CSE/CS 8803 PNA: Parallel numerical algorithms Spring 2008

Homework 1 — Due Feb. 21

Prof. Richard Vuduc *Assigned: Jan. 31*

1 Parallel conjugate gradients (CG) for linear systems

Your assignment is to implement a parallel version of the conjugate gradients (CG) algorithm for solving a linear system, $Ax = b$, where A is symmetric positive definite. In addition, you will be asked to evaluate the performance of your implementation as directed below. You may choose any parallel programming model for your implementation, including those covered in class (PThreads, OpenMP, MPI, Cilk, UPC, Co-Array Fortran), and do your experimental evaluation on any parallel platform of your choice.

For your convenience, I've provided a serial C implementation of the conjugate gradient algorithm shown in Listing 1. This implementation is available on the course website.

Working in teams: If you choose to work in teams (pairs or teams of 3), your submission should do a meaningful head-to-head comparison of performance and "productivity" of alternative programming models on the same platform, or a comparison of multiple platforms in the same model, or both.

What to submit: Submit a tarball of your actual implementation(s), and a short write-up (5-page max, plus any supplemental references or material) that answers the evaluation questions below. We'll *try* to use the T-Square homework submission system.

2 How to "evaluate" your implementation

Recall that CG consists essentially of a matrix-vector product and several dot-products and vector-scale operations, so that's essentially what you need to parallelize.

Data distribution. In your program, you *must* assume the input vector for the right-hand side, b, is initially distributed among all processors, though you may use any data distribution you like.

Test matrices. To evaluate your implementation, I'm providing two test matrices from structural automotive applications, and I'm asking you to "generate" a third stencil matrix; see Section 3.2 for details.

Measurement. You should measure the time of your implementation to solve the system, as the number of processors p increases. However, I am primarily interested in the performance of your solver, so you do not have to include the time of, *e.g.*, reading the test matrices from disk, or constructing the initial matrix.

Write-up. In your write-up, you should explain your approach and evaluate the performance of your implementation, specifically addressing the following:

- *Correctness.* How do you know your implementation is correct?
- *Per-processor performance.* Estimate the best possible per-processor performance, and explain your estimate. How close is your achieved per-processor performance to this peak? How does your per-processor performance compare to the serial implementation provided? You may use standard benchmarks, such as the STREAM benchmark [McC], to evaluate the memory bandwidth of your system.
- *Scaling on the 27-point stencil.* How does your implementation scale with n for the stencil matrix? How does this compare to our "Poisson" table from class?
- *Load balance.* Achieving good load-balance for the stencil matrix should have been easy; how did you (or could you have, if you didn't) achieve load balance for the other matrices?
- *Computation and communication.* What can you say about the computation vs. communication requirements of your implementation?
- *Modeling.* Try to come up with an analytical or empirical model of performance of your implementation. What can you say about the time, memory, speedup, and efficiency of your implementation?

3 Miscellaneous information

3.1 Accounts on WARP

You may use any parallel machine, including the Interactive High Performance Computing Lab (IHPCL) "Warp" cluster, which should be available to all registered students of this class. The log-in node for Warp is `ccil.cc.gatech.edu`; some minimal documentation on building an MPI program on Warp appear here:

`http://www-static.cc.gatech.edu/projects/ihpcl/mpi.html`

3.2 Test matrices

You will evaluate the performance of your implementation on three test problems, a "synthetic" matrix which I ask you to create, and two test matrices which I provide:

Listing 1: **A conjugate gradient algorithm**.

```
vector_t cg (matrix_t A, vector_t b, real_t ϵ)
{
  vector_t x_k; // Current guess at solution
  vector_t r_k; // Residual, Ax − b
  vector_t s_k; // Next "step", i.e., search direction
  vector_t z_k; // Temporary vector
  real_t α; // Temporary scalar
  int k = 0; // Current iteration

  x_0 ← 0;
  s_0 ← b;
  r_0 ← b;

  do {
    k ← k + 1;
    z_k ← A · s_{k−1};
    α ← ||r_{k−1}||_2^2 / (s_{k−1}^T z_k);
    x_k ← x_{k−1} + α · s_{k−1};
    r_k ← r_{k−1} − α · z_k;
    s_k ← r_k + (||r_k||_2^2 / ||r_{k−1}||_2^2) s_{k−1};
  } while (||r_k||_2 > ϵ||b||_2);

  return x_k;
}
```

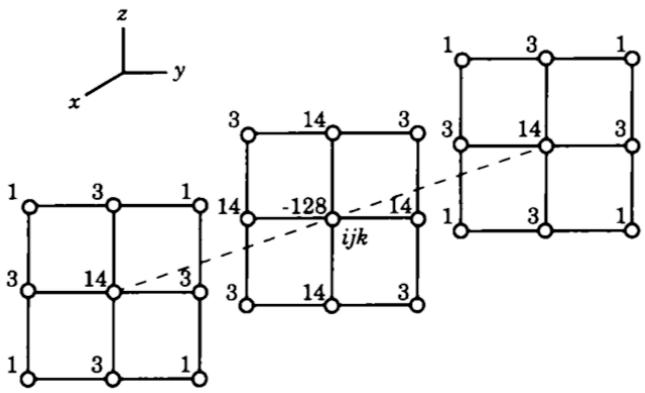

Figure 1: Coefficients for a 27-point 3-D stencil derived for Poisson's equation, taken from Spotz and Carey [SC96].

- *Synthetic matrix.* Your synthetic matrix should represent a 27-point stencil for a 3-D $n \times n \times n$ grid, using the coefficients indicated in Figure 1. You'll be asked to consider performance of your implementation as n increases.

  Since you can take the coefficients as fixed and you only need to provide CG with a kernel that effectively multiplies a given vector by the matrix, you don't necessarily need to explicitly construct a matrix; whether or not you do is up to you.

- *Test matrices.* I have also provided two test matrices in binary data files, along with some helper routines for reading in these matrices. These matrices come from automotive structural analysis problems (MSDOOR [uflb] and AUDI [ufla]), but have been "reconditioned" numerically so you will not have to consider implementing a preconditioner. You may access these matrices from my UNIX home directory (on, *e.g.*, Warp):

  `~richie/projects/bebop/matrices/*.flat`

3.3 Serial CG implementation

The serial CG implementation I've provided has some helper routines which you may find useful.

Timers: I've provided a "stop-watch" module, which is a wrapper around the portable cycle-counter reader, "cycle.h", which is included in FFTW [FJ]. This module is effectively a portable way to use the cycle counters available on most CPUs.

Caveat: If you are using a platform which dynamically scales the clock frequency (not the case on WARP), then this approach may be less useful.

Matrix I/O: The serial implementation has utility routines to read the two pre-generated test matrices, so you'll likely want to use those as well. One "feature" of these matrix I/O routines is that they define an interface for reading an arbitrary block of rows from the matrix, which will hopefully prove useful in the parallel case for distrbuting the matrix.

References

[FJ] Matteo Frigo and Steven G. Johnson. FFTW home page. http://fftw.org.

[McC] John McCalpin. STREAM benchmark. http://www.streambench.org/.

[SC96] W.F. Spotz and G.C. Carey. A high-order compact formulation for the 3d Poisson equation. *Numerical Methods for Partial Differential Equations*, 12:235–243, 1996.

[ufla] University of florida sparse matrix collection: Matrix 1252–audikw_1. http://www.cise.ufl.edu/research/sparse/matrices/GHS_psdef/audikw_1.html.

[uflb] University of florida sparse matrix collection: Matrix 1664–msdoor. http://www.cise.ufl.edu/research/sparse/matrices/INPRO/msdoor.html.