Parallelization of Bresenham's Line and Circle Algorithms

William E. Wright
Southern Illinois University at Carbondale

I present parallel algorithms for line and circle drawing. Based on Bresenham's line and circle algorithms, my algorithms are applicable on raster scan CRTs, incremental pen plotters, and certain types of printers. I assume that we run the algorithms in a MIMD environment, in which each processor has local data as well as access to shared memory. I also assume that the raster memory is shared. Each processor should set appropriate bits in the raster memory to draw a portion of the line or circle.

Furthermore, I assume the processors are dedicated and assigned to the task (of line or circle drawing). We do not need to "fork" or otherwise activate the processors, but we might "synchronize" them in an appropriate manner to schedule and coordinate their work.

One approach to parallelization is to take several procedures that require independent execution and assign one or more processors to each procedure so that they can be executed simultaneously, or at least with a high degree of overlap. For example, in a graphics system we can assign one processor to do transformations, another clipping, and another line drawing.

Another approach to parallelization is to take a single procedure that requires execution for several different sets of arguments and assign multiple processors to execute the procedure simultaneously (over multiple sets of arguments). This is somewhat analogous to a single instruction, multiple data model of parallel computing.[1] For example, we can assign multiple processors to the line-drawing procedure in a graphics system, with different processors assigned different lines to draw.

I use a different approach in this article. I take a basic procedure such as line or circle drawing and implement it using parallel processors and a parallel algorithm designed for that procedure. This is the most elementary level of parallelization, and we can use it with either or both of the other approaches. Moreover, while the other two approaches are relatively straightforward, the algorithms for this approach require some development.

My algorithms assume that each pixel position has a distinct memory address. If this is true, then synchronization should require very little time, because I designed the algorithms to avoid *address contention,* in

0272-17-16/90/0900-0060$01.00 ©1990 IEEE

which two processors attempt to access the same pixel position simultaneously. If the assumption is not true, then I'll have to modify the algorithms to eliminate the possibility of contention or to provide proper coordination. This article does not include such modifications.

Robert Sproull presented a different parallel technique for line drawing.[2] Pike discussed a different technique for "starting a line in the middle,"[3] which is one of the features of the line algorithm I present. I also present the parallel line algorithm here for the sake of completeness and to serve as an introduction to the circle algorithm.

The line algorithm

Let's suppose that we need to determine the raster representation of the straight line between two given endpoints (x_0, y_0) and (x_f, y_f) where x_0, y_0, x_f, and y_f are integer coordinates. Let $dx = x_f - x_0$ and $dy = y_f - y_0$. Then the slope m of the line is given by $m = dy/dx$ (if $dx \neq 0$). I will suppose without loss of generality that $dx \geq dy \geq 0$, so that $0 \leq m \leq 1$. It is convenient to design an algorithm for this case and then apply straightforward extensions or modifications for other cases ($m > 1$ or $m < 0$).

Let $n = x_f - x_0$ and let $x_i = x_0 + i$ for $i = 1, \cdots, n$. The problem for the raster algorithm is to determine integer values $y_1, y_2, \cdots, y_n = y_f$ such that, for $1 \leq i \leq n$, the point (x_i, y_i) is closer to (or as close as) the true line than any other point with x coordinate x_i and an integer y coordinate.

Clearly, y_i can be chosen as Round($f(x_i)$) where $y = f(x)$ is the equation of the line. (For precision, let's assume that Round($a + 0.5$) = $a + 1$, for any nonnegative integer a.) If $dx \neq 0$, then an equation for the line is $y = mx + b$, where $b = y_0 - m(x_0)$. Hence, we get y_i = Round($m(x_i) + b$) = y_0 + Round(mi).

The sequential line algorithm

Bresenham designed an efficient sequential line algorithm,[4] described in a number of sources (such as the work by Foley and Van Dam[5]). I refer primarily to the algorithm as Hearn and Baker presented it.[6] Bresenham's line algorithm determines the values for $y_1, y_2, \cdots, y_n$ without using floating-point arithmetic.

The algorithm does not initially require that $x_f \geq x_0$, but it swaps the endpoints if not. It does assume that after the swapping (if needed), $y_f \geq y_0$, so that in either case the slope of the line lies between 0 and 1. This swapping will cause problems with the implementation of line style attributes. Hence, in some systems it might be preferable to eliminate the swapping and insert appropriate logic to always draw the line from (x_0, y_0) to (x_f, y_f). I limit my discussion to the algorithm and its parallelization as stated here, but the parallelization of the other version is very similar.

Basically, the algorithm moves the x coordinate step by step from left to right along the line. At each step, we choose the y coordinate to be the same as or one larger than the previous y coordinate. Which choice to make for the y coordinate depends on whether the current value of the integer variable p is negative or nonnegative. We also update the value of p at each step, again depending on whether its current value is negative or nonnegative.

Dividing the work and getting started

The parallel version given here for Bresenham's line algorithm divides the interval from x_0 to x_f into P approximately equal subintervals, where P is the number of processors available. In effect, each processor determines the raster representation for the line segment for the x coordinates lying in the subinterval assigned to it. Pixel addresses determined by different processors do not overlap.

The main trick for each processor (except the first) is to get started, that is, to determine the proper initial values of y and p. Each processor will in effect "jump into the middle" of the sequence of calculations normally performed by the sequential algorithm in determining y and p.

Let's start by defining w as the ceiling of (dx/P). Then w is the width of the subinterval assigned to processor k, except that the last processor (and possibly other processors as well) may have a smaller (or even 0) width to accommodate truncation errors. Letting Div denote integer division, we get

$$w = \text{Ceiling}(dx/P) = (dx + (P - 1))\text{Div}P \qquad (1)$$

Now assume that the processor number is k, where $0 \leq k \leq (P - 1)$. Let $x(k)$, $y(k)$, and $p(k)$, respectively, denote the proper initial values for variables x, y, and p for processor k. To be precise, $y_0 + y(k)$ is the y coordinate of the pixel whose x coordinate is $x_0 + x(k)$, and $p(k)$ is the value that p would have in the sequential algorithm exactly when the pixel $(x_0 + x(k), y_0 + y(k))$ is set. I will define $x(k)$ by

$$x(k) = kw \qquad (2)$$

We can now see from the sequential algorithm that $y(k) = \text{Round}(m \cdot x(k))$ and $p(k) = 2dy - dx + 2\,dy(x(k)) - 2dx(y(k))$.

We can eliminate floating-point computations and otherwise reduce the computations required by using the following derivations:

$$t(k) = dy \cdot x(k) \tag{3}$$

$$\begin{aligned} y(k) &= \text{Round}(m \cdot x(k)) \\ &= \text{Truncate}(m \cdot x(k) + 0.5) \\ &= \text{Truncate}\left(\frac{(2 \cdot dy \cdot x(k) + dx)}{(2 \cdot dx)}\right) \\ &= (2 \cdot t(k) + dx)\text{Div}(2 \cdot dx) \end{aligned} \tag{4}$$

$$p(k) = 2 \cdot dy - dx + 2 \cdot t(k) - 2 \cdot dx \cdot y(k) \tag{5}$$

We thus have the basis for an algorithm requiring three integer multiplications and one or two integer divisions. (I exclude multiplication or division by a power of two, which can be implemented as a left or right shift.) The multiplications are $k \cdot w$ in Equation 2, $dy \cdot x(k)$ in Equation 3, and $dx \cdot y(k)$ in Equation 5. Integer division is required in Equation 4 and also in Equation 1 if P is not a power of two.

The parallel line algorithm

Figure 1 shows the parallel line algorithm in Pascal. In this case, the formal parameter k is the processor number and P_Minus_1 is a constant equal to $P - 1$.

Basically, the unnumbered lines in the algorithm are the same as in the sequential algorithm. Line 1 declares new variables, and Lines 2 through 13 constitute new executable statements. (Actually, Lines 4 and 13 replace statements in the sequential algorithm.) Note that all of the additional computation falls outside the While loop.

Analysis of the line algorithm

The time for executing the sequential algorithm on a particular line segment is given by $t_s(dx, dy) = A + C_1(1 - m)dx + C_2 \cdot m \cdot dx$. A is the time component corresponding to the setup statements preceding the While loop, C_1 is the time required for one iteration of the While loop when y is not incremented, and C_2 is the time for one iteration when y is incremented. Since C_1 and C_2 are relatively close, it is reasonable to use the simpler approximation given by $t_s(dx) = A + C \cdot \text{dx}$.

The time for executing the parallel algorithm on a particular line segment is given similarly by $t_p(P,dx) = B + C\,(\text{Ceiling}(dx/P)) \approx B + C\,(dx/P)$. P is the number of processors, B is the time component corresponding to the setup statements preceding the While loop, and C is the same as for the sequential algorithm. Clearly, then, the parallel algorithm has a larger constant setup cost, and the loop time, which is proportional to dx, is reduced by a factor of P.

The speedup $S(P, dx) = t_s(dx)/t_p(P, dx)$ of the parallel algorithm approaches the perfect value of P as dx gets larger.

Of course the values of A, B, and C depend on the implementation, but I made a rough analysis by simulating an implementation in Turbo Pascal on an Intel 8088-based personal computer. I don't intend the results to demonstrate the precise performance of the two algorithms, but rather to show their relative performance in general terms. Other implementations will probably yield slight differences, but the basic pattern will be the same.

The results are as follows, using an imaginary time unit in which the time through one iteration of the While loop is 1:

$$C = 1$$
$$A = 3.74$$
$$B = 6.54 = 1.75A = A + 2.80$$

The setup cost for the sequential algorithm is the equivalent (in time) of about 4 loop iterations, and the setup for the parallel algorithm requires the equivalent of about 3 loop iterations more.

The timing formulas now become:

$$t_s(dx) = 3.74 + dx$$
$$t_p(P, dx) = 6.54 + {}^{dx}\!/_{P}$$

For example, when we set dx to 100, we get $t_s(100) = 104$, $t_p(4, 100) = 32$, and $S(4, 100) = 3.3$.

Using the values above, we can show that the parallel algorithm will be faster than the sequential algorithm when $dx \geq 2.8P/(P - 1)$. Thus, if $P = 2$, then the parallel algorithm is faster for all $dx \geq 6$. If $P = 3$, then the parallel algorithm is faster for all $dx \geq 5$. If $P = 4, 5, \cdots, 14$, then the parallel algorithm is faster for all $dx \geq 4$, and if $P \geq 15$, the parallel algorithm is faster for all $dx \geq 3$.

The sequential circle algorithm

Using an approach similar to the one just presented for straight lines, I developed a parallel algorithm for drawing circles. Bresenham designed an efficient sequential circle algorithm,[5-7] and McIlroy,[8] Suenaga, Kamae, and Kobayashi,[9] and others described different circle algorithms. Nevertheless, I refer primarily to Bresenham's work.

The basic approach of the algorithm is to move the x coordinate step by step from its value at the top of the circle (at 90 degrees or due north) to its value on the circle at 45 degrees or northeast. In other words, x

```
Procedure par_bres_line(k, x0, y0, xf, yf : Integer);
  Var
    dx, dy, x, y, x_end, p, const1, const2 : Integer;
    w, xs, ys, t1, t2 : Integer;                        (1)
Begin
    dx := Abs(x0 – xf );dy := Abs(y0–yf );
    (Do not compute p yet)
    const1 := 2 · dy;
    (Do not compute const2 yet)
    If x0>xf Then
        Begin xs := xf; ys := yf End
      Else Begin xs := x0; ys := y0 End;
    w := (dx + P_Minus_1) Div P;                        (2)
    x := k · w;                                         (3)
    x_end := x + w;                                     (4)
    If x_end > dx Then x_end := dx;                     (5)
    t1 := dx + dx;                                      (6)
    t2 := x· const1;                                    (7)
    y := (t2 + dx) Div t1;                              (8)
    p := const1 – dx + t2 – t1 · y;                     (9)
    const2 := const1 – t1;
    x := x + xs;                                        (10)
    x_end := x_end + xs;                                (11)
    y := y + ys;                                        (12)
    If k = 0  Then set_pixel(xs,ys);                    (13)
    While x < x_end Do
        Begin
      x := x + 1;
      If p < 0 Then p := p + const1
        Else Begin y := y + 1; p := p + const2 End;
      set_pixel(x,y);
    End
End
```

Figure 1. The line algorithm in parallel.

ranges from 0 to $x_{\max}$ = Floor(radius · cosine(45 degrees)).

The y coordinate starts with its value at the top of the circle (y = radius), and at each step the algorithm chooses the new y coordinate to be either the same or one less than the previous y coordinate. The algorithm chooses the y coordinate based on whether the current value of the integer variable p is negative or nonnegative. The algorithm also updates the value of p at each step, again by an amount depending on whether its current value is negative or nonnegative.

In this manner the algorithm determines the best raster representation of the true semicircle in the angle from 90 to 45 degrees. Symmetry determines the points for the remaining seven eighths of a circle.

Dividing the work

There are two obvious approaches for dividing this work among P processors. One approach involves dividing the range of the x coordinate, from 0 to $x_{\max}$, into P fairly equal segments, then assigning a different processor to each segment to perform the appropriate computations.

This approach requires each processor (other than the first one) to determine its initial y coordinate after determining its initial x coordinate. This determination will require the computation of a square root or some other relatively expensive function. This follows from the equation of a circle with center at the origin and a radius of r, which is $x^2 + y^2 = r^2$, or $y = \pm((r^2 - x^2))^{0.5}$. I can also write $y = r \sin(\arccos(x/r))$.

In any case, the dependency of x and y on r make a simpler calculation impossible, hence this approach requires a start-up overhead that is probably unsatisfactory.

An alternative might be to use a table of stored values of y as a function of r and the processor number. We might incorporate the use of interpolation to reduce the size of the table. The number of values required would be $(P - 1) \cdot x_{\max}$ without interpolation (since no values would be needed for Processor 0). This is a classic trade-off of space for time—its implementation is straightforward and requires no further discussion here.

The second approach for dividing the work among processors involves dividing the arc of the circle, from 90 to 45 degrees, into P nearly equal subarcs and assigning a different processor to do the computations for each subarc. For example, if $P = 4$, then each processor would be responsible for a subarc of approximately 45 degrees/4 = 11.25 degrees. We can employ some tricks to efficiently determine the initial values for the x and y coordinates and for p, therefore this approach leads to an efficient parallel algorithm for circle drawing.

Initial values for variables *x*, *y*, and *p*

Let k denote the processor number, where $0 \le k \le P - 1$. Let $x(k)$ be the initial value of variable x for processor k, let $Y(k)$ be the initial value of variable y for processor k, and let $y(k)$ be an initial approximation for $Y(k)$. (I will often omit the reference to k in x, Y, y, and other values where the dependency on k is understood.) Let $G(k) = (k/P)45$ degrees.

Temporarily, we use the following formulas:

$$x(k) = \text{Round}(r \cdot \cos(90 - G)) = \text{Round}(r \cdot \sin(G))$$
$$y(k) = \text{Round}(r \cdot \cos(G))$$

Note that the algorithm can use stored values of the sine and cosine functions for fixed P and for $k = 0, 1, \cdots, P - 1$. Therefore, the sine and cosine calculations are not performed within the algorithm itself. Accordingly, let

$$A''(k) = \sin(G) \tag{6}$$
$$B''(k) = \cos(G) \tag{7}$$

Then

$$x = \text{Round}(r \cdot A'') \tag{8}$$

and

$$y = \text{Round}(r \cdot B'') \tag{9}$$

It is important, of course, to avoid the floating-point arithmetic implicit within Equations 8 and 9. Accordingly, let $r_{\max}$ denote the maximum possible value of the radius r (that is, 1,023 or 32,767), and let

$$t = 1 + \text{Ceiling}(\log_2(r_{\max} + 1)) \tag{10}$$

Then let $A'(k) = 2^t \sin(G)$, $B'(k) = 2^t \cos(G)$, $A(k) = \text{Round}(A')$, and $B(k) = \text{Round}(B')$. Now replace Equations 8 and 9 with

$$x = \text{Round}\left(\frac{r \cdot A}{2^t}\right) \tag{11}$$

$$y = \text{Round}\left(\frac{r \cdot B}{2^t}\right) \tag{12}$$

As mentioned previously, division by a power of 2 is implemented as a right shift, hence the computation of Equations 11 and 12 requires an integer multiply and a shift for each. In effect, we approximate the sines and cosines to an accuracy of t binary bits.

Some finite arithmetic errors are inevitable in approximating A and B and in dividing by 2^t. An error in computing $x(k)$ is of no direct concern, since the start of each subarc is somewhat arbitrary. However, it is imperative that $Y(k)$ be exactly the same as the y coordinate that we determined for the x coordinate $x(k)$ (using the sequential circle algorithm). In other words, we must have

$$Y(k) = \text{Round}(g \cdot x(k)) \tag{13}$$

where $g(x) = ((r^2 - x^2))^{0.5}$ is the equation of the circle.

Because of errors in computing y (or x), it is possible that Equation 13 will not be satisfied for $Y = y$. The algorithm therefore tests three possible values for Y, namely $y_1 = y$, $y_2 = y + 1$, and $y_3 = y - 1$. The value chosen will be the one that gives a point closest to the true circle, that is, the one that minimizes the formula $r^2 - x^2 - y_i^2$, $i = 1, 2, 3$. Let

$$d_1 = r^2 - x^2 - y^2 = (r + x)(r - x) - y^2 \tag{14}$$
$$d_2 = r_2 - x^2 - (y + 1)^2$$
$$d_3 = r^2 - x^2 - (y - 1)^2$$

Then

$$d_2 = d_1 - 2y - 1 \tag{15}$$
$$d_3 = d_1 + 2y - 1 \tag{16}$$

We can now use Equations 14, 15, and 16 to determine the proper value for Y.

After determining the correct starting values $x(k)$ and $Y(k)$, we need to determine the correct starting value $p(k)$. As Hearn and Baker[6] and Bresenham[7] showed, $p(k) = 2(x + 1)^2 + Y^2 + (Y - 1)^2 - 2r^2$. If $Y = y$, then we get $p(k) = -2d_1 + 4x - 2y + 3$. If $Y = y + 1$, then we get $p(k) = -2d_1 + 4x + 2y + 3$. If $Y = y - 1$, then we get $p(k) = -2d_1 + 4x - 6y + 7$. In any case, the computation of $p(k)$ requires no additional multiplications or divisions beyond that used to compute $x(k)$ and $Y(k)$.

Proof of closeness

Now we need to prove that the correct value of $Y(k)$ is one of the values y, $y + 1$, or $y - 1$. To prove this we use one lemma. First, let's introduce the following notation:

```
Procedure par_bres_circle
      ($k$, x_center, y_center, radius : Integer);
  Const
    $A$, $B$, and $W$ are multiplied by 2 in order to
    facilitate Rounding in Lines 2 and 3.
$A$ = Fill in the correct integer constant using the
formula $A = 2\ \mathrm{Round}(2^t \sin((k / P)\ 45°))$
$B$ = Fill in the correct integer constant using the
formula $B = 2\ \mathrm{Round}(2^t \cos((k / P)\ 45°))$
$W$ = Fill in the correct integer constant using the
formula $W = 2(2^t)$
Var
    $p, x, y$ : Integer;
    $d1, d2, d3$, abs_$d1$, $y2$, xstop : Integer;          (1)
Begin
    $x := (\mathrm{radius} \cdot A + 1)$ Div $W$;              (2)
    $y := (\mathrm{radius} \cdot B + 1)$ Div $W$;              (3)
    $y2 := y + y$;                                       (4)
    $d1 := (\mathrm{radius} + x) \cdot (\mathrm{radius} - x) - y \cdot y$;       (5)
    $d2 := d1 - y2 - 1$;                                 (6)
    $d3 := d1 + y2 - 1$;                                 (7)
    Abs_$d1$ := Abs($d1$);                                (8)
    $p := 4 \cdot x - 2 \cdot d1$;                              (9)
    If Abs($d2$) < Abs_$d1$ Then                          (10)
      Begin $y := y + 1$;                                (11)
          $p := p + y2 + 3$ End                           (12)
      Else If Abs($d3$) < Abs_$d1$ Then                   (13)
        Begin $y := y - 1$;                              (14)
            $p := p - y2 - y2 - y2 + 7$ End               (15)
        Else $p := p - y2 + 3$;                           (16)
    x_stop($k$) := $x$;                                   (17)
    Synchronize;                                       (18)
    xstop := x_stop($k + 1$)                             (19)

    While $x <$ xstop Do                                  (20)
      Begin
        plot_circle_points;
        If $p < 0$ Then $p := p + 4 \cdot x + 6$
          Else Begin $p := p + 4 \cdot (x - y) + 10$;
                     $y := y - 1$ End;
        $x := x + 1$
      End;
                                                       (21)
    End;
```

The procedure for Processor $P - 1$ is the same as above, except for the following minor changes:
Line 19 is not needed.
Line 20 should be changed to:

```
  While $x < y$ Do                                      (20)
```

Line 21 should become as follows:

```
  If $x = y$ Then plot_circle_points                    (21)
```

The variable "xstop" is not needed in Line 1.

Figure 2. The circle algorithm in parallel. This example is the procedure for $0 \le k \le P - 2$.

$$x' = r \cdot \sin(G) \qquad y' = r \cdot \cos(G)$$
$$x'' = \frac{r \cdot A}{2^t} \qquad y'' = \frac{r \cdot B}{2^t}$$
$$d_{x1} = x - x'' \qquad d_{y1} = y - y''$$
$$d_{x2} = x'' - x' \qquad d_{y2} = y'' - y'$$

Lemma 1. $|x - x'| < 0.75$ and $|y - y'| < 0.75$.

Proof. It follows immediately from the above notation that $x' = rA'/2^t$ and $y' = rB'/2^t$. By definition of the Round function, $|d_{x1}| = |\mathrm{Round}(rA/2^t) - rA/2^t| \le 0.5$ and $|A - A'| = |\mathrm{Round}(A') - A'| \le 0.5$. From Equation 10 we get $t - 1 \ge \log_2(r_{\max} + 1)$. Therefore, $2^{t-1} \ge r_{\max} + 1$, therefore $2^{t-1} > r_{\max}$, and therefore $r_{\max}/2^t < 0.5$.

Now we can get $|d_{x2}| = |rA/2^t - rA'/2^t| = r/2^t\ |A - A'| \le r_{\max}/2^t\ |A - A'| < 0.5 \cdot 0.5 = 0.25$. We therefore get $|x - x'| \le |x - x''| + |x'' - x'| = |d_{x1}| + |d_{x2}| < 0.5 + 0.25 = 0.75$.

By a symmetrical argument for the y coordinate, we get $|y - y'| < 0.75$.

We are now ready to prove the following theorem:

Theorem 1. $Y(k)$ is equal to one of the values $y(k)$, $y(k) + 1$, or $y(k) - 1$.

Proof. From Equation 13 we need to show that $\mathrm{Round}(g(x)) = y$, $\mathrm{Round}(g(x)) = y + 1$, or $\mathrm{Round}(g(x)) = y - 1$. This is equivalent to showing that $y - 1 \le \mathrm{Round}(g(x)) \le y + 1$, that is, $y - 1.5 \le g(x) < y + 1.5$. We therefore need only show that $|g(x) - y| < 1.5$.

Recall that $g(z) = (r^2 - z^2)^{0.5}$. Therefore, the derivative $dg(z)/dz = -z\,(r^2 - z^2)^{-0.5}$. In the interval $0 \le z \le r \cos(45$ degrees), we have $(r^2 - z^2)^{0.5} \ge z$. Therefore, $|dg(z)/dz| \le 1$ on this interval. Therefore, $|g(x) - g(x')| \le |x - x'| < 0.75$ by Lemma 1. Now $g(x') = g(r \sin(G)) = r \cos(G) = y'$. Therefore, $|g(x) - y'| < 0.75$.Using the second part of Lemma 1, we finally get $|g(x) - y| \le |g(x) - y'| + |y' - y| < 0.75 + 0.75 = 1.5$, and the proof is complete at last.

The parallel circle algorithm

I can now present the parallel circle algorithm in Pascal, as in Figure 2. The algorithm makes use of the following global array in shared memory:

x_stop: Array(0..P_Minus_1) of Integer

The starting x coordinates for processors $0, 1, \cdots, P-1$ are stored in this array. The stopping x coordinate for processor k, $0 \leq k \leq P-2$, is taken to be the starting x coordinate for processor $k+1$. There is a slightly different procedure for processor $P-1$. Note that this technique, including appropriate synchronization, can be avoided if each processor computes both x_start and x_stop, but that would reduce concurrency.

I assume that the Synchronize procedure invoked below delays each of the parallel processors until all of them have executed the Synchronize. The procedure plot_circle_points, given in Step 7, uses symmetry to plot eight points for each (x, y) pair, except that if $x = 0$ or $x = y$, then it should plot only four points.

Line 1 in the parallel algorithm declares new variables, and Lines 2 through 20 constitute new executable statements. (Actually, Lines 2, 3, 9, and 20 replace statements in the sequential algorithm.) Note that the additional computation requires no divides and only four multiplies (in Lines 2, 3, and 5), and that it lies entirely outside the While loop. I assume that multiplication or division by a power of two is implemented as a shift.

Analysis of the circle algorithm

The analysis of the parallel circle algorithm is not as clean as for the parallel line algorithm for two primary reasons. First, the algorithm does not evenly distribute the iterations of the While loop among the P processors. Second, the rate of execution of each of the two paths within the While loop is not a constant over the range of the loop control variable.

The lower numbered processors, whose subarcs start closer to the top of the circle, will have more iterations of the While loop because they cover a greater horizontal distance. On the other hand, lower numbered processors will also more often take the shorter path through the While loop (that is, the path for $p < 0$) because they will decrease y less often.

In general, the time $W(k)$ for the execution of the While loop by processor k is given by $W(k) = C \cdot \mathrm{I}(k)(\mathrm{u}(k) + (1-u(k))v)$, where C is the time for execution of one iteration when the shorter path is taken, $I(k)$ is the number of iterations, $u(k)$ is the fraction of iterations that take the shorter path, and v is the ratio of longer path time to shorter path time (103 percent in my implementation). For almost all practical cases, $W(k)$ is maximized at $k = 0$, hence I shall take the processing time for the parallel algorithm as a whole to be the time for processor 0.

Letting B denote the setup time for the statements preceding the While loop, the processing time for the parallel algorithm becomes $t_p(P, r) = B + W(0) = B + C \cdot r$ (sin(45 degrees$/P$) + $(v-1)$ (1 − cos(45 degrees$/P$))). Letting A denote the setup time for the statements preceding the While loop in the sequential algorithm, the time for the sequential algorithm is given by $t_s(r) = A + C \cdot r$ (sin(45 degrees) + $(v-1)$(1 − cos(45 degrees))) $\approx A + C \cdot r\,(0.414 + 0.293v)$.

As r gets large, the effect of A and B becomes insignificant. Approximating v as 1.0, we get $L(P) = \lim_{r\to\infty} S(P, r)$ = sin(45 degrees)/sin(45 degrees$/P$). For example, $L(4)$ is approximately 3.62, and $L(16)$ is approximately 14.4. Letting $R(P) = L(P)/P$ denote the ratio of $L(P)$ to the maximum possible speedup of P, we have

$$\begin{aligned}\lim_{P\to\infty} R(P) &= \sin(\pi/4)/\lim_{P\to\infty}(P \sin(\pi/4P)) \\ &= \sin(\pi/4)/(\pi/4) \approx 90.03\text{percent}\end{aligned}$$

It can be shown that $R(P)$ is a decreasing function in P, therefore $R(1) > R(2) > \cdots > 90.03$ percent.

As for the line algorithm, I made a rough analysis using a simulated implementation of the circle algorithm in Pascal on an Intel 8088-based personal computer. The results are as follows, using an imaginary time unit in which the time for one iteration of the While loop is 1: $C = 1$, $A = 0.26$, $B = 0.86$, $v = 1.03$.

Using these parameters, the timing formulas become

$$\begin{aligned}t_s(r) &= 0.26 + 0.716r \\ t_p(P,r) &= 0.86 + r(\sin(45^\circ/P) \\ &\quad + 0.03(1-\cos(45^\circ/P))\end{aligned}$$

For example, $t_s(100) = 72$, $t_p(4, 100) = 20$, and $S(4, 100) = 3.5$.

Using the parameters above, we can see that the parallel algorithm with $P \geq 2$ will be faster than the sequential algorithm for all $r > 2$.

Concluding remarks

I developed and analyzed two efficient parallel algorithms for drawing graphics primitives on a raster scan display device. One algorithm generates lines and the other generates circles. The line algorithm approaches a perfect speedup of P as the line length approaches infinity, and the circle algorithm approaches a speedup greater than $0.9P$ as the circle radius approaches infinity.

Both algorithms have a relatively small setup overhead. For the line algorithm the setup includes only three integer multiplies and one or two integer divides and is equivalent to about seven loop iterations. For the circle algorithm the setup includes only four integer multiplies and no divides and is equivalent to about one loop iteration.

We can extend this approach for parallelizing elementary algorithms to other graphics processes. Drawing ellipses and filling polygons look like two good candidates for future parallel work. ■

References

1. M.J. Quinn, *Designing Efficient Algorithms for Parallel Computers,* McGraw-Hill, Hightstown, N.J., 1987.
2. R.F. Sproull, "Using Program Transformations To Derive Line-drawing Algorithms," *ACM Trans. Graphics,* Vol. 1, No. 4, Oct. 1982, pp. 259-273.
3. R. Pike, "Graphics in Overlapping Bitmap Layers," *ACM Trans. Graphics,* Vol. 2, No. 2, Apr. 1983, pp. 135-160.
4. J.E. Bresenham, "Algorithm for Computer Control of a Digital Plotter," *IBM Systems J.,* Vol. 4, No. 1, Jan. 1965, pp. 25-30.
5. J.D. Foley and A. Van Dam, *Fundamentals of Interactive Computer Graphics,* Addison-Wesley, Reading, Mass., 1982, pp. 433-436 and 442-445.
6. D. Hearn and M.P. Baker, *Computer Graphics,* Prentice-Hall, Englewood Cliffs, N.J., 1986, pp. 58-61 and 67-69.
7. J.E. Bresenham, "A Linear Algorithm for Incremental Digital Display of Circular Arcs," *Comm. ACM,* Vol. 20, No. 2, Feb. 1977, pp. 100-106.
8. M.D. McIlroy, "Best Approximate Circles on Integer Grids," *ACM Trans. Graphics,* Vol. 2, No. 4, Oct. 1983, pp. 237-263.
9. Y. Suenaga, T. Kamae, and T. Kobayashi, "A High-speed Algorithm for the Generation of Straight Lines and Circular Arcs," *IEEE Trans. Computers,* Vol. C-28, No. 10, Oct. 1979, pp. 728-736.

William Wright is a professor of computer science at Southern Illinois University, Carbondale. His primary area of research is file organization, especially tree-structured files. He has also done research in data structures and in cluster analysis. He is a member of ACM and the IEEE Computer Society.

Wright received a masters degree in mathematics from the University of Illinois in 1968 and a doctorate in applied mathematics and computer science from Washington University in 1972.

Wright can be reached at Southern Illinois University, Computer Science Department, Carbondale, IL 62901-4511.

3D at HP

Hewlett-Packard Laboratories currently has two challenging positions in 3D computer graphics research and development.

All applicants should have UNIX* operating system, X-Window application, graphics algorithms and extensive programming coursework or experience.

Hardware Design Engineer

Your expertise will be used to architect, characterize, and design 3D raster graphics hardware in conjunction with high performance computer workstation design. Experience in high speed digital design, VLSI circuit design, RISC processor based computer design, and parallel processing hardware design is required. Experience in the application of graphics algorithm techniques is desired. A PhD or MS with job related experience in EE, CS, or equivalent with expertise in hardware design for 3D graphics workstations is required.

3D Graphics Software Engineer

You will be responsible for designing 3D graphics rendering algorithms for a high performance workstation architecture. Experience in polygon rendering and advanced rendering algorithms is required. Familiarity with object oriented programming and window system programming is also preferred. We require a PhD or MS, or equivalent, with job related experience in CS or a related area with expertise in 3D graphics rendering algorithms.

Please send your resume to: **Hewlett-Packard Laboratories, 1501 Page Mill Rd., Bldg. 1U, Dept. 6010, Palo Alto, CA 94304.** Hewlett-Packard Company is an Equal Opportunity/Affirmative Action Employer.

*UNIX is a registered trademark of AT&T in the U.S.A. and other countries.

There is a better way.

HEWLETT PACKARD