Assignment 2

Using Paraguin to Create Parallel Programs

C. Ferner and B. Wilkinson
Minor clarification Oct 11, 2013

Overview

The goal of this assignment is to use the Paraguin compiler to create parallel solutions using MPI to run on a distributed-memory system.

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).

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

#include <stdio.h>

int __guin_rank = 0;

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

      printf("Master process %d starting.\n", __guin_rank);

      #pragma paraguin begin_parallel

      gethostname(hostname, 255);
      printf("Hello world from process %3d on machine %s.\n",
                                        __guin_rank, hostname);

      #pragma paraguin end_parallel

      printf("Goodbye world from process %d.\n",
                                                __guin_rank);

      return 0;
}
```

Figure 1: Hello World Program

Compiling

Compile this program with the Paraguin compiler using this command:

```
scc -DPARAGUIN -D__x86_64__  -cc mpicc hello.c -o
hello.out
```

This command should all be on one line. Take a screenshot of the output from compiling your program to include in your submission document.

If the program is correct, the compiler will produce a file called **hello.out**. You can also see how the compiler parallelizes your program by producing a source file with MPI commands. (This is not required.) To do this, use the command:

```
scc -DPARAGUIN -D__x86_64__  hello.c -.out.c
```

This will create the file **hello.out.c**, which is the MPI version of the hello world program. You are free to inspect and even modify it. You can compile this file to an executable using **mpicc** to produce a file call **hello.out.**

Running

Create a job description file to submit your program to the SGE scheduler with **qsub**. To execute the program, you need to submit a job to the Sun Grid Engine (SGE) job scheduler. To do this, you will need to create a file call **hello.sge** with the contents shown in Figure 2.

```
#!/bin/sh
#
# Usage: qsub hello.sge

#$ -S /bin/sh

#$ -pe orte 4           # Specify how many processors we
want

# -- our name ---
#$ -N Hello           # Name for the job
#$ -l h_rt=00:01:00   # Request 1 minute to execute
#$ -cwd               # Make sure that the .e and .o file
arrive in the working directory
#$ -j y                     # Merge the standard out and
standard error to one file

mpirun -np $NSLOTS ./hello.out
```

Figure 2: Job Submission File

The first line of this script indicates that this is a shell script. The line **`#$ -pe orte 4`** indicates to SGE that you want 4 processing elements (processors). The line **`#$ -N Hello`** indicates that the output files should be named "HelloXXX", where XXX contains other characters to indicate what it is and the job number. The line **`#$ -cwd`** tells SGE to use the current directory and the working directory (so that you output files will go in the current directory). The line **`#$ -j y`** will merge the stdout and stderr files into one file called "Hello.oXXX", where XXX is the job number. Normally, SGE would create separate output files for the stdout and stderr, but this isn't necessary.

The line **`#$ -l h_rt=00:01:00`** tells SGE to kill the job after a minute. This helps to keep the system clean of old jobs. If you find that a job is still in the queue after it has

```
Master process 0 starting.
Hello world from process 0 on machine compute-0-0.local.
Hello world from process 1 on machine compute-0-0.local.
Hello world from process 2 on machine compute-0-0.local.
Hello world from process 3 on machine compute-0-0.local.
Goodbye world from process 0.
```

Figure 3: Hello world output

finished, you can delete it using **`qdel #`**, where # is the job number. If you expect your job to take longer than a minute to run, you will need to increase this time.

The last line of the submission file tells SGE to run the hello program using MPI run and the same number of processors as indicated in the **`#$ -pe orte 4`** line above.

To submit your job to the SGE, you enter the command:

```
qsub hello.sge
```

You job is given a job number, which you will use for other SGE command. The output of your program will be send to a filed called Hello.oXX, where XX is the job number. There will also be a file called Hello.poXX, which you don't need. ***You will want to delete the output files you don't need. Otherwise, your directory will fill up.***

To see the list of jobs that have been submitted and have not yet been terminated, use the command **`qstat`**. If you find that a job is still in the queue after it has finished, you can delete it using **`qdel #`**, where # is the job number.

After your job has completed, you should get output in the file Hello.oXX, with contents that should look something like that of Figure 3.

Re-run this program using different numbers 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) A copy of your SGE job submission file;
5) Screenshot from compiling your hello world program with Paraguin;

6) Screenshot of submitting your job to the SGE schedule and output of qstat showing your job queued up;
7) Screenshot of the output running on the largest number of processors for which you are able to run successfully the program.

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

Task 1 – Create the program

In this part of the assignment, you will be implementing the matrix multiplication algorithm using the Scatter/Gather pattern. The Scatter/Gather pattern is actually done as a template instead of a single **pragma** construct.

First you will need some input. Copy the following files from the directory **/home/cferner/working/work1/**into your directory:

- **input2x512x512Doubles**
- **output512x512mult**

The first file contains *two* 512x512 matrices of a floating point numbers that you will use as input. The other file contains the answers of applying matrix multiplication to the two 512x512 matrices in the input file. You will use these files to determine if your programs are producing the correct output.

Create a program using the skeleton program from Appendix A. You need to fill in the sections labeled "TODO". You will need to implement the algorithm to perform matrix multiplication. In order to perform matrix multiplication, you will need to implement in parallel the following formula:

$$C_{i,j} = \sum_{k=0}^{N-1} A_{i,k} \times B_{k,j} \quad \text{for all} \quad i, j \in [0 \ldots N) \qquad (2)$$

This will require 3 nested for loops. The outermost loop should be a **forall** loop.

Since each value in the resulting matrix is a dot product of the rows of A with the columns of B, you won't be able to simply scatter the B matrix. The reason is because you are computing an entire row of the C matrix. To do this, you will need EVERY column of the B matrix; in other words the entire matrix. So you will need to broadcast the B matrix instead of scattering it.

After you calculate the product, you will need to gather the partial results back onto the master process. After the scatter/broadcast, computation, and gather, the master process will take another time stamp (to calculate the time to perform matrix addition), then print the answer.

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

Task 2 – Run the parallel version

Create a job submission file for this program. *You will need to add the name of the input file, input2x512x512Doubles, as another command line argument after your program on the last line of the submission file.* Run the program using the number of processors in the range: 1, 4, 8, 12, 16, 20, 24, 28, and 32. Rename the output files to **`output.mult.1`**, **`output.mult.4`**, **`output.mult.8`**, etc. After each run, execute the command:

```
diff output512x512mult output.mult.<P>
```

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

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 4. Use the elapsed times to create a graph of the runtimes as a function of the number of processors. In other words, you should have a graph with number of processors on the x-axis and seconds on the y-axis. Create a second graph of speedup. Calculate the speedup using the single processor time as the sequential time (which is actually not the same, but we'll use it). You should also have on this graph the ideal speedup curve for comparison.

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

*Figure 4: Example output from **`diff`** command*

What to submit

Your submission document should include the following:

1) A copy of your matrix addition source program;
2) A copy of the job submission file
3) Screenshot of compiling your program with Paraguin;
4) Screenshot of running the parallel version of the program (**`qsub`** and **`qstat`**);
5) Screenshot of running the **`diff`** command on the output files;

6) And copies of your graphs.

Heat Distribution (UG – 39% ; G – 35%)

In this part of the assignment you will be using a stencil pattern to model the heat distribution of a room with a fireplace. Although a room is 3 dimensional, we will be simulating the room with 2 dimensions. The room is 10 feet wide and 10 feet long with a fireplace along one wall as depicted in Figure 5. The fireplace is 4 feet wide and is centered along one wall (it takes up 40% of the wall, with 30% of the walls on either side). The fireplace emits 100º C of heat (although in reality a fire is much hotter). The walls are considered to be 0º C. The boundary values (the fireplace and the walls) are considered to be fixed temperatures.
Using a Jacobi Iteration, the heat distributed is calculated to look something like that of Figure 6. Each value is computed as an average of its neighbors. There needs to be two copies of the matrix. The newly computed values need to be stored into the second matrix; otherwise the values being computed would not be based on the same values in the previous iteration. Once all the new values are computed, the newly computed values can replace the values in the last iteration.

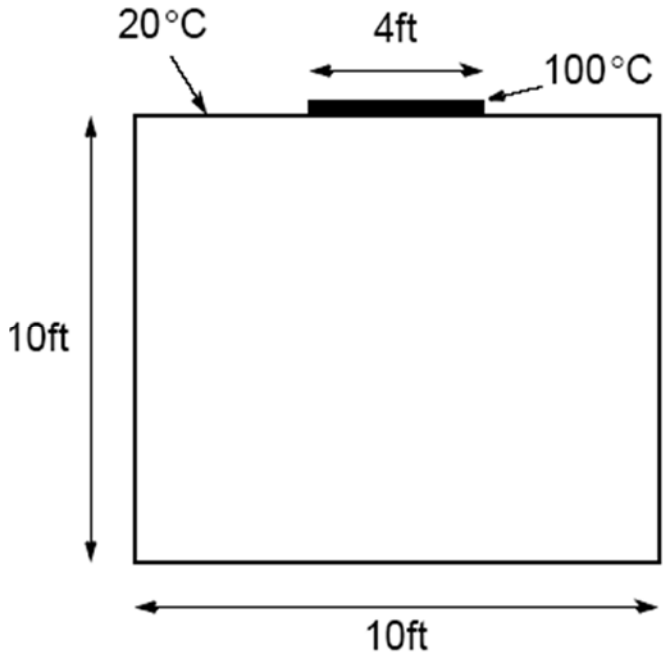


Figure 5: 10x10 Room with a Fireplace

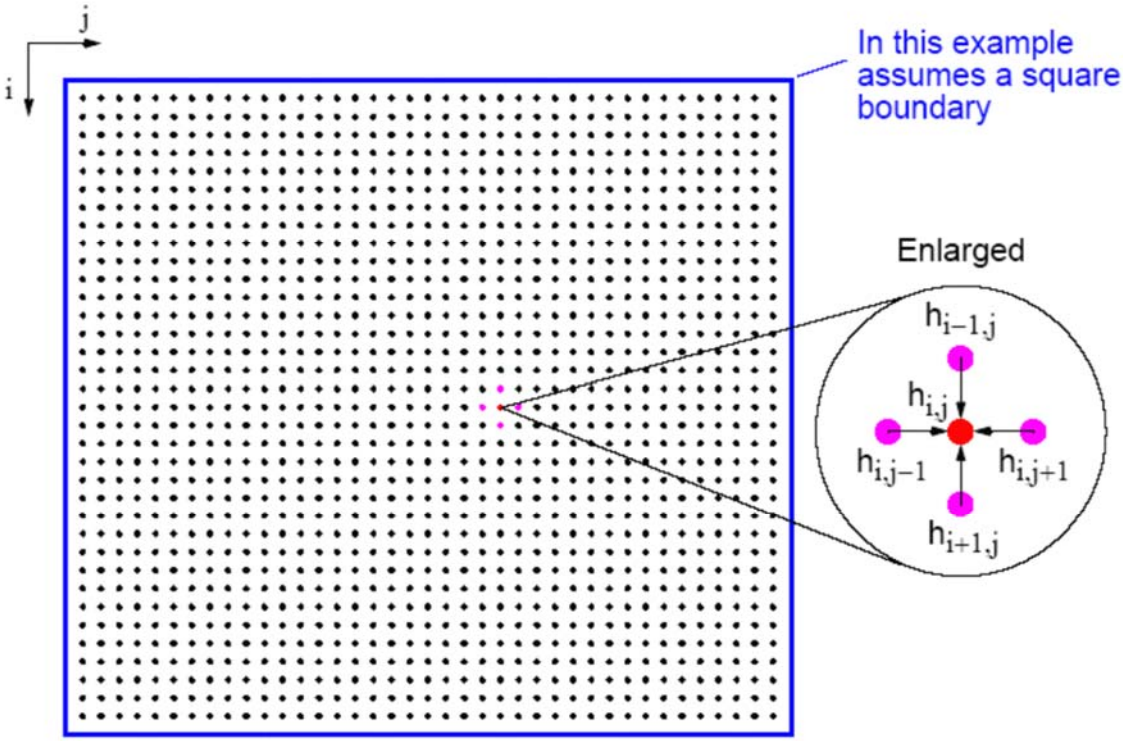
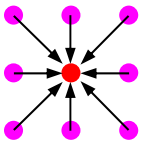


Figure 6: Jacobi Iteration

Create a program to use the Paraguin stencil pattern to compute the heat distribution of the room described above. The data should be a 3 dimensional array, where the size of the 1st dimension is 2, and the sizes of the 2nd and 3rd dimensions is how ever many points you decide to use. The number of points for the room should be large (more than 100x100). The more points there are, the smoother the simulation of the spread of heat.

1) Initialize the values in both copies of the room all zeros (degrees Celsius) except for 40% of one wall centered where the values will be 100.
2) Using your matrix program as an example, set up a barrier and take a time stamp.
3) Use the Paraguin stencil **pragma** to indicate a stencil pattern
    a. The number of iterations should be at least 5000 in order for the data to converge. You can try larger and smaller values for the iterations to see if the data changes or not. You would like the number of iterations to be just large enough that the computed values do not change by significant sizes.
    b. The compute function should be a function to calculate the average of the value at location i and j and its nearest neighbors. Unlike the averaging showed in Figure 6, this would be an average of 9 values as shown below:



(The average should include the value at position i, j as well as the 8 neighbors.) The reason for using the value in position i, j as part of the average for the new value for position i, j is to reduce the oscillations.

4) Take another time stamp and calculate and report the elapsed time.
5) Print the final values of the matrix

As with the matrix programs, compile your program with Paraguin and take a snapshot of this process. Create a job submission file to execute your program on 1, 4, 8, 12, 16, 20, 24, 28, and 32 processors. The output should go into files named: **`output.heat.1`**, **`output.heat.4`**, **`output.heat.8`**, etc. You should run the **`diff`** command on these files against the first one to make sure each program produces the same output. Include snapshots of these **`diff`** commands in your assignment submission document.

Also create a graph of the final data. Figure 7 and Figure 8 show two examples of graphing the final data. The first was created using X11, and the second is a surface graph using Microsoft Excel. See the course home page for help with creating graphics using X11.

Figure 7: Graph of Heat Distribution from X11

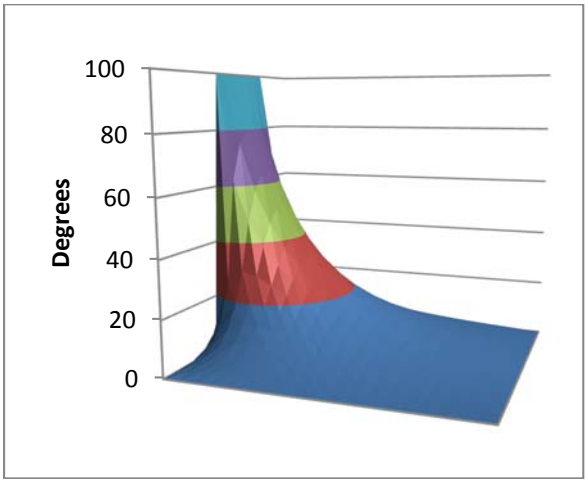


Figure 8: Graph of Heat Distribution from Excel

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

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 and count the number of points within the semicircle;
4) Reduce (as a summation) the count back to the master;
5) Print the result, error, and elapsed time from the master. (The error can be computed as the absolute difference between the value you computed and the true value of π. You can use the value 3.1415926535 as the true value of π.

Compile this program, create a job submission file, and run the parallel program on 4, 8, 12, 16, 24, 28, 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) A copy of your job submission file;
4) Screenshot of the results from running your program;
5) And copies of your graphs.

Appendix A – Matrix Multiplication Skeleton Program

```
#define N 512

#ifdef PARAGUIN
typedef void* __builtin_va_list;
#endif


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

int __guin_rank = 0;

void print_results(char *prompt, double a[N][N]);


int main(int argc, char *argv[])
{
    int i, j, error = 0;
    double a[N][N], b[N][N], c[N][N];
    char *usage = "Usage: %s file\n";
    FILE *fd;
    double elapsed_time;
    struct timeval tv1, tv2;

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

TODO: Broadcast the error have have all processors exit is there is an exit

```
    // Read input from file for matrices a and b.

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

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

    fclose(fd);
```

```
    #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.
    #pragma paraguin barrier
    #pragma paraguin end_parallel

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

TODO: Start a parallel region
TODO: Scatter/Broadcast the input
TODO: Create a loop nest to compute the matrix multiplication of matrices a and b and store the result in matrix c. The outermost loop should be executed in parallel.
TODO: Gather the partial answers
TODO: End the parallel region

```
    // 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);

}



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

    #pragma paraguin begin_parallel
    printf ("\n\n%s\n", prompt);
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            printf(" %.2lf", a[i][j]);
        }
        printf ("\n");
    }
    printf ("\n\n");
    #pragma paraguin end_parallel
}
```