images using $R = 24$ and a 1:1 image aspect ratio for the regions are shown in figure 5.11. The speedup graph can be seen in figure 5.12. Although an aspect ratio of 1:1 for the regions was desired for all values of P, this could not be achieved for each combination of R and P. Therefore, an aspect ratio as close to 1:1 was used when needed (all regions have the same aspect ratio for a given value of P).

We will now analyze this algorithm with regard to the issues identified previously to determine the various overhead percentages.

5.1.2.4. Scheduling (0.004% - 0.013%)

In this decomposition method, it is necessary to schedule all regions as separate tasks, but the number of regions varies as the number of processors is increased. To determine the maximum number of regions to be scheduled, we used the granularity ratio (R) of 24 to 1 on 96 processors, which results in a total of 2,304 tasks to schedule. The time to run a task consisting solely of the background color for one of these areas has been measured as T_{back} = 2.3 msec. The reason this is larger than in the scan line decomposition case is that a separate block transfer is necessary for each scan line in the area and each block transfer requires a setup cost. Since 2.3 msec is exactly the overhead time to schedule 96 tasks on 96 processors (T_{sched}), scheduling will not become a bottleneck in this algorithm. The overhead due to scheduling then involves plugging the values for this algorithm into equation 4.5, as shown below. This time represents a percentage overhead for the different images, ranging from 0.004% for the mountain image to 0.013% for the stegosaurus image.

$$\text{Scheduling } \% = \frac{\frac{(95 \bullet 96)}{2} \bullet 24\ \mu\text{sec} + (2{,}304 - 96) \bullet 24\ \mu\text{sec}}{T_p \bullet 96} \bullet 100\ \% \qquad (5.5)$$

5.1.2.5. Memory Latency (1.4% - 3.6%)

In this algorithm, memory latency is measured the same way as the previous method, namely by counting the number of remote references and using equation 4.6 to determine the overall percentage. Fewer references to the shared data are needed than in the previous approach due to the coherence maintained within a region. Consequently, the overall latency is reduced.

Rectangular Region Algorithm (UD Scheme) Performance

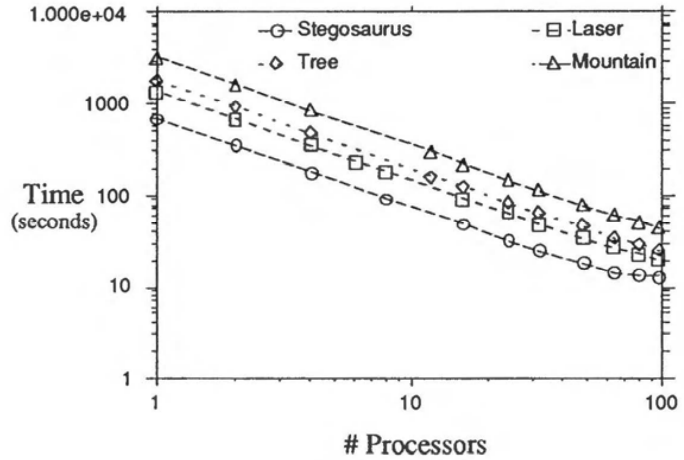

Figure 5.11: Rectangular region performance (UD scheme)

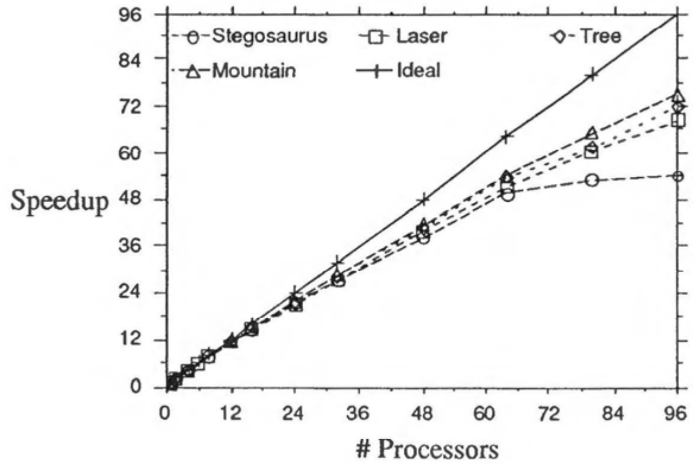

Figure 5.12: Speedup of rectangular region decomposition (UD scheme)

The percentage time of latency for this algorithm for the different test images ranges from 1.4% for the stegosaurus image to 3.6% for the mountain image. The latency increases corresponding to an increase in dataset size, which is the same phenomenon observed previously.

Most remote references occur for the first scan line of a region since it is necessary to retrieve the data from remote memory and store it in local data structures. Once this is accomplished, the majority of references are local, with the exception being a small amount of remote referencing required in the anti-aliasing portion of the code (this would be the same for each of these algorithms). The remote referencing in the anti-aliasing section stems from the need to obtain the plane equation for each polygon for stochastic sampling purposes. Since the previous algorithm incorporates no vertical scan line coherence, all the data for each scan line must be referenced remotely. The rectangular region partitioning scheme capitalizes on coherence within the region so that remote referencing is reduced.

5.1.2.6. Network Contention (5.6% - 33.1%)

The network contention is calculated in the same manner for this algorithm as it was for the last one. Using this technique, the percentage effect of network contention for the various test images varies from 5.6% for the tree image to 33.1% for the stegosaurus image.

The contention in this algorithm as compared to the parallel scan line approach is worse for the stegosaurus and Laser images, but improved for the tree and mountain images. It is difficult to speculate as to the reason for this without further image analysis. Regardless, one can see that even with a reduced number of references (as compared to the parallel scan line approach), contention is still a major degradation factor. Most of the increase in network contention occurs as the number of processors is increased from 64 to 96 processors, indicating that the switch network becomes overloaded with requests somewhere in this range.

5.1.2.7. Load Imbalance (4.3% - 11.5%)

The granularity ratio in this algorithm provides a better load balanced system than the last algorithm, although network contention increases the execution time and this varies the overall finishing times. The load imbalance percentages measured at 96 processors for the different images varies from 4.3% for the mountain image to 11.5% for the tree image. When compared to the previous algorithm, the load imbalance overhead is less for this algorithm, with the

exception of the tree test image. The granularity ratio comparison in figure A.3 for this image indicates that load balancing is not particularly good at any value of R and in fact gets worse after $R = 24$.

In general, though, this algorithm yields better load balancing than the scan line approach since the granularity ratio provides enough tasks to minimize the load imbalance over a wide range of processor configurations.

5.1.2.8. Code Modification (7.9% - 9.6%)

This algorithm has a different amount of coherence overhead than the scan line algorithm since rectangular regions are generated as tasks. Due to the rectangular nature of the regions, coherence is taken advantage of in both the vertical and horizontal directions within a single task. On the other hand, the lack of vertical scan line coherence at the beginning of an area results in extra work required to start the first scan line of a region. In addition, the lack of horizontal coherence at the boundary to the left causes an overhead of interpolating parameters for polygons which extend beyond this boundary.

The code modification overhead is measured the same as before using equation 4.8. Based on the measured values, the overhead percentages vary from 7.9% for the mountain image to 9.6% for the tree image. Considering the fact that there are many more tasks used in this scheme versus the parallel scan line approach (2,304 vs. 484), this overhead factor does not seem out of line in comparison.

5.1.2.9. Explanation of Results

The rectangular region decomposition scheme achieves reduced overheads primarily in memory latency and to some degree in network contention and load balancing, in comparison to the parallel scan line algorithm. The reduction in latency is due to the fact that most remote referencing occurs for the first scan line of an area, and this effect is reduced in the rectangular region algorithm. This suggests that the rectangular region decomposition algorithm will perform better than the scan line algorithm in the general case due to its performance advantages in the tests given. The scan line algorithm exhibits poor scalability as P is increased since load balancing will suffer as the number of scan lines approach the number of processors. The rectangular region algorithm uses a fixed granularity ratio which allows better load balancing as the number of processors is increased; thus its scalability is superior to the parallel scan line approach.

It is important to note that this algorithm still has its share of problems. Network contention still represents a significant overhead. The code modification overhead is not reduced in this algorithm in comparison to the scan line approach. A comparison of the major degradation factors is given in figure 5.13.

Since the number of remote references is reduced in this algorithm but contention was not significantly reduced, another tactic is necessary to solve this problem. Consequently, it is necessary to implement a different memory referencing scheme that is designed to reduce the network contention noted in parallel implementations. This memory referencing strategy is referred to as the locally cached (LC) scheme and is described in the next section.

5.1.3. Rectangular Region Decomposition (LC Scheme)

The algorithm described here is implemented exactly the same as the last one, with the only exception being the remote memory referencing strategy. A brief description of this strategy, denoted the Locally Cached or LC scheme, follows. Instead of referencing globally

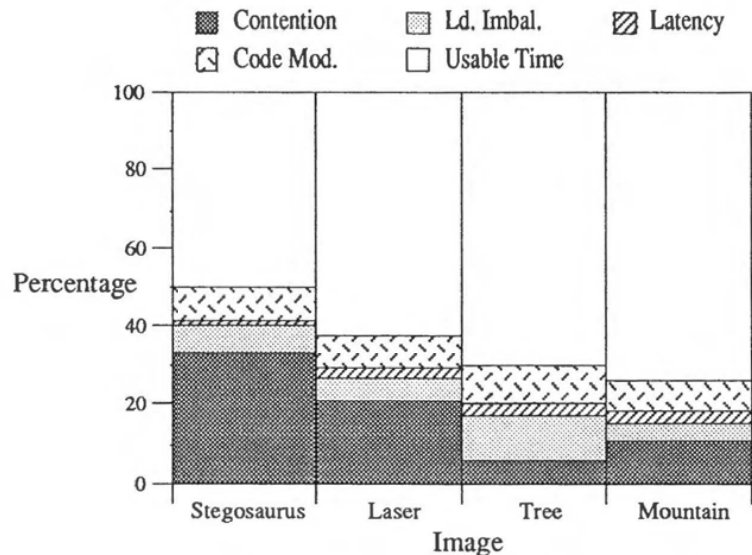

Figure 5.13: Degradation factors for rectangular region decomposition, (UD Scheme, $P = 96$)

shared data remotely, the data is cached into the local memory module prior to referencing, thus allowing it to be accessible quickly. Although others have used an elaborate software caching mechanism for computer graphics rendering ([Gree89] and [Bado90]), we rely on the fact that the exact data needed for a given task can be copied directly to each processor prior to the computation of a given region. If a data element is relevant to more than one processor, it is copied to all processors which would reference it, incurring a space penalty. The extra memory required due to duplication is shown in the appendix in figures A.9, A.10, A.11, and A.12. These figures show the duplication of data by copying data elements as a function of the total number of regions. Although the extra memory required is wasteful, a tradeoff of space versus time is necessary to achieve faster memory referencing than the previous Uniformly Distributed (UD) approach. The cost of non-local memory access is eliminated by block transferring data from its global storage location to the local memory of the processor(s) that need it. Details of the LC scheme are given in chapter 6.

5.1.3.1. Granularity Ratio

The granularity ratio R was re-evaluated for this algorithm to see what a good ratio would be, since a different memory referencing scheme is used. This ratio was tested at values of 2, 4, 8, 12, 16, 20, 24, 28, 32, 36, and 40 using the maximum configuration of 96 processors. Figure 5.14 shows the comparison for the Laser image as an example. In the appendix, figures A.5, A.6, A.7, and A.8 show the data for all the images. The downward slope of the curves is primarily due to a reduction in load imbalance as a higher granularity ratio is used. Then the curves continue upwards after a point since the other culprits introduce more overhead cost for the higher ratios. The minimum point on the curve is the optimal granularity ratio (R) to use for a particular scene. Each scene exhibits different characteristics which affect the choice of this optimal R so a compromise must be made so that a single value of R may be used in the general case.

In this case, the choice of a good ratio spans a broader range than in the UD scheme. The reason for this is the reduction in communication and contention costs versus the previous method. It seems like a good choice for R can be anywhere in the range from 12 to 1 up to 32 to 1. Since $R = 24$ was chosen for the previous algorithm and the performance for that ratio with this scheme is nearly optimal in most cases, this value will again be used. This ratio should provide

good results for most imagery, given this machine configuration. Graphs for the time and speedup of this algorithm with the LC memory referencing scheme are given in figures 5.15 and 5.16.

The overhead factors for the rectangular region decomposition are now discussed, using the LC memory referencing scheme evaluated at 96 processors.

5.1.3.2. Scheduling (0.004% - 0.017%)

The time to run a background task (T_{back}) in this scheme is the same time as the previous one, since the only difference between the two is the memory referencing, which does not affect the background task. This algorithm is faster than the previous one, so the overhead percentage is slightly higher. Equation 5.5 is again used for evaluating the overhead due to scheduling for this algorithm. Based on this equation, the overhead due to scheduling varied from 0.004% overhead for the mountain image to 0.017% overhead for the stegosaurus image.

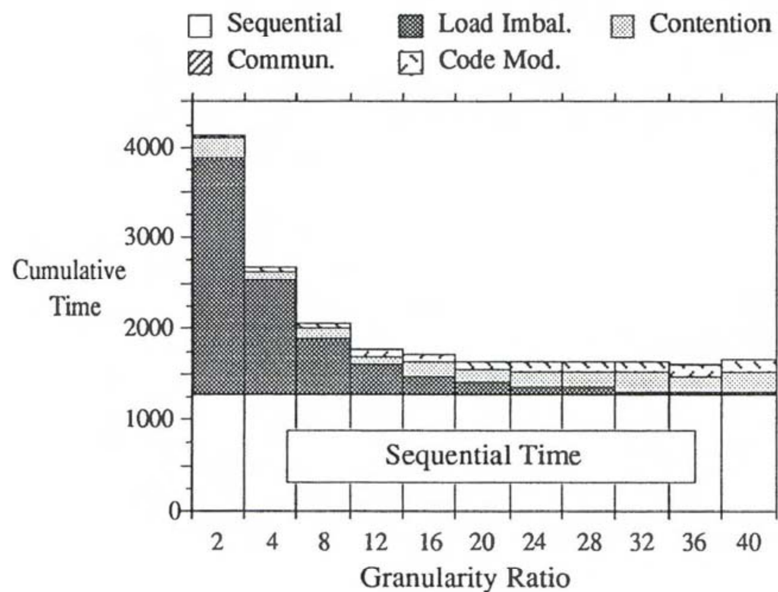

Figure 5.14: Comparison of ratios for Laser image

Rectangular Region Algorithm (LC Scheme) Performance

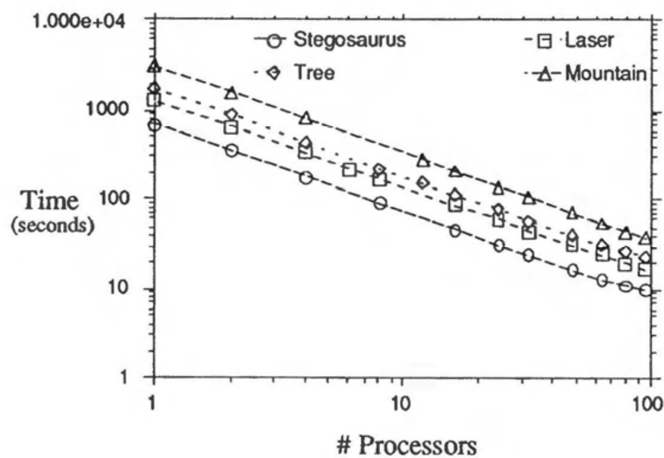

Figure 5.15: Tiling time for rectangular region partitioning (LC)

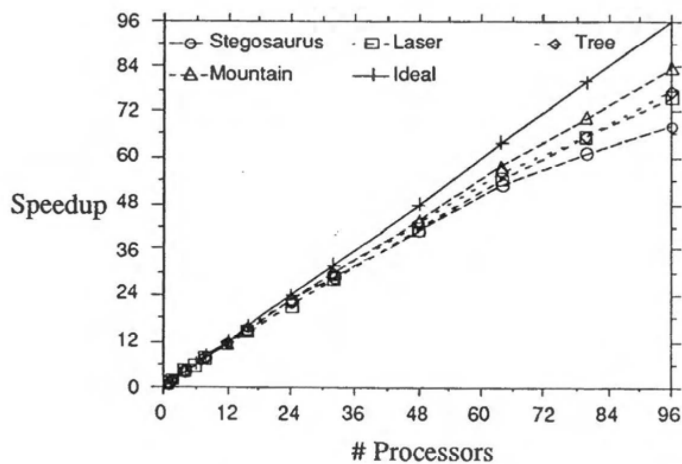

Figure 5.16: Speedup for rectangular region partitioning (LC Scheme)

5.1.3.3. Communication Overhead (0.03% - 0.05%)

Instead of latency due to remote referencing as in the UD case, communication occurs in blocks in this algorithm, resulting in a different overhead factor. This communication overhead is not present in the UD referencing method. Recall that this overhead is measured by a count of the total number of bytes transferred in the system during the computation. Using equation 4.7 given in the previous chapter, the communication overhead range is 0.03% for the stegosaurus image up to 0.05% for the mountain image. One can see that these values represent a significant drop in the amount of time necessary to transfer data in the system. Since the data is copied into local memory, all future references occur locally. This means that the total amount of data transferred is also reduced in comparison to the UD referencing scheme.

5.1.3.4. Network Contention (3.1% - 16.3%)

Although there is some overhead necessary to set up the blocks of data to be transferred in this algorithm, the deficit is more than made up for by a reduction in network contention when compared to the UD scheme. The calculated network contention overhead varies from 3.1% for the tree image to 16.3% for the stegosaurus image. The contention in this scheme is significantly less than in the previous one. This indicates that the locally cached memory referencing scheme does in fact reduce the messages in the system, which results in reduced chances for a blocked switch node.

5.1.3.5. Load Imbalance (4.5% - 11.1%)

The load imbalance in this algorithm is measured the same as before, using equation 4.11. The overhead percentages for load imbalance vary from 4.5% for the mountain image to 11.1% for the tree image. These values are nearly the same as those from the previous algorithm, which is to be expected since they both use the same partitioning method.

5.1.3.6. Code Modification (5.4% - 6.4%)

The code modification overhead using the LC scheme is less than in the UD scheme in all cases except the tree image. The measured overhead ranges from 5.4% for the mountain image to 6.4% for the stegosaurus image. The probable reason for the difference is that communication is not completely factored out of the measurement method. Recall that the measurement technique used for this

overhead involves timing the program running on a single processor, using MIN memory modules. The UD scheme involves remote referencing to these memory modules, while the LC scheme does not. Although the communication cost is factored out of the measured time by counting the number of remote references or bytes transferred respectively, it is impossible to factor out the system overheads. Since the LC scheme will not likely include these to the degree that the UD scheme does due to the method of memory allocation and deallocation, the resultant code modification is a generally lower figure here.

5.1.3.7. Explanation of Results

Latency is no longer a factor using this memory referencing scheme, and although communication overhead is introduced, it is minimal. The change in memory referencing scheme also affects the overall code modification, as reported above.

The load imbalance is nearly the same as the previous algorithm, with the slight difference due to the effect of reduced contention in this algorithm. The chart in figure 5.17 indicates the overheads for the various images.

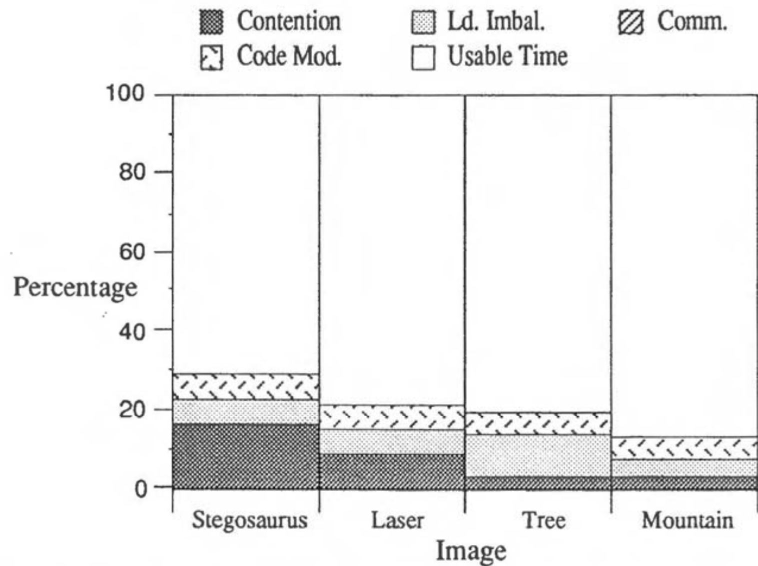

Figure 5.17: Degradation factors for rectangular region decomposition (LC Scheme, $P = 96$)

As one can see from the chart, network contention is still a problem, although it is significantly reduced in comparison to the UD scheme, especially for the more complex imagery. Load imbalance is also a problem, although the algorithm should scale up fairly well, especially when one takes into account the reduced contention using this memory reference strategy.

In the next section, we introduce another partitioning scheme which achieves even better load balancing.

5.2. Data Adaptive Partitioning Scheme

In a data adaptive algorithm, load balancing is achieved by constructing tasks which are estimated to take nearly the same amount of time. By using image space partitioning in a parallel graphics rendering program, tasks can be determined based on the location of data within the image. If the task work can be accurately predicted by using a heuristic, then the granularity ratio R can be reduced, resulting in less communication and scheduling. In fact, if the adaptation can produce exactly the same size tasks in terms of work, R can be reduced to 1. It is not generally possible to pick a very accurate heuristic since factors such as depth complexity, polygon area, and anti-aliasing all affect the time it takes to render a pixel. Pre-processing of the data cannot take all of these factors into account; otherwise it would require too much time. Following is a brief description of several algorithms which fall under the data adaptive category.

Whelan [Whel85] uses a data adaptive approach in his Median Cut algorithm, although his application was for a hardware architecture. His primary motivation was to reduce the scheduling overhead associated with the type of dynamic task assignment used in the algorithms discussed thus far. This is not necessary in a software multiprocessor approach since the Uniform System provides scheduling with a very small overhead. Whelan's approach involves task partitioning so that each task contains the same number of polygons. He uses the centroids of the polygons to determine their screen space location; however, extensive sorting is necessary to determine the locations to place the screen space partitions. His algorithm provides excellent load balancing, but the overhead cost of creating the areas outweighs the benefit of adaptive partitioning.

Roble's [Robl88] approach is another data adaptive method which also uses polygon location as a heuristic for determining tasks. His approach involves a large amount of communication prior to the tiling phase, and thus exhibits too much overhead as well.

Although there are many different decomposition methods that fall under the data adaptive method, one algorithm was chosen as a representative example for implementation. The goal here was to eliminate the excess overhead associated with this type of approach. This algorithm is described next.

5.2.1. Top-down Decomposition

A partitioning scheme similar to Whelan's Median Cut algorithm is used which takes comparatively less time to determine the task partitions. This scheme is based solely on the number of data elements in a region, regardless of the location of their centroids. The heuristic in this algorithm is based on the assumption that the number of polygons in a region is linearly related to the time it takes to tile that region. Using this simple heuristic, good load balancing can be achieved with a small overhead. The LC memory referencing scheme is used in the implementation of this algorithm based on the results shown in the previous section. The implementation is described below.

A 2D mesh is created as in the rectangular region decomposition, but this time the mesh is 4 times as dense (i.e. $\#regions = R \bullet P \bullet 4$). Polygons are placed into the mesh during the front end portion of the program as before, based on their screen space bounding boxes. Prior to tiling, adjacent meshes are combined hierarchically and a sum of the combined regions is stored in a tree data structure. This process is repeated until a point is reached where the entire screen is in a single region. Then, a data structure is created which consists of a hierarchical binary tree of counts referring to the number of data elements in each area.

After the tree is created, it is traversed in top-down fashion and the area with the most polygons at a given point is then split into its two components. This process is repeated by considering all areas created thus far, splitting the one with the next most polygons. The splitting process is stopped when the desired number of tasks has been reached. A count of the number of polygons in each small area is used, so it is not necessary to sequentially go through the entire list of potential polygons to determine which polygons are relevant to each area at this time. The limiting factor in the splitting is the leaf level, which is why a fairly dense mesh is created at the beginning. An example of this type of decomposition is illustrated graphically in figure 5.18 and also in color plate 3.

After the tree has been traversed, each of the regions is available for rendering in parallel. Some computational overhead exists for this scheme prior to the tiling phase, but fewer tasks are created than in the previous rectangular region approach. Figure 5.19 shows the performance for the various images using the data adaptive approach, with a value of $R = 10$. This value of R was determined empirically similar to the methods used previously. It is less than the value needed for the rectangular region scheme for good load balancing. A perfect match would result in a ratio of $R = 1$ but that situation is almost impossible to achieve using a heuristic which has minimal overhead cost. The relative speedup for the top-down scheme is shown in figure 5.20.

The time to build the tree data structure is not included in these timings since it is not part of the tiling section of the program. This time is fairly small anyway, but it is included in the overall algorithm comparison presented in chapter 6. We now analyze the top-down decomposition method with regard to the possible overhead factors.

5.2.1.1. Scheduling (0.003% - 0.01%)

This partitioning scheme uses regions that are not the same size, so each background task does not take the same time. The areas consist of groups of scan lines as before, but the number of scan lines and their size differ.

The average time to render the different background areas was measured for the different images. The results were fairly consistent, with an average background task time of 4.48 msec.

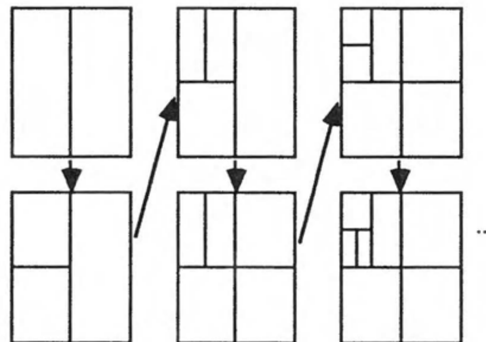

Figure 5.18: Top-down partitioning scheme

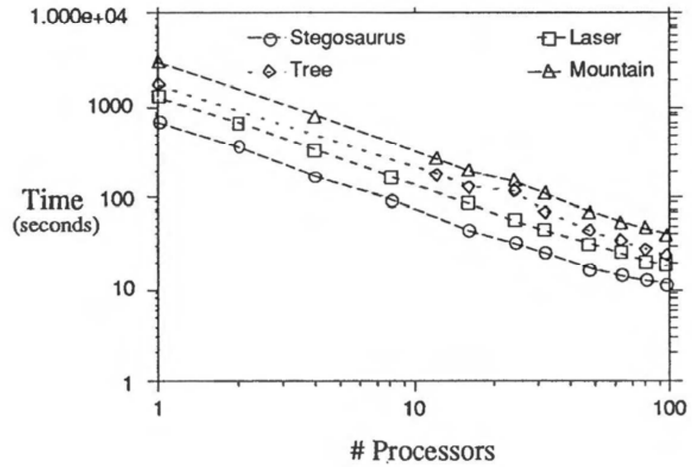

Figure 5.19: Top-down decomposition performance

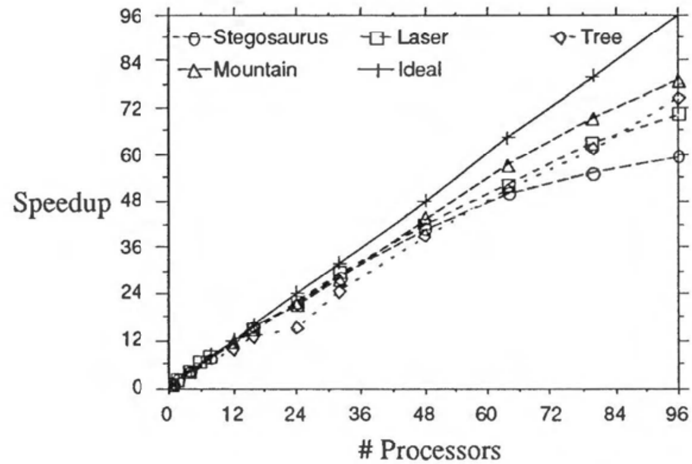

Figure 5.20: Speedup of top-down decomposition method

This is more than T_{sched} (2.3 msec), which is the time it takes to schedule 96 tasks, so no bottleneck will occur due to scheduling. As before, the scheduling overhead is determined by plugging the values for this algorithm into equation 4.5, as shown in equation 5.6. Using this equation for 96 processors, the scheduling overhead for the test images ranges from 0.003% for the mountain image to 0.01% for the stegosaurus image.

$$\text{Scheduling \%} = \frac{\frac{(95 \bullet 96)}{2} \bullet 24\ \mu\text{sec} + (960 - 96) \bullet 24\ \mu\text{sec}}{T_P \bullet 96} \bullet 100\ \% \qquad (5.6)$$

5.2.1.2. Communication Overhead (0.02% - 0.04%)

The communication overhead is measured the same way as in the previous algorithm, by determining the number of bytes transferred in the system and using equation 4.7 to calculate the overhead. The values vary from 0.02% for the stegosaurus image to 0.04% for the mountain image. The communication overhead percentage in this algorithm is slightly less than in the rectangular region (LC) method since there are fewer areas.

5.2.1.3. Network Contention (11.8% - 34.9%)

Unfortunately, network contention is a significant factor in this algorithm, even more so than in the previous one. The network contention overhead ranges from 11.8% for the mountain image to 34.9% for the stegosaurus image. The reason for this increase in network contention is given here.

As was explained at the beginning of this section, a 2D dense mesh is created, from which small regions are clustered together to form tasks. The LC scheme requires communication from each of these small regions which form the larger clusters in order to obtain the data necessary for rendering a particular task area. Figure 5.21 illustrates this situation.

In order to render the cluster composed of sub-regions 1, 2, 3, and 4, it is necessary to retrieve the polygons from these sub-regions. This requires a block transfer from each of the sub-regions, whereas the rectangular region algorithm requires only one block transfer for the entire region. There may be even more than four sub-regions which are part of a larger cluster. Although the total amount of data is not large (evident by the communication factor given previously), the number of messages is higher than in the rectangular region

algorithm due to this copying from sub-regions. In addition, the frequency of these communications is greater since they proceed one right after another. The block transfer mechanism in the GP1000 which is utilized in the LC scheme holds a message path open for as long as it is needed to transfer the data. Therefore, more collisions are likely to occur in this algorithm due to the increased number of messages required, resulting in high network contention..

5.2.1.4. Load Imbalance (1.5% - 6.9%)

The goal of better load balancing was achieved in this algorithm, using a smaller granularity ratio than the rectangular region approach. The percentage overhead for load imbalance varies from 1.5% for the stegosaurus and mountain images to 6.9% for the Laser image. This algorithm achieves better load balancing than the previous algorithm, with minimal expense required to build the hierarchical tree data structure. It therefore overcomes the limitation noticed in Whelan's and Roble's algorithms, which also used a data adaptive scheme. More details on the overhead time required for the tree construction are given in chapter 6.

5.2.1.5. Code Modification (2.5% - 3.3%)

The overhead due to code modification is much smaller than in the rectangular region approach. This overhead ranges from 2.5% for the mountain image to 3.3% for the Laser image. The reason for the reduction is that there are fewer total tasks and each task area is larger, reducing the overall coherence loss.

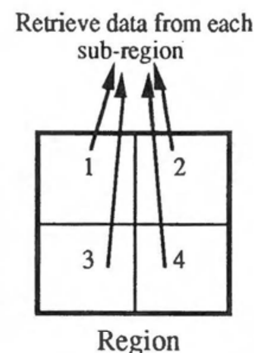

Figure 5.21: Block transfer of data from sub-regions for top-down decomposition

Looking back at figure 5.21, it can be seen that it is likely that a number of polygons cross over several sub-regions but are singularly contained within the main region to be rendered. Unfortunately, short of a direct comparison of all polygons there is no way to detect if a given sub-region is sending the same polygon as another sub-region, due to the usage of the LC memory referencing scheme. If a polygon is sent from two or more different sub-regions as a result of its overlapping these regions, that polygon is rendered more than once. This is a direct function of the duplication factor for the given mesh size. The overhead of this occurrence is difficult to determine since not all polygons which are duplicated are rendered more than once, only those that are duplicated across sub-regions and are part of the same higher region. This duplication of rendering is included in the code modification overhead given previously.

5.2.1.6. Explanation of Results

The goal of the data adaptive top-down scheme is to maintain good load balancing. The implementation here achieves this goal, but due to the method of data transfer required by the LC scheme, additional contention is introduced. There is also the additional cost of constructing the tree data structure, but this cost is offset by the reduction in the number of regions resulting in reduced code modification overhead. The times for the tree building are not included here since this chapter deals with a comparison of the algorithms' tiling section, but they are given in the next chapter. The chart in figure 5.22 shows the overhead comparison for the various images.

It can be seen that all of the overhead factors have been reduced compared to the previous approaches, with the exception of network contention. This algorithm requires a dense mesh to be created for determination of the regions. As P is increased, the mesh will need to be even denser, and this may result in even higher network contention overhead and duplication of polygons. As a result, this algorithm may not exhibit good scalability for very dense meshes.

It might be possible to create the mesh in some other manner which does not result in as much overhead, but other methods were not explored here. For example, if one were to try to determine the clusters from the top down, a pseudo-parallel method could be used whereby tasks are spawned off according to the level of the tree traversed. A large amount of synchronization would be necessary to implement this technique, and the result might involve more overhead than in the current implementation. One of the problems with the algorithms discussed thus far is that they rely on a good choice for the granularity ratio.

Unfortunately, empirical testing must be employed to determine what the best value is for a given situation. In fact, it is possible that the value might need to be changed when the number of processors is increased significantly beyond 96. The next section covers an algorithm that does not rely on a pre-determined granularity ratio, but instead achieves load balancing by dynamically partitioning existing tasks into smaller ones when a processor needs work.

5.3. Task Adaptive Partitioning Scheme

The task adaptive methodology relies on an algorithm's capability to dynamically partition tasks as the program is running. If tasks cannot be adaptively partitioned, then that algorithm is not well suited for dynamic task splitting. Fortunately, the serial scan line Z-buffer algorithm upon which these parallel algorithms are based consists of independent regions, and there is no required order of execution between these regions. The task adaptive algorithm consists of the following steps:

1. When a processor needs work (call this processor P_S), it searches among the other processors for the one which contains

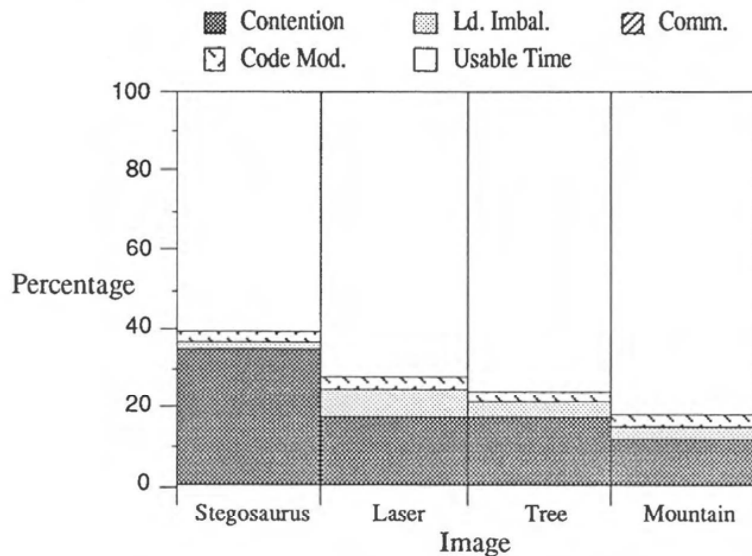

Figure 5.22: Degradation factors for top-down decomposition (P = 96)

the most amount of work left to do (call this processor P_{max}).

2. The P_s processor then sets a lock preventing any other processor from splitting P_{max}.
3. P_s partitions P_{max}'s work into two segments; the first segment goes to P_{max} and the second segment goes to P_s.
4. P_s then copies the data necessary for it to work on the second segment.
5. P_s unsets the lock and starts doing its work.

This task adaptive scheme could be tacked onto any of the previous algorithms, so that additional load balancing would be ensured toward the end of the computation. For the implementation here, the rectangular region decomposition scheme was chosen as a basis parallel algorithm since it is fairly simple to work with in developing the heuristic for step 1. A description of this parallel algorithm is given next.

Instead of attempting to choose an optimal granularity ratio, the number of areas is initially set equal to the number of processors (R = 1). When a processor has finished computing its area, it executes steps 1 through 5 above. In order to do this, it was necessary to come up with a method for determining the amount of work a given processor has left to do. Since all of the areas are the same size, the number of scan lines left to render in a particular area is used as an indication of how much work there is left on a given processor. This proceeds as follows.

During the tiling portion of the computation, each processor updates a shared variable corresponding to the number of scan lines it has left to compute. P_s quickly runs through these variables checking for the processor that has the maximum number of scan lines left. Once it finds the processor with the most scan lines left (P_{max}), P_s proceeds to split P_{max} as is shown in figure 5.23. Color plate 4 shows an illustration of this process after completion. P_{max} is not interrupted during this time.

The splitting mechanism prevents a race condition from occuring if several processors attempt to split the same region simultaneously or, alternatively, P_{max} attempts to work on a portion of its region which is to be split. The first instance is solved by using a test and lock methodology in which a splitting processor checks to see if P_{max} is currently being split and if so, this splitting processor finds another processor to split. The second case is solved by updating a shared variable which P_{max} checks to determine the last scan line for it to calculate. Neither case requires P_{max} to be interrupted from its work, thus avoiding any synchronization delay.

A threshold must be chosen which limits partitioning of tasks when the cost of the actual partition exceeds the cost of running the task serially. Through empirical testing, it was determined that partitioning a task with only two scan lines left does in fact yield good performance, so this was the threshold limit set. A task which contains no polygons is not allowed to be split since the only work involved is sending the scan lines to the virtual frame buffer.

Since P_s splits P_{max} into two tasks, it makes sense for P_{max} to continue working on the upper task while P_s takes the lower one. This allows coherence to be maintained in P_{max}'s region without any additional overhead. The performance for the task adaptive scheme is given in figure 5.24. The speedup for this scheme is shown in figure 5.25. Although a bit of extra coding is required to handle the splitting operation and data retrieval processes, the algorithm is fairly straightforward to implement.

During the splitting process, it is necessary for the P_s processor to obtain data from the P_{max} processor. Instead of determining exactly which data is relevant to the region that P_s will work on and retrieving only this data, it is simpler for P_s to retrieve all of the data from P_{max} and discard the portion that is not relevant to this new region. This requires a bit of extra communication, but the overhead is minimal compared to any method where either P_s or P_{max} would try to determine the exact relevant data. This is due to the fact that extra synchronization would be required in determining the exact dataset, whereas the "copy and discard" method requires no synchronization at all.

We now analyze the task adaptive scheme with regard to the various overhead factors. One of the problems in determining these

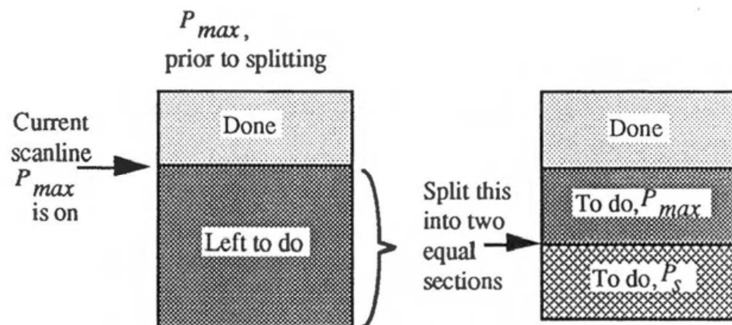

Figure 5.23: Dynamic splitting of regions for task adaptive scheme

factors is the measurement of the total number of tasks. A new task occurs when a processor tries to split another region. The task time includes the time to split a processor's work plus the rendering time. Since the number of tasks varies somewhat depending on the run, it was determined based on an average of five runs. This number varied by less than 1%, so the average is a fairly good indication of what might be considered the actual number of tasks.

5.3.1. Scheduling (0.00006% - 0.00023%)

The number of areas in this scheme is not known ahead of time since the tasks adapt to the work available. Once all of the regions are started, parallel scheduling ceases since the task adaptation is then run on each processor locally. Therefore, the total scheduling time is just T_{crit} * 96 or 2.3 msec. This represents an overhead ranging from 0.00006% for the mountain image to 0.00023% for the stegosaurus image.

5.3.2. Synchronization (0.16% - 2.3%)

It is necessary to determine the amount of time wasted by spinning in a lock, in addition to the extra work needed to determine which processor to split. These two factors constitute the synchronization overhead which was given in equation 4.9. The value for this overhead varies from 0.16% for the tree image to 2.3% for the Laser image. While the time wasted in synchronizing may not be particularly small in some cases, it is necessary in order to facilitate the dynamic partitioning scheme of the task adaptive algorithm.

5.3.3. Communication Overhead (0.11% - 4.2%)

The communication overhead in this algorithm is measured the same as the previous algorithms. The overhead varies from 0.11% for the stegosaurus image to 4.2% for the Laser image. The number of bytes communicated in this algorithm is much higher here than in the other approaches, which accounts for the higher overhead percentage. The reason for this is given next.

At the time a task is split, the splitting processor (P_s) retrieves *all* of the data relevant to the splittee (P_{max}). The data which is unnecessary for the portion of the task which P_s is to work on is then discarded. At the end of the computation, a large amount of splitting occurs due to dynamic load balancing.

Task Adaptive Algorithm (LC Scheme) Performance

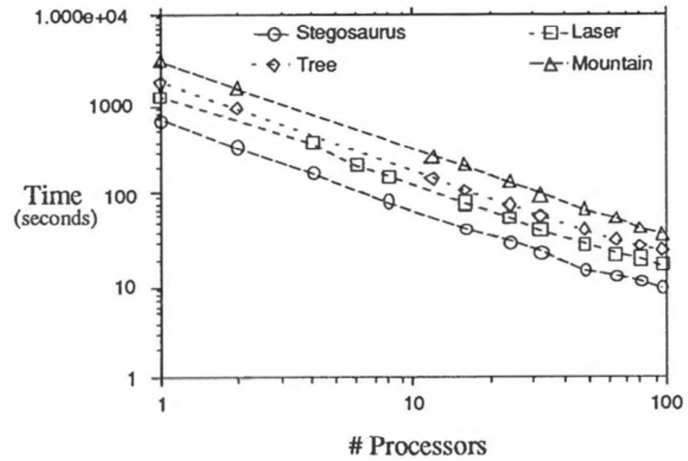

Figure 5.24: Tiling time for task adaptive scheme

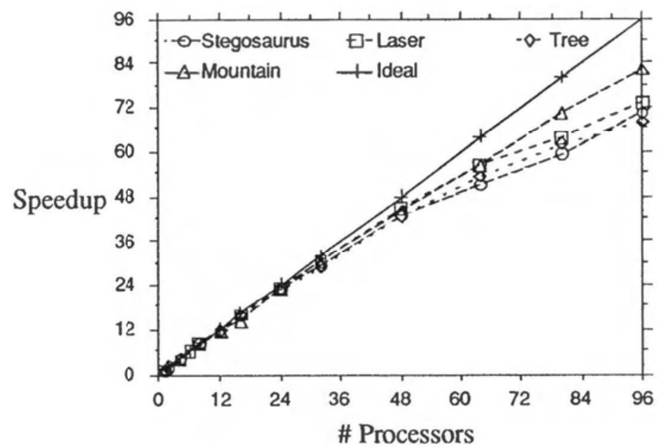

Figure 5.25: Speedup of task adaptive scheme

The areas to be split at this point in the computation are small, but the amount of data to be transferred is large since it is derived from the initial decomposition area. This creates communication of unnecessary data, which is then discarded. Reducing this communication requires extra synchronization, but preliminary studies indicated that performance degraded even worse than if it was not done. In the case of the Laser image, most of the initial areas assigned as work involve background color. These processors finish quickly and then start splitting other processors' work. Since the few processors which are split contain the bulk of the data, a lot of communication occurs. A solution which relieves the extra data transfer in this situation would reduce the communication and contention overheads if it were possible to implement it without significantly increasing the synchronization costs.

5.3.4. Network Contention (5.5% - 11.7%)

The overhead percentage for network contention ranges from 5.5% for the mountain image to 11.7% for the stegosaurus image. Even with the extra communication, the contention measured in this algorithm is only slightly higher than in the rectangular region (LC) scheme.

5.3.5. Load Imbalance (9.2% - 22.5%)

This algorithm tries to minimize load imbalance by using heuristics to dynamically split tasks during parallel execution. The limit of the task size which can be split is set to two scan lines. The load imbalance overhead percentages vary from 9.2% for the mountain image to 22.5% for the tree image. If the only tasks that are left are single scan line tasks, processors which are idle will not be able to find a task to work on. Since the granularity of tasks which cannot be split (a single scan line within an area) is fairly large, the idle time for a processor with no work left can be high, resulting in additional load imbalance. Of course single scan line tasks could be split into two parts as well, but this feature has not been implemented at this point. Further research is needed to see if these tasks can be split, or if some other solution is possible to reduce the excess idle time.

5.3.6. Code Modification (0.4% - 1.5%)

The code modification overhead is measured the same as in the other algorithms. The overhead percentages are fairly small and range in value from 0.4% for the stegosaurus image to 1.5% for the Laser

image. These figures are expected since the number of tasks is the smallest of all the algorithms. Consequently, most of the tasks consist of large areas where coherence is maintained. In addition, even when a task is split, the split processor is not interrupted and coherence is not lost for its task.

5.3.7. Explanation of Results

The task adaptive method is an attempt to directly load balance the system by dynamically extracting work when a given processor would otherwise be idle. The solution allows a granularity ratio of $R = 1$ for the initial decomposition. A graph showing the primary overhead contributors is given in figure 5.26.

Unfortunately, the load balancing of this scheme was not as good as was anticipated. Since load balancing is due to the total idle time at the end of the computation, this suggests that processors have quit looking for work too early. The threshold for splitting work imposed here is that a single scan line task cannot be split. Perhaps a scheme could be worked out to allow horizontal splitting, but this would be

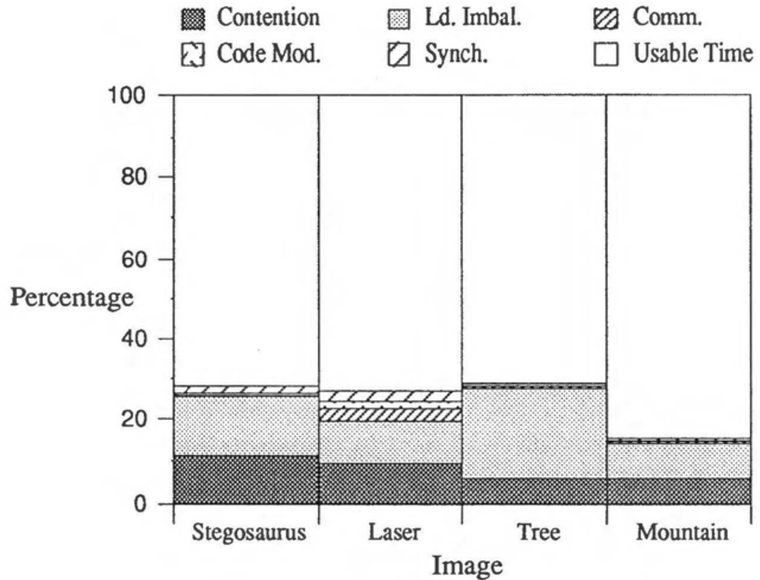

Figure 5.26: Degradation factors for task adaptive algorithm ($P = 96$)

difficult to implement and the synchronization involved may outweigh the benefit of splitting.

Synchronization is an additional overhead in this algorithm, but it was not a significant factor in performance degradation. The communication cost in this algorithm is somewhat larger than the other LC schemes, due to the dynamic partitioning of this dataset. The code modification here is the smallest of all the algorithms since the number of areas generated is initially equal to *P*. In addition, coherence is maintained in the upper portion of a split area reducing the parallel execution overhead. Network contention seems to be only slightly worse than in the rectangular region (LC) scheme. Toward the end of the computation when dynamic load balancing is taking place, there is a flurry of communication, and this causes network contention to increase at this point. The burst of communication is due to the dynamic splitting of small tasks at the end of the computation. Reducing this last amount of communication is rather difficult in the LC scheme: the reason is described next.

In the task adaptive algorithm, the splitting processor copies all of the data necessary for the entire original size area and then deletes the excess data locally. Ideally, it would be desirable to only copy the data which is needed for the scan lines for which this processor is responsible. The time required to do this would be prohibitive since there are only two ways: 1) the splitting processor remotely determines which polygons are relevant or 2) the processor being split must be synchronized to stop what it is doing and then determine which polygons are relevant for the splitting processor. The first method would require more communication than in the current implementation. The second method requires extra synchronization, plus P_{max} would have to construct a new data structure, and this takes time away from its primary work. Thus, when using the LC scheme, it only makes sense to copy all of the data for the area. In the next chapter, the performance of the task adaptive scheme is analyzed using both the UD and LC memory referencing schemes for the entire program, to see if any difference is noted.

In an attempt to explore other load balancing strategies, different heuristics were tried in order to estimate the maximally loaded processor. For instance, instead of just using the number of scan lines left as the heuristic, the total number of polygons per scan line for all the scan lines left was used. The idea was to evaluate the work in terms of polygons since the lower half of a region to be split could possibly contain no polygons. This method required the splitting processor to retrieve from shared memory an additional value which corresponded to the heuristic. It was also necessary to update this

heuristic from scan line to scan line, whereas the previous heuristic required just a simple subtraction operation. As a result, the benefit of this new heuristic was outweighed by its cost, and it proved to have worse performance than the simple one.

Finally, as was mentioned previously, it may be worthwhile to try breaking scan lines into half scan lines to allow a splitting processor to split single scan lines. This would require extra synchronization, but it is possible that the load imbalance would be reduced if the overhead to do this is small. This was not implemented in the test program, and could be done as part of future research.

This algorithm does exhibit good scalability since the algorithm adapts to the scene and divides the tasks accordingly. Its principal advantage is that the number of tasks does not need to be chosen initially, making the granularity ratio analysis unnecessary. In addition, in the next chapter it is shown that the overhead in the front end for this scheme is less than in the other algorithms due to the reduction in the total tasks required in the initial decomposition.

5.4. Conclusions

In this chapter, the maximum potential performance for each of the implemented algorithms is evaluated. This is done by analyzing the tiling portion of the programs. A summary of the results obtained with regard to the influence of the various overhead factors is presented next.

The scheduling overhead is minimal for all of the algorithms discussed here. Since the execution time for the simplest task (background color) is greater than the critical time needed for scheduling, this overhead is not a factor in performance degradation in any of the algorithms.

Synchronization is an important consideration in the task adaptive algorithm due to the dynamic task partitioning. The overhead of synchronization does not degrade the performance significantly, as it turns out, so it is not considered to be a major degradation factor.

The issues of latency, communication, and network contention are all intertwined since they are related to passing data through the interconnection network. Memory latency is relevant to the scan line algorithm and the rectangular region algorithm since those algorithms are implemented using the UD memory referencing scheme. The latency is somewhat smaller for the latter method, due to the reduction in the number of remote memory requests as a result of better exploitation of coherence. Communication comes into play

for the LC schemes and results in more efficient use of the interconnection network, with the benefit being a reduction in contention. Graphs which show the total amount of performance degradation for each image are included here so that all of the algorithms may be compared side by side. These are shown in figures 5.27 through 5.30 at the end of the chapter. The graphs are shown in such a way that the total of each column is the total processor-time space. This is the same as the parallel execution time T_p multiplied by the number of processors P (in this case, $P = 96$). Therefore, the column with the least height is the best performing algorithm for that particular image. Based on the data shown in these graphs, one can see that the rectangular region (LC) algorithm results in the lowest overheads, and consequently the best performance in the tiling section.

Hot spot contention is not a factor in any of these algorithms. This is because the large data structures are distributed across the memory modules. Copying of small data structures to local memory is also employed if these structures are referenced frequently. Although there may be frequent references to common data structures, this method of scattered storage ensures that performance is not degraded since no hot spots exist in any of the programs.

Load balancing is a primary goal of any parallel implementation. The only algorithm in which the load imbalance is significantly reduced is the data adaptive algorithm. The task adaptive algorithm exhibits the worst load balancing of all the algorithms. The probable reason for this is the lack of splitting at the scan line level (that is, below the threshold). Surprisingly, the scan line algorithm does not exhibit much worse load balancing than the others. This changes as the number of processors is increased since the number of tasks available for each processor is reduced.

The primary overhead due to code modification is the loss of coherence. The parallel scan line algorithm exhibits total loss of vertical scan line coherence. The number of regions created in the two rectangular region schemes introduces some loss of coherence in both the horizontal and vertical directions. Since the top-down and task adaptive algorithms require fewer regions than any of the other approaches, the code modification overhead for these methods is small.

Scalability is one of the most important characteristics of a parallel algorithm. In evaluating these implementations, it seems evident that the parallel scan line algorithm does not exhibit particularly good scalability. In table 5.2, each of the implemented algorithms is compared for each image, using 96 processors. The

times listed are an average of 3 runs, although the difference between each run was less than 1%.

From the table we can see that the data non-adaptive rectangular region (LC) scheme provides the best results in most cases. The comparison is only for the tiling section of the program and does not include the overheads inherent in each of the LC algorithms. Also, the overhead of building the tree data structure necessary for the the top-down data adaptive algorithm is not included. It is important to not make any judgments as to the usefulness of any of these algorithms at this point since there are numerous other factors that must be examined to determine how well they will perform in the general case. The analysis here is purely with respect to the performance of the tiling section of the algorithms since this section of the program is where the most parallelism can be exploited.

Table 5.2: Comparative times in seconds of tiling for all algorithms on 96 processors

	UD Scheme		LC Scheme		
Images	Scan line	Rect. Region (UD)	Rect. Region (LC)	Top-Down Adaptive	Task Adaptive
Stegosaurus	12.66	13.38	10.16	12.93	9.94
Laser	21.29	19.94	17.06	18.72	18.33
Tree	24.88	25.70	22.72	23.66	26.41
Mountain	59.85	44.33	38.35	40.31	39.98

The setup operations prior to the tiling section vary depending on the algorithm used for task decomposition. If these costs are high for a particular method, the overall performance is affected. These costs are included in the analysis in the next chapter to give a better overall view of the performance of the implementations. The different shared memory referencing strategies are investigated and analyzed in the next chapter as well.

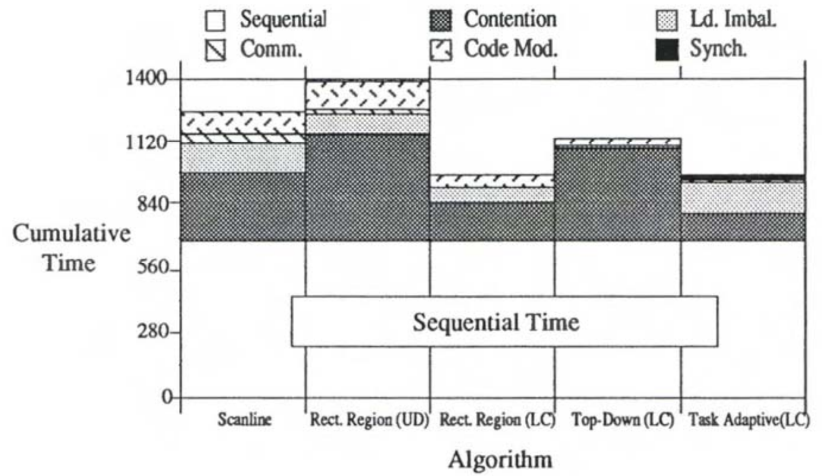

Figure 5.27: Comparison of overheads for algorithms, stegosaurus image

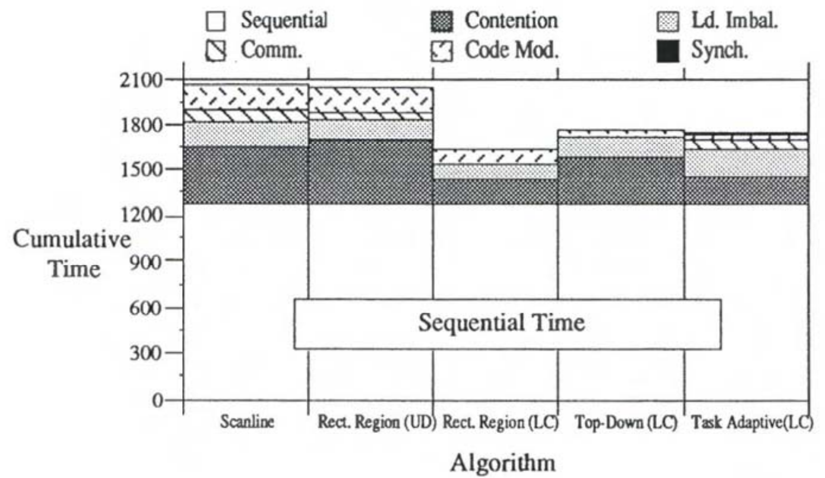

Figure 5.28: Comparison of overheads for algorithms, Laser image

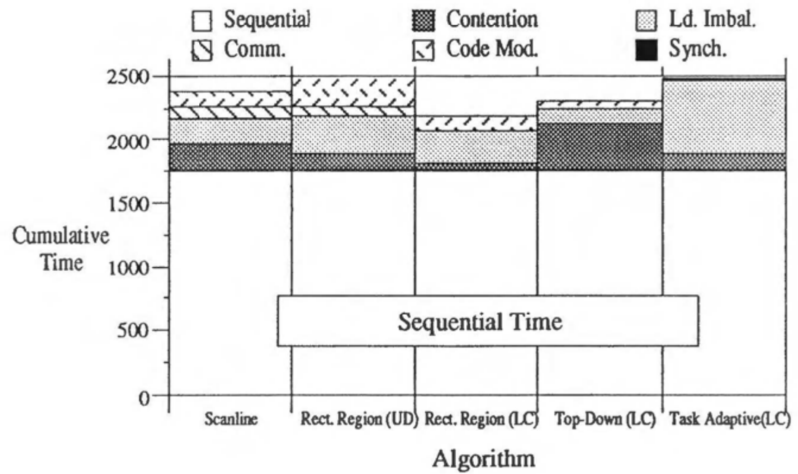

Figure 5.29: Comparison of overheads for algorithms, tree image

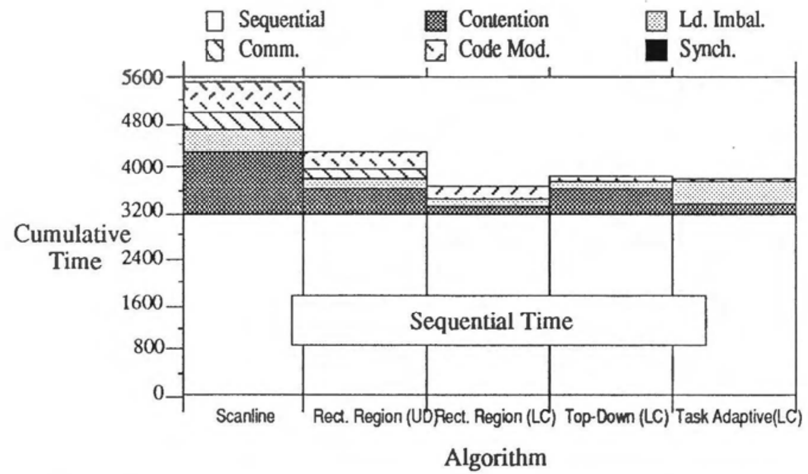

Figure 5.30: Comparison of overheads for algorithms, mountain image

6

Characterization of Other Parameters on Performance

In this chapter, a number of parameters are investigated which differ from those discussed thus far. The purpose here is to produce a comprehensive study of the shared memory referencing strategies and further evaluate the performance of the various algorithms under a variety of conditions. Several different shared memory storage and referencing methods are analyzed in the first section. The implementations of the Uniformly Distributed and Locally Cached schemes are described in detail in this section. A framework is presented which allows a straightforward comparison of these schemes using the task partitioning implementations discussed in the previous chapter. In the second section, the effect of machine parameters such as the operating system and architectural characteristics are evaluated in regard to algorithm performance. In the third section, a number of additional characteristics such as image and object complexity are varied to see how overall algorithmic performance is affected. The comparisons in this chapter are

intended to provide a broader base for determining the relative merits of each of the parallel approaches which have been implemented.

6.1. Shared Memory Storage and Referencing

The idea of partitioning image space segments for use in a parallel graphics rendering algorithm can be extended to memory referencing as well. The scene data used in the graphics rendering algorithms is read in from disk and then processed in the front end. The polygons are then transformed from three-dimensional space to image space and become read-only data thereafter. As such, the read-only data can be partitioned in numerous ways for referencing during the tiling portion of the program. Three alternative data storage and access schemes for use in a parallel graphics display algorithm are presented in the subsections which follow. A brief description of these schemes is given next.

If enough memory is available, all of the data could be copied to each processor's local memory; then no remote memory access is necessary after the copying phase is completed. This storage and access scheme is analyzed in the first subsection below. The second scheme involves scattering the data among the memory modules in the system and referencing it remotely. In the third technique, the data is scattered initially as in the second scheme, but then a reorganization is required to allow the data to be copied to the local processor's memory as it is needed. This last method allows local referencing after the copy is completed and is described in the third subsection. The second and third methods are the same as the UD and LC memory referencing strategies discussed previously. Here, their theoretical performance is analyzed, and a full description of the implementation details is presented.

A dataset which would contain 100,000 points and 100,000 polygons is used for theoretical analysis. The front end process removes a number of backfacing polygons, conservatively eliminating 1/3 of the original data (this assumes a given polygon is not both front and backfacing). Below, the amount of memory required for this dataset is given after transformations and backface rejection have been applied. The assumption in this case is that a mesh of size 48 x 48 has been placed over the image. This corresponds to the number of regions generated with a granularity ratio of $R = 24$ on 96 processors using the rectangular region task partitioning scheme.

When applying a mesh over the image, a polygon (or polygon pointer) needs to be duplicated for each region that a given polygon crosses over. This duplication is based on both the size of the

polygons and the granularity of the mesh. The mountain image contains approximately 83,000 polygons after backfaces are removed. As an example of the duplication, figure A.12 in the appendix shows that for this image, 130,000 polygons are created after duplication using a 48 x 48 mesh which is an increase of 57%. Using the 100,000 polygon test situation and this same percentage increase as an example, we can expect to lose 33,333 polygons to backface rejection and then gain 57% more polygons from duplication, resulting in a final total of 105,000 polygons. We assume that this results in 105,000 points as well, although this latter value is typically smaller.

The analyses given next take into account the additional time required to access data beyond a normal local memory access. This includes any setup time specific to each scheme in addition to any latency incurred.

6.1.1. Copy Data to all Processors

This method involves copying all the data to all of the processors in the system. No remote referencing is required after the data is copied, so no communication overhead is incurred during the tiling portion of the program. To ascertain the cost of copying the data to all processors, let us estimate the time to copy 105,000 points and 105,000 polygons to 96 processors. This copying can be accomplished in parallel by creating a binary tree of processes in which the data is copied throughout the network from processors that contain data to neighboring ones that do not. This copying process is repeated until all processors contain data. The number of times this is repeated is the height of the tree, namely ceil($\log_2(96)$) or 7. Each data point contains 3 floating point values consuming 12 bytes, and it is assumed that each polygon is a quadrilateral. Using the storage data format described previously in section 4.1.1, a single polygon takes up 10 bytes.

The memory required for all the data is then 105,000 * (12 + 10) or 2.31 million bytes. Equation 6.1 shows the communication cost with block transfers of 256 bytes, each using the binary tree copying technique.

$$T_{comm} = \#levels \; * \; \#transfers \; * \; (T_{setup} + 256 * T_{bt}) \qquad (6.1)$$

T_{setup} is 8 μsec and T_{bt} is 0.25 μsec/byte for block transfers on the Butterfly GP1000. The number of levels is 7, and the number of block transfers is (2.31 million)/256 or 9,023. Plugging these numbers into the equation results in an overhead time of **4.548** seconds. This time

does not include the time to copy normals or the polygon information data structure which contains the bounding box of the polygon, a pointer to its location in the polygon list, and other information. If these are required, the time would be more than double, although it is possible to create both data structures locally on each processor instead. The memory required for all of these data structures, in addition to the data structures needed for scan conversion, exceeds the 4 megabyte limit per processor available in the BBN GP1000.

The preceding analysis assumes that no network contention occurs during the copying process. This will not be the case after a few levels of the copying tree have been completed since there are not that many unique switch paths in the Butterfly and some may become blocked. This might be avoided by copying less data simultaneously, but that adds levels to the tree. There is also the issue of copying the normals and other necessary data structures or regenerating these locally. Regenerating the normals adds time to the computation, but not to the copying process. Alternatively, the potential for increased network contention exists if the normals are copied. A more detailed analysis is needed to adequately evaluate this issue, but it is not necessary for the purposes here since conclusions can be drawn without such an analysis.

This copying scheme uses a huge amount of memory so that subsequent references to all data can be local. The amount of data that any processor really needs to perform its tasks is significantly less than the entire input dataset, since each task will likely refer to only a small subset during the tiling operation. Therefore, this scheme makes inefficient use of the network and storage resources. The potential for network contention increases as larger processor configurations are used. The reason is that the number of processors increases linearly, while the number of switch paths increases logarithmically. In addition, more memory is required than is available per processor, so this scheme is not generally usable except for smaller datasets. Even for machines which might have enough memory per processor, it is still evident that this method is inadequate for general use. The next scheme makes better use of the memory in the system.

6.1.2. Global Referencing

The basic idea in global referencing of shared data is to distribute the data and references throughout the system. This avoids hot spot contention since the data is not in a single location, although latency and network contention are introduced during the tiling section. This

technique allows the aggregate memory available in the system to be used so that it can be considered as one globally shared memory. The data is stored so there is only one copy in the system, which conserves system memory in addition to the time savings resulting from not copying unnecessary data.

This method is essentially the same as the shared memory storage in bus-based architectures such as the Encore Multimax or Sequent Balance. These computers, known as Uniform Memory Access (UMA) architectures, use such a scheme in all programs since a global view is provided of memory in these architectures. They incorporate a number of different processor boards connected to a bus, on the other side of which is a number of memory boards, as was illustrated in chapter 3, figure 3.2. The term UMA refers to the fact that every processor is the same distance from global memory, resulting in an equally distributed communications overhead. This technique can be emulated in software on the Butterfly, where it will be referred to as the Uniformly Distributed (UD) approach to shared memory referencing. A brief description of this scheme was given in section 4.1.1.2, which presented the design of the front end to all the algorithms. The data is scattered throughout the memory modules as it is read-in and then referenced remotely in the tiling portion of the program. After this scattering of data, each processor contains approximately N/P polygons; that is, the dataset is evenly divided among the memory modules. Since the data is scattered throughout the system uniformly, an average of $1/P$ of the references to shared memory will actually be to data stored locally. Although this percentage is an average, it is likely that the deviation from this average must be large. The worst case situation, where all of the data referenced by a given processor is stored remotely, is actually a more realistic scenario. The reason for this expectation is due to the screen space locality of data. Most of the references for a given task will likely be to a particular processor or group of processors rather than scattered throughout the entire system. An estimate of the remote referencing time overhead in the tiling section using this shared memory referencing strategy is presented here with the assumption that all references to global memory are remote references.

The integration of scattering the data with reading in objects in the front end allows the front end work to be accomplished on each processor without any remote referencing. The time for the front end work does not need to be accounted for in the following analysis since there is no difference among the memory strategies in the way this is performed. The remote referencing time overhead is given in equation 6.2.

$$T_{latency} = \#refs * (T_{rref} - T_{lref}) \qquad (6.2)$$

T_{rref} is the remote referencing time which is 7 μsec. T_{lref} is the local referencing time, which is 0.53 μsec. The latency factor is the time difference between these two values. The number of remote references in the tiling section is based on a number of factors. Due to the construction of local edge lists, each point must be referenced 3 times and each polygon once. Since each point contains 3 floats, the number of point references is 3 * 3 * 105,000, or 945,000 point references. The number of polygon references is 105,000 polygons * 5 shorts per polygon, or 525,000. In addition, about 5 references are needed per polygon to obtain the polygon pointer from remote memory, as well as other polygon information adding up to 525,000 more references. There is also one reference for each normal, which results in 3 floats per normal * 105,000, or 315,000 references for normals. The total number of references per processor on 96 processors in parallel is then 1/96 * (945,000 + 525,000 + 525,000 + 315,000) or approximately 24,063 references per processor. The communication time is then: 24,063 * 6.47 μsec or **0.1557** second.

This analysis is *very* simplified since network contention is not taken into consideration. The edge list data is stored locally after it is remotely referenced, so it does not need to be referenced remotely again. A number of remote references to the points list are required in the anti-aliasing portion of the program which are not accounted for in the values derived above. That section of the code could be optimized to allow only one remote reference per point by using temporary storage, but we have not implemented such an optimization. As shown here, the small overhead for this scheme makes it attractive for implementation. Next, the details of implementation are described in regard to this scheme.

Implementation of the UD Scheme

During the front end, as the polygons are read in, it is necessary to determine in which area(s) of the 2D screen mesh a given polygon may belong. A short pseudo-code segment shows how this is done:

```
On each processor:
For all polygons on this processor              O(N/P)
    For all areas in mesh this polygon
    crosses over                                  O(c)
       Lock mesh(i,j)
       Load polygon pointer into end of
       area[i][j] linked list
       Unlock mesh(i,j)
```

The time complexity is based on the number of polygons on a given processor after backface rejection (N/P) multiplied by a constant (c). This constant is the number of areas a polygon can cross over, and is related to the size of the polygons and the size of the mesh. The duplication graphs in the appendix in figures A.9 through A.12 indicate the total number of polygons after duplication, based on mesh size. The duplication factor is the number of polygons after duplication divided by the original number of polygons. This factor, which would be the average number of iterations for the inner loop above, goes from approximately 4 for the stegosaurus image to 1.5 for the mountain image, with a mesh size of 48 x 48 (2048 areas). The locks are needed so that only one processor at a time adds a link to the shared link list (area[i][j]). A separate lock exists for each area in the mesh. Figure 6.1 illustrates the storage of polygon pointers in the area mesh.

During the tiling operation, a separate area is assigned to each processor as a single task. The processor then traverses the polygon linked list and constructs local edge lists for use in the tiling operation. The pointers in these links are scattered throughout global memory so a global reference is required for each link, but this is only needed during the initial traversal of the list. These global references are included in the preceding analysis. This implementation of the

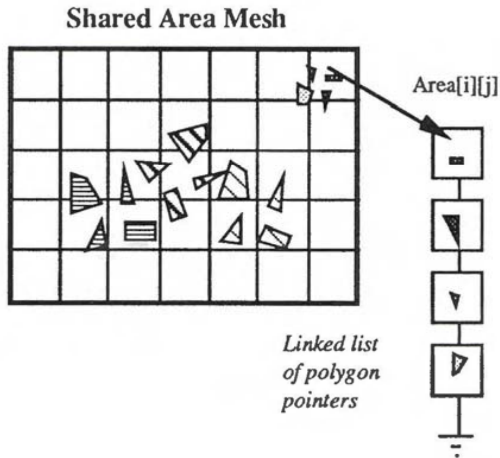

Figure 6.1: Area mesh storage of polygons pointers

Uniformly Distributed (UD) scheme was used in the scan line and rectangular region algorithms described in chapter 5, sections 5.1.1 and 5.1.2.

6.1.3. Software Caching

The last type of referencing scheme is designed to optimize memory access on a Non-Uniform Memory Access or NUMA architecture. The term NUMA refers to the fact that some references to shared data require less time than others since a processor can access shared data stored in its local memory module without retrieving it remotely across the interconnection network. The UMA architectures described previously use a local cache which contains the most recently referenced data, thus allowing (potentially) faster access to the shared data. The UMA architectures use sophisticated cache coherence schemes so that the copy of the data in the cache is the same as what is stored in global memory. NUMA architectures such as the BBN Butterfly typically do not exploit cache coherence (even if they have a cache); the programmer is responsible for maintaining cache coherence. Since cache coherence is not normally available in an NUMA machine, it is not recommended to copy writable shared data to local processor *private* memory. Read-only shared data can be copied to private memory, and the data is then accessible locally, as was the case in the first scheme described previously. In the scheme described here, however, only the data needed for a particular screen area is copied rather than the entire dataset.

Implementation Details for LC Scheme

The method for local referencing we have implemented for NUMA machines is called the Locally Cached (LC) memory referencing scheme. The basic idea is to copy the appropriate data into the local memory of the processor which will use it for tiling a given region. This scheme allows local referencing of data without any latency or possible network contention, except during the copying operation. The data is read in during the front end, as was done in the previous UD scheme. After the front end, each processor contains on its local memory module an average of N/P polygons as before. For this analysis, it is assumed that the 48 x 48 rectangular region partitioning is used as before. The data is arranged into contiguous blocks (arrays) prior to copying in the tiling section. An explanation of why this is done is given after the pseudo-code is presented below. The implementation proceeds as follows:

In Parallel: Complexity

[1st pass]

```
For all polygons on this processor          O(N/P)
    For all areas in mesh this polygon
    crosses over                            O(c)
        Accumulate memory needed for
        each of the following 4 arrays:
        (points,normals,polygon connectivity,polygon info)
```

[2nd pass]

```
For all polygons on this processor          O(N/P)
    For all areas in mesh this polygon
    crosses over                            O(c)
        Allocate memory for each of 4 arrays
        for area[i][j] if not done yet
        Add polygon and point data to
        the 4 arrays listed above
Free up original scattered data.
```

This code is executed prior to the tiling section of the program and was not included in the measurements in chapter 5. The first pass is necessary to determine how much memory to allocate for a particular region, and the second pass actually allocates the memory on the local processor and copies the data into it. A barrier synchronization is necessary between the passes so that the data is updated properly for all regions. All of the work in these phases is done using local memory, so no remote referencing occurs here. The inner loops in the first and second passes are of the same time complexity as the inner loop described in the previous section. Figure 6.2 illustrates the storage of the arrays in each local processor's memory.

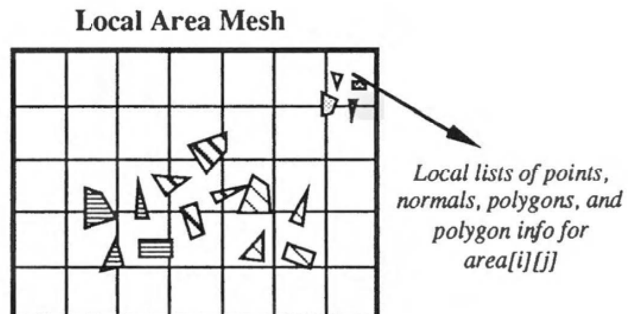

Figure 6.2: Locally cached memory storage mechanism

The LC method is more than just a "block copy then local reference" scheme. It consists of a complicated set of instructions which involve constructing data structures for later block transfer. The principal advantage of this scheme in a non-blocking network such as in the Butterfly is as follows. The setup cost is incurred only once for a block of data, and thereafter the message proceeds at the full bandwidth of the interconnection network. This is faster than individually copying each remote value to local memory since the setup time for that method would be incurred for each single reference. The disadvantage to this block copy method is that the data must be arranged into a contiguous array. If a blocking interconnection network were to be used, the data could then be transferred byte by byte instead. The LC scheme consists of a method of organizing data primarily for later local referencing while using minimal memory usage. The data structures and setup routines necessary to achieve this set it apart from a pure software caching scheme.

The pseudo-code presented for this scheme sets up the blocks for copying, but the copying phase is actually executed during the tiling portion of the program. If a completely uniform distribution of the data occurs, then each processor would contain exactly $1/P$ of the data for a particular area. In general, this is not the case, as was stated before based on the locality of screen space data. For this analysis, it is assumed that the data is distributed in such a way to encounter a worst case scenario (i.e., all the data needed for each region is stored remotely). For a particular task, it is necessary to use P separate block transfer groups to retrieve the data. This is shown in figure 6.3 on the next page.

To simplify matters, each processor is assumed to execute exactly R tasks so that the total number of block transfer groups is $(4 * R * P)$. Four refers to the fact that it is necessary to retrieve the points, normals, polygon connectivity, and polygon info arrays separately. Each block transfer retrieves on average $1/(R * P)$ of the total amount of data.

Based on the analysis at the beginning of this section, the total amount of data after backface rejection and duplication is 105,000 polygons, so the amount of data per area of the 48 x 48 mesh is approximately 46 polygons. Recall that for a block transfer, T_{setup} is 8 μsec and T_{bt} is 0.25 μsec/byte. If the data is evenly scattered as was stated above, each polygon (in the worst case) is on a separate processor, requiring 46 separate groups of 4 block transfers each.

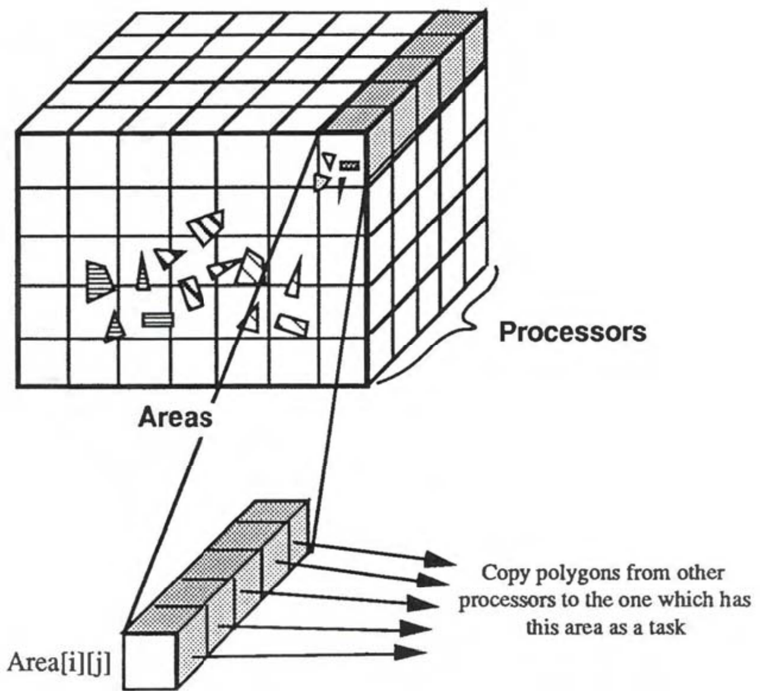

Figure 6.3: Block transfer of data

The data is scattered among the 46 processors which contain polygons for a given area, so the total time to retrieve data for one region is 46 * 38.5 μsec or 1.77 msec. Since each processor works on an average of 24 regions, the total time for a processor to retrieve all the data it needs to work on its regions during the tiling section is 24 * 1.77 msec or **0.0425** second. Again, it is assumed that no network contention exists for this analysis. This time is executed in parallel so 0.0425 second is the parallel communication time. The time for block transfer for a single polygon is then:

Block	Time
1. 10 bytes/polygon	7.5 μsec
2. 20 bytes for polygon info	15.0 μsec
3. 12 bytes/point • 4 points/polygon	8.0 μsec
4. 12 bytes/normal • 4 normals/polygon	8.0 μsec
Total	38.5 μsec

This time is significantly better than any of the times listed above for the previous methods of memory storage. The second pass is necessary to set up the arrays for block transferring, but this has not been taken into account in the preceding analysis. Since this time is extra, it needs to be accounted for as well. In the next paragraph, the second pass algorithm is described, and its time complexity is analyzed.

In the second pass, new arrays are constructed which correspond to the data that is relevant to each area of the 2D mesh in the local processor. In constructing these new arrays, it is desirable to not create any unnecessary new data points. In order to do this, a backwards reference list is used to determine which points have been stored in this area thus far. In order to keep the amount of memory within limits for this backwards reference list, a fairly sophisticated data structure is used. This data structure is an array which corresponds to the points list, but contains links which indicate when any point that has been previously stored in this area is part of a new polygon. The backwards reference list data structure is shown in figure 6.4.

The diagram shows that the backwards reference list corresponds to each point in the original object. The small array to the right is used to indicate the areas each point is referenced in (the polygons which contain it can be in more than one area) and the value of the point's index for the new points list in each of these areas. The new points list allows us to sequentially go through the polygon list in the front end. This data structure uses less memory than a separate

backwards list array for each area since that type of list would be relatively sparse. The list requires some time to manage, and this time is considered as part of the analysis.

The backwards reference list is also required for the first pass, but in that case the reason is to determine how much memory to allocate for the contiguous arrays. The top loop given in the pseudocode is of time complexity $O(N/P)$ which in the case given here corresponds to approximately 1094 polygons. The inner loop would be approximately of time complexity (constant = 2) for a theoretical 100,000 polygon dataset based on the analysis of the mountain image, but we will use the value (constant = 4) for a possible worst case scenario. The management of the backwards reference list requires us to run through each point in the polygon, so there really is a third inner loop that would be of time complexity (constant = 4), assuming quadrilateral polygons. The only difference between the first pass and the second pass is the time required to allocate memory for the areas not already allocated and to store the data in the new arrays while updating the count for these lists. The GP1000 contains 2.5 MIPS MC68020 processors, and based on the amount of work in the inner loop of the second pass, we estimate the time to complete this operation to be 20 μsec per iteration. This results in a time for the second pass of 1094 * 4 * 4 * 20 μsec or **0.35** second. This analysis is simplified, but the purpose is to show the additional overhead incurred by the LC scheme. The first pass time is not measured since

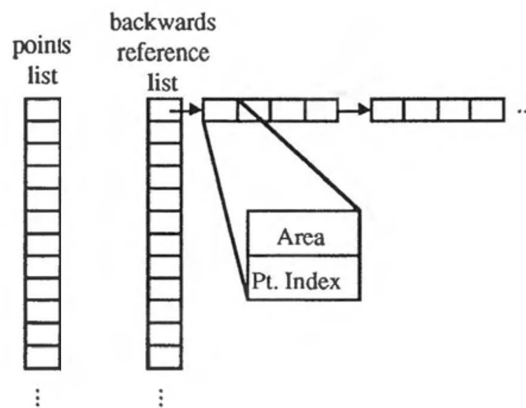

Figure 6.4: Backwards reference list data structure

it takes approximately the same amount of time as its counterpart loop in the UD scheme. The total time for this scheme is then 0.35 seconds for the second pass plus the time of 0.0425 second for communication, resulting in a sum total of **0.3925** second.

There have been several other graphics algorithms which incorporate the idea of local caching in a distributed memory environment. Green and Paddon [Gree89] as well as Badouel, *et al.* [Bado90] have both implemented a software caching mechanism in a distributed memory environment. Both algorithms use ray tracing for hidden surface elimination and rely on the concept of ray coherence for minimizing remote references. Ray coherence is defined to be the property in which rays in adjacent pixels are likely to intersect the same objects. Once these objects are brought into the local memory of a processor by the cache mechanism, the rays sent out by this processor will intersect these same objects in local memory. Based on this fact, Badouel was able to achieve a 95% or better hit ratio into the caches. An area screen space distribution of the pixels to processors is used for task decomposition, similar to the approaches given here.

These algorithms were designed to allow one to distribute a large graphical database on a message passing multiprocessor such as the Intel iPSC, which provides no support for shared memory referencing. The caching employed in Green's algorithm involves statically partitioning local memory for caching purposes, while Badouel's method uses a more dynamic approach without any preprocessing. Badouel's algorithm allows virtual memory to be distributed by taking advantage of the aggregate memory in the system, whereas Green's approach requires the host to maintain virtual memory. In Badouel's algorithm, the object database is statically divided up into pages and scattered throughout the system in a way similar to the scattering of data used in the LC scheme described previously. If a page is not resident in the local processor's memory or cache, the page is retrieved from the processor memory module where it is resident and put into the local processor's cache using a least recently used (LRU) cache replacement policy. Badouel has shown significant speedup on the Intel iPSC with this caching scheme built into a multiprocessor ray tracing algorithm. Several faults exist with this scheme if it is to be applied to a conventional scan line algorithm such as those outlined in the previous chapter.

The first issue is the amount of memory available in each processor. A ray tracing algorithm might use a hierarchical tree structure such as an octree to speed up calculating ray-object intersections, and this tree must be stored in all processor memories.

No other additional memory is required during the execution of the program. In a scan line algorithm, edge lists, anti-aliasing data structures, and interpolation parameter arrays must be built which all take up a significant amount of local memory. More local memory is necessary in a scan line algorithm than is needed for ray tracing, so less would be available for the cache. The reduction in cache size would result in a lower hit ratio, giving lower performance. In a ray tracing algorithm, it is impossible to know a priori which polygons might be needed in local memory since a ray can be spawned to any direction in three-dimensional space. It therefore makes sense to bring in the data as needed using a LRU replacement policy. In the algorithms presented in the previous chapter, the exact polygons that are needed for rendering are known ahead of time, so only those should be brought into the local memory module. Furthermore, since those polygons are only used for a single task, the original (remotely stored) polygons can be deleted. This provides additional free space, allowing more room for local data structures.

The second issue is the amount of communication and potential contention problems in the caching mechanism. The amount of memory brought in using the LC scheme is exactly what is needed, so no unnecessary message traffic is required. Badouel's caching scheme copies pages one at a time, and it possible that only one item of an entire page is required. The results of speedup in his ray tracing algorithm are based on images which take minutes to render on 64 processors and would typically take hours to render on a single processor. This is due to the fact that ray tracing is a slower, less efficient rendering algorithm than the image space methods described in this document. The ratio of computation time to message traffic time is so high in ray tracing that any possible bottlenecks in message passing are masked due to the high computation time. The higher efficiency of the scan line algorithm reduces computation time, so these bottlenecks are more likely to degrade overall performance than in a less efficient algorithm. This is shown by the reduction in network contention determined for the larger datasets using the LC algorithms in the previous chapter. Badouel's approach requires more communication than the LC scheme given here since pages are brought into memory as needed. Therefore, his approach is likely to result in greater contention when compared to the LC scheme. While a multiprocessor can sufficiently speed up a costly algorithm such as ray tracing, the benefits of using that type of method are generally not needed in most applications. The real need by most scientists and other users is to be able to display extremely complex datasets in a reasonable amount of time. Therefore, if reflections are needed, one

should use a ray tracer. If high quality scenes need to be generated quickly without reflections, an image space algorithm such as those illustrated here is more appropriate.

In the next section, the results of the UD and LC schemes are compared, including the overheads required in the front end and the second pass, to see how these affect the overall performance of each of the algorithms.

6.1.4. Results

The total time for remote referencing of the LC scheme is **0.3925** versus the time of **0.1557** second in the UD scheme based on the theoretical analysis used here. On the surface it would seem that the UD scheme is the better alternative even with its remote reference strategy. However, one important factor missing from this analysis is network contention. From the data given in chapter 5, contention contributed significantly more to degradation of performance in the UD scheme than in the LC scheme for the rectangular region partitioning scheme. The primary reason is that the LC scheme uses the network in bursts of communication which take a very short amount of time, minimizing the chance of a blocked path. The UD scheme relies on a large number of small messages which can eventually saturate the network.

To illustrate the differences between the two memory referencing strategies, we compare them using the data for the tiling section from chapter 5. The data from running the task adaptive algorithm using the UD scheme has also been included. The UD task adaptive algorithm is not nearly as efficient as the rectangular region UD implementation since each time an area is started, the entire polygon list from the split area must be traversed. These polygons are traversed from shared memory, while in the LC implementation of the task adaptive scheme, local memory is used. Latency causes the algorithm's efficiency to go down as the number of processors is increased.

The graphs for these algorithms for the tiling section time are shown in figures 6.5, 6.6, 6.7, and 6.8. This is the same data that was presented in chapter 5 with the addition of the task adaptive version of the UD scheme, but here all the data is put on the same graph to allow direct comparisons. The comparisons in this case only involve the rectangular region and the task adaptive algorithms since these are the only algorithms which were implemented using both strategies. The data is shown above 48 processors so that the reader may get a clearer idea as to the performance difference, which is

mainly evident at high processor configurations. Based on this data, one can see that the LC scheme is consistently better than the UD scheme. While these graphs show that the LC scheme is clearly superior to the UD scheme in the tiling section, it is only fair to look at the total picture. By this we mean that all of the algorithms should be compared by evaluating the parallel execution time plus the setup time from the front end, as shown in the formulas on the page following the graphs. The total front end time will not be included here since disk access is used in that section of the program. Disk access time is affected by other parameters which cannot be controlled unless the machine is put into single user mode. In general, all of the algorithms employ the same disk read-in scheme anyway, so this is not an issue.

The primary differences in the algorithms occur in the following phases:

1. The time to load polygons into the area bucket data structure (or y-bucket list in the case of the parallel scan line algorithm) according to their screen space location.
2. The additional time necessary in the second pass for those algorithms which use the LC scheme.
3. The time to build the hierarchical tree for the data adaptive top-down scheme.
4. The tiling section time.

The table below shows how the comparison times are determined for each algorithm, including the memory referencing scheme and granularity ratio. Using these formulas, a fair comparison of all the algorithms is now possible since the different overheads prior to tiling are included. The primary variation in the setup time is due to the difference in cost for the total number of regions to be started ($R \bullet P$) in the implemented algorithms.

Algorithm (memory scheme)	*Phases*	*Granularity Ratio*
Data Non-Adaptive		
Scan line Algorithm (UD):	Phase 1 + Phase 4	(R varies with P)
Rectangular Region (UD):	Phase 1 + Phase 4	($R = 24$)
Rectangular Region (LC):	Phase 1 + Phase 2 + Phase 4	($R = 24$)
Data Adaptive		
Top-Down (LC):	Phase 1 + Phase 2 + Phase 3 + Phase 4	($R = 10$)
Task Adaptive		
Task-Adaptive (UD):	Phase 1 + Phase 4	($R = 1$)
Task-Adaptive (LC):	Phase 1 + Phase 2 + Phase 4	($R = 1$)

UD vs. LC Tiling Section Timing Comparisons

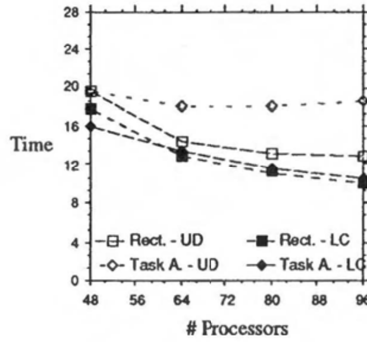

Figure 6.5: UD vs. LC stegosaurus image, tiling section only

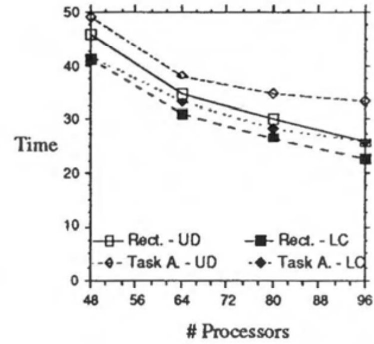

Figure 6.7: UD vs. LC tree image, tiling section only

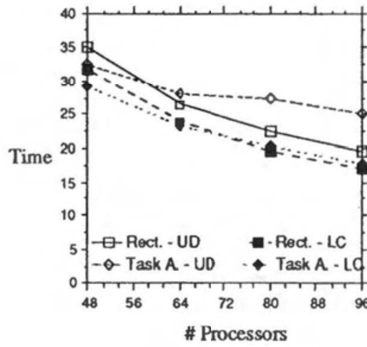

Figure 6.6: UD vs. LC Laser image, tiling section only

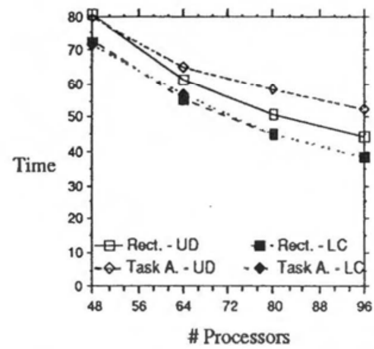

Figure 6.8: UD vs. LC mountain image, tiling section only

The graphs which result from these summations for each algorithm are shown in figures 6.9, 6.10, 6.11, and 6.12. The graphs are shown above 64 processors.

Based on the data shown in these graphs, it can be seen that the task adaptive algorithm utilizing the locally cached (LC) memory referencing scheme is clearly superior for all of the images. This algorithm requires fewer regions at the beginning of the program than any of the other algorithms. The overhead time for loading polygons into the area bucket list, as well as the second pass time, is fairly small as a result. The rectangular region algorithm, which is slightly faster for some images in the tiling section only, requires significantly more setup prior to tiling, degrading overall performance. One might have thought that the second pass section of the LC scheme would require too much setup time to benefit the total algorithmic performance, but this turned out not to be the case. While the second pass does add some time to the LC schemes, the benefits of local referencing in the tiling section far outweigh the cost of the setup operations since they can be done in parallel. This also indicates that network contention is a major factor in the resultant performance of each approach since the disparity in performance is greater than what was indicated in the theoretical analysis from the previous subsection. It seems clear that these results are consistent and valid for the tests done so far, but it is desirable to be able to generalize these statements by evaluating the various algorithms under a variety of other conditions. Some of these conditions are investigated in the next section.

6.2. Machine Parameters

Although the performance of the different algorithms has been analyzed previously, these circumstances represent only one possible machine configuration. There are various hardware and system software changes which may affect overall algorithmic performance, most of which are beyond the programmer's control. These types of parameters are investigated in this section. For instance, the operating system can have a significant impact on performance. In the implementation of the Mach operating system on the GP1000, single jobs are scheduled onto processors based on the current least loaded processor; however, the Uniform System takes over this task within a parallel program. The operating system does intervene to some degree in this machine by handling virtual memory, I/O, and general MACH system operations. Changes in the operating system

All Algorithms Compared including Setup Time

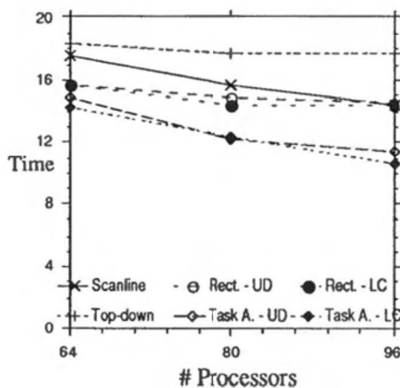

Figure 6.9: All algorithms compared, stegosaurus image, total time

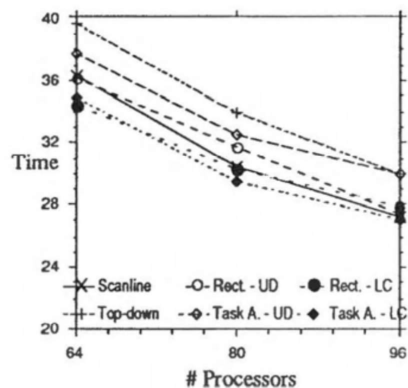

Figure 6.11: All algorithms compared, tree image, total time

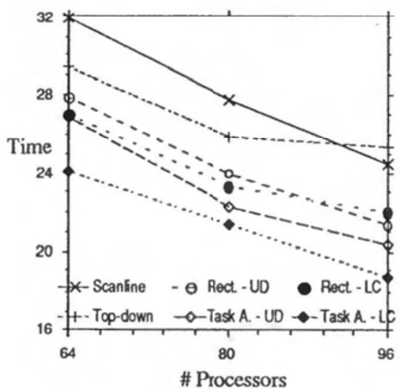

Figure 6.10: All algorithms compared, Laser image, total time

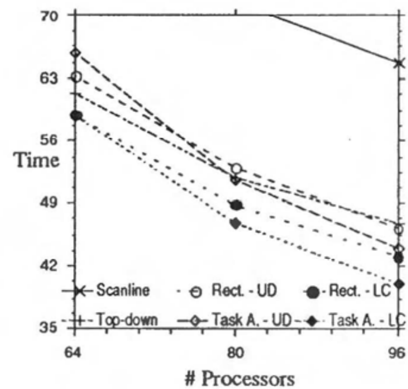

Figure 6.12: All algorithms compared, mountain image, total time

can change program performance; this is described in the first subsection below.

In the second subsection, we investigate the differences in two versions of the Butterfly multiprocessor: the GP1000 and the TC2000. Since the TC2000 is a logical extension of the GP1000 with different physical characteristics, it is interesting to compare the performance of these two machines.

6.2.1. Operating System

Since 1987, BBN has made a number of improvements to the GP1000, but none were so dramatic as the improvement made to the GP1000 version of the Mach operating system in the summer of 1990. The author previously reported preliminary results on this project in [Whit90] early in the summer of 1990, and the limits of the graphs were set to only 32 processors since inconclusive data was obtained above that. The primary reason was the previous version of Mach implemented on the GP1000.[1]

The older version of the GP1000 operating system had the following major problem: when any references occurred to a memory page which was not resident, only one page fault at a time was allowed to be serviced in the entire system. As an example, if processor **i** had a local page fault and processor **j** had a local page fault simultaneously, these page faults proceeded only serially even though they had nothing to do with each other. In a graphics algorithm such as the one described here, the amount of memory required is tremendous, and this serial page faulting had an extremely negative impact on performance. BBN rectified this problem and released a new version of the operating system in the summer of 1990; then performance changed dramatically. As an example of the difference in performance, we compare the rectangular region algorithm using the UD scheme in figures A.13, A.14, A.15, and A.16. A comparison of the LC scheme version is shown in figures A.17, A.18, A.19, and A.20. These figures are given in the appendix, but a copy of a representative graph for the mountain data using each of these schemes is shown in figures 6.13 and 6.14 on the next page. All of these graphs are comparisons of the tiling sections only.

As one can see from the graphs, the performance in the old operating system starts to tail off after about 48 processors in the UD scheme. The LC scheme is somewhat better since local rather than

[1]Note, the TC2000 has had the new version of the operating system since its delivery in the beginning of 1990.

global data referencing is taking place, although performance tails off here as well. Unfortunately, the amount of testing done using the old operating system was limited, so additional results could not be obtained. It is clear from these results, though, that the operating system in a shared memory multiprocessor has significant impact on the overall performance. We feel confident that the latest version of the GP1000 operating system is better geared to the current machine and does indeed provide exceptional performance.

6.2.2. Comparison of Architectural Differences

In addition to the impact of the operating system, other factors can affect overall algorithmic performance. For instance, one would like to compare what would happen if a faster CPU or a faster switch node were to be employed in the machine. BBN has continually updated the Butterfly family of machines from the Butterfly 1, which used MC68000 processors with 1 megabyte of memory per board, to the current generation GP1000, which uses the MC68020 with 4 megabytes of memory per board. We were not able to test the algorithms on the original Butterfly, but we were able to test them on the next generation BBN multiprocessor, the TC2000. The TC2000 is a similar design to the GP1000 but there are significant differences which are illustrated in the tables on the page following the graphs. Table 6.1 shows the difference in processor characteristics, while table 6.2 shows the difference in the memory characteristics for the GP1000 and TC2000. In general, the primary differences between the two machines are the faster CPU in the TC2000, as well as a change in the basic switch node component from a 4 x 4 crossbar to an 8 x 8 crossbar.

Table 6.1: Comparison of BBN multiprocessor CPU characteristics

Machine	CPU	Clock Speed	MIPS	MFLOPS
GP1000	M68020	16 Mhz	2.5	0.6
TC2000	M88100	20 Mhz	19	20

The faster CPU in the TC2000 necessitates a faster switch with increased path width, and an 8 x 8 crossbar switch component solves this problem. One impact of the increased size of the crossbar switch

GP1000 Operating System Comparison

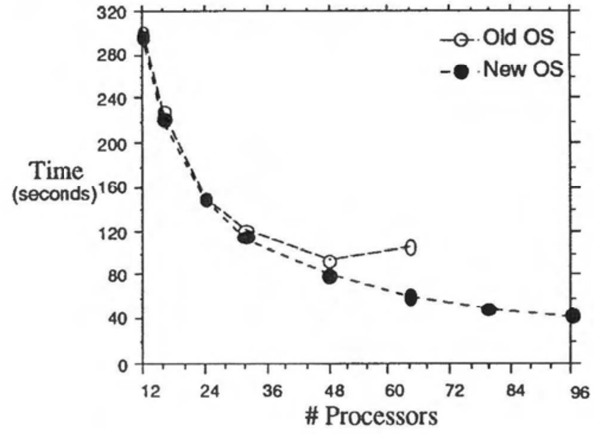

Figure 6.13: Comparison of old OS vs. new OS for mountain image, rectangular region UD

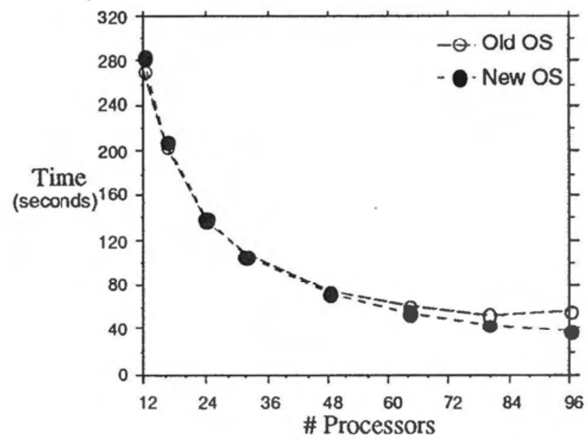

Figure 6.14: Comparison of old OS vs. new OS for mountain image, rectangular region LC

is that fewer wires are needed between the switch columns in the interconnection network. The 8 x 8 crossbar is more costly to produce than the 4 x 4 but it does allow 8 simultaneous messages to be output, whereas a 4 x 4 only supports 4 messages at a time.

Table 6.2: Comparison of BBN multiprocessor memory characteristics

Machine	Cache	Memory per Board	Switch Speed	Path Width	Basic Switch Node
GP1000	no	4 meg	8 Mhz	4 bit	4 x 4
TC2000	yes	4 or 16 meg	38 Mhz	8 bit	8 x 8

From the programmer's point of view, the TC2000 is functionally the same as the GP1000. There are several small differences regarding communication, however. The GP1000 supports the block transfer mechanism in hardware, whereby a path is held open long enough for 256 byte length messages to go from one board to another. In the TC2000, this operation is supported through software emulation rather than hardware implementation. The TC2000 does contain a memory cache which allows data to be allocated as cachable or non-cachable. Although using the cache significantly enhances performance, judicious management of this memory is required by the programmer since no cache coherence scheme is supported. The primary goal here is to compare the different algorithms under different CPU and switch characteristics, so the algorithms were not modified to take advantage of the cache.

The results, including times for the setup phase from the front end plus the tiling time, are shown in figures 6.15, 6.16, 6.17, and 6.18. A thorough analysis of the scan line algorithm was deemed unnecessary on the TC2000 due to its performance limitations noticed on the GP1000. It is, however, included for comparison purposes in the next section of this chapter.

These graphs indicate similar performance in the algorithms when compared to the previous graphs for the GP1000. The only problem with this comparison is that the results on the TC2000 were limited for most of the tests to a maximum of 48 processors, while with the GP1000, 96 processors were consistently available.[2]

[2]We have included some data obtained on the TC2000 at 96 processors in table 6.3. In general, though, due to the other users on the machine, only 48 processors were used for most of the tests.

TC2000 Tiling + Setup Time Comparisons

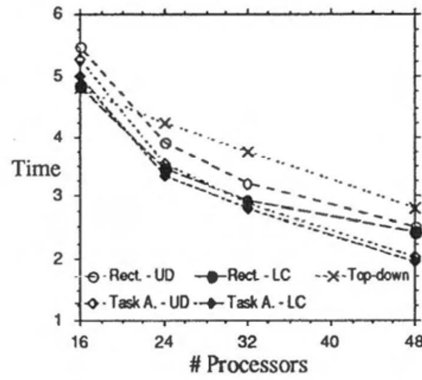

Figure 6.15: TC2000 algorithm comparison, stegosaurus image

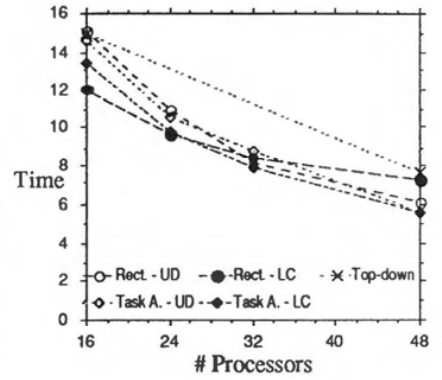

Figure 6.17: TC2000 algorithm comparison, tree image

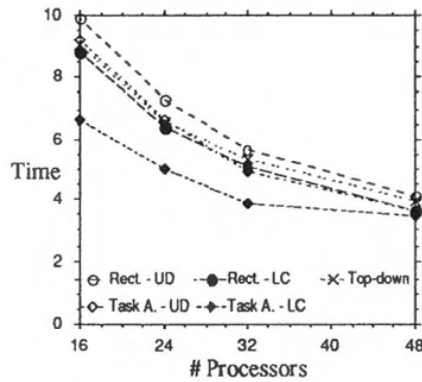

Figure 6.16: TC2000 algorithm comparison, Laser image

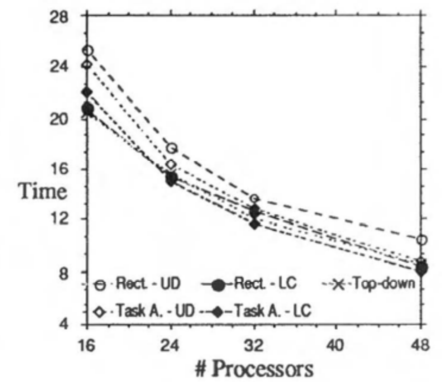

Figure 6.18: TC2000 algorithm comparison, mountain image

In order to allow a fair comparison between the two machines, the speedup was computed for each of the algorithms at 96 processors. The task adaptive algorithm is used for this comparison, and the results are shown in table 6.3.

Table 6.3: GP1000 and TC2000 speedup and time ratio comparison using 96 processors

Machine	Stegosaurus	Laser	Tree	Mountain
GP1000 Speedup	70.3	73.3	68.1	82.1
TC2000 Speedup	61.3	59.3	56.4	70.6
Ratio of Execution Times at $P = 96$: GP1000/TC2000	8.6	8.7	8.0	8.8

As can be seen from the table, the TC2000 exhibits slightly reduced speedup when compared to the GP1000 on 96 processors for most of the images. This could be caused by a number of factors, ranging from the amount of work per task to the processor-to-switch speed ratio. The last row in the table indicates the ratio of parallel execution times of the TC2000 divided by the GP1000. From this data, it appears that on 96 processors, the TC2000 is approximately 8.5 times faster than the GP1000 for this problem.

6.2.3. Relationship of Machine Parameters to Performance

In this section, we evaluate the various overheads on both machines to see their differences. The comparison involves examining the total processor-time space and comparing the results on the two machines. Here, the overheads are evaluated with respect to P and comparison values are shown to the right of each graph for the overhead percentages at 48 processors. Also, the speedup is given at each processor configuration. All of the algorithms are compared on the GP1000 and the TC2000 for the Laser image as a representative example. Due to the volume of data and the CPU time involved in the tests, only one image was used for comparison. Different results would be obtained for the different test images, but the main interest

here was to evaluate the trend in performance and directly compare the percentages on various processor configurations.

6.2.3.1. Comparison of Overheads

The next five pages provide a direct comparison of the overhead factors for all of the algorithms. The graphs include the total processor-time space for each particular processor configuration, with the overheads clearly marked as a percentage. Although the results were measured up to 96 processors for the GP1000, the overhead values given on the right side of the graph are for 48 processors so they can be compared to the values for the TC2000 below.

6.2.3.2. Analysis

These results present a number of interesting phenomena not noticed in any previous graphs. In the parallel scan line algorithm, the latency and code modification overheads constitute almost the same overhead percentage regardless of the processor configuration on both machines. This makes sense since the number of tasks is constant regardless of the number of processors in this algorithm. In the other cases, since the total number of tasks increases with the number of processors, the overhead effects increase as well. In some cases, the load balancing may go down at some point but this may be due to an increase in another factor as explained next.

With the exception of the task adaptive algorithm, the load balancing is better on the TC2000 than in the GP1000. On the other hand, the network contention, code modification, and latency/communication are significantly worse. It seems that the increased delay due to communication overheads and contention contribute to even out the load in the algorithms on the TC2000 (recall that load balancing cannot be measured independently from other factors). Since these overheads are larger in the TC2000 than in the GP1000, they contribute to an increase in the average task execution time. This changes the load balancing since it is based on dynamic scheduling of the tasks, as well as their execution time.

In the case of the task adaptive algorithm, the load balancing is a direct result of dynamic task partitioning, and it is possible that the tasks cannot be partitioned near the end of the computation due to the imposed threshold. This effect may be more pronounced in the TC2000 than in the GP1000 due to the difference in the synchronization and communication mechanisms.

Comparison of Overhead Factors, GP1000 vs. TC2000, Laser Image, Scan line Algorithm

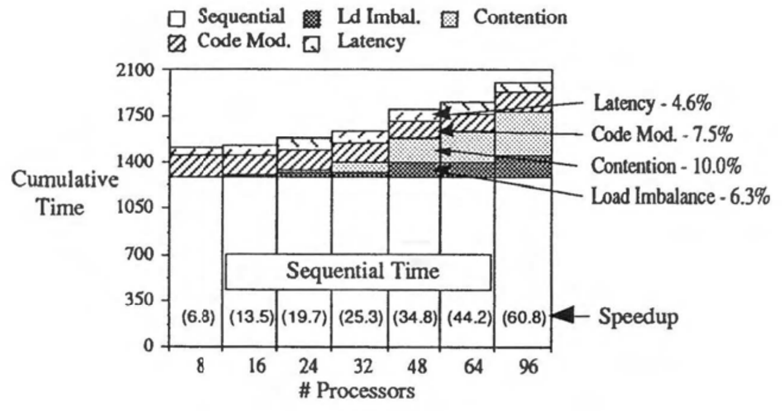

Figure 6.19: GP1000, scan line algorithm, UD, overhead comparison

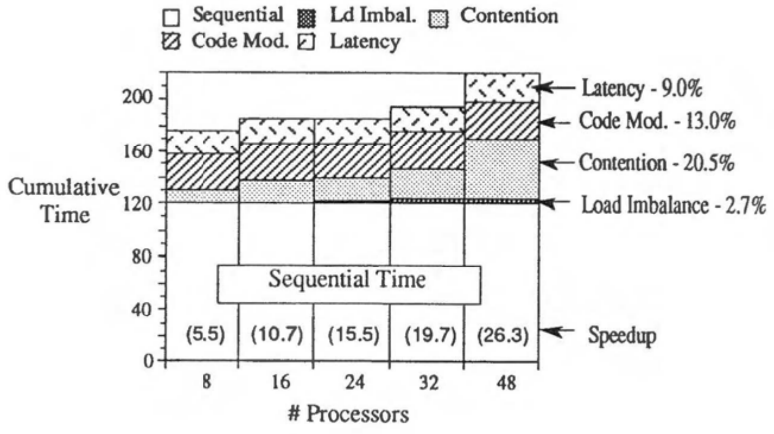

Figure 6.20: TC2000, scan line algorithm, UD, overhead comparison

Comparison of Overhead Factors, GP1000 vs. TC2000, Laser Image, Rectangular Region Algorithm, UD Scheme

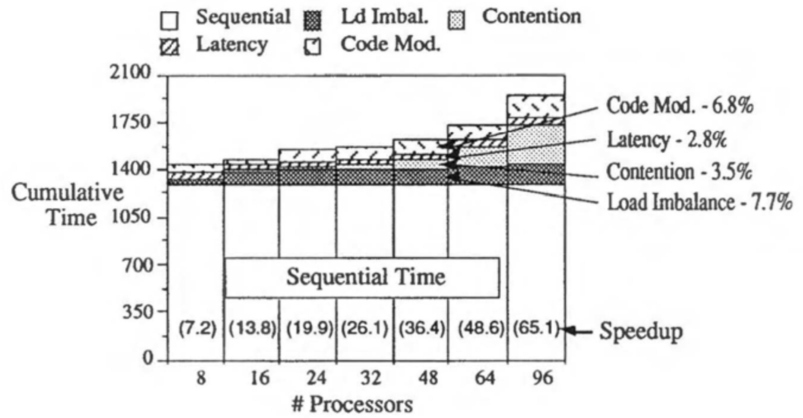

Figure 6.21: GP1000, rectangular region algorithm, UD, overhead comparison

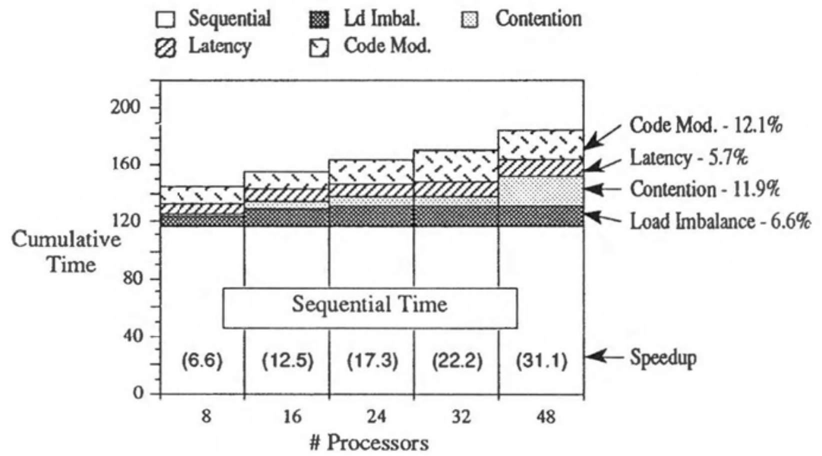

Figure 6.22: TC2000, rectangular region algorithm, UD, overhead comparison

Comparison of Overhead Factors, GP1000 vs. TC2000, Laser Image, Rectangular Region Algorithm, LC Scheme

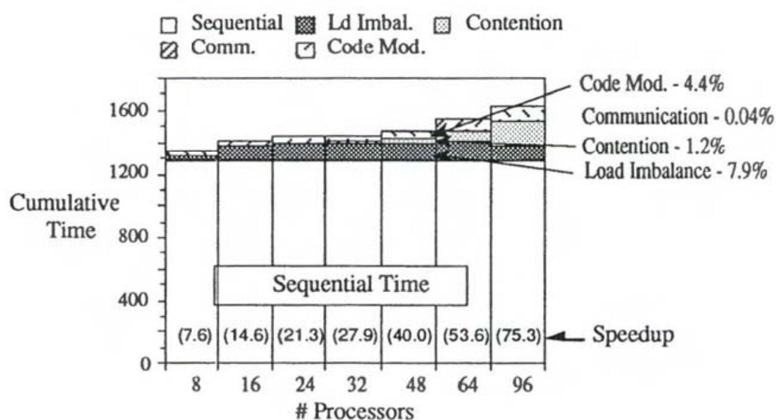

Figure 6.23: GP1000, rectangular region algorithm, LC, overhead comparison

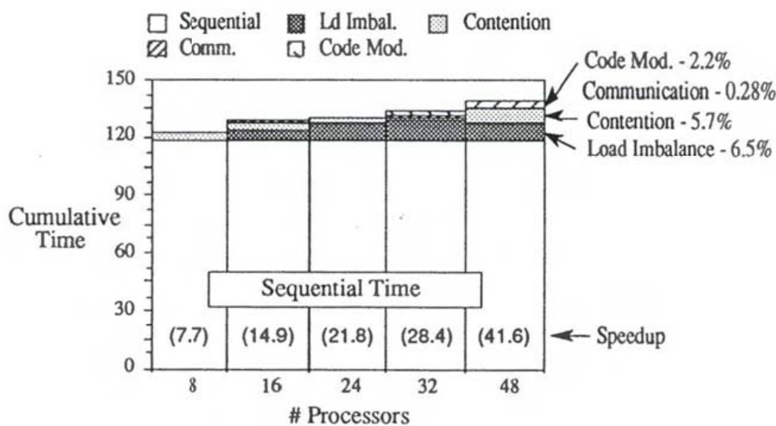

Figure 6.24: TC2000, rectangular region algorithm, LC, overhead comparison