machine rather than simulated, it is also possible to determine quantitatively how well each implementation has succeeded.

2.3. Conclusions

In the first section of this chapter, a number of criteria are given to evaluate the different parallel decompositions which are presented in the second section. These criteria for evaluation include granularity, type of parallelism, use of coherence, load balancing characteristics, methods of data access, and scalability.

In the second section, a number of past as well as yet untested possible parallel approaches are presented and categorized into a taxonomy. The image space pixel decompositions based on areas of pixels seem to hold the most promise for high performance on an MIMD architecture. In chapter 5, the implementations of approaches are described in an effort to conclusively show that one technique is optimal. Since most of the past work has involved simulations and not multiprocessor implementations, these implementations allow us to compare different task decompositions and memory referencing strategies on an equal basis for the first time.

3

Issues in Parallel Algorithm Development

In this chapter, different advanced parallel computer architectures are compared according to their suitability for implementation of a graphics display algorithm.

In the first section, the architectures are presented and analyzed with regard to the development of a parallel graphics rendering algorithm. Although SIMD architectures (single instruction, multiple data path) have been used for graphics applications in the past, this mode of operation generally requires task execution in lock step fashion. Work done at the pixel level can be accomplished in this manner, but higher level tasks are not well suited to this type of parallel approach.

The type of architectures investigated here is restricted to MIMD machines (multiple instruction, multiple data path) since different tasks can proceed simultaneously under separate control flow. These machines can be configured with small to large processor counts, allowing flexibility in performance versus cost. An inexpensive MIMD architecture can be obtained for as little as $10,000, while

extremely high performance machines can cost several million dollars. The algorithms presented here are designed to be useful on a small system containing only 2 processors, as well as a large system of 100 or more processors.

The second section involves a comparison of the two main architectural choices in MIMD hardware. These two methods, message passing and shared memory, are analyzed with regard to a parallel graphics implementation. Finally, we describe the programming environment which is specific to the BBN Butterfly multiprocessor since this machine was chosen for the implementation comparisons here.

3.1. Architectural Choices

There are currently a number of commercial message passing architectures (Intel iPSC, NCube, Inmos transputer based machines) and shared memory multiprocessors (Sequent Balance, Encore Multimax, Alliant FX/2800, and BBN Butterfly[1]) on the market. Coarse grained MIMD architectures such as those offered by Cray, Convex, Silicon Graphics, and others allow only limited parallelism. The issues in developing programs for the latter machines are not as pronounced as for the previously mentioned machines due to their small processor counts (typically 2 to 8 processors).

In this section, we describe a few specific commercial machines to illustrate the differences among the classes of architectures. These differences allow us to evaluate how well the various types of computer image generation algorithms can be expected to run on a variety of parallel computers.

The first subsection describes the impact of using conventional MIMD hardware to perform computer graphics rendering. Different issues by which the architectures are evaluated are given in this section. The amount of memory used in a graphics application, as well as the high data movement involved in this application, influences the choice for the appropriate architecture suitable for this type of application.

Distributed memory architectures are useful for applications where the data can be partitioned initially among the nodes[2] of the system with little communication thereafter. These types of machines typically have a high message passing cost, and any time spent

---

[1]BBN Advanced Computers, Inc. is no longer marketing the Butterfly, although it is still being supported.

[2]The terms *processors* and *nodes* are used interchangeably here.

communicating is time wasted from computation. The Intel hypercube family of machines are examples of distributed memory architectures in which the processors are connected in a hypercube topology. Although hypercube connections are a common design, Intel is also experimenting with a mesh design on what is currently reported to be the fastest computer in the world: the prototype Intel Touchstone machine installed at California Institute of Technology.

The BBN Butterfly contains physically distributed memory, but it is classified as a shared memory architecture since remote memory can be logically shared. The Encore Multimax is an example of global shared memory architecture which uses caching to speed up references to the memory modules. These machines are analyzed as representative examples of their genre in the second subsection.

3.1.1. Impact of Graphics Rendering on System Requirements

Utilizing a multiprocessor for an application such as computer graphics rendering imposes certain demands on the system that other applications might not introduce. The large amount of data movement in this type of algorithm presents unusual problems for certain types of architectures. Following are two characteristics of graphics display algorithms which can affect the performance of the computer architecture to be used for implementation of the software algorithm.

3.1.1.1. Image Quality

In this book, we are primarily interested in trying to achieve good performance when rendering highly complex images in a graphics display algorithm on a parallel processor. This increase in image complexity can arise from several factors given in [Whit89]:

Anti-aliasing. This is a correction mechanism for the typically inadequate sampling of high frequencies in a computer generated scene. More information about the geometric structure of the scene must be available to do anti-aliasing, and this affects the size of the data structures and the amount of information which must be shared by each processor. In general, polygon fragment or sub-pixel information must be stored to perform anti-aliasing.

Mapping. The data structures required for texture, bump, and reflection mapping are very large and must be shared by a large number of processors, which increases the communication between processors. These mapping operations enhance the quality of the image by simulating different types of surface attributes for the objects in the scene.

Shadows. Some shadow casting algorithms require data structures as large as those required for texture, bump, and reflection mapping, with the same resultant communication problems. If a Z-buffer shadow algorithm is used, the visibility calculations are repeated for each light source and this adds time complexity [Will78]. The shadow volumes technique requires additional geometric data and this adds to the memory requirement [Crow77].

Resolution. Instead of generating 640 x 484[3] images, 1280 x 968 or evengreater pixel resolution is desired to enhance image quality.

Data Elements. An increase in the number of geometric elements provides enhanced realism in the scene.

Anti-aliasing, mapping, and shadowing involve increasing the complexity of the rendering calculations. Increased resolution raises the number of rendering calculations necessary since more pixels are displayed. Shadow casting and higher resolution increase the overall realism in the scene description by providing more detail. Each factor which increases image quality introduces distinct problems into the parallelization process. The random access memory referencing patterns associated with mapping and shadow casting can degrade performance significantly if this data is not managed effectively. The implementation of these factors in a parallel environment is strongly dependent on the decomposition and general memory referencing scheme chosen. Because this book focuses primarily on the analysis of the decomposition and memory referencing strategies, the focus here is on anti-aliasing, resolution, and number of data elements, leaving mapping and shadowing to be analyzed in a future work. Both greater resolution in image size and larger datasets require more memory to be available in the system.

If the frame buffer is stored internally in RAM and resolution is increased from 640 x 484 to 1280 x 968, the memory storage requirements quadruple. If a Z-buffer [Catm74] or A buffer [Carp84] hidden

[3]This refers to a display resolution of 640 pixels across by 484 pixels down the screen, which is standard video resolution.

surface algorithm is used, even more memory is needed due to the additional data stored per pixel. For a 1280 x 968 image maintaining 4 bytes for red, green, blue, and coverage in addition to 4 bytes for the z-value, 8 megabytes of memory is needed. This memory needs to be accessible by all of the processors.

An increase in the number of geometric elements also requires a corresponding linear increase in the memory required. If we assume that the elements used are quadrilateral polygons, each polygon requires 12 bytes to store each point, 12 bytes to store each normal, and 10 bytes (minimally) to store the connectivity information. This adds up to 106 bytes per polygon, although we can in general assume that each normal and point are shared by 4 other polygons. This results in 32 bytes per polygon. Based on these values, the amount of memory can be determined for different levels of image quality, as shown next.

The following are scenarios for memory usage based on image quality. Memory requirements for each image quality level are variable within a certain range, depending on the features included and the algorithm chosen.

Case 1. A "low quality" image generated today might involve 1,000 to 10,000 geometric elements at a resolution of 640 x 484 (standard video resolution).
Memory requirement for data: 32K up to 320K

Case 2. A "normal" image would involve 10,000 to 70,000 geometric elements, include at least anti-aliasing and possibly additional visual effects. Resolution would be 640 x 484 up to 1280 x 968.
Memory requirement for data: 320K up to 2.2 megabytes

Case 3. A "high quality" image would involve 70,000 up to 1,000,000 geometric elements, include anti-aliasing and one or more visual effects. Resolution would be 1280 x 968 up to 4K x 4K.
Memory requirement for data: 2.2 megabytes up to 32 megabytes

These estimates do not include storage for maintaining edgelists and interpolation values, in addition to the space required for advanced features such as anti-aliasing. Frame buffer or Z-buffer storage in RAM will make matters worse at all levels. The actual memory requirements are at least double and possibly quadruple the values given previously for the data storage. Access to all of this data remotely on a shared memory machine will likely cause problems

with excessive usage of the interconnection network unless the data is carefully managed. On message passing architectures, access to remote data requires knowledge of the location of the data and a complex mechanism for distributing it among the processors in the system. In addition, the time to pass the data back and forth in a message passing machine will degrade performance. One of the reasons to use a multiprocessor for graphics rendering is to handle large and complicated scenes. Therefore, the memory requirements and manipulation of data in the system need to be carefully evaluated to reduce the overhead effects.

3.1.1.2. I/O

Some advanced architectures use parallel disk setups such as the data vault mechanism in the Connection Machine. In general, though, most multiprocessors employ only a single disk for I/O. The large scene descriptions required in various applications may require a time of several seconds up to many minutes to read in the data from disk. This can only be accomplished sequentially on a conventional von Neumann architecture. In a parallel machine, however, this presents a bottleneck in performance if only one processor interacts with the disk. One can see that it is desirable to be able to exploit some type of parallelism in disk usage. At the end of the graphics computation, an image is sent out either to an external frame buffer or to the disk for storage. This operation should also be parallelized. The assumption in this book is that only a single disk is available for I/O operations, so parallelism may take the form of pipelining. For machines which allow parallel I/O, performance will greatly benefit if this feature is exploited.

As was stated in chapter 1, the I/O phases of a display algorithm are not the focus of this book. Too often, though, computer graphics specialists have ignored this portion of the program in their parallel algorithms. The I/O can directly affect how the algorithm is structured and optimization might not be feasible within the context of any given parallel tiling algorithm. An example parallel implementation of the I/O operations is illustrated later in this text.

3.1.2. Message Passing

In a message passing architecture, all of the processors are connected by an interconnection network through which messages are passed. This is the only form of communication between processors because they do not share any memory. The Intel iPSC is an example of this

type of architecture. The processor nodes in the iPSC are connected in a hypercube fashion whereby each processor can communicate with n other nodes which differ by exactly one bit in their addresses in a fully configured system containing 2^n nodes. The more recent versions of the Intel hypercube use a type of message passing mechanism that is called "wormhole" routing [Nuge88]. This routing mechanism allows processors to communicate with each other directly even if they are not directly connected. A path which is held open for the entire length of the message is created from the source to the destination. This type of path is only created for sufficiently long messages to prevent unfair use of the interconnection network.

The programmer's burden of "mapping" a parallel algorithm onto the hypercube is lessened since optimizing the algorithm for nearest neighbor communication is no longer a necessary consideration for this architecture. The issue of multiple messages contesting for portions of the same path still exists in this scheme since the wormhole is maintained as long as a message is being transferred. If another route is not available, any other messages vying for a portion of this path must wait. In addition to the above hypercube connections between nodes, there exists another processing node called the *cube manager* which is connected to all the processors via ethernet link. A three-dimensional hypercube is illustrated in figure 3.1.

Some choices for algorithm decomposition favor one type of architecture over another solely because of the method of memory distribution. Much work has been directed at the problem of mapping

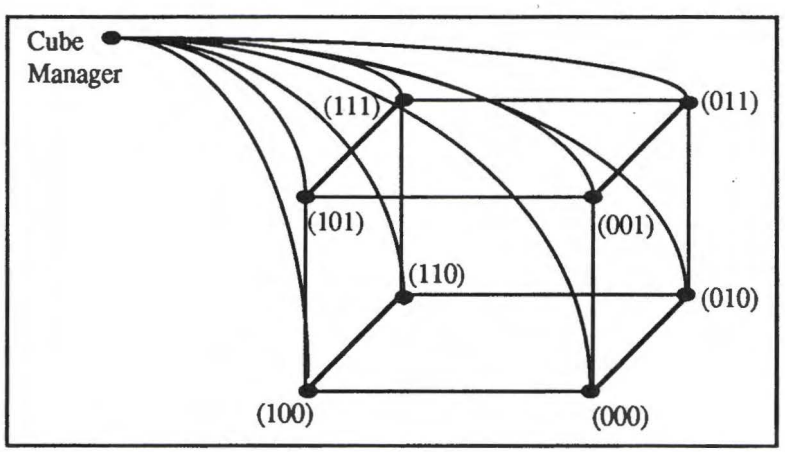


Figure 3.1: Example of hypercube architecture (8 nodes)

algorithms to architectures [Berm87], but no single methodology is optimal across different architectural models and algorithmic paradigms. Since the optimal mapping problem is NP-complete in most realistic settings [Chen88], heuristics are usually used. Sadayappan and Ercal [Sada87] have developed a technique in which data locality is exploited by a nearest neighbor mapping to reduce communication costs in a mesh architecture. This approach would be applicable to a graphics display application when it is to be implemented on such a machine.

The Intel hypercube has several limitations which make this machine a less likely candidate for a graphics image generation implementation. This type of architecture is primarily intended for problems which exhibit high parallelism and high computation costs, but little data movement. The cost of sending messages between the cube manager and each processor, as well as between the processors themselves, is very high and is only reduced when a wormhole is used. A message must contain greater than 256 bytes in order for wormhole routing to occur. The low bandwidth of the ethernet and high set-up time for messages allow only limited dynamic load balancing to be accomplished since data movement is costly. For a graphics display algorithm, the cost of propagating part or all of the database to the nodes is high, as is the cost for retrieving the rendered pixels. Parallel I/O cannot be achieved in this machine since only the cube manager processor can access the disk. The Intel hypercube was not designed for high data movement applications, and therefore the bottleneck created by the cube manager for disk I/O, as well as the cost of message passing, limits the use of this machine for graphics algorithms.

A reasonable solution might involve an initial communication of data to be scattered throughout the processors with successive communication involving very small amounts of data. This would be viable in a graphics context in which the computing cost overwhelmingly outweighs the communication cost, such as in a ray tracing program. Badouel [Bado90] has used a ray tracing algorithm with good success on the iPSC by employing a caching scheme for subsequent communication after the initial data distribution.

3.1.3. Shared Memory

There are a number of distinct interconnection network strategies for connecting processors to a global memory. Next we describe two types of shared memory multiprocessor architectures: bus-based

tightly coupled shared memory and multistage switch-based shared memory.

3.1.3.1. Bus-based Shared Memory

The shared memory paradigm allows the programmer to think in terms of parallel tasks, rather than assigning tasks to processors as in a message passing design. The Encore Multimax is an example of a shared memory multiprocessor which uses a single bus for processor to memory communication. Other bus-based systems may use multiple buses for faster communication and fewer conflicts. The Multimax can contain up to 20 processors and from 32 to 128 megabytes of memory. A drawing of this type of architecture is given in figure 3.2, where *P* indicates a processor.

The Encore's primary limitation is the fixed bandwidth of the bus, which restricts the number of processors that can be used efficiently. This does not allow testing of the algorithm on large scale processor configurations. Another factor weighing against this type of architecture is the notion of a single contiguous memory shared by all processors. Processors do not have any local memory to access, and therefore contention for the bus can become a performance degradation factor even when referencing data that does not need to be shared. A memory cache provides a form of local access and alleviates this bottleneck somewhat. For some algorithmic choices and for initial implementation and debugging, bus-based architectures could be a good choice for a graphics image generation algorithm, but they do not provide the scalable performance that is necessary to achieve fast processing of large graphics databases.

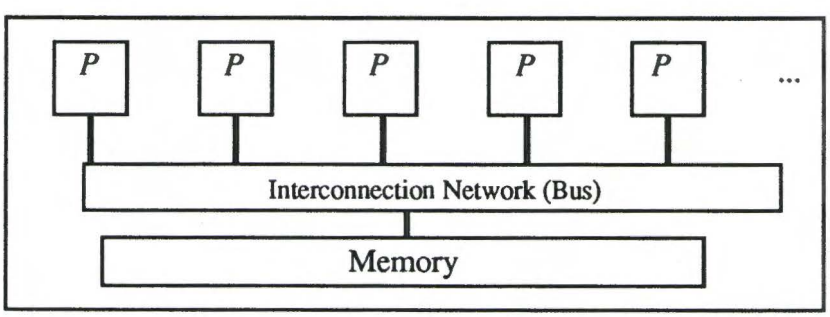


Figure 3.2: An example of a bus-based multiprocessor architecture

3.1.3.2. Switch-based Shared Memory

The BBN family of multiprocessors (which includes the Butterfly GP1000 and TC2000) are shared memory multiprocessors which utilize a complex interconnection network to connect processors to shared memory modules. For the purposes of this discussion, we will restrict ourselves to analyzing the architecture of the Butterfly GP1000. The Butterfly GP1000 is a scalable multiprocessor which can be configured from 1 to 256 processors, each containing 4 megabytes of memory. The network in this machine is built up from a basic 4 x 4 crossbar switch, which allows simultaneous communication between processors and memory as long as more than one processor does not try to communicate with the same memory module at the same time. This switch is shown in figure 3.3. A description of the advantages of this type of switch over a bus-based interconnection network is given in [BBN84].

The memory in this machine can be logically shared, but it is physically distributed across a multi-stage network switch. The processors each have access to their own local memory, as well as to remote memory modules, by making references across the switch. This puts the Butterfly into the NUMA (non-uniform memory access) class of shared memory multiprocessors since the local data access is faster than the remote data access. Other NUMA machines include the Cedar [Kuck86] project from University of Illinois as well as the NYU Ultracomputer [Gott83]. The software interface provided for the programmer in the Butterfly makes this remote referencing transparent. The interconnection network provides very high performance with a 32 Megabit/second communication bandwidth. This network is illustrated in figure 3.4.

There is no notion of a single main processor in the Butterfly, although one of the processors is connected to the multibus and serves as the processor through which I/O is accomplished. This could create a possible I/O bottleneck similar to the one stated previously for the Intel iPSC. In the case of the Butterfly, however, each processor can access the disk transparently through the I/O processor, whereas the cube manager is the only processor which can access the disk in the iPSC. This transparent disk access allows any processor to perform a disk read or write operation, although semaphores may be required to prevent interference. The I/O bottleneck still exists in the GP1000, but it is easier to program the reading in of data in the GP1000 than in the iPSC.

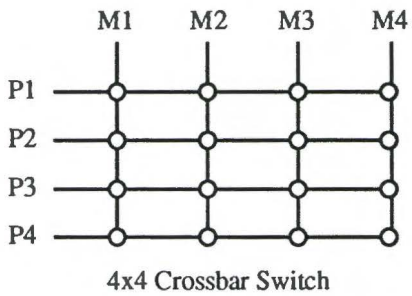


Figure 3.3: Connection of processors to memory with crossbar switch

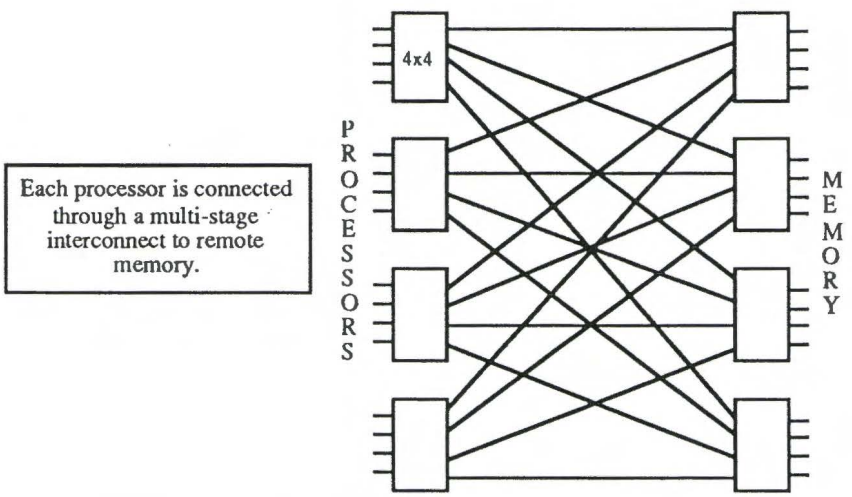


Figure 3.4: Multi-stage interconnection network in BBN Butterfly

3.2. Comparison of MIMD Methodologies

In comparing shared memory and message passing architectures for a graphics display algorithm, several items are worth considering. One important item to note when choosing an architecture for implementation is the fact that the amount of memory to be managed when generating high quality imagery can grow to be very large. Based on the algorithmic requirements given previously, it is desirable to develop parallel graphics algorithms on a machine with the following characteristics:

1. Ease of programming
2. High performance interconnection network
3. Scalable to high processor configurations

First, the shared memory programming paradigm is generally considered to be easier than message passing for the programmer to work with in developing code for an MIMD computer. The reason is that data which all processors need access to can be stored once in the system in logically shared memory. It does not need to be copied to each processor, which would otherwise waste time and/or space. Nor does the actual physical memory module location need to be specified by the programmer and sent to each processor. This is handled by the operating system (or hardware) as if the collection of memory modules is in one global address space (in the case of the Butterfly, for instance).

Secondly, the interconnection network performance in shared memory architectures is generally better suited for frequent communication of very small to very large packets of data. This is not usually the case for message passing machines. On the other hand, recent work described in [Nitz91] indicates that it is feasible to simulate a shared memory environment on a distributed memory parallel computer. In the future, this type of architecture/programming methodology might provide the type of scalable performance for which a graphics application would be well suited. Message passing architectures would be well suited to complex image generation algorithms such as ray tracing or radiosity because the cost of image generation is amortized when the task time is large in comparison to the data transfer time.

The performance of the Butterfly multistage interconnection network is much higher than the interconnection network in the iPSC, although the former is more costly. In addition, the Butterfly

network can scale to large processor configurations, and therefore provide better performance than a bus-based shared memory system.

Since the Encore Multimax and other bus-based systems are limited in the number of processors they can support, low or normal quality image generation would fit well onto that type of architecture. Higher quality imagery demands more compute power as well as more memory than is normally available on bus architectures. One exception is the Alliant FX/28000, which uses up to 28 Intel i860 processors. The number of processors cannot be increased beyond this, but very high performance has been obtained on this system. Still, the Butterfly TC2000 is a faster version of the GP1000 and does provide the scalability necessary to render extremely large datasets. If judicious distribution of graphics data is used, memory latency can be reduced to a negligible overhead.

Although all of these architectures are suitable for graphics algorithms, the Butterfly environment provides a stronger case for high performance. In summary, with regard to the issues discussed previously, the BBN Butterfly is the best example of a shared memory multiprocessor which meets these requirements. The distributed memory modules within the Butterfly allow the programmer to take advantage of local memory access while a global view is provided of shared memory. The performance of the interconnection network and memory modules is better than a bus-based system such as the Encore Multimax, and the number of processors that the machine is capable of supporting allows massive parallelism.

The BBN Butterfly GP1000 at The Ohio State University's Computer and Information Science Department was used for the primary development and debugging of the algorithms presented in this book. This machine only had 10 processors, so it was desirable to test the programs on a larger machine. The Naval Research Lab, Georgia Institute of Technology, and Michigan State University all provided access to machines with larger configurations for this purpose. Final testing was done on the Butterfly GP1000 located at the headquarters of BBN Advanced Computers, Inc. This machine contains 107 processors, but we have limited testing of the programs to 96 processors. Under all circumstances, this machine was not being used by others at the time of testing, so no other processes interfered with the timings. The BBN TC2000 is their next generation multiprocessor which contains numerous enhancements to the design used in the GP1000. For some of the tests given in chapter 5, access was provided to a 47 processor TC2000 at Argonne National Laboratory, as well as to a 128 processor TC2000 at Lawrence Livermore National Laboratory. As a note to the reader, for further

reference in the rest of this book, the term *processor* is used to denote a processor-memory module on the Butterfly.

3.3. The BBN Programming Environment

An algorithm can be designed to take advantage of a particular machine's characteristics to enhance overall performance. The algorithms illustrated here were designed for an MIMD architecture and optimized for implementation on the BBN Butterfly. Ideally, one would like to design program code so that it will run unmodified on a variety of parallel architectures. Although some parallel environments are available on a wide variety of machines (notably Linda [Ahuh86]), we did not have access to these programming tools. The parallel programming paradigms available on the Butterfly at Ohio State (where the code was originally developed) are BBN's Uniform System and a version of C-Threads originally developed at Carnegie-Mellon University [Coop87]. Additionally, Lawrence Livermore provides a split-join programming model called PCP, but it is currently only supported on the TC2000 [Gord91], and not the GP1000.

The Uniform System approach is a BBN specific parallel programming scheme [BBN89a]. This method uses the concept of generators which spawn off parallel tasks in a number of different ways selectable by the programmer. The second paradigm is C-Threads, which was ported to the Butterfly at The Ohio State University [Sami89]. The Uniform System was used for implementation purposes since it is supported by BBN and the program code would be able to run unmodified on the TC2000 as well.

The Uniform System parallel programming paradigm is designed to allow the programmer to develop parallel applications which are insensitive to the actual number of processors in the system. The program code does not need to be modified, nor special cases taken into account when it is run under different processor configurations. This also allows for debugging and testing on a small number of processors and later using all of the processors in the system for timing measurements. The Uniform System is a library of routines that the user links with in a C or Fortran program. In the case of the algorithms presented here, all of the code is written in C. The Uniform System can basically be divided into two sections: the shared memory portion and the task assignment portion. In addition, special routines are available to handle atomic operations, locks for synchronization, spin waits, and other configuration operators. Memory can be allocated in the system as:

1. Local variables
2. Global variables
3. Dynamic storage
4. Shared dynamic storage
5. Copied storage

Local, global, and dynamically allocated memory are treated the same as in any normal C program, with each processor having its own copy of a variable. Processor i and processor j may both reference a variable *l*, but its value is different on each processor since each has its own local copy.

Shared dynamic storage allows one to create space for data which can be stored somewhere in global memory. The data is available to each processor, but is only stored physically in one processor's memory module so that if processor i updates the value to shared variable *s*, processor j would also see the new value. Of course, synchronization must be used to correctly update a shared variable if two processors could possibly change it simultaneously. Routines exist in the Uniform System where a shared variable can be specified to be stored on a particular memory module (for instance the local processor's module) or scattered somewhere in the system. This allows the programmer to create efficient access to shared memory and to prevent hot spot contention. Hot spot contention occurs when a large number of remote references backup on a single switch node, causing delays in the network. The process node controller (PNC) on the processor board determines the location of a shared memory reference and handles the message traffic to complete the write or read operation. The memory allocator forces all shared memory to be allocated to locations which are above a given virtual address fence register.

The last type of storage, copied storage, allows data to be copied to all of the processors, effectively providing local access to a common variable. After the variable is copied, each processor has its own local copy of the variable so a modification will not be propagated to all the other processors, as in the shared memory case. This method is used to copy read-only data to all processors. It is also used to allow processors to know the location of a shared memory variable. As an example, a processor allocates a shared variable and places a value in the memory location, only that particular processor knows which memory location is used in shared storage. In order to allow the other processors to know the location, a call is made to the Uniform System routine *Share*, which propagates the *address* of the shared variable to

all of the other processors so that they all can refer to the single memory location. The *Share* routine can also be used to just copy data, so instead of propagating the address of the variable, the value of the variable itself is propagated. The latter method of usage is an example of the copied storage approach.

Task assignment in the Uniform System is handled by a mechanism known as *task generators* which is a distributed task assignment system that works as follows. A program starts off running on a single processor. When a parallel environment is created on P processors, shared memory can be allocated and the other (P - 1) processors start a spin-loop where they execute code that detects if there are any tasks available to work on. As soon as a generator (a procedure initializing a parallel environment) is executed by the first processor, a specified number of processors (any number from 1 to P with the choice of inclusion of the initiating processor) execute a task activator procedure to generate the next task. If a parallel ***for*** loop is desired, the task activator would consist of an atomic operation which increments the ***for*** loop index. As soon as the index is atomically updated on processor i, that processor begins work using the index as a parameter to a worker procedure. This happens throughout the system so that each processor essentially finds work for itself rather than using a central controlling mechanism. As soon as a worker is finished, that processor tries to find additional work by checking the task activator again. Note that more than one task activator can be running at the same time by using recursive generator calls, although the order of execution is difficult to predict. When all of the tasks are exhausted, the generator finishes, and if no other generators have created tasks, the initial processor proceeds serially while the others spin-wait until more work comes along.

3.4. Summary

In the first section of this chapter, a number of multiprocessor architectures are presented for the purpose of examining their characteristics with regard to a graphics display algorithm. In the second section, criteria for evaluation of these architectures is given, and each type of machine (of which all were available for testing) is scrutinized based on its characteristics and suitability for implementation of a parallel graphics algorithm. The BBN Butterfly is shown to be the computer most suitable for implementation of a parallel graphics display algorithm. In the third section, the Butterfly programming environment is described to give the reader a

better understanding of its operational characteristics from a software point of view.

The next chapter describes the serial algorithm upon which the different parallel decompositions are based. In addition, timing measurements as well as measurements of performance analysis are discussed.

4

Overview of Base Level Implementation

In this chapter, we describe the choices that were made for a base level implementation of the different parallel graphics decompositions. The approach used for developing the parallel programs is to devise a single basis graphics rendering algorithm, and then build different parallel task partitioning and memory referencing schemes on top of it. This allows an equal comparison of a number of different approaches for parallelism since the underlying algorithm is the same. This basis algorithm is not compromised by the parallel algorithms since it can be modified to the specifics of each particular parallel approach. The first section of this chapter presents this basis algorithm by describing the underlying serial approach, as well as the choices that were made which were common to all of the parallel formulations. The second section of this chapter describes the measurement techniques used to obtain timings for the different programs. The last section gives the performance analysis measures used to analyze the different parallel implementations that are presented in the next chapter.

4.1. Design of the Basis Algorithm

The purpose of implementing a number of parallel graphics rendering algorithms is to analyze different parallel work decompositions and shared memory referencing schemes to determine which method is the most viable for general use. In order to make a straightforward comparison of the possible decompositions, a serial algorithm has been developed upon which the various parallel formulations are based. Most parallel rendering algorithms developed in the past were designed similarly. Essentially, some portion of the parallel algorithm consists of a single task resembling a serial algorithm in a smaller context. This approach makes it easier to compare the parallel implementations because their relative speed for the basic portion of the algorithm is the same. This may seem like we are compromising the aspects of a parallel machine by using smaller serial tasks, but such is not the case. It just turns out that for this type of problem, the solutions presented are the most straightforward and yield the highest performance compared with a functional work decomposition.

Based on the taxonomy and algorithm analysis presented in chapter 2, several variations on an image space parallel decomposition have been implemented. Each single task of the parallel algorithms consists of solving the rendering problem in a serial manner for a particular area of the image space. Chapter 5 describes these algorithms, which vary in their method of task assignment to processors, area size, and memory referencing characteristics. In this chapter, the basis sequential algorithm for the front end and the single task tiling portion are described.

This serial basis algorithm is a scan line Z-buffer algorithm [Myer75] which incorporates the stochastic sampling method [Cook86] for anti-aliasing as an extension. The serial algorithm used here was originally developed separately as part of a project to enable scientists to render polygonal datasets at varying degrees of accuracy. Several different anti-aliasing methods can be used including a straight Z-buffer, an analytic method, and a stochastic sampling method with 16 samples per pixel. In addition, several illumination models are available, including those developed by Gouraud [Gour71], Phong [Phon75], Blinn [Blin77], and Cook-Torrance [Cook82]. The rendering method which is used for the comparison tests incorporates stochastic sampling for anti-aliasing with the Blinn shading model using various images rendered at a resolution of 640 x 484. We will elaborate on the test scenes in section 4.2.1.

As stated previously, each algorithm involves a break-up of the image space into different areas, and the image rendering problem is essentially solved serially in a given area. Clipping is done initially for the entire screen, but for each individual area, single scan line clipping is used instead of polygon clipping to the area boundaries. It might be interesting to compare scan line versus polygon clipping, but this would only minimally affect the overall performance of the algorithms.

4.1.1. Front End

The files which are used for the test object data are called *detail* files. This format was developed at The Ohio State University Computer Graphics Research Group (now known as the Advanced Computing Center for Arts and Design) in the 1970s. This format is fairly compact and the data in the file is stored in binary. The format is shown here:

```
num_pts num_polys
x1 y1 z1
x2 y2 z2
.
.
.
num_vertices point1 point2 point3 ...
num_vertices point1 point2 point3 ...
.
.
.
```

The number of points and number of polygons are 16 bit integers, which means that only 32,767 points and polygons are allowed in a single object. The points list follows these two values, and each point is represented as three floating point values. After that, the polygon list includes the number of vertices in a polygon and indices to the points list above. An edge is implied between adjacent vertices in the list. The polygons are guaranteed to be convex, and it is assumed that the last vertex listed for a polygon is connected to the first by an edge. Since the object detail files limit the number of vertices and polygons, larger objects must be broken up into smaller objects so that the same format may be used.

The front end of each parallel graphics program consists of the following phases:

1. Read in object data files from disk.
2. Perform necessary transformations.
3. Reject back-facing polygons and clip polygons to screen borders.
4. Place polygons into shared data structure.

A parallel pipelined implementation of this front end is described in the following subsections.

4.1.1.1. Reading in Objects

It was necessary to modify the detail file format so that the regular objects could be broken up into sub-objects to allow sufficient parallelism in the front end. This allows the object data to be distributed across the memories of the processors. In order to do this, a separate program was written which reads in an object file and creates a new object file, consisting of the same original object but subdivided into components. The sub-object size is determined based on the number of polygons in the original object; it can be as small as 100 polygons for simple objects and increases to 1000 polygons for more complex objects. The only problem with subdividing the object is that the original normals at the vertices need to be kept with the points since a new normal calculated for a sub-object alone could be incorrect.

An incorrect normal would be calculated in the following scenario. A polygon which was previously part of the original object is moved into a new sub-object. The normal of this polygon will no longer influence the vertex normals of its neighbors unless this normal is calculated prior to the subdivision of the object and copied along with its vertex. Thus, the untransformed vertex normals are calculated and stored with the sub-object directly after the points list in each file. This creates a somewhat larger data file format, but is the only solution that allows distributing the objects across memories. There is a slight problem with storing the untransformed normal if the object transformation from object space to eye space includes non-uniform scaling operations, so this situation was prevented from occurring in the test cases used here. The time for manipulating the datafiles prior to program execution is not accounted for in the timings since this is just a variation of the original object format which could be output from any data generation package.

4.1.1.2. Parallelizing the Front End

The diagram in figure 4.1 illustrates how phases 1, 2, and 3 of the front end can be overlapped in parallel execution. When a processor is available to do a read operation, it performs a single pass check of a global array to see how much data is currently stored on all the other processors. If a given processor contains fewer polygons than the average of all the processors, then this processor puts itself on the queue to read an object from disk into its own local memory.

The number of polygons read in and determined to be front facing is then stored in a global array. By using this scheme, the input data is scattered among all the memory modules in roughly equal portions. The data is also sharable so that all processors have access to it. After the data is read in on a given processor, the disk is available for the next processor to access. This algorithm creates a pipeline which is faster than serially processing the data and distributing it.

This scattering of polygons allows a nearly uniform scattering of data as well as work for the front end, so that each processor's work is approximately of time complexity $O(N/P)$ where N is the total number of polygons read in. For datasets which are small, the pipeline does not become completely saturated if the number of sub-objects is less than the number of processors. This situation would not provide enough work for all of the processors during the front end phases, nor would the object data get completely scattered throughout the

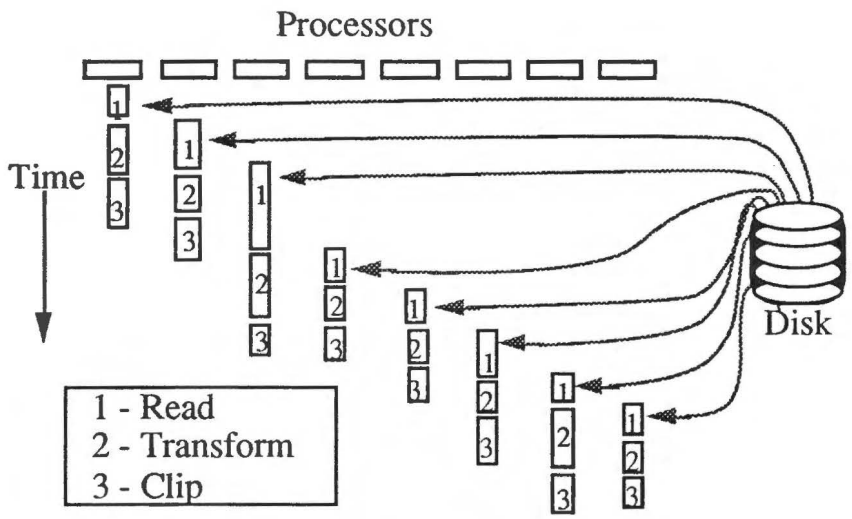


Figure 4.1: Overlapped disk access with front end phases

memories in the system. For the test cases here, we were not interested in completely optimizing the front end; it was developed merely as an efficient method to read-in and scatter the object datasets. Obviously this portion of the program needs to be optimized for each combination of object data format and chosen hardware. The method given here is a general outline of one way to parallelize this section of code.

The front end should be optimized depending on the intended application of the renderer. For animated sequences, reading in data is only necessary when a new object enters the scene for a given frame. More optimization might be spent on the transformation and clipping phases for this application. If still images are required, it might be desirable to optimize the entire front end. Regardless, it can be seen that it is worthwhile to parallelize this section of the parallel program.

This example parallelization of the front end is the same for all of the algorithms described herein. The only difference occurs in the section where polygons are put into data structures depending on their screen space location, as described in section 4.1.1.3. Since this latter portion of the front end of each parallel algorithm may be slightly different in complexity, we include the time differences for this portion of the front end in our overall algorithm analysis in chapter 6. The data structure used for storing the polygons according to their location is described in the next subsection.

4.1.1.3. Placing Polygons in Shared Data Structure

Each of the parallel algorithms is a variation on an area screen space subdivision algorithm. Prior to the parallel tiling and rendering phases, the polygons must be tagged as to which subdivision(s) they belong to. The polygons are placed in a data structure which is shared among all of the processors so that each processor can obtain the exact polygons relevant to any subdivision. Each parallel decomposition scheme employs an approach in placing the data in this structure that is slightly different than the one used in a traditional serial scan line algorithm. In a serial approach, this section consists of loading polygons into an array of linked lists called a y-bucket list (see [Roge85] for details on how this is done). Each linked list (or y-bucket) corresponds to a scan line and contains pointers to polygons which have their minimum y-extent on that particular scan line. During the tiling section of a serial algorithm, the program investigates the active y-bucket to determine which new polygons start on the given scan line and should be stored in the

active polygon list. For the parallel image space decompositions, it is necessary to do a pre-culling operation prior to creating a *y*-bucket list for a given area on the screen. This pre-culling operation involves loading the polygons into an *area bucket* data structure so that it is possible to find out which polygons are relevant to a given area on the screen, not just a single scan line.

Although the various parallel algorithms rely on different size areas for their partitions, the basic culling operation is similar in each. The area bucket list is used to store polygons for the areas which will later become separate parallel tasks in the tiling section. During the front end, the bounding box of each polygon read in by a processor is checked against the area mesh created for the given parallel decomposition. A pointer to the polygon is then stored in each area bucket that the bounding box crosses. The polygons are not clipped to these areas, nor is a more stringent test employed to see if the actual polygon (not just its bounding box) intersects the area. Although a stricter test could be used, the approach used here is fast and uncomplicated, with the only drawback being lack of accuracy. In other words, a polygon's bounding box might cross over an area in which the polygon itself is not actually present. The common thread to this portion of the program, which is the same for all the parallel formulations, is that the area bucket list is a shared data structure available to all processors. As the polygons are read in during the front end, each processor determines the appropriate area bucket to place a shared pointer to the polygon. A lock is used to prevent more than one processor from placing the pointer into the same area bucket list at the same time. This constitutes what we will call the Uniformly Distributed (UD) memory referencing scheme since the polygons themselves are scattered throughout global memory, in addition to a scattering of the polygon pointers in the area bucket data structure.

As was mentioned previously, each partitioning scheme implements this section of the front end in a slightly different manner since the size and number of areas is dependent on the parallel partitioning scheme. Specific details on the usage of the area bucket data structure are given with each individual algorithm description in chapter 5.

4.1.2. Tiling

The tiling section of a scan line based graphics rendering algorithm is the most time consuming portion of this type of program. It consists primarily of the following phases:

For each scan line in an area:

1. Determine which polygons are new to the current scan line.
2. Build edge lists for those polygons and determine which edge pairs start on the current scan line.
3. Update any edge pairs from the previous scan line in which one or both of the original edges in the pair is no longer active.
4. Interpolate vertically from the previous scan line to the current scan line for all parameters, such as edge position, color, and normal.
5. Perform hidden surface elimination and anti-aliasing.
6. Shade the fragments which are visible within each pixel for the scan line.
7. Send finished scan line out to the frame buffer.
8. Deactivate polygons, edge pairs, and edge lists which end on the current scan line.

In a traditional serial approach, the area refers to the entire screen and each scan line is the width of the screen. In the case of a parallel image space algorithm, an area is a single task and a scan line is the width of that area (each area may be different in size, though). We will now systematically go through each of the preceding phases, pointing out the choices made which are common to all of the parallel implementations.

In the tiling phase, a task corresponding to a particular area on the screen is obtained by a processor. The polygons which are relevant to this area are stored in an area bucket in shared memory during the front end. During the tiling phase, this processor determines which polygons to work with by examining the appropriate area bucket. The polygons which are in this area bucket are loaded into that processor's local memory y-bucket list. The y-bucket list contains the pointers to the polygons in shared memory. Since some polygons could have started above the first scan line of the area, these polygons are stored in the top scan line y-bucket for that area. Phase 1 of the tiling operation involves traversing the y-bucket for the current scan line and extracting those polygons which start on this scan line.

Phase 2 involves building the edge lists and edge pairs and storing them in the processor's local memory. These are put in local memory because this type of access is much faster than remotely referencing the data. There are several reasons why this is wasteful, however. If a polygon crosses the boundaries of more than one area, the edge lists are constructed for each area in addition to the

duplication of memory required to store these data structures in each processor's local memory. It is possible to store these data structures in shared memory but the following complications could arise. Synchronization would be required if two processors try to build the same edge lists at the same time. In addition, although the initial interpolation parameters and delta values are the same for a duplicated edge, the current value of an interpolation parameter depends on which scan line is active for each processor sharing the edge. It is likely that the active scan line is different in each processor. Consequently, the memory savings of storing the data in each processor's local memory is more than offset by the additional remote referencing cost that would otherwise be incurred. Therefore, although the local referencing method may be slightly wasteful in memory usage and involve duplication to build some data structures, it is superior in speed. As a result, the tiling portion of the program will execute faster and the interconnection network will not be used.

Phase 3 (updating the edge pairs) of the tiling operation proceeds as in a traditional scan line algorithm, and no remote referencing is incurred here. Phase 4 (scan line interpolation) is also the same as a serial method with no remote referencing. Phase 5 (hidden surface removal) involves the use of a stochastic sampling anti-aliasing technique which allows hidden surface removal and anti-aliasing to occur simultaneously. Some remote referencing is required in the Uniformly Distributed memory referencing scheme to obtain the plane equation for a given polygon from shared memory. Phase 6 (shading) involves performing the illumination calculation on the visible polygon fragments left over from phase 5. After the fragments' colors are determined, a box filter is used to convolve the fragments with each pixel to determine the overall pixel color. At the current time, the filter is limited to the width of a single pixel since it is difficult to handle pixels which are beyond the border of the current area in a parallel environment. An area for future work might include applying a Gaussian filter which extends beyond a single pixel boundary. After the the pixel colors are determined, they are loaded into a single scan line buffer and block transferred to the virtual frame buffer in phase 7. The block transfer is faster than updating each pixel in the virtual frame buffer one at a time. After this phase, the polygons, edges, and edge pairs which expire on the current scan line are deactivated in phase 8. More information on how the scan line data is stored in the frame buffer and how it is written out is included in the next subsection.

4.1.3. Back End

Since the areas on the screen are computed in a random order which depends on the scheduling of tasks onto processors, full scan lines cannot be output onto the frame buffer during the computation of the image. On the other hand, each individual area could be output as it is calculated, but this is distracting to the user, in addition to creating additional network traffic. The solution then, is to store the frame buffer internally in memory and output the pixel data after the tiling is completed. This is accomplished in the following manner. A virtual frame buffer is stored in the physical memory of the machine by scattering the rows of the frame buffer among the different memory modules. This allows uniform scattering of the data and avoids hot spot contention. As stated previously, rather than have the processors directly write pixel values to remote memory, each processor contains a small local memory buffer corresponding to the width of a scan line. After an area-width scan line is finished, this buffer is block transferred from local memory to its place in the globally shared virtual frame buffer. This continues throughout the tiling portion of the program. At the end of the tiling operation, the virtual frame buffer is completely filled and can be displayed.

After the image has been calculated, each scan line is compressed using run-length encoding. Run-length encoding allows the image to be stored using less space than is required with a simple pixel map method. This operation is done in parallel but no information is kept from one scan line to the next, though this would be done in a sequential implementation. As the run-length encoding is proceeding, each processor checks a shared variable that indicates the current scan line which is available to write out to disk. If the variable indicates that a given processor's scan line can be written out, that processor writes out the run-length encoded scan line and proceeds to find more scan lines to process. This allows the image to be written out in scan line order while the run-length encoding is performed in parallel. Of course, the scan line order forces a bottleneck situation, but this is the only way the image can be written out. Modifications to this section of the program would allow tuning to a particular application.

The preceding descriptions above constitute the basis serial algorithm and parallel extensions which are common to all of the parallel partitioning schemes. These schemes are outlined in detail in chapter 5 and compared to each other, along with an evaluation of shared memory referencing in chapter 6. The next section is a description of

the testing procedures and performance analysis methods employed in the analysis of the algorithms.

4.2. Testing Procedures

In this section, we discuss how the algorithms are evaluated according to a number of criteria. The scenes which are used to test the various algorithms are described, as is the timing procedure utilized.

4.2.1. Test Scenes

It is important that the three-dimensional scenes which are used as test data accurately reflect what might occur in everyday usage of a graphics display program. Since it is hard to imagine what an "average" scene might entail, numerous researchers have developed their own methods of analyzing algorithms by providing a number of test circumstances. The only unfortunate aspect of this situation is that the data is not available to other researchers to test their own programs on. To rectify this situation, Eric Haines has developed what he calls the standard procedural database (SPD), which is used for the sole purpose of testing rendering algorithms [Hain87a]. His intended application was ray tracing, but these scenes can be used for testing any display algorithm. Among the scenes available are: a group of balls, a set of gears, a tetrahedron, a tree, a group of rings, and a fractal mountain. Each scene can be generated to create as many polygons as the user desires since the programs create the database procedurally. It was decided to test the algorithms here using small to large databases. The tree was generated with approximately 106,000 polygons, while the mountain was generated with 131,000 polygons. Haines has given view parameters as well which are also part of the database specification. In the case of the mountain image, a much denser version was created than Haines used in his testing (his mountain only contained 8K polygons). Using his viewing matrix, a majority of the polygons would have been clipped out of the scene. To rectify this, a new view matrix was constructed to allow the entire mountain to be seen. The 4 x 4 matrix used for viewing is included in the appendix in equation A.1.

It was also desirable to test the algorithms on some real world type data, so we used a stegosaurus image which was designed for an animation and a Chrysler Laser automobile which was designed from a CAD/CAM program. The stegosaurus is not rendered with its plates since they were unavailable at the time of testing. The stegosaurus contains approximately 10,000 polygons, while the

automobile contains approximately 46,000 polygons. The stegosaurus data was created by John Donkin of ACCAD (the Ohio State University Advanced Computing Center for the Arts and Design) for the Fernbank Museum in Atlanta. The car was created by Chrysler and obtained from Evans & Sutherland Computer Corporation. All tests were performed at a resolution of 640 x 484 using Blinn's [Blin77] light model for the specular component. An illustration of these images appears in color plate 1.

In chapter 6, higher density images were used to test the algorithms with more demanding data. For these tests, a version of Haines' rings database was created with approximately 568,000 polygons, and a denser version of the tree database was generated with 851,000 polygons.

4.2.2. Timing

It is important to state how the programs were timed on the system to show what is included in the performance graphs given later in this book. On a multiprocessor such as the Intel iPSC, program code must be loaded into all of the processors prior to running the program itself. On the Butterfly, this is not the case. The Butterfly is a virtual memory machine, and the program code is paged into memory as it is needed [BBN89a]. Prior to starting a parallel environment, the first processor contains the resident code and any allocated memory. When a parallel environment is started and a processor references a page of memory or program code that is not resident in that processor's local memory module, this item is paged in automatically. The first time this happens, it may occur almost simultaneously on all processors since they will likely be executing the same code at the start of a parallel environment. A bottleneck situation occurs since all of the processors are trying to obtain pages of program code at nearly the same time. Although this method is significantly faster than the loading operation in the iPSC (primarily due to the speed and topology of the Butterfly interconnect), any timings which include this startup cost are somewhat misleading since they do not indicate the true performance of the machine. An adequate solution used by most researchers is to run the parallel portion a second time within the program itself and evaluate program performance for this second iteration only. This method is used for the results given in chapter 5.

Another overhead is encountered due to the implementation of the Uniform System on the Butterfly which occurs when a task generator is called for the first time. This is due to the fact that all of the processors need to be notified that a generator is available for

execution, and this takes $O(P)$ time. To alleviate this effect, each processor starts off its first task inside the generator by hitting a barrier. Once all processors have hit the barrier, the processors are released and the timing starts. On the Butterfly GP1000, a time of approximately 2 seconds has been measured for our maximum test configuration of 96 processors to hit this barrier. This time is seen only once at the onset of a generator and would not be present in other parallel programming models.

The timing mechanism used on the GP1000 is a routine called *getrtc* (get real-time clock) which gives time increments of 64.5 μsec. On the TC2000, the clock ticks are accurate to 1 μsec. The performance of all algorithms is evaluated using these routines from within the program. There is no differentiation between system time and program time since during most of these tests, no other processes were running besides the normal MACH system processes. The Uniform System ensures that only one user process is assigned to a processor, resulting in no interference caused by other user processes. The nature of interprocessor communication and overhead due to system processes can change the performance, and as a result, the same test run multiple times can vary by several tenths of a second. In most cases, this amounts to less than 1% of the total parallel time, so we did not rerun the programs to obtain an average time. This was done in all cases except in the case of the final timing on 96 processors which was averaged over five runs for additional accuracy. The tests consumed much CPU time and took many months of programmer time to initiate and gather results.

4.3. Performance Analysis

Various graphics researchers in the past have written simulators in software to verify their parallel algorithms which may have been designed for hardware or a conventional multiprocessor. At the time of their research, the hardware was either not available or was too costly to obtain for actual tests. Unfortunately, some very real factors such as communication, network contention, and load balancing cannot be fully analyzed in a simulated environment.

The first subsection here describes the measures usually used to gauge performance of a parallel program: time, speedup, and efficiency. The value of these factors is that they provide an indication of the actual realized performance, relative parallel performance, and processor utilization, respectively. It is not sufficient to look at any one of these measurements alone since one might be misled by not observing the whole picture.

In the context of parallelizing a circuit simulation application, Sadayappan and Visvanathan [Sada88] develop a framework where the overall performance of a parallel algorithm is evaluated by breaking it down into relevant component factors. It would be desirable to develop a similar framework to interpret performance of a parallel graphics algorithm in terms of quantitative measures characterizing relevant factors.

A number of overhead factors are introduced when a program is implemented on a parallel architecture and these are outlined in the second subsection. In some cases, these factors relate to the machine itself, and in other cases they relate to the changes necessary to allow the program to run in parallel. Initially, a program is modified or specifically designed for implementation on a parallel architecture. The changes represent the differences between the serial and parallel versions of the program. While these changes are introduced by the programmer, additional overhead factors are introduced by the fact that a particular machine architecture is used and communication is necessary to allow the processors to work on the problem simultaneously.

4.3.1. Time, Speedup, and Efficiency

Due to the size of the datasets, the 4 megabyte limit in memory per processor (on the GP1000), and the size of the data structures needed for algorithm storage, it was necessary to initiate the timing tests above one processor. The reason this was done was to avoid paging during the computation if the datasets did not fit into physical memory. Since there is only a single disk on the Butterfly, paging would cause a serial bottleneck which would not provide realistic parallel timings for the given datasets. By checking to see if a given timing run is paging (using the MACH command vmstat), the minimum number of processors which can be used without paging effects can be determined. This minimum number for each dataset is as follows: 2 processors for the stegosaurus image, 6 processors for the Laser image, and 12 processors for the tree and mountain images. These minimum processor configurations are used since it is not possible to determine the amount of time incurred due to paging and then eliminate that time from the results.

The potential speedup in a parallel algorithm is calculated by the formula given in equation 4.1 (also called *Amdahl's law* [Amda67]).

$$\frac{T(1)}{T(P)} = \frac{(s+p)}{\left(s+\frac{p}{P}\right)} \tag{4.1}$$

$T(x)$ is the computation time on x processors, P is the number of processors, s is the sequential portion of the computation, and p is the parallel portion (identified on one processor). If $s = 0$, then we have obtained linear speedup. The potential speedup as given in Amdahl's law is limited by the sequential portion of the program. If one assumes that s and p are percentage values, then we can rewrite the law with a numerator of 1 since the total computation time on 1 processor adds up to 100%.

Amdahl's law has gained acceptance as a predictor of the maximum expected speedup of parallel algorithms on multiprocessors. In a sense, this law expresses doubt about the amount of speedup attainable by parallel algorithms on real machines. The doubt is well-founded since most algorithms involve some inherently sequential portions of code. Operations such as synchronization of processes, message passing delays, and memory latency also reduce the potential parallelism.

For parallel programs, Amdahl's law can be applied if we know the serial percentage of the algorithm for a given value of P. Often, this is difficult to calculate prior to running the program on P processors, but it does serve as a good indicator of performance if we calculate the percentages after running the program. Instead, the speedup is used as a guideline to relative program performance as the number of processors is increased. Equation 4.2 shows how speedup is normally determined. The time using 1 processor for a parallel program and its single memory module is divided by the time using P processors and P memory modules. This is indicated in equation 4.2 where $T_x(y)$ refers to the time using x memory modules on y processors. The speedup is basically an indication of the effective number of processors utilized.

$$\text{Speedup} = \frac{T_1(1)}{T_P(P)} \tag{4.2}$$

The minimum number of processors (MIN) in which each test scene fits into the physical memory of the machine without paging precludes direct testing of the algorithms on one processor to determine speedup (recall MIN is image dependent). Instead, a work-around solution is used to evaluate the time that the algorithms

would take if they could be run on a single processor with enough physical memory.

The programs are started for each image on MIN number of processors so that there is enough physical memory available without paging. The data is read in, transformed, and clipped using this number of processors, which incorporates the Uniformly Distributed memory scattering scheme. The tiling section is run using only one processor, however. This processor retrieves the data from global memory as is necessary. This would be the fastest method to run each sequential algorithm on the physical machine, given the amount of memory actually available per processor. Equation 4.3 is used to calculate the estimated speedup on 96 processors.

$$\text{Estimated Speedup} = \frac{T_{MIN}(1)}{T_{96}(96)} \tag{4.3}$$

Efficiency is calculated using equation 4.4, where P_{max} is the maximum number of processors used for testing (here, P_{max} = 96). Efficiency is an indication of the utilization of the processors in the system.

$$\text{Efficiency} = \frac{\text{Estimated Speedup}}{P_{max}} \tag{4.4}$$

For the results shown in chapter 5, the time and speedup graphs are shown only for the tiling section of the programs since this is the most time consuming, as well as the most parallelizable portion of the program. In chapter 6, a comparison of all the algorithms' times is included, along with the time for the initial startup operations from the front end which are specific to each approach. This gives us a fair basis for comparison of each of the algorithms. The total front end and back end times are not included, although considerable effort was spent in parallelizing this portion of the code. The reason is that the optimization of this portion of the code would be handled differently depending on the application intended.

Next we describe a number of different overhead factors and their effects on a parallel program.

4.3.2. Overhead of Parallel versus Serial Implementation

Factors which are introduced by the difference between the uniprocessor and multiprocessor version of the programs include: schedul-

ing, memory latency, communication utilizing block transfers, overhead due to adaptation of the algorithm for parallel execution, load imbalance, contention, and synchronization. In this section, we describe each overhead factor and the testing method used to evaluate each factor's impact on the algorithms presented in chapter 5. The testing method used to determine each overhead factor is generally common to all of the parallel algorithms presented in chapter 5, but some differences in measurement occur due to the nature of these algorithms. These differences are elaborated upon at their point of reference in chapter 5. All of the testing to evaluate these overheads was done at 96 processors since the maximum number of processors used provides the worst case scenario as far as the overhead factors are concerned. It should also be noted that this testing was done separate from the performance timing for the algorithms.

The different overhead factors discussed in this subsection are determined as a percentage of the total processor-time space, as shown in figure 4.2.

Each processor may take a different amount of time to complete its work. The leftover time is idle time, as is illustrated in the figure. The time when the last of P processors finishes its work is $T(P)$ or just Tp. There may be a number of different ways of evaluating the overhead factors, but the method chosen here is to determine a percentage value of the overhead with respect to the total processor-time space ($P * Tp$). This expression is used as the denominator in the

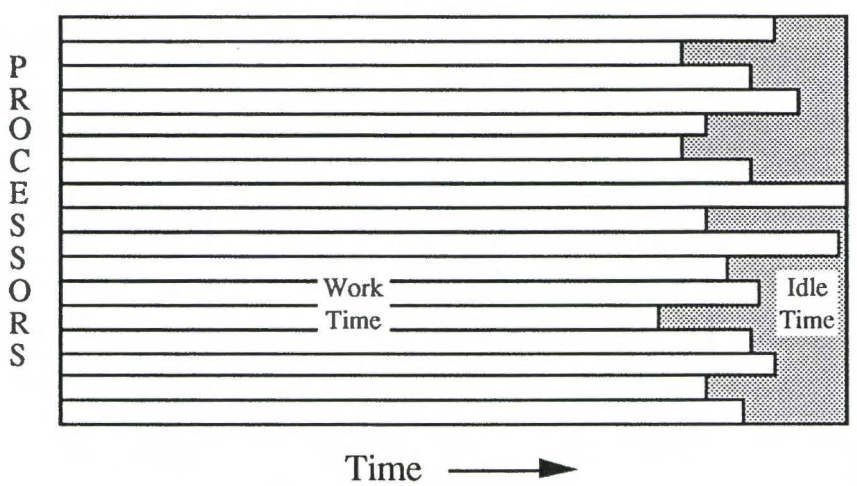


Figure 4.2: Processor-time space

equations since a processor must wait until all other processors are finished before it is able to work on some other job outside of the given program. The overhead percentage value should be taken to mean the effect of the overhead on the total processor-time space.

4.3.2.1. Scheduling

In discussing scheduling overhead, we first describe our definition of this effect and then go over the mechanism by which tasks are assigned in the Uniform System. Scheduling overhead in a parallel context refers to the time it takes for a given task to be scheduled by the system. In this case, it is assumed that a parallel environment has already been started and a set of tasks are available for execution. Scheduling is accomplished by use of a critical section which must be executed by one processor at a time. The time of scheduling overhead for a single task is the amount of time it takes to run through this critical section.

Scheduling in the Uniform System works by dynamic task assignment, as stated previously. A worker routine constitutes an individual task that is called with a given parameter list. The example used previously referred to a parallel version of a ***for*** loop as shown below:

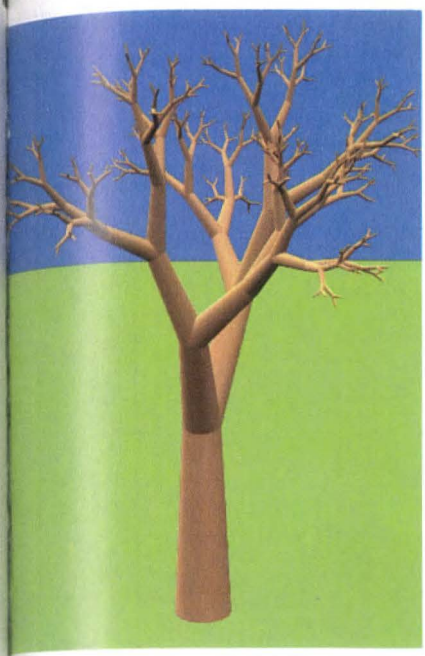
```
for (i = 0; i < range; i++)
        do_work(i);
```

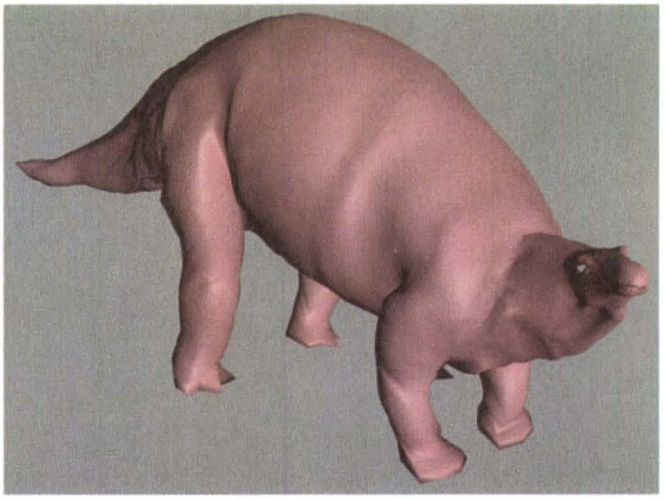
The Uniform System code to accomplish this involves calling the generator as shown below:

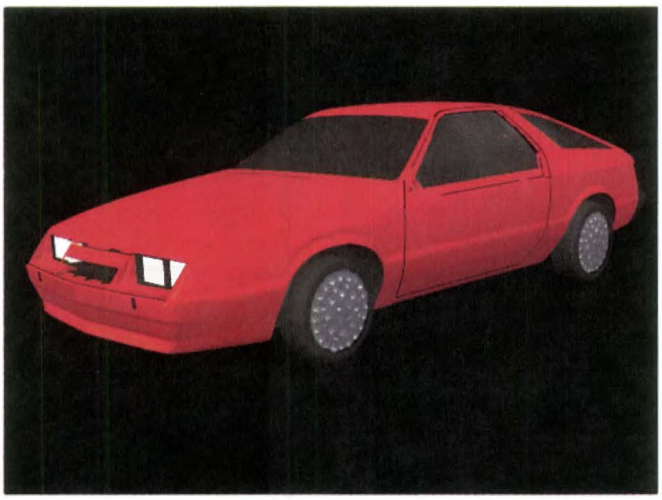
```
GenOnI(do_work, range);
```

One processor calls the generator GenOnI, and then all processors request work from this generator. The non-deterministic nature of dynamic scheduling insures that processors are assigned individual iterations of the ***for*** loop and execute the routine do_work for their given iteration. The Uniform System provides other generators which can generate tasks in a more complex manner; this example just illustrates how a simple one works.

The Uniform System generator mechanism is fairly efficient and easy to use, but there do exist some overheads. The real overhead in scheduling that occurs in parallel programs is the entering of the critical section for obtaining the next task. In this case, scheduling tasks consists of incrementing the shared loop index and assigning the next task to an available processor. Using the Uniform System,







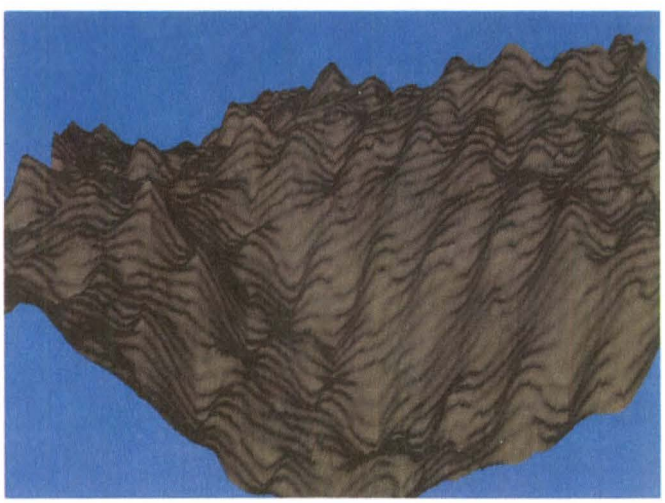


Plate 1. Test images for parallel programs: tree, stegosaurus, Laser, and mountain.

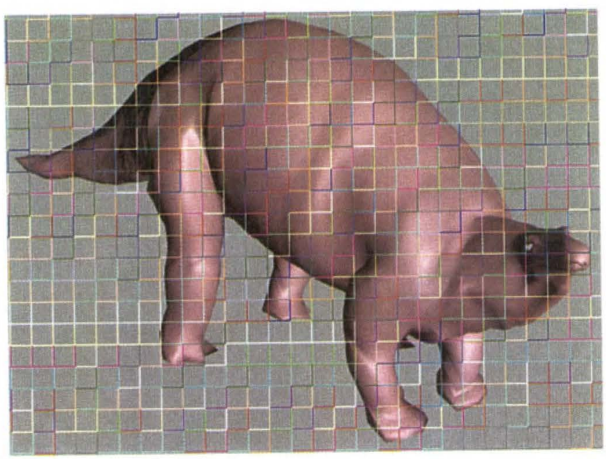


Plate 2a. Rectangular, data non-adaptive decomposition scheme.

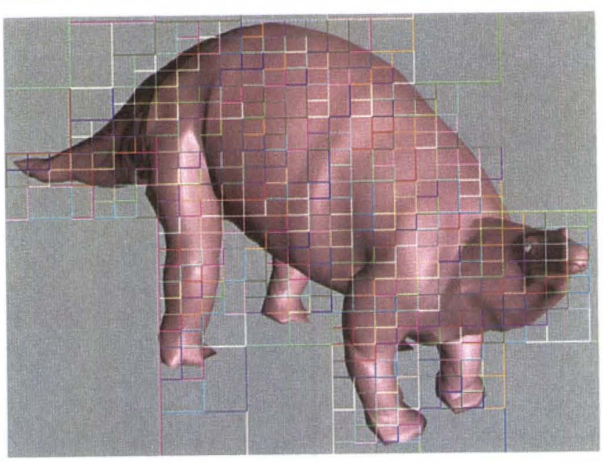


Plate 2b. Top-down, data adaptive decomposition scheme.

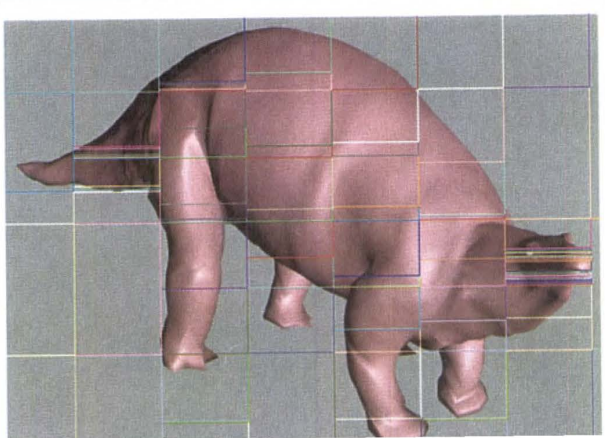


Plate 2c. Task adaptive

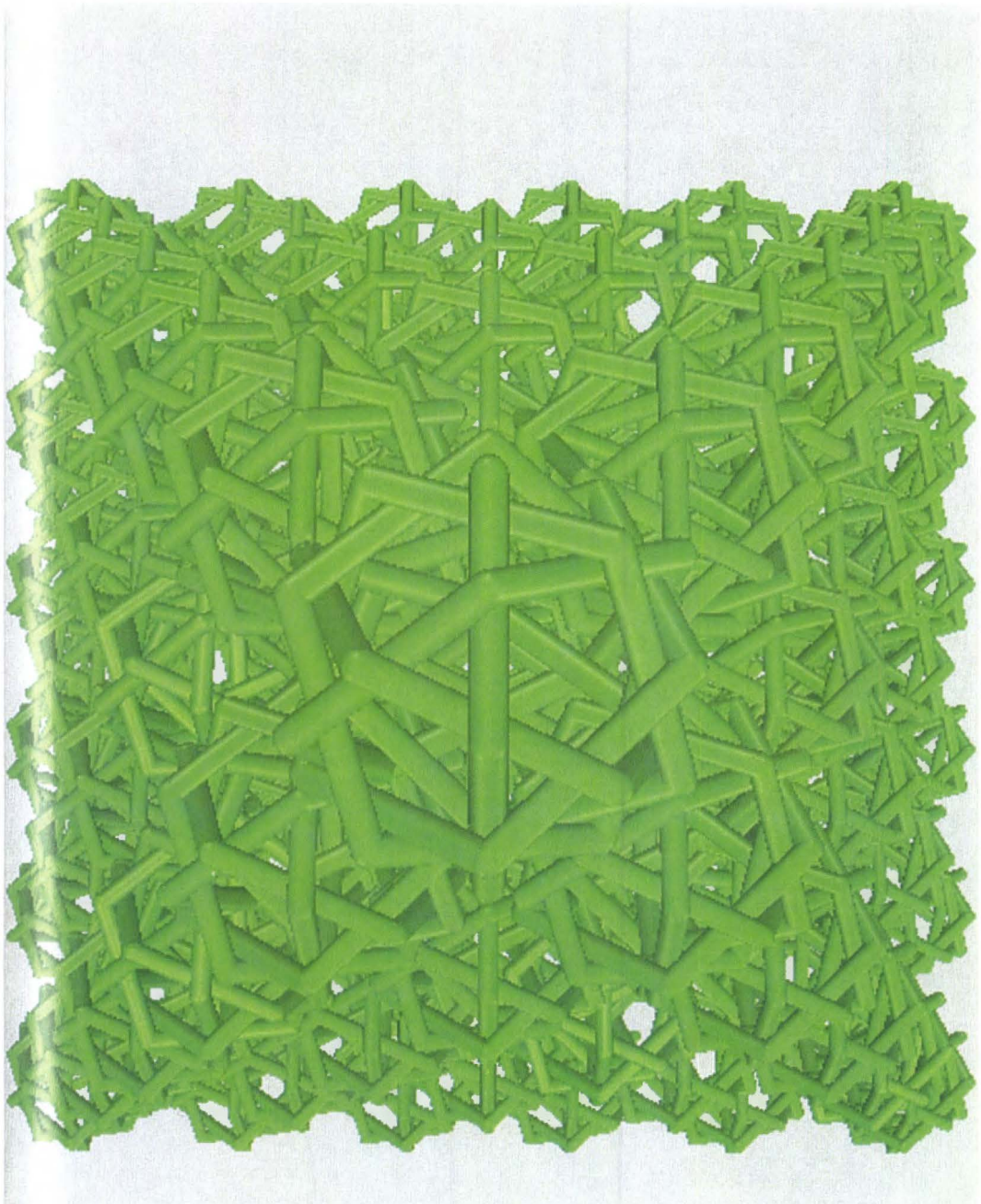
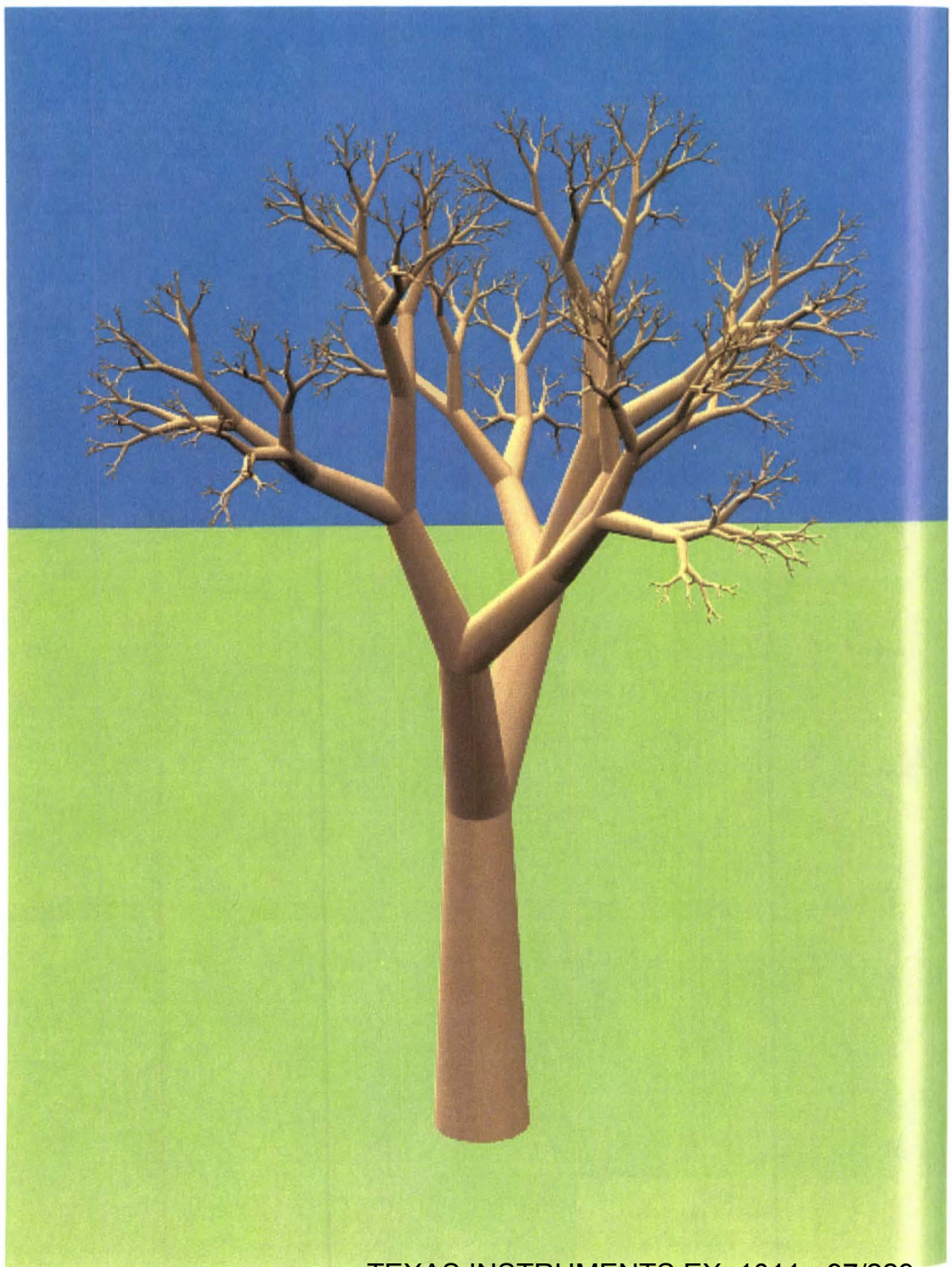


Plate 3. Rings image.



the time has been measured as 24 μsec on the GP1000 for a single task (call this T_{crit}). The scheduling is an inherent sequential process, so a task must take longer than $P * T_{crit}$ for a bottleneck due to scheduling to be avoided. On 96 processors, the minimum task time which would limit bottlenecks due to scheduling needs to be at least 2.3 msec (denoted T_{sched}). The scheduling overhead for the various images is the total time devoted to scheduling tasks divided by the total parallel execution time, as shown in equation 4.5.

$$\text{Scheduling } \% = \frac{\sum_{i=0}^{P-1} i \bullet T_{crit} + (N - P) \bullet T_{crit}}{T_p \bullet P} * 100\ \% \tag{4.5}$$

For the ith processor to start working, it has to wait for $(i - 1)$ tasks to be scheduled before it. It therefore takes a total time of $(i \bullet T_{crit})$ to be scheduled. Summing this up for all processors yields the first term in the numerator in equation 4.5. The second term is the additional scheduling time required for each of the $(N - P)$ tasks left after the first one is scheduled for each processor (N is the total number of tasks to be scheduled) assuming that each task takes more than $(P - 1) \bullet T_{crit}$ time to execute. Thus, a key factor in evaluating scheduling overhead involves making sure that no bottleneck results from a task taking less than $(P - 1) \bullet T_{crit}$ to execute. For a graphics algorithm, the minimum task time is based on sending the background color to the virtual frame buffer. Therefore, this time is measured to see if there is potentially a bottleneck. The denominator is the total processor-time space where T_p is the ending parallel execution time on the maximum processor configuration.

4.3.2.2. Memory Latency

Memory latency in a multiprocessor refers to the extra time required to send (write) or send and receive (read) a request for/from remote memory. This time is somewhat dependent on the number of switch nodes the message must travel through, but since we are dealing with a 4 column switch on the test GP1000, the times measured are as follows: a 4 byte read takes 7 μsec, while a 4 byte write takes 4 μsec. In contrast, a local read takes 0.53 μsec, while a local write takes 0.38 μsec.

Memory latency can be measured by counting the number of remote references during program execution and adding up the additional time of remote versus local referencing time required for

all of those references. This is done by using one of the Uniform System library calls which can detect if a given shared memory reference is a reference to a local or a remote memory module. The memory latency overhead (latency %) is determined by using equation 4.6.

$$\text{Latency } \% = \frac{\left[\#\ refs\ *\ (T_{rref}(P) - T_{lref}(P))\right]}{T_p \bullet P} * 100\ \% \tag{4.6}$$

In the equation, *#refs* is the average number of remote references per task, T_{rref} refers to the remote reference time, T_{lref} refers to the local reference time, and, as before, *Tp* is the ending time in parallel on the maximum processor configuration.

4.3.2.3. Communication

Although any kind of message traffic across the network could be termed *communication*, with regard to the BBN Butterfly, we are specifically referring to messages which are initiated using the machine's block transfer mechanism. This mechanism is built into the hardware of the GP1000 switch and allows a message path to stay open as long as necessary in order to get the message through. This requires a one time setup cost for the message of 8 µsec (T_{setup}) plus a cost of 0.25 µsec per byte transferred (T_{bt}) [BBN89b]. To alleviate blocking in the switch, block transfers are limited to 256 bytes and a software mechanism is provided to allow the programmer to use block transfers of longer messages. Since this cost is incurred once for a block of data and thereafter the data element is referenced locally, it is prudent to use block transfers if data can be partitioned into contiguous chunks. The cost of transferring these messages is the communication overhead but after this is taken into account, memory latency is no longer a factor since the data is available locally.

This overhead factor is derived by measuring the total number of bytes transferred in the system. Using the data for block transfer time for this number of bytes, taking into account that each message is a maximum of 256 bytes long, the total communication time is derived and shown in the numerator of equation 4.7.

$$\text{Comm. Overhead } \% = \frac{(\frac{\#\ bytes}{256} \bullet T_{setup}) + (\#\ bytes \bullet T_{bt})}{T_p \bullet P} * 100\ \% \tag{4.7}$$

4.3.2.4. Overhead due to Adaptation for Parallelism

It is generally necessary to either modify a serial algorithm with parallel constructs or develop an entirely new parallel approach to run a program on a multiprocessor. There is an inherent overhead built into this new parallel algorithm which is not present in the serial algorithm. New constructs and task setup instructions used to achieve parallelism are required for the parallel implementation. For numerical parallel algorithms, this overhead is typically very small. For a graphics display algorithm, one of the benefits of executing in a sequential fashion is compromised: graphical coherence. When a large number of tasks is required, such as on 96 processors, the loss due to coherence may become a significant factor. The overhead due to adaptation for parallelism is measured as part of the total processor-time space as well; it primarily consists of loss due to the lack of coherence. The other portion of this overhead involves the additional setup costs for each task prior to its execution on a processor.

The overhead due to lack of coherence is directly related to the number of tasks in most cases. That is, if more tasks are used, more coherence is lost. Since this overhead is measured at the maximum processor configuration (which represents the most tasks for a particular algorithm), the loss due to coherence is highest and this represents the worst case scenario.

Different algorithms may have different amounts of this overhead. This fact should be taken into account when looking at the speedup graphs of a parallel algorithm. If the speedup curve is nearly linear but the overhead due to adaptation is large, the performance of a given algorithm may not be good in comparison to other algorithms if the other parallel algorithms have smaller overheads. If the speedup in an algorithm with a small loss due to coherence is better than another algorithms' speedup, it will eventually provide better parallel performance as P increases.

The overhead due to adaptation, or more accurately, the code modification overhead, can be measured in the following manner. A sequential algorithm with no code modification overhead is analogous to the situation of $T_{MIN}(1)$. Recall that $T_x(y)$ refers to a situation where x is the number of memory modules and y is the number of processors. If more than one area is used, the number of areas is subscripted afterward so that $T_x(y)_r$ refers to using r areas with x memory modules on y processors. The actual amount of work on P processors is thus $T_P(P)_{R \bullet P}$, where R is the granularity ratio ($R = \#tasks/P$). To simulate this work without any contention effects,

$T_{MIN}(1)_{R \bullet P}$ is measured since this gives the time overhead and if the communication cost is deducted, the actual computation cost is derived. The same is done for an essentially sequential task, which corresponds to $T_{MIN}(1)_1$. Subtracting the difference between these two values results in the exact extra work which is involved in setting up the tasks necessary for parallel execution. For instance, the coherence lost in both the vertical and horizontal directions is inherently included in this value. Equation 4.8 is used to measure this overhead based on the work assigned to P processors.

$$\text{Code Mod. Overhead } \% = \frac{T_{MIN}(1)_{R \bullet P} - T_{MIN}(1)_1}{T_p \bullet P} * 100\ \% \tag{4.8}$$

4.3.2.5. Synchronization

Arvind and Ianucci [Arvi86] identify several basic synchronization situations:

a) *Producer-Consumer* - a data structure is produced by a given task to be used by another task on another processor. In order to insure that the consumer waits for the producer, synchronization must occur.

b) *Fork and Join* - a *join* operation indicates that two or more tasks have completed from a previous fork. To implement the join, a synchronization event must occur. The fork operation is basically a scheduling overhead, as discussed previously.

c) *Mutual Exclusion* - when two or more parallel tasks wish to execute a given region that only one is allowed in, this presents a critical section of the code which requires synchronization.

In the algorithms presented here, these situations do not come up that frequently in the tiling portion of the programs. The producer-consumer situation does occur in one algorithm but for the most part, the data structures used by the algorithms are mutually readable and very few are read-write. The overhead due to synchronization is measured as follows. The time in which a processor waits at a semaphore lock is summed up throughout the system. This time is indicated in the numerator in equation 4.9. The denominator is the total processor-time space as before.

$$\text{Synchronization \%} = \frac{\sum_{i=1}^{\#tasks} T^{i}_{synch}}{T_p \bullet P} * 100\ \% \tag{4.9}$$

In the equation, T^{i}_{synch} refers to the synchronization wait time for task i. The fork operation occurs at the beginning of a generator, while the join operation occurs at the end of a generator. These factors are measured as part of the scheduling overhead and thus will not be a part of the synchronization measurement. Mutual exclusion is used for serial sections in the programs, but the measurements given here ignore the small serial sections at the beginning of the program.

4.3.2.6. Network Contention

Network contention refers to the slowdown incurred when more than one message attempts to use the same switch node at the same time. This can be categorized as the probability that a memory request will block at a switch node due to another message already using the given path. BBN refers to this phenomenon as switch contention in their literature. Tree saturation [Kuma86] will not occur since the Butterfly interconnect is a non-blocking network. That is, the Butterfly switch forces uncompleted messages to retreat back to their source rather than buffer-up behind a blocked switch node. An alternate route is then tried for the message after some random delay time. The amount of time taken to serve a given request will increase as more messages enter the network since it may take longer to find a free path in this situation. In fact, if we distribute the dataset uniformly throughout the processor memories, this probability increases non-linearly as P is increased since the number of switch paths in the Butterfly grows as $log(P)$ while the number of processors grows linearly.

Network contention in the different algorithms is measured in the following manner. First, the time for task i is measured with P active processors (in this case, $P = 96$). Then, the time for task i is measured with only a single processor active, but using the memory of MIN processors (so no paging occurs). The time difference between the two scenarios is a result of contention in the P processor case, but no contention in the single processor case. Latency and/or communication costs are factored out of the times in each situation. Equation 4.10 shows the formula for computing network contention. The superscripted i refers to a particular task i.

$$\text{Switch Contention \%} = \frac{\sum_{i=1}^{\#tasks} \left[T(P)^i - T(1)^i\right]}{T_p \bullet P} * 100\ \% \tag{4.10}$$

Another form of contention which is more specific to a particular switch location is called *hot spot contention*. This occurs when a disproportionately large amount of references are aimed at either: 1) a particular memory location or 2) a particular switch node. The first situation can usually be rectified by copying this data item to all memory modules in the system except in situations in which the data item is writable where there would be no solution. This copying involves the use of the network which can contribute to other contention problems. The second situation is less easily identifiable, but scattering of the shared data structures uniformly throughout the network is the usual way to solve this problem. Both types of solutions are used when possible in the algorithms described in this book and no adverse effects due to hot spot contention were noticed.

4.3.2.7. Load Imbalance

Load balancing is the primary focus of most designers of parallel programs. It is usually desirable for all processors to finish working on a problem at the same time so that none are left idle while others are busy. This is almost impossible to achieve in general practice, though. The idle time delay in which processors wait until all tasks are finished is due to load imbalance. Any solution used to solve this problem should take into account the performance of the given machine with regard to scheduling time, its CPU speed, and of course parallel program decomposition.

In measuring the contribution of load imbalance in each algorithm, the finishing times of each processor are noted. The average of these finishing times is the theoretical ideal finishing time if load balancing is perfect. To calculate the percentage overhead, the difference in time between the last processor's finishing time and the average of all processors is recorded. This difference is used as the numerator for calculating the load imbalance percentage, as shown in equation 4.11. This is essentially the same as adding up the total idle time at the end of the computation for all processors in the system.

$$\text{Load Imbalance \%} = \frac{T_{max} \bullet P - T_{avg} \bullet P}{T_p \bullet P} * 100\ \% \tag{4.11}$$

It is very difficult in reality to isolate how well a particular algorithm is load balanced. A problem with equation 4.11 is that it does not delete the effects of network contention. In fact, it may turn out that this overhead percentage is artificially increased or decreased in a situation where contention is significant. Although we cannot isolate how load balancing would be treated in each instance if contention is not present, the load balancing percentage determined by equation 4.11is a rough indication of this overhead factor. This allows direct comparisons to be made between the different algorithms by using a common measurement parameter.

4.4. Summary

In this chapter, we present the serial algorithm on which all of the parallel algorithms are based. A detailed description is provided of each of the important phases of the program, elaborating upon the data structures which are common to all the implementations in their use of shared memory. Information is given about the test scenes used for timing purposes. Next, methods describing performance analysis for the parallel programs are elaborated upon. The traditional performance measurements of time, speedup, and efficiency as well as other overhead factors are described.

In the next chapter, these additional overhead factors are quantitatively presented in an analysis of parallel graphics display algorithms to show where the performance degradation actually occurs. In addition, the different parallel partitioning schemes identified in chapter 2 were implemented on the Butterfly, and their results are presented. An analysis of their performance is included in regard to: the execution time of the tiling section, speedup, and effect of the overhead factors.

5

Comparison of Task Partitioning Schemes

In this chapter, we describe a number of parallel decomposition schemes and our implementations of these schemes on the BBN Butterfly GP1000. In these algorithms, tasks are assigned to regions of image space, but there are a number of different ways of determining the size and number of these regions. Task partitioning can be divided into two main techniques which are discussed in this chapter: data non-adaptive and data adaptive. In the data non-adaptive method, tasks are determined without regard to the input data set. In the data adaptive approach, the number and size of tasks are based on the input dataset. The data non-adaptive partitioning scheme relies on dynamic scheduling of tasks onto processors. These tasks are determined in a simple manner, so little overhead is needed prior to tiling. Load balancing is achieved by creating enough tasks so that the tasks left to work on at the end of the computation are fairly small. Data adaptive partitioning in a graphics context involves creating tasks based on the location of the data elements on the screen. The basic idea in this method is that tasks are chosen prior to tiling, so that each task takes approximately the same amount of time to finish. Extra work is required to set up these tasks prior to tiling

but the benefit of this is reduced scheduling overhead. For each of these methods, one can assign a number of tasks equal to the number of processors ($T = P$) or greater than the number of processors ($T > P$). This is illustrated in figure 5.1. There is also an important extension to the data non-adaptive technique known as *task adaptive*.

Each section in this chapter describes a different task partitioning scheme in detail. The algorithm implementations which are discussed in chapter 2 are presented in detail here and categorized according to their task partitioning scheme. Each implementation is then evaluated according to the parallel program measurements of time and speedup. Then, the overhead factors of scheduling, memory latency, communication, network contention, load imbalance, overhead due to adaptation for parallel execution, and synchronization are quantified for each algorithm. The results from this analysis help form a basis for comparison of all of the implemented approaches. The values reported in this chapter pertain only to the tiling section of the parallel programs. In the next chapter, a comparative analysis of the operations required prior to tiling is included along with the time of the tiling section.

The various schemes are described in the following sections. We analyze the implementations with regard to the issues discussed in the previous chapters. The task partitioning schemes as discussed in this chapter are given in the following order: data non-adaptive approach, data adaptive approach, and task adaptive approach.

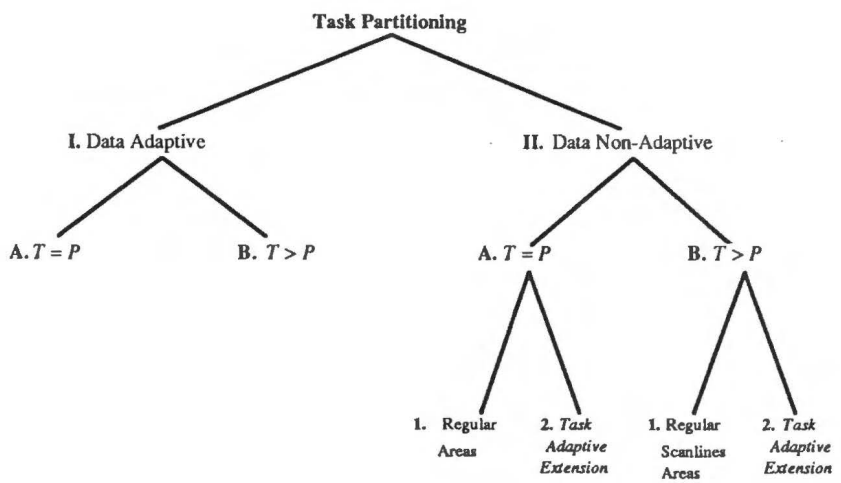


Figure 5.1: Task partitioning techniques

5.1. Data Non-Adaptive Partitioning Scheme

The data non-adaptive partitioning scheme relies on subdividing the image space regardless of the screen location of the polygon datasets. When this method is implemented under the Uniform System, it employs a dynamic scheduling mechanism whereby tasks are scheduled onto processors as each processor is available for work. Each task is a single region or area; a single scan line is a task in the first method, and a small rectangular region is a task in the second method. The granularity ratio, which is the ratio of total areas to the number of processors (R=#*tasks*/P), must be chosen carefully since it can have a significant impact on load imbalance and execution overheads. We will see how this value affects the overall performance in the decomposition schemes outlined below.

In this partitioning method, image space is broken up into a number of rectangular areas, all the same size. The two methods from chapter 2 which were implemented are Hu and Foley's dynamic scan line scheme and the rectangular area scheme suggested by Kaplan and Greenberg, as well as Whelan.

5.1.1. Scan line Decomposition

A scan line decomposition is probably the most natural parallel partitioning scheme. It was first suggested by Hu and Foley in their paper describing a hardware parallel rendering machine [Hu85]. The basic idea involves partitioning the image space such that each scan line is a task by itself (T=#*scan lines*). The granularity ratio R varies as P is increased since T is constant for all values of P. The algorithm has less flexibility than if T could be increased with P since load balancing is limited as this number is approached by P. Some details regarding the implementation of this decomposition in parallel are given here.

As in a serial scan line algorithm, a y-bucket list is used to store the polygons relevant to a particular scan line. Additional work is required in constructing the y-bucket data structure for a parallel implementation because *all* polygons relevant to that scan line are stored in a given y-bucket, not just those that start on a given scan line. The set of operations necessary to construct this shared data structure is performed in the front end prior to tiling. In reference to a conventional serial algorithm, extra memory is required for the y-bucket list in addition to the extra time necessary to store the data. These extra requirements are small when compared with the benefit gained through parallel processing.

The storage and access for the *y*-bucket list is accomplished in the following manner. Only one processor works on a given scan line, so no synchronization is necessary for extracting the polygons from this data structure. The *y*-bucket list is stored as an array corresponding to the number of scan lines, where each element of the array is a pointer to a linked list of the polygons relevant to that scan line (a single *y*-bucket). Since all processors need to reference this array of pointers, it is copied to each processor's local memory to avoid hot spot contention. Prior to tiling, the links for each scan line are loaded into the *y*-bucket data structure and scattered throughout the memory modules; thus there is only one copy of each link. This achieves a uniform distribution of the polygonal dataset, without adding contention. The *y*-bucket array is read-only in the tiling phase of the program, which is why it can be copied to all the processors. The links could also be copied, but the time to do so and the memory required make this inefficient.

Hu and Foley's research showed that dynamic assignment of single scan lines to processors resulted in better performance than interleaving groups of successive scan lines statically. The reason the dynamic technique was superior was that it minimized load imbalance, and this had a greater impact on performance than maximizing coherence in a group of contiguous scan lines. This dynamic assignment method was implemented and tested on the Butterfly to evaluate the algorithm on a real machine. A graph of the times for each of the test images is given in figure 5.2.

It is important to note that Hu and Foley achieved their results from simulation data rather than from an actual implementation. In addition, their design was intended for hardware implementation and required all the data to be present in each processor, while we are using a more flexible memory model in a software algorithm. Consequently, issues like remote memory referencing come into play in this implementation, whereas Hu and Foley did not analyze their algorithm with regard to these issues. The results given here compare favorably with their results, although different test cases were used and exact speedup was not recorded in their paper. The relative speedup for the images is shown in figure 5.3.

The equation to calculate speedup is known as *Amdahl's law* and is shown in equation 5.1 where P is the maximum number of processors used (in this case, $P = 96$). As we explained in the previous chapter, all the tests were started above one processor, so the actual speedup must be estimated. The estimated speedup is derived by using $T_{MIN}(1)$ in the numerator of equation 5.1. This value refers to the fact that the program is run on one processor using MIN memory

modules with a deduction for the communication cost. This was shown in equation 4.3 in the previous chapter.

$$\text{Speedup} = \frac{T(1)}{T(P)} \tag{5.1}$$

We will now discuss the various issues associated with parallel computation, described in section 4.3.2 for this algorithm. As a guide to the reader, the range of percentage contributions for each overhead based on the minimum and maximum overhead of the four test images is included in parentheses after the section title. The overhead factors determined are based on the percentage of the processor-time space using the equations given in chapter 4. The appendix contains all of the results from the tests used to determine these overheads.

In the previous chapter, a description is given of how each overhead factor is actually measured. Due to the variance in the way each algorithm works, some changes in the way these overheads are measured is required. These changes are described as necessary here.

All of the tests were run on a maximum of 96 processors. Since this maximum value of P represents the worst case scenario in relation to the overhead factors, these factors are evaluated at $P = 96$.

5.1.1.1. Scheduling (0.002% - 0.01%)

Scheduling in the parallel scan line algorithm proceeds by calling a generator procedure which is provided by the Uniform System. It is equivalent to a parallel *for* loop based on the number of scan lines in the final image. Processors extract iterations from the generator as each processor is available to work on a task. The minimum time that it takes to render a scan line occurs when no polygons are present and only the background color is displayed. The average time to calculate background for a scan line of 640 pixels and then write the scan line to the virtual frame buffer is $T_{back} = 2.0$ msec.

As stated in section 4.3.2.1, the time to schedule the first task on each of the 96 processors is 96 * T_{crit} (the critical region time T_{crit} = 24 μsec) which is 2.3 msec and is denoted as T_{sched}. T_{sched} is the total serial scheduling time overhead. It is slightly larger than T_{back} (the background color rendering time), so it is possible to create a bottleneck if a very high proportion of the tasks to be executed are background tasks.

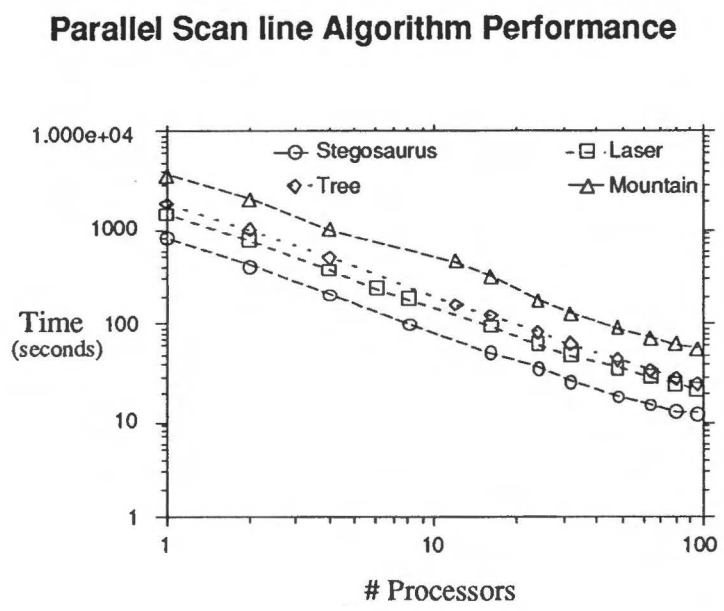


Figure 5.2: Scan line data non-adaptive performance

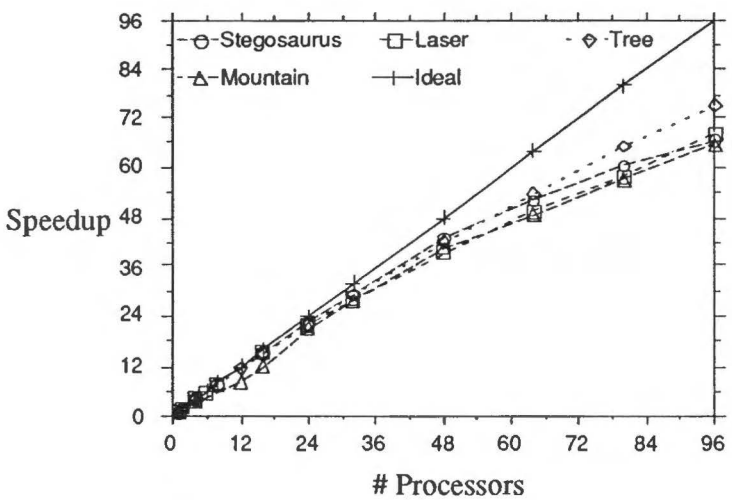


Figure 5.3: Speedup for scan line data non-adaptive algorithm

This bottleneck is unlikely to occur in practical use and would only degrade overall performance by a fraction of a millisecond if it should occur. Using equation 4.5, we plug in values of $P = 96$, $N = 484$, and $T_{crit} = 24$ μsec as shown in equation 5.2. Note that $(P * (P - 1))/2$ can be substituted for the summation in equation 4.5. T_p is the parallel execution time on 96 processors for each image.

$$\text{Scheduling \%} = \frac{\frac{(95 \bullet 96)}{2} \bullet 24\ \mu\text{sec} + (484 - 96) \bullet 24\ \mu\text{sec}}{T_p \bullet 96} \tag{5.2}$$

The overhead due to scheduling in the parallel scan line algorithm ranges from 0.002% for the mountain image to 0.01% for the stegosaurus image.

5.1.1.2. Memory Latency (3.0% - 5.6%)

Recall that memory latency is the additional time delay incurred when a reference is made to a remote rather than a local memory module. The latency is calculated using equation 4.6, by using the total number of remote references during the computation to determine the extra time spent in accessing non-local data.

We have calculated the latency overhead percentage for the parallel scan line algorithm at 96 processors for the various images to be a minimum of 3.0% for the stegosaurus image and up to the maximum of 5.6% for the mountain image. Although it might be expected that the larger datasets require more remote references, it is interesting to note that the percentage overhead due to latency also increases with dataset size. In other words, even though the larger datasets require more execution time, the latency requires an even greater percentage of this time. This suggests that latency might become a major degradation factor for particularly large input datasets.

5.1.1.3. Network Contention (8.9% - 23.1%)

Network contention is a function of the probability that a conflict will occur in the interconnection network for a particular memory reference. As P is increased, the likelihood of a blocked network path increases since the number of remote references is proportional to P^2 while the number of switch paths only increases by $P \bullet \log(P)$. In the appendix, the network contention is quoted as two percentage values. The first value, denoted "% of Total-Processor Time Space," is measured using equation 4.10, as given in the previous chapter. The second value is calculated as described next.

We assume that the measurements used for load imbalance, memory latency, communication, code modification, and scheduling are all somewhat accurate. This is a reasonable assumption since with the exception of load imbalance, all of these overheads can be measured independently from the others. Load balancing is affected by all of the overhead values, but this cannot be avoided in normal timings or in specialized performance measurement situations. The value given for load imbalance is probably a culmination of other factors as well. Network contention is a completely separate matter. Although the measurement technique used for this culprit should be somewhat indicative of the effect of this overhead, the method given in chapter 4 does not involve a true measurement of the actual network contention. Doing so would require hardware monitoring which can only be done by the manufacturer. Therefore, the assumption given above is used to help estimate the actual network contention. This is done by subtracting the sequential time and overheads, which are assumed to be accurate from the total processor-time space, as is shown in equation 5.3.

$$\text{Contention} = T_p \bullet P - \left[\begin{array}{c} T_{MIN}(1) + \text{Code Mod.} + \text{Latency/Comm.} + \\ \text{Load Imbal.} + \text{Synch.} + \text{Sched.} \end{array} \right] \quad (5.3)$$

In other words, we assume that the total of the sequential time plus all overhead factors is exactly the parallel execution time. Therefore, if all other overheads are deemed to be accurately measured, then the only overhead left is contention. In most cases, our measured value of contention using equation 4.10 and the calculated value of contention did not differ by a large amount, meaning that the measurement technique is fairly reasonable. This can be seen in the values given in the appendix. This calculated value for contention is given in the header of this subsection and the other algorithms' subsections as well.

Since this algorithm requires a large amount of remote references, as shown above, it is likely that the contention is fairly high as well. The results from the tests bear this out. Based on the scan line algorithm overhead measurements, the calculated network contention ranges from 8.9% for the tree image to 23.1% for the stegosaurus image.

5.1.1.4. Load Imbalance (6.8% - 10.4%)

It is hard to obtain good load balancing in this task partitioning scheme since the number of processors comes close to the number of total tasks. In this case, 96 processors and 484 scan lines provide

approximately a 5 to 1 ratio of tasks to processors. Since it is entirely possible that any given task will take more than 5 times longer than another task, it is possible that load balancing will not be adequate with this number of tasks. For 96 processors, the load imbalance for the test images has been measured from 6.8% for the mountain image to 10.4% for the laser image.

5.1.1.5. Code Modification (4.9% - 9.8%)

Overhead due to parallel processing is fairly significant in this algorithm. The main contributor to the overhead in this rendering algorithm is the loss of coherence incurred by starting a new scan line in parallel rather than continuing execution on the same processor. While this factor is constant (since the number of tasks is constant regardless of the number of processors used) and will not affect the speedup of the parallel algorithm, it can be used to determine the relative performance of this algorithm versus the other parallel algorithms. The overall percentage effect due to code modification varies from 4.9% for the tree image to 9.8% for the mountain image.

5.1.1.6. Explanation of Results

The two primary contributors to performance degradation in this algorithm include overhead due to code modification and network contention. Memory latency and load imbalance also degrade overall performance, although to a lesser degree. The percentages for each of the major overhead factors as related to each test image are given in figure 5.4. The effects of scheduling are so minimal in comparison to the other factors that it is not worth consideration as a problem area here.

Recall from the previous chapter that the dataset sizes are as follows:

1. Stegosaurus	9.7K polygons
2. Laser	46.3K polygons
3. Tree	106.4K polygons
4. Mountain	131.1K polygons

As one can see from the figure, latency increases with dataset size. The overhead measured for load imbalance is stable, although it is reduced slightly for the mountain image. Since the mountain data is more uniformly spread across the screen, this may be the reason that load balancing is better for this image than the others.

It seems remarkable that even with an average of only 5 tasks per processor using test cases in which the data is not uniformly distributed, the load imbalance is less than 10% for most of the test images. It is certainly true, however, that if the number of processors were to be increased significantly beyond 96, load balancing would suffer due to a reduction in the number of tasks available for parallel execution. This provides a motivation to seek algorithms which are better able to handle large processor configurations as well as variable size resolution images.

Lack of vertical scan line coherence is the primary contributor to the overhead in adapting this algorithm for parallel processing. This is manifest as the total degradation due to code modification. The code modification overhead is less than 10% for all the test images. The actual time due to code modification is invariant to the number of processors in the system. Unfortunately, as P is increased significantly, load balancing tends to suffer to a large degree in this algorithm. A better algorithmic solution would be one which does not have this overhead effect.

Network contention is a major contributor to performance degradation, and it increases as a function of the number of

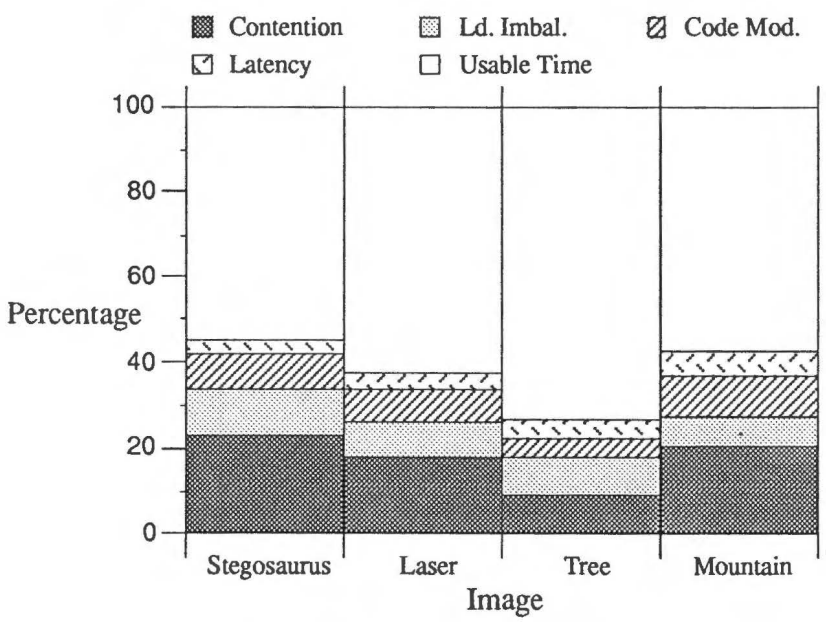


Figure 5.4: Degradation factors for scan line decomposition for (P = 96)