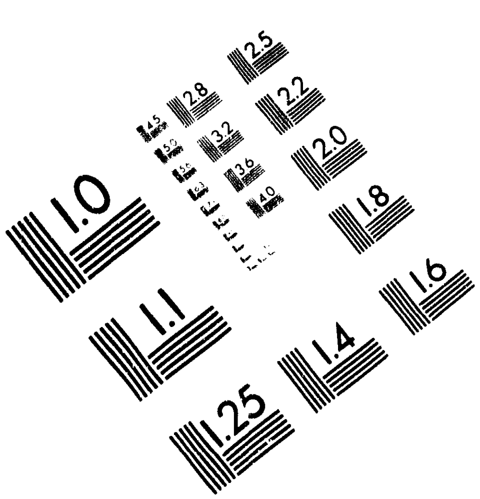
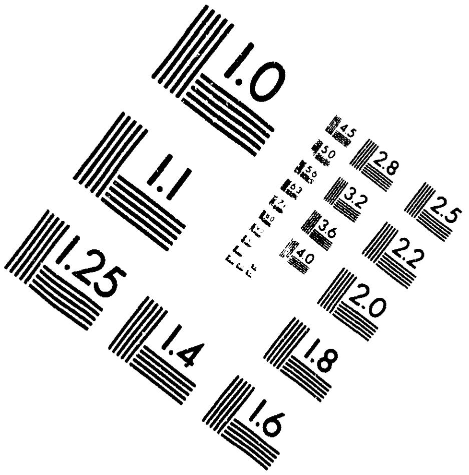
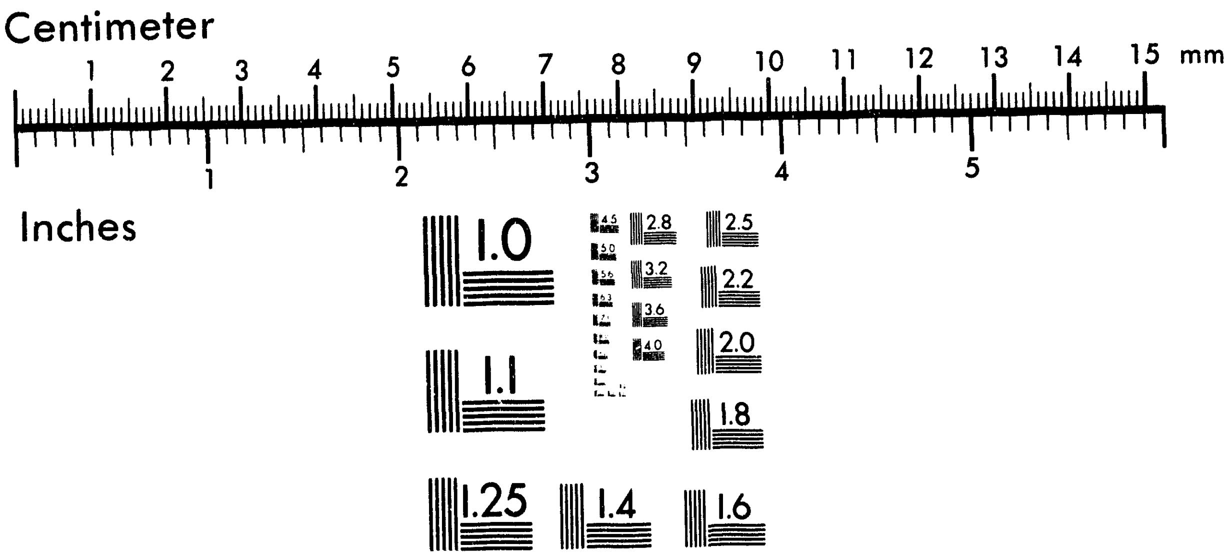
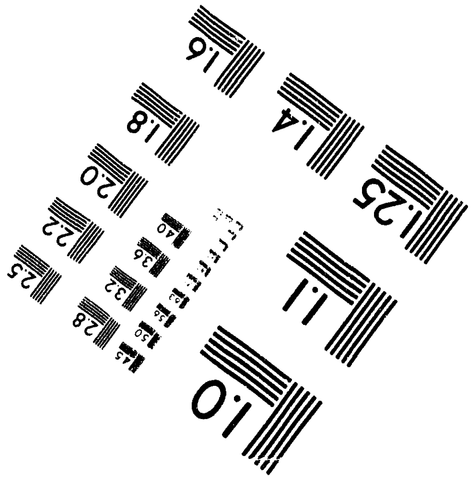
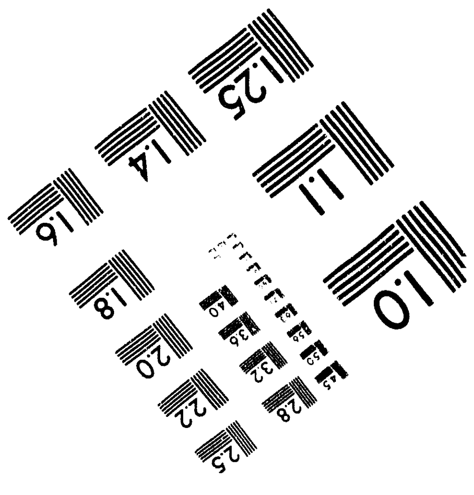





Centimeter
mm
Inches





√

Conf-930881--3

UCRL- JC-112572
PREPRINT

A Task Adaptive Parallel Graphics Renderer

Scott Whitman

This paper was prepared for submittal to
ACM Siggraph '93
Anaheim, CA
August 1-6, 1993

December 21, 1992

Lawrence Livermore National Laboratory

A Task Adaptive Parallel Graphics Renderer

Scott Whitman *
Lawrence Livermore National Lab
P.O. Box 808, L-301
Livermore, CA 94550

ABSTRACT

This paper presents a graphics renderer which incorporates new partitioning methodologies of memory and work for efficient execution on a parallel computer. The Task Adaptive domain decomposition scheme is an image space method involving dynamic partitioning of rectangular pixel area tasks. We show that this method requires little overhead, allows coherence within a parallel context, handles worst case scenarios effectively, and executes efficiently with little processor synchronization necessary. Previous research in the area of memory and work decompositions for graphics rendering has been primarily limited to simulation studies and little practical experience. The algorithm presented here has been implemented on a scalable distributed memory multiprocessor and tested on a variety of input scenes. We present a theoretical and practical analysis in order to contrast its predicted and actual success. The implementation analysis indicates that load imbalance is the major cause of performance degradation at the higher processor counts. Even so, on a variety of test scenes, an average rendering speedup of 79 was achieved utilizing 96 processors on the BBN TC2000 multiprocessor with a processor efficiency range of 66% to 94%.

CR Categories and Subject Descriptors: D.1.3 [**Concurrent Programming**]: Parallel Programming; I.3.3 [**Computer Graphics**]: Picture/Image Generation - Display algorithms; I.3.7 [**Computer Graphics**]: Three-Dimensional Graphics and Realism - Visible line/surface algorithms.

General Terms: Algorithms, Design, Performance

Additional Keywords and Phrases: Scan line, BBN Butterfly, parallel processing, three-dimensional, load balancing, SIMD, MIMD, shared memory

1 Introduction

Scientists, engineers, and other users of computer graphics may occasionally generate datasets containing hundreds of thousands of polygons. In order to aid their research process effectively, quick turn around of the rendering of these datasets is required. To this end, we have developed a general purpose polygon rendering algorithm which is suitable for implementation on distributed memory parallel computers. The discussion here will focus on the design and implementation of a parallel polygon based scan conversion renderer. This algorithm is intended to support fast rendering of highly complex datasets rather than real-time update of moderately complex scenes. We show that good speedup and parallel efficiency can be obtained with only a small overhead when compared to an optimized serial renderer. Several advantages to using a software based approach include the feasibility of adding special rendering features to the program and the capability to allow integration of a parallel scientific application with the graphics renderer on the same machine simultaneously. Using this latter approach on a parallel computer facilitates fast inter-processor communication of data rather than the slower external I/O typically used in post-processing graphics data.

*email: slim@llnl.gov

We present a new work decomposition strategy called *Task Adaptive* which is based on dynamically partitioning the amount of computational work left at a given time. The algorithm uses a heuristic in which image space tasks are partitioned without requiring the partitioned processor to be interrupted. We employ a sophisticated memory referencing strategy which is integrated into the Task Adaptive algorithm to allow local access to graphics data during the rendering process. The exact data which is needed for a particular graphics rendering task is copied to each processor without the need for remote referencing. This approach on a MIMD parallel computer allows one to obtain reasonable speedup as more processors attack the problem. The algorithm is also amenable to using either a shared or message passing programming paradigm. An in-depth analysis of the overheads accompanied with parallel processing is presented to find out where performance is adequate or could be improved. The analysis is from both a theoretical and practical point of view to understand the degradation factors as a function of number of processors (P). This solution for domain and data decomposition can also be used in other graphics rendering algorithms and may be suitable for hardware implementation.

The structure of the paper is as follows. Section 2 outlines previous work in the area of parallel graphics rendering. Section 3 describes the three main phases of the parallel display algorithm and section 4 discusses the complexity of this algorithm. Section 5 presents the strategies for storage of graphics data and section 6 gives performance results for several test datasets. A glossary is included in the appendix for readers unfamiliar with some of the terminology.

2 Historical Perspective

In the past 15 years, there have been numerous approaches to using parallelism in the tiling operation of polygon display algorithms. Much of this research has focused primarily on the domain (or work) decomposition of the rendering process. While there has been some work on parallel object space methods ([Abra86, Fran90] among others), the bulk of the algorithms have been of the image space variety. A taxonomy of these approaches appears in [Whit92] which are briefly summarized here. Polygon decompositions can involve independent tasks such as used by [Fium83] where span segments of a different polygons on a scan line are processed in parallel. Ghosal and Patnaik [Ghos86] assign each processor scan lines of a given polygon to scan convert at the same time. Allison [Alli91] uses a shared Z-buffer where the objects are processed in parallel and rendered to a common frame buffer. Crockett and Orloff [Croc91] alternate each processor on an Intel iPSC from computing the rendering of image space tasks to communicating polygonal data to balance the load effectively.

In using pixels as a basic building block for domain decomposition, researchers have devised the following tasks for parallel processing: horizontal strips (screen wide) of scan lines [Chan81, Hu85, Kapl79, Whel85], vertical strips (screen height) of pixels [Whel85], and rectangular areas of pixels [Croc91, Kapl79, Park80, Robl88, Whel85]. The solutions can be divided into the following categories: *Data Non-Adaptive* and *Data Adaptive*. The Data Non-Adaptive methodology relies on an initial decomposition of image space that is not related to the input data. The idea is that if many tasks of different work loads are assigned to different processors, the overall load will become balanced. In the Data Adaptive case, the size of the tasks (that is, the area of the pixel regions) are adjusted according to the input data in an attempt to obtain better load balancing. Data Adaptive schemes have been implemented by Whelan [Whel85], Roble [Robl88], and Whitman [Whit91] in the context of scan conversion methods, in addition to Dippe [Dipp84] in a ray tracing algorithm.

We have implemented a number of the aforementioned algorithms to determine their relative strengths and weaknesses. The polygon based approaches were found to be too limiting in the parallelism or usable context. In the Data Adaptive algorithms, it was found that creating nearly equal work tasks requires too much pre-processing time. Of the Data Non-Adaptive partitioning schemes (horizontal scan lines or groups

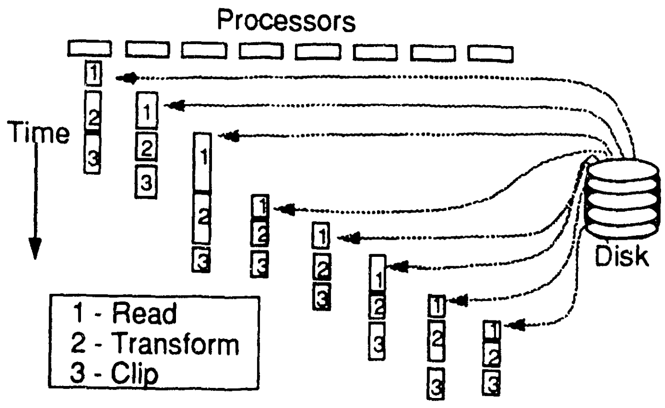


Figure 1: Pre-processing phase pipelined parallelism (task size is 1 object)

of scan lines, vertical strips, or rectangular areas of pixels), empirical test results indicate that the rectangular method works the best using a 1:1 ratio of sides since this layout minimizes the perimeter while maximizing overall coherence. The algorithm presented in this paper is based on this rectangular method but improves upon it by limiting overhead in pre-processing, handling worst case scenarios, and adaptively partitioning the work load. It was designed for fast rendering of animated as well as static input datasets.

3 Algorithm

In this section, we discuss the work decomposition strategy for the following three distinct phases of computer image synthesis. In the descriptions below, we include the average percent time used by each phase in a serial algorithm for a variety of test scenes.

1. *Pre-processing* (13 %) - this consists of data read-in, transformation of points, normals calculation, back-face rejection, clipping, and perspective projection.
2. *Rendering* (86 %) - includes hidden surface removal, shading, anti-aliasing, and any other visual effects.
3. *Post-processing* (1 %) - this involves displaying the image on a frame buffer or storage in a file.

The Task Adaptive domain decomposition is a variation on using rectangular areas of pixels as individual tasks. This algorithm uses less overhead than the rectangular approach in terms of pre-processing and later communication, however. The primary focus in this document will be on the rendering portion since this phase takes the bulk of the computation time. Some methods for parallelizing the first and third phases are given but implementations of these will generally be specific to the environment where the overall program is to run. The discussion will focus on a shared memory programming paradigm but specifics for a message passing implementation will be presented as well.

3.1 Pre-processing Phase

The implementation of the pre-processing phase depends to a large degree on the amount of parallelism available. If there are a large number of objects (i.e. $> P$, the number of processors), then each object can be read into a different processor's memory (see figure 1). Since each object may be a different size, the time to process a given object may vary as indicated by the size of the rectangles in the figure. Each

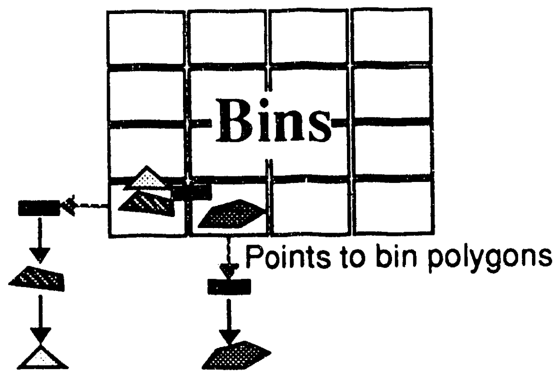


Figure 2: 2D array of bins containing polygon data

processor can then perform the transformations, clipping, etc. on the data in local memory. If the number of objects is instead $< P$, we then have two possible scenarios. In the first, if the user is animating these objects over a long period of time without a change in the objects' shapes, then each object can be split into multiple sub-objects which can then be processed independently. This provides enough parallelism to keep all of the processors busy. If the user is only creating a single image, then the cost of splitting the objects cannot be amortized. A possible parallelization in this case would be to independently process iterations of the individual loops for transforming or clipping the objects. This would work in a shared memory machine but not for a message passing architecture. Alternatively, a reader process could assign portions of the data to individual processors.

Another part of the pre-processing phase involves setting up the data for the domain decomposition so that each processor will have access to the data it needs for a particular task. Each initial task corresponds to a small area of the image space so the polygons are placed into *bins* (see figure 2) where a bin is associated with a particular task. The bounding box of the polygon is used to judge which bin(s) that polygon is placed into. The data structures for these bins is discussed in section 5. The average speedup for this section of the algorithm was 9.4 on 96 processors. The main limitation to further speedup is the sequential nature of the disk access for reading in data. If the data is coming directly from a simulation program running simultaneously on the same computer, this bottleneck would not exist.

3.2 Rendering Phase

As discussed previously, assigning rectangular regions of pixels as tasks seems to be a very reasonable solution for solving the rendering problem in parallel. Each task consists of a region of pixels for which the tiling problem is solved serially for those polygons which are present in the region. Here, we employ a modified scan line Z-buffer algorithm [Myer75] which uses stochastic sampling (16 samples/pixel)[Cook86] for anti-aliasing. The limitations of a basic rectangular assignment method is that a specific load balancing mechanism must be chosen so as to assign equal work to the processors. An example load balancing scheme is to divide the image space into $R \cdot P$ tasks which are dynamically assigned to processors (see figure 6 for a simple rectangular decomposition). R is the *granularity ratio* and must be chosen properly so as to minimize overhead and maximize load balance. The larger a value of R chosen, the more work involved in pre-processing, communication, and polygon duplication, but the better the load balancing. Conversely, smaller values of R result in less overhead but inadequate load balancing. The Task Adaptive approach is based on a rectangular decomposition and attempts to bridge this gap by using a small granularity ratio (in this case, $R = 2$ is used) to minimize overheads along with a dynamic load balancing scheme. A ratio of $R = 1$ could be used, but that presents a situation that requires more communication due to the load balancing mechanism utilized. After a processor has finished its first task, it attempts to dynamically retrieve additional tasks off the queue. The pseudo code similar to what each processor executes is shown below:

```
j = P
forall (i = 0; i < P; i++)    /* static scheduling */
    work_on_task(i);
while (j < (2 * P))           /* dynamic scheduling */
   work_on_task(atomic_add(j));
while (work_available > threshold)
   partition();               /* start partitioning */
```

This shows the code for a shared memory implementation where the shared variable j is atomically accessed. For a message passing implementation, one of the processors (or the host) could be designated as the controlling processor (CP) which would keep track of which processor is assigned a given task. The `partition` routine is executed so as to dynamically balance the load among the processors. Since each image space task differs in its amount of work, steps are taken to steal part of another processor's work when there aren't any initial tasks left. The adaptive nature of this work decomposition is outlined in the steps given below which are essentially part of the `partition` routine.

1. When a processor needs work (call this processor P_s), it searches among the other processors for the one which contains the most amount of work left to do (call this processor P_{max}). If there is sufficient work to do, proceed, otherwise return.
2. The P_s processor then sets a lock preventing any other processors from splitting P_{max} in addition to setting its own lock to prevent unwanted blocking of processors.
3. P_s partitions $P_{max}'s$ remaining work into two segments; the first goes to P_{max} and the second to P_s.
4. P_s then copies from P_{max} the data necessary for it to work on the second segment.
5. P_s unsets both its lock and $P_{max}'s$ and starts doing work. After completion of its work, P_s repeats these steps.

The terminology P_{max} refers to the processor index of the maximally loaded processor as determined by a particular P_s at a given time. The usage of the lock is to prevent more than one processor from splitting a given P_{max} at a time. The lock is implemented as part of the operating system and guarantees that one processor proceeds through it at a time and if it is not open, the processor waits until it opens. After P_s completes its new work, it calls `partition`. This routine is repeatedly called until there is no work available for splitting above a certain threshold. Based on empirical studies, splitting is still worthwhile even down to two scan lines so that is the threshold set. Since P_s splits the work of P_{max} that remains to be rendered into two vertical tasks, it makes sense for P_{max} to continue working on the upper task while P_s takes the lower one. This also allows coherence to be maintained in $P_{max}'s$ region without any additional overhead and it allows P_{max} to continue working on its task uninterrupted. Other splitting mechanisms were investigated (such as creating two side-by-side regions or even a combination of side-by side and top-down) but their performance was found to be inferior to the method outlined here. Figure 3 illustrates the splitting process.

Because the splitting process relies on work proceeding in a task in a top to bottom fashion, only scan line oriented hidden surface removal algorithms may be utilized as tasks. On the other hand, any scan line algorithm could be implemented since it does not matter what the attributes of the particular algorithm are. Since the initial task regions are split horizontally, the regions which have been split become further deviate from square. In order to find out the effect of the region size on splitting, we measured the performance for aspect ratios other than square (1:1). A horizontally oriented region such as what is produced with a 2:1 (2 pixels across for every 1 down) ratio resulted in poor performance overall. The following average percentage improvements over a 2:1 ratio were noted: **1:1 - 2.5%, 1:2 - 6.4%, 1:3 - 6.7%**, and **1:4 - 1.7%**. The ratio of 1:3 produced the best results while 1:4 resulted in too much loss of coherence.

Heuristic for Splitting

In order to find P_{max}, it is necessary to come up with a method for determining the amount of work a given processor has left to do. A heuristic which can be used is the number of scan lines left to work on by a processor, since this is indicative of the amount of work left. Another possible heuristic is the sum of all polygon fragments per scan line for the scan lines which remain to be rendered. A third option is just computing the average number of polygons per scan line since this is simpler to calculate. The latter choices are superior to the first one since they take into account the complexity of work in the area. However, after comparing these possibilities in practice, the potential added benefit of the latter two heuristics did not overcome their increased work although the overall difference between all three choices was not significant.

During the tiling portion of the computation, each processor updates its own shared variable corresponding to the number of scan lines it has left to compute. P_s performs a quick search of the other processors' variables in order to find the largest one, which is then denoted P_{max}. Figure 7 shows a final illustration of the splitting process for an iso-surface dataset where the larger areas are the size of the initial tasks and the smaller areas are initial areas that have been split (here, only 20 processors were used for clarity of illustration). The bottom and right side of each area are color-coded according to the processor which worked on it.

For a message passing machine, the following changes are necessary for task splitting. A controlling processor (CP) is used to direct the assignment of tasks to processors. After each scan line is completed, all processors send a message to the CP to update the number of scan lines they have left to render. When a processor needs to perform a split, it sends a message to the CP to retrieve the current P_{max}. The CP determines the current P_{max} by using the same heuristic as above as well as taking into account how many hops the splitting processor is from a potential P_{max}. This is obviously dependent on the topology of the architecture and should be modified accordingly.

Anomalous Situations

For a shared memory implementation, additional synchronization code is required to combat any possible race conditions or deadlocks which could occur during splitting. For instance, it is possible (and highly likely, especially at the end of a run) that more than one processor might try to split a given P_{max} at nearly the same time. If a semaphore lock is used to prevent simultaneous splitting, the processors could be backed up for some time trying to partition the same P_{max}, not doing any useful work. This is solved by using a test and

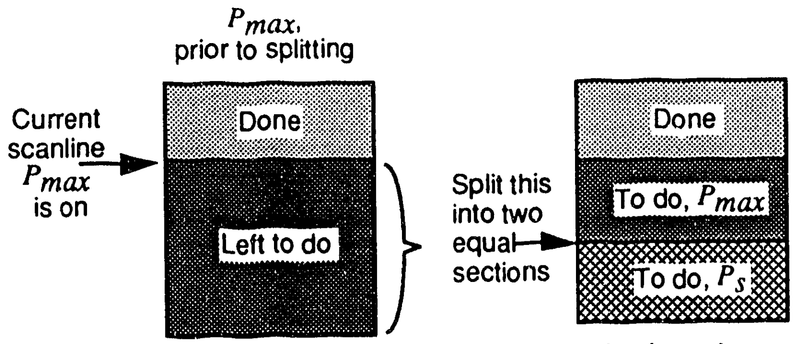

P_{max}, prior to splitting
Current scanline P_{max} is on
Done
Left to do
Split this into two equal sections
Done
To do, P_{max}
To do, P_s

Figure 3: Dynamic splitting of regions for task adaptive scheme

lock methodology in which P_s (as it is searching for P_{max}) checks each processor's split lock to see if the lock is already set as illustrated below.

Assume that P_s has determined so far that processor 6 meets the heuristic for the most amount of work left. If processor 6's lock has not been set (i.e., this processor is not being split at the moment), then the number 6 is stored in $P_s's$ local variable p_max and 6's work left is stored in p_max_work_left. If processor 6's lock has been set, then store the number 6 only as a potential P_{max} in pot_p_max and its work left in pot_p_max_work_left. If, after the search is completed, there is a processor number stored in p_max, P_s splits this one. Otherwise, P_s splits the processor designated as pot_p_max. One should note that the processor being split does not need to be interrupted from its work during the splitting process. Of course, even with this scheme, it is possible that some processors will have to wait at the semaphore before they can proceed. It is also possible that after a processor has proceeded through the lock, there is no work left since it was all completed in the meantime. If that is the case, P_s will recursively call partition to obtain additional work. Based on our measurements, the time spent in a lock by a processor is no more than 0.1% of the execution time and in general is much less.

3.3 Post-processing Phase

This phase consists of outputting the image to the frame buffer or disk, depending on the needs of the user. Sending the image to the frame buffer can be accomplished in a number of ways. The most straightforward method is to have each processor keep a local buffer to store the pixel colors for its screen area. When a given screen area's rendering has been completed, the buffer is sent as a message on a scan line basis to the frame buffer for display.

If the image is being converted to a file for disk storage, the easiest output method is to send it from top to bottom for later display. This is done by storing a virtual copy of the frame buffer in the memory of the multiprocessor. Each scan line of the buffer is stored on a separate processor. As each partial scan line is rendered, it is sent to the memory module which contains that corresponding line in the virtual frame buffer. After the rendering phase is completed, the virtual frame buffer can be copied to a file for storage. This can be accomplished by having each processor run-length encode (in parallel) a scan line for final output. The run-length buffers are then sent to disk in a sequential manner. The average speedup for this section of the code was 1.6 on 96 processors, again limited by the sequential disk access.

4 Complexity

The overall complexity of the rendering phase outlined in this paper can be analyzed using the CREW PRAM[1] model of computation. We assume that there are N polygons used for input with P processors applied to the rendering computation. Running on a parallel computer, the time complexity of the entire program is:

$$T = T_{rend} + T_{split} + T_{comm}$$

where T_{rend} is the rendering time, T_{split} is the time to perform the task adaptive splitting, and T_{comm} is the time for communication of data to the processors. The analysis given in [Myer75] for the serial scan line Z-buffer algorithm shows that T_{rend} is proportional to $O(N)$ in time complexity. The algorithm described in this paper is a parallel version of the scan line Z-buffer where each task computed by a processor can be considered to be executing a serial version of this same algorithm. For a base analysis, we assume that no task splitting takes place during the program which results in $t = 2 \cdot P$ tasks with $\frac{N}{P}$ polygons independently

[1]CREW PRAM - Concurrent Read Exclusive Write, Parallel Random Access Machine

and identically distributed (i.i.d.) to each processor. Assuming each processor (i) receives exactly the average amount of polygons, the number of polygons per task is half the average (since $R = 2$) so the time complexity of a single task is:

$$T_{rend_i} = O\left(\frac{N}{2 \cdot P}\right)$$

The total amount of work for all tasks is then:

$$T_{rend} = 2 \cdot P \cdot T_{rend_i}$$

or just $O(N)$. If this work is done in parallel by P processors, then T_{rend} can be reduced to $O(\frac{N}{P})$. We now show that task adaptive rendering results in the same complexity as the ideal distribution of polygons used in the base analysis described above.

Contrary to the assumption of i.i.d. assignment of polygons above, most datasets will result in some processors obtaining more than $\frac{N}{P}$ and some less. The task splitting mechanism compensates for this deviation in the following manner. Task splitting occurs when a given processor does not have enough work (i.e., it's original polygon distribution is $< \frac{N}{P}$). When this is the case, a lightly loaded processor steals some work (polygons) from a more heavily loaded one in order to bring its total closer to the ideal distribution. The processor it steals from (P_{max}) is the most heavily loaded processor according to a greedy selection criteria. Thus, P_{max} must have more polygons than $\frac{N}{P}$ and need to give some up to come closer to the ideal distribution. The task adaptive splitting mechanism essentially ensures that processors come as close to the ideal polygon distribution as possible. Since the result of task adaptive splitting is a situation where each processor ends up with a total workload nearly equal to $\frac{N}{P}$ polygons, the previous analysis for rendering is therefore accurate for the time complexity of the parallel task adaptive approach (i.e., $O\left(\frac{N}{P}\right)$).

It can be shown that for splitting, the time to find P_{max} is $O(P)$ and the number of splits is proportional to P, but the splits occur in parallel which results in a time complexity of $O(c)$. The communication is proportional to $O\left(\frac{N}{P}\right)$, so the entire program is of complexity $T = O\left(\frac{N}{P}\right)$. Of course, the PRAM model has the limitation in that it assumes a constant access to shared memory regardless of how large P becomes. Since this is unrealistic, performance degradation is bound to occur as larger processor configurations are used.

5 Graphics Data Decomposition

Two memory referencing schemes are discussed in this section; the first scheme is designed for a shared memory machine while the latter scheme is suitable for distributed memory computers which utilize either shared memory referencing or message passing. The data decomposition consists of 1) setting up the data for later rendering and 2) accessing the data during the rendering phase of the computation.

5.1 Uniformly Distributed (UD) Scheme

In the Uniformly Distributed scheme, data is stored in global memory and is referenced remotely. This is implemented in two different fashions depending on the machine architecture. For a UMA machine (Uniform Memory Access) such as the Encore Multimax or Sequent Balance, the data is stored in globally shared memory and hardware caching is generally used to speed up access to frequently referenced data items. On a NUMA machine (Non-Uniform Memory Access) such as the Cedar, Kendall Square Research KSR1, or BBN Butterfly family, the data is scattered throughout the local memories of the processors but it can also

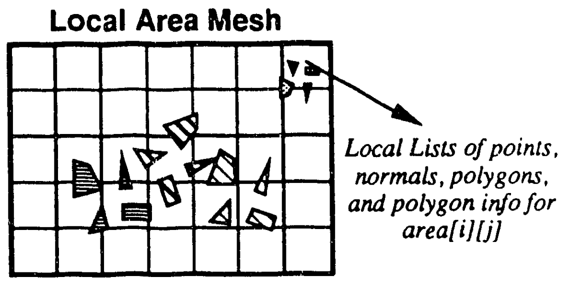

Local Area Mesh
Local Lists of points, normals, polygons, and polygon info for area[i][j]

Figure 4: Bin storage for later local access

be referenced remotely. After the polygon data has been pre-processed, it is not changed so it can be made cachable for faster referencing.

The bins discussed in section 3.1 are used to store the transformed polygonal data for later access. The bins in the UD scheme are implemented as a shared two dimensional array of pointers to shared linked lists of polygons (see figure 2). To prevent multiple processors from appending to the same linked list simultaneously, each bin has a lock which is used to allow atomic access to its linked list. During the rendering stage, polygons are retrieved from a particular bin and put into a local memory *y*-bucket list for a given processor's scan conversion task. This scheme relies on hardware caching of frequently referenced remote data items. While it is adequate for UMA machines, a scheme which supports local memory referencing may provide better results on certain NUMA architectures and facilitates implementation on message passing machines. This is described below.

5.2 Locally Cached (LC) Scheme

The LC memory referencing strategy involves initial storage of data scattered throughout the memories of the machine while using a software caching technique to bring data into local memory during processing. A similar type of mechanism has been used before for parallel graphics rendering [Bado90, Gree89]. In those cases, though, the instance was ray tracing and involved an implementation of a technique known as shared virtual memory which emulates shared memory on a message passing system. Instead of using shared memory however, we employ explicit copying of the *exact* data which is needed for a given task so that: no unnecessary communication is required, the minimum amount of memory is used, and a cache replacement policy is unnecessary. This explicit copying is not amenable to ray tracing since the exact data needed for a given portion of the image space is not know a priori in that type of algorithm.

During the pre-processing phase, polygons are put into bins as discussed above but the implementation here differs since only local memory is utilized. Each processor maintains its own set of bins which are stored locally and implemented as a two-dimensional array of structures. The structures contain the following four arrays: a points list, a normals list, a polygon connectivity list (indices into the points list), and a polygon information list (such as bounding box, color, etc.). Each structure contains data for those polygons which cross over into that particular bin as illustrated in figure 4. Before storing the polygons in these bins, processors look at all the polygons in their local memory to see how many belong in each bin. This information is used to determine how much array memory is necessary to allocate for a particular bin. The reason that arrays (as opposed to linked lists) are constructed is that each array can later be sent out as a message to a processor which will tile the area during the rendering phase.

After the memory has been allocated, the local polygon data is placed into the separate arrays for each bin. The data in a given object's polygon topology array contains index pointers to that object's points array. These indices must be modified to point to the correct place in each bin's points array. It is desirable to store

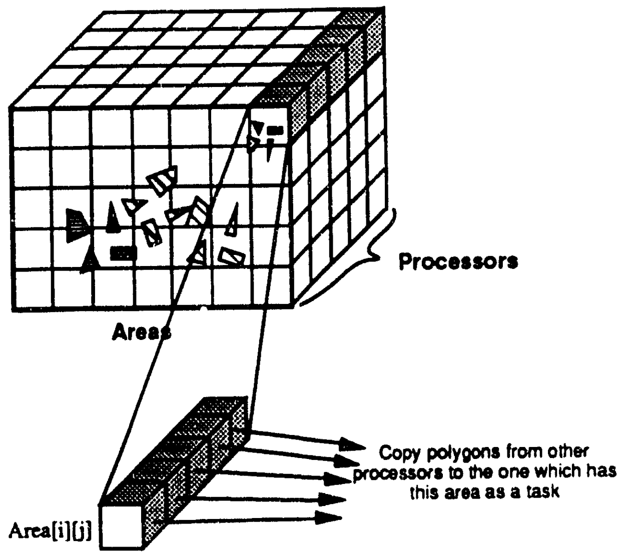

Processors
Areas
Copy polygons from other processors to the one which has this area as a task
Area[i][j]

Figure 5: Consolidation of remote polygon arrays to local memory

a point only once in the new points list and record the reference index value once rather than place a new copy of the point into the points list for each polygon which references it. If the latter were done for say, quadrilateral polygons, we might end up using four times as much memory as is really needed. After the data is processed into messages for each bin, it is "cached" into each processor's memory during the rendering phase.

In order to tile an area during the rendering phase, a processor must obtain the transformed polygon data from the other processors (if they contain any) which is relevant to its assigned area ([i][j]) as shown in figure 5. This is done by querying each processor individually and retrieving that processor's data (if it has any) in contiguous messages. This method is useful on any distributed memory computer (even one which supports shared memory) since the caching operation described here is under programmer control. Note, this is happening in parallel for all of the processors simultaneously so the network might potentially become clogged for a short time. After the burst of communication to obtain initial task data, no more communication is necessary for this task since the data has now been "cached" into the rendering processor's memory.

A method to reduce communication during rendering has been discussed in the PixelFlow design [Moln92]. This solution involves image compositing after each processor renders the entire image space using the local data only. The design trades the communication problem for the expenses of: synchronization, later communication (albeit potentially smaller than discussed above), and a non-adaptive load balancing scheme based primarily on the data distribution to processors. This method seems primarily suitable for very large (say $>$ 1000) processor configurations since the load balancing relies totally on the data distribution and communication is only a function of image resolution.

Data Movement During Partitioning

When a processor is going to split another processor's pixel area, it must retrieve the four arrays in the bin structure of that pixel area. The pointers to these arrays are stored in the memory of the remote processor and can be readily retrieved. Ideally, it would be nice to only obtain the data for the polygons which are relevant to the lower task after the split. Since that involves stopping the execution of the remote processor, it was not deemed worthwhile to do in the shared memory implementation. However, a simple method can be used to reduce the amount of data to be copied over time by performing a quick clip test after the arrays are received.

Polygons which are no longer relevant to the new partitioned task are deleted. Although this operation takes a small amount of time, it reduces the communication for further splits of the same area. Testing indicates that this is a worthwhile addition to the algorithm since it reduces the amount of communication near the end of the program when most of the areas to be split are small and have already been split at least once.

Changes ıor Message Passing Implementation

The LC scheme is amenable to implementation on message passing architectures, although some changes are necessary. On these machines, processors cannot explicitly copy data from another processor, it must be sent in a message to them. For the rendering phase, the shared memory approach involves every processor checking every other processor's memory to find out if there is any data residing on a remote memory module which is relevant to its assigned task. Instead, for message passing each processor is assigned its first task area according to its processor number, with processor 0 getting all of the background tasks. The processors then *know* which one is responsible for a particular task area. If a processor's memory contains data for another one's task, it can then send a message there which contains the relevant data or a flag which indicates that it has no data relevant to the particular area. In this way, each processor retrieves all of the data necessary from the others for its assigned task. Both methods require the same amount of communication but the shared memory method benefits from dynamic self scheduling of tasks which is not amenable to message passing. After the first set of tasks, a processor can retrieve which task it is to be assigned from the controlling processor (CP). The CP will then send messages to the others to indicate which tasks are now executing on which processors. For partitioning, each processor must check every so often to see if there is a message on its queue requesting a split. If so, the processor then sends exactly the data needed for the lower half of its remaining work to the requester. It also sends a message back to the CP to update the number of scan lines left. Unfortunately, using the CP requires slightly more overhead than the direct query method used in the shared memory implementation.

6 Results

The algorithm was tested on a BBN Butterfly TC2000 multiprocessor using the Task Adaptive domain decomposition and the LC memory referencing scheme. Three input datasets of varying complexity were used, two of which (*rings* and *tree*) are from Eric Haines' SPD database [Hain87]. The other image, *layers*, i several transparent iso-surface layers from a fusion plasma turbulence simulation generated by Tim Williams at Lawrence Livermore Lab. These test images[2] are shown in figures 7, 8, and 9. All rendering utilized 16 samples/pixel anti-aliasing with Phong smooth shading at standard video (640×484) resolution.

6.1 Performance

In table 1, we compare average time for rendering including algorithmic specific overheads for the test images using the Data Adaptive algorithm (described in section 2), the base rectangular region method, and the Task Adaptive algorithm using the UD and LC memory referencing schemes and static versus dynamic scheduling of tasks. Recall, the UD scheme cannot be implemented on message passing computers. It should also be noted that although the performance of the UD scheme is equal to that of the LC scheme, this will vary depending on the machine topology and hardware cache support. The timing, speedup, and efficiency results for the tiling section of the program are given in table 2 with a graph of the speedup in figure 10. The speedup measured is self speedup and the program utilized remote memory transfers even on single processor

[2]The layers image shows the area borders color-coded according to which processor worked on a particular screen area.

Table 1: Comparison of various algorithms' performance at 96 processors

Algorithm	Average Time
Data Adaptive - LC	25.9 seconds
Rectangular - LC	20.9 seconds
Task Adaptive - LC (Static)	9.8 seconds
Task Adaptive - UD (Dynamic)	9.4 seconds
Task Adaptive - LC (Dynamic)	9.4 seconds

Table 2: Rendering time, speedup, and efficiency of Task Adaptive (LC dynamic) algorithm on 96 processors on the BBN TC2000

	Layers	Rings	Tree
# Polygons	*64.1K*	*568K*	*851K*
Timc	*6.2 sec*	*9.6 sec*	*8.8 sec*
Speedup	*83.2*	*90.1*	*63.1*
Efficiency	*0.86*	*0.94*	*0.66*

runs. This was required since the input datasets would not fit into the memory of a single processor. To compensate, the extra cost of communication and code modification were measured and subtracted out of the single processor time in order to come up with a realistic estimate of the sequential time (noted in the graph) for speedup measurements. According to our measurements, there is only a 15% measured overhead cost difference between an optimized serial renderer and the parallel program described here when run on a single processor. The average efficiency of the parallel algorithm is 82% which means that the parallel version provides high performance relative to itself as well as to a serial renderer.

In order to show the actual causes of performance degradation from linear speedup, various overhead effects were measured for the implemented program on the input datasets as a function of the number of processors (P). They include: load imbalance, network contention, code modification to allow parallel execution, and communication (see the glossary for definitions). Other factors (which were also measured) such as memory latency, synchronization, and scheduling represent such a small fraction (typically $< 0.1\%$) of the work that they are not presented in the graphs. These overheads are not taken into account in the PRAM model of computation. Note, these latter factors are small due to the fact that the splitting mechanism involves very little synchronization of processes. The equations given in the appendix show how the factors are calculated. The graphs which depict the measured degradation factors as a function of P for the layers, rings, and tree images are given in figures 11-13. The overheads shown at one processor represent the effects of code modifications necessary to run the code on a parallel machine and the communication required to transfer the data from shared memory to local memory. Table 3 indicates the computed overheads at 96 processors.

6.2 Analysis

From the graphs, we can see that several of the overhead effects use an increasingly higher percentage of runtime as P is increased, meaning that different factors come into play at higher processor counts. Load balancing, in particular, is directly affected. As P increases, more processors perform dynamic partitioning

Table 3: Parallel overhead percentages at 96 processors on the BBN TC2000

Overhead	Layers	Rings	Tree
Communication	*1.2%*	*2.2%*	*7.9%*
Network Contention	*1.7%*	*2.2%*	*7.8%*
Load Imbalance	*9.5%*	*3.5%*	*24.8%*
Code Modification	*2.2%*	*1.7%*	*0.8%*

simultaneously. Consequently, at the end of a run, some processors have work to do while others are trying to obtain work. By the time they obtain the lock to split the work, there may not be any work left. As was stated previously, the heuristic for splitting is not particularly accurate which explains the high load imbalance particularly for the tree input dataset. The degradation due to code modification has to do with the fact that additional coherence is lost as the number of processors and consequently total tasks, increase. Communication overhead increases with P for two reasons. As P increases, the size of the task regions becomes smaller since $\#tasks = 2 \cdot P$, so polygons cross over into more areas (which means more copying of this polygon data). Secondly, as P is increased, more processors become available to partition others' work, which requires communication of data to obtain the new tasks. As a result of this increased communication, network contention increases as well.

If the number of polygons (N) increases and the number of processors (P) stays the same, we observe that the overall communication will increase as well. But, according to the time complexity $O\left(\frac{N}{P}\right)$, rendering time also increases with N so communication is not likely to become a higher percentage cost. However, if the size and distribution of polygons is changed significantly, this will affect the overall rendering time. For instance, the polygons in the tree image are smaller and more densely packed than in the rings image. Since the (N) is larger in the tree image, communication is higher but is also a higher percentage of overall rendering time since the rendering time per polygon is smaller (comparing the two serial execution times). As one can see, depth complexity, polygon area, number of polygons as well as microprocessor speed, communication speed, and data transfer methodology all contribute to the efficiency of a parallel renderer. It is obvious, for instance, that doubling the output image resolution increases the work without increasing the overheads, so the program appears to be more efficient. With all of these variables, it is impossible to generalize a solution to meet all users' needs. One should take into account the the main requirements of the system when designing the data decomposition mechanism and use optimizations for data locality and load balancing appropriately. The results here indicate that the LC scheme exploits data locality at a minimal expense of communication overhead and can be used in both shared memory and message passing environments.

This algorithm has some deficiencies in dealing with image space tasks which have a high degree of local complexity since these types of scenes may not be amenable to the splitting mechanism. An example is the tree input dataset which resulted in a high load imbalance. Another situation is flight simulation where data can become concentrated at the horizon. It would be ludicrous to assume that a single algorithm can be designed to perform equally well under all types of scenarios. The algorithm presented here is a compromise which can handle a moderate amount of local image complexity with reasonable performance. Ray tracing solutions may have enough complexity to allow parallel processing even at the pixel level to be able to remedy this situation. However, ray tracing is inherently slower than scan conversion methods and its features are not needed in a number of applications. Unfortunately, assigning pixel sized tasks would be at the expense of higher loss of coherence when using polygon scan conversion. Additionally, the ratio of communication to computation is not high enough in scan conversion methods to support task subdivision at this level.

7 Conclusion

The main goal of this project was to achieve good speedup and efficiency using a parallel algorithm for rendering of complex geometric scenes. The Locally Cached memory strategy allows the algorithm to be implemented on either shared memory or message passing MIMD computers. Although the absolute performance of this algorithm already ranges from 50,000 to 100,000 anti-aliased Phong shaded polygons/second, [3] the clear metric for success is that we have achieved an average 82% efficiency utilizing 96 processors. This indicates that with even faster microprocessors and/or larger parallel computers, rendering rates can be increased even further. In addition, using this algorithm as part of scientific simulation system on a parallel computer allows direct memory transfer of data and supports fast creation of "movies in minutes."

The algorithm outlined here can also be used for a number of other problems in graphics. For instance, $2\frac{1}{2}$-d polygon overlaying for geographical planning is a computationally intensive problem that can benefit from parallel processing. The Task Adaptive approach to work decomposition can also be useful in a cluster (or distributed) computing environment to harness cycles from idle machines for graphics rendering.

Acknowledgments

This work performed under the auspices of the U.S. Department of Energy by the Lawrence Livermore National Laboratory under contract number W-7405-ENG-48.

References

[Abra86] Abram, Greg *Parallel Image Generation with Anti-Aliasing and Texturing*, Ph.D. dissertation, University of North Carolina at Chapel Hill, (1986).

[Alli91] Allison, Michael. Private communication, (July 1991).

[Bado90] Badouel, Didier, Bouatouch, Kadi, and Priol, Thierry "Ray Tracing on Distributed Memory Parallel Computers: Strategies for Distributing Computations and Data." Course Notes for Course 28, Siggraph (August 1990) pp. 185-198.

[Chan81] Chang, P. and Jain, Ramesh "A Multi-Processor System for Hidden Surface Removal." *Computer Graphics 15*, 4 (December 1981).

[Cook86] Cook, Robert L. "Stochastic Sampling in Computer Graphics." *ACM Transactions on Graphics* (January 1986).

[Croc91] Crockett, Thomas W. and Orloff, Tobias "A Parallel Rendering Algorithm for MIMD Architectures." ICASE Technical Report No. 91-3, NASA Langley Research Center, (June 1991).

[Dipp84] Dippe, Mark and Swensen, John "An Adaptive Subdivision Algorithm and Parallel Architecture for Realistic Image Synthesis." *Computer Graphics, Proceedings of Siggraph 18*, 3 (July 1984) pp. 149-158.

[Fium83] Fiume, Eugene, Fournier, Alain, and Rudolph, Larry "A Parallel Scan Conversion Algorithm with Anti-Aliasing for a General Purpose Ultracomputer." *Computer Graphics, Proceedings of Siggraph 17*, 3 (July 1983) pp. 141-149.

[Fran90] Franklin, Wm. Randolph and Kankanhalli, Mohan S. "Parallel Object-Space Hidden Surface Removal." *Computer Graphics, Proceedings of Siggraph 24*, 4 (August 1990) pp. 87-94.

[3] It does not make sense to compare this algorithm to special purpose graphics hardware because: 1) Current generation workstations cannot do Phong *shading* (SGI's machines currently only support Phong lighting). 2) This scan line based algorithm requires more operations than a Z-buffer algorithm. 3) This algorithm is software based and therefore more general purpose. In addition, the performance quoted here is using a 3 year old CPU (Motorola 88100) which is not as powerful as today's microprocessors.

[Ghos86] Ghosal, Dipak and Patnaik, L. M. "Parallel Polygon Scan Conversion Algorithms: Performance Evaluation of a Shared Bus Architecture." *Computers & Graphics 10*, 1 (1986) pp. 7-25.

[Gree89] Green, S. and Paddon, D. "Exploiting Coherence for Multiprocessor Ray Tracing." *IEEE Computer Graphics and Applications* (November 1989) pp. 12-26.

[Hain87] Haines, Eric "A Proposal for Standard Graphics Environments." *IEEE Computer Graphics & Applications 7*, 11 (November 1987) pp. 3-5.

[Hu85] Hu, Mei-Cheng and Foley, James D. "Parallel Processing Approaches to Hidden-Surface Removal in Image Space." *Computers & Graphics 9*, 3 (1985) pp. 303-317.

[Kapl79] Kaplan, Michael and Greenberg, Donald P. "Parallel Processing Techniques for Hidden Surface Removal." *Computer Graphics, Proceedings of Siggraph* (July 1979) pp. 300-307.

[Moln92] Molnar, Steven, Eyles, John and Poulton, John "PixelFlow: High-Speed Rendering Using Image Composition." *Computer Graphics, Proceedings of Siggraph 26*, 2 (July 1992) pp. 231-240.

[Myer75] Myers, Allan J. "An Efficient Visible Surface Algorithm." Report to the NSF for Grant Number DCR74-00768 A01, (July 1975).

[Park80] Parke, Frederic I. "Simulation and Expected Performance Analysis of Multiple Processor Z-Buffer Systems." *Computer Graphics, Proceedings of Siggraph 14*, 3 (July 1980) pp. 48-56.

[Robl88] Roble, Doug R. "A Load Balanced Parallel Scanline Z-Buffer Algorithm for the iPSC Hypercube." In *Proceedings of Pixim '88*, Paris, France, (October 1988).

[Whel85] Whelan, Daniel S. *Animac: A Multiprocessor Architecture for Real-Time Computer Animation*, Ph.D. dissertation, California Institute of Technology, (1985).

[Whit91] Whitman, Scott *Utilizing Scalable Shared Memory Multiprocessors for Computer Graphics Rendering*, Ph.D. dissertation, The Ohio State University, Columbus, Ohio, (1991).

[Whit92] Whitman, Scott "Parallel Graphics Rendering Algorithms," *Proceedings of the 3rd Eurographics Workshop on Rendering*, Bristol, UK, (May 1992).

A Parallel Overhead Factors

Load imbalance is measured by determining the maximum finishing time of all processors $T_{max}(P)$, and comparing that to the average finishing time of the P processors. This average constitutes what would be the theoretical *ideal* finishing time if all the processors had exactly the same amount of work.

$$Load \quad imbal. \ \% = \frac{T_{max}(P) - T_{avg}(P)}{T_{max}(P)} * 100\% \qquad (1)$$

Communication in the BBN TC2000 system proceeds at a measured $T_{comm} = 3\mu sec/(4 \ byte \ word)$ using a special function which transfers data in 16 byte chunks. By determining the number of words transferred for each initial task plus the number transferred for each task created by a partition, we can determine the total amount of communication in the system. Latency, on the other hand, is the extra time required for a remote memory reference of a particular data item versus the time for that reference on a local memory module. In the algorithm outlined here, very little data is referenced remotely; it is directly copied using the special communication function and the algorithm outlined previously. Below, (# words) refers to the number of words transferred per task. The subscripted notation used here is as follows: the i in each sum relates to information about each task i. The first sum is over all the initial tasks while the second sum refers to tasks generated after the splitting operation occurs.

$$Comm.\ \% = \frac{\left(\sum_{i=1}^{R\cdot P}(\#words)_i + \sum_{i=1}^{\#partitions}(\#words)_i\right)\cdot T_{comm}}{T_{max}(P)\cdot P} * 100\% \qquad (2)$$

Network contention consists of delays in message transfer as a result of excessive simultaneous requests for the existing paths in the system. The problem is run on a single processor (where no contention occurs) and then run on multiple processors, measuring the time for each task to execute. The difference in the times for task execution (excluding the extra coherence and communication cost) is a direct result of contention in the network. $T(P)_i$ refers to the time for task i to finish when using P processors. The second summation on the numerator refers to the number of partitions that are created as a result of the dynamic task splitting which occurs in the task adaptive approach.

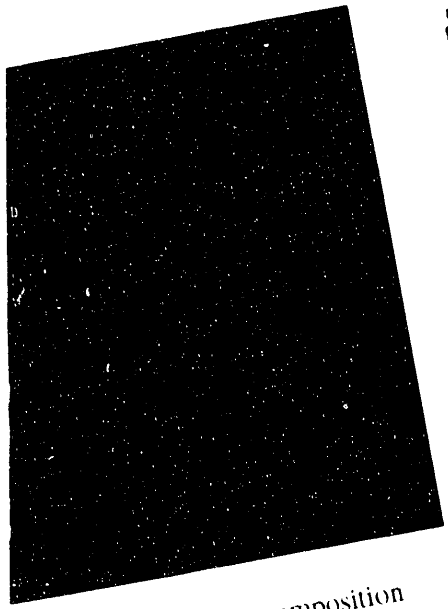
$$Contention\ \% = \frac{\sum_{i=1}^{R\cdot P} T(P)_i + \sum_{i=1}^{\#partitions} T(P)_i - \sum_{i=1}^{R\cdot P} T(1)_i}{T_{max}(P)\cdot P} * 100\% \qquad (3)$$

Code modification overhead consists primarily of the loss of coherence due to creation of a new region in a parallel setting. It can be measured by timing the extra work T_{coh} that is associated with all polygons that would have been updated from another region due to coherence.

$$Code\ \ mod.\ \% = \frac{\sum_{i=1}^{R\cdot P} T_{coh}(P)_i + \sum_{i=1}^{\#partitions} T_{coh}(P)_i}{T_{max}(P)\cdot P} * 100\% \qquad (4)$$

B Glossary

MIMD	Multiple instruction stream, multiple data stream computer.
SIMD	Single instruction stream, multiple data stream computer.
Load imbalance	A measurement corresponding to inadequacies in how well the total work is distributed among the processors.
Scheduling	Determining the mapping of tasks to processors.
Latency	The *additional* time to retrieve a data item from remote memory versus obtaining it locally.
Communication	The time to send data from one memory module to another.
Network contention	The problem of multiple messages attempting to use a single interconnection network path or switch node at the same time.
Synchronization	The act of bringing together 2 or more processes to a given point in a program simultaneously.
Code modification	The additional time resulting from changes to code necessary to run the program in a parallel environment. In the instance here, this is exemplified as the loss of graphical coherence.


le rectangular area decomposition

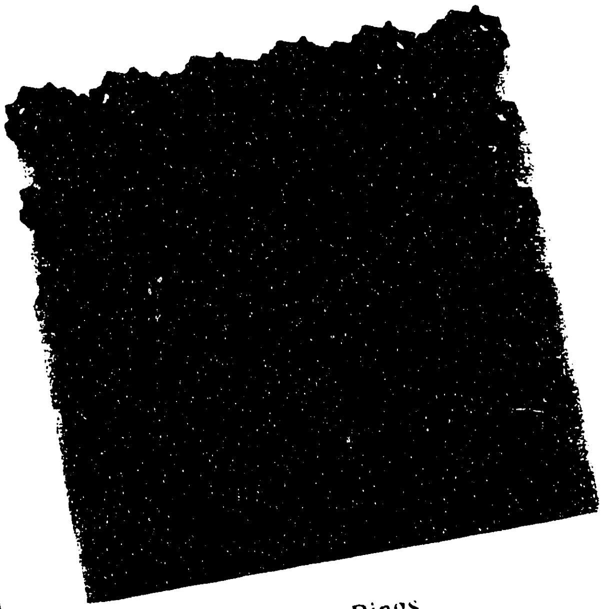


Figure 8: Rings

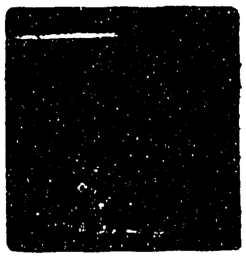
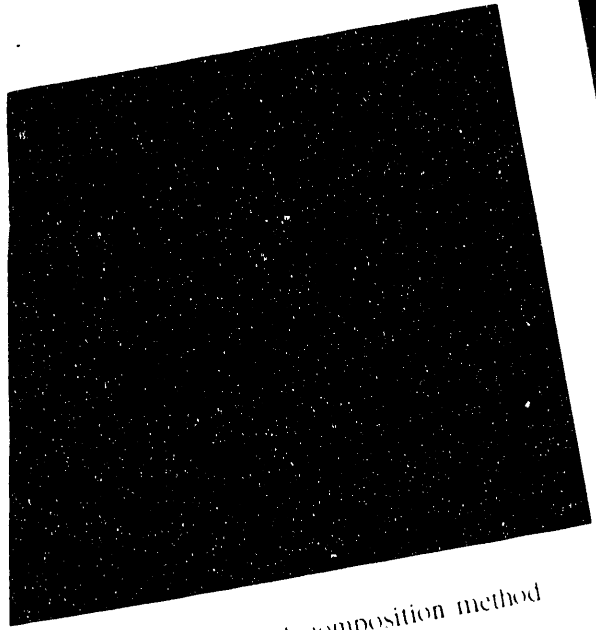


Figure 9: Tree


gure 7: Task adaptive decomposition method

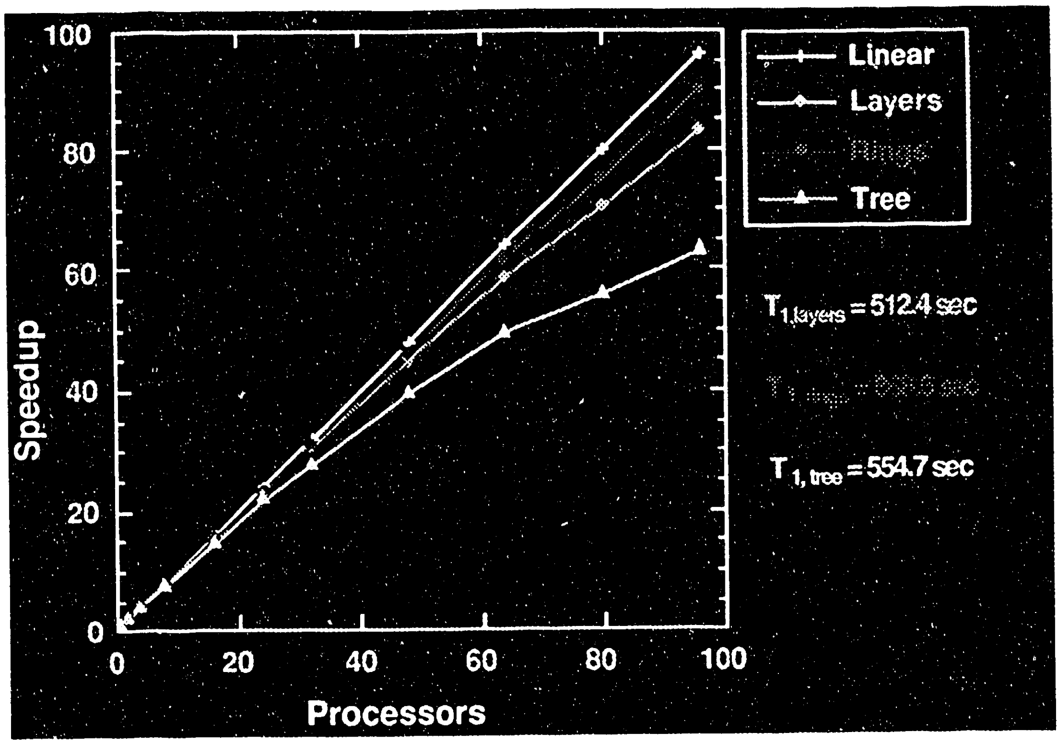


Figure 10: Task adaptive speedup

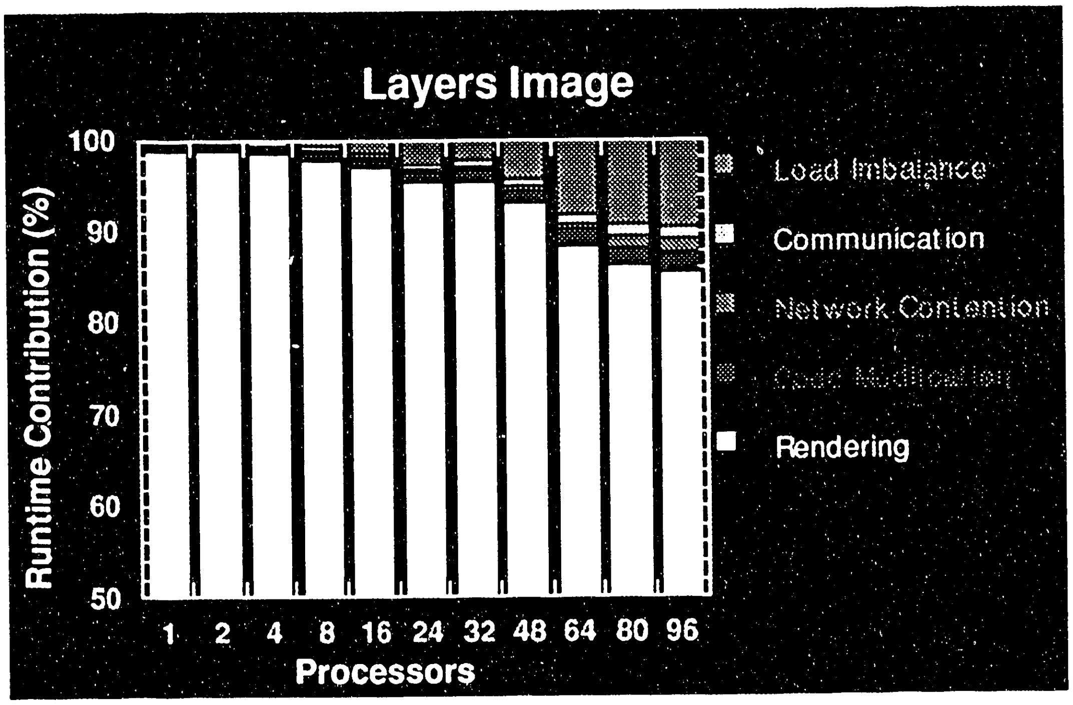


Figure 11: Analysis of overheads, layers image

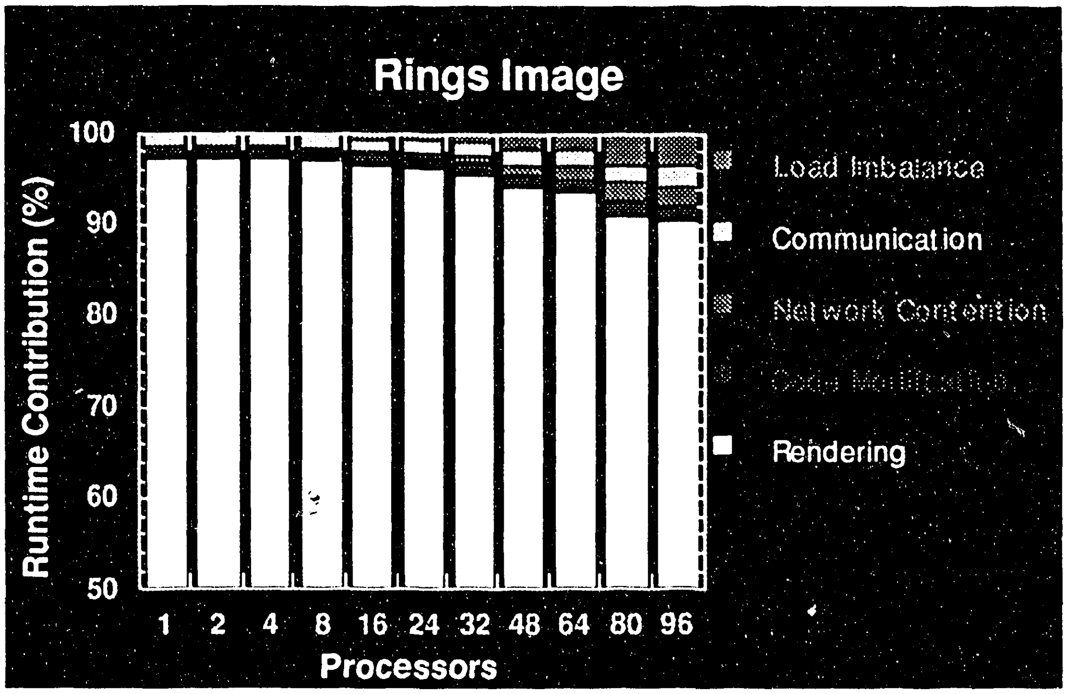


Figure 12: Analysis of overheads, rings image

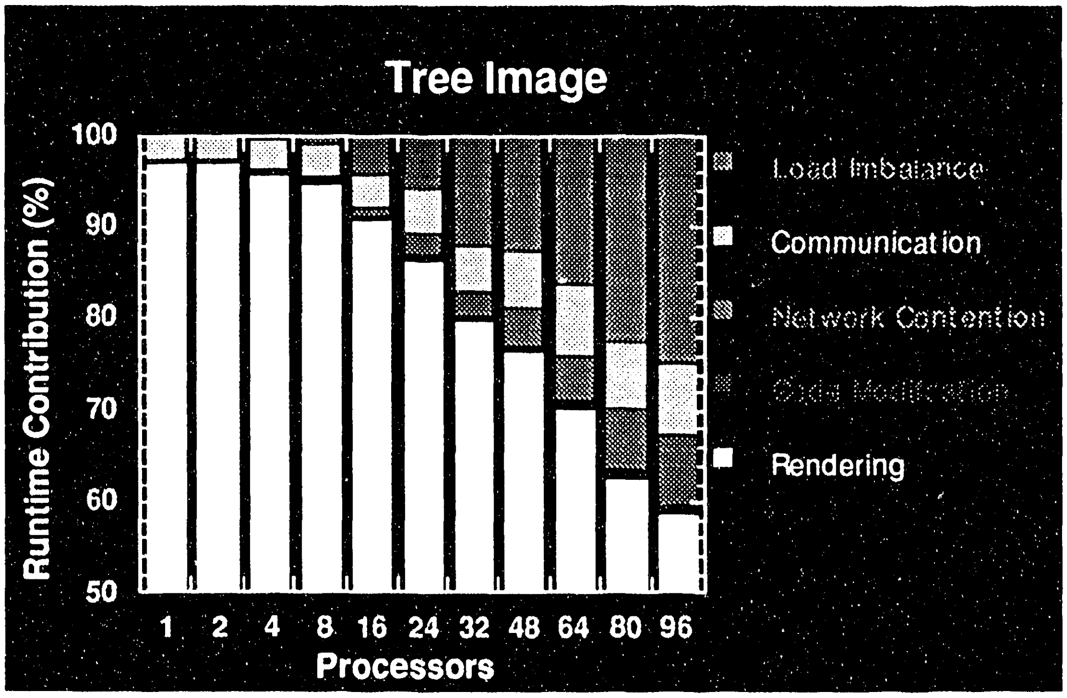


Figure 13: Analysis of overheads, tree image