Data Parallel C Extensions Applied to Image Processing:

A Faster Search for Extraterrestrial Objects

David J. McNamara
Pacific-Sierra Research Corp.

Abstract

Issues in porting an existing C image processing application into Data Parallel C are discussed. This paper shows several examples of Data Parallel C code. Performance of the application on two distributed parallel systems is presented.

Introduction

Many scientists (amateur and professional alike) have spent considerable efforts in searching for signs of life beyond our solar system. The focus of the search for extra-terrestrial intelligence (SETI) has been to look outside our solar system at radio frequencies for unusual signals.

Unnatural Objects

We took a more earthly and, perhaps, more practical approach to detecting extra-terrestrial objects. In fact, what we are really looking for are “unnatural” objects in natural terrain. Unnatural, in our terminology, refers to objects that would not normally occur in nature; for instance, a plane in the middle of the desert. It would be safe to assume that such an object would not normally have occurred in nature and, in detecting it, we could conclude that someone (or something) put the object in those surroundings. This was the approach described in an article by Mark Carlotto[1]. Carlotto points out that non-natural objects tend to lack self similarity while fractals have the property of being self-similar, that is, fractals look the same across a range of scales or resolutions. He wrote a C program for detecting man-made objects in natural terrain.

The Goal

To be able to process images fast, we decided to rewrite the original C application in Data Parallel C Extensions (DPCE)[2]. DPCE is an ANSI approved set of new features for ANSI C that allows programmers to write efficient code for parallel systems. DPCE brings the data parallel programming model to C by having the ability to declare and use parallel variables, which can be manipulated much more efficiently on parallel systems than normal C arrays. For distributed memory systems, sections of each parallel variable can be placed in different memories on different processors by the compiler, so that those processors can operate in parallel on different pieces of the data. The target parallel systems were an IBM SP2 and a cluster of four DEC Alpha stations.

Data Parallel C

This article is not intended to be a full description of DPCE. Briefly, DPCE gives the programmer the ability to declare data objects as parallel objects along with the necessary syntax to describe how the data objects should be distributed. In addition to providing a way of describing parallel objects, DPCE defines parallel operators to manipulate the data. DPCE is, in some ways, the C language analog of HPF[3].

This article will describe the process of taking an existing C program, whose algorithms fit the data parallel programming model, rewriting the code in DPCE, and analyzing the execution performance of the code on the two parallel systems mentioned above.

Data Parallel Programming

Before we continue with the details of converting the application, a brief discussion of data parallelism would be appropriate.

In the data parallel programming model, the programmer specifies the distribution of data at a very high level, and the program is written as if the data were globally addressable by all processors with a single logical execution thread. This is much easier for the programmer than attempting to directly deal with data that may be on many different processors and with many different execution threads. The compiler creates and manages all of the parallel tasks and data transfer between them based on the user's original distribution of the data. In a nutshell, “data parallel” means to “perform the same instructions on all of the data.” How the actual work is completed is up to the compiler (based on the data distribution information), the program’s algorithm, and the underlying system configuration.

Data parallel programming is most often associated with distributed memory machines, as the syntax for data parallel languages often describe how to *distribute* data; however, there is nothing to prevent compilers of data parallel languages from generating efficient code for other types of architectures, such as shared memory. In fact, the array syntax that is part of DPCE lends itself to exposing parallelism at the loop level. Compilers can take advantage of such syntax and generate efficient code for many types of parallel architectures.

The Application

Getting back to our search for extra-terrestrials, I determined that Mark Carlotto’s program for finding non-natural objects was an excellent candidate for executing in a data parallel environment. The program deals with large arrays and spends most of its time performing calculations on these arrays in a linear fashion.

The program uses a method that estimates the fractal dimension of the image intensity surface within a rectangular window that is about the size of objects one would like to detect, along with the error that results from assuming fractal, or self-similar, behavior. The algorithm calls for processing an entire image by performing linear regression on a

pixel-by-pixel basis within a "sliding window". Two images are produced: one is the local fractal dimension, the other is the local fractal model-fit image.

The application contains one main routine for performing all of the computation. This routine accepts information about the size of the image being processed and the resolution (number of scales) desired. The first step was to allocate the necessary space to store the image and additional computational space:

```
in      = (unsigned char *) malloc(xdim*ydim*sizeof(unsigned char));
top = (short int *) malloc(2*xdim*ydim*sizeof(short int));
bot = (short int *) malloc(2*xdim*ydim*sizeof(short int));
area = (float *) malloc(xdim*ydim*sizeof(float));
sxy = (float *) malloc(xdim*ydim*sizeof(float));
syy = (float *) malloc(xdim*ydim*sizeof(float));
my      = (float *) malloc(xdim*ydim*sizeof(float));
tmp = (float *) malloc(max(xdim,ydim)*sizeof(float));
```

Shapes

Already we've unveiled one deficiency in C, which is the ability to dynamically allocate multi-dimensional arrays and still express them as arrays. Instead, the program allocates the storage and then does the necessary index manipulation in a linear manner. In DPCE, we can define a dynamic *shape*, declare parallel variables of the shape, and then use the objects as the arrays they were intended without having to bother with the complicated linearization of indices.

```
Shape [][]Image;
Shape []Array;
Image = newshape( 2, xdim, ydim );
Array = newshape( 1, max(xdim,ydim) );
…
unsigned char:Image in;
short int:Image top1, bot1;
short int:Image top2, bot2;

float:Image area,my,sxy,syy;
float:Array tmp;
```

You'll notice that we've introduced two additional variables. In the original C, the variables `top` and `bot` were allocated to be twice the size of the image we are processing. The algorithm alternately used the lower half or upper half of these arrays during computation. In the DPCE implementation I choose to define these as separate variables, hence, `top` was broken into `top1` and `top2` while `bot` became `bot1` and `bot2`.

Initial Calculation

The original application contains a large outer loop whose iteration count is determined by the required resolution. For each outer loop iteration, the program determines which half of the large image arrays was to be used as the initial condition, calculated offsets

into the image space for nearest neighbors, and then selected either min or max values from the image:

```
for(y=k; y<ydim-k; y++){
        nw = off + (y-1)*xdim + k - 1;
        n = nw + 1;
        ne = nw + 2;
        w = off + y*xdim + k - 1;
        c = w + 1;
        e = w + 2;
        sw = off + (y+1)*xdim + k - 1;
        s = sw + 1;
        se = sw + 2;
        nc = offn + y*xdim + k;
        for(x=k; x<xdim-k; x++){

                if (comm == 0){
                        bot[nc] = min(bot[ne],bot[se]);
                        …

                        }

                else{
                        bot[nc] = min(bot[n],bot[e]);
                        …
                }
```

Since we've dynamically allocated the space for the image, there is quite a bit of indexing that we need to worry about. What the complicated indexing above boils down to is determine which half the image we use as our initial condition and what are the locations for a given pixel's nearest neighbors. In DPCE, this initial condition loop becomes:

```
 if (k%2 == 1){
   for ( i = k; i < xdim-k; i++ ) {
     for ( j = k; j < ydim-k; j++ ) {
       [j][i]bot2 = min([j-1][i+1]bot1,[j+1][i+1]bot1);
         …
     }
   }
  }
 else
{
    for ( i = k; i < xdim-k; i++ ) {
       for ( j = k; j < ydim-k; j++ ) {
         [j][i]bot1 = min([j-1][i]bot2,  [j][i+1]bot2);
         …
     }
   }
 }
```

This code segment was basically the only part that changed significantly from the original C program. The main reason for this change was to undo the complex linearization of indexing that was required in the C implementation. Once the indexing was solved, the

resulting DPCE was straightforward, using simple loops, left-indexing and nearest neighbor indexing. Left-indexing is the method used for referencing elements of a parallel variable in DPCE. It is not required in DPCE to use indexing in this situation, but I did so for clarity and ease of modifying the original C program.

Array Syntax

It is possible to rewrite this code without the loops and use array syntax:

```
[k:ydim-k-1][k:xdim-k-1]bot1 = min( [k-1:ydim-k-2][k:ydim-k-1][k+1:xdim-k]bot2 );
```

This eliminates the two for loops but I do not think that it makes the code easier to write or understand. My preference for using array syntax is when there are no offsets required on the indices.

Volume Calculation

The next step in the program was to calculate the volume. The original C code looked like the following:

```
for(y=k; y<ydim-k; y++){

        for(x = (k + 1); x < (xdim - k); x++)
            area[x+xdim*y] += area[(x-1)+xdim*y];

        for(x = (wx2 + k); x < (xdim - wx2 - k); x++)
            tmp[x] = area[(x+wx2m1)+xdim*y] - area[(x-wx2)+xdim*y];

        for(x = (wx2 + k); x < (xdim - wx2 - k); x++)
            area[x+xdim*y] = tmp[x];
    }

    for(x=k; x<xdim-k; x++){

        for(y = (k + 1); y < (ydim - k); y++)
            area[x+xdim*y] += area[x+xdim*(y-1)];

        for(y = (wy2 + k); y < (ydim - wy2 - k); y++)
            tmp[y] = area[x+xdim*(y+wy2m1)] - area[x+xdim*(y-wy2)];

        for(y = (wy2 + k); y < (ydim - wy2 - k); y++)
            area[x+xdim*y] = tmp[y];
    }
```

In converting this to DPCE, I simply used our matrix-left indexing capability and eliminated the need for the temporary vector.

```
for(y=k; y<ydim-k; y++)
    for(x = (k + 1); x < (xdim - k); x++)
      [y][x]area += [y][x-1]area;

  [k:ydim-k-1][k+wx2:xdim-k-wx2-1]area =
                        [k:ydim-k-1][k+wx2+wx2m1:xdim-k-wx2-1+wx2m1]area -
                        [k:ydim-k-1][k:xdim-k-1-wx2-wx2]area;
```

```
for(x=k; x<xdim-k; x++)
   for(y = (k + 1); y < (ydim - k); y++)
      [y][x]area += [y-1][x]area;


[k+wy2:ydim-k-1-wy2][k:xdim-k-1]area =
                                   [k+wy2+wy2m1:ydim-k-wy2-1+wy2m1][k:xdim-k-1]area -
                                   [k:ydim-k-1-wy2-wy2][k:xdim-k-1]area ;
```

Accumulation Loop

Still within the outer loop on resolution, there was a loop to accumulate the necessary statistics:

```
temp = log10((double) k);
        mx += temp;
        sxx += (temp * temp);
        for(y = (wy2 + k); y < (ydim - wy2 - k); y++){
                for(x = (wx2 + k); x < (xdim - wx2 - k); x++){
                        area[x+xdim*y] = log10((double) area[x+xdim*y]);
                        my[x+xdim*y] += area[x+xdim*y];
                        sxy[x+xdim*y] += area[x+xdim*y]*temp;
                        syy[x+xdim*y] += area[x+xdim*y]*area[x+xdim*y];
                        }
                }
```

In DPCE, this became:

```
temp = log10((double) k);
   mx += temp;
   sxx += (temp * temp);

   for(y = (wy2 + k); y < (ydim - wy2 - k); y++){
        for(x = (wx2 + k); x < (xdim - wx2 - k); x++){
                [y][x]area = log10((double) [y][x]area);
                [y][x]my += [y][x]area;
                [y][x]sxy += [y][x]area*temp;
                [y][x]syy += [y][x]area*[y][x]area;
        }
   }
```

Finally, there was a doubly nested loop to normalize the statistics and compute the output error; the only changes made to the original C code in this case was moving the right indexing to left indexing.

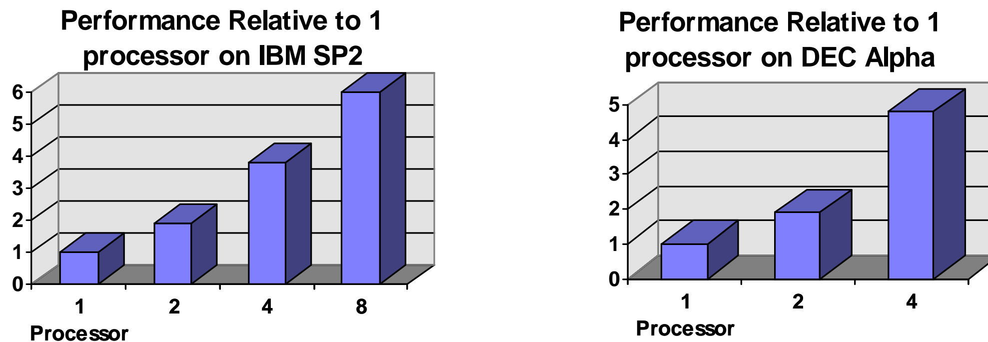
The last step was to write out the resulting images that were produced. This and the remainder of the code remained identical between the original C and DPCE implementations. The resulting DPCE code was compiled on two systems using VAST-DPC, a DPCE compiler developed by Pacific-Sierra Research Corp.

Porting Effort

Overall, it took roughly 4 hours to rewrite this routine in DPCE. The experienced data parallel programmer has the advantage of knowing what types of codes will run efficiently, however; for the programmer new to DPCE, the task can be as simple as re-declaring the appropriate data structures and moving right indexing to the left. This porting effort can then be refined as the programmer identifies bottlenecks.

Performance Results

The resulting application was executed on two parallel systems with the following results:

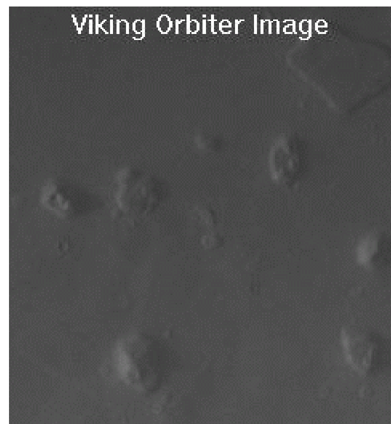


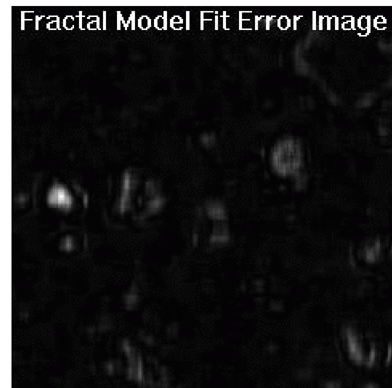
On the IBM SP2, the application scaled well up to 8 processors. On the DEC Alpha we achieved super-linear speed up on four processors which can be attributed to gains made due to more efficient use of cache.

Conclusion

In doing this conversion, I found that very little was required in order to get significant performance improvements. DPCE provided a way to simply and efficiently execute this application in parallel on two different systems without significant rewriting or having to deal with other parallel implementations, such as explicit message passing. The resulting DPCE code is easy to maintain and more clearly reflects the mathematics of the algorithm.

This application was used on images taken by the Mars/Viking Orbiter (Viking frames 35A72, 35A73). Show below is the original image, the fractal model fit error image, and a enlargement of the original image in the area highlighted by the fractal model fit image.





References

[1] *Journal of the British Interplanetary Society*, Vol. 43, pp. 209-217, 1990.

[2] *Data Parallel C Extensions*, Numerical C Extensions Group of X3J11, DPCE subcommittee, Technical Report, Version 1.6, X3J11/94-080, WG14/N395, December 31, 1994

[3] *High Performance Fortran Language Specification*, High Performance Fortran Forum, October 19, 1996, Version 2.0

[4] *Data-Parallel Programming*, Philip J. Hatcher and Michael J. Quinn, MIT Press, 1991