Additional Class Projects

1. Choose a number of the problems listed below and implement them, instrument them with PAPI, and illustrate their behavior and performance. Then implement them in parallel and describe the performance.

   Problems
   1. Matrix-matrix multiplication
   2. Matrix transposition
   3. Solution of triangular linear systems
   4. Solution of tridiagonal linear systems by odd-even reduction
   5. Solution of block tridiagonal linear systems
   6. Cholesky factorization
   7. LU factorization by Gaussian elimination with partial pivoting
   8. LDL' factorization of symmetric indefinite matrix by Bunch-Kaufman or Aasen algorithm
   9. Conjugate gradient method for linear systems
   10. Asynchronous iterative methods for linear systems
   11. Matrix inversion by Gauss-Jordan elimination
   12. Matrix inversion by Newton's iteration
   13. QR factorization by Householder method
   14. QR factorization by Givens method
   15. QR factorization by Gram-Schmidt method
   16. Bisection and inverse iteration for symmetric tridiagonal eigenvalue problems
   17. Lanczos method for symmetric eigenvalue problems
   18. Reduction to tridiagonal form by Householder method
   19. Reduction to Hessenberg form by Householder method
   20. Bidiagonalization of rectangular matrix by Householder method
   21. Newton's method for systems of nonlinear equations
   22. Direct search methods for multidimensional optimization
   23. One-dimensional adaptive quadrature
   24. Multiple integrals by Monte Carlo method
   25. Multigrid method for elliptic PDEs
   26. Domain decomposition method for elliptic PDEs
   27. Fast Fourier transforms
   28. N-body problem

   Issues to address
   1. Performance comparison of different algorithms for same problem
   2. Performance comparison of different machines for same problem
   3. Performance comparison of different programming models for same problem
   4. Performance comparison of 1-D and 2-D partitioning for same problem

5. Performance comparison of row-oriented and column-oriented partitioning for same problem
6. Performance comparison of different implementations of communication patterns (e.g., broadcast, reduction) for same problem
7. Modeling and empirical testing of performance and scalability
8. Dynamic load balancing, where appropriate

2. Parallel Implementation of Your Own Application

   Some students take this class because they plan to parallelize an algorithm or application from their own research. This can be an acceptable project, but the approach and objectives must be clearly stated from the outset. Your project proposal should specify the problem your code will address, what parts have already been parallelized, if any, and what further progress you expect to make as part of your class project.

3. Implement a parallel sparse incomplete Cholesky preconditioner

4. Implement a 2D fast multipole solver for N-body problems.

5. Conjugate Gradient Method

   There are many techniques for solving systems of linear equations, i.e., solving for x in A*x = b. When the system is dense, the routines in the LAPACK or ScaLAPACK libraries are appropriate. It is often the case that the system is sparse, because every variable in system depends upon only a small number of other variables. We will be covering various techniques for solving sparse systems in lecture. The Conjugate Gradient method, called CG for short, is suitable for solving any linear system where the coefficient matrix A is both symmetric, i.e. A = A', and positive definite. Three equivalent definitions of positive definiteness are:

   all of A's eigenvalues are positive,
   x'*A*x > 0 for all nonzero vectors x, and
   an LU factorization of A of the form A = L*L' exists (the so-called Cholesky factorization).

   CG is often used for sparse systems, because it only requires a single operation, matrix-vector multiplication; it does not even require that you actually have A, but only that you have a function for multiplying by it.

   Here is the CG algorithm:

   x = initial guess for inv(A)*b
   r = b - A*x ... residual, to be made small

```
p = r          ... initial "search direction"
repeat
  v = A*p
     ... matrix-vector multiply
  a = ( r'*r ) / ( p'*v )
     ... dot product (BLAS1)
  x = x + a*p
     ... compute updated solution
     ... using saxpy (BLAS1)
  new_r = r - a*v
     ... compute updated residual
     ... using saxpy (BLAS1)
  g = ( new_r'*new_r ) / ( r'*r )
     ... dot product (BLAS1)
  p = new_r + g*p
     ... compute updated search direction
     ... using saxpy (BLAS1)
  r = new_r
until ( new_r'*new_r small enough )
```

Here is a rough motivation for this algorithm. CG maintains 3 vectors at each step, the approximate solution x, its residual r=A*x-b, and a search direction p, which is also called a conjugate gradient. At each step x is improved by searching for a better solution in the direction p, yielding an improved solution x+a*p. This direction p is called a gradient because we are in fact doing gradient descent on a certain measure of the error (namely sqrt(r'*inv(A)*r)). The directions pi and pj from steps i and j of the algorithm are called conjugate, or more precisely A-conjugate, because they satisfy pi'*A*pj = 0 if i.ne.j. One can also show that after i iterations xi is the "optimal" solution among all possible linear combinations of the form:

a0*x + a1*(A*x) + a2*(A^2*x) + a3*(A^3*x) + ... + ai*(A^i*x)

For most matrices, the majority of work is in the sparse matrix-vector multiplication v=A*p in the first step. The other operations in CG are easy to parallelize. The saxpy operations are embarrassingly parallel, and require no communication. The dot-products require local dot-products and then a global add using, say, a tree to form the sum in log(p) steps.

The rate of convergence of CG depends on a number kappa called the condition number of A. It is defined at the ratio of the largest to the smallest eigenvalue of A (and so is always at least 1). A roughly equivalent quantity is norm(inv(A))*norm(A), where the norm of a matrix is the magnitude of the largest entry. The larger the condition number, the slower the convergence. One can show that the number of CG iterations required to reduce the error by a constant g<1 is proportional to the square root of kappa. There are methods

similar to CG that are suitable for matrices A that are not symmetric or positive definite. Like CG, most of their work involves computing the matrix vector product A*p (or possibly A'*p), as well as dot-products and saxpy's. For on-line documentation and software, including advice on choosing a method, Templates for the Solution of Sparse Linear Systems.

The Assignment

Parallize conjugate gradient on a cluster. You are given with a serial version of conjugate gradient code using register and cache blocking. Both register and cache blocking code are used to optimize the sparse matrix-vector multiply operation in CG.