# CSc 422 — Parallel Programming Project

Programs due Thursday, March 10 (by midnight)
Reports due Thursday, March 31 (in class)

In this project you will develop efficient parallel programs for a grid computation, conduct timing experiments to analyze the performance of your programs, and write a report describing your results and what you have learned. The project is worth 60 points. I will award up to 6 bonus points (10%) for outstanding projects and reports.

You may work with a classmate on this project, and I encourage you to do so. However, you may do this project on your own if you prefer. Be sure to read and understand the entire assignment before you begin programming. There are lots of details, and they all matter!

Your programs must be turned in electronically by midnight on March 10. You will be doing timing experiments the week after Spring break. I will set up a schedule for the use of Parallel; each group will get several hours of stand-alone testing time. Your reports are due the second Thursday after Spring break.

## Programs

The starting point for the project is the following four programs. You may write the programs in MPD or in C with the Pthreads library. Develop your programs on Lectura or at home, but run experiments on Parallel.

1. Sequential Jacobi iteration program
2. Parallel Jacobi iteration program
3. Sequential multigrid program
4. Parallel multigrid program

Jacobi iteration is described in Sections 11.1.2 and 11.1.3 of the text. An outline of a sequential program is given in Figure 11.2; an outline of a parallel program is given in Figure 11.3. You will need to convert these to MPD or C plus Pthreads and to fill in all the details.

Multigrid methods are described in Section 11.1.6, but I do not give actual code. I will cover the algorithm in class; see the accompanying handout for information that should help you develop the multigrid programs.

Your task is to write *efficient* programs. See the discussion on pages 539-40 of the text for the kinds of programming "tricks" that help make programs fast.

Initialize the boundary points of the grids to 1.0 and the interior points to 0.0. This will make it easy for you to check the correctness of your programs and the quality of the results for different algorithms, because the final values for all interior points should be 1.0.

For the parallel programs, divide the grid into horizontal *strips* of rows of points. This includes all the grids in the multigrid programs. Use one worker process per strip. Each strip should contain about the same number of rows so as to balance the computational load.

In the parallel programs, implement an efficient *dissemination* barrier and use it when you need barrier synchronization. Use counter variables and busy waiting, as described at the end of Section 3.4. *Make sure your barrier is correct!*

For the multigrid programs, use a four-level V cycle as illustrated in Figure 11.8. Use the restriction and interpolation operators described on pages 550-51 of the text. Use Jacobi iteration for the iterations on each level. Use exactly *four* iterations on each of the finer grids, and use the command-line argument `numIters` (see below) for the number of iterations on the coarsest (smallest) grid.

**Input and Output**

Your programs should have three command-line arguments *in the following order*:

`meshSize` — the mesh size of the finest (largest) grid
`numIters` — the number of iterations to use on the finest grid
`numWorkers` — the number of worker processes for the parallel programs

Assume that all grids are square. You may assume that `meshSize` is a multiple of 16 times the number of workers. (This simplifies restriction and interpolation in the multigrid programs.) You should check this in your main function and exit with an error message if this is not the case.

For the parallel Jacobi iteration program, the number of *interior* rows in the grid is `meshSize-1`, and the *total* number of rows counting the top and bottom boundaries is `meshSize+1`. This means that each worker is responsible for `meshSize/numWorkers` rows, except for the last worker which has one less row.

For the multigrid programs, `meshSize-1` is the number of interior rows in the *finest* (largest) grid. The mesh sizes of the coarser (smaller) grids should thus be `meshSize/2`, `meshSize/4`, and `meshSize/8`. As illustrated in Figure 11.7, the physical boundaries of all grids are the same, but the mesh size varies, and hence the distance between points also varies.

The output from your programs should be:

the command-line arguments
the execution time, in seconds, for the computational part
the maximum error in final values on the finest (largest) grid
the final values on the finest (largest) grid

Write the first three items to standard output. Write the data values to file `data.out`. The final data values are mainly for your purposes when debugging the program. (You may wish to output additional values while developing your programs, such as subsets of the rows in various grids.)

To calculate the execution time of the computational part, read the clock *after* you have initialized all variables. You should initialize grids *in parallel* in the parallel programs. Hence, you should have a barrier at the end of initialization (see Figure 11.3), and you should read the clock right after this barrier. It is fine to have each worker read the clock; because of the barrier, they will all read about the same value.

Read the clock again after you have finished the iterations and the calculation of the maximum error, but *before* you write the output. Again, make sure there is a barrier before you read the clock the second time.

The maximum error in final values should be the maximum difference between the final values of points on the finest (largest) grid and 1.0. One does not normally know what the final values should be, of course, but the maximum error is more interesting than the value of `epsilon` described in Section 11.1.

**Timing Experiments**

Your second task (during or after Spring break) is to run a series of timing experiments. In particular, you are to execute your programs for the following combinations of command-line arguments:

> program 1 for mesh sizes of 192 and 384
> program 2 for mesh sizes of 192 and 384 and for 1-4 worker processes
> program 3 for mesh sizes of 192 and 384 (largest grid)
> program 4 for mesh sizes of 192 and 384 and for 1-4 worker processes

There are a total of 20 different timing tests. *Run each timing test at least twice* to make sure you are getting results that are close to each other. If the two times are not close, then repeat the timing test until you are confident about the average execution time for that combination of command-line arguments.

For each *sequential* program and grid size, first figure out what the value of `numIters` should be so that the execution time of the program is about 30 seconds for that grid size. Then use the *same* value of `numIters` for the parallel versions of that sequential program. You will thus be using four different values for the `numIters` argument.

If you write your programs in MPD, use the `age()` function to calculate execution times. The return value from each call of `age()` is the time in *milliseconds* since the program began execution. Hence, you will need to divide the elapsed time by 1000 to convert it to seconds. Be sure to set the `MPD_PARALLEL` environment variable to the appropriate number of processors for each test. Do not set `MPD_PARALLEL` just once; change its value every time you change the number of processors you are using in a test.

If you write your programs in C and Pthreads, use the `times` function, as illustrated in the `clock.c` and `matrix.mult.c` programs I handed out in class. (You can view the man page by executing "`man -s 2 times`".) The return value from `times` is the number of "clock ticks" since some time in the past (when the OS was last booted). The value of `CLK_TCK` is 100, hence you will need to divide the elapsed time by 100 to convert it to seconds.

**Compiler Optimization Experiments**

Compilers such as `gcc` and `mpd` by default produce code that is correct but not terribly efficient. These compilers will produce much better code if you turn on optimizations. With `mpd`, this is done using the `-O` flag. With `gcc`, you have three choices: `-O`, `-O2`, and `-O3`, where `-O3` is the highest level of optimization.

In addition to doing the basic timing tests described above, you are also to determine how much faster your programs are if you turn on compiler optimizations. In particular, compile your programs with `-O` for `mpd` or with `-O3` for `gcc`. Then repeat some or all of the timing tests. You do not need to repeat all 20 timing tests, but you do need to repeat enough of them to be able to reach conclusions about how much improvement one can get by using compiler optimizations.

**Important Point**: If you use C and Pthreads, be sure to declare shared synchronization variables such as barrier flags as `volatile` variables. With the `-O3` level of optimization, `gcc` tries to put frequently used variables in registers. This is fine for sequential programs, but it usually breaks parallel programs. In particular, if a barrier flag is put in a register, then a change to the flag made by one process will not be seen by any other process!

## Reports

Once you have done all the timing tests, write a report to explain what you have done and what you have learned. Your report should be a few pages of text plus tables and figures. It should have five sections, as follows:

- *Introduction.* Give a brief introduction to your report and a brief overview of your results.
- *Programs.* Summarize how each program works, and describe in reasonable detail all the program-level optimizations you have implemented to make your programs fast. Describe all changes you made to the programs that you turned in on March 10 and why you made those changes.
- *Timing Experiments.* Present the results from the timing experiments. Use tables to present the raw data and graphs to show speedups and comparisons. You do not have to use a graph-drawing program; it is fine to draw graphs by hand if you wish. *Also explain your results.* What do the results show? Why?
- *Compiler Optimization Experiments.* Explain the experiments you conducted to measure the effect of turning on compiler optimizations and the results you observed.
- *Conclusion.* Briefly summarize what your report has shown, *and describe what you have learned from this project.*

## Electronic Turnin and Reports

By midnight on March 10 use `turnin` to submit your programs. The assignment name is `parallel`. The programs should be named `jacobi.seq`, `jacobi.par`, `multigrid.seq`, and `multigrid.par`. Also submit a Makefile that we can use to compile your programs. In particular, if we execute

```
make jacobi.seq
```

your makefile should compile the sequential Jacobi iteration program and produce an executable file that resides in `a.out`. We should then be able to execute the program with the command-line arguments specified above. You might also wish to turn in a README file.

We expect you to have completed efficient, working versions of your four programs by March 10. However, when you run the timing experiments, you may find that you made mistakes and hence need to modify your programs. Before you turn in your report on March 26, turn in final versions of your four programs and Makefile. Use the same assignment and program names. (We will make a backup copy of the turnin area on March 11, and we will look at the differences between the original and final versions of your programs.)

Your reports are due *in class* on March 30. Please append commented listings of the final versions of your four programs to your report. If you changed your programs, *your report should also explain what you changed and why.* You do not need to turn in the actual output from any of your tests, but you should have it available or readily be able to reproduce it. In short, your report should contain all the information someone else would need to reproduce your results.