View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by NASA Technical Reports Server

N87-26559

Algorithmic Commonalities in the Parallel Environment

Michael A. McAnulty and Michael S. Wainer
Department of Computer and Information Science
University of Alabama in Birmingham
Birmingham AL 35294

ABSTRACT

The ultimate aim of this project is to analyze procedures from substantially different application areas to discover what is either common or peculiar in the process of conversion to the MPP. Three areas were identified - molecular dynamic simulation, production systems (rule systems), and various graphics and vision algorithms. To date, only selected graphics procedures have been investigated. They are the most readily available, and produce the most visible results. These include simple polygon patch rendering, raycasting against a constructive solid geometric model, and stochastic or fractal based textured surface algorithms. Only the simplest of conversion strategies, mapping a major loop to the array, has been investigated so far. It is not entirely satisfactory.

Key Words:Graphics, Stochastic Surface, Constructive Solid Geometry, Conversion Strategy, Fractal Surfaces

INTRODUCTION

Intuitively, certain procedures appear to map easily to an array machine, and others do not. The latter class includes those with long stretches of inherently single-thread code, for which even incredible speed-ups in their parallel part will produce little net improvement, and those whose internal communication structure is irregular, such as highly general neural nets. The three classes to be discussed were chosen not for their impossibility, but rather for their *prima facie* differences, as well as for their presumed availability during this work.

Molecular dynamics simulates the motion of complex macromolecules, which can be shown to exhibit a quite interesting ensemble of configurations. The simulation relies upon the locality of effects, so that each atom interacts only with a few neighbors, and accuracy and stability are sought by using a sufficiently small time step. The locality of interaction and the uniformity of the calculations for each atom appear to make this mappable to an array machine, albeit with some significant programming work. We intended to use some existing code which has been vectorized, but this is now being marketed, and we are still discussing the possibility of using it without violating its proprietary nature.

Rule systems invoke a branching chain of inferences using a domain rule data base suitably encoded. A rule is activated by determining that certain facts in a fact data base satisfy its left hand side, and it executes by making modifications to the fact data base. Some variant of this underlies most expert systems currently available. As it happens, the expansion of an inference is often a highly branched tree process. The necessary depth of a search is the discouraging component of the process, but this is often outweighed by a considerable breadth as well. The major bottleneck in production systems appears to be, in fact, not the depth of the process, but its breadth which is necessarily invoked in passing left hand sides against the fact data base to detect satisfied rules, a process termed matching. Noting that, with the broadcast facility, an array machine may function reasonably well as a large associative memory, it appears possible that rule systems could be significantly optimized in an array machine. Unfortunately we have lost the investigator who was to have pursued this line, using an existing production system, and it has not been followed up yet.

The final area, graphics and vision algorithms, is represented only in the graphics area. We may plead, citing a fairly common observation, that in fact vision and graphics appear to be the same process run in two different directions between an image and its description (Ref. 7), but in fact they are nowhere near that in practice. A short section will address vision issues. The bulk of the following represents graphics issues that have emerged during our involvement with the MPP.

POLYGON PATCHES

The description of a scene as a collection of simple patches that together describe surfaces of solids is perhaps among the oldest of solid models, and a discussion of its aspects may be found in several

PRECEDING PAGE BLANK NOT FILMED

comprehensive references (e.g., Ref. 1). There is a considerable leap between models with simple planar patches and those with simple curved patches, this discussion treats only planar patches. The process of mapping a patch to an array of display elements (pixels), which is termed *scan-conversion,* is relatively straightforward, if tiresome, for planar patches, and not nearly so for nonplanar patches.

To render (that is, display) a polygonal patch requires projecting it to the two-dimensional display surface, a simple scaling and shifting operation, and then determining which pixels it in fact projects to. This last operation, the actual scan-conversion, will usually consist of counting from one boundary to the next in pixel-sized increments. It is roughly as time consuming as the number of pixels actually covered, although there is also some overhead involved in setting up the control parameters of the loops involved. Polygons are as easily represented as a combination of half-spaces, simple linear constraints, as they are by vertices, and one form may be generated from the other with comparable facility. A quite straighforward array process for scan conversion may be described, which consists of each array element testing its location against the list of half spaces, and 'lighting itself up' if it passes all the constraints. There is an added advantage to working on the array, in that each processor may serve as a 'z-buffer', keeping track of the depth which its current pixel represents (if indeed it has been filled). This can be simply calculated from the 3-space plane of the polygon, and is useful when patches overlap. If a new patch does project on a processor at a nearer depth, it simply wipes out the previously stored value.

The simple (but effective) method of passing linear constraints to the array, one polygon at a time, is in fact no less onerous than it would be to pass them to some other firmware scan-converter, the usual process in the sequential world. The process is advantageous if rather large polygons are involved, the scan-generation process becomes a single 'multi-step', but one is reminded that during this step most of the array elements are not doing 'useful' work.

An alternative approach, to somehow load the array with patch definitions and let each scan generate in parallel, confers both advantages and considerably more work. The advantage is that an animated sequence, where the collection of patches moves in time, does not require a sequential scan of the patch data base for each new frame. The challenge is that scan conversion is no longer as direct, a process on the array may not be located in the pixel it is eventually supposed to illuminate, and thus the values the process generates must then be routed to the correct pixel. We have concentrated on processes that map array elements directly to pixels, and the costs of a general routing process remain to be more carefully considered. It should be noted that the hidden surface problem is still nicely addressed, as destination pixels may still function as z-buffers.

The simple half-space approach corresponds to a general methodology, unwrapping a controlling loop and spreading it over a processor array. In the planar patch model, the innermost loops that direct scan generation are the ones unwrapped. The outer loop, which counts through the model data base exactly once (since occlusion is handled by the z-buffer), is totally determined. The inner loop in fact counts over pixels, but in a piecemeal and unpredictably repetitive fashion. An alternative approach is to count over pixels, hitting each exactly once, and for each pixel traverse the model data base to see which surface, if any, projects to that pixel. This is the control structure of the next rendering method to be discussed.

RAYCASTING SOLID MODELS

For complicated but man-made objects the constructive solid geometry (CSG) models are unsurpassed for direct representation and conciseness. They are particularly well-adapted to rendering by ray-tracing, which is still the best method with which to exercise full control over the lighting model, including shadowing, matte and specular reflection, and refraction. (see, for example, Ref. 10) In the single thread implementation, the major loop counts through all pixels, constructs a ray through each pixel, and determines its intersection with the model. The outer loop, once per pixel, can be mapped onto the processor array quite easily, so that each element holds a different ray, and the model may be broadcast one node at a time. The model need be traversed only once (but see further), which is of some modest advantage for CSG because models tend to comprise components numbering from scores to hundreds. However, for other models such as oct-trees, where model traversal is more expensive, this advantage increases.

The operation at each pixel, intersecting it with the solid model, is directed by traversing the model, formed as a tree, in preorder. Leaf nodes represent one of a small set of primitives (cube, sphere, cone, torus), interior nodes represent set combinations of their subtrees, which may be intersection, union, or subtraction, and these are always binary. Each visit to a leaf produces a list of intersections, always in in-out pairs. The various intersections are represented as a parameter value along the ray, with larger values being further from the viewer. The grazing phenomenon,

where a ray touches an object once, can always be forced out at this level. Each visit to an internal node produces two such lists which must then be combined into a single list according to the operation represented at the node.

It would be sufficient to retain only the nearest in-out pair if it weren't for the quite powerful set subtraction operation- no matter how many hits one saves, they may all be subtracted further up the tree, and it may turn out that one needed just the ones thrown away. It is possible to force subtraction to be done at low levels in the model tree, which will alleviate this problem, although the general control structure remains unchanged. At any interior node in a tree, several lists of intersections from left subtrees at higher levels may be in effect, waiting to be combined with right subtrees, and each list may be zero or more intersections. There will, if empty lists are explicitly treated, be exactly the same number of stacked lists for each ray (processor), so that a combine operation which mixes a current right branch intersection list with the topmost left intersection list may be done in parallel, with processors that exhaust their left list simply shutting off for the duration of the combine. Because there should be no limit on tree depth, there is thus no limit on the stack size even if we restrict a single intersection list to two members, thus it appears that we shall require the staging memory to stack the lists. More onerous, since a parallel reference must be to the same address in PE memory (and by extension in the stager as well), it appears that stack operations should physically relocate the entire current stack. The solution to this problem is still under consideration, and may perhaps be addressed by restricting single lists to pairs, which may have real values or dummies that indicate no intersection.

The stacking problem relates only to the traversal of the model. When this has been accomplished, only the foremost intersection matters. Any further work to be done on that pixel consists of casting further rays from the intersection point. We may need to cast shadow feelers to light sources, reflected rays from specular surfaces, or refracted rays at non-opaque surfaces. It becomes necessary, then, to include with all intersections in the stack the physical properties of the object being intersected, which include color, transmittance, specularity, and the surface normal of the object at the intersection point. More important, we require parallel management of these further ray-casts.

Shadow feelers, for simple shadowing, are straightforward since every intersection must cast to the same light sources, combining the results of the cast with its matte reflectance and color. At this point, non-specular and non-transmitting intersections may turn themselves off. Those intersections which are specularly reflective to any degree may cast only once, unless this second ray also encounters a highly reflective surface. For many applications we might restrict the number of times this might occur. A similar consideration applies to transmitting surfaces, which could be treated as a pair of in-out rays with a final modulated intersection.

The extra contributions to a pixel's value from refracted or reflected objects need to be combined in weighted fashion to calculate the value. This applies as well to further recursive extensions of such feelers. At each extension, the net contribution of a particular feeler to the ultimate value of the original pixel, its weight, grows less and less. Thus, we may generate as many rays as we need, attach to each its weight, and continue generating feelers until no new feeler has a weight above some threshold. Each new feeler results in the stacking of weighted values, which will generally have zero weights for dull surfaces, and combination may be a straightforward stack operation. However, this results in a doubled data requirement, and as many extra model traversals as extra rays are generated. The anticipated speed-up of 16,000 will not be attained, since only a small minority of the rays, in general, require further casting, so the majority of rays are idle (processors doing no useful work) while reflectors and transmitters are followed, each stage of which requires a further sequential traversal of the model.

One might, in the case of a really fancy lighting model offload certain rays to other processors which are no longer propogating, but the nature of routing such rays appears to be infeasible, and there is in fact little paralellism in the feelers, one must generally follow another.

Interim Discussion

The half-space polygon rendering model has been implemented and run, in highly primitive fashion, on the MPP, largely as an exercise. The CSG raycaster is still an incomplete paper exercise. The only conversion methodology applied in either case is the quite straightforward unwrapping of major control loops. The major challenge appears to be manipulation of model elements entirely within the array. This was alluded to in the discussion of patch models, and is necessary for anything like real-time manipulation, in three dimensions, of the model. Before an actual implementation of the ray-caster is attempted, further study of its internalization will be done.

While "late vision" algorithms have some relation to graphic modelling, a major challenge at this stage is the matching of elements of the model with the image, an interesting problem we have yet to deal with. Early vision procedures are of lively current interest, and tend to be highly parallel and local computations involved in regularization and related relaxation procedures. (Ref. 9) These appear, intuitively, to be "easy" procedures to map to a matrix machine. However, the graphics procedures we have dealt with also appear easy in much the same sense. The initial methodology has been to do those easy tasks and see what stumbling blocks in fact appear. For both patches and CSG neither "easy" mapping has been entirely satisfactory. Further, there is a strong possibility that many algorithms may be too numerical in nature and realization, numerical computation having been our major medium in the past. We should investigate more efficient uses of connected bit-serial processors which minimize arithmetic.

The final and dominant segment of this project (Wainer) maps a different process to an array in some detail, and has been fully implemented. The control structure is again recursive, but unlike the CSG traversal the recursion itself, rather than a "major loop", is mapped directly to the array. Further, some use is finally made of the communication structure of the array.

STOCHASTIC INTERPOLATION OF RECTANGULAR PATCHES

Stochastic interpolation of rectangular patches as described in (Refs. 2,3,8) is useful in generating dense patch data from a small number of input values. In computer graphics the method is used to create surfaces, textures and sky among other things (Refs. 2-4,8). The values generated are pseudorandom and approximate a fractal distribution. The characteristics of the surface are controlled by the h and scaling parameters. Values of h near 1 give a smoother surface than those near 0. The scaling parameter is used to adjust the magnitude of the stochastic contribution to the interpolation function. For more about fractional brownian motion and its applications see references 3 and 8.

The process begins with a parent patch whose corner data values are given. These values may be interpreted as altitudes or colors or some other attributes which are to be interpolated across the patch. The parent patch is repeatedly subdivided until the child patches are of the desired resolution (ie. pixel sized). The interpolated values are formed from the deterministic average of neighboring data values and a stochastic component which is determined by the parameters and pseudorandom numbers.

A graphical depiction of the method is shown in Figure 1. The four corners of the parent are used to calculate the center point of the child. Side points along an edge of the original parent use only the data derived from the two original points which define that edge. Side points in the interior of the original patch use the four values from their vertical and horizontal neighbors. By stipulating that points that lie along the edges of the original patch depend only on the vertices that define those edges, continuity between adjacent input patches can be guaranteed.

The stochastically interpolated values are formed from both a deterministic and a stochastic component.

$$I = I_{\mathrm{det}} + I_{stoch} \quad (1)$$

I_{det} is the average of two or four neighboring vertices as shown in Figure 1. The same neighbors which determine I_{det} are used to create a seed value for I_{stoch}. A pseudorandom number is selected using this seed and is conditioned by the generating parameters h and scale. This method of forming the seeds for pseudorandom number generation assures continuity between adjacent patches as mentioned above. To prevent the selection of pseudorandom numbers solely on the basis of the original parent patch's corner data values, seed values are also provided with each original data value. Points whose data values were given at the start of the procedure supply seed values for I_{stoch} of their dependents.

Complexity of the Process

It is easy to see that the number of patches grows exponentially in powers of four. Even though patches share some of the same vertices, we cannot better the overall complexity of the process by more than a constant factor. (Consider that each patch in the interior has $1/4 + 1/4 + 1/4 = 3/4$ new points to calculate: this is true in every generation. Patches on the edge must generate $1/2 + 1/2 + 1/4 = 5/4$ new points.) Thus whatever sequential algorithm may be used, its time complexity will still be at least $O(4^L)$ where L is the number of subdivision levels. Furthermore even though the algorithm appears inherently parallel, the data dependencies of children on their

parents limits the best we can do to an algorithm which has a time complexity proportional to the number of subdivisions or equivalently to the log base 4 of the number of final patches. The data dependencies make matters even worse for the sequential algorithm which now becomes $O\left(\sum_{1}^{L} 4^i\right)$

Mapping the Algorithm on to the MPP

First note that we can map any arbitrary rectangle into a square be scaling one of its sides. For our purposes then we treat the input patch to the subdivision algorithm as if it were a square. At each subdivision the child patches formed are also square and have sides of length one half of their parent's side. This geometry is ideal for the array unit which is square and has a side length of 128 (a power of two). After seven subdivisions the parent patch will have created 4^7=16384 descendants or one per MPP PE. A mapping using the vertices of the patches instead of the entire patch is not so neat and after seven subdivisions yields a 129x129 array of data values. Here we use the mapping of patches to PEs.

Child patches must be able to obtain data from their parents and also their siblings and sometimes even cousins. Since a seven level subdivision will fill each PE with a patch we will have to be able to recycle PEs which were parents in earlier generations. This may seem a lot to ask of the mesh connected communication network of the MPP but there is a straightforward and somewhat elegant solution.

Observe that all the child patches are dependent solely on the algorithm's parameters and the data of the input parent (These critters are asexual.). It is easy to initialize all the PEs with the input patch's data. Treat the entire array as one group of PEs which all represent the input patch. The first subdivision will yield four new patches each of which will also be represented by a group of PEs. Each child patch required no communication overhead to obtain parent data because, in reality, it already contained all of its parent's data. Which PEs belong to which child patch is determined by an id contained in each PE. The id corresponds to which child patch the PE represents at each level of the subdivision; after seven subdivisions the id in each PE maps it one-to-one with a single patch. Until the final subdivision, each patch is being mapped onto a group of PEs which are redundantly calculating the subdivision for that patch. The redundant calculations are occurring simultaneously so they incur no time cost. The benefit of this method, besides its simplicity, is that parent data is passed onto the children without need of the communication network.

Data routing is still used to obtain values from neighbors. Here the term neighbor is with respect to groups of PEs rather than individual ones. Since groups are largest in earlier generations, communication distances are largest for them too. During the first subdivision a routing distance across half the array is necessary for PEs of neighboring groups to communicate. At the final subdivision, adjacent PEs are group neighbors so the communication distance becomes just a single PE.

Even though the number of PE groups grows exponentially, the number of different PE types remains constant at four which correspond to the quadrants of the parent patch. Figure 2 shows how PE groups and types map onto the array. Before subdividing the first input patch, the id for each PE is determined. This is a simple process which forms seven 2-bit numbers from the concatenation of the row and column indexes already present in each PE. Figure 3 identifies the PE id types with the quadrants they correspond to. The subdivision algorithm is outlined in pseudocode below.

algorithm subdivide(levels:integer; h,scale:real);

Subdivide the parent patch defined by the values at its 4 corners to the number of levels given by "levels". H and scale are the fractal h parameter and user selected scaling factor respectively. Initializations which need be done only once per run of the algorithm such as valid = TRUE and the ID set up are assumed to have already been done and are not included in this analysis.

```
{ **** set up ***** }

1     route := 128;  { # PEs on side of PE array }
2     ratio := 2^-h ; { from fractal h parameter }
3     std := scale; { user determined scaling parameter }

4     initflag(flag); { flag shows original (input patch data)
                        where true. Used in determining the
                        calculation of the seed for the pseudo-
                        random numbers used in the I_stoch }

{ **** subdivisions **** }

5     FOR level := 1 to levels DO BEGIN
6           route := route div 2;
7           std := std * ratio;
8           center_pts(level,std);
```

```
9            side_pts(level,route,std);
10     END
```

Analysis of subdivide

The set up statements in lines 1 through 4 are executed once in constant time no matter what the value of levels. Initflag, initializes a parallel array of booleans but this uses the broadcast function and so is not dependent upon the array size. Flag is used to mark whether or not the corner data values were given as input (TRUE) or derived. Vertices marked as true forward their seed values for calculation of I_{stoch} . Flag is updated each time new data values are calculated since the new values may replace a value that was marked as TRUE in the previous generation.

Line 6 determines the routing distance between neighboring PE groups. This is halved at each subdivision until it reaches one at the final subdivision. Route is used by side_pts when obtaining information from neighboring PE groups. At line 7, std, derived from the h and scaling parameters, is adjusted for the next application in the calculation of I_{stoch} .

The lines of most interest are 8 and 9 which compute the subdivisions in two phases: center points and side points. Computation of the center points is done simultaneously throughout all PE groups without the need for shifting data. The four corner points of the parent patch are used and these are already contained within each PE. I_{det} is the average of these point values and I_{stoch} uses the corners in the selection of the pseudorandom number which is adjusted by the current value of std. The time complexity of this step will remain constant with varying levels and array sizes.

The computation of the side points is more complicated and involves shifting data in from neighboring PE groups. Since the size of a PE group varies according to what level is being processed the communication time will also vary. The algorithm as described here uses communication for the following reasons:

a. To establish if there is a valid neighbor along a particular side.
b. To gather data from a valid neighbor.
c. To transmit shared points to a neighbor.

In each of the above, neighbor refers to a neighboring PE group.

Since the calculation of side points is the only step which requires communication, we shall now shed some more details on this process. To satisfy edge constraints, interpolates of values along an outside edge (the edge of the parent patch) must depend ultimately, only on the two corner values of the parent patch's edge. When this holds patches can be matched up simply by assuring that their common edge values and generating parameters are identical. Edges not along the outside of the parent patch use values above and below and right and left to calculate their values.

The deterministic interpolate is the average of its two (when on an outside edge) or four nearest neighbors. Two of the neighbors are the vertices of the side that it lies on and the other two (when four are used) are the just calculated center points which form a perpendicular bisector of the side the interpolate lies on.

A sentinel is used to easily determine if the new vertex has a valid neighbor. Define VALID to be a parallel array of type boolean. Using the MPP broadcast instruction it is easy to set each bit of VALID contained in each PE to TRUE in constant time. Recall that attempts to shift data in from beyond the edges of the array, read in 0's (FALSE). Thus a neighbor is valid only if its VALID bit is TRUE.

Each PE group first calculates its center point then the edge interpolates. We will first discuss only the deterministic part of the calculations. The center point is trivial; it is merely the average of its four corners (plus a stochastic component). The basic algorithm for the deterministic edge components is as follows.

Note that the center point has already been computed and its value is now stored in the corners of the new child patches which lie at the center of their parent patch. There are now four different types of patches forming. The differences are based solely on the positions of the new patches with respect to their parents (see Figure 3). Each type is computed concurrently with all the others like it, thus there are four different types of edge interpolate calculations. All these are very similar and simply correspond to the particular edge being computed: top, bottom, left or right.

We can characterize the time complexity of the side point calculation step as being composed of two components:

$$T_{side_step} = T_{calc} + T_{comm(level,array_size)} \qquad (2)$$

In other words, the time for the side points step calculation, T_{side_step} , is made up of a nonvarying calculation portion, T_{calc} , plus a communication time, T_{comm}

. The communication time is a function of what subdivision level is being processed and of the size of the PE array. T_{comm} decreases exponentially with level and increases by the square root with array_size. Equation 2 can be rewritten as equation 3 by combining level and array_size to form the routing distance.

$$T_{side_step} = T_{calc} + T_{comm(route)} \quad (3)$$

If we let ROUTE be the routing distance at the first subdivision and we are using the array to process a single input patch, then ROUTE is half the number of PEs on the array side. While holding the dimensions of the array constant, the execution time of the algorithm will be a function of L, the number of subdivision levels.

Let T_{step} be the combined times for the nontime varying calculations done at each level of the subdivision. Thus T_{step} includes T_{calc} from equations 2 and 3 plus the time to calculate the center points. Execution time of the algorithm and its startup time are denoted by T_{exec} and $T_{startup}$ respectively. Assuming that the shift function is straightforwardly implemented and communication time is directly proportional to the number of bits being routed and the distance that they cover, then communication time can be expressed by a constant, K_{comm} , multiplied by the distance data must be routed (equation 4). Rewriting equation 4 to remove the summation we can derive equation 5.

$$T_{exec} = T_{startup} + \sum_{0}^{L-1} \left[T_{step} + K_{comm} * \frac{ROUTE}{2^i} \right] \quad (4)$$

$$T_{exec} = T_{startup} + L * T_{step} + K_{comm} * (2ROUTE - 1) \quad (5)$$

Since ROUTE is a fixed value for a given machine, the time complexity is not affected by it or the other constant terms. The execution time complexity of the algorithm, denoted $Comp(T_{exec})$, is directly proportional to L, the number of subdivision levels. The complexity is log the number of final patches produced and, due to the data dependencies inherent in the algorithm, this is the best result that can be expected.

$$Comp(T_{exec}) = L \quad (6a)$$

$$Comp(T_{exec}) = log(number\ of\ final\ patches) \quad (6b)$$

To be able to subdivide more levels using the same algorithm, larger arrays of PEs must be built. (It should be noted that the 128x128 array size of the MPP is a reasonable size for graphics applications which are still typically 512x512 pixels per screen.) If we allow the array size to grow and continue the algorithm to the maximum level, $L_{\max}$, that the array size will allow, we can compute $L_{\max}$ from ROUTE (equation 7).

$$L_{\max} = \log_2(ROUTE) + 1 \quad (7)$$

Using equation 7 and substituting into equation 5 with $L = L_{\max}$ we obtain equation 8. This gives the execution time as a function of L when the architecture of the machine varies to always have one PE per patch after the final subdivision.

$$T_{exec} = T_{startup} + L * T_{step} + K_{comm} * (2^L - 1) \quad (8)$$

As L grows large, the third term begins to dominate. But the number of final patches being produced is growing as 4^L . Taking the limit as L grows large shows that the parallel algorithm is still substantially better than the sequential one (equation 9). Recall from earlier analysis that the sequential time algorithms must be at least $O(4^L)$.

$$\frac{2^L}{4^L} = \frac{2^L}{2^{2L}} = \frac{1}{2^L} \rightarrow 0 \text{ as } L \rightarrow \infty \quad (9)$$

The time complexity can be decreased below L if the data dependencies can be eliminated. This is true if multiple input patches can be processed concurrently. Instead of enbedding a single patch in the array, begin by embedding four or sixteen, etc. Each

patch is processed concurrently using a ROUTE which corresponds to the number of PEs it covers. Slight modifications to the identifier labels and the data sentinel for boundary detection would be required to the basic algorithm.

Conclusions

A parallel algorithm to subdivide rectangular patches using stochastic interpolation was developed. The algorithm was designed for mesh connected SIMD computers and was implemented on the MPP at Nasa Goddard Space Flight Center. For a fixed architecture SIMD, the algorithm has time complexity of log the number of final patches produced; this is the best that can be expected due to the data dependencies imposed.

Timing data was collected for a nonoptimized version of the algorithm using parallel pascal on the MPP. Approximately 30 milliseconds of processing time is required to subdivide one patch seven levels into 16384 final patches. This figure compares favorably to the 60 milliseconds required on the special purpose STINT processor (Ref. 8). Figure 4, Color Plate VI shows the evolution of the algorithm as it produces a 512x512 pixel "sky" from 16 input patches. Figure 5, Color Plate VII shows a scene composed of textures generated similarly but using different parameter values and color mappings.

A similar algorithm which recursively subdivides triangles (Ref. 2) also maps well to SIMD mesh connected machines and is detailed in (Ref. 11). Parallel machines such as the MPP, besides running existing paradigms faster allow insights into ways they may be expanded. A direction for future research is how the generating parameters of this stochastic interpolation algorithm may be increased to higher dimensions to make the generic algorithm more powerful.

References

1. *Fundamentals of Interactive Computer Graphics,* Foley, J. D. and van Dam, A., Addison-Wesley Reading MA, 1983
2. Computer Rendering of Stochastic Models, Fournier, A., Fussel, D., Carpenter, L., Comm ACM, 25(6), pp. 371-384, Jun 82
3. Frame Buffer Algorithms for Stochastic Models, Fournier, A., Milligan, T., IEEE CG & A, 5(10), pp. 40-46, Oct 85
4. Using Stochastic Modeling for Texture Generation, Haruyama,Shinichiro, Barsky, B., IEEE CG & A, 4(3), pp. 7-19, Mar 84
5. Fractional Brownian Motions, Fractional Noises and Applications, Mandelbrot, B.B., Van Ness, J.W., SIAM Review, 10(4), pp. 422-437, Oct 68
6. The Fractal Geometry of Nature Rev. ed., Mandelbrot, B.B., W.H. Freeman and Company, San Francisco,1983
7. *Algorithms for Graphics and Image Processing,* Theo Pavlidis, Computer Science Press, Rockville MD, 1982, pp.1-4
8. A Hardware Stochastic Interpolator for Raster Displays, Piper, T.S. Fournier, A., Comp Graphics, 18(3), pp. 83-92, Jul 84
9. Early Vision: From Computational Structure to Algorithms and Parallel Hardware, Tomaso Poggio, *Computer Vision, Graphics, and Image Processing 31:* 139-155 (1985)
10. Ray Casting for Modelling Solids, Scott D. Roth, *Computer Graphics and Image Processing 18(2):* 109-144, Feb. 1982
11. Generating Fractal-like Surfaces on General Purpose Mesh-Connected Computers, Wainer, M., submitted to IEEE Trans on Comp., Jan 86

Acknowledgments

Mr. Wainer's work was made possible by a NASA Graduate Student Fellowship, Training Grant NGT 02-002-800, and on-site sponsorship and assistance of Dr. James Strong. Dr. McAnulty's access to the MPP was made easier through the cooperation of Dennis Gallagher, ES-01, Marshall Space Flight Center, who enabled direct access to the SPAN network at the Space Science Laboratory.

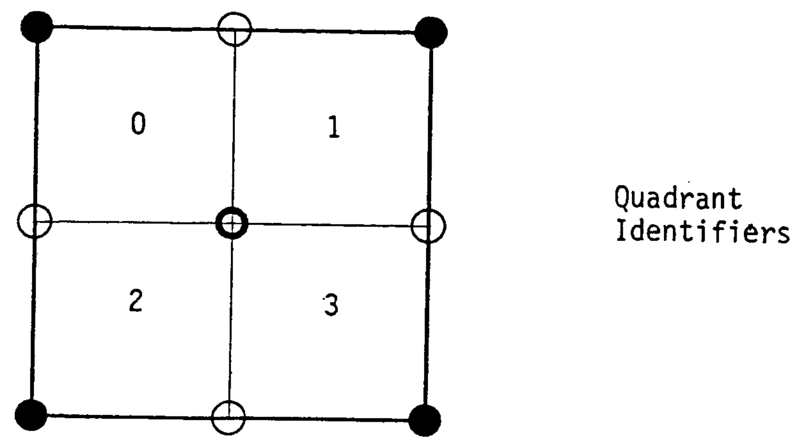

Figure 3: Quadrants and their type designations

1st Subdivision

2nd Subdivision

- parent edge
- sides of previously computed patches
- sides of child patches
- previously determined data values
- original input values
- current center point values
- current side values

Figure 1: A Graphical Description of the Subdivision Algorithm

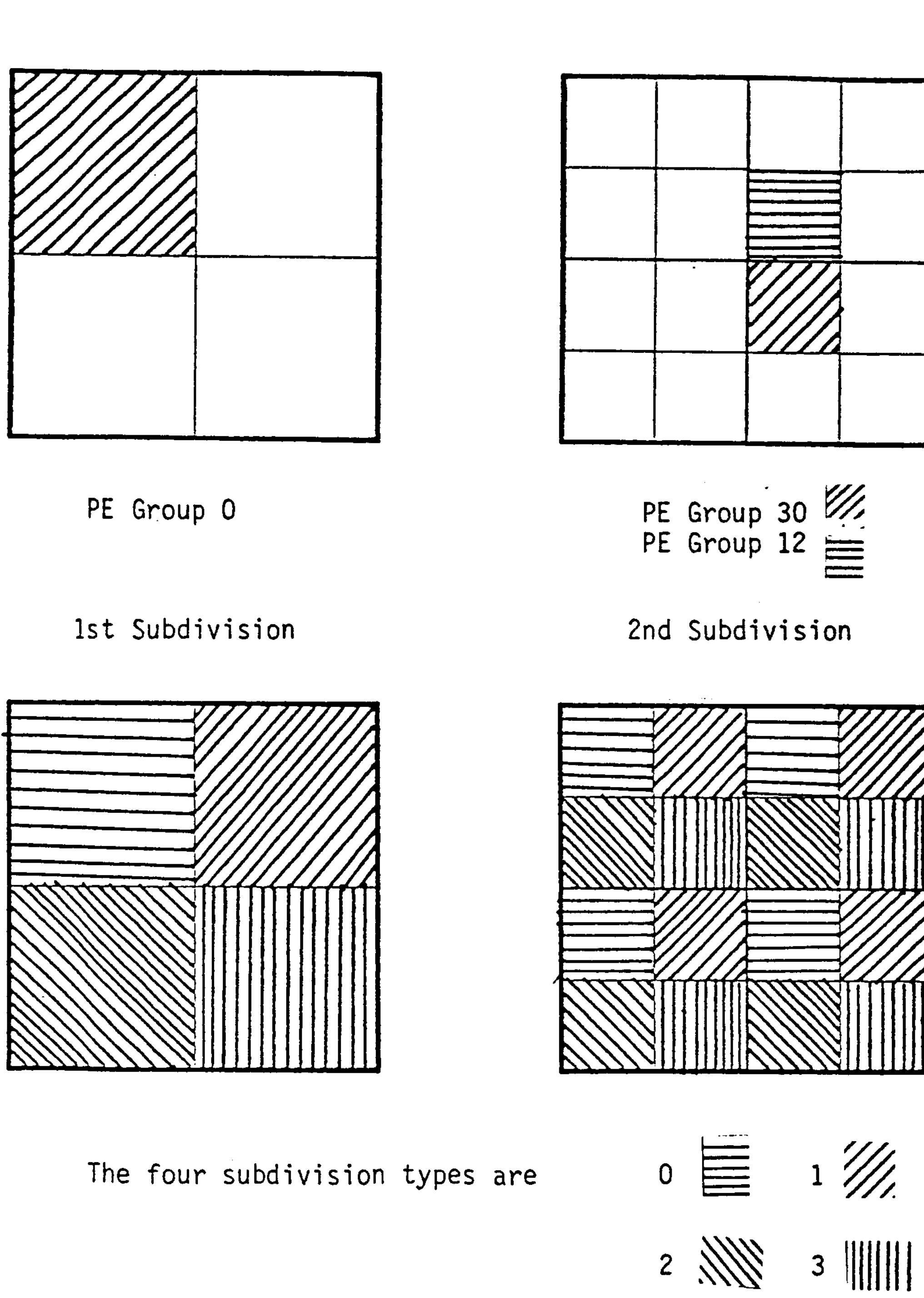

Figure 2: Examples of PE groups and types mapped on to the array. A group corresponds to a child patch. A type corresponds to particular quadrant of the parent patch.