PARALLEL ALGORITHMS FOR INTERACTIVE MANIPULATION OF DIGITAL TERRAIN MODELS*

E. W. Davis, D. F. McAllister, and V. Nagaraj

Department of Computer Science
North Carolina State University
Raleigh, North Carolina 27695-8206

ABSTRACT

Interactive three dimensional graphics applications, such as terrain data representation and manipulation, require extensive arithmetic processing. Massively parallel machines are attractive for this application since they offer high computational rates, and grid connected architectures provide a natural mapping for grid based terrain models. This paper presents algorithms for data movement on the MPP in support of pan and zoom functions over large data grids. It is an extension of earlier work that demonstrated real-time performance of graphics functions on grids that were equal in size to the physical dimensions of the MPP. When the dimensions of a data grid exceed the processing array size, data is packed in the array memory. Windows of the total data grid are interactively selected for processing. Movement of packed data is needed to distribute items across the array for efficient parallel processing. Execution time for data movement was found to exceed that for arithmetic aspects of graphics functions. Performance figures are given for routines written in MPP Pascal.

Keywords: interactive graphics, parallel algorithms, MPP, terrain models

INTRODUCTION

Multiprocessor architectures have been used for several years to meet the demanding computational requirements of interactive, 3D graphics. The computing resources may take the form of specialized hardware that exploits the vector and pipeline suitability of graphics problems.(Refs. 4, 5, 7, 9, and 13). However, there are several architectures which were not designed specifically with graphics applications in mind, but are versatile enough to be used advantageously on the vectorizable nature of the computations (Refs. 1, 2, 3, 6, and 11). This paper focuses on the use of one such machine, the MPP, for a specific graphics problem: representation of digital terrain data and interactive manipulation of corresponding terrain images.

Prior work has shown the feasibility of interactive manipulation of stereo pair images of small terrain models on the MPP (Ref. 10). In the prior work grids of terrain data were 128 by 128 points, exactly matching the dimensions of the MPP and leading to a natural mapping of terrain data to the processing grid. In order to increase the possible applications, it is necessary to implement interactive graphics operations on much larger databases of terrain points. Data structures and algorithms reported in this paper are for pan and zoom functions on larger databases. The work is more completely described in Ref. 8.

* This work was partially supported by NASA Goddard Space Flight Center through the MPP Working Group.

PARALLEL ALGORITHMS

Data Representation

Grid-based digital terrain models contain an m by n rectangular grid of points (x_i, y_j), $1 \le i \le m$, $1 \le j \le n$, which correspond to longitude and latitude values on the earth's surface. Each grid point has an associated value $z_{i,j}$ which is the elevation above sea level at the point (x_i, y_j).

Typically, a 128 by 128 grid of elevation points is considered to match the MPP's architecture. Elevation points are assigned to processing elements (PEs) in a straightforward way with $PE_{i,j}$ containing the grid points $(x_i, y_j, z_{i,j})$. A grid that just matches the PE array size constitutes a small terrain model and a limited display. In order to examine a different part of the terrain, it is necessary to input a new set of coordinates with elevation points from the host or staging memory. We wish to make a large database available within the array unit at the outset, and be able to display arbitrary parts of the terrain in real-time, under interactive selection control.

The methods described in this paper can handle a terrain database up to size 512 by 512 in the limited 1K per PE memory of the MPP. However, for purposes of illustration, we consider a model with a 4 by 4 array of PEs and an 8 by 8 array of terrain data. That is, there are four data points per PE.

In order to exploit the full parallel capabilities of the MPP it is necessary that terrain data points to be processed be spread across the available PEs. This will require some movement of data within the processing array. A particular storage mapping, shown in figures 1 and 2, is chosen because it supports the movements used in pan and zoom functions. The original 8 by 8 data array, I in figure 1, is reformatted into four 4 by 4 subarrays, A, B, C, and D in figure 2. Data elements are mapped as follows:

$A = \{a_{i,j}\}$ where $a_{i,j} = I_{2i,\ 2j}$;
$B = \{b_{i,j}\}$ where $b_{i,j} = I_{2i,\ 2j+1}$;
$C = \{c_{i,j}\}$ where $c_{i,j} = I_{2i+1,\ 2j}$;
$D = \{d_{i,j}\}$ where $d_{i,j} = I_{2i+1,\ 2j+1}$;
where $0 \le i \le 3$ and $0 \le j \le 3$.

00	01	02	03	04	05	06	07
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

Figure 1. Terrain data in scan line order. Array I.

After reformatting, $PE_{i,j}$ will contain data points from the same position in each of the four data arrays. That is, the terrain data points shown in figure 2 as $a_{i,j}$, $b_{i,j}$, $c_{i,j}$, and $d_{i,j}$, collectively called $t_{i,j}$, are mapped onto $PE_{i,j}$.

00	02	04	06
20	22	24	26
40	42	44	46
60	62	64	66

Array A

01	03	05	07
21	23	25	27
41	43	45	47
61	63	65	67

Array B

10	12	14	16
30	32	34	36
50	52	54	56
70	72	74	76

Array C

11	13	15	17
31	33	35	37
51	53	55	57
71	73	75	77

Array D

Figure 2. Reformatted data, as stored in a 4 by 4 PE array.

Windows

A subset of terrain points that exactly conforms to the dimensions of the MPP is called a window. Figure 3, where a square corresponds to a single PE, shows that a window only includes data points that are localized to part of the PE array. To exploit the parallelism of the machine, it is necessary to spread the subset of points over the entire PE array such that each PE has one data point from the window. Figure 4 shows the distributed data produced by the "spread" function, described below.

a00	b00	a01	b01	a02	b02	a03	b03
c00	d00	c01	d01	c02	d02	c03	d03
a10	b10	a11	b11	a12	b12	a13	b13
c10	d10	c11	d11	c12	d12	c13	d13
a20	b20	a21	b21	a22	b22	a23	b23
c20	d20	c21	d21	c22	d22	c23	d23
a30	b30	a31	b31	a32	b32	a33	b33
c30	d30	c31	d31	c32	d32	c33	d33

W

Figure 3. Packed data with a window selected.

d10	c11	d11	c12
b20	a21	b21	a22
d20	c21	d21	c22
b30	a31	b31	a32

Figure 4. Distribution of the selected window over the array.

The "Spread" Function

This function consists of a series of bit plane data movement operations. All statements in this routine are executed simultaneously on all PEs. Two data movement masks NSMASK and EWMASK are created. MPP Pascal (Ref. 12) primitives such as "rotate" and "any" are used in conjunction with these data planes to route the selected data to the target PE. The "spread" routine is invoked four times so that elements of A, B, C, and D enclosed by the window can be moved to their target positions, one array at a time. Parameter SOURCE is the particular source array; either A, B, C, or D. Parameter INDEX is the position of the upper left corner of the window when it is packed in the array.

```
routine SPREAD
        MAKEMASK(A, B, C, D, SOURCE, INDEX, NSMASK,
        EWMASK);
        where (NSMASK ≠ 0)
                rotate EWMASK and SOURCE to positions
                indicated by NSMASK;
        end where;
        where (EWMASK ≠ 0)
                rotate SOURCE to position indicated by
                EWMASK;
        end where;
end routine.
```

Routine SPREAD calls the following routine, MAKEMASK.

```
routine MAKEMASK (A, B, C, D, SOURCE, INDEX, NSMASK,
EWMASK);
      for PEs outside the selected window
            NSMASK = 0;
            EWMASK = 0;
            SOURCE = 0;
      end for;
      for PEs inside the selected window
            depending on INDEX
                  SOURCE = A or B or C or D;
            COMPUTEMASK (NSMASK, EWMASK);
      end for;
end routine.
```

Routine MAKEMASK calls routine COMPUTEMASK.

```
routine COMPUTEMASK (NSMASK, EWMASK);
      If ti,j in PEi,j needs to be moved to PEx,y
            NSMASK = x - i;
            EWMASK = y - j;
      end if;
end routine.
```

Once data has been spread over the entire array, further functions such as intensity calculations, hidden surface removal, and rendering, can be executed in parallel with each PE handling one data point as in Ref. 10.

Pan and Zoom Functions

We are interested in the ability to move the window about in the array of terrain point sets. This process of moving a window in object space is called "panning". The routine that follows provides the pan function as simply a selection of the window to be spread. This routine is used prior to computation of intensities and image rendering.

```
routine PAN ( ORIGIN);
      choose window based on user-defined ORIGIN;
      SPREAD;
end routine.
```

Another means of examining the entire database is to sacrifice resolution for extent of coverage. By choosing representative data points from the database it is possible to zoom-in or zoom-out using the routine below. The spacing between adjacent chosen points determines the resolution or extent of zoom. For our small example there are only two zoom settings. Maximum resolution is achieved by choosing a window and carrying out a SPREAD. For minimum resolution, it is possible to simply select one of the four arrays A, B, C, or D. With greater terrain data packing factors, intermediate levels of resolution,, involving different extents of data movement, are possible.

We note again that once data has been spread over the entire array, further functions such as intensity calculations, hidden surface removal, and rendering, can be executed in parallel.

```
routine ZOOM (ORIGIN, RESOLUTION, INDEX);
      if RESOLUTION = MAX
            choose window based on user-defined ORIGIN;
            SPREAD;
      end if;
      if RESOLUTION = MIN
            depending on INDEX
            FINAL = A, or B, or C, or D;
      end if;
end routine.
```

TIMING ANALYSIS

Graphics programs were written in MPP Pascal (Ref. 11) for a database of 256 by 256 terrain points. Execution time can be determined using system provided timing routines. The structure of a typical graphics program loop with pan and zoom operations is to distribute the data points from the selected window over the entire array, then compute intensities of all pixels in parallel, then render the image. Table 1 gives actual timing measurements

distribute data	2283 milliseconds
compute intensities	5 milliseconds
render image	819 milliseconds

Table 1. Measured timing for a 256 by 256 database.

The time taken to distribute data is almost entirely accounted for by 28 calls to the SPREAD routine. It is invoked seven times for each of the four arrays A, B, C, and D discussed earlier. Moreover, the time taken to execute SPREAD once is almost entirely accounted for by an inner loop which uses the MPP Pascal "rotate" function extensively. The measured time for one "rotate" is 204 microseconds. The time taken just for executing "rotate" functions while distributing data is 2172 milliseconds. SPREAD is also used in rendering an image and contributes greatly to its execution time.

An equivalent to the "rotate" function can be achieved in lower level languages of the MPP in 3.3 microseconds, rather than the 204. Table 2 is derived from the measured timing by substituting the much lower rotate time.

distribute data	142 milliseconds
compute intensities	5 milliseconds
render image	53 milliseconds

Table 2. Expected timing with efficient "rotate".

A necessary condition for real-time graphics image generation is that one pass through the .loop of the program must take no more than 33 milliseconds. Even with the expected time table, each pass takes 200 milliseconds, yielding only five frames per second.

An alternative approach is to bypass data distribution in favor of iteratively using a smaller portion of the processing array.

Image generation time using this approach, for the same database as above, was measured at 73 milliseconds. Execution time is reduced but full parallelism of the array is not used. As the number of data points in the database is increased by a factor of K, the number of active PEs in the array unit is decreased by K. This will result in a factor of K increase in time for the intensity computation alone.

A third approach is to maintain the database in the staging memory and bring in only the data points needed for each computation. Image generation time reduces to 59 milliseconds. However, data in the staging memory is only accessible along certain predefined boundaries. This complicates pan and zoom functions.

CONCLUSION

Prior work has shown massive parallelism to be suitable for graphics applications on data arrays which fit the processing array size. When larger arrays must be handled, the time involved in moving data becomes the dominant part of the problem and can take the performance out of the real-time realm.

REFERENCES

1. Batcher, K. E., "Design of a Massively Parallel Processor", *IEEE Trans. on Computers*, Vol. C-29, No. 9, Sept. 1980, pp. 836-840.

2. Davis, E. W., and J. H. Reif, "Architecture and Operation of the BLITZEN Processing Element", Proc. of the Third Int. Conf. on Supercomputing, Boston, MA, May 1988.

3. Blevins, D. W., E. W. Davis, R. A. Heaton, and J. H. Reif, "BLITZEN: A Highly Integrated Massively Parallel Machine", Proc. of the Second Symposium. on the Frontiers of Massively Parallel Computing (this proceedings), Fairfax, VA, October 1988.

4. Fuchs, H., et al, "Fast Spheres, Shadows, Textures, Transparencies and Image Enhancements in Pixel-Planes", SIGGRAPH, Vol. 19, No. 3, July 1985, pp. 111-120.

5. Glassner, H., and H. Fuchs, "Hardware Enhancements for Computer Graphics", *Fundamental Algorithms in Computer Graphics*, R. A. Earnshaw (Ed.), 1985, pp. 631-658.

6. Hillis, D. W., *The Connection Machine*, The MIT Press, Cambridge, MA, 1986.

7. Levinthal, A., and T. Porter, "CHAP - A SIMD Graphics Processor", *ACM Computer Graphics*, Vol. 18, No. 3, July 1984.

8. Nagaraj, V., *Graphics Algorithms for Parallel Architectures*, M.S.Thesis, Dept. of Electrical and Computer Engineering, North Carolina State University, 1988.

9. Niimi, H., Y. Imai, M. Murakami, S. Tomita, and H. Hagiwara, "A Parallel Processor System for Three Dimensional Color Graphics", *ACM Computer Graphics*, Vol. 18, No. 3, July 1984.

10. Pol, S., D. F. McAllister, and E. W. Davis, "An Application of the MPP to the Interactive Manipulation of Stereo Images of Digital Terrain Models", Proc. of the First Symposium on the Frontiers of Massively Parallel Scientific Computing, NASA/GSFC, October 1986.

11. Potter, J., Editor, *The Massively Parallel Processor*, The MIT Press, 1985.

12. Science Applications Research, MPP Pascal Programmer's Guide, March 1988.

13. Stellar Computer, Inc., *Stellar Graphics Supercomputer Model GS1000 System Overview*, 1987.