A General Technique for Creating SIMD Algorithms on Parallel Pointer-Based Quadtrees

Thor Bestul
Center for Automation Research
University of Maryland, College Park, MD 20742

Abstract

This paper presents a general technique for creating SIMD parallel algorithms on pointer-based quadtrees. It is useful for creating parallel quadtree algorithms which run in time proportional to the height of the quadtrees involved but which are independent of the number of objects (regions, points, segments, etc.) which the quadtrees represent, as well as the total number of nodes. The technique makes use of a dynamic relationship between processors and the elements of the space domain and object domain being processed.

1 Introduction

A quadtree is a data structure for indexing planar data. It is a tree with internal nodes of degree four, where the root represents a planar rectangular region, and the four sons of each internal node represent the four quadrants of the node's region. Generally, each node stores some information about the region it represents and also a color, with the internal nodes being considered gray and the leaf nodes having some color derived from the data in their regions. A particular variety of quadtree is typically defined by giving a *decomposition rule*, which determines whether a region should be subdivided and the corresponding node given sons. For example, for quadtrees to represent binary images (i.e. *region quadtrees*), the rule is that if a region contains pixels with both binary values, it is decomposed and the corresponding node is an internal gray node with four sons. If a region consists entirely of only one the binary values, the corresponding node is a leaf node and has, say, the color black if the single value is 1 and white if it is 0. For our purposes, the division of a region into quadrants is always done uniformly, although the definition of quadtree does not necessitate this.

This paper describes a technique for creating quadtree algorithms intended to run in a parallel processing environment with many processors sharing a single instruction stream (Single Instruction stream Multiple Data stream or SIMD) and possessing a general facility for intercommunication among the processors. The algorithms are for building and processing quadtrees stored with one quadtree node per processor, and with non-leaf node pro-

cessors possessing pointers to the processors representing their sons. We call such a quadtree implementation a parallel pointer-based quadtree.

The target architecture consists of many (several thousand) processors each with a modest amount (a couple Kbytes) of local storage, and all of which simultaneously perform a single sequence of instructions in lockstep. The exception to this is that each processor can ignore instructions depending on the current values in its local memory. Each processor has access to anywhere within the local memory of any of the other processors. Thus if each processor possesses a pointer to data in some other processor's memory, they may all dereference their pointers in lockstep. Simultaneous reads of data from a single location are supported. Simultaneous writes of data to a single location are also supported, as long as a contention resolution operation is specified along with the write operation, such as summing the received values, min or maxing them, performing various boolean operations on them, or selecting one of them arbitrarily.

A fundamental operation used often in the following is that of processor *allocation*, in which a processor obtains a pointer to some other processor not in use and initializes its local memory in some fashion, causing it to then become part of the active computation underway. Processors can also be *de-allocated*, meaning that they no longer contribute to the active computation, and lay idle waiting to be allocated again by an active processor. This can be done in parallel for many processors which desire to allocate other processors by using the *rendezvous* technique [4].

This algorithm creation technique combines two paradigms for parallel computation in the arena of spatial data structures and the objects they represent. One paradigm is space parallelism, in which the two or three-dimensional space represented by our data structure (in this case a quadtree) is divided up among the processors, each of which operates serially across the entire set of objects. The other paradigm is object parallelism, in which the set of objects involved is divided up among the processors, each of which operates serially across the entire space. The technique described here uses parallelism across both space and the set of objects. In order to accomplish this combination of paradigms the technique leans heavily on the facility of general intercommunication among processors, and in particular on the capability of the handling of multiple reads and writes.

The technique succeeds partially because of its use of a very fine-grained parallelism in which we have parallel processors distributed both across the spatial elements and across the objects in the object set. However, the technique only attains its full generality when we discover a mechanism to press beyond even this level of granularity when necessary, and to make use of a a dynamic relationship between processors and the elements of the space and object domains being processed.

2 A Degenerate Case

It is simplest to describe the algorithm creation technique by first describing a degenerate case of it. We use as an example problem the task of constructing a PR quadtree for a collection of points in a plane, and create a parallel algorithm for this task. In the construction of a PR quadtree, a node should

be assigned the color gray and subdivided if more than one point lies within its boundary. A node should be white if it contains no points and black if it contains exactly one.

We allow a processor for each point, containing that point's coordinates, and initially have one processor to represent the root of the quadtree under construction. The algorithm consists of a single loop iterating from the top to bottom level of the quadtree being built, constructing all the quadtree nodes for each particular level at once. Each point processor contains a pointer to a node processor, and initially all the point processors point to the single root node processor. The algorithm is illustrated in Figures 1 through 4 for a small set of points.

For the first iteration of the loop, all points retrieve from the root node information about where the boundary of the region it represents lies, and then compute whether or not they lie within that boundary. All points outside of the root boundaries set their node pointers to null. All points with non-null node pointers then send the value "1" to the root node. These values are summed at the root node as they are received. The root node processor then checks this sum, and if it is greater than 1, meaning that more than one point lies within its boundary, it assigns itself the color gray, and allocates processors for its four son nodes. If the sum is 0 then the node assigns itself the color white, and black if the sum is 1. Each point then checks the node it points to (still the root node during this first iteration), and if it is gray, computes which quadrant of the node it is in, and fetches the corresponding son pointer. Each point processor now possesses a pointer to the node processor corresponding to the quadrant within which the point lies, if it lies in any, or has a null node pointer otherwise.

On the second iteration of the loop each point again sends the value "1" to the node to which it points, the nodes check the sums they receive, and all those nodes found to have more than one point within their boundaries are set to gray and allocate sons. Each point processor then selects the appropriate son to point to. This process is repeated moving down the quadtree being created until no node has more than one point within its boundary, or some limit on the number of quadtree levels has been reached.

Below is the PR-quadtree construction algorithm. The main procedure is 'PR_quadtree()'. This procedure takes as an argument a pointer to a node processor, which it uses as the root of the quadtree constructed. Only those point processors active when the routine is called are used for the construction of the quadtree. This is so that some subset of all the stored points could be selected as a basis for the quadtree, by having the routine called from within a parallel conditional statement. The effect of a conditional statement in a parallel context is to deactivate for the duration of the statement those processors whose currently stored values do not satisfy the conditional, as will be discussed below.

In the procedure 'PR_quadtree', every node is given a flag called *live*. At the beginning of each iteration of the procedure's loop, we set the *live* flag to true in all those nodes which have some point processor pointing to them, and to false in all other nodes. Only those nodes for which *live* is true are operated on during the rest of the iteration.

We give here a description of our programming paradigm and algorithm notation.

The algorithm is given as a procedure, possibly with other supporting procedures. There is no nesting of procedure declarations. Besides procedures there are global variables; a variable is global if it is declared outside of any procedure. A procedure may return a value; if so, the type of the value is given before the 'procedure' keyword.

A variable can be either parallel or non-parallel; in the latter case we say that the variable is mono. Every parallel variable belongs to some particular parallel processor type, such as point processor or node processor. A variable is declared as mono with the construct "mono *declaration*" and as parallel with the construct "parallel *processor-type declaration*". In certain contexts the default type is mono and in certain other contexts the default is parallel; in these contexts the prefixes "mono" or "parallel *processor-type*" may be omitted. Global declarations of variables are mono by default, so when global declarations of parallel variables are made the prefix "parallel *processor-type*" must be used, and furthermore the declarations must be given in a record-style block delimited by the keywords 'begin' and 'end'. Only one such global block of variables is permitted for each processor type. Each processor type has its own namespace for global variables.

If a variable is a pointer, both the pointer itself and the type of object pointed to can be either parallel or mono. The declaration "mono pointer mono integer p" specifies a simple mono pointer to a mono integer, and corresponds to the usual notion of pointers in serial architectures. The declaration "mono pointer parallel apple integer p" specifies that 'p' gives a uniform offset into the storage of all processors of type 'apple' and that at that offset is found a parallel variable of type integer. The declaration "parallel apple pointer parallel orange integer p" specifies that the parallel pointer 'p' belongs to the processors of type 'apple', and that each instance of 'p' points to some datum of integer type in some processor of type 'orange'. The declaration "parallel apple pointer mono integer p" specifies that the parallel pointer 'p' belongs to the processors of type 'apple' and that each instance of 'p' points to some mono integer datum.

Procedures can also be either parallel or mono. A procedure is specified as parallel by prefixing its declaration with "parallel *processor-type*", otherwise it is taken to be mono. The arguments, local variables, and return value of a mono procedure are taken to be mono by default. The arguments, local variables, and return value, if any, of a parallel procedure are by default parallel values. Furthermore, within the body of a parallel procedure, parallel global names are interpreted in the context of the processor type given in the procedure declaration. Any of the parameters, local variables, or return values of any procedure can be forced to be of some type other than their default by use the 'mono' and 'parallel' prefixes.

Declarations of procedure parameter types are given between the procedure argument list and the procedure's 'begin' statement.

The construct "in_every *processor-type* do *statement-list*" causes precisely the set of all processors of the given type to become active at the beginning of the statement list. Furthermore, within the statement list the names of global variables are interpreted in the context of the given processor type. At the end of the statement list, the set of processors which were active before the statement list was entered is re-established as the active set.

The are two variants to "in_every". One is "in_every *boolean-expression*

processor-type do *statement-list*" and the other is "in_every *processor-type* having *boolean-expression* do *statement-list*". Both are equivalent to "in_every *processor-type* do if *boolean-expression* then *statement-list*".

Outside of a parallel procedure for a given type or an 'in_every' statement for a given processor type, use of parallel global variable names for that processor's type is considered an error.

In the construct "if *conditional* then *statement-list*", if the conditional is a parallel expression (one involving parallel variables or values), then the subset of the present active set of processors whose current values satisfy the conditional are made the active set at the beginning of the statement list. At the conclusion of the statement list, the set of processors which were originally active is re-established as the active set.

When a parallel procedure is called, the set of processors active at the time of invocation will be the set active at the beginning of the execution of the procedure body.

If 'p' is a pointer to a processor, and 'f' is a parallel variable in the processors of the type of that pointed to by 'p', then the notation 'f<p>' indicates the value of the variable 'f' in the particular processor pointed to by 'p'.

The symbol '<-+' indicates an assignment statement involving a possibly-multiple write, and in which the write contention is to be resolved by summing the multiply-written values. There are other similar assignment symbols such as '<-or' and '<-min'.

The algorithm is as follows:

```
node || pointer node father;
node || pointer node array son[4];
node || integer level;
node || node_color color;
node || real left, right, bottom, top;

point ||  real x, y;


node || procedure allocate_sons();
/* Allocates four sons for each node active when
   the procedure is called and fills each node's
   son array accordingly. */


procedure PR_quadtree(root)
pointer node root;
/* Builds a PR quadtree for all the points which are in
   allocated point processors. Assumes that root points to
   a node whose level and boundary have been initialized,
   and uses this node as the root of the quadtree
   constructed. */
begin
  node || integer total;
  point || pointer node node_ptr;
  point || integer my_quadrant;
  integer l;

  /* Make all points within the root's boundary point to the
```

```
    root, give all other points NULL node pointers. */
  in_every point do
    begin
      node_ptr <- NULL;
      if x >= left<root> and x <= right<root> and
         y >= bottom<root> and y <= top<root> then
        node_ptr <- root;
    end;

  /* Loop from level of root to bottom. */

  for l <- level<root> downto 0 do
  begin
    /* Initialize point total for all nodes on present
       level to zero. */
    in_every node having level = l do
      total <- 0;

    /* Each point contributes 1 to the point total for the
       node containing it. */
    in_every point having node_ptr <> NULL do
      total<node_ptr> <-+ 1;

    /* Nodes with no points in them are white.  Nodes with
       one point in them are black.  Nodes with more than
       one point in them are gray.  If we're not at the bottom
       level we allocate sons for the gray nodes. */
    in_every node having level = l do
      begin
        if total = 0 then color <- WHITE
        else if total = 1 then color <- BLACK
        else begin
          color <- GRAY;
          if l > 0 then allocate_sons();
        end;
      end;

    /* If at bottom level then we're done. */
    if l = 0 then return;

    /* The points in each gray node divide themselves among
       the sons. */
    in_every point having node_ptr <> NULL and
                          color<node_ptr> = GRAY do
      begin
        /* Each point determines which subquadrant it is in. */
        my_quadrant <- 0;
        if x > 0.5 * (left<node_ptr> + right<node_ptr>) then
          my_quadrant <- my_quadrant + 1;
        if y < 0.5 * (bottom<node_ptr> + top<node_ptr>) then
          my_quadrant <- my_quadrant + 2;

        /* Each point fetches the pointer to the corresponding
           node son. */
        node_ptr <- son<node_ptr>[my_quadrant];
      end;

  end;
end;
```

To summarize the technique so far, we allow one processor per object and one processor per quadtree node. Each object is given access to a sequence of shrinking nodes which contain it; initially all objects have access to the root node. By having each object obtain information from its node, and by combining at the node information from all of the objects who access that node, the objects make decisions about descending the quadtree from that node.

3 The General Technique

Now consider instead the task of constructing a PM quadtree for line segment data. In constructing a PM quadtree, a node should be assigned the color gray and subdivided if its boundary contains more than one endpoint, or if its boundary has two segments which enter it but which do not have a common endpoint within it. Initially, we have one processor allocated for the quadtree root, and one processor for each line segment, containing the coordinates of the segment's endpoints.

Consider creating an algorithm, similar to the one given above, to construct the PM quadtree for this segment data. Each segment processor initially possesses a pointer to the quadtree root processor. Each segment processor computes how many of its segment's endpoints lie within the boundary of the node to which the segment processor points; this will be 0, 1, or 2. Each segment then sends this value to the node it points to, and both the maximum and minimum of these values are computed at the node. Any node which receives a maximum value of 2 assigns itself the color gray, since this means that some single segment has both endpoints in the node's boundary. Any node which receives a maximum of 1 and a minimum of 0 also assigns itself the color gray, since this means that there are at least two segments in the node's boundary, one which passes completely through it and one which terminates within it.

Then each segment with exactly 1 endpoint in the node it points to sends the coordinates of that endpoint to the node. The node receives the minimal bounding box of the coordinates sent to it (this, of course, amounts simply to applying min and max operations appropriately to the coordinate components). If this minimal bounding box is larger than a point, the node assigns itself the color gray, since this means that some two segments entering the node have non-coincidental endpoints within the node.

Finally each segment with 0 endpoints in the node it points to determines whether it in fact passes through the interior of the node at all, and if so it sends the value "1" to the node, where these values are summed. If the sum received by the node is greater than 1, the node assigns itself the color gray, since this means that some two segments passing through the node do not have any endpoints in the node, which implies that they do not have a common endpoint in the node. Then all gray nodes allocate son processors. Any nodes which were not given the color gray should be colored white if no segments entered their interior (the sum is zero), and black otherwise (the sum is one).

At this point in the algorithm, we would like to have all segment processors which point to gray nodes compute which of the node's sons they belong

to, and retrieve from the node the appropriate son pointer, just as in the case of the PR quadtree construction algorithm. Of course in this case, as opposed to the case of the point data, a given segment can intersect more than one of the node's sons, and we are left with the situation of wanting to assign up to four son pointers to the segment processor's node pointer, and processing each of the corresponding sons. The solution to this dilemma is to allocate *clones* of each such segment processor, that is, to create multiple processors which represent the same segment, and all of which contain (almost) the same information. So for each segment processor pointing to a gray node, we allocate three clone processors, all of which contain the segment's endpoints and a pointer to the same node as the original segment processor. In addition, the original and its clones each contain a *clone index* from 0 to 3, with the original containing 0 and each of the clones containing a distinct index from 1 to 3. Now the original and its clones each fetch a son pointer from the node that they all point to, each one fetching according to its clone index, so that each gets a different son pointer.

The subsequent iterations of the algorithm proceed as the first, with each segment processor determining how many of its endpoints lie within the interior of the node it points to, and with the eventual computation of the colors of all the nodes on each particular level. At this point in each iteration, notice that any segment processors pointing to leaf nodes, or whose segments do not pass at all through the interior of the node to which they point, will not have any further effect of those nodes, and can thus be de-allocated and re-used later. This reclaiming of segment processors keeps the number of clones allocated for each segment from growing exponentially. In fact the number of processors required for a given segment at a given level in the construction of the quadtree will be only roughly as many as there are nodes in that level of the tree through whose interior the segment passes.

To summarize the general technique then, we allow one processor per quadtree node, and initially allow one processor per object. Each object is given access to a sequence of shrinking nodes which contain part of it; initially all objects have access to the root node. By having each object obtain information from its node, and by combining at the node information from all of the objects who access that node, the objects make decisions about descending the quadtree from that node. For those objects which do descend, it is desirable for their various parts which lie in various quadrants of the node to descend in parallel. Thus we allow duplicate or 'clone' processors for each object, and have each processor handle just that portion of the object relevant to one quadrant of the node. Duplicate processors which determine that they can no longer effect the the node to which they point, because that node is a leaf, or because the object they represent does not overlap that node, can deactivate themselves so that they may be used later in the computations for some other object.

We see then that this technique allows us to go beyond the level of granularity of one processor for every element (space component or object) to a level where there are multiple processors for certain elements and none for others; where the processors are being used and disposed in a dynamic fashion.

4 Other Applications

The same general technique can be applied to create algorithms for several other quadtree tasks. For example consider the task of shifting a quadtree. Suppose we have already created somehow a quadtree with one processor per node, and wish to compute a new quadtree to represent the original one shifted by some amount. Using this technique we create the following algorithm.

Have each black leaf node of the old quadtree compute its own shifted position. Then allocate a new processor for the root node of the new (shifted) quadtree, and give each old black leaf node a pointer to this new root node. Iterate the following from top to bottom of the new quadtree.

Each old black leaf node fetches the boundary of the new node it points to, and computes whether, in its new shifted position, it encloses that node. All old black leaf nodes which do enclose the new node they point to send the value TRUE to the new node, which combines the received results by or-ing them. Any new node which thus determines it is enclosed by some old black leaf node assigns itself the color black. Then each old black node computes whether it intersects the new node it points to even if it doesn't enclose it, and if so sends TRUE to the new node, which combines the received results by or-ing them. The new node then assigns itself the color gray if it is not already black and if some old black leaf node intersects it, i.e. if the received result is TRUE. Any new node which does not determine itself to be black or gray in this way assigns itself the color white. All new gray nodes allocate sons for themselves. Each old black leaf node pointing to a new gray node allocates clones for itself, and divides up among itself and its clones the son pointers of the new gray node to which they all point.

In the above procedure, before clones are allocated, any processor representing an old black leaf node which points to a black or white new node should de-allocate itself so that it may be re-used, since it will no longer affect the new node it points to. Of course, this de-allocation should not be done for those processors which originally represented the quadtree to be shifted, if it is desired that this original quadtree not be lost, but these processors can be specially marked to avoid their being de-allocated.

It is not hard to see how this same technique can also be used to create algorithms for quadtree rotation and expansion which run in time proportional to the height of the new quadtree, by computing in parallel the rotated or expanded version of each old black leaf node, and building the new quadtree using cloning. One can also create algorithms for the simultaneous insertion of many polygons or arbitrary regions into a quadtree. Some of these algorithms will require an additional post-processing phase in which any node with four sons of the same color is given that color and has its sons discarded. This can be done in a single bottom-up pass over the new quadtree in time proportional to its height.

5 A Hidden Edge Algorithm Using Cloning

To show the flexibility of our technique, we use it here to create an algorithm for computing hidden edges in a scene consisting of polygons lodged in 3-

space. The algorithm builds an MX quadtree of the pixels in the viewplane. In an MX quadtree [3], all pixel sized leaf nodes through which an edge passes are black, and all other leaf nodes are white.

This algorithm is based on the Warnock algorithm for hidden edge computation [1] [5]. The essential idea of the algorithm is that while recursively decomposing the viewplane into quadrants, if it can be determined that all of the pixels which compose some entire quadrant at some level of decomposition should be white, then the quadrant does not need to be further decomposed. In order to determine if this is so for a given quadrant, we consider the planes (in 3-space) in which our polygons lie. After computing the projections onto the viewplane of all polygons (which is done in parallel by the polygon processors), we consider the planes of those polygons whose projections completely enclose the given quadrant. We wish to determine if the plane of any of those polygons is "closer" to the viewpoint than the planes of the other polygons whose projections enclose the quadrant. To determine this, we compute the inverse projections of the quadrant corners onto the planes of the enclosing polygons, and if one plane is found to be nearer to the viewpoint for all four corners, it is deemed the closest plane.

The algorithm proceeds as follows. We initially assign one processor per polygon, and have one processor representing the root node of the viewplane quadtree being constructed. Initially each polygon processor possesses a pointer to the root node. The following procedure is iterated from top to bottom of the quadtree being built.

Each polygon computes its projection onto the viewplane (these can be pre-computed since they are fixed), and determines the relationship of its projection with the quadtree node to which it points. Specifically, it determines whether its projection encloses the quadrant, or is involved with it, meaning it overlaps but does not enclose the quadrant, or whether it is outside the quadrant altogether.

Each polygon whose projection encloses its quadrant computes the inverse projection of each of the four corners of its quadrant onto its plane. This computation produces for each corner a distance from the viewpoint to the polygon's plane. Each of these polygons then sends this distance for each of the four corners to its quadrant (node) processor, which computes the minimum of these values as they are received. Each polygon then reads back the minimum distance for each of the four corners, and if all four minimum distances are equal to the corresponding distances which the polygon computed for its own plane, the polygon concludes that its plane is closest to the viewpoint. The polygon then informs its quadrant that it is enclosed by the projection of a polygon whose plane is closest to the viewpoint, and based on this the quadrant assigns itself the color white.

Then all polygons which are involved with (i.e. overlapping but not enclosing) their quadrant send the value TRUE to their quadrant, which combines the values sent to it by or-ing them. Any quadrant not already assigned the color white and which determines it has some polygon involved with it assigns itself the color gray. All other quadrants have no polygons whose projections either enclose them or are involved with them, so they assign themselves the color white. All gray quadrants allocate sons.

Those polygons which point to a quadtree leaf node, or which are outside the quadrant to which they point, de-allocate themselves, since they will no

longer affect those nodes. All remaining polygon processors point to a gray nodes. Each remaining polygon processor allocates clones, and divides up among itself and its clones the son pointers of its node.

On the last iteration of the algorithm, that is, the pixel-level iteration, the procedure above is modified so that any node which is involved with some polygon assigns itself the color black instead of gray. After this last iteration, the quadtree constructed is an MX quadtree representation of the viewplane of the projection, with hidden edges eliminated.

Below is the hidden-edge algorithm. The main procedure is 'hidden_edge()', which takes as an argument a pointer to a node processor, and uses this as the root of the quadtree constructed. As with 'PR_quadtree()', only those polygon processors active when the routine is called are used for the construction of the hidden-edge image quadtree.

```
node || pointer node father;
node || pointer node array son[4];
node || integer level;
node || node_color color;
node || real left, right, bottom, top;

/* Vertex projections onto viewplane. */
polygon || real array x[NPOINTS], y[NPOINTS];
/* Number of vertices in polygon. */
polygon || int npts;
/* Parameters of polygon plane. */
polygon || real a, b, c;

polygon || real polygon || procedure poly_plane_dist(x, y);
polygon || real x, y;
/* For each active polygon, returns the distance from the
   viewpoint to the polygon plane via the point (x, y) on
   the viewplane. */

polygon || procedure allocate_clones();
/* Allocates four clones for each active polygon.  The
   clones get the clone indices 0, 1, 2, and 3. */

polygon || procedure deallocate_clones();
/* Deallocate all active clones. */

node || procedure allocate_sons();
/* Allocates four sons for each active node. */

polygon || relation
  polygon || procedure find_relation(left, right, bottom, top);
polygon || real left, right, bottom, top;
/* Each active polygon determines the relationship (INVOLVED,
   OUTSIDE, ENCLOSES) of its projection with the rectangle
   defined by the parameters passed. */

procedure hidden_edges(root)
value pointer node root;
/* Builds a parallel quadtree to represent the scene of
   all the polygons.  Performs hidden edge elimination
   based on a projection using the plane of the quadtree
   leaves as viewplane.  The pointer passed is assumed to
   point to a quadtree node whose level and boundaries
```

```
   have been initialized and is used as the
   root of the quadtree constructed. */
begin
  polygon || relation rel;
  polygon || real pleft, pright, pbottom, ptop;
  polygon || pointer node node_ptr;
  polygon || real pll, plr, pul, pur;
  polygon || integer clone_index;
  integer l;

  /* Start off with all polygon clones pointing to the root. */
  in_every polygon do
    node_ptr <- root;

  /* Loop from level of root node to bottom. */

  for l <- level<root> downto 0 do
  begin
    /* Each polygon fetches the boundaries of the node it
       points to and determines its relationship with it. */
    in_every polygon do
      begin
        pleft <- left<node_ptr>;
        pright <- right<node_ptr>;
        pbottom <- bottom<node_ptr>;
        ptop <- top<node_ptr>;
        rel <- find_relation(pleft, pright, pbottom, ptop);
      end;

    /* Each node on the current level initializes the minimum
       distance for its four corners to be infinity. */
    in_every node having level = l do
      begin
        ll <- INFINITY;
        lr <- INFINITY;
        ul <- INFINITY;
        ur <- INFINITY;
      end;

    /* Every polygon processor whose projection is not outside
       its node determines the distance from the viewpoint to
       the polygon's plane for each of the four corners of the
       node.  For each of the four corners, the minimum
       distance, computed over the set of planes of all such
       polygons, is accumulated at the node processors. */
    in_every polygon having (rel <> OUTSIDE) do
      begin
        pul <- poly_plane_dist(pleft, ptop);
        pur <- poly_plane_dist(pright, ptop);
        pll <- poly_plane_dist(pleft, pbottom);
        plr <- poly_plane_dist(pright, pbottom);

        ul<node_ptr> <-min pul;
        ur<node_ptr> <-min pur;
        ll<node_ptr> <-min pll;
        lr<node_ptr> <-min plr;
      end;

    /* Each node on the current level initializes to FALSE
       a flag which indicates that it is enclosed by the
```

```
   projection of the closest polygon, and to TRUE a flag
   which indicates that the projections of all polygons
   are outside it. */
in_every node having level = l do
  begin
    enclosed_by_closest <- FALSE;
    all_outside <- TRUE;
  end;

/* Each polygon whose projection encloses its node
   determines if its plane is closest (among the planes of
   all such polygons) at all four corners of the node.
   The disjunction of these results is accumulated at the
   node processors. */
in_every polygon having (rel = ENCLOSES and
                         pul = ul<node_ptr> and
                         pur = ur<node_ptr> and
                         pll = ll<node_ptr> and
                         plr = lr<node_ptr>) do
    enclosed_by_closest<node_ptr> <-or TRUE;

/* Each polygon knows if it is outside the node it
   points to.  The conjunction of these results is
   accumulated at the node processors. */
in_every polygon having (rel <> OUTSIDE) do
    all_outside<node_ptr> <-and FALSE;

/* Finally we determine the color for each node on the
   current level. */
in_every node having level = l do
  begin
    if enclosed_by_closest or all_outside then
      color <- WHITE;
    else begin
      if l = 0 then color <- BLACK;
      else color <- GRAY;
    end;
  end;

/* Each polygon clone pointing to a black or white node,
   or which is outside of the node it points to,
   is de-allocated. */
in_every polygon having (color<node_ptr> = WHITE or
                                 color<node_ptr> = BLACK or
                                 rel = OUTSIDE) do
  begin
    deallocate_clones();
  end;

/* If at the bottom level then we're done. */
if l = 0 then return;

/* Each gray node on the current level allocates sons. */
in_every node having level = l and color = GRAY do
  allocate_sons();

/* Each remaining polygon allocates four clones and
   tags itself as an old clone. */
in_every polygon do
begin
```

```
      allocate_clones();
      in_every polygon do new_clone <- TRUE;
      new_clone <- FALSE;
    end;

    /* Then the new polygon processors each get a pointer
       to one of the node's sons, and the old polygon
       processors are deallocated. */
    in_every polygon do
      if new_clone then
        node_ptr <- son<node_ptr>[clone_index];
      else
        deallocate_clones();

  end;
end;
```

6 Some Timing Results

In this section we present some timing results for the PR quadtree building algorithm and the hidden edge algorithm for implementations of these algorithms on a Connection Machine. A Connection Machine is a SIMD architecture based on a multi-dimensional cube. The vertices of the cube correspond to processors, and the edges correspond to direct communication links between the processors. The illusion of direct access from one processor to the memory of any other is supported by a sophisticated routing algorithm, which deals with bottlenecks and which also supports simultaneous read access and simultaneous write access using several contention resolution operations. Due to the nature of the contention resolution mechanism, the amount of time required to perform a simultaneous write to or read from a single location tends to be proportional to the log of the number of processors performing the simultaneous access. The Connection Machine also support virtual processors, meaning that each processor can emulate several processors, with a proportional reduction in processing speed and memory per processor. The mechanism of virtual processors in transparent to the code which runs on the Connection Machine.

The algorithms were implemented in C*, a parallel version of C, using floating point for all geometric coordinates and were run on a 16384 processor CM-2 without floating point hardware. For each algorithm and number of objects processed, two times are given. One is the real elapsed time, and one is the amount of time spent actually performing operations on the Connection Machine. The tables reveal that the running times of the algorithms on a Connection Machine are not in fact completely independent of the number of objects represented, which was expected since the execution of multiple reads and writes takes time proportional to the log of the number of processors involved in the simultaneous access. This fact, together with the fact that such intercommunication operations tend to be the most time consuming operations on a Connection Machine, explains the approximate log dependency seen in the tables of the algorithm running times on the number of objects represented.

Table 1 shows timing results for the PR quadtree building algorithm for various numbers of points distributed randomly over a square region, for a

quadtree with a maximum depth of eight levels.

Number of Points	Elapsed Time (s)	CM Time (s)
10	0.88	0.61
100	2.15	1.78
1000	4.09	3.25
10000	6.98	6.02

Table 1:

Table 2 shows timing results for the hidden edge algorithm for various numbers of square polygons distributed randomly over a parallelpiped region with a pre-computed parallel projection onto a viewplane parallel to one of the faces of the parallelpiped. The MX quadtree constructed has a maximum depth of eight levels, i.e. it is the MX quadtree for a 128 by 128 pixel image.

Number of Polygons	Elapsed Time (s)	CM Time (s)
5	9.49	8.57
50	12.64	11.24
500	18.79	15.49

Table 2:

7 Summary

This paper has presented a technique for creating SIMD algorithms for parallel pointer-based quadtrees. It combines parallelism both across the elements of the space represented by the quadtree and across the elements of the set of objects represented. It produces algorithms wherein a dynamic relationship is maintained between elements and processors, with elements having perhaps several processors operating on them simultaneously, and with elements disposing of their processors when they are no longer required, so that they may be re-used by other elements.

8 Future Plans

We will continue to apply this technique in the construction of parallel algorithms for a variety of quadtree tasks. In addition, we point out that we presented this technique as an embodiment of a control mechanism which can exploit fine-grained parallelism to create a useful dynamicism between processors and elements of our processing domain. In the future we plan to expand on the notion of this sort of dynamicism and apply it to other data structures and problem domains.

References

[1] John E. Warnock, "A Hidden Line Algorithm for Halftone Picture Representation", Technical Report 4-5, Computer Science Department, University of Utah, Salt Lake, May 1968

[2] Hanan Samet and Robert E. Webber, "Hierarchical Data Structures and Algorithms for Computer Graphics", *IEEE Computer Graphics and Applications*, May 1988

[3] Gregory Hunter and Kenneth Steiglitz, *Operations on Images Using Quad Trees*, IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. PAMI-1, No. 2, April 1979

[4] W. Daniel Hillis, *The Connection Machine*, M.I.T. Press, Cambridge, MA, 1985, Section 6.3

[5] J.D. Foley and A. Van Dam, *Fundamentals of Interactive Computer Graphics*, pp. 565-568, Addison-Wesley, Reading, MA, 1982

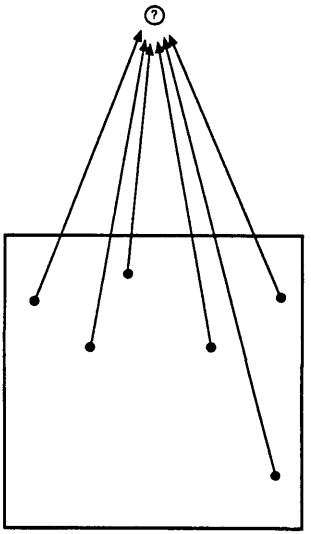

Figure 1: Before the first iteration

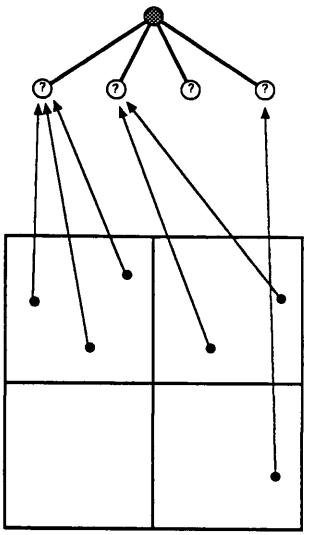

Figure 2: After the first iteration

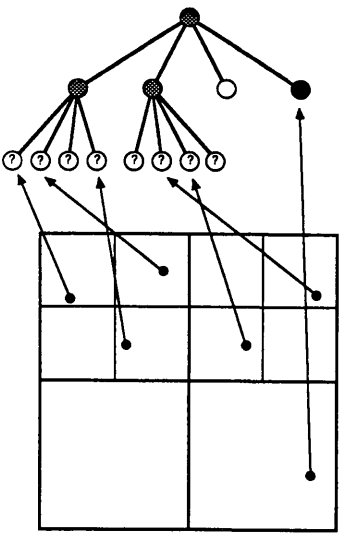

FIgure 3: After the second iteration

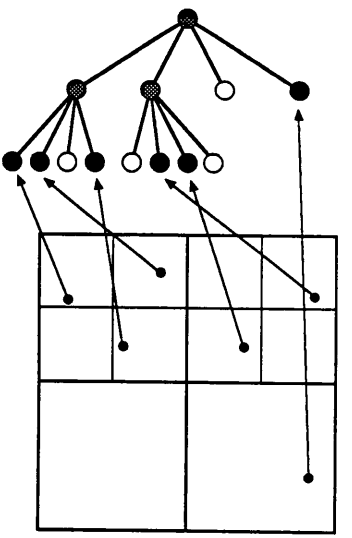

Figure 4: After the final iteration