Dynamic Load Balancing for Parallel Polygon Rendering

Scott Whitman
David Sarnoff Research Center

A new graphics renderer incorporates novel partitioning methods for efficient execution on a parallel computer. It requires little overhead, executes efficiently, and demands minimal processor synchronization.

Using parallel processing for visualization speeds up computer graphics rendering of complex data sets. Such sets arise in disciplines like global climate modeling, molecular dynamics, and finite element modeling.

A parallel algorithm designed for polygon scan conversion and rendering—the topic of this article—supports fast rendering of highly complex data sets using advanced lighting models. Dedicated graphics rendering engines do not necessarily suit such data sets, although they can support real-time update of moderately complex scenes using simple lighting. Advantages to using a software-based approach include the feasibility of adding special rendering features to the program and the capability of integrating a parallel scientific application with a parallel graphics renderer.

A new work decomposition strategy presented here, called *task adaptive*, is based on dynamically partitioning the amount of computational work left at a given time. The algorithm uses a heuristic for dynamic task decomposition in which image-space tasks are partitioned without requiring interruption of the partitioned processor. A sophisticated memory referencing strategy lets local memory access graphics data during rendering. This permits implementation of the algorithm on a distributed memory multiprocessor. An in-depth analysis of the overhead costs accompanying parallel processing shows where performance is adequate or could be improved.

Historical perspective

The past 15 years have seen 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 some work has focused on parallel object-space methods,[1,2] most algorithms have been of the image-space variety. A taxonomy of these approaches appears elsewhere,[3] but I briefly summarize them here. Polygon decompositions can involve independent tasks, such as used by Fiume, Fournier, and Rudolph,[4] to process span segments of different polygons on a scan line in parallel. Crockett and Orloff[5] performed object-space computations, determined where to send the results, and communicated this data to the appropriate image-space processors, which then rasterized the output and sent it to the z-buffer.

In using image-space pixels as a building block for domain decomposition, researchers devised the following tasks for parallel processing: horizontal strips (screen-wide) of scan lines,[5-7] vertical strips (screen-high) of pixels,[7] and rectangular areas of pixels.[6-8]

The image-space parallel algorithms for load balancing divide with respect to how specific tasks are determined, into *data nonadaptive* and *data adaptive*. The data nonadaptive methodology relies on an initial decomposition of image space unrelated to the input data. Many easily constructed image-space tasks of varying work loads are assigned for parallel processing. The

0272-17-16/94/$4.00 © 1994 IEEE

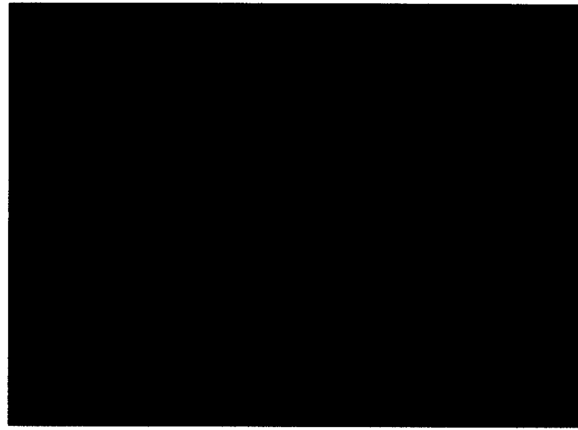


Figure 1. Simple rectangular decomposition.

success of this approach depends on how these tasks are scheduled onto the processors (statically or dynamically). In the data adaptive case, the sizes 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[7-9] in the context of scan conversion methods. Similar methods were used in ray tracing and direct volume rendering. The PixelFlow design[10] discussed a method to reduce communication during rendering. The PixelFlow solution involves image compositing after each processor renders the entire image space using data present on the local processor only. That design trades the communication problem for the expenses of synchronization, later communication (albeit potentially smaller than discussed above), and a data nonadaptive load-balancing scheme based on the distribution of objects to processors.

Previously, I implemented a number of these algorithms to determine their relative strengths and weaknesses,[3,9] with the results briefly summarized here. Of the data nonadaptive schemes, empirical tests indicated that the rectangular method provides the highest performance. A possible explanation for this result is that the rectangular layout minimizes the perimeter while maximizing coherence in both the horizontal and vertical directions. In regard to the data adaptive algorithms, Roble's algorithm and my implementation of Whelan's algorithm resulted in approximately 25 percent additional preprocessing overhead versus the data nonadaptive algorithms. Data adaptive schemes therefore seem less suitable for generating single images, although we could perhaps amortize their cost for animation purposes.

The typical data nonadaptive load-balancing scheme involves tiling image space into $R \times P$ rectangular areas dynamically assigned to processors, where R is the granularity ratio and P is the number of processors. Figure 1 shows a simple rectangular decomposition. Dynamic assignment typically produces better load balancing with minimal overhead compared to the static scheduling alternative. Dynamic scheduling can be implemented in a multicomputer environment by a single control processor, while in a shared memory environment all processors can self-schedule. R 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 preprocessing, communication, and polygon duplication, but the better the load balancing. Conversely, smaller values of R result in less overhead but inadequate load balancing. In my experience, an R value of 20 provided good performance, although other researchers have used slightly lower values with some success.

The algorithm presented in this article uses $2 \times P$ rectangular areas as a starting point. Doing so improves upon this approach by allowing us to use $R = 2$ instead of $R = 20$ to achieve adequate load balancing while reducing preprocessing overhead.

Algorithm

Now let us consider the work decomposition strategy for three distinct phases of computer image synthesis. The average percent of total time per phase in a sequential version of the test program (based on the input data given later) appears in parentheses.

1. Preprocessing (13 percent)—Consists of data read-in, transformation of points, normals calculation, back-face rejection, clipping, and perspective projection.
2. Rendering (86 percent)—Includes hidden-surface removal, shading, antialiasing, and any other visual effects.
3. Postprocessing (1 percent)—Involves displaying the image on a frame buffer or storing it in a file.

Task-adaptive domain decomposition is a variation on using rectangular areas of pixels as individual tasks. I will focus primarily on the rendering portion here, since this phase takes the bulk of the computation time. I also give methods for parallelizing the first and third phases, but implementations of these are generally specific to the environment where the overall program will run. Figure 2 shows a brief overview of the parallel algorithm.

Preprocessing phase

The implementation of the preprocessing phase depends to a large degree on the amount of parallelism available. If there are a large number of objects (for example, greater than P, the number of processors), then objects are read into processors' memories through a round-robin distribution of objects to processors. Figure 3 shows the pipelined parallelism of the preprocessing phase. Pipelining lets us start processing objects (one after another) as soon as a given processor has released access to the controlled resource—in this case, the disk. Note, data can come from sources other than the test program, such as another program, a HiPPI interface, or a parallel disk farm. 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. Once the objects are loaded and processed, each processor can perform transformations, clipping, and so forth on the data in its local memory.

If the number of objects is instead less than P, we have two possible scenarios. In the first, if we are animating these objects over a long period of time, then we can split each object into multiple subobjects to be processed independently. In the second, if we want to create a single image, a possible parallelization could involve parallel processing iterations of the individual loops used for transforming or clipping the objects. This would work in a shared memory machine but not for a message-pass-

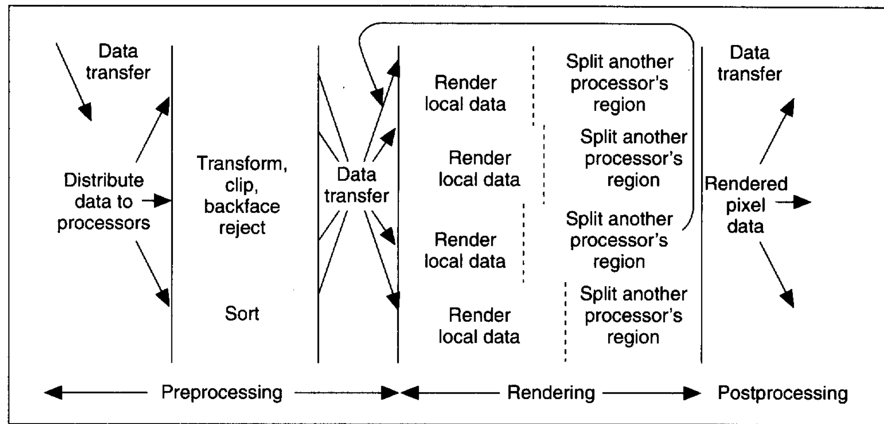


Figure 2. Overall algorithm with synchronization points.

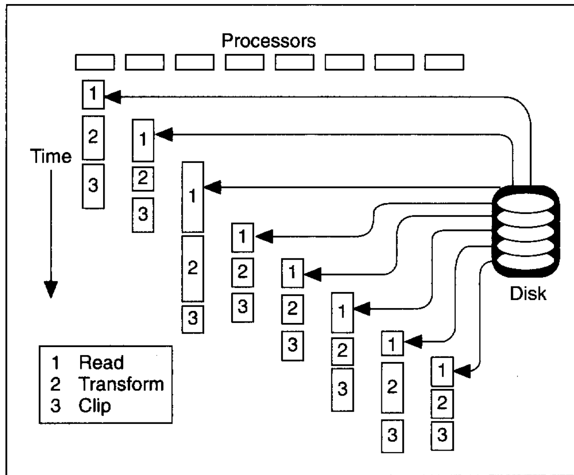


Figure 3. Preprocessing phase pipelined parallelism.

ing architecture. Alternatively, a reader process could assign portions of the data to individual processors. This part of the preprocessing phase is important to note, since certain algorithms[10] rely on the partitioning of objects to individual processors as their primary load-balancing mechanism.

The second part of the preprocessing phase involves setting up the data for 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 image space, in which polygons are placed into bins corresponding to a particular task. The bounding box of the polygon determines which bin(s) that polygon enters.

The average speedup for the algorithm's preprocessing portion as implemented on the BBN TC2000 multiprocessor using this partitioning strategy was 9.4 on 96 processors. The main limitation to further speedup is the sequential nature of the disk access for reading in data. Machines with parallel disk I/O could alleviate this restriction, as could using data already partitioned and provided directly from a simulation program running on the same computer.

Rendering phase

To solve the rendering problem in parallel, the task-adaptive algorithm treats rectangular regions of pixels as basic tasks. The tiling problem is solved serially for those polygons present in each task's region. Here, I employ a modified scan-line z-buffer algorithm[11] that uses stochastic sampling (16 samples per pixel) for antialiasing. We might be able to use other hidden-surface-removal approaches (for example, z-buffer) with some modification to the task-adaptive algorithm. The only difficulty in using a scan-line algorithm is that all polygons must be present in memory before hidden-surface removal occurs. Nevertheless, the memory referencing scheme elaborated upon later uses this feature to minimize data transfer.

The task-adaptive approach can use a small granularity ratio (in this case, $R = 2$), which significantly reduces overheads compared to the normal rectangular approach (when R is higher). For instance, using a value of 2 for R instead of 24 resulted in an average reduction in work of approximately 10-fold in the preprocessing and data transfer sections. We could use a ratio of $R = 1$, but that situation requires more communication because of the load-balancing mechanism employed. Here, we use a simple assignment of the first P regions encountered as the first set of tasks. After a processor has finished its first task, it dynamically retrieves additional tasks from the queue until no tasks remain.

The sidebar (next page) gives the pseudocode similar to what each processor executes. The code is shown for a shared-memory implementation where the shared variable j is atomically accessed.

The `partition` routine dynamically balances 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 no initial tasks remain. The steps given in the sidebar outline the adaptive nature of this work decomposition. They are essentially part of the `partition` routine. The terminology P_{max} refers to the processor index of the maximally loaded processor, as determined by a splitting processor P_s at a given time.

Using a lock prevents more than one processor at a time from splitting a given P_{max}. The lock, typically implemented as part of the operating system, guarantees that only one processor proceeds through it at a time. If it is not open, the processor spin waits until the lock opens. After P_s completes its new work, it calls `partition` recursively, until there is no work available for splitting above a certain threshold.

Based on empirical studies, splitting is worthwhile even down to two scan lines. However, splitting a single scan line in half did not prove beneficial.

P_s splits P_{max}'s remaining work into two tasks. P_{max} continues

Processor-executed pseudocode

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

1. When a processor (call it P_s) needs work, it searches among the other processors for the one containing the most work left to do (call it P_{max}). If there is sufficient work to do, proceed through the remaining steps, otherwise return.
2. The P_s processor then sets a lock preventing other processors from splitting P_{max} and sets 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 lock and starts

to work on the upper task, while P_s takes the lower one. This also allows maintaining coherence in P_{max}'s region without additional overhead, and P_{max} can continue working on its own task uninterrupted.

Other splitting mechanisms investigated (such as creating two side-by-side regions or even a combination of side-by-side and top-down) demonstrated performance inferior to the method outlined here. Figure 4 illustrates the splitting process.

Note that without this splitting mechanism, the load imbalance is woefully inadequate unless we increase the granularity ratio (number of regions per processor). In some cases, using a ratio of $R = 2$ without splitting would have more than doubled the execution time.[9] Because the splitting relies on work in a task proceeding in a top-to-bottom fashion, scan-line-oriented hidden-surface-removal algorithms suit tasks best.

The initial task regions are split horizontally, hence the split regions deviate further from square. To find the effect of region size on splitting, I tested various aspect ratios for the initial areas. A horizontally oriented region such as that produced with a 2:1 ratio (two pixels across for every one down) resulted in the poorest parallel performance overall. I noted the following average percentage improvements over a 2:1 ratio:

- 2.5 percent for 1:1
- 6.4 percent for 1:2
- 6.7 percent for 1:3
- 1.7 percent for 1:4

These results indicate that a pixel aspect ratio of 1:3 for initial areas produced the best performance.

Heuristic for splitting

To find P_{max}, we must come up with a method for determining the amount of work a given processor has left to do at any given time. One possible heuristic is the number of scan lines left for a processor to work on, since this indicates the amount of

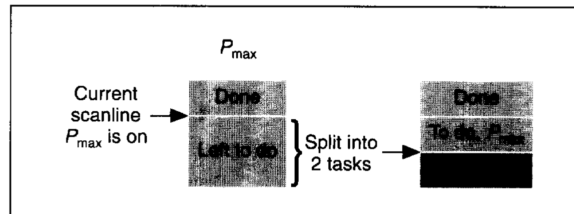


Figure 4. The splitting process.

work left. Other heuristics investigated did not perform as well.

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 quickly checks the other processors' remaining scan lines to find the largest number, which is then denoted P_{max}.

Figure 5 shows a final illustration of the splitting process for an isosurface data set where the larger areas are the size of the initial tasks and the smaller areas are initial areas that have been split. The bottom and right side of each area are color-coded according to the processor that worked on it.

Anomalous situations

Additional synchronization code is required to combat any possible race conditions or deadlocks that could occur during splitting. For instance, more than one processor might try to split a given processor 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. Using a test and lock methodology solves this: P_s (as it searches for P_{max}) checks each processor's split lock to see if the lock is already set.

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 (that is, 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 remaining work is stored in `p_max_work_left`. On the other hand, if processor 6's lock has already been set, then store the number 6 only as a potential P_{max} in `pot_p_max` and its remaining work in `pot_p_max_work_left`. If, after the search is completed, there is a processor number stored in `p_max`, P_s splits that processor. Otherwise, P_s splits the processor designated `pot_p_max`.

Of course, even with this scheme, some processors might have to wait at the lock before they can proceed. Also, after a processor has proceeded through the lock, there might be no work left, it all having been completed in the meantime. If so, P_s will recursively call `partition` to obtain additional work.

Based on my measurements, the time spent waiting in locks by a processor is no more than 0.1 percent of the execution time. In general, it is even less.

Postprocessing phase

The postprocessing phase involves outputting the image to the frame buffer or disk, depending on the user's needs. In the most straightforward method, each processor keeps 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.

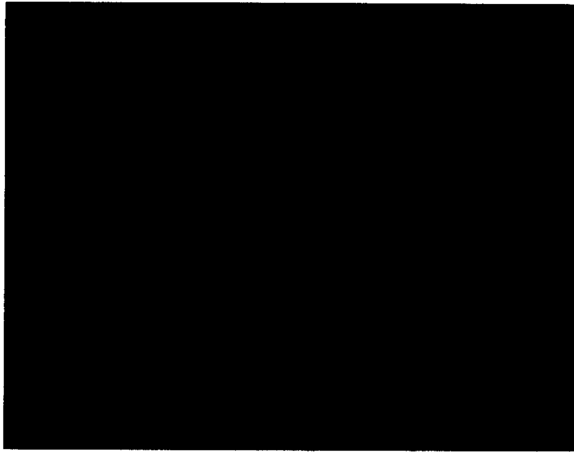


Figure 5. The splitting process for an isosurface (the layers data set).

If the image is being converted to a file for disk storage, the easiest output method involves sending it from top to bottom for later display by storing a virtual copy of the frame buffer in the multiprocessor's memory. 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 containing the corresponding line in the virtual frame buffer. After the rendering phase ends, the virtual frame buffer can be copied to a file for storage 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.

Graphics data decomposition

Several possible data decomposition methods handle storing the graphics data in a multiprocessor. One scheme involves storing data in globally shared memory, which all processors can access remotely. This method of memory access was denoted the uniformly distributed (UD) scheme. Previously,[9] I demonstrated that this method incurred a large overhead and did not scale well. A second scheme, the locally cached (LC) scheme, capitalizes on distributing the data among the memories and later moving it around for local access during the rendering phase.

Distributing data among memories

The LC memory referencing strategy involves initially storing data scattered throughout the machine's memories and using a software caching technique to bring data into local memory during processing. A similar mechanism, previously used in parallel ray tracing, is an implementation of a technique known as "shared virtual memory,"[12,13] which emulates shared memory on a message-passing system.

Instead of using a demand-driven paging scheme of shared virtual memory, the LC scheme employs explicit copying of the exact data needed for a given task. Thus, 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 parallel ray tracing implementations, since the exact data needed for a given portion of the image space is not known a priori in that type of algorithm.

During the preprocessing phase, the LC method puts polygons into bins in each processor corresponding to the screen-space areas to be assigned as tasks. (Other systems, developed independently of this one—for example, Pixel-Planes 5—have used a similar approach.) 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 first pass determines how much array memory a processor must allocate for a particular bin. The LC scheme constructs arrays (as opposed to linked lists) so that each array can later be sent out as a contiguous message to another processor, which will tile the area during the rendering phase. After processing into messages for each bin, the data is "cached" into each processor's memory during the rendering phase.

To tile an area during the rendering phase, a processor must obtain from the other processors the transformed polygon data relevant to its assigned area ($[i][j]$)—if they contain any. A processor does this by querying the other processors individually and retrieving each processor's data (if any is relevant) in contiguous messages. Note, this happens in parallel for all processors simultaneously, so the network might become clogged for a short time. However, no further communication is required after the burst of communication that provides each processor its initial task data—the data has now been "cached" into the rendering processor's memory.

Data movement during partitioning

For a processor to split another processor's pixel area, it must retrieve the data arrays in the bin structure of that pixel area. The pointers to these arrays are stored in the remote processor's memory and are readily retrieved. Ideally, the LC scheme would obtain only the data for the polygons relevant to the lower task area after the split. Since that involves stopping the remote processor's execution, it seemed not worth implementing in the shared memory system.

However, a simple way to reduce the amount of data to be copied over time is to perform a quick clip test after the arrays are received. Polygons no longer relevant to the new partitioned task are deleted. This reduces the amount of communication for further splits of an area—especially likely near the end of the computation, when most areas to be split are small and have already been split at least once. Because of the block transfer of data and local memory access, this scheme can also be implemented on a message-passing computer with minimal modifications.

Results

I tested this algorithm on a BBN Butterfly TC2000 multiprocessor using task-adaptive domain decomposition and the LC memory referencing scheme. I used three input data sets of varying complexity, two—rings and tree—from Eric Haines' SPD database.[14] The third image, layers, consisted of several transparent isosurface layers from a fusion plasma turbulence simulation generated by Tim Williams at Lawrence Livermore Lab.

Figures 5 through 7 show these test images. (The layers image shows the area borders color-coded according to which processor worked on a particular screen area.) All rendering used 16-samples-per-pixel antialiasing with Phong smooth shading

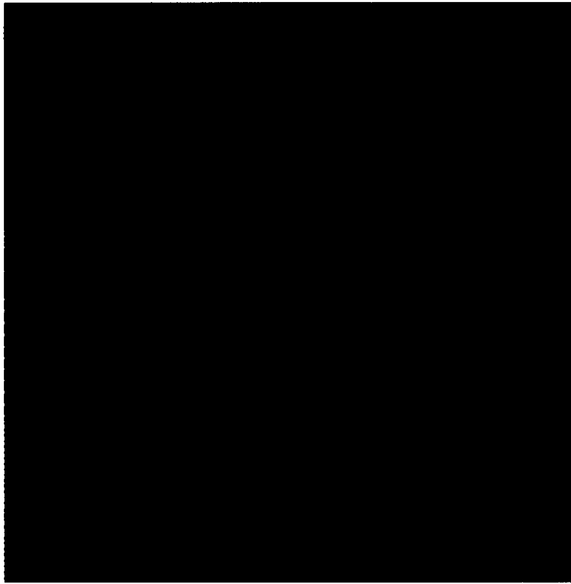


Figure 6. Rings test image.

Figure 7. Tree test image.

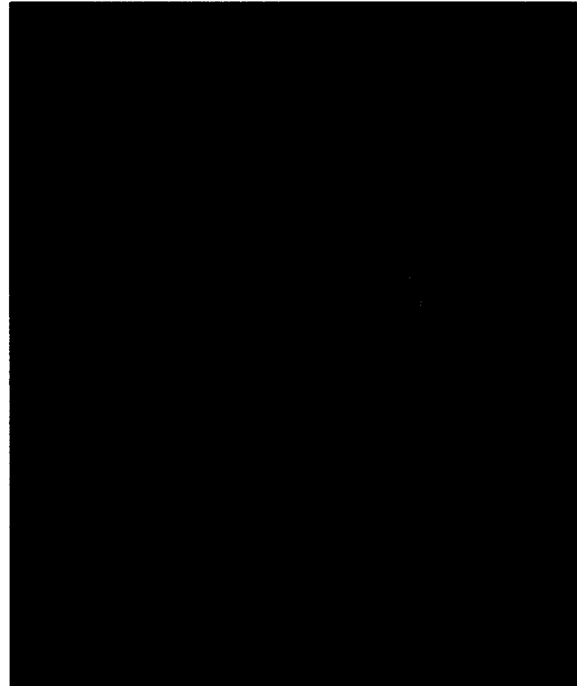
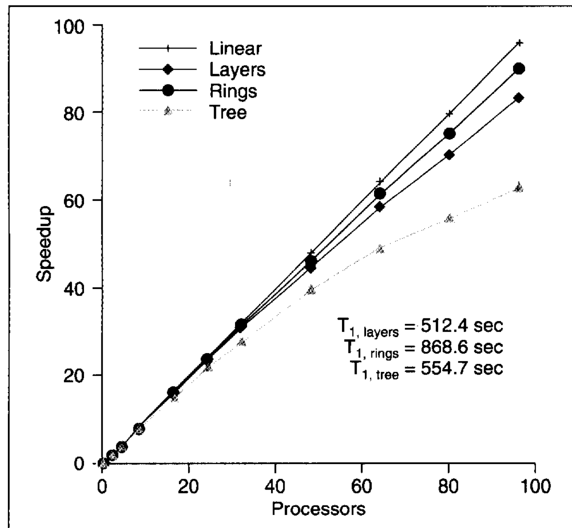


Table 1. Rendering time, speedup, and efficiency of task adaptive (LC dynamic) algorithm on 96 processors on the BBN TC2000.

	Layers	Rings	Tree
No. of Polygons	64.1K	568K	851K
Time	6.2s	9.6s	8.8s
Speedup	83.2	90.1	63.1
Efficiency	0.86	0.94	0.66

Figure 8. The speedup for the program's tiling section.



(including normal interpolation per pixel) at standard video resolution (640 × 484).

Performance

Table 1 gives the timing, speedup, and efficiency results for the tiling section of the program. Figure 8 includes a graph of the speedup. The speedup measured is self speedup (that is, performance of the program on *P* processors relative to one processor), and the program used remote memory transfers even on single processor runs. This was necessary because the input data sets would not fit into the memory of a single processor. To compensate, the extra cost of communication and code modification were measured and subtracted from the single processor time. This gave a realistic estimate of the sequential time (noted in the graph) for speedup measurements.

Using stochastic sampling antialiasing, this algorithm achieved an average rendering rate of 55,400 polygons per second. Without antialiasing, the average was 93,600 polygons per second. Measurements indicated that the parallel implementation presented here when run on a single processor had only 15 percent more overhead cost than an optimized uniprocessor renderer. The parallel implementation achieved an average efficiency of 82 percent, meaning that the parallel version provides high performance on *P* processors. In addition, the overhead of employing parallelism is not significant when compared to the optimized serial renderer.

To find the actual causes of performance degradation from linear speedup, I measured various overhead effects for the implemented program on the input data sets as a function of the

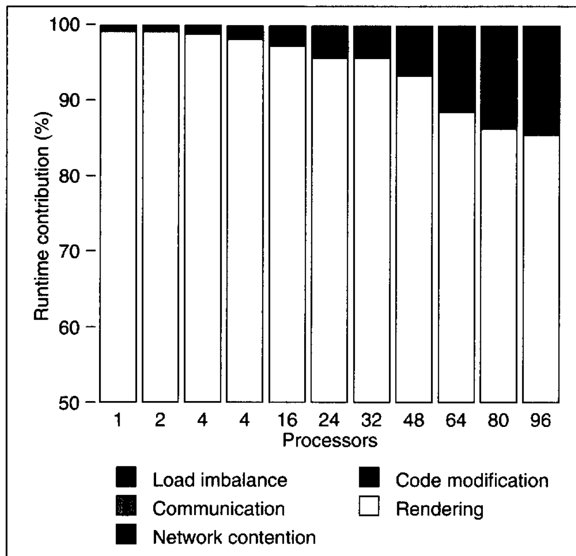


Figure 9. Measured degradation factors as a function of P for the layers data set.

Figure 10. Measured degradation factors as a function of P for the rings data set.

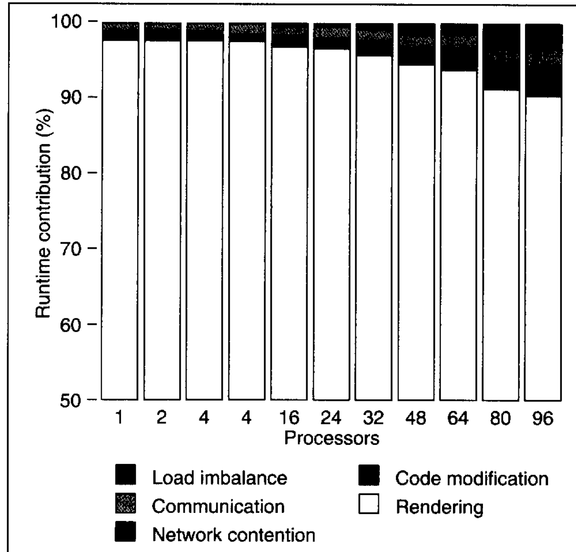


Figure 11. Measured degradation factors as a function of P for the tree data set.

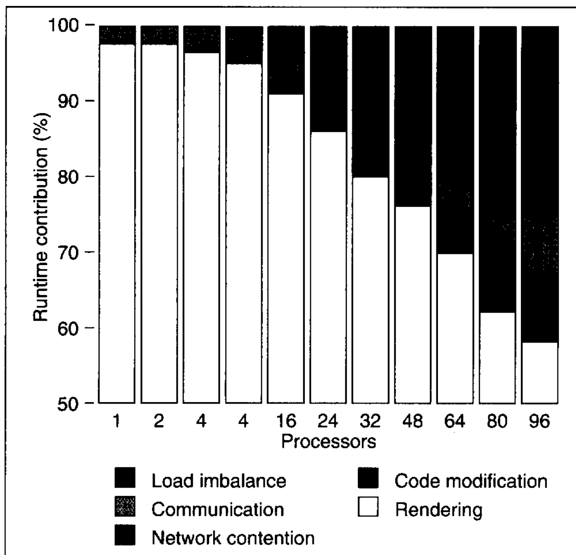


Table 2. Parallel overhead percentages at 96 processors on the BBN TC2000.

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%

number of processors (P). These effects included load imbalance, network contention, code modification to allow parallel execution, and communication. Other measured factors such as memory latency, synchronization, and scheduling represent such a small fraction (typically less than 0.1 percent) of the work that the graphs do not include them. The latter factors are small because the splitting mechanism involves very little synchronization of processes.

The graphs that depict the measured degradation factors as a function of P for the layers, rings, and tree images appear in Figures 9 through 11. The overheads shown at one processor indicate the effects of code modifications necessary to parallelize the program and the communication required to transfer the data from shared memory to local memory. Table 2 indicates the computed overheads at 96 processors.

Analysis

From the graphs, we can see that several of the overhead effects use a higher percentage of runtime as more processors are added, 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 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 stated previously, the heuristic for splitting is not particularly accurate, which explains the high load imbalance for the tree input data set in particular.

The degradation from code modification has to do with the loss of additional coherence as the number of processors—and

consequently total tasks—increases.

Communication overhead increases with P for two reasons. As P increases, the size of the task regions becomes smaller because the number of tasks is $2 \times P$, so polygons cross over into more areas (which means more copying of the polygon data). Secondly, as P increases, 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, the overall communication increases as well. But, according to the time complexity $O(N/P)$, rendering time also increases with N, so communication is not likely to become a higher percentage of cost. However, if the size and distribution of polygons changes 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 N is larger for the tree image, communication is higher—but it is also a higher percentage of overall rendering time, since the rendering time per polygon is smaller (comparing the two serial execution times).

Depth complexity, polygon area, number of polygons, microprocessor speed, communication speed, and data transfer methodology all contribute demonstrably to a parallel renderer's efficiency. For instance, doubling the output image resolution obviously increases the work without increasing the overheads, so the program appears more efficient. The results here indicate that the LC scheme exploits data locality at a minimal expense of communication overhead.

This algorithm does have deficiencies in dealing with image-space tasks having a high degree of local complexity, since these types of scenes may not be amenable to the splitting mechanism. For example, in flight simulation, data can become concentrated at the horizon. But, we can still maintain reasonable speedup, even under these adverse conditions, as exemplified by the tree input data set. The task-adaptive algorithm provides a compromise approach that can handle a moderate amount of local image complexity with reasonable performance.

Conclusion

The main goal of this research was to achieve good speedup and efficiency using a parallel algorithm for rendering complex geometric scenes. The locally cached memory strategy reduces message traffic and supports local memory referencing on a multiple instruction, multiple data stream (MIMD) computer. The absolute performance of this software algorithm already ranges up to 100,000 antialiased Phong-shaded polygons per second (on a four-year-old architecture). However, a good measure of its success is its average 82 percent efficiency using 96 processors with minimal parallel overhead compared to a uniprocessor implementation. This indicates that even faster microprocessors and/or larger parallel computers can increase rendering rates further. In addition, using this algorithm as part of a scientific simulation system on a parallel computer allows direct memory transfer of data and supports fast creation of single images or "movies in minutes." Future enhancements might include resolving the memory and time problems associated with access to texture and shadow maps. ❑

Acknowledgments

Thanks go to all those who read and critiqued this article and its previous versions: Rick Parent, Nelson Max, Brian Cabral, Mohan Kankanhalli, Becky Springmeyer, Charles Grant, and Derek Paddon. Some of this work was performed under the auspices of the US Department of Energy by the Lawrence Livermore National Laboratory under contract number W-7405-ENG-48. Thanks also go to the anonymous *CG&A* and Parallel Rendering Symposium reviewers who provided many useful suggestions on ways to improve the manuscript.

References

1. G. Abram, *Parallel Image Generation with Anti-Aliasing and Texturing*, PhD dissertation, University of North Carolina at Chapel Hill, 1986.
2. W.R. Franklin and M.S. Kankanhalli, "Parallel Object-Space Hidden Surface Removal," *Computer Graphics* (Proc. Siggraph), Vol. 24, No. 4, Aug. 1990, pp. 87-94.
3. S. Whitman, "Parallel Graphics Rendering Algorithms," *Proc. 3rd Eurographics Workshop on Rendering*, Consolidation Express, Bristol, UK, May 1992 pp. 123-134.
4. E. Fiume, A. Fournier, and L. Rudolph, "A Parallel Scan Conversion Algorithm with Anti-Aliasing for a General Purpose Ultracomputer," *Computer Graphics* (Proc. Siggraph), Vol. 17, No. 3, July 1983, pp. 141-149.
5. T.W. Crockett and T. Orloff. "A MIMD Rendering Algorithm for Distributed Memory Architectures," *Proc. Parallel Rendering Symp.*, ACM Press, New York, Oct. 1993, pp. 35-42.
6. M. Kaplan and D.P. Greenberg, "Parallel Processing Techniques for Hidden Surface Removal," *Computer Graphics* (Proc. Siggraph), Vol. 13, No. 2, July 1979, pp. 300-307.
7. D.S. Whelan, *Animac: A Multiprocessor Architecture for Real-Time Computer Animation*, PhD dissertation, California Institute of Technology, Pasadena, Calif., 1985.
8. D.R. Roble, "A Load Balanced Parallel Scanline Z-Buffer Algorithm for the iPSC Hypercube," *Proc. Pixim 88*, Hermes, Paris, France, Oct. 1988, pp. 177-192.
9. S. Whitman, *Multiprocessor Methods for Computer Graphics Rendering*, AK Peters, Wellesley, Mass., 1992.
10. S. Molnar, J. Eyles, and J. Poulton, "PixelFlow: High-Speed Rendering Using Image Composition," *Computer Graphics* (Proc. Siggraph), Vol. 26, No. 2, July 1992, pp. 231-240.
11. A.J. Myers, "An Efficient Visible Surface Algorithm," Report to the NSF for Grant No. DCR74-00768 A01, July 1975.
12. D. Badouel, K. Bouatouch, and T. Priol, "Ray Tracing on Distributed Memory Parallel Computers: Strategies for Distributing Computations and Data," Course Notes for Course 28, Siggraph, ACM Press, New York, Aug. 1990, pp. 185-198.
13. S. Green and D. Paddon, "Exploiting Coherence for Multiprocessor Ray Tracing," *IEEE CG&A*, Nov. 1989, Vol. 9, No. 6, pp. 12-26.