International Journal of Parallel Programming, Vol. 19, No. 5, 1990

Parallel Algorithms for Line Generation

Rok Sosič and Richard F. Riesenfeld[1]

Received April 1990; Revised October 1990

A new, parallel approach for generating Bresenham-type lines is developed. Coordinate pairs which approximate straight lines on a square grid are derived from line equations. These pairs serve as a basis for the development of four new parallel algorithms. One of the algorithms uses the fact that straight line generation is equivalent to a vector prefix sums calculation. The algorithms execute on a binary tree of processors. Each node in the tree performs a simple calculation that involves only additions and shifts. All four algorithms have time complexity $O(\log_2 n)$ where n in the form 2^m denotes the number of points generated and $n-1$ is the number of processors in the tree. This compares to $O(n)$ for Bresenham's algorithm executed on a sequential processor. Pipelining can be used to achieve a constant time per line generation as long as line length is less than n.

KEY WORDS: Line generation; parallel algorithms; binary processor tree; middle cut algorithm; prefix sums; binary summation.

1. INTRODUCTION

Line generation is a fundamental geometric calculation that appears in many areas: computer graphics, image processing, scientific visualization, medical imaging, computer aided design, and computer aided manufacturing. Since its introduction, Bresenham's algorithm,[1,2] with many subsequent variations and optimizations, has become a standard approach for generating straight lines. A recent survey of the field was compiled by Brons.[3] The original Bresenham view of line generation as embodied in his algorithm, which is essentially one involving two coordinated counters and a cumulative error, may lead to thinking of line generation as a fundamentally serial

[1] Department of Computer Science, University of Utah, Salt Lake City, UT 84112. E-mail: sosic@cs.utah.edu, rfr@cs.utah.edu.

828/19/5-3

0885-7458/90/1000-0389$06.00/0 © 1990 Plenum Publishing Corporation

process, and explains why not many attempts were made to parallelize it. There have been, however, some work done on parallel methods for line generation.

Sproull[4] presents the (n, n) algorithm where n processors operate with a phase shift and each processor generates points spaced n units apart. The first processor generates points at $x = 0$, n, $2n$,..., the second processor generates points at $x = 1$, $n + 1$, $2n + 1$,..., and so on.

Wright[5] describes an algorithm where a line is divided into subintervals and each processor generates one subinterval.

The algorithm used in Pixel-Planes[6] simulates a straight line as a long stretched polygon. This is the only algorithm, mentioned in this paper, that is not a variation of Bresenham's approach.

Blelloch[7] uses a scan operation, called in this paper *prefix sums* operation, as a basis for generating straight lines. His approach is probably similar to our prefix sums algorithm, but his description is not sufficiently detailed to make a precise comparison.

Our paper is organized as follows. First, we show how points that approximate a straight line can be represented as elements of an arithmetic sequence and the derivation of such a sequence is explained. Next, four approaches to parallel computation of the points are described.

2. SEQUENCE ALGORITHM

This section derives the relation between points that approximate a straight line and elements of a specialized arithmetic sequence (for alternative derivations see Refs. 1, 3, 4, 8). Numbers are assumed to be integers. Divisions are integer divisions with truncation.

It is assumed without loss of generality that a straight line is drawn from the origin $(0, 0)$ to a point $(\Delta x, \Delta y)$, where $\Delta x \geqslant \Delta y \geqslant 0$, $\Delta x > 0$, and the end points are center aligned. This nontrivial line lies completely in the first octant. An arbitrary straight line can be generated by using translation and symmetry properties.

In more formal terms, the problem is to compute a sequence of 2D points

$$(i, y_i);\ 0 \leqslant i \leqslant \Delta x \tag{1}$$

on a square grid, such that the vertical distance between the points and the line from $(0, 0)$ to $(\Delta x, \Delta y)$ is minimal.

$\Delta x + 1$ points are distributed close to the line so that for every x, $0 \leqslant x \leqslant \Delta x$, there is exactly one 2D point in sequence (1) whose first coordinate is equal to x. The y coordinate to a given x is chosen as a point

on the grid that is vertically closest to the intersection between the theoretical straight line and the vertical line through x.

The line equation is

$$y = \frac{x \Delta y}{\Delta x} \tag{2}$$

The division in Eq. (2) must be done with rounding. Rounding can be obtained from ordinary integer division with truncation after adding $\Delta x \div 2$ to the dividend. $\Delta x \div 2$ can be rounded up or down depending on the application.[9] Equation (2) becomes

$$y = \frac{x \Delta y + \Delta x \div 2}{\Delta x} \tag{3}$$

The general formula for the ith element of a line from (0, 0) to $(\Delta x, \Delta y)$ is

$$(i, (i \Delta y + \Delta x \div 2) \div \Delta x) \tag{4}$$

An algorithm with unit time complexity follows immediately from Eq. (4) where unit time is taken to mean a single evaluation of y_i. If we have $\Delta x + 1$ processors then the ith processor should evaluate the expression

$$y_i = (i \Delta y + \Delta x \div 2) \div \Delta x \tag{5}$$

Although algorithm in Eq. (5) uses constant time, it may not be suitable because integer multiplication and division are used to obtain the y coordinate. In the following sections we develop parallel algorithms with only addition, subtraction, and shifting. The y sequence in Eq. (5) is an arithmetic sequence, a property exploited by the algorithms later. The main difference between the algorithms is in the method they use to compute the arithmetic sequence.

The division by Δx in Eq. (5) can be avoided in the following way. The actual sequence evaluated by the algorithms is

$$(0, \Delta x \div 2), (1, \Delta y + \Delta x \div 2), (2, 2\Delta y + \Delta x \div 2), \ldots, (\Delta x, \Delta x \, \Delta y + \Delta x \div 2) \tag{6}$$

The y coordinate of the sequence elements is kept in the form (p, q), where

$$y = p \, \Delta x + q, \; 0 \leqslant q < \Delta x. \tag{7}$$

This is essentially the y coordinate expressed in base Δx. Then p is the quotient and q is the remainder of the integer division of y by Δx. p corresponds to the y coordinate of the final point, but we also need to keep track of q during computation to obtain the final result. The representation of a sequence element is thus a triplet (x, p, q) instead of a pair (x, y).

The y coordinate to a given x is chosen as a point on the grid that is vertically closest to the ideal straight line. This is also a property of Bresenham's algorithm. It follows that sequence of Eq. (6) represents the same points as those generated by Bresenham's algorithm.

The points on a line are now characterized by a particular arithmetic sequence. Efficient parallel computation of this sequence is our remaining task.

3. FOUR PARALLEL ALGORITHMS

This section describes four new parallel algorithms which are based on the sequence algorithm derived in Eq. (6). The algorithms substitute additions and shifts for divisions or multiplications that are used in the sequence algorithm.

3.1. General Comments

The underlying computer topology for each of our algorithms is a binary tree of processors. Each node in the tree is capable of some simple computation and exchange of data with other nodes. The flow of data proceeds in layers from the root of the tree to leaves. The only exception to this unindirectional flow is the prefix sums algorithm which has an additional phase from the leaves to the root. Each leaf processor can generate two points. The length of the longest computable line is thus the number of leaf processors multiplied by 2.

The depth of the tree is proportional to $O(\log_2 n)$ where n denotes the number of processors. Since the number of execution passes through the tree is at most two, the execution time is also proportional to $O(\log_2 n)$. The cost of the algorithms, the number of processors multiplied by the time complexity, is $(n-1)\log_2 n = O(n \log_2 n)$.

Only one layer of nodes in the tree is used at each step. Nodes that are not part of this layer can be utilized by pipelining. For example, at time t layer k is working on line i, layer $k+1$ on line $i+1$, layer $k+2$ on line $i+2$, and so on. At time $t+1$, layer k is working on line $i+1$, layer $k+1$ on line $i+2$, layer $k+2$ on line $i+3$, and so on. By using pipelining, one straight line can be generated each clock cycle. The cost of the algorithms

becomes $O(n)$, which represents the optimal speedup of the n processor version over the sequential version.

Next, we describe four different algorithms to generate straight lines in detail. During the discussion we illustrate each algorithm for the line with values $\Delta x = 6$ and $\Delta y = 2$.

3.2. Middle Cut Algorithm

Two versions of the middle cut algorithm are presented. The first one is a general algorithm and the second one is an optimized version for a fixed tree of processors where a part of the computation can be omitted.

Probably the most obvious approach to straight line generation in parallel is the *divide-and-conquer* technique: compute the midpoint between two endpoints and repeat the process on both subintervals until the interval size equals one (Fig. 1).

The root node is initialized with the value of the first and last elements in sequence of Eq. (6): $((0, 0, \Delta x \div 2), (\Delta x, \Delta y, \Delta x \div 2))$. The last point on the line is not produced and must be added separately.

The essential part of the computation is the function *get_middle* (see Fig. 2). Since $l = (l_x, l_p, l_q)$ and $r = (r_x, r_p, r_q)$ are both part of an arithmetic sequence, the element $m = (m_x, m_p, m_q)$ between them can be calculated as their component-wise average. For p and q components, relation in Eq. (7) must be considered as well. That means that the value of m_q must be kept in the range $0 \leqslant m_q < \Delta x$ by adjusting m_p and m_q to satisfy relation in Eq. (7). If the number of elements between l and r is even, the element at the position before the center is computed.

In general, the computation will terminate at different levels, which complicates hardware implementation. Moreover, the algorithm is too general. Under the assumption that we are using a binary tree of processors, the algorithm can be simplified considerably.

We can use all processors in every line generation without any time penalty. The line is extrapolated to length MAXL, which denotes twice the number of leaf processors. Thus a line of length MAXL is computed every

```
procedure middle_cut(l, r)
begin
        if distance(l, r) ≤ 1 then draw(l);
        m = get_middle(l, r);
        middle_cut(l, m);  middle_cut(m, r);
end;
```

Fig. 1. The middle cut algorithm.

```
function get_middle((l_x, l_p, l_q), (r_x, r_p, r_q))
begin
                { Δx and Δy are global line generating parameters }
    m_x = (r_x + l_x) div 2;
                { compute the element to the left of the center }
    if even(r_x - l_x) then r_q = r_q - Δy;
    m_p = l_p + r_p; m_q = l_q + r_q;
                { make m_p divisible by 2 }
    if odd(m_p) then begin  m_p = m_p + 1; m_q = m_q - Δx;  end;
    m_p = m_p div 2; m_q = m_q div 2;
                { normalize 0 ≤ m_q < Δx }
    if m_q < 0 then begin  m_p = m_p - 1; m_q = m_q + Δx;  end;
    return((m_x, m_p, m_q));
end;
```

Fig. 2. Function *get_middle.*

time regardless of the original Δx, but only the first $\Delta x + 1$ points at leaves of the tree are taken. Some preprocessing must be added to compute the last point on the extended line. The additional cost of the preprocessing takes only constant time, because it is performed once per line generated.

Since every node computes for a fixed value of x, we can omit all computations related to x. Because line length is fixed to MAXL, we can omit the recursion end test. Because midpoints always lie on the grid, we can omit the parity test for the number of points between endpoints. Similar optimizations are applied in the other parallel algorithms described in this article.

The final algorithm is presented in Fig. 3. Procedure *middle_cut_line1* performs initialization. Procedure *middle_cut1* is calculation that is performed by each computation node of the tree. MAXL denotes twice the number of processors in the bottom row of the processor tree, which is always a power of 2.

An example of the middle cut algorithm with values $\Delta x = 6$ and $\Delta y = 2$ is shown in Fig. 4. Each circle in the figure represents a computing node. The pair of values at each circle are input values to the corresponding node. This pair is denoted as (l_p, l_q) and (r_p, r_q) in function *get_middle1* in Fig. 3. Values at the bottom of the tree represent the (l_p, l_q) part of the pair at this node. The sequence of l_p values gives y values for the line. Since Δx equals six, only first seven points are needed. The resulting line for $\Delta x = 6$ and $\Delta y = 2$ is: (0, 0), (1, 0), (2, 1), (3, 1), (4, 1), (5, 2), (6, 2).

At this point we have fully specified the details of the conceptually simplest algorithm.

3.3. Prefix Sums Algorithm

The prefix sums of a vector $a = (a_1, a_2, ..., a_n)$ constitute a vector p, such that its ith component is the partial sum $p_i = a_1 + a_2 + \cdots + a_i$. The vector of y components of sequence in Eq. (6) is the prefix sums of a vector $(\Delta x \div 2, \Delta y, \Delta y, ..., \Delta y)$ with $\Delta x + 1$ coordinates. Because the x component of sequence in Eq. (6) is uniquely identified by the processor position, there is no need to compute it. Hence, line generation is equivalent to prefix sums calculation, a process whose optimizations and parallel implementations have been studied.[7,10,11]

Prefix sums can be computed on a parallel computer with linear speedup dependent upon the number of processors.[11,12] Although our

```
procedure middle_cut_line(Δx, Δy)
begin
        distribute Δx to all nodes;
        n = MAXL * Δy + Δx div 2;
        middle_cut1((0, Δx div 2), (n div Δx, n mod Δx));
end;

procedure middle_cut1(l, r)
begin
        m = get_middle1(l, r);
        middle_cut1(l, m);  middle_cut1(m, r);
end;

function get_middle1((l_p, l_q), (r_p, r_q))
begin
                        { Δx is a global parameter }
        m_p = l_p + r_p;  m_q = l_q + r_q;
                        { make m_p divisible by 2 }
        if odd(m_p) then begin  m_p = m_p + 1;  m_q = m_q - Δx;  end;
        m_p = m_p div 2;  m_q = m_q div 2;
                        { normalize 0 ≤ m_q < Δx }
        if m_q < 0 then begin  m_p = m_p - 1;  m_q = m_q + Δx;  end;
        return((m_p, m_q));
end;
```

Fig. 3. The optimized version of the middle cut algorithm.

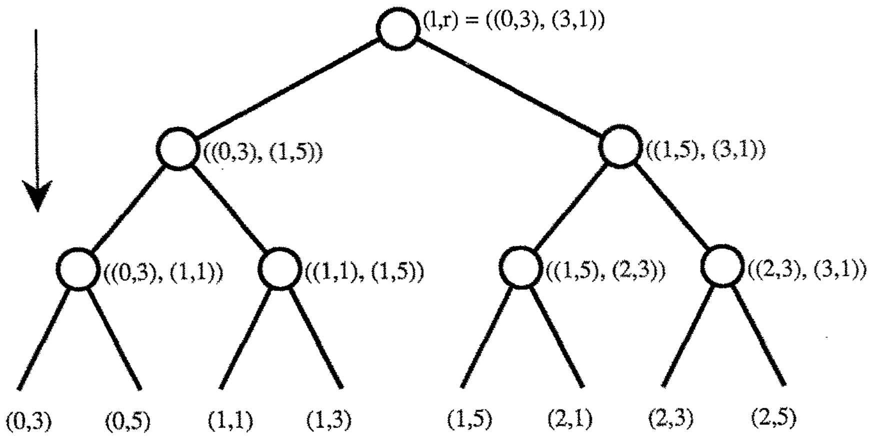

Fig. 4. The middle cut algorithm with $\Delta x = 6$ and $\Delta y = 2$.

algorithm does not achieve this optimal speedup, it leads to a very simple node implementation. We present an independently discovered line generation algorithm based on the prefix sums calculation method by Blelloch.[7] He describes the application of prefix sums operation to line generation, but he does not present details.

Since the prefix sums of a vector can be computed in $O(\log_2 n)$ time, it is clear that line generation has the same time complexity. The algorithm works in two phases. The up phase proceeds from leaves to the root and the down phase from the root to leaves. This algorithm is the only one in this paper where two phases are required. All other algorithms require only the down phase.

The *up phase* computes the sums of subtrees. The sum of a subtree is stored at each corresponding node. The *down phase* propagates the sum of all elements to the left of a given node down the tree using previously computed values (see Fig. 5). Procedure *prefix_sums_line* is the initialization part. Procedure *up_phase* is performed at each node during the up phase. Procedure *middle_phase* is a short computation between the up phase and the down phase that saves the root value and sets its new value to zero. Procedure *down_phase* is performed at each node during the down phase. All numbers are expressed in base Δx, so additions in the algorithm must also be in base Δx (see Fig. 6).

The algorithm computes the prefix sums vector shifted one position to the right to ensure that the computation in the leaves remains the same as that in internal nodes. The last element missing from the vector at the right is the value of the root after the up phase.

Values of nodes in the tree with $\Delta x=6$ and $\Delta y=2$ after up phase and down phase are shown in Figs. 7 and 8, respectively. Final values at the bottom are shifted one place to the right. The root value after the up phase is the value missing on the right, although it is not needed for this particular line. The result of the prefix sums algorithm is the same as that of the middle cut algorithm.

```
procedure prefix_sums_line(Δx, Δy)
begin
        set first leaf to (0, Δx ÷ 2);
                        { test is necessary for the slope of 45 degrees }
        if Δy = Δx then set other leaves to (1,0)
        else set all other leaves to (0, Δy);
        distribute Δx to all nodes;
        up_phase(root);  middle_phase(root) ;  down_phase(root);
end;

procedure up_phase(tree)
begin
        tree.value = up_phase(tree.left) + up_phase(tree.right);
        return(tree.value);
end;

procedure middle_phase(tree)
begin
        save tree.value;  tree.value = 0;
end;

procedure down_phase(tree)
begin
        tree.right.value = tree.value + tree.left.value;
        tree.left.value = tree.value;
        down_phase(tree.left);  down_phase(tree.right);
end;
```

Fig. 5. The prefix sums algorithm.

```
procedure +(a, b)
begin
        c_p = a_p + b_p; c_q = a_q + b_q;
        if c_q ≥ Δx then begin c_p = c_p + 1; c_q = c_q − Δx; end;
        return(c);
end;
```

Fig. 6. Addition in base Δx.

Pipelining of the prefix sums algorithm is more complex to implement than pipelining of other algorithms in this paper because the nodes in the tree must maintain intermediate values for lines that are being processed. Each leaf node must keep $2 \log_2 n$ values, the nodes one layer up must keep $2(\log_2 n - 1)$ values, and so on.

3.4. Binary Summation Algorithms

For hardware implementations, it may be preferable to trade a higher complexity class for simpler operations at each node. The next two algorithms have that goal. They transform multiplication into addition.

If we subtract $\Delta x \div 2$ from the right coordinate of every pair in sequence of Eq. (6), the remaining sequence is

$$(i, i * \Delta y); \quad 0 \leqslant i \leqslant \Delta x \tag{8}$$

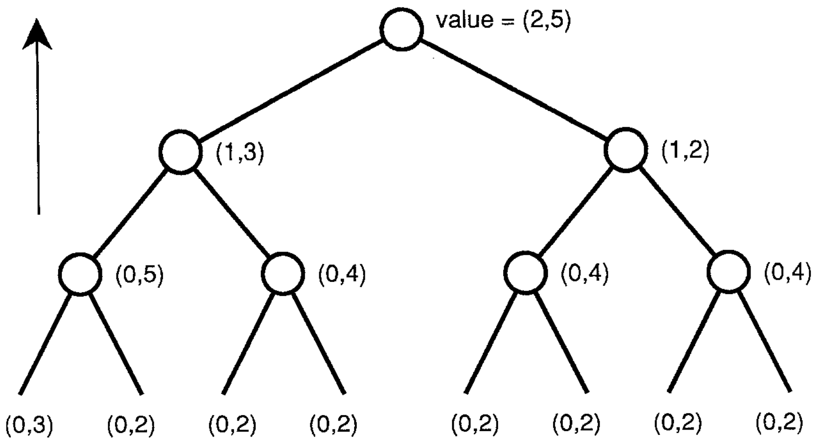

Fig. 7. The prefix sums algorithm after the up phase with $\Delta x = 6$ and $\Delta y = 2$.

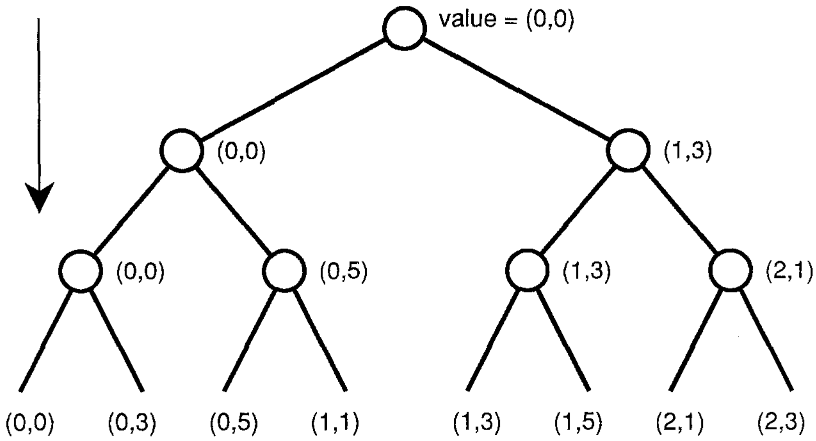

Fig. 8. The prefix sums algorithm after the down phase with $\Delta x = 6$ and $\Delta y = 2$.

Let $(i_n, i_{n-1}, \ldots, i_1, i_0)$ denote the binary expansion of index i. Sequence in Eq. (8) can be computed in two ways.

The first one is the most significant first summation algorithm (MSF):

- reset all elements to zero
- add $2^n \Delta y$ to all elements i, such that $i_n = 1$
- add $2^{n-1} \Delta y$ to all elements i, such that $i_{n-1} = 1$
- ...
- add $2^0 \Delta y$ to all elements i, such that $i_0 = 1$

This type of binary expansion is implemented in Pixel-Planes.[6] However, the MSF algorithm is not a trivial extension of the Pixel-Planes algorithm, because it must maintain numbers in base Δx, while Pixel-Planes works with a single number.

Another way of computing Eq. (8) is the least significant first summation algorithm (LSF), which is similar to the MSF algorithm, only the summation is executed in opposite order:

- reset all elements to zero
- add $2^0 \Delta y$ to all elements i, such that $i_0 = 1$
- add $2^1 \Delta y$ to all elements i, such that $i_1 = 1$
- ...
- add $2^n \Delta y$ to all elements i, such that $i_n = 1$

Both methods convert multiplication into binary summation. They are highly amenable to a binary tree implementation. It is simple to add term

$\Delta x \div 2$ to get sequence in Eq. (6) from sequence in Eq. (8). The root element in the tree is initialized to $\Delta x \div 2$ instead of 0.

The MSF method uses large terms $2^i \Delta y$ that must be added at each node. Terms $2^i \Delta y$ are easily computed by shifting Δy. Ordinarily, they would be expressed in base Δx. Instead of forcing this requirement, we represent terms $2^i \Delta y$ in nodes at height i in base $2^i \Delta x$. They are expressed as a pair (p, q) where

$$p \Delta x 2^i + q; \quad 0 \leqslant q < \Delta x 2^i \tag{9}$$

Since $\Delta x \geqslant \Delta y \geqslant 0$, the value of $2^i \Delta y$ in base $2^i \Delta x$ is simply $(0, 2^i \Delta y)$, if $\Delta x > \Delta y$, or $(2^i, 0)$, if $\Delta x = \Delta y$.

Because computation proceeds from the root to the leaves where the height is 0, the end result is in the required base Δx (see Fig. 9). Procedure *msf_line* represents the initialization of the MSF algorithm, while procedure *msf_sum* is the calculation performed at each node.

An example with $\Delta x = 6$ and $\Delta y = 2$ is shown in Fig. 10. Values at each circle represent inputs to that node. They are denoted as (p, q) in procedure *msf_sum* (Fig. 9). Each branch in the tree shows the value that is added to the node value.

The LSF method is straightforward (see Fig. 11). Procedure *lsf_line* performs the initialization of the LSF algorithm. Procedure *lsf_sum* is the calculation performed at each node in the tree. Since we need values $2^i \Delta y$ (*fp* and *fq* in procedure *lsf_sum*) from the smallest to the largest, they can

```
procedure msf_line(Δx, Δy)
begin
        distribute Δx and Δy to all nodes;
        msf_sum(0, Δx ÷ 2);
end;

procedure msf_sum(p, q)
begin
                        { scale down the range of q, i is height-1 }
        if q ≥ 2^i Δx then begin  p = p + 2^i;  q = q − 2^i Δx;  end;
                        { add term to the right subtree }
        rp = p;  rq = q + 2^i Δy;
                        { scale down the range of rq, i is height-1 }
        if rq ≥ 2^i Δx then begin  rp = rp + 2^i;  rq = rq − 2^i Δx;  end;
        msf_sum(p, q);  msf_sum(rp, rq);
end;
```

Fig. 9. The Most Significant First (MSF) Summation Algorithm.

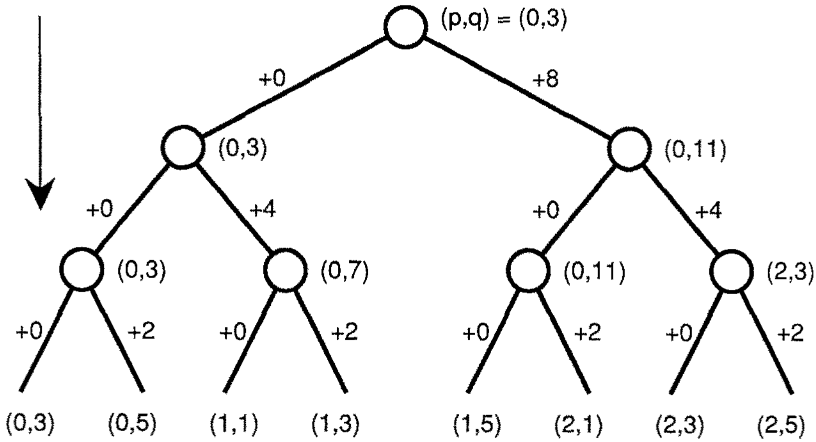

Fig. 10. The Most Significant First Binary Summation with $\Delta x = 6$ and $\Delta y = 2$.

```
procedure lsf_line(Δx, Δy)
begin
        distribute Δx to all nodes;
        lsf_sum(0, Δx ÷ 2, 0, Δy);
end;

procedure lsf_sum(p, q, fp, fq)
begin
                        { add term to the right subtree }
        rp = p + fp; rq = q + fq;
                        { normalize }
        if rq ≥ Δx then begin  rp = rp + 1; rq = rq − Δx;  end;
                        { multiply terms }
        fp = 2 fp; fq = 2 fq;
                        { normalize }
        if fq ≥ Δx then begin  fp = fp + 1; fq = fq − Δx;  end;
        lsf_sum(p, q, fp, fq); lsf_sum(rp, rq, fp, fq);
end;
```

Fig. 11. The Least Significant First (LSF) Summation Algorithm.

be easily maintained in base Δx with only one test after multiplication by 2 at each level. Values *fp* and *fq* can be computed only once per each level of nodes, altough they are shown as a part of the individual node computation.

An example with $\Delta x = 6$ and $\Delta y = 2$ is shown in Fig. 12. Values at each circle represent inputs to that node denoted as (p, q) in procedure *lsf_sum* (Fig. 9). Each branch in the tree shows tha value that is added to the node value. Since the LSF algorithm starts adding the smallest numbers first, the final order of nodes in the result is mixed. The position of a node is determined as the reversed binary expansion of its index. For example, node 6 (110 in the binary representation) contains the y value for x equal to 3 (011 in the binary representation). The proper positions of nodes is shown below the computed values.

The MSF and LSF algorithms are expandable in the following sense. If the straight line length is greater than the number of tree leaves, the result of the computation from the last leaf can be fed to the tree root and the line can be extended as long as the numbers do not overflow the implementational constraints.

3.5. Implementational Issues

All four presented algorithms (middle cut algorithm, prefix sums algorithm, and two binary summation algorithms) have the same time complexity $O(\log_2 n)$ for a tree with $O(n)$ processors. None of the four algorithms is superior over another in time complexity. We discuss some considerations in their implementation.

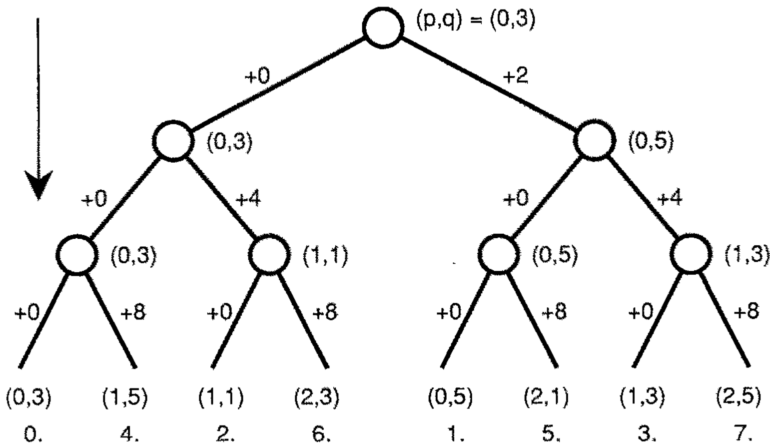

Fig. 12. The Least Significant First Binary Summation with $\Delta x = 6$ and $\Delta y = 2$.

The middle cut algorithm requires multiplication and division during the initialization process. This may become a bottleneck if pipelining is employed. The prefix sums algorithm needs two phases to complete. In addition, each node must maintain some storage for intermediate values, which is not the case with other algorithms. The LSF algorithm has potentially the simplest node calculation, but the results are not produced in correct order. This can cause inefficiency in routing the signals.

The most likely candidate for a practical implementation is the MSB algorithm which has a simple initialization, uses simple computational nodes, and presents its results in correct order. A more detailed comparison of algorithms depends on implementational technology and it is not discussed here.

These algorithms can be extended to generating antialiased lines without fundamental difficulties. Using existing subpixel approaches, antialiased lines can be generated with greater hardware cost but no fundamental change in complexity of the algorithms. This is an additional feature of these line generating methods.

4. CONCLUSION

We have shown that straight line generation is not a fundamentally serial $O(n)$ problem. We have developed algorithms with constant and $\log_2 n$ complexity bounds.

Using only shifts and additions, four parallel algorithms were designed to generate the same lines as those in Bresenham's algorithm. All have time complexity $O(\log_2 n)$, where n denotes the number of points generated, and can be pipelined to achieve unit time line generation. The algorithms can be implemented efficiently with parallel computers in the form of a binary tree.

Conceptually the simplest, the middle cut algorithm is basically a divide-and-conquer algorithm. Each step calculates the midpoint between two endpoints and poses the generation of a line as the generation problem of two shorter lines. The process is repeated until the length of the line equals one.

The prefix sums algorithm uses the fact that the elements of an arithmetic sequence are equivalent to the prefix sums of a certain vector. A variation of the prefix sums algorithm is included and exploited for the purpose of parallel line generation.

The other two algorithms expand multiplication in the arithmetic sequence computation into binary summations. They differ in the order in which summation is performed.

This paper develops algorithms for generating straight lines with a

massively parallel approach in a tree topology. Using pipelining, one line (regardless of its length) can be generated per time unit. It is recognized that these algorithms may generate lines faster than traditional memory subsystems can store them. If there is no need to store lines, it is possible to fully exploit this generation speed. Once memory subsystems of suitable speed are available, these algorithms should provide significant speedup over sequential algorithms.

ACKNOWLEDGMENTS

The authors would like to thank Majna Pleško, Gilad Bracha, Michael Cohen, Kris Sikorski, Elaine Cohen, Jurg Nievergelt, Robin Forrest, Irving Miller, and anonymous referees for their valuable comments on this paper. This work was supported in part by DARPA (DAAK1184K0017 and N00014-88-K-0688). All opinions, findings, conclusions or recommendations expressed in this document are those of the authors and do not necessarily reflect the views of the sponsoring agencies.

REFERENCES

1. J. E. Bresenham, Algorithm for Computer Control of a Digital Plotter, *IBM Systems Journal*, **4**(4):25–30 (1965).
2. J. D. Foley and A. Van Dam, *Fundamentals of Interactive Computer Graphics*, Addison-Wesley, pp. 433–436 (1982).
3. R. Brons, Theoretical and Linguistic Methods for Describing Straight Lines, *Fundamental Algorithms for Computer Graphics*, (ed.), R. A. Earnshaw, NATO ASI Series F, Volume 17, Springer-Verlag, pp. 19–57 (1985).
4. R. F. Sproull, Using Program Transformations to Derive Line-Drawing Algorithms, *ACM Transactions on Graphics*, **1**(4):259–273 (October 1982).
5. W. E. Wright, Parallelization of Bresenham's Line and Circle Algorithms, *IEEE Computer Graphics and Applications*, **10**(9):60–67 (September 1990).
6. H. Fuchs, An Introduction to Pixel-Planes and other VLSI-intensive Graphics Systems, *Theoretical Foundations of Computer Graphics and CAD*, (ed.), R. A. Earnshaw, NATO ASI Series F, Volume 40, Springer-Verlag, pp. 675–688 (1988).
7. G. E. Blelloch, Scans as Primitive Parallel Operations, *IEEE Transactions on Computers*, **38**(11):1526–1538 (November 1989).
8. J. Van Aken and M. Novak, Curve-Drawing Algorithms for Raster Displays, *ACM Transactions on Graphics*, **4**(2):147–169 (April 1985).
9. J. E. Bresenham, Ambiguities in Incremental Line Rastering, *IEEE Computer Graphics and Applications*, **7**(5):31–43 (May 1987).
10. R. E. Ladner and M. J. Fischer, Parallel Prefix Computation, *J. ACM*, **27**(4):831–838 (October 1980).
11. H. Meijer and S. Akl, Optimal Computation of Prefix Sums on a Binary Tree of Processors, *Int'l. JPP*, **16**(2):127–136 (1987).
12. R. Cole and U. Vishkin, Faster Optimal Parallel Prefix Sums and List Ranking, *Information and Control*, **81**(3):334–352 (June 1989).