Parallel Conjugate Gradients Assignment

David Bindel

October 17, 2001

1 Introduction

In this assignment, we will tune a parallel implementation of the conjugate gradient algorithm (CG), a technique for solving linear systems. The CG algorithm, like a great many algorithms with similar structure, does a lot of difficult-to-optimize memory-intensive operations like dot products and sparse matrix-vector multiplications. This characteristic, along with the simplicity of the algorithm, makes makes it an attractive benchmark: the NAS parallel benchmarks include CG, as does SPEC FP. CG is also popular in real applications.

Unfortunately, the conjugate gradient algorithm appears incredibly mysterious at first (and even second and third!) sight. The focus of these notes (and of the assignment) is getting high performance out of CG; don't worry too much if you have difficulty following the mathematics.

There are a variety of good references on CG and related algorithms. The book *Templates for the Solution of Linear Systems* [**?**] is available online, and includes techniques for parallel implementation, information on sparse representations, tuning advice, pointers to current literature, etc. A survey paper "Parallel Numerical Linear Algebra" [**?**] by Demmel, Heath, and van der Vorst discusses the parallel implementation of a variety of techniques from numerical linear algebra, including some interesting variants of CG, and is a good complement to the *Templates* book. Shewchuk's article, "Conjugate Gradients without the Agonizing Pain" is a gentle introduction to the mathematics of the method, assuming a minimum of mathematical sophistication. More concise descriptions are given in books by Golub and Van Loan [**?**], Demmel [**?**], Trefethen and Bau [**?**], and in other standard texts on numerical linear algebra.

2 Conjugate Gradients: The Idea

A matrix A is symmetric and positive-definite (SPD) if $A_{ij} = A_{ji}$ and $y^T Ay > 0$ for any $y \neq 0$. Symmetric positive-definite matrices are common in problems from engineering and the physical sciences, where they correspond to a minimization principle (nature following the "path of least resistance").

```
1     Compute r^(0) = b − Ax^(0) for some initial x^(0)
2     p^(0) = 0, ρ_{−2} = 1
3     for i = 1, 2, ...
4           ρ_{i−1} = r^(i−1)^T r^(i−1)
5           β_{i−1} = ρ_{i−1}/ρ_{i−2}
6           p^(i) = r^(i−1) + β_{i−1} p^(i−1)
7           q^(i) = A p^(i)
8           α_i = ρ_{i−1}/p^(i)^T q^(i)
9           x^(i) = x^(i−1) + α_i p^(i)
10          r^(i) = r^(i−1) − α_i q^(i)
11          Check convergence; continue if necessary
12    end
```

Figure 1: The CG algorithm (adapted from *Templates* figure 2.5)

CG is a method for solving

$$Ax = b$$

where A is a symmetric positive-definite matrix. CG is an *iterative method*: given an initial guess $x^{(0)}$ at the solution, the method generates a sequence of increasingly improved approximate solutions $x^{(1)}, x^{(2)}, \ldots$. The iteration terminates when the error is deemed "small enough," or when the user decides that the iteration has run for too long. A common measure of error is the *residual*:

$$r^{(i)} = b - Ax^{(i)}$$

The conjugate gradient iteration keeps track (approximately) of the residual $r^{(i)}$ from step to step, so a convergence criterion based on the residual will be inexpensive.

For a symmetric positive-definite system, solving $Ax = b$ is equivalent to minimizing

$$\phi(x) = \frac{1}{2}x^T Ax - x^T b$$

The method of conjugate gradients works by moving "downhill" toward this minimum. To get the approximate solution $x^{(i)}$, the algorithm picks a search direction vector $p^{(i)}$ (also called the *gradient*) and then computes

$$x^{(i)} = x^{(i-1)} + \alpha_i p^{(i)}$$

where α_i is chosen to minimize $\phi(x^{(i)})$. The gradients $p^{(i)}$ are chosen to be *A-conjugate* or *A-orthogonal*; that is,

$$p^{(i)^T} Ap^{(j)} = 0 \text{ for } i \neq j$$

These are the conjugate gradients that give the method its name.

3 Conjugate Gradients: The Code

The code for the conjugate gradient method is shown in Figure 1. Each step involves

- Two inner products (lines 4 and 8),
- A sparse matrix-vector multiplication (line 7),
- Three vector updates (lines 6, 9, and 10),

Notice that A is *only* used in the matrix-vector product in line 7. The matrix entries are never used explicitly. This feature of the conjugate gradient algorithm is a major benefit for problems where it is expensive to get the coefficient matrix A, or where multiplication by A can be done much more quickly with some sort of fast transform than by the usual algorithm. Also, note that the method has modest memory requirements: besides the overhead to store the matrix and a few scalars, we only need four storage vectors.

4 Performance of CG

4.1 Decomposing the problem

There are many possible ways to order the variables and equations, and to assign the variables to processors. Such orderings are equivalent from the point of view of the exact solution, but not from the perspective of communication required for matrix-vector multiply. When the matrix corresponds to a discretization of a partial differential equation, there may be some physical intuition about how to partition the problem. When A has no physical meaning (that you know about), problem partitioning is tricker.

You are allowed, but not required, to use a package like METIS or ParMETIS to partition your problem in a way that minimizes the communication between processors needed.

4.2 Sparse matrix-vector multiply

Getting even good serial efficiency from conjugate gradients can be a challenge. Inner products and sparse matrix-vector multiplies both tend to use too much data per floating point operation to get very good cache re-use. It is possible to optimize the sparse matrix-vector multiply some, though. The "Sparsity" package written by Eun-Jin Im (a recent graduate advised by Kathy Yelick) addresses some of the challenges of sparse matrix-vector multiplication. You are welcome to use code from Sparsity, as long as you acknowledge its source.

In the parallel case, you additionally have to worry about the communication overhead of making sure each processor has the parts of the vector it needs from other processors. A good partition of the problem reduces this communication overhead. If some form of split-phase or non-blocking communication

is available, and each processor has sufficient work computing the part of the matrix-vector product only involving the variables it owns locally, it may be possible to hide the communication cost of retrieving variables from other processors.

4.3 Inner products and synchronization

As written in figure 1, the two inner products are synchronization points. There is no way to proceed until the inner product is computed, and there is no way to compute the inner product without global communication. There is a way to rewrite the iteration, described in figure 4.2 of the templates book, which would allow you to overlap the communication needed for the inner products with useful computation.

There are other ways to decrease the number of synchronization points, but they involve fundamentally recasting the algorithm. If you are interested in them, read the *Templates* book or the survey article by Demmel, Heath, and van der Vorst (or come ask during office hours).

5 Preconditioners: The Idea

The convergence of the conjugate gradient algorithm depends on the *spectrum* (the set of eigenvalues) of A. A crude but useful way to characterize the spectrum of A is with the *spectral condition number*

$$\kappa_2(A) = \lambda_{max}/\lambda_{min}$$

We can (but won't in these notes) prove that the conjugate gradient algorithm in exact arithmetic will reduce some measure of the error by at least a factor like $\alpha = (\kappa(A)-1)/(\kappa(A)+1)$ at each step. The actual convergence behavior is somewhat more complicated, but in general the method of conjugate gradients is at its worst when the eigenvalues of A are spread out. For more details, see one of the listed references, or come ask in office hours.

Preconditioning is a technique for accelerating convergence by solving an equivalent problem

$$M^{-1}Ax = M^{-1}b$$

where the *preconditioning matrix* M is chosen so that

- M is symmetric and positive-definite,
- The eigenvalues of $M^{-1}A$ are more tightly clustered than those of A,
- And solving $M^{-1}r$ is easy.

CG is almost always used with a preconditioner. The choice of a good preconditioner is problem-specific, and much research is devoted to developing preconditioners. Popular preconditioners often correspond to similar but easier-to-solve versions of the original problem (e.g. discretizations on coarser or more regular

1 Compute $r^{(0)} = b - Ax^{(0)}$ for some initial $x^{(0)}$
2 $p^{(0)} = 0$
3 **for** $i = 1, 2, \ldots$
4 Solve $Mz^{(i-1)} = r^{(i-1)}$
5 $\rho_{i-1} = r^{(i-1)^T} z^{(i-1)}$
6 $\beta_{i-1} = \rho_{i-1}/\rho_{i-2}$
7 $p^{(i)} = z^{(i-1)} + \beta_{i-1} p^{(i-1)}$
8 $q^{(i)} = Ap^{(i)}$
9 $\alpha_i = \rho_{i-1}/p^{(i)^T} q^{(i)}$
10 $x^{(i)} = x^{(i-1)} + \alpha_i p^{(i)}$
11 $r^{(i)} = r^{(i-1)} - \alpha_i q^{(i)}$
12 Check convergence; continue if necessary
13 **end**

Figure 2: Preconditioned CG (adapted from *Templates* figure 2.5)

meshes, similar problems with different boundary conditions, or versions of the problems with some dependencies between variables artificially removed).

6 The Preconditioned CG Algorithm

The code for the preconditioned conjugate gradient method, as presented in the *Templates* book, is shown in Figure 1. The structure is very similar to unpreconditioned CG; the only addition is the preconditioner solve in line 4.

There is a lot of information on preconditioners in the *Templates* book: one of the five chapters of the book is devoted entirely to preconditioning. The ordering of variables makes a difference in the communication required for some preconditioners. There is a trade-off between the effectiveness of a preconditioner and how easy it is to parallelize. The choices that minimize total wall-clock time will vary from problem to problem and from machine to machine. The code we provide includes two preconditioners: a *block Jacobi* preconditioner (which is trivially parallelizable), and a *symmetric successive over-relaxation (SSOR)* preconditioner (which is more difficult to parallelize).

7 Base Implementations

We will provide three base implementations:

- A parallel unpreconditioned CG code written in UPC
- A serial preconditioned CG code written in C
- A serial (?) preconditioned CG code written in Titanium

You might also find it interesting to look at the code associated with the *Templates* book, which is available at netlib. The code is written in Fortran using a programming paradigm called "reverse communication."

Besides Sparsity and METIS, another library you might be interested in checking out is PETSc (Portable Extensible Toolkit for Scientific Computing). PETSc includes CG, along with many other iterative linear solvers, and also a lot of preconditioners. PETSc is almost certainly installed on the Cray. I'll get a pointer to the exact location.

The implementations use an internal storage format for sparse matrices called "Compressed Sparse Row" (or CSR). The CSR data structure consists of

- An array of elements of the matrix, omitting zeros
- A parallel array of column indices
- An array of indices telling the location of the start of each row

For instance, the algorithm to print out the explicitly represented matrix entries is

```
for (i = 0; i < m; ++i)
  for (j = row_start[i]; j < row_start[i+1]; ++j)
    printf("A(%d, %d) = %g", i, col_index[j], values[j]);
```

A good source of test problems is the *Matrix Market*. The web page will have a pointer to several Matrix Market matrices, and at least one of the provided codes contains routines to read files in Matrix Market format and convert them to an internal CSR data structure.

8 Tasks

The basic task is to

- Run and time both implementations. Do some performance experiments.
- Pick an implementation and work on optimizing it. Base your work on experimental results and on an understanding of the code bottlenecks.
- Pick some extension and try it out. Again, try to relate your observations to the performance.

Possible optimizations include

- Optimizing the serial performance of the sparse matrix-vector multiply kernel.
- Optimizing the partitioning of work between processors.
- Optimizing the communication patterns.

- Improving the preconditioner performance.
- Fiddling with the block sizes for the block Jacobi preconditioner.

Possible extensions include

- Picking a specific class of matrices and studying optimizations that take advantage of any special structure that the matrix might have.
- Implementing the re-arranged algorithm with a split preconditioner.
- Choosing a good preconditioner for some specific problems. If you have a large SPD problem that you really care about, feel free to focus on that problem.
- Implementing a blocked CG algorithm. The simplest form of blocking would be to basically run the ordinary algorithm for several right hand sides in parallel. More sophisticated block CG algorithms generate a descent *space* at each step consisting of multiple search directions, and then minimize in that space.