Introduction to Parallel Computing — Matthias K. Gobbert
Wintersemester 2011/2012 — Universität Kassel
Homework 3 — due on Tuesday, December 20, 2011

1. [20 points.] Consider the program for the trapezoidal rule in Chapter 4 of Pacheco.

   (a) Obtain the code, run it, and verify that it works properly for you. Discuss any issues that you feel are short-comings of this code and how to fix them.

   (b) This and the following parts of this problem are dedicated to applying some obvious fixes to Pacheco's original code.

   Improve the output of the program by also outputting the true value of the integral, the true error, the quantity h^2, the step size h, number of intervals n, and the number of processes p. These numbers can be used to check that the code is performing properly. For optimal readability, organize each output in a separate line and align the numbers by proper formatting. Make sure to choose a suitable format conversion for all numbers; why did you choose the format conversion that you did choose?

   (c) Program command-line input for the pertinent problem parameters a, b, and n. Notice that a and b are real numbers, while n is an integer.

   (d) Install error checking for all pertinent problems; in this context, consider n an input, whose validity needs to be checked. You should print an informative message from Process 0 and use the `MPI_Abort` function to abort with a non-zero error code.

   (e) Demonstrate that the code as given in Pacheco is not accurate, whenever the number of processes p does not divide the number of subintervals of the trapezoidal rule n. Explain this.

   Fix the program such that the number of processes p does *not* have to divide the number n of subintervals any more. In the cases, where p does not divide n, different processes will do different amounts of work. For best performance, the work should be divided as evenly as possible, of course. Here, the work is proportional to the number of subintervals that each process handles. Devise a scheme for load-balancing that ensures that the number of subintervals does not differ by more than 1 between any two processes. Explain the algorithm you implement, and argue that it works for all possible cases of n and p. If your code does not handle all possible cases, e.g., leaves out some trivial case, implement at least an error check.

   The final result of this change is a computer program, but I suggest that you use mathematical notation to explain and document your ideas. Introduce notation as required to do so. Assuming that you use Pacheco's program as a starting point, you only have to explain your changes to it. For testing purposes, you may want to have each process output a message like "Process `id`: `local_n` subintervals from node index `l_ia` to `l_ib`" to demonstrate that you are covering all nodes from 0 to n

   (f) Run your code with as many nodes as available and with different numbers of processes per node. Present your results in one table of the form of Table 1.1 of the tech. report HPCF–2010–2, that is, the rows reflect the number of processes per node and the columns the number of nodes. (The case of 6 processes per node is a special case for this tech. report only; you do not need to do that case.) Discuss the performance you observe. What steps have you taken to be sure that your timing results are accurate? What limitations of your code do you encounter, if any? Finally, remember to check the output of the code as set up earlier in the assignment to ensure that the code is computing correct results: *Code that computes incorrect answers is never acceptable!* Implement improvements to the code as needed.

Two notes on performance studies: (i) One big problem in parallel computing is communication time. Therefore, only wall clock time is an honest measure of whether your code performs well. This can simply be accomplished by using the MPI command `MPI_Wtime`, which saves a current time stamp in seconds. Notice that the time as such is meaningless (see its man-page) and that only differences of two such time stamps can be relied upon. To get the most conservative time, it is necessary to wait for the slowest process to complete its job, hence it is customary to force a synchronization of all processes with the `MPI_Barrier` command. Thus, the outline of code looks like

```
MPI_Barrier(MPI_COMM_WORLD);
startTime = MPI_Wtime();

/* ... code to time here ... */

MPI_Barrier(MPI_COMM_WORLD);
endTime = MPI_Wtime();
if (id == 0) { /* output from Process 0 only */
  tsec = endTime - startTime;
  printf ("elapsed wall clock time in seconds = %9.2f\n", tsec);
}
```

which prints the difference of the start and end time to `stdout`. (The above `printf` is just a starting point for your coding; you might need to work on the formatting of the output to get useful data for all cases.)

(ii) A practical aspect is how to organize the large number of studies that you want to do in a way that ensures that results do not overwrite earlier ones and such that you have a record saved in case of future questions. I have found that the only safe way is to have one directory for each run, which contains the job submission script and all other necessary files for each run. The only difference from one directory to the next are the job submission script's entries for `nodes` and `ppn`. A naming scheme for theses directories might thus be `nXXppnYY`, where `XX` = 01, 02, 04, etc. denotes the number of nodes and `YY` = 01, 02, 04, etc. the number of processes per node.

What to submit at minimum as part of your report: (1) There should only be one final version of code; include it in your report as listing. (2) Include the contents of one `qsub.out` in your report that shows the output of your code for $n = 8192$ in its formatted form. (3) Include a table of observed wall clock times in the format of a summary table (rows for processes per node, columns for number of nodes, as Tables 1.1 and 1.2 in the tech. report HPCF–2010–2). (4) Discuss the changes you made to the code and whether your final code is optimal.