Parallel Computing for Partial Differential Equations — Matthias K. Gobbert
Sommersemester 2012 — Universität Kassel
Homework 4 — due on Wednesday, May 02, 2012

General note: This project is designed to study the overall process of taking a mathematical algorithm and implementing it as a parallel program. For convenience and clarity, the process is separated into different problems, but only one code (consisting of several files) and one set of runs is needed. In addition, this homework introduces the use of a Makefile.
I will post a file `ver0.0suppliedfiles.tgz` along with this lab that expands into a directory with the name `ver0.0suppliedfiles`; it contains the needed files to get started on this project.
One of these files is a Matlab code of the numerical method discussed below that you may use as a guide for creating your code; this approach reflects the approach of testing a method in Matlab first and then using its code as template for the parallel code. Additionally, a Matlab driver file is enclosed that you may want to run to create an example that you can compare to the results of your code for debugging.
A Makefile is supplied that can be used to create the executable `power`. This file assumes that you use the GNU compiler gcc and includes some useful compile options in `CFLAGS`. It also links to the math library with loader flag `-lm` in `LDFLAGS`. As you solve the problems below, you should add additional source and header files for your routines; you will have to add references to these new files in the `OBJS` list of the Makefile. It will of course be necessary to add include statements to your files in main.h and calls to your routines in main.c. However, no changes to the other supplied files should be necessary.
The code demonstrates the one-dimensional matrix storage scheme by column and its use, including basic improvements to efficiency such as the ordering of the inner and outer loops to access the matrix elements. (You could interchange this ordering and check for slower execution time.) Notice that the supplied main program also demonstrates the use of command-line arguments, so it actually has the correct calling sequence for the power method already, as `power n tol itmax` with integers `n` and `itmax` and real number `tol`. Also some error handling is already included for demonstration purposes.
Write a README file to document which files make up your code. List which functions are contained in each file and what their purpose is. Provide a snippet of Linux command line that demonstrates that all of your code compiles cleanly; use `make clean` to delete all object files first to force a recompile of everything. Also submit your code to me in the form of a tgz-file, similar to the way in which I am making the files available to you. In order to ensure that I can distinguish the directories from different students, please incorporate your name into the name of the directory. Your code must compile cleanly without errors. Include instructions in the README file for how to run your code.

1. [4 points.] This problem asks you to write various auxiliary routines. For each part, explain how you implemented its solution in parallel, and how you tested your code for correctness.

   (a) The matrix $A \in \mathbb{R}^{n \times n}$ is split on the p parallel processes by blocks of n/p consecutive columns. On each process, its part of the matrix is stored as a one-dimensional vector in memory. See the setup routine `setup_example` for an example.

   Write an output routine that assembles a temporary copy of the matrix on Process 0 and outputs it to the screen. Such a routine is a useful utility to make sure that small sample matrices are set up correctly; of course, you will not use this routine in production runs involving large matrices. (If you want to use a function like this in production runs, you would output to a file instead, and you would not actually assemble the entire matrix before outputting.)

(b) Write a function to compute the dot product between two (column) vectors x and y. The data structure of such vectors has blocks of rows stored on the p processes. The result of the dot product should be available on all processes.

(c) Write a function to compute the Euclidean vector norm $\|x\|_2$ of a (column) vector x with blocks of its rows spread across the p processes. Its result should be available on all processes. (Hint: I suggest that you simply implement this function by calling the dot product function and taking the square root of the result.)

(d) Write a matrix-vector product routine that computes $y = Ax$ with a given matrix A and column vector x, both of which are stored as detailed above. The resulting vector y should have the same data structure as x, i.e., it should be spread across the p processes by blocks of consecutive rows.

2. [6 points.] The power method is used to compute approximations to the largest eigenvalue in magnitude and associated eigenvector of a matrix; see any book on Numerical Linear Algebra for more information. For your convenience, the power method is given in the Matlab file supplied; notice that the algorithm has been formulated to minimize the computation of matrix-vector products. Using the utility functions from the previous problem, program a parallel version of the power method. Use the standard initial guess for the eigenvector $x \in \mathbb{R}^n$ with components $x_j = 1/\sqrt{n}$ for all $0 \leq j < n$. Your power method function should return the eigenvalue approximation λ, the eigenvector approximation x, and the number of iterations taken. As a summary at the final time, print out the maximum number of iterations allowed, the chosen tolerance, the eigenvalue approximation λ, the norm of the residual $\|Ax - \lambda x\|_2$, and the number of iterations taken, using suitable format conversions for each. Are your results independent of the number of processes used? Explain your implementation and how you made sure that it as well as your results themselves are correct.

   For several values of n, including $n = 8192$, copy-and-paste the contents of `slurm.out` with properly formatted output of all significant variables.

3. [10 points.] Use the function `MPI_Wtime` to time execution of the power method alone, excluding all other parts of the code. This is an example of how it is possible to selectively time a particular function (your power method only) instead of the entire code, which is relevant if file I/O or other serial operations were involved in the execution of your code. Demonstrate the scalability of the method for several large matrix dimensions. Approach this task by first estimating the memory usage of your code, that is, count the large variables and compute how much memory the total code should use. Explain why you chose the selected matrix dimensions in your studies. For simplicity and to ensure that the number of processes divide the matrix dimension n, you may want to restrict yourself $n = 2^\nu$ with integer ν. Explain exactly how you obtained your timing results. Collect your timing results in the form of Table 1.1 of the tech. report HPCF–2010–2. (In order to have at least one case to compare for all of us, include $n = 8{,}192$ in your choices of values.)

   In addition to raw timing data in the form of a summary table, plot the observed speedup and the observed efficiency for several values of n. Consider figures in the tech. report as guide. You will notice that there are several choices for some data points, such as if data for 8 MPI processes should be from runs using 1, 2, 4, or 8 nodes. You may accept the conclusion of many studies here that using all cores on a node is preferred and plot like in Figure 4.5 of the tech. report HPCF–2010–2.