CS 240A: Applied Parallel Computing // Homework 3

Assigned April 12, 2010

Due by 11:59pm Monday, April 26

You may do this homework in groups of two—in fact, I prefer that you do so. You may form groups however you want, but I encourage groups that have students from two different departments.

The object of this problem is to write a parallel program, using MPI, to use conjugate gradients (CG) to solve the finite difference discretization of Poisson's equation in two dimensions on a regular square grid of $n = k^2$ points. (This is also known as the "model problem.")

If we write the discretized problem as $Ax = b$, then

- A is the sparse n-by-n matrix (whose nonzeros are all 4's and -1's) representing the discretized operator. For this homework, you will *not* generate or store any of A explicitly.
- b is an n-vector containing the boundary conditions and any forcing terms. You will write a routine to generate b for debugging, and we will write one for testing and grading.
- x is an n-vector giving the answer.

The CG algorithm is outlined in the course slides for April 7 and April 12, and is described in detail in the references on the course resources page. There is a sequential Matlab code for CG linked to the course web page under Homework 3. You will need to write three routines: `DAXPY` (which adds a scalar multiple of one dense vector to another, $y = y + \rho z$); `DDOT` (which computes the inner, or dot, product of two dense vectors, $\sigma = y^T z$); and `MatVec` (which multiplies a specified dense vector by the matrix A, $y = Az$).

Your main CG program should be set up so that it never needs to know anything about how A is represented; only `MatVec` cares about that. In this case, because A has such a simple structure, `MatVec` does not contain a data structure for A at all; it just computes each element of y from the relevant elements of z directly. Therefore your program will not use any storage for A at all, only for the various dense vectors in the CG algorithm. However, your main program should still work correctly if a completely different `MatVec` were substituted.

Since all the data is in the vectors, the performance of the program depends on how the vectors are distributed among the processors and what communication you do. Think of each of the four or so vectors in CG as having one element for each grid point, and describe the data layout by describing how grid points are mapped to processors. You should experiment with at least two layouts, namely giving each processor a rectangular block of about k/p rows, and giving each processor an (approximately) square block of dimension about $k/\sqrt{p}$. You should also experiment with using various numbers of layers of "ghost cells", or grid points that are stored redundantly on adjacent processors. With l layers of ghost cells, you can perform l iterations of CG with no communication at all in `MatVec` (although there's still communication at every iteration in `DAXPY` and `DDOT`).

The communication with ghost cells is simpler for the rectangular block layout than for the square block layout, because in the square case for $l > 1$ each processor needs a few ghost cells from the diagonally adjacent blocks as well as the ones above, below, left, and right. If you wish, you may do your ghost cell experiments only for the rectangular case.

Our data generator will generate the vector b one element at a time (that is, you call it with i and it returns $b(i)$). Thus you can create the vector in whatever layout you want. In your experiments, of course you should only time the CG solve, not the data generation.

The tradeoff between computation and communication will be kind of subtle. See how thoroughly you can understand it. Your goals are (1) to make your program run as fast as possible for the largest problem you are able to run within a few minutes time, on any number of processors; and (2) to experimentally understand the tradeoffs well enough to say what layouts and ghost cell choices will be most efficient for various problems sizes and numbers of processors.

You should be able to do pretty large problems on a parallel machine; on my laptop, the sequential Matlab code takes about 2 minutes for a 1,000,000-by-1,000,000 matrix. The number of iterations of CG to converge will grow as the problem size grows. The necessary number of iterations should theoretically grow proportionally to k, the linear dimension of the mesh, which is the square root of n. Since the time for the matvec is $O(n)$, the overall work should grow as $O(n^{3/2})$.

In analyzing your program, you will want to accumulate the run time for the various subroutines separately. Theoretically, `MatVec` does more computation and more communication than either `DAXPY` or `DDOT`, but you may be surprised by what you find (and you may find that your code has unexpected performance bugs).

Grading on this problem will be one third on correctness, one third on performance, and one third on completeness of your experiments and report. (Don't panic! You can get a good grade on performance by being careful with data layout and communication.) For performance, we will measure your code running on the sample data in the test harness, on the largest possible array sizes. The measure of performance will be a combination of raw speed (on a fixed number of processors and a fixed array size) and speedup (ratio of time on one processor to time on p processors).

You can do this assignment on any machine you like, but you should make sure that it runs correctly on Triton at SDSC because that's where we'll run it to grade it.

The FAQ under Homework 3 on the course web page will have some details of how the harnesses work. We'll be adding to that file during the course of the assignment as questions come up.