A Parallel Implementation Of The Conjugate Gradient Method

- project for CS 554 -

Elena Caraba

May 4, 2010

Abstract

The conjugate gradient method and the methods deriving from it are some of the most effective tools for solving large sparse symmetric positive-definite systems. In this project we implement the conjugate gradient algorithm in parallel using MPI and do a performance analysis while attempting to solve linear systems of different sizes on an increasing number of processors.

Algorithm Implementation

The serial conjugate gradient algorithm is briefly illustrated by the following algorithm:

```
$x_0$ = initial guess
$r_0 = b - Ax_0$
$s_0 = r_0$
for k=0,1,2,...
    $\alpha_k = r_k^T r_k / s_k^T A s_k$
    $x_{k+1} = x_k + \alpha_k s_k$
    $r_{k+1} = r_k - \alpha_k A s_k$
    $\beta_{k+1} = r_{k+1}^T r_{k+1} / r_k^T r_k$
    $s_{k+1} = r_{k+1} + \beta_{k+1} s_k$
end
```

Since CG is an iterative method and the result at the next step depends on the result at the current step, it is hard to simultaneously compute the intermediate steps (such as step size, search direction, etc). A more promising way to parallelize CG is by exploiting geometric parallelization: distribute the data across processors and make each processor responsible for computations on its local data.

As one can see, the CG algorithm requires one matvec (matrix-vector multiply), and several dot products (or inner products). These operations can be easily parallelized.

Thus, implementing the algorithm in parallel, requires the following:

- Step 1. Read the data from the input file and divide it across processors, using MPI_Bcast and MPI_Scatter(v).

- Step 2. Compute the inner product locally, and do a sum reduction (MPI_Allreduce) across all processes.
- Step 3. Do the matrix-vector product in parallel using MPI_Allgatherv, to gather all the local parts of the vector into a single vector, and then to do the multiplication.

Although one can choose MPI_Send and MPI_Recv, we have seen from *mp1* that there are several problems one can encounter when using them. Instead, we choose MPI_Allgatherv, which is known to be an efficient way to communicate data between processes. [1]

For the implementation of the algorithm, for which we chose C++ and MPI, we used the starting code tarballs provided to students enrolled in CSE 6230 (Fall 2009)[2] and CSE8803pna (Spring 2008)[3], both courses at Georgia Institute of Technology. The starter codes provide a serial implementation of the CG algorithm, as well as auxiliary functions to read a matrix and to time the main computations. The geometric parallelization idea comes from the UC-Berkeley course CS 267 Fall2002, Assignment 3.[4]

A pseudocode of the Step 1 would be the following:

```
 if procRank == 0
use mmio.h (provided by the Matrix Market Factory) to read
the matrix A stored in .mtx files
get matrix dimension n, number of nonzeros, the coordinates of
the nonzero elements, as well as their values
 end
 MPI_Bcast((void *)&n,1,MPI_INT,0,MPI_COMM_WORLD);
 divide data across processors:
     send blocks of rows to each processor;
     and divide the rows into blocks (column-wise):
     MPI_Scatterv(sendRowPointer,sendRowCounts,rowDispl,MPI_INT,recvRowPointer,
     localCount,MPI_INT,0,MPI_COMM_WORLD);
     MPI_Scatter(sendColCounts,1,MPI_INT,&recvLocal,1,MPI_INT,0,MPI_COMM_WORLD);
     and
     MPI_Scatterv(sendColIndex,sendColCounts,colDispl,MPI_INT,recvColIndex,
     localCount,MPI_INT,0,MPI_COMM_WORLD);
 MPI_Scatterv(sendVal,sendColCounts,colDispl,MPI_DOUBLE,recvVal,recvLocal,
 MPI_DOUBLE,0,MPI_COMM_WORLD);
```

The C++ implementation just of the cg routine is illustrated in Appendix A.

As test matrices we used sparse and symmetric positive definite matrices provided by the Matrix Market Repository.[5] We chose the BCS Structural Engineering matrices (bcsstk), for no specific reason other than these matrices have the required properties for a conjugate gradient algorithm. They are shown in the following table:

Table 1

matrix	size	NNZ	CNR
bcsstk14	1806	32630	1.3E+10
bcsstk16	4884	147631	6.5E+01
bcsstk17	10974	219812	6.5E+01

Performance Analysis

For the performance analysis we will use the bcsstk17[6] matrix. In the implementation of the algorithm, we have considered the matvec to be done with 2-D agglomeration on a 2-D mesh, and the inner product as a coarse-grain parallel algorithm on a 2-D mesh. Using the formulas derived in class, we can create a performance model.

$$T_{p_{dot}} \approx t_{c_{dot}} \frac{n}{p} + (t_s + t_w)2\sqrt{p}$$

$$T_{p_{matvec}} \approx t_{c_{matvec}} \frac{n^2}{p} + 2(t_s\sqrt{p} + t_w n)$$

$$T_{axpy} = t_{\alpha x+y} n$$

where $t_{\alpha x+y}$ is the time computing $\alpha \times x + y$, or one addition and one multiplication.

$t_{c_{dot}}$, $t_{c_{matvec}}$, $t_{\alpha x+y}$ can be computed by analyzing how many floating point operations each take, and multiply the total number by 1/(x MFlops), where x MFlops is the number of floating point operations in one second.

The alternative, would be to calculate the t_c's by running the serial version of the algorithm and timing each of the main operations: dot, axpy and matvec.
$t_{c_{dot}} = 0.000333071$
$t_{c_{matvec}} = 0.0215402$
$t_{c_{axpy}} = 0.000209093$

t_s and t_w were computed by writing a small code that first sends a double, and then an integer, and measure the total time $t_s + t_w * nbytes$. This results in a linear system with two equations that we solve, and for which we obtain:
$t_s = 0.0399060$ seconds
$t_w = 0.0099843$ seconds.

If the method converges in k steps, the total time spent by the parallel code can be approximated by the total time required to do 2 inner products, 1 matrix-vector multiply and 3 scalar-vector-multiply-vector-addition, or axpy.

$$T_p = T_{p_{dot}} \times 2k + T_{p_{matvec}} \times k + T_{axpy} \times 3k$$

The serial time for this algorithm can be analytically computed with the following formula:

$$T_s = T_{s_{dot}} \times 2k + T_{s_{matvec}} \times k + T_{s_{axpy}} \times 3k$$

where:

$$T_{s_{dot}} = nt_{c_{dot}}$$

$$T_{s_{matvec}} = n^2 t_{c_{matvec}}$$

$$T_{s_{axpy}} = nt_{c_{axpy}}$$

To keep constant efficiency we must have:

$$T_s = E(pT_p)$$

$$2nt_{c_{dot}} + n^2 t_{c_{matvec}} + 3nt_{c_{axpy}} = E(2nt_{c_{dot}} + 4p\sqrt{p}(t_s + t_w) + n^2 t_{c_{matvec}} + 2p(t_s\sqrt{p} + t_w n) + 3pnt_{c_{axpy}})$$

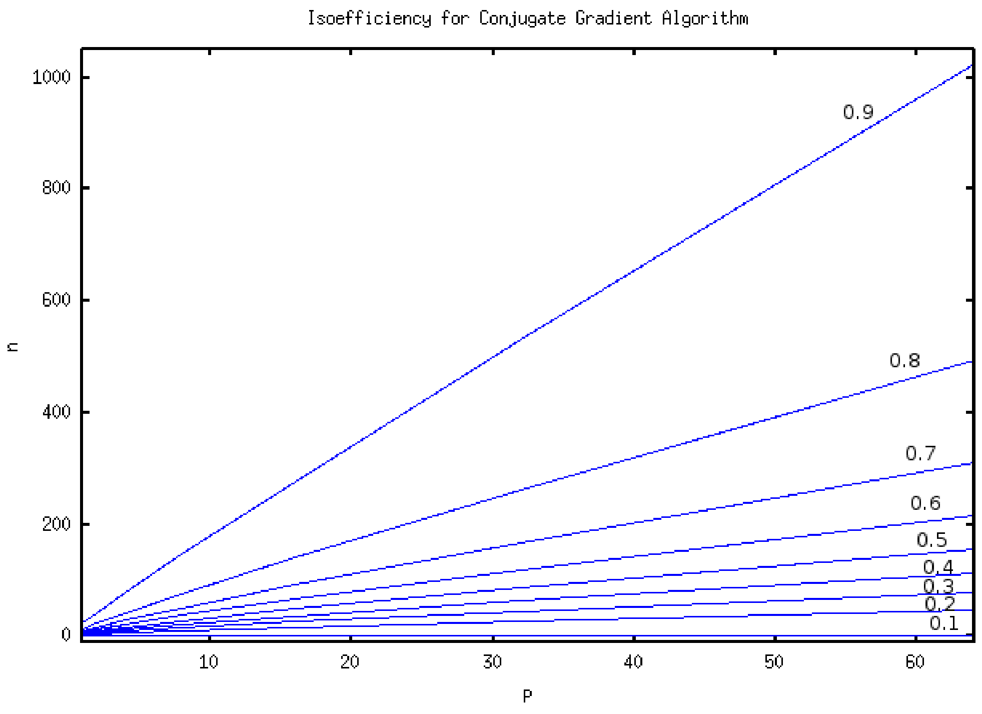


Figure 1: Isoefficiency of the conjugate gradient method using the bcsstk17 matrix.

which holds for sufficiently large p if $n = \Theta(p)$. Since serial work W_1 is of order $\Theta(n^2)$, isoefficiency function is $\Theta(p^2)$.

The isoefficiency equation reduces to solving the following quadratic:

$$n^2(1-E)t_{c_{matvec}} + n[(1-E)(3t_{c_{axpy}} + 2t_{c_{dot}}) - 2pEt_w] - Ep\sqrt{p}(6t_s + 4t_w) = 0$$

Figure 1 shows its graph.

Observed Efficiency and Speedup

In Figure 2 we plot the efficiency of the algorithm for the three sparse and symmetric positive definite (SPD) matrices listed in Table 1. We observe that the efficiency for all three matrices is about the same. And it decreases fast as we use a higher number of processors.

Looking at the speedup plot, Figure 3, we see that for a small number of processors the speedup increases somewhat fast, but it drops for small matrices as we increase the number of processors, and remains about constant for large matrices. One explanation could be that the overhead from communication and latency is affecting more the overall time as we add in more processors.

Visualizing the Implementation

Using Jumpshot and compiling the codes with `-mpilog`, we can get a better understanding of how the parallel conjugate gradient is implemented, and whether

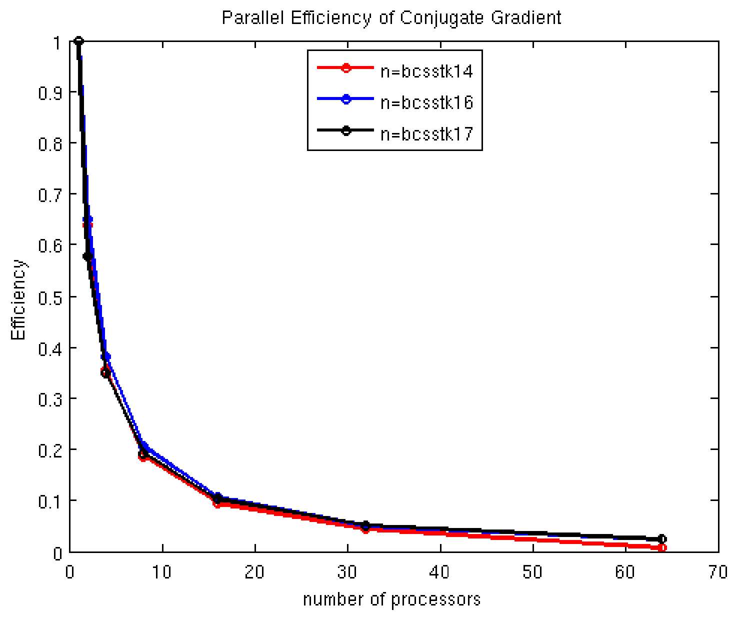


Figure 2: Efficiency of the conjugate gradient method for different sparse SPD matrices

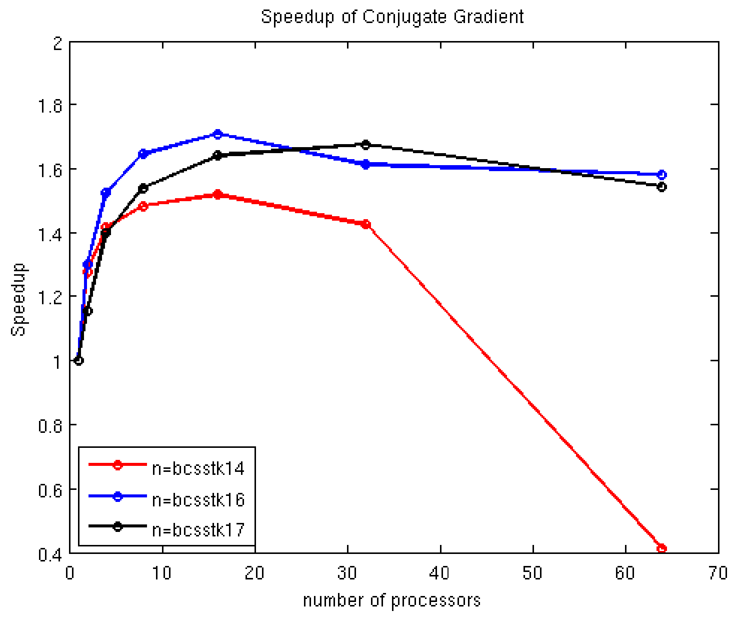


Figure 3: Speedup of the conjugate gradient method for different sparse SPD matrices

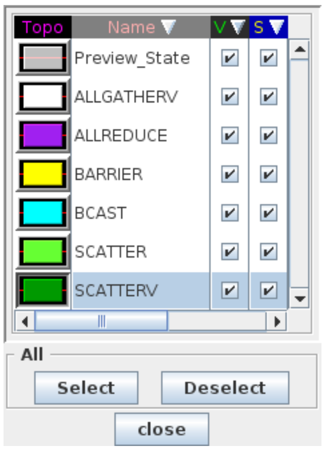


Figure 4: Legend for Jumpshot Screenshots

or not it is efficient.

We have previously stated the algorithm first loads the data in one of the processors, uses MPI_Bcast to send the dimension of the matrix to the other processors, and then "divides" the data into subsets that are sent to the other processors via MPI_Scatterv and MPI_Scatter.

This is illustrated by Jumpshot in Figure 5. Processor zero is assigned initially the data, and then it starts broadcasting subsets of it to Processor 1, 2 and 3.

Figure 4 shows the legend of the screenshots.

The inner product is calculated on parallel by computing a local sum on each processor and then doing a sum reduction, using MPI_Allreduce. We illustrate the algorithm below.

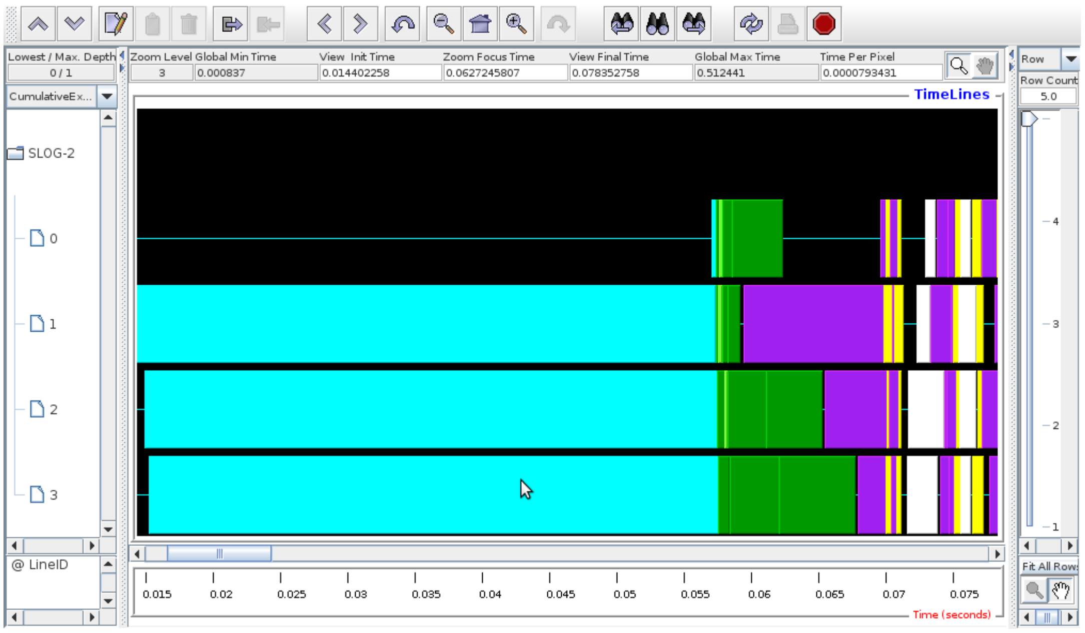
Routine calculating the inner product:

```
double dot (const double *x, const double *y, int n)
{
   int i;
  double sum= 0.0;
  double finalSum = 0.0;

  for (i = 0; i < n; ++i)
    sum += x[i] * y[i];
  MPI_Allreduce( &sum,&finalSum,1,MPI_DOUBLE,MPI_SUM,MPI_COMM_WORLD);
  return finalSum;
}
```

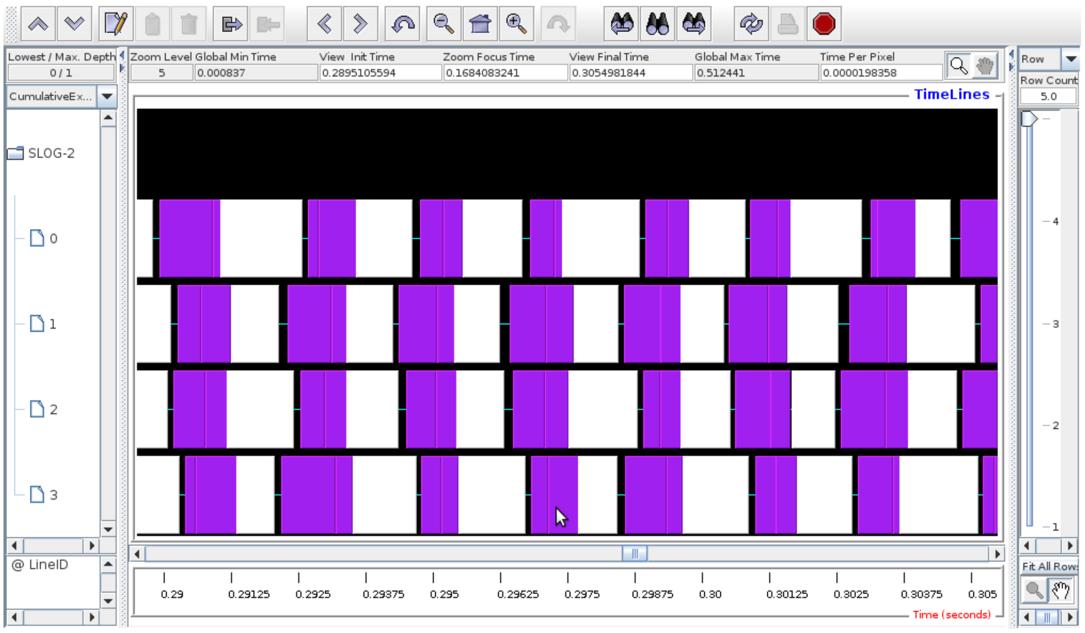
Figure 6 illustrates the parallelism of the inner product implementation: the purple color corresponds to MPI_Allreduce and we see that all are happening simultaneously (more or less), and each are followed by a strip of white color, which corresponds to MPI_Allgather from the matvec. These are also executed in simultaneously across all processors.

The code snippet computing the matvec in parallel follows.



Lowest / Max. Depth 0 / 1
Zoom Level 3
Global Min Time 0.000837
View Init Time 0.014402258
Zoom Focus Time 0.0627245807
View Final Time 0.078352758
Global Max Time 0.512441
Time Per Pixel 0.0000793431
Row Count 5.0

Figure 5: "Data Broadcasting" via MPI_Bcast



Lowest / Max. Depth 0 / 1
Zoom Level 5
Global Min Time 0.000837
View Init Time 0.2895105594
Zoom Focus Time 0.1684083241
View Final Time 0.3054981844
Global Max Time 0.512441
Time Per Pixel 0.0000198358
Row Count 5.0

Figure 6: Gathering and Reducing via MPI_Allgather and MPI_Allreduce

Routine calculating a matrix-vector product:

```
void mymatvec(double *y, void *Adata, double *xsend, int m)
{
    csr_matrix_t *A = (csr_matrix_t *) Adata;
    int *rowInd = A->ptr;
    int *colInd = A->ind;
    double *localval = A->val;
    double xrecv[DIM],elemA;
    int col;

    MPI_Allgatherv(xsend,m,MPI_DOUBLE,xrecv,rowCounts,
rowDispl,MPI_DOUBLE,MPI_COMM_WORLD);

    for (int i = 0; i < m; ++i) {
        y[i] = 0;
        for (int j = rowInd[i]; j < rowInd[i + 1]; ++j) {
            elemA = *localVal++;
            col = *colInd++;
            y[i] = y[i] elemA * xrecv[col];
        }
    }
}
```

MPI_Allgatherv and MPI_Allreduce are used only once, in the matvec and the dot functions, respectively.

Conclusions or What I have learned

In this project, we have implemented a parallel version of the conjugate gradient method and we have tested it for three different size matrices. The implementation is simplistic in high-level language, such as Matlab, but it is slightly more difficult to implement the method in C++. And although we could have used PETSC or other libraries that have the conjugate gradient method, we chose to implement it ourselves because we wanted to gain experience parallelizing a serial code and learn about the difficulties one encounters when doing so. We learned that a good testing framework is very important, and without one in place from the beginning, several errors have to be corrected. Doing the geometric parallelization has proved to be the most challenging part of the project, together with the isoefficiency analysis. We also learned that using MPI routines such as MPI_Allgather, MPI_Sendrecv or MPI_Bcast one can make use of efficient communication between processors, and avoid risking deadlocking or poor communication latency.

Having access to a visualization software such as Jumpshot that shows you what happens in your code, how and when the function calls occur, has proved useful in the later stages of the project when we wanted to improve efficiency.

References

[1] *Using MPI: Portable Parallel Programming with the Message-Passing Interface - second edition*, W. Gropp, E. Lusk, A. Skjellum, The MIT Press, 1999

[2] http://insomnia.cc.gt.atl.ga.us:8080/cse6230-hpcta-fa09/assignments

[3] http://vuduc.org/teaching/cse8803-pna-sp08/hw1/

[4] http://www.eecs.berkeley.edu/ binetude/course/cs267/cs267.htm

[5] http://math.nist.gov/MatrixMarket/

[6] http://math.nist.gov/MatrixMarket/data/Harwell-Boeing/bcsstruc2/bcsstk17.html

1 Appendix A

The serial CG routine provided by the GeorgiaTech homework code tarball doesn't have to change too much for the parallel version; only the matvec and dot routines have to change as shown earlier in this report.

```
int
cg (matvec_t mymatvec, void* Adata, const double* b,
    double *x, double rtol, int n, double* rhist, int maxiter)
{
  const int nbytes = n * sizeof(double);


  double bnorm2;              /* ||b||^2 */
  double rnorm2, rnorm2_old;  /* Residual norm squared */
  double rho;
  double alpha;

  double *s;                  /* Search direction */
  double *r;                  /* Residual         */
  double *z;                  /* Temporary vector */

  int i;                      /* Current iteration */

  s = (double *)malloc (nbytes);
  assert (s);
  r = (double *)malloc (nbytes);
  assert (r);
  z = (double *)malloc (nbytes);
  assert (z);


  bnorm2 = dot (b, b, n);
  memset (x, 0, nbytes);
  memcpy (r, b, nbytes);
  memcpy (s, b, nbytes);

  rnorm2 = dot (r, r, n);
  for (i = 0; i < maxiter ; ++i) {
    #if defined (DEBUG_ME)
      fprintf (stderr, "Iteration %i: ||r||_2 = %g\n", i, sqrt (rnorm2));
    #endif

    rnorm2_old = rnorm2;
    mymatvec (z, Adata, s, n);
    alpha = rnorm2_old / dot (s, z, n);
    axpy (x, alpha, s, x, n);

    axpy (r, -alpha, z, r, n);
    rnorm2 = dot (r, r, n);
    axpy (s, rnorm2 / rnorm2_old, s, r, n);

    if (rhist != NULL)
      rhist[i] = sqrt(rnorm2 / bnorm2);
    if(rnorm2 <= bnorm2 * rtol * rtol)
                break;

  }

  free (z);
  free (r);
  free (s);

  if (i >= maxiter)
    return -1;

  return i;
}
```