Assignment 3

Using Paraguin to Create MPI Programs

C. Ferner and B. Wilkinson
October 9, 2012

Overview

The goal of this assignment is to use the Paraguin compiler to create parallel solutions using MPI to a couple of embarrassingly parallel applications.

Log in to **babbage.cis.uncw.edu**.

Setup

This part only needs to be done *once*. Edit the file **.bash_profile** in your home directory (notice that the filename begins with a period). Enter the following lines at the bottom of the file and save:

```
export MACHINE=x86_64-redhat-linux
export SUIFHOME=/share/apps/suifhome
export COMPILER_NAME=gcc
`perl $SUIFHOME/setup_suif -sh`
```

Note that the symbols on the last line are *back* single quotes, not single quotes.

Then run the command:

```
. .bash_profile
```

The periods are important, and there is a space between them.

Then create a directory in which to do this assignment and **cd** to that directory. You do not need to provide screenshots of this first section in your submission document.

Hello World (22%)

Create a "hello world" program using the code in Figure 1. Make sure you include the empty lines (those with only a semicolon).

Compile this program with the Paraguin compiler using this command:

```
runparaguin hello.c
```

Take a screenshot of the output from compiling your program to include in your submission document.

```
#ifdef PARAGUIN
typedef void* __builtin_va_list;
#endif

#include <stdio.h>

int __guin_mypid = 0;

int main(intargc, char *argv[])
{
    char hostname[256];

    printf("Master thread %d starting.\n", __guin_mypid);

    ;
    #pragma paraguin begin_parallel

    gethostname(hostname, 255);

    printf("Hello world from thread %3d on machine %s.\n",
                                    __guin_mypid, hostname);

    ;
    #pragma paraguin end_parallel

    printf("Goodbye world from thread %d.\n", __guin_mypid);

    return 0;
}
```

Figure 1: Hello world program

If the program is correct, the compiler will produce a file **hello.out.c**, which is the MPI version of the hello world program. It will also compile this MPI program using **mpicc** to produce a file call **hello.out**. Inspect the **hello.out.c** program to see how Paraguin parallelized your hello program. The header files, such as **stdio.h**, will be included in the **hello.out.c**, so you will need to go to the bottom of the file to see the parallelized version of the hello world program.

Create a job description file to submit your program to the SGE schedule as you did in assignment 2, then submit the program with **qsub**.

```
Master thread is ready, running on compute-0-0.local
Greetings from thread 1, running on compute-0-0.local!
Greetings from thread 2, running on compute-0-0.local!
Greetings from thread 3, running on compute-0-0.local!
```

Figure 2: Hello world output

The output should look something like that of Figure 2.

Re-run this program using different number of processors (up to 32). Take a screenshot of the output of the hello world program after running on the largest number of processors you are able to run it on. (That means that you only need to provide 1 screenshot of the output.)

What to submit

Your submission document should include the following:

1) Your name and school;
2) Whether you are a graduate or undergraduate student;
3) A copy of the hello world source program (hello.c ***not*** hello.out.c);
4) Screenshot from compiling your hello world program with Paraguin;
5) Screenshot of executing the hello world program on the largest number of processors for which you are able to run successfully the program;
6) Screenshot of the output running on the largest number of processorsfor which you are able to run successfully the program.

Matrix Multiplication (UG – 39% ; G – 35%)

Task 1 – Create the program

First you will need some input. Copy the file **/home/cferner/working/work1/input3** to your directory. This file contains a large number of floating point numbers that you can use as input.

Create a program using the skeleton program from Appendix A. You need to fill in the sections labeled "TODO". The first TODO is to broadcast the input. You will use the Paraguin bcast to scatter the input data instead of MPI_Scatter, because Paraguin does not have Scatter implemented. (Alternatively, you could insert the actual MPI_Scatter as you did in assignment 1.)

The second TODO is to specify the partitioning of the **for** loop to execute matrix multiplication (which you will need to add). We can simply execute the outermost loop (i) in parallel. In other words, the partitioning of p=i will work.

The third TODO is to specify the gather. Since we need to gather all values of the **c** matrix, we want to specify the expression to be k=0 in the gather.

The last TODO is the write the for loop nest that will do the calculation of matrix multiplication.

After you are able to compile successfully your matrix multiplication with Paraguin, take a screenshot of the compilation for your submission document.

Task 2 – Create the sequential version

Also make a sequentially executed version of the program using **gcc**. It should look something like this:

```
gcc matrix.c -o matrix
```

You should be able to compile the same source with both **gcc** as Paraguin. Run the sequential matrix using a command like this:

```
./matrix input3 > output.seq
```

Run the command:

```
wc output.seq
```

Take a screenshot of the commands in this task to include in your submission document. These can be in one screenshot or you can take multiple screenshots.

Task 3 – Run the parallel version

Create a job submission file for this program and run the program using the the number of processors in the range: 1, 4,8, 16, 32. Rename the output files to **`output.par.1`**, **`output.par.4`**, **`output.par.8`**, etc. After each run, execute the command:

```
diff output.seq output.par.<P>
```

where **`<P>`** is the number of processors used.

As with Assignment 2, if your program is implemented correctly, it should output the same answers as the sequential version of the program. If so, then the result of the **`diff`** command should be only 4 lines that look something like Figure 3.

```
1c1
<elapsed_time= 0.332111 (seconds)
---
>elapsed_time= 0.298292 (seconds)
```

*Figure 3: Example output from **`diff`** command*

Record the elapse times for the parallelized program running on the different number of processors as well as the elapse time for the sequential version.Create graph of the execution times compared with sequential execution and the speedup curve with linear speedup, just as you did in Assignment 2. Make sure that you provide axes labels, a legend (of there is more than one line) and a title to the graphs. Include copies of the graphs in your submission document.

What to submit

Your submission document should include the following:

1) A copy of your matrix multiplication source program;
2) A copy of the job submission file
3) Screenshot from compiling your matrix multiplication program with **`gcc`**, running the sequential version of the program, and the result of the **`wc`** command (this can be done in one screenshot);
4) Screenshot of compiling your matrix multiplication program with Paraguin;
5) Screenshot of running the parallel version of the matrix multiplication program (**`qsub`** and **`qstat`**);
6) Screenshot of running the **`diff`** command on the output files;
7) And copies of your graphs.

Sobel Edge Detection (UG – 39% ; G – 35%)

Edge Detection is the process of determining where in a photograph the contrast between shades changes dramatically enough to represent the edge between one object in the photograph and another. This is very useful for determining the content of images in photographs, such as facial recognition. For example, Figure 4 shows an image in grayscale and Figure 5 shows the result of applying Sobel Edge Detection to the image.

Figure 4: Example PGM File in Grayscale

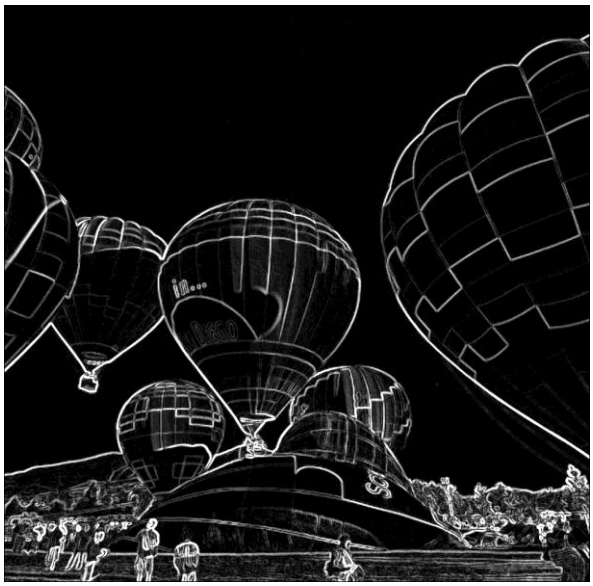

Figure 5: Result of Edge Detection Applied to Figure 4

Task 1 – Setup Up

First you will need some input. Copy the files /home/cferner/working/work1/sobel/*.pgm to your directory. These files contain grayscale images in PGM format with 1000x1000 pixels. Feel free to make your own input files if you have the capability to create PGM format image files. Make sure that you convert the images to grayscale as well as crop or scale it to be a square image. Also, you will either have to convert the image to be 1000x1000 or change the input size in the program to match the number of pixels in your image.

(If anyone would like to convert the program to read other types of image files, by all means, do so. My only request is that you share that code with me so that I can use it.)

Task 2 – Create the program

Create a program using the skeleton program show in Appendix B. The program will read one of the PGM image files as input. It will then produce a PGM file of the edges (white edges on a black background).

You need to fill in the section label "TODO". These sections include:

1) Initialize the masks (GX and GY) in parallel;
2) Take a time stamp;
3) Broadcasting the input data to all the processors;
4) Specifying the partitioning of the for-loop nest;

5) Specifying the gather of partial results back to the master process;
6) Take another time stamp and print the elapsed time.

After you are able to successfully compile your edge-detection program with Paraguin, take a screenshot of the compilation for your submission document.

Also make a sequentially executed version of the program using **`gcc`**. Include a screenshot of this compilation in your submission document as well.

Run the sequential version of the program for the image files. Send the results for each execution into a file called **`<X>Edge.seq`**, where **`<X>`** is the name of the original image.

Run the command to execute the parallelized version of the edge-detection program using the number of processors in the range: 1, 4, 8, 12, 16, 20, 24, 28, 32. Send the output into files **`<X>Edge.par.<P>`**, where **`<P>`** is the number of processors. For each parallel output file, run the command:

```
diff <X>Edge.seq <X>Edge.par.<P>
```

If your program is implemented correctly, it should output the same answers as the sequential version of the program. If the **`diff`** command produces output that consists of many lines of numbers, then your parallelized edge-detection program is not producing the same answers as the sequential version. You need to figure out why not and fix it. When your parallelized program can produce the same output as the sequential version, include screenshots of the output of the **`diff`** commands.

You may also want to download and view the images to see that the program is able to detect the edges. Why does the rainbow not show up in the edge image?

Record the elapse times for the parallelized program running on the different number of processors. Create graphs like you did for the previous part of the execution times and speedup. Make sure that you provide axes labels, a legend (of there is more than one line) and a title to the graphs. Include copies of the graphs in your submission document.

What to submit

Your submission document should include the following:

1) A copy of your SobelEdge Detection source program;
2) Screenshot from compiling your program with **`gcc`**;
3) Screenshot of compiling your program with Paraguin;
4) Screenshot of running the **`diff`** command on the output files;
5) And copies of your graphs.

(Required for Graduate Students; Extra Credit (8 points) for Undergraduates

Monte Carlo Estimation of π (UG – 8% extra credit; G – 8%)

Using the sequential program to estimate PI using the Monte Carlo algorithm, create a program that can be parallelized using the Paraguin compiler. To do this, you will need:

1) Execute a Barrier and take a time stamp;
2) Broadcast the number of iterations processors should use;
3) Run the creation of points within the square in a parallel region;
4) Gather the partial results back to the master process;
5) Print the result and elapse time from the master.

Compile this program, create a job submission file, and run the parallel program on 4, 8, 16, and 32 processors. Record the results and create graphs of the results. You should have a graph showing the error as a function of the number of processors, and you should have a graph showing the error as a function of the elapsed time.

What to submit

Your submission document should include the following:

1) A copy of the Monte Carlo source program;
2) Screenshot from compiling your Monte Carlo program with Paraguin;
3) Screenshot from compiling your matrix multiplication program with **gcc**;
4) And copies of your graphs.

Appendix A – Matrix Multiplication Skeleton Program

```
#define N 512

#ifdef PARAGUIN
typedef void* __builtin_va_list;
extern int MPI_COMM_WORLD;
extern int MPI_Barrier();
#endif


#include <stdio.h>
#include <math.h>
#include <sys/time.h>


print_results(char *prompt, float a[N][N]);
```

```
int main(intargc, char *argv[])
{
    int i, j, k;
    float a[N][N], b[N][N], c[N][N];
    char *usage = "Usage: %s file\n";
    FILE *fd;
    double elapsed_time, start_time, end_time;
    struct timeval tv1, tv2;


    if (argc < 2) {
        fprintf (stderr, usage, argv[0]);
        return -1;
    }


    if ((fd = fopen (argv[1], "r")) == NULL) {
        fprintf (stderr, "%s: Cannot open file %s for reading.\n",
                                                    argv[0], argv[1]);
        fprintf (stderr, usage, argv[0]);
        return -1;
    }


    // Read input from file for matrices a and b.
    // The I/O is not timed because this I/O needs
    // to be done regardless of whether this program
    // is run sequentially on one processor or in
    // parallel on many processors. Therefore, it is
    // irrelevant when considering speedup.

    for (i = 0; i< N; i++)
        for (j = 0; j < N; j++)
            fscanf (fd, "%f", &a[i][j]);

    for (i = 0; i< N; i++)
        for (j = 0; j < N; j++)
            fscanf (fd, "%f", &b[i][j]);

#ifdef PARAGUIN
    ;
    #pragma paraguin begin_parallel
        // This barrier is here so that we can take a time stamp
        // Once we know all processes are ready to go.
        MPI_Barrier(MPI_COMM_WORLD);
    #pragma paraguin end_parallel
#endif


    // Take a time stamp
    gettimeofday(&tv1, NULL);
```

```
    ;
    #pragma paraguin begin_parallel

    // Broadcast the input to all processors.  This could be
    // faster if we used scatter, but Bcast is easy and scatter
    // is not implemented in Paraguin
```

TODO: Add a pragma to Broadcast the input matrices a and b.

TODO: Add a pragma to run the following for loop partitioning the outermost (i) loop

```
    // Parallelize the following loop nest assigning iterations
    // of the outermost loop (i) to different partitions.

    // We need to gather all c[i][j].  However, array reference
    // one is inside the k loop.  If we put in an empty gather
    // then we'll have N copies of each c[i][j] send to the
    // master.  To send just one, then we use k = 0.
```

TODO: Add a pragma to gather the results. The gather should specifythat k should be equal to zero. In other words, the expression should be k=0.

TODO: Add code to implement matrix multiplication (C=AxB). C_{i}{j} = SUM{k=0..N } A_{i}{k} x B_{k}{j}

```
    ;
    #pragma paraguin end_parallel


    // Take a time stamp.  This won't happen until after the master
    // process has gathered all the input from the other processes.
    gettimeofday(&tv2, NULL);

    elapsed_time = (tv2.tv_sec - tv1.tv_sec) +
                        ((tv2.tv_usec - tv1.tv_usec) / 1000000.0);

    printf ("elapsed_time=\t%lf (seconds)\n", elapsed_time);


    /*print result*/
    print_results("C = ", c);

}



print_results(char *prompt, float a[N][N])
{
    int i, j;

    printf ("\n\n%s\n", prompt);
```

```
    for (i = 0; I < N; i++) {
        for (j = 0; j < N; j++) {
            printf(" %.2f", a[i][j]);
        }
        printf ("\n");
    }
    printf ("\n\n");
}
```

Appendix B – Sobel Edge Detection Skeleton Program

```
#ifdef PARAGUIN
typedef void* __builtin_va_list;
extern int MPI_COMM_WORLD;
extern int MPI_Barrier();
#endif

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <ctype.h>
#include <sys/time.h>

#define N 1000

static int clamp(int val) {
    if (val < 0)
        val = 0;
    else if (val > 255)
        val = 255;
    return val;
}

int main(int argc, char **argv) {

    FILE *inFile, *oFile;
    int grayImage[N][N], edgeImage[N][N];
    char type[2];
    int w, h, max;
    int r, g, b, y, x, i, j, sum, sumx, sumy;

    int GX[3][3], GY[3][3];
    double elapsed_time;
    struct timeval tv1, tv2;
    int tID;
    int c;
    int __guin_p = 0;
    char buffer[BUFSIZ];
```

TODO: This section that initializes the GX and GY arrays should be inside a parallel region

```
/* 3x3 Sobel masks. */
GX[0][0] = -1; GX[0][1] = 0; GX[0][2] = 1;
GX[1][0] = -2; GX[1][1] = 0; GX[1][2] = 2;
GX[2][0] = -1; GX[2][1] = 0; GX[2][2] = 1;

GY[0][0] =  1; GY[0][1] =  2; GY[0][2] =  1;
GY[1][0] =  0; GY[1][1] =  0; GY[1][2] =  0;
GY[2][0] = -1; GY[2][1] = -2; GY[2][2] = -1;



inFile = fopen(argv[1], "r");
if (inFile == NULL) {
    fprintf (stderr, "Open failed\n");
    fprintf (stderr, "Usage: %s target\n", argv[1]);
    return EXIT_FAILURE;

} else {

    // Read the type

    // Some graphics programs put comments in the file
    // If we find a #, throw away the line
    do
        fgets (buffer, BUFSIZ-1, inFile);
    while (buffer[0] == '#');

    if (buffer[0] != '#')
        sscanf (buffer, "%s", type);
    else type[0] = '\0';


    // Read the width and height

    // Some graphics programs put comments in the file
    // If we find a #, throw away the line
    do
        fgets (buffer, BUFSIZ-1, inFile);
    while (buffer[0] == '#');

    if (buffer[0] != '#')
        sscanf (buffer, "%d%d", &w, &h);
    else w = h = max = 0;


    // Read the max

    // Some graphics programs put comments in the file
    // If we find a #, throw away the line
    do
        fgets (buffer, BUFSIZ-1, inFile);
    while (buffer[0] == '#');

    if (buffer[0] != '#')
        sscanf (buffer, "%d", &max);
```

```
    else w = h = max = 0;
}

if (type[1] != '2'){
    fprintf (stderr, "%s is not a valid pgm file", argv[1]);
    return EXIT_FAILURE;
}

// Finally, read the pixels

for (i=0; i < N; i++){
    for (j=0; j < N; j++){

        fscanf (inFile, "%d", &grayImage[i][j]);
    }
}
fclose (inFile);
```

TODO: Create a parallel region to implement a Barrier. Then take a time stamp. (See the matrix multiplication skeleton program for an example.)

TODO: Add a pragma to Broadcast the input data: grayImage, w, and h.

TODO: Add a pragma to run the following for loop partitioning the outermost (x) loop. Note that there are four nested loops.

TODO: Add a pragma to gather the results. You want to gather the 5th (4) array reference (which is edgeImage[x][y]. **All values of x and y should be used in the gather. Therefore, the expression in the gather should be 0=0.**

```
for (x=0; x < N; x++){
    for (y=0; y < N; y++){
        sumx = 0;
        sumy = 0;

        // handle image boundaries
        if(x==0 || x==(h-1) || y==0 || y==(w-1))
            sum = 0;

        else{
             //x gradient approx
            for (i = -1; i <= 1; i++)  {
                for (j = -1; j <= 1; j++){
                    sumx += (grayImage[x+i][y+j] * GX[i+1][j+1]);
                }
            }

            //y gradient approx
            for (i = -1; i <= 1; i++) {
                for (j = -1; j <= 1; j++) {
                    sumy += (grayImage[x+i][y+j] * GY[i+1][j+1]);
                }
            }
```

```
                //gradient magnitude approx
                sum = (abs(sumx) + abs(sumy));
            }
            edgeImage[x][y] = clamp(sum);
        }
    }
```

TODO: End the parallel region, take a time stamp, and print the elapsed time. (See the matrix multiplication skeleton program for an example.)

```
    oFile = fopen(argv[2], "w");
    if(oFile != NULL){

        fprintf(oFile, "P2\n%d %d\n%d\n", w,h,max);
        for (i = 0; i < h; i++){
            for (j = 0; j < w; j++){
                fprintf (oFile, "%d ", edgeImage[i][j]);
            }
            fprintf (oFile, "\n");
        }
        fclose (oFile);
    }
    return EXIT_SUCCESS;
}
```