Parallel GMRES Algorithm for Solving Giant Sparse System of Linear Equations Validated for 1, 2, 3, 6, 16 and 42 Processes

Arash Ghasemi

Project 3

Parallel Scientific Computing

Prof. D. Hyams

April 23, 2012

0.1 Algorithm and Implementation

The GMRES (Generalized Minimal RESidual) algorithm is completely presented and discussed in [2, 3]. Since the implementation of the algorithm is done in C-programming language in a modular and function oriented manner, the main-file is very short and looks like a pseudo-code itself! So we bring the main file here with some small details (like local variables, allocations, comments etc ..) omitted for brevity. In the next pages we describe the algorithm based on the implementation.

Listing 1: The main function of the parallel code implements the GMRES algorithm

```
1       // Set up MPI session
        ..
3       ..
        //read the sparse matrix from the file
5       matrix_reader(argv[1], p, my_rank, &nnz, &my_nnz, &nrows,
                      &ia, &ja, &A);
7       // reading the rhs vector from the input file
        read_rhs(argv[2], p, my_rank , nrows, nrowsI , b);
9       //localize
        localize(my_rank, p, nrows, ia, ja, commun, &receive, &cntowner,
11              &to_be_sent, &cntsent);

13      int precond_flag = atoi(argv[4]);
        double gmres_res = 1.e-10; //gmres residual
15      //ntotal is equal to nlocal+n phantom points
        int n_total = max_int_array(ja, my_nnz)+1;
17      double norm_r0 = 0.;

19      //pick the diagonal entries for preconditioning (if it is preconditioned)
        double *diag = (double *) calloc( nrowsI , sizeof(double) );
21      double *yk = (double *) calloc( n_total , sizeof(double) );
        if( precond_flag ) //should it be preconditioned?
23            for ( i = 0 ; i < nrowsI; i++ )
                    for ( j = ia[i] ; j < ia[i+1]; j++ )
25                        if( ja[j] == i ) //then it's on the main diagonal
                                diag[i] = A[j];
27
        //init x0
29      for( i = 0; i < nrowsI; i++)
              x0[i] = 1.0;
31      //gather/sync
        gather_sync(my_rank, p, nrows, ia, ja, receive, cntowner, to_be_sent,
33                  cntsent, x0, commun);
        /* //matrix vector product */
35      sparse_matmult(A, x0, r0, ia, ja, nrowsI, commun);
        //updating the final r0
37      for( i = 0; i < nrowsI; i++)
              r0[i] = b[i] - r0[i];
39
        // normalization ...
41      dotproduct(r0, r0, &norm_r0, nrowsI, commun); //dot product
        norm_r0 = sqrt(norm_r0); //root
43      for( i = 0; i < nrowsI; i++)
              r0[i] /= norm_r0; //normalizing
45
        // initial allocation of v
47      v[0] = (double *)calloc( n_total , sizeof(double) );
        for( i = 0; i < nrowsI; i++)
49            v[0][i] = r0[i]; //initializing ...
        g[0] = norm_r0; //initial g
51
        //main gmres loop
53      for( k = 0; k < k_end; k++) //will be fancier soon!
        {
55            //reallocating vars to have enough space for the next items
              ..
57            ..
              ..
59
              for ( j = 0 ; j <= (k+1); j++)
61                  h[j] = (double *)realloc( h[j], (k+1) *sizeof(double) );

63            //NOTE : THE FOLLOWING IF STATEMENT INCREASES PERFORMANCE WHEN IT IS
              //NOT SUPPOSED TO BE PRECONDITIONED BECAUSE IT JUST SKIPS THE
65            // INITIALIZATION OF VECTOR [yk].
              if( precond_flag ) //should it be preconditioned?
67            {
                    for ( i = 0 ; i < nrowsI; i++ )
```

```
69                      if( diag[i] )
                            yk[i] = v[k][i] / diag[i];
71                      else
                            yk[i] = v[k][i];
73              //gather/sync
                gather_sync(my_rank, p, nrows, ia, ja, receive, cntowner, to_be_sent,
75                          cntsent, yk, commun);
                /* //matrix vector product */
77              sparse_matmult(A, yk, u, ia, ja, nrowsI, commun);
            }
79          else
            {
81              //gather/sync
                gather_sync(my_rank, p, nrows, ia, ja, receive, cntowner, to_be_sent,
83                          cntsent, v[k], commun);
                /* //matrix vector product */
85              sparse_matmult(A, v[k], u, ia, ja, nrowsI, commun);
            }
87
            for ( j = 0 ; j <= k; j++)
89          {
                dotproduct(v[j], u, &h[j][k], nrowsI, commun); //dot product
91              for( ss = 0; ss < nrowsI; ss++)
                    u[ss] -= h[j][k] * v[j][ss];
93          }
            dotproduct(u, u, &h[k+1][k], nrowsI, commun); //dot product
95          h[k+1][k] = sqrt(h[k+1][k]); //norm
            //updating v[k+1]
97          for( ss = 0; ss < nrowsI; ss++)
                v[k+1][ss] = u[ss] / h[k+1][k];
99
            for ( j = 0 ; j < k; j++)
101         {
                delta = h[j][k];
103             h[j][k] = c[j] * delta + s[j] * h[j+1][k];
                h[j+1][k] = -s[j] * delta + c[j] * h[j+1][k];
105         }
            gamma = sqrt(h[k][k] * h[k][k] + h[k+1][k]*h[k+1][k]);
107         c[k] = h[k][k] / gamma;
            s[k] = h[k+1][k] / gamma;
109         h[k][k] = gamma;
            h[k+1][k] = 0.;
111         delta = g[k];
            g[k] = c[k] * delta + s[k] * g[k+1];
113         g[k+1] = -s[k] * delta + c[k] * g[k+1];
            resi[k] = fabs(g[k+1]);
115         //report the residual on the root process
            if ( my_rank == root )
117             printf("%d %e\n", k+1, resi[k]);
            if( resi[k] < gmres_res)
119         {
                                //compute alpha
121             //allocate alpha
                double *alpha = (double *)calloc(k+1 , sizeof(double) );
123             //solve backward
                for ( j = k ; j >= 0; j--)
125             {
                    alpha[j] = g[j]/h[j][j];
127                 for ( ss = (j+1) ; ss <= k; ss++)
                        alpha[j] -= (h[j][ss]/h[j][j] * alpha[ss]);
129             }
                                //compute zk
131             double *zk = (double *)calloc(nrowsI , sizeof(double) );
                for ( j = 0 ; j <= k; j++)
133                 for ( ss = 0 ; ss < nrowsI; ss++)
                        zk[ss] += alpha[j] * v[j][ss];
135
                if( precond_flag ) //should it be preconditioned?
137                 for ( i = 0 ; i < nrowsI; i++ )
                        if ( diag[i] )
139                         zk[i] /= diag[i];

141                         //compute solution
                double *X = (double *)calloc(nrowsI , sizeof(double) );
143             for ( ss = 0 ; ss < nrowsI; ss++)
                    X[ss] = x0[ss] + zk[ss];
145                     //report the solution
                print_matrix_double("x", X, nrowsI, 1);
147
                                //clean ups
149             ....

151                             //terminate
                break; //terminate everything immediately!
153         } // end of if statement
        } //end of the main gmres loop
155
        //clean ups and finishing MPI session.
```

In line 5 each process reads its won share of the input matrix. In line 7 each process reads the corresponding section of right hand side array. The process of localization (hacking indices) is done in line 11. Then to increase the performance of the code we pick the diagonal entries of the matrix for only one time in line 19-25 and store them in array `diag`. So in each part of preconditioning when we need to have diagonal entries, instead of looping over matrix `A` we just use the values stored at this array.

In line 29 the initialization is done using 1. We will see that if we initialize with zero we will get different convergence behavior in the following sections. In lines 31-51 the initial vector of Krylov subspace is created based on normalization of the residual of the initial guess. Also this residual is stored in `g [ 0 ]` (the right hand side vector.

Then we will have the main GMRES loop in line 53 where the columns of the Krylov matrix are eventually created during this loop. Please note that the final implementation of GMRES which is presented in appendix I includes an outer loop for restarting. In lines 55-61 there is a section for dynamic allocation using `realloc` function which is neglected here. In lines 67-87 depending on the user flag for preconditioning, the appropriate type of matrix-vector multiplication is selected and applied. Now to make the resulting vector orthogonal to the previous vectors, a modified Gram-Schmidtt procedure is implemented in lines 87-93 where the components of the new vector along the previous vectors is gradually eliminated to have one final component which is normal to all previous vectors. Finally based on the normalization of the resulting vector, a new search vector is created in lines 95-99. Then previous givens are applied to the new column of Hessenberg matrix in lines 99-105. Once this is done we need to compute new givens to apply them the new column of the Hessenberg matrix. This is done in line 107-111. In lines 111-115 given are applied to the RHS vector `g` where we obtaine the residual of the least-square problem and store it in `g [ k+1 ]`. This is interesting because without solving $r = b - Ax$ and doing matrix vector multiplication we directly know the residual from the least-square problem that we solved using givens.

Then in line 119 we decide that if the residual is less than particular value, then we terminate the gmres loop and compute the final optimal update and report the solution. In this case the optimal coefficient α is obtained by backward solution of the upperdiagonal system in lines 123-129. Then the update value `zk` is computed and in the case of preconditioning it is divided by the diagonal entries. Finally the final solution is computed in line 143 and reported to user in line 147.

0.2 Verification and Validation

In this section, we verify the implementation.

0.2.1 Matrix A : dense6x6.mtx

The first argument in the code is the location and name of input matrix, the second argument is the RHS matrix, the third argument is the number of GMRES iterations and the last argument is 0 for normal GMRES and 1 for preconditioned GMRES algorithm. Running the parallel gmres code for k=6 and p=1 we obtain

```
$mpirun -np 1 ./gmres ../matrices/dense6x6.mtx (void) 6 0

$ cat out.0

 contents of x :
0.18861997210055,
-0.15060264446066,
1.02527163328527,
-0.43822000212772,
0.98705425807485,
-1.21946494911443,
```

Note that since we don't read the rhs matrix (and actually in the source code I simply comment it here!), I put an arbitrary string `(void)` to preserve the number and arrangment of the arguments.

For k=6 and p=2, the parallel code gives us

```
$mpirun -np 2 ./gmres ../matrices/dense6x6.mtx (void) 6 0

$ cat out.0 out.1

contents of x :  ---> process 0
0.18861997210055,
-0.15060264446066,
1.02527163328527,

 contents of x : ---> process 1
-0.43822000212772,
0.98705425807485,
-1.21946494911443,
```

For k=6 and p=3, the parallel code gives us

```
$mpirun -np 3 ./gmres ../matrices/dense6x6.mtx (void) 6 0

$ cat out.0 out.1 out.3

 contents of x :  ---> process 0
0.18861997210055,
-0.15060264446066,
```

```
 contents of x :  ---> process 1
1.02527163328527,
-0.43822000212772,
 contents of x :  ---> process 2
0.98705425807485,
-1.21946494911443,
```

For the extreme case of six processes p=6, the parallel code gives us

```
$mpirun -np 6 ./gmres ../matrices/dense6x6.mtx (void) 6 0

$ cat out.0 out.1 out.2 out.3 out.4 out.5

 contents of x : ---> process 0
0.18861997210055,

 contents of x : ---> process 1
-0.15060264446066,

 contents of x : ---> process 2
1.02527163328527,

 contents of x : ---> process 3
-0.43822000212772,

 contents of x : ---> process 4
0.98705425807485,

 contents of x : ---> process 5
-1.21946494911443,
```

Using the GMRES program written in GNU Octave (Appendix II), we will obtain

```
ans =

   0.188619972100554
  -0.150602644460663
   1.025271633285273
  -0.438220002127719
   0.987054258074850
  -1.219464949114430
```

which is in exact agreement with the parallel C-code. The residual history (including the initial residual before GMRES main for loop) is plotted in fig.(0.2.1). As we see, it converges to machine zero after 6 iterations as we expected before.

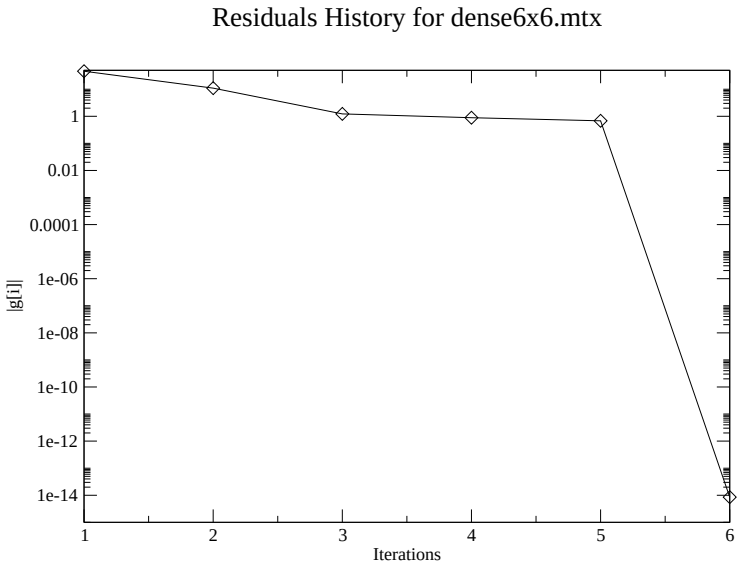

Figure 1: The absolute value of gi vector (residuals) versus iteration i for dense6x6 matrix.

0.2.2 Matrix B : fidapm05.mtx

In this section we consider the residual history of matrix fidapm05.mtx which is a 42x42 sparse matrix. The structure of the matrix is shown in fig.(2).

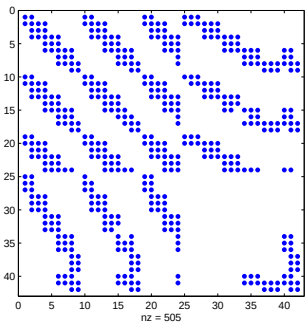

Figure 2: The structure of matrix B.

First we set k=42 for the case that we have all search vectors. Therefore we expect that the residual should reach to machine zero. This is shown in fig.(3).

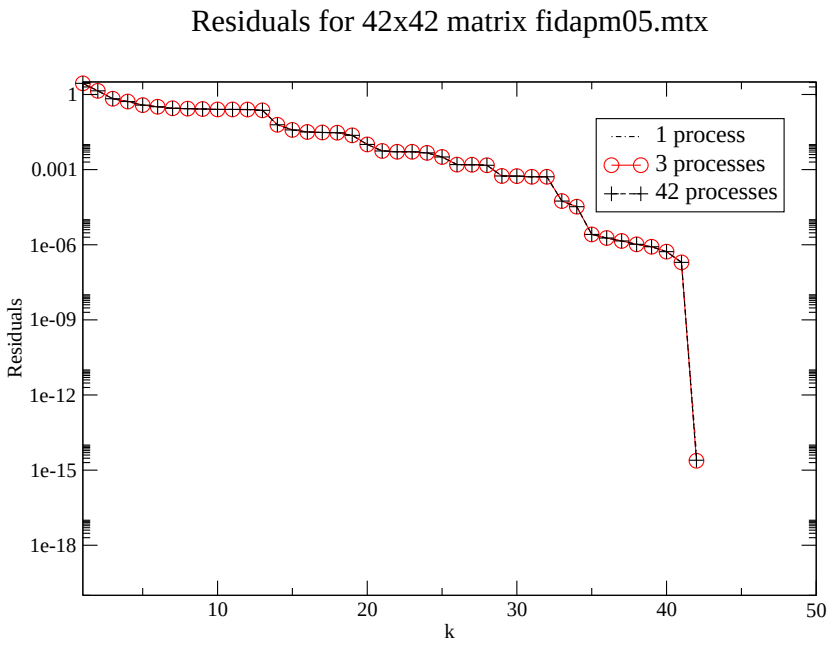

Figure 3: Residuals for Matrix B for various processes.

To validate the parallel implementation, here we increase the number of processes to $p = 3$ and $p = 42$. As shown the residual curves are identical.

Another examination is to test the effect of preconditioning. For this purpose we first read the RHS vector for 42x42 given in the input file `fidamp05_rhs1.mtx`. The residual curves for GMRES with preconditioning and without preconditioning are shown in fig.(4).

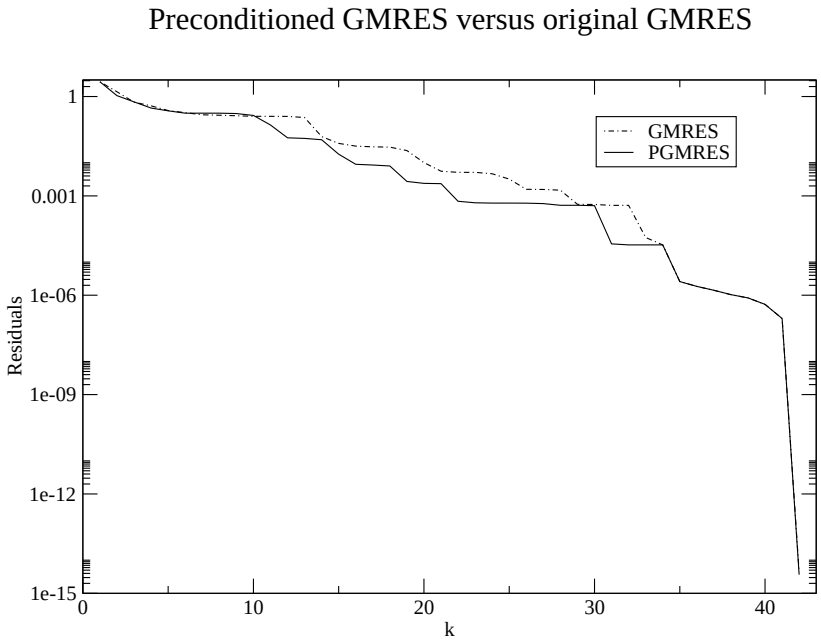

Figure 4: The effect of preconditioning on the convergence of 42x42 matrix.

As we see there isn't much difference (compared to the matrix D that will be discussed in the following section) between residual curves. Thus the procedure of diagonal preconditioning does improve the convergence in the intermediate stages but generally there is no big difference. This is mainly because the matrix B42x42 is highly off-diagonal[1] so when we approximate it with only diagonal entries, we induce considerable ambiguity into the preconditioner. Therefore we expect that diagonal preconditioning shouldn't work in this case as perfectly as it works for close-to-diagonal matrices. Here I expect that if we use the tridiagonal form of matrix B42x42 instead of only main diagonal, the convergence should improve.

0.2.3 Matrix D : s3dkq4m2.mtx

This a huge 90448x90448 matrix will all non-zero elements close to the main diagonal. So we expect that diagonal preconditioning works well for this matrix because the main diagonal seems to be a good estimate of the matrix D itself. Here we fixed the number of GMRES iterations to 40 and without restarting we run the code with/without preconditioning option for one, two and sixteen processes. The results are presented in fig.(5).

[1] Please refer to fig.(2).

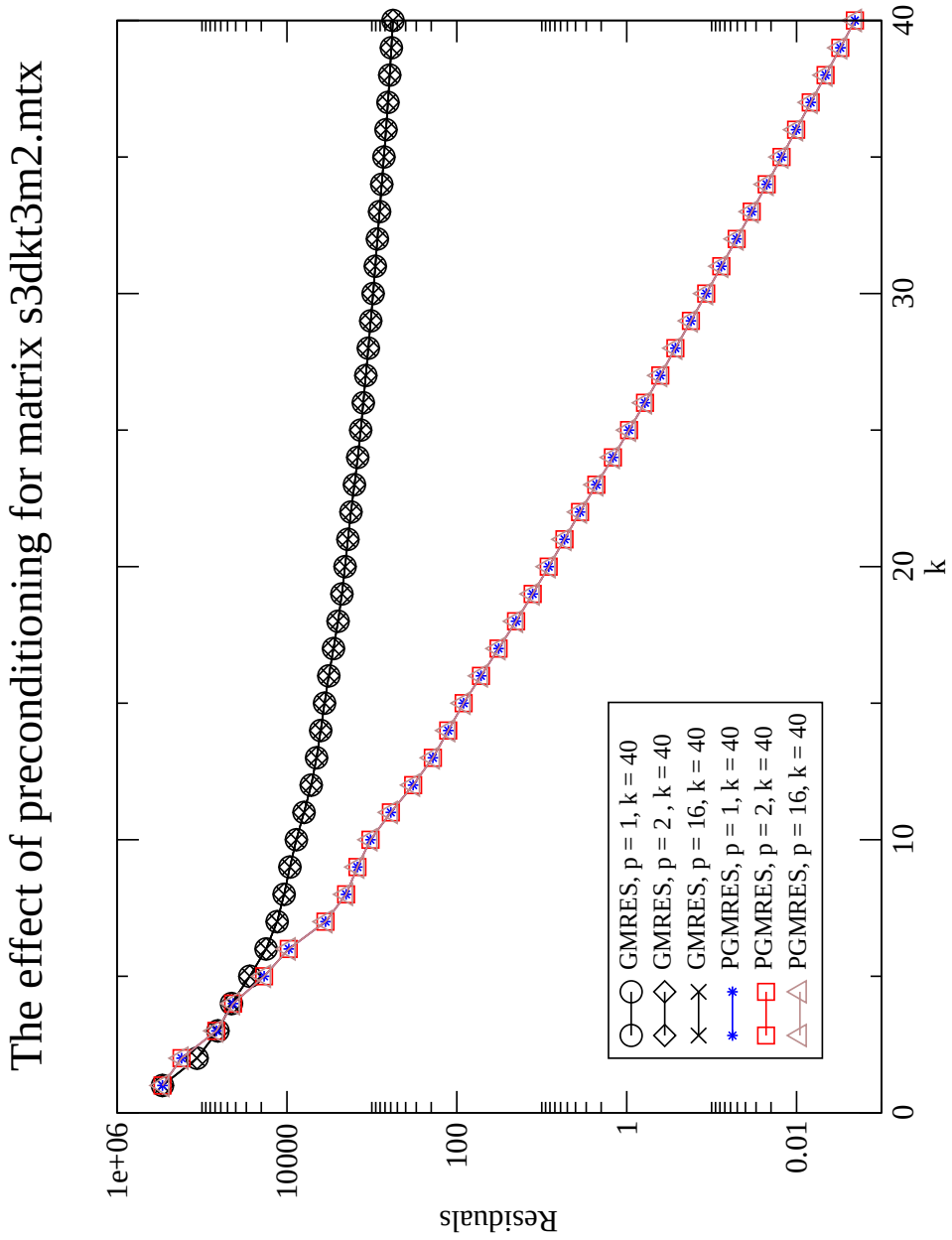

Figure 5: Preconditioned GMRES versus original gmres in solving $Dx = b$.

As we have expected, the diagonal preconditioning greatly improves the convergence of the original GMRES algorithm. As mentioned before, the reason for improving convergence for matrix D is that the main diagonal is a good choice for estimating the matrix since the matrix is close-to-the-main-diagonal oriented. We also notice that variation in the number of processes has absolutely no effect in the convergence curves and again curves are identical (to the eye).

0.3 Performance

To investigate the effect of restarting on the performance of PGMRES, we run the code for a couple of test cases represented by PGMRES(k,m) where k is the number of iterations and m is the number of restarts. For $k = 10$, we found that for $m = 8$ restarts, the code converges to 1.e-10. However for larger k values, the number of required restarts greatly reduces. Therefore since number of restarts 'm' should be same for all cases we keep $m = 8$ same for all of them. The residual curves for PGMRES(10,8), PGMRES(20,8), PGMRES(40,8) and PGMRES(80,8) are shown in figs(6, 7, 8, 9) respectively. The wall time for different number of precesses/iterations is measured using `MPI_Wtime( void )` function. The results are presented in table(1). As we see there is no difference in residual curves for different number of processes. For the rest of discussion please see to the last page.

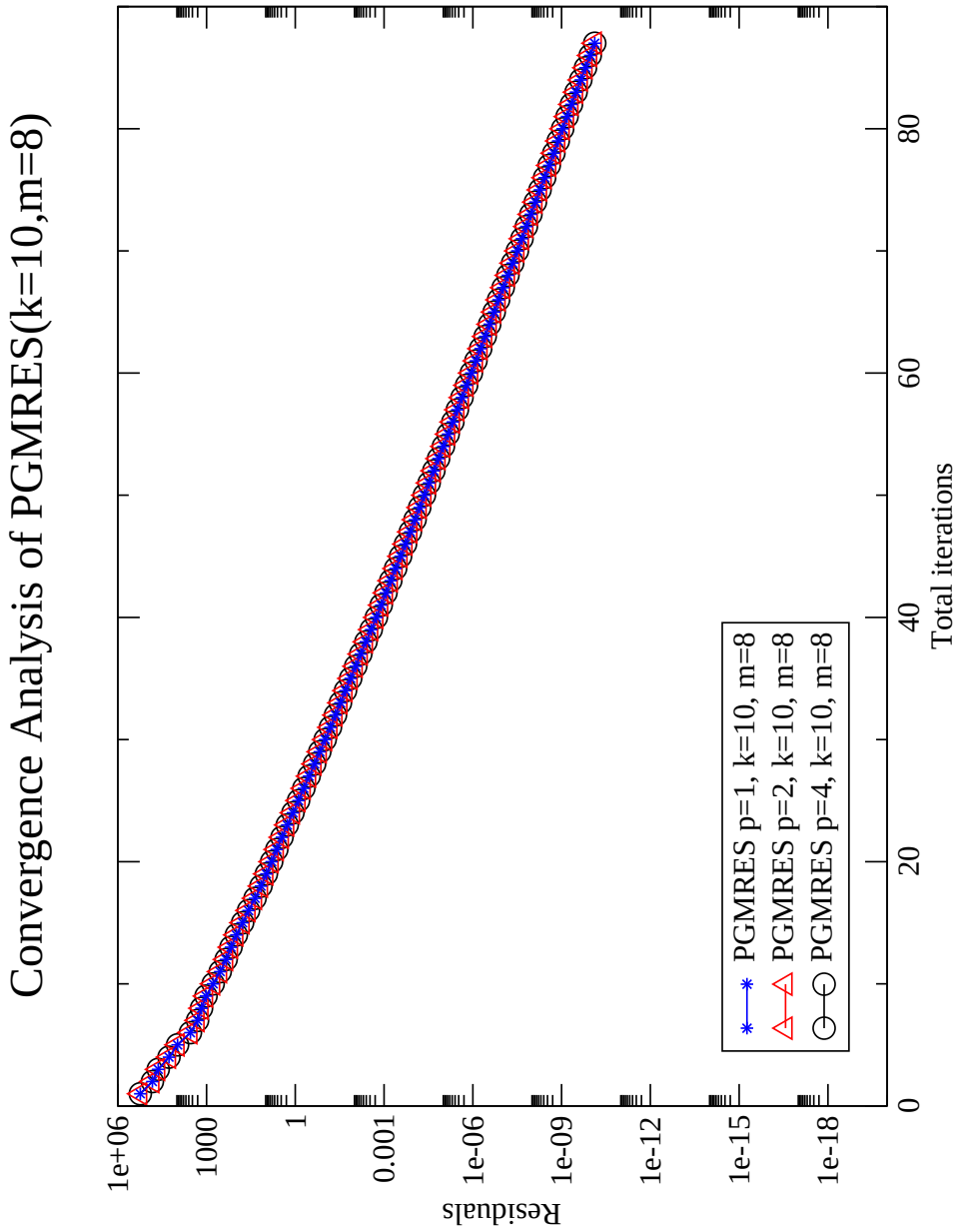

Figure 6: Convergence curves for initial guess $x_0 = 1$..

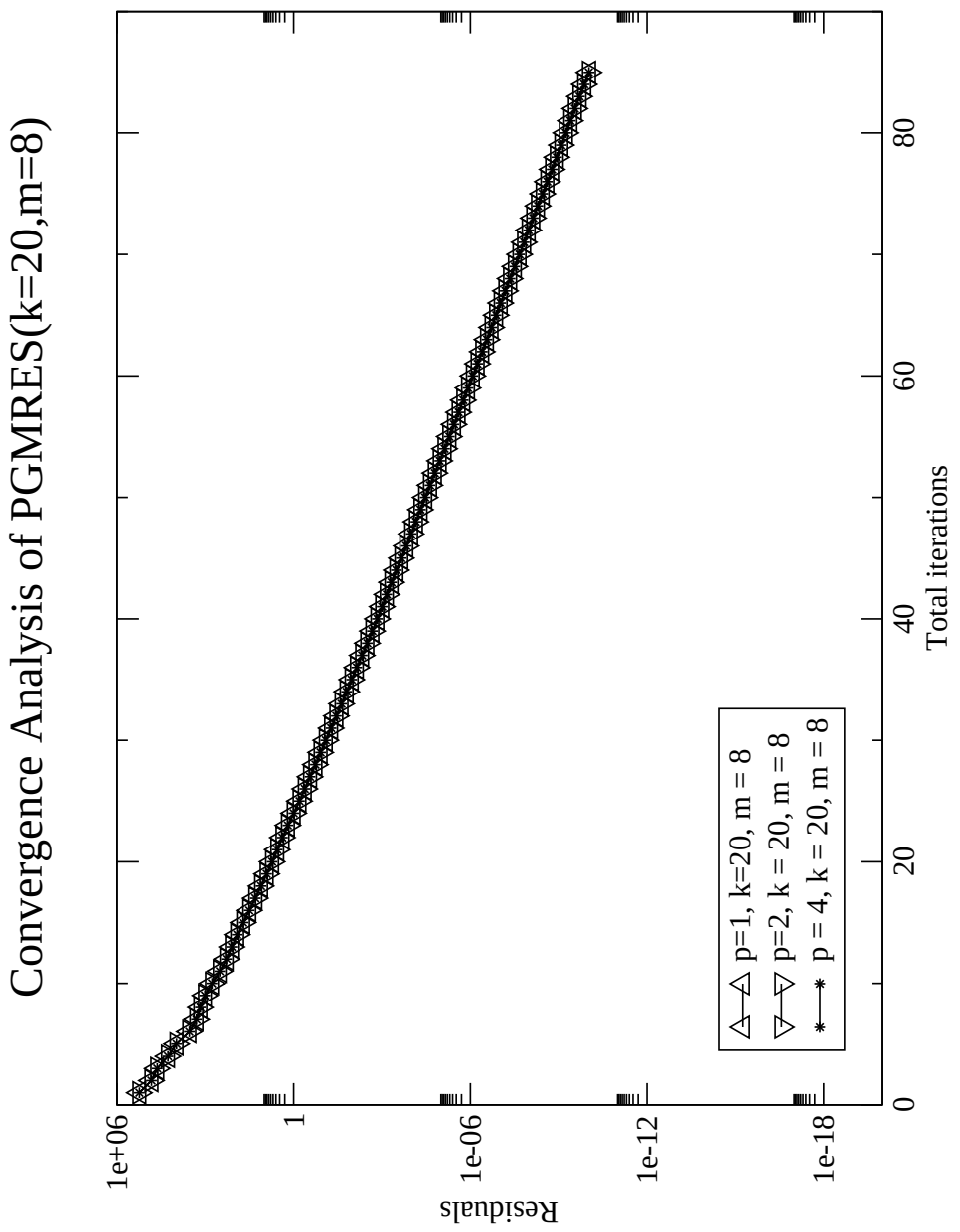

Figure 7: Convergence curves for initial guess $x_0 = 1$..

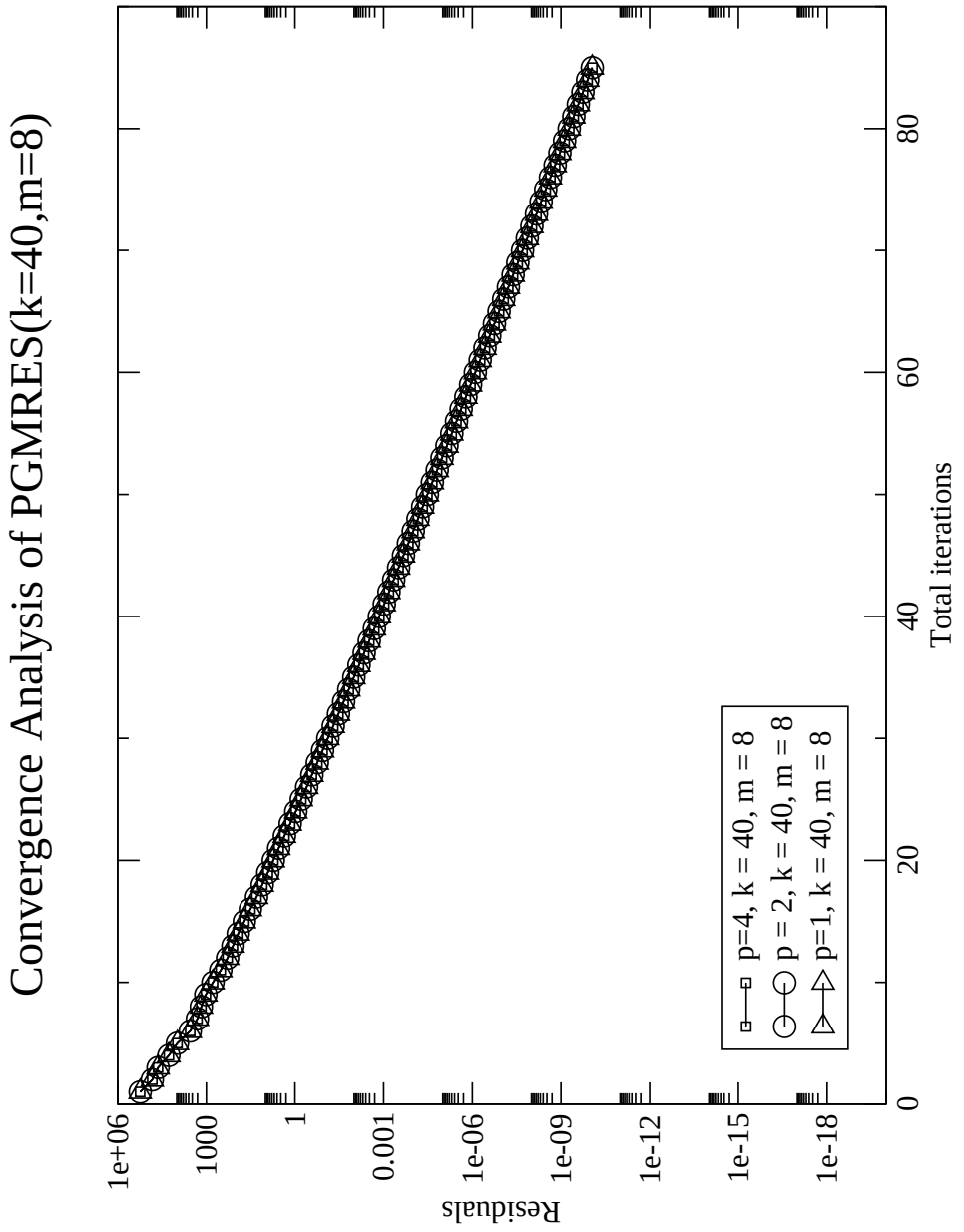

Figure 8: Convergence curves for initial guess $x_0 = 1$..

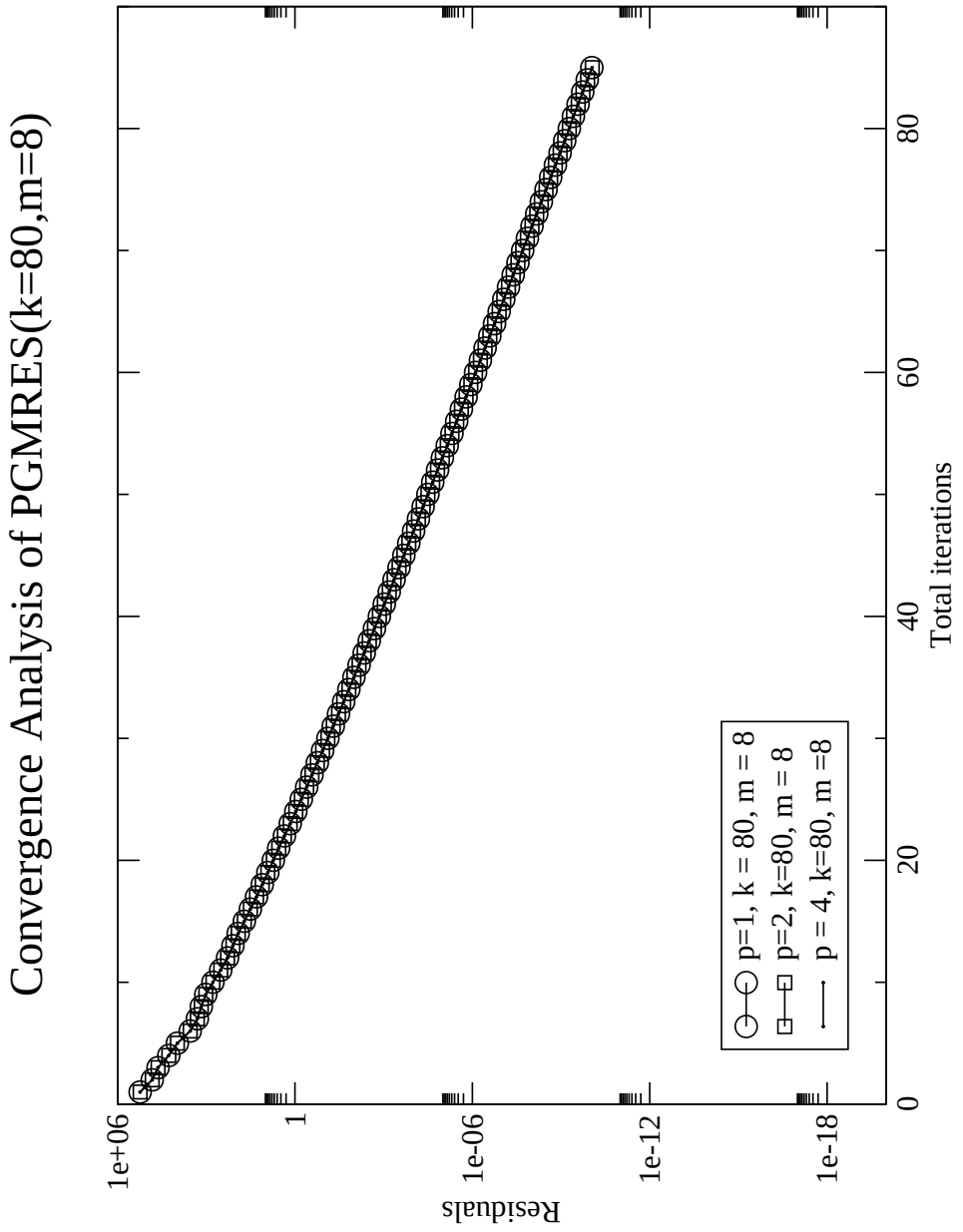

Figure 9: Convergence curves for initial guess $x_0 = 1$..

p	k=10	k=20	k=40	k=80
1	2.693337	3.155758	4.296086	6.696444
2	1.668714	1.719922	2.803091	3.771479
4	0.874024	1.041473	1.482399	1.839851

Table 1: Timing for PGMRES and Matrix D. Values are in seconds.

According to time table(1), we observe this key point that for small number of iterations "k" the computation time decreases. This is mainly because in the GMRES algorithm, there are two interior loops that depends to the value of k.

```
  //main k loop
  ...
for ( j = 0 ; j <= k; j++)
{
     dotproduct(v[j], u, &h[j][k], nrowsI, commun); //dot product
     for( ss = 0; ss < nrowsI; ss++)
     u[ss] -= h[j][k] * v[j][ss];
}

  ... updates

for ( j = 0 ; j < k; j++)
{
     delta = h[j][k];
     h[j][k] = c[j] * delta + s[j] * h[j+1][k];
     h[j+1][k] = -s[j] * delta + c[j] * h[j+1][k];
}
```

The first loop is off course very expensive, because a collective dot product must be performed in each cycle, So for very large number of iterations "k", we expect the number of dot products to dramatically increases. Therefore instead of increasing "k", we prefer to reach to some point in the iteration space and the restart the process again by the new solution.

The result of numerical experiments on the etowah machine validates that the restarting approach decreases computation time. For one process, and $k = 80$, we see that the computation time is 6.7 seconds. If we decrease "k" to 10 we get 2.7 seconds which is 40 percent of case $k = 80$. We also note that the parallelization leads to speed-up which is limited by Amdahl's Law. For $k = 10$ and $m = 8$, we get $(1.61 < 2)$ speed-up when we increase the number of processes from 1 to 2. Also we get $(3.1 < 4)$ when we increase $p = 1$ to $p = 4$. However, for large "k" the speed-up improves. For $p = 1$ to $p = 2$ in case $k = 80$ we get 1.8 speed-up while for $p = 1$ to $p = 4$ in the same case we get 3.6 which is better than 3.1 for case $k = 10$.

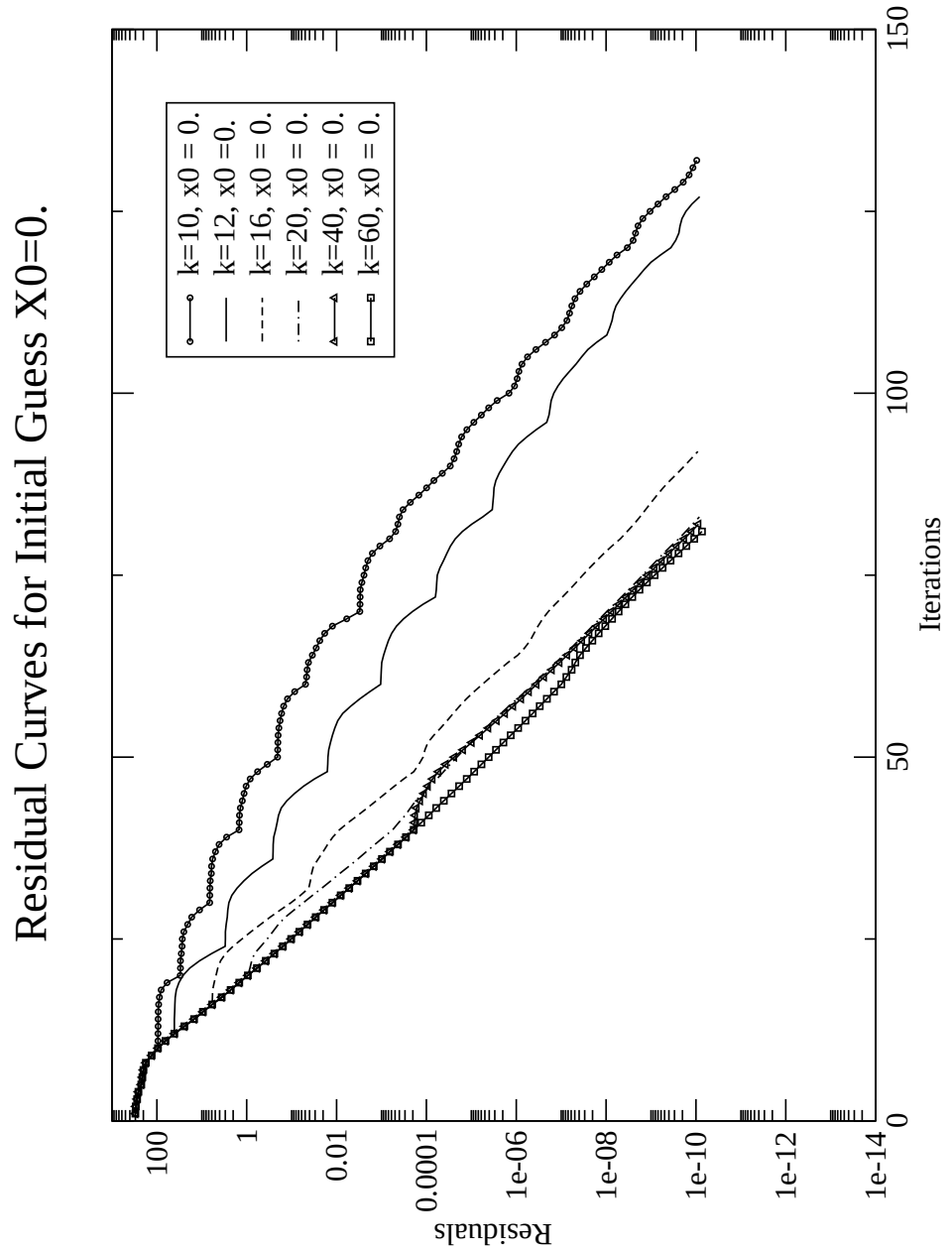

Figure 10: The same experiment for different initial condition $x_0 = 0$..

Another important thing that should be mentioned here is the choice of the initial solution (guessed solution) and the way it affects the residual curves. For the previous cases the initial solution was $x_0 = 1.0$. However we change this condition to $x_0 = 0.$ to see what happens in the residual curves when we restarting. Below the residual curves for restarting case is plotted. As we see the behavior which was a monotonically descending straight curve before is changing to curves which bumps.

As shown in fig.(10), the more the number of gmres iterations increased, the better the solution is estimated before each restart. Therefore another important parameter is the choice of initial guess x_0. In the previous section where we used $x_0 = 1.0$ the reason that residual curves were very similar was that after each restart the solution is a very good estimate of final solution so there is some bump but they are almost flat. But here after each restart, the solution needs to be improved and hence for each search vector that we find the accuracy of the solution dramatically improves and hence we see a visible bump.

0.4 References

[1] D. G. Hyams, "Practical Programming Tips", V1.6, University of Tennessee at Chattanooga, Spring 2012.
[2] D. G. Hyams, "Lecture Notes - Parallel Scientific Computing", University of Tennessee at Chattanooga, Spring 2012.
[3] Y. Saad and M.H. Schultz, "GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems", SIAM J. Sci. Stat. Comput., 7:856-869, 1986.