structures. In *Computer Graphics (SIGGRAPH '80 Proceedings)*, pages 124–133, July 1980.
45. A. Geist, A. Beguelin, J. Dongarra, W. Jian, R. Machek, and V. Sunderam. *PVM: Parallel Virtual Machine*. MIT Press, 1994.
46. G. Geist, J. Kohl, and P. Papdopoulos. PVM and MPI: A Comparison of Features. In *Calculateurs Paralleles*, volume 8(2), 1996.
47. C. Giertsen and J. Petersen. Parallel volume rendering on a network of workstations. *IEEE Computer Graphics and Applications*, 13(6):16–23, 1993.
48. Christopher Giertsen. Volume visualization of sparse irregular meshes. *IEEE Computer Graphics and Applications*, 12(2):40–48, March 1992.
49. A. Glassner, editor. *An Introduction to Ray Tracing*. Academic Press, 1989.
50. Andrew Glassner. *Principles of Digital Image Synthesis (2 Vols)*. Morgan Kaufmann Publishers, Inc. ISBN 1-55860-276-3, San Francisco, CA, 1995.
51. Heinrich Müller and Michael Stark. Adaptive generation of surfaces in volume data. *The Visual Computer*, 9(4):182–199, January 1993.
52. B. Hendrickson and R. Leland. The chaco user's guide (version 1.0). Tech. Rep. SAND93-2339, Sandia National Laboratories, Albuquerque, N.M., 1993.
53. J. Hennesy and D. Paterson. *Computer Architecture: A Quantitative Approach*. Morgan-Kaufmann, 1990.
54. D. Hillis. *The Connection Machine*. MIT Press, 1985.
55. W. Hsu. Segmented ray casting for data parallel volume rendering. In *1993 Parallel Rendering Symposium Proceedings*, pages 7–14. ACM Press, October 1993.

© The Eurographics Association and Blackwell Publishers 1999.

56. M. Hu and J. Foley. Parallel processing approaches to hidden-surface removal in image space. *Computers & Graphics*, 9(3):303–317, 1985.

57. J. Hubbell. Network rendering. In *Autodesk University Sourcebook Vol. 2*, pages 443–453. Miller Freeman, 1996.

58. James T. Kajiya. The rendering equation. In David C. Evans and Russell J. Athay, editors, *Computer Graphics (SIGGRAPH '86 Proceedings)*, volume 20, pages 143–150, August 1986.

59. James T. Kajiya and Brian P. Von Herzen. Ray tracing volume densities. In Hank Christiansen, editor, *Computer Graphics (SIGGRAPH '84 Proceedings)*, volume 18, pages 165–174, July 1984.

60. Arie E. Kaufman. *Volume Visualization*. IEEE Computer Society Press, ISBN 908186-9020-8, Los Alamitos, CA, 1990.

61. S. Kleiman, D. Shah, and B. Smaalders. *Programming With Threads*. Prentice Hall, 1995.

62. M. Kumanoya. Trends in high-speed dram architectures. *IEICE Trans. Electron.*, E79-C(4), April 19.

63. Philippe Lacroute and Marc Levoy. Fast volume rendering using a shear–warp factorization of the viewing transformation. In Andrew Glassner, editor, *Proceedings of SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994)*, Computer Graphics Proceedings, Annual Conference Series, pages 451–458. ACM SIGGRAPH, ACM Press, July 1994. ISBN 0-89791-667-0.

64. T. Lee, C. Raghavendra, and J. Nicholas. Image composition methods for sort-last polygon rendering on 2d mesh architectures. In *IEEE/ACM Parallel Rendering Symposium '95*, pages 55–62, 1995.

65. Marc Levoy. Display of surfaces from volume data. *IEEE Computer Graphics and Applications*, 8(3):29–37, May 1988.

66. Marc Levoy. Efficient ray tracing of volume data. *ACM Transactions on Graphics*, 9(3):245–261, July 1990.

67. Marc Levoy. Volume rendering by adaptive refinement. *The Visual Computer*, 6(1):2–7, February 1990.

68. B. Lewis and D. J. Berg. *Threads Primer: A Guide to Multithreaded Programming*. Prentice Hall, 1996.

69. P.-W. Liu, L.-S. Chen, S.-C. Chen, J.-P. Chen, F.-Y. Lin, and S.-S. Hwang. Distributed computing: New power for scientific visualization. *IEEE Computer Graphics and Application*, 16(3):42–51, 1996.

70. William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. In Maureen C. Stone, editor, *Computer Graphics (SIGGRAPH '87 Proceedings)*, volume 21, pages 163–169, July 1987.

71. K. Ma, J. Painter, C. Hansen, and M. Krogh. A data distributed parallel algorithm for ray-traced volume rendering. In *1993 Parallel Rendering Symposium Proceedings*, pages 15–22. ACM Press, October 1993.

72. K. Ma, J. Painter, C. Hansen, and M. Krogh. Parallel volume rendering using binary-swap compositing. *IEEE Computer Graphics and Applications*, 14(4):59–68, 1994.

73. K. Ma, J. Painter, C. Hansen, and M. Krogh. Parallel volume rendering using binary-swap compositing. *IEEE Computer Graphics and Applications*, 14(4):59–68, 1994.

74. Kwan-Liu Ma. Parallel volume rendering for unstructured-grid data on distributed memory machines. In *IEEE/ACM Parallel Rendering Symposium '95*, pages 23–30, 1995.

75. A. Maccabe, K. McCurley, R. Riesen, and S. Wheat. Sunmos for the Intel Paragon - A Brief User's Guide. In *Proceedings of the Intel Supercomputer Users' Group 1993 Annual North America Users' Conference*, 1993.

76. S. R. Marschner and R. J. Lobb. An evaluation of reconstruction filters for volume rendering. In *IEEE Visualization '94*, pages 100–107, 1994.

77. Nelson Max. Optical models for direct volume rendering. *IEEE Transations on Visualization and Computer Graphics*, 1(2):99–108, June 1995.

78. Nelson Max, Roger Crawfis, and Barry Becker. New techniques in 3D scalar and vector field visualization. In *First Pacific Conference on Computer Graphics and Applications*. Korean Information Science Society, Korean Computer Graphics Society, August 1993.

79. Nelson Max, Pat Hanrahan, and Roger Crawfis. Area and volume coherence for efficient visualization of 3D scalar functions. In *Computer Graphics (San Diego Workshop on Volume Visualization)*, pages 27–33, November 1990.

80. Nelson L. Max. Efficient light propagation for multiple anisotropic volume scattering. In *Fifth Eurographics Workshop on Rendering*, pages 87–104, Darmstadt, Germany, June 1994.

© The Eurographics Association and Blackwell Publishers 1999.

81. James V. Miller, David E. Breen, William E. Lorensen, Robert M. O'Bara, and Michael J. Wozny. Geometrically deformed models: A method for extracting closed geometric models from volume data. In Thomas W. Sederberg, editor, *Computer Graphics (SIGGRAPH '91 Proceedings)*, volume 25, pages 217–226, July 1991.

82. S. Molnar. Combining Z-buffer engines for higher-speed rendering. In *Advances in Computer Graphics Hardware III*, pages 171–182, 1988.

83. S. Molnar. *Image Composition Architectures for Real-Time Image Generation*. Ph.D. thesis, University of North Carolina, Chappel Hill, 1991.

84. S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs. A sorting classification for parallel rendering. *IEEE Computer Graphics and Applications*, 14(4):23–32, 1994.

85. C. Montani, R. Perego, and R. Scopigno. Parallel volume visualization on a hypercube architecture. In *1992 Workshop on Volume Visualization Proceedings*, pages 9–16. ACM Press, October 1992.

86. J. Montrym. Infinite reality: A real-time graphics system. In *Computer Graphics (Proc. Siggraph)*, pages 293–302, August 1997.

87. C. Mueller. The sort-first rendering architecture for high-performance graphics. In *Proc. 1995 Symposium on Interactive 3D Graphics*, pages 75–84, New York, 1995. ACM Press.

88. Henry Neeman. A decomposition algorithm for visualizing irregular grids. In *Computer Graphics (San Diego Workshop on Volume Visualization)*, pages 49–56, November 1990.

89. U. Neumann. Parallel volume-rendering algorithm performance on mesh-connected multicomputers. In *1993 Parallel Rendering Symposium Proceedings*, pages 97–104. ACM Press, October 1993.

90. B. Nichols, D. Buttlar, and J. Proulx Farrel. *Pthreads Programming*. O'Reilly & Associates, Inc., Sebastopol, 1st edition, 1996.

91. J. Nieh and M. Levoy. Volume rendering on scalable shared-memory mimd architectures. In *1992 Workshop on Volume Visualization Proceedings*, pages 17–24. ACM Press, October 1992.

92. Gregory M. Nielson and Bernd Hamann. The asymptotic decider: Removing the ambiguity in marching cubes. In *Visualization '91*, pages 83–91, 1991.

93. S. Norton, M. Depasquale, and M. Dipasquale. *Thread Time: The MultithreadedProgramming Guide*. Prentice Hall, 1997.

94. M. Olano and T. Greer. Triangle scan conversion using 2d homogeneous coordinates. In *Proc. 1997 Siggraph/Eurographic Workshop on Graphics Hardware*, pages 89–96, New York, 1997. ACM Press.

95. J. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1993.

96. C. Pavlakos, L. Schoof, and J. Mareda. A visualization model for supercomputing environments. *IEEE Parallel & Distributed Technology*, 1(4):16–2, 1996.

97. J. Pineda. A parallel algorithm for polygon rasterization. *Computer Graphics (Proc. Siggraph)*, 22(4):17–20, August 1988.

98. Thomas Porter and Tom Duff. Compositing digital images. In Hank Christiansen, editor, *Computer Graphics (SIGGRAPH '84 Proceedings)*, volume 18, pages 253–259, July 1984.

99. F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, New York, NY, 1985.

100. C.R. Ramakrishnan and C.T. Silva. Optimal processor allocation for sort-last compositing under bsp-tree ordering, submitted for publication, 1997.

101. D. Roble. A load balanced parallel scanline z-buffer algorithm for the ipsc hypercube. In *Proc. Pixim '88*, pages 177–192, Paris (France), October 1988.

102. J. Rowlan, E. Lent, N. Gokhale, and S. Bradshaw. A distributed, parallel, interactive volume rendering package. In *IEEE Visualization '94*, pages 21–30, 1994.

103. Paolo Sabella. A rendering algorithm for visualizing 3D scalar fields. In John Dill, editor, *Computer Graphics (SIGGRAPH '88 Proceedings)*, volume 22, pages 51–58, August 1988.

104. Hanan Samet. *Applications of Spatial Data Structures*. Addison-Wesley, Reading, Massachusetts, 1990.

© The Eurographics Association and Blackwell Publishers 1999.

105. J. Schechter. Unix workstations stand their ground. *Computer Graphics World*, pages 36–46, October 1997.

106. B.-O. Schneider. A processor for an object-oriented renderin system. *Computer Graphics Forum*, 7:301–310, 1988.

107. B.-O. Schneider. Parallel rendering on pc workstations. In *Proc. 1998 International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, July 1998.

108. B.-O. Schneider and J. van Welzen. Efficient polygon clipping for a simd graphics pipeline. *IEEE Transactions on Visualization and Computer Graphics*, 4(3), July 1998. To appear.

109. P. Schroeder and J. Salem. Fast rotation of volume data on data parallel architectures. In *Visualization '91 Proceedings*, pages 50–57. IEEE CS Press, 1991.

110. William J. Schroeder, Jonathan A. Zarge, and William E. Lorensen. Decimation of triangle meshes. In Edwin E. Catmull, editor, *Computer Graphics (SIGGRAPH '92 Proceedings)*, volume 26, pages 65–70, July 1992.

111. R. Sethi and J. Ullman. The generation of optimal code for arithmetic expressions. *Journal of the ACM*, 17(4), 1970.

112. Peter Shirley and Allan Tuchman. A polygonal approximation to direct scalar volume rendering. In *Computer Graphics (San Diego Workshop on Volume Visualization)*, pages 63–70, November 1990.

113. C. Silva. *Parallel Volume Rendering of Irregular Grids*. Ph.D. thesis, State University of New York at Stony Brook, 1996.

114. C. Silva and A. Kaufman. Parallel performance measures for volume ray casting. In *IEEE Visualization '94*, pages 196–203, 1994.

115. C. Silva, A. Kaufman, and C. Pavlakos. PVR: High-Performance Volume Rendering. In *IEEE Computational Science and Engineering*, 1996.

116. C.T. Silva and J.S.B. Mitchell. The lazy sweep ray casting algorithm for rendering irregular grids. *IEEE Transations on Visualization and Computer Graphics*, 3(2), 1997.

117. J. P. Singh, A. Gupta, and M. Levoy. Parallel visualization algorithms: Performance and architectural implications. *IEEE Computer*, 27(7):45–55, 1994.

118. L. Sobierajski and R. Avila. A hardware acceleration method for volume ray tracing. In *IEEE Visualization '95*. IEEE CS Press, 1995.

119. Lisa Sobierajski and Arie Kaufman. Volumetric ray tracing. In Arie Kaufman and Wolfgang Krueger, editors, *1994 Symposium on Volume Visualization*, pages 11–18. ACM SIGGRAPH, October 1994. ISBN 0-89791-741-3.

120. Don Speray and Steve Kennon. Volume probes: Interactive data exploration on arbitrary grids. In *Computer Graphics (San Diego Workshop on Volume Visualization)*, pages 5–12, November 1990.

121. M. Steckermeier and F. Bellosa. Using Locality Information in Userlevel Scheduling. Technical Report TR-14-95-14, Computer Science Department, University of Erlangen-Nuremberg, 1995.

122. Craig Upson and Michael Keeler. VBUFFER: Visible volume rendering. In John Dill, editor, *Computer Graphics (SIGGRAPH '88 Proceedings)*, volume 22, pages 59–64, August 1988.

123. Sam Uselton. Volume rendering for computational fluid dynamics: Initial results. Tech Report RNR-91-026, Nasa Ames Research Center, 1991.

124. R. Weinberg. Parallel processing image synthesis with anti-aliasing. *Computer Graphics (Proc. Siggraph)*, 15(3):55–62, August 1981.

125. Lee Westover. Footprint evaluation for volume rendering. In Forest Baskett, editor, *Computer Graphics (SIGGRAPH '90 Proceedings)*, volume 24, pages 367–376, August 1990.

126. D. Whelan. A rectangular area filling display system architecture. *Computer Graphics (Proc. Siggraph)*, 16(3):147–153, July 1982.

127. S. Whitman. *Multiprocessor Methods for Computer Graphics Rendering*. Jones and Bartlett, Boston, London, 1992.

128. S. Whitman. Dynamic load balancing for parallel polygon rendering. *IEEE Computer Graphics & Applications*, pages 41–48, July 1994.

129. S. R. Whitman. A survey of parallel algorithms for graphics and visualization. In International Workshop on High-Performance Computing for Computer Graphics and Visualization, Swansea, United Kingdom, 1995.

© The Eurographics Association and Blackwell Publishers 1999.

130. Jane Wilhelms and Judy Challinger. Direct volume rendering of curvilinear volumes. In *Computer Graphics (San Diego Workshop on Volume Visualization)*, pages 41–47, November 1990.

131. Jane Wilhelms and Allen Van Gelder. A coherent projection approach for direct volume rendering. In Thomas W. Sederberg, editor, *Computer Graphics (SIGGRAPH '91 Proceedings)*, volume 25, pages 275–284, July 1991.

132. Peter L. Williams and Nelson Max. A volume density optical model. *1992 Workshop on Volume Visualization*, pages 61–68, 1992.

133. R. Yagel, D. Cohen, and A. Kaufman. Discrete ray tracing. *IEEE Computer Graphics and Applications*, pages 19–28, 1992.

134. K. Zuiderveld. *Visualization of Multimodality Medical Volume Data using Object-Oriented Methods*. PhD thesis, University of Utrecht, The Netherlands, 1995.

135. K. Zuiderveld, A. Koning, and M. Viergever. Acceleration of ray-casting using 3D distance transforms. In *Visualization in Biomedical Computing '92*, pages 324–335. SPIE, 1992.

© The Eurographics Association and Blackwell Publishers 1999.