DEPARTMENT OF MECHANICAL ENGINEERING & MECHANICS
COLLEGE OF ENGINEERING & TECHNOLOGY
OLD DOMINION UNIVERSITY
NORFOLK, VIRGINIA 23529

ITERATIVE METHODS FOR LARGE SCALE STATIC ANALYSIS OF STRUCTURES ON A SCALABLE MULTIPROCESSOR SUPERCOMPUTER

By

Nahil A. Sobh, Principal Investigator

Final Report
For the period ended May 31, 1992

Prepared for
National Aeronautics and Space Administration
Langley Research Center
Hampton, Virginia 23665

Under
Research Grant NAG-1-1291
Ronnie E. Gillian, Technical Monitor
SMD-Computational Mechanics Branch

Submitted by the
Old Dominion University Research Foundation
P.O. Box 6369
Norfolk, Virginia 23508-0369

June 1992

Final Report

ITERATIVE METHODS FOR LARGE SCALE STATIC ANALYSIS OF STRUCTURES ON A SCALABLE MULTIPROCESSOR SUPERCOMPUTER

NAHIL ATEF SOBH[1]

A parallel Preconditioned Conjugate Gradient (PCG) iterative solver has been developed and implemented on the iPSC-860 scalable hypercube . This new implementation makes use of the PARTI (Parallel Automated Runtime Toolkit at ICASE) primitives to efficiently program irregular communications patterns that exist in general sparse matrices and in particular in the finite element sparse stiffness matrices.

The iterative PCG has been used to solve the finite element equations that result from discretizing large scale aerospace structures. In particular, the static response of the High Speed Civil Transport (HSCT) finite element model cited in reference [3] is solved on the iPSC-860.

The preconditioned conjugate gradient algorithm implemented on the iPSC is outline below.

We have parallelizied the following three crucial steps:

1. The matrix vector multiplication (Kp).
2. The inner products ($z^T r$ and $p^T(Kp)$).
3. The solution of the preconditioning step ($Mz = r$).

[1] Principal Investigator, Assistant Professor Department of Mechanical Engineering and Mechanics

1. u_0		(Initialize displacements)
2. $r_0 = f - Ku$		(Initialize residual 'out of balance forces')
3. for $k = 1, 2, ..$ do		(Iteration loop on k)
4. $if \quad \|\|r_{k-1}\|\| \leq$ tolerance, exit		(If equilibrium is satisfied 'exit')
5. $Solve \quad Mz_{k-1} = r_{k-1}$		(Solve for the iteration vector)
6. $\beta_k = \frac{z_{k-1}^T r_{k-1}}{z_{k-2}^T r_{k-2}}$	$(\beta_1 = 0)$	(Evaluate the scaling parameter)
7. $p_k = z_{k-1} + \beta_k p_{k-1}$	$(p_1 = z_0)$	(Update search direction)
8. $\alpha_k = \frac{z_{k-1}^T r_{k-1}}{p_k^T K p_k}$		(Evaluate new step length)
9. $u_k = u_{k-1} + \alpha_k p_k$		(Update displacements)
10. $r_k = r_{k-1} - \alpha_k K p_k$		(Update residuals 'out of balance forces')
11. end of for loop		(go to 3)

Good preconditioners are those which approximate or mimic the operational role of the original problem. Thus the preconditioning step given in the PCG algorithm involves solving a system, $Mz = r$, which is very close to the original system, $Ku = f$. The communications overhead constraint imposed by MIMD machines limit the use of highly coupled direct solvers. Therefore we have used an approximation to K that has a loosely coupled structure. In particular, we have used a diagonal, i.e., point *Jacobi* preconditioner. *Jacobi* type preconditioners parallelize very well on MIMD and SIMD supercomputer architectures. Although they may look simple it has been shown that their overall performance can be very competitive when compared to other preconditioners. We have also implemented a block SSOR family of preconditioners. The SSOR family of preconditioners accelerate the convergence rate but at the expense of extra computations and less communications. The current work is focusing on the trade off between communications and computations when using the SSOR preconditioners.

Computer listing of the C program is given in the appendix.

This research has accomplished the following tasks :

1. Implemented a conjugate gradient iterative solver for symmetric and sparse matrices on the Intel iPSC-860 hypercube at ICASE.
2. Implemented SSOR and point -Jacobi preconditioners to accelerate convergence of the conjugate gradient iterative solver and reduce communications.
3. Generated large-scale finite element models of structures using the COMET software at NASA Langley Research Center. Performed linear static analysis on the generated structural models on the Intel iPSC-860 at ICASE/NASA-Langley.
4. A general purpose software has been implemented to generate full stiffness matrices given the symmetric part from COMET.

No. of Processors	**CPU time (seconds) diagonal preconditioning**	**CPU time (seconds) SSOR**
High Speed Civil Transport (HSCT) (16,146 equations)		
8	1342	
16	786	764
32	565	438
Blade Siffened Panel (BSP) (1,824 equations,)		
1	2.5	3.0
2	2.1	2.1
4	1.8	1.4
8	1.9	1.3
16	1.7	1.2

The table shown above summarizes the performance of the preconditioned conjugate gradient iterative solver. The first column is the number of processors used to solve the above mentioned problem. the second column corresponds to the amount of cpu time in seconds required to solve a given problem using the diagonal of the stiffness matrix as a preconditioner. The third column is the cpu time in seconds required to solve a given problem using a symmetric successive over relaxation preconditioner.

The speedup performance on the large problem (HSCT) is modest. The speedup performance is expected to be better as the number of equations is increased. Although one may think that a direct solver will be much more efficient on these problems, one should not ignore the fact that direct solvers require extensive amount of storage which limit the amount of equations one can solve. Thus ***iterative solvers will find most of their efficient use on super large problems*** which exceed in their size the core memory available on a given supercomputer. This limitation should drive one to use out-of-core solvers which are slow compared to iterative solvers and are not easy to parallelize. Therefore efforts should concentrate on developing efficient parallel out-of-core solvers and iterative solvers with emphasis on parallelizable preconditioners to efficiently use the future technology of massively parallel supercomputers.

References

1. Berryman, H. and Saltz, J. *A Manual for PARTI Runtime Primitives*, ICASE Interm Report 13, September 1990.

2. Golub, G. H. and Van Loan, C. F., *Matrix Computations*, The John Hopkins University Press, 1983.

3. Poole, E. L., Knight, N. F. and Davis, D. D., *High-Performance Equation Solvers and Their Impact on Finite Element Analysis*, Finite Element Methods in the Design Process, ed. by Robinson, J., England, 1990

Appendix

Computer Listing

```
#include <stdio.h>

#define BUFFER_SIZE 1000  /* Length of the buffer array          */
#define ROW_COEF_MSG 111  /* I.D. for stiffness coefficients     */
#define ROW_INDX_MSG 222  /* I.D. for row-location coefficients */
#define SETUP_MSG 22323   /* I.D. for                            */
#define MAX_NUM_PROC 32   /* Maximum number of processors        */
main(argc,argv)
int argc;
char *argv[];
{
  int i,                  /* Dummy integer counter                    */
      j,                  /* Dummy integer counter                    */
      buffer_size[MAX_NUM_PROC], /* Array of buffers sizes for sending data. */
                          /* Max MAX_NUM_PROC, reflects the number of  */
                          /* processors available (update to 1024,..) */
      indx_buffer[MAX_NUM_PROC]
         [BUFFER_SIZE]; /* Array of buffers for row-location coefficients  */
                          /* distributed over available processors (max=MAX_NUM_PROC) */

  int rows_per_proc,      /* No. of rows assigned to a processor */
      setup_buffer[1 + MAX_NUM_PROC];/*                                */

  int nmyrows[MAX_NUM_PROC],      /* Integer number of rows per processor */
      nmysends[MAX_NUM_PROC],     /*                                      */
      nonzeros,           /*                                  */
      nrows,              /* No. of rows                      */
      *ptr,               /* An integer pointer               */
      proc,               /* A processor counter              */
      ncols;              /* Number of columns                */

  int neqns,              /* Number of equations              */
      junk,
      lk11,
      numjnt,             /* Number of nodes                  */
      dof,                /* Degrees of freedom per node      */
      jdof,               /* Total number of unconstraint equations */
      lth,                /* Row-length for sparse matrix storage   */
      nproc;              /* Number of processors allocated         */

  int remainder,          /*                                  */
      curr_max;           /*                                  */


  float coef_buffer[MAX_NUM_PROC][BUFFER_SIZE], /* Stifness coefficients   */
                                  /* in lower triangluar      */
                                  /* form.                    */
       *ptr2;                     /* Pointer to a floating    */
                                  /* point number.            */

  float tol;  /*CG param. */
  int   max_itr; /*CG param. */

 FILE *fp,     /* Generic file pointer        */
      *ptrs,   /*                             */
      *indxs,  /* Index location file         */
      *coefs,  /* Stiffness coefficients file */
      *diag,   /* Diagonal stiffness file     */
      *forc,   /* Nodal Forces file           */
      *info;   /* Problem Statement file      */

 /* ------------------------------------------------- */
 /* Check if input files are available then open them. */
 /* ------------------------------------------------- */


 /*
  * ------------------
  * Problem definition.
  * ------------------
  */

 if( ( info=fopen("K.INFO","r") ) == NULL )
   {
     fprintf(stderr,"Error: cannot open %s for input\n","K.INFO");
     exit(1);
   }

 /*
  *
  */

 if( ( ptrs=fopen("K.PTRS","r") ) == NULL )
   {
     fprintf(stderr,"Error: cannot open %s for input\n","K.PTRS");
     exit(1);
   }

 /*
  * ------------------------------------------------
  * Columns locations for sparse matrix data-structure.
  * ------------------------------------------------
  */

 if( ( indxs=fopen("K11.INDXS","r") ) == NULL )
   {
     fprintf(stderr,"Error: cannot open %s for input\n","K11.INDXS");
     exit(1);
   }

 /*
  * ------------------------------------------------------
  * Stricly lower triagular stiffness matrix coefficients.
  * ------------------------------------------------------
  */

 if( ( coefs=fopen("K11.COEFS","r") ) == NULL )
   {
     fprintf(stderr,"Error: cannot open %s for input\n","K11.COEFS");
     exit(1);
   }

 /*
  * --------------------------------
  * Diagonal stiffness coefficients.
  * --------------------------------
  */

 if( ( diag=fopen("K.DIAG","r") ) == NULL )
   {
     fprintf(stderr,"Error: cannot open %s for input\n","K.DIAG");
     exit(1);
   }


 /*
  * -------------------
  * Applied nodal forces
  * -------------------
```

```
 */

if( ( forc=fopen("K.FORC","r") ) == NULL )
  {
    fprintf(stderr,"Error: cannot open %s for input\n","K.FORC");
    exit(1);
  }

/*
 * ----------------------------------------
 * Set NX/RX  process id=1 for host program.
 * ----------------------------------------
 */

setpid(1);

/*
 * ------------------------------------------------------------------
 * Load all "-1" nodes  with program "unst4" with process id "0".
 * ------------------------------------------------------------------
 */

load("elp2",-1,0);

/*
 * ----------------------------------------
 * Get the number of total allocated nodes.
 * ----------------------------------------
 */

nproc=numnodes();

/*
 * ------------------------------------------------------------
 * Diagnostic message to compare
 * total number allocated nodes with those defined by the user.
 * ------------------------------------------------------------
 */

printf("begin host: number of processors is %d\n",nproc);


/*
 * ------------------------------------------------------
 * Initialize buffers equal the number of allocated nodes.
 * Initialize number of messages sent to a given node.
 * ------------------------------------------------------
 */

for( i=0; i<nproc; i++ )
   {
     buffer_size[i] = 0;
     nmysends[i] = 0;
   }


/* read info file containing problem size parameters
 * numjnt - number of joints in FEM model
 * dof    - degrees of freedom per joint
 * jdof   - total number of degrees of freedom before constraints
 *          (numjnt X dof)
 * neqns  - number of equations in non-singular input matrix
 *          (jdof - number of constrained dofs)
 * lk11   - number of non-zero coefficients below main diagonal
```

```
   */

  fscanf(info,"%d %d %d %d %d %d %f %d",
      &numjnt, &dof, &jdof, &neqns, &junk, &lk11, &tol, &max_itr);
  printf(" numjnts= %d\t dofs=%d\t total dofs=%d\n neqns=%d\t ncoefs=%d\n tol=%f\t max_it
r=%d\n", numjnt,dof,jdof,neqns,lk11,tol,max_itr);

/*   csend(333,tol,4,-1,0);*/
 /* compute number of rows per processor for blocked row assignment,
  * whenever the number of equations is not an even multiple of
  * the number of processors, the remainng rows are distributed in
  * sequential order to the nodes.
  */

  nrows = neqns;

  rows_per_proc = nrows / nproc;      /* Equal Shares of rows */

/* Error in logic here .. feb. 26, 1991 */
/*  remainder = nrows % rows_per_proc; */

/* Remaining shares if not zero */
  remainder = nrows % nproc;

  printf("remainder = %d \n", remainder);
  if( remainder > 0 )
     curr_max = rows_per_proc;
  else
     curr_max = rows_per_proc - 1;

/*
 * for each row determine which processor then read in that row
 * from data files and place coefficients in the appropriate buffer
 */

 proc=0;
 for( i=0; i<nrows; i++ )
    {
      if( i > curr_max )
        {
          if( remainder>proc+1 )
              curr_max = curr_max + rows_per_proc + 1;
          else
              curr_max = curr_max + rows_per_proc;
          proc=proc+1;
        }

      if( (proc>=nproc) | (proc<0) )
        {
          fprintf(stderr,"Grrrrrrr.....nproc=%d, proc= %d\n",nproc,proc);
        }

      /*
       * get number of nonzeros in current row
       */

      fscanf(ptrs,"%d",&lth);

      /*
       * if buffer doesn't have enough room, flush it.
       */

      if( BUFFER_SIZE<=buffer_size[proc] + lth+2 )
        {
```

```
/*            printf("sending rows off to processor %d \n",proc);

            printf(" message: type, proc, length \n %d, %d, %d \n",
                   ROW_COEF_MSG,proc,buffer_size[proc]);

            printf(" message: type, proc, length \n %d, %d, %d \n",
                   ROW_INDX_MSG,proc,buffer_size[proc]);
*/

            csend( ROW_INDX_MSG,indx_buffer[proc],
                   sizeof(int)*buffer_size[proc],proc,0);

            csend( ROW_COEF_MSG,coef_buffer[proc],
                   sizeof(float)*buffer_size[proc], proc,0);

            nmysends[proc]++;

            buffer_size[proc] = 0;
          }

        /*
         * Add current row to buffer for appropriate processor
         * First, add lth (number of coefs in this row) and row number
         * to index_buffer;
         * Second, add the diagonal coefficient for this row to first
         * location in coef_buffer for this row (i.e. ptr2[0])
         * Finally, add the column index values
         * to index_buffer and actual coefficients to coef_buffer.
         */

        ptr  = &indx_buffer[proc][buffer_size[proc]];

        ptr2 = &coef_buffer[proc][buffer_size[proc]];

        buffer_size[proc] += 2+lth;

        ptr[0] = i+1;

        ptr[1] = lth;

        fscanf(diag,"%f",&ptr2[0]);
        fscanf(forc,"%f",&ptr2[1]);

        for( j=0; j<lth; j++ )
           {
             fscanf(indxs,"%d",&ptr[j+2] );
             fscanf(coefs,"%f",&ptr2[j+2]);
           }

    }

/*
 * Flush all buffers with something in them
 */
for( i=0; i<nproc; i++ )
   {
     if( buffer_size[i] > 0 )
       {
/*           printf("sending messages:type, proc, length\n %d,%d,%d\n %d,%d,%d\n",
                   ROW_COEF_MSG,i,buffer_size[i],
                   ROW_INDX_MSG,i,buffer_size[i]);
*/
         csend( ROW_INDX_MSG,indx_buffer[i],
                sizeof(int)*buffer_size[i],i,0);

         csend( ROW_COEF_MSG,coef_buffer[i],
                sizeof(float)*buffer_size[i],i,0);

         nmysends[i]++;
       }
     }

  /*
   * Tell each processor how many messages to expect
   */

  for( i=0; i<nproc; i++ )
     {
       setup_buffer[i] = nmysends[i];
     }

  setup_buffer[nproc] = nrows;

/*   printf(" send message: type, proc, length \n %d, %d, %d\n ",
   SETUP_MSG,-1,nproc+1);
*/

  csend( SETUP_MSG,setup_buffer,sizeof(int)*(numnodes()+1),-1,0);

  /*
   * close all the files openned at the beginning
   */

  fclose(info);
  fclose(indxs);
  fclose(coefs);
  fclose(ptrs);
  fclose(diag);

  waitall(-1,0);
}
```

```
/*
 * node program.
 */

#include <cube.h>
#include <stdio.h>
#include <math.h>
#include "parti.h"
#include "main.h"

main(argc,argv)
int argc;
char *argv[];
{
  int i,ii,
      j,
      count;
  int n_iter;

  TTABLE *table;

  SCHED *sr;

  float *z;

  long start_time;  /* time in milliseconds */
  long elapsed_time;/* pcg timing in milliseconds */

  int n_off_proc;

  me = mynode();        /*  Get each node number */

  sprintf(name,"out.%d",me);

  myfile=fopen(name,"w");

  printf(" begin node program processor %d filename= %s \n",me,name);

  fprintf(myfile," begin node program processor %d \n",me);
  fflush(myfile);

  /*
   * ------------------------------------
   * Get sparse matrix from host program.
   * ------------------------------------
   */



  get_sparse_mat();

  fprintf(myfile," my nonzeros =  %d \n",Mynonzeros);
  fflush(myfile);


  /*
   * ------------------------------------------------------
   * Build translation table by scattering Row to the table.
   * IN: Row[i]       OUT: table
   * ------------------------------------------------------
   */
```

```
  table = build_translation_table(BLOCKED,Row,Myrows);
  localize(table,&sr,Cols,LocalCols,Mynonzeros,&n_off_proc,Myrows);

  /*
   * ----------------------------------------
     Solve Kx = f (Iteratively using pcg algor.)
   * ----------------------------------------
   */

  z  = (float *) malloc(sizeof(float)*Myrows);

  start_time = mclock();
             z  = pcg(sr,n_off_proc,Force,0.001,50000,&n_iter);
  elapsed_time = mclock() - start_time;

  fprintf(myfile," elapsed time = %d \n number of iterations = %d\n",
                  elapsed_time, n_iter);
  printf(" elapsed time = %d \n number of iterations = %d",
                  elapsed_time, n_iter);



  pr_float_array( "z", z, Myrows);

}
 /* END OF NODE PROGRAM */



/* ------------------------------------------- */
/* This function is used to read in the sprse mat. */
/* It should be ignored if at all possible.        */
/* ------------------------------------------- */


void get_sparse_mat()
{
  int size,
      indx_buffer[BUFFER_SIZE];

  float coef_buffer[BUFFER_SIZE];

  int type,
      rows_expected;

  rows_expected = -1;
  Myrows = 0;
  Mynonzeros = 0;

  gsync();


  while( (Myrows<rows_expected) | (rows_expected<0) )
       {
         cprobe(-1);

         type = infotype();

         size = infocount()/sizeof(int);


        if( type==ROW_INDX_MSG )
          {
            crecv( ROW_INDX_MSG,indx_buffer,size*sizeof(int));
```

```
            crecv( ROW_COEF_MSG,coef_buffer,size*sizeof(float));
            unpack_row_data(indx_buffer,coef_buffer,size);
          }

        if( type==SETUP_MSG )
          {
            crecv(SETUP_MSG,indx_buffer,size*sizeof(int));
            rows_expected = indx_buffer[mynode()];
            Nrows = indx_buffer[numnodes()];
          }
      }
  gsync();

}

/* ---------------------------------------- */
/*                                          */
/* ---------------------------------------- */


void unpack_row_data(indx_buffer,coef_buffer,size)
int *indx_buffer,size;
float *coef_buffer;
{
  int count,
      i,
      j,
      row,
      ncols,
      count2,
      ixx,
      ist;

  float sum;

  static int col_count = 0;


  for( count=0; count<size; )
     {
       Row[Myrows] = indx_buffer[count];
       Diags[Myrows] = coef_buffer[count];
       Force[Myrows] = coef_buffer[count+1];
       ncols = Ncols[Myrows] = indx_buffer[count+1];
       count=count+2;
       Mynonzeros += ncols;

       if( Myrows >= MAX_ROWS )
         {
           fprintf(stderr,"Error on node %d : too many rows!!!\n",mynode());
           exit();
         }

       if( Mynonzeros >= MAX_NONZEROS )
         {
           fprintf(stderr,"Error on node %d : too many nonzeros!!!\n",
              mynode());
           exit();
         }

      for( j=0; j<ncols; j++)
         {
           Cols[col_count] = indx_buffer[count];
           Vals[col_count] = coef_buffer[count];
           col_count++;
           count++;
         }

       Myrows++;
     }
    printf("The value of my nonzeros %d\n",Mynonzeros);
  }



/* ---------------------------------------- */
/*                                          */
/* ---------------------------------------- */


void print_my_rows( first_row, last_row, option )
int first_row,
    last_row;
char *option;
{
  int *ptr1;

  float *ptr2,
        *ptr3;

  int i,
      j,
      lth;

  if( strcmp(option,"all") == 0 )
    {
      /* find the starting positions */
      printf(" me=%d, print_my_rows from %d to %d\n",
           me,first_row,last_row);
       fflush(stdout);
      if( first_row > Myrows )
          printf(" me= %d, cannot print rows starting with %d, Myrows=%d\n",
          me,first_row,Myrows);
      else
         {
           ptr1 = &Cols[0];
           ptr2 = &Vals[0];
           ptr3 = &Diags[0];

           for( i=0; i<first_row-1; i++ )
              {
                 ptr1 = ptr1+Ncols[i];
                 ptr2 = ptr2+Ncols[i];
                 ptr3++;
              }

           if( last_row > Myrows )
             {
               printf(" me= %d, cannot print rows past %d, Myrows=%d\n",
               me,last_row,Myrows);
               last_row=Myrows;
             }

           for( i=first_row; i<=last_row; i++ )
              {
                lth=Ncols[i-1];
                fprintf(myfile," Myrow= %d, Row= %d, Diag Coef= %f, lth= %d\n",
                        i-1,Row[i-1],*ptr3,lth);
```

```
                fflush(myfile);
                ptr3++;

                for( j=0; j<lth; j++ )
                   {
                     fprintf(myfile,"  column= %d, Coef= %f\n",*ptr1,*ptr2);
                     fflush(myfile);
                     ptr1++;
                     ptr2++;
                   }
              }
         }
    }

else if( strcmp(option,"limits") == 0 )
        {
          /* find the first row */
          printf(" me=%d, print_my_rows option=limits\n",me);
          if (first_row > Myrows)
           printf(" me= %d, cannot print row %d, Myrows= %d\n",
              me,first_row,Myrows);
      else{
           ptr1 = &Cols[0];
           ptr2 = &Vals[0];
           ptr3 = &Diags[0];
           for( i=0; i<first_row - 1; i++ )
              {
                ptr1=ptr1+Ncols[i];
                ptr2=ptr2+Ncols[i];
                ptr3++;
              }
           i=first_row;
           lth=Ncols[i-1];
           printf(" Myrow= %d, Row= %d, Diag Coef= %f, lth= %d\n",
              i-1,Row[i-1],*ptr3,Ncols[i]);
           fprintf(myfile," Myrow= %d, Row= %d, Diag Coef= %f, lth= %d\n",
              i-1,Row[i-1],*ptr3,lth);
           ptr3++;
           for( j=0; j<Ncols[i-1]; j++ )
              {
                fprintf(myfile,"  column= %d, Coef= %f\n",*ptr1,*ptr2);
                ptr1++;
                ptr2++;
              }
         }

  /* find the last row */

  if( last_row > Myrows )
     printf(" me= %d, cannot print row %d, Myrows= %d\n",
        me,first_row,Myrows);
  else
     {
       ptr1 = &Cols[0];
       ptr2 = &Vals[0];
       ptr3 = &Diags[0];
       for( i=0; i<last_row-1; i++)
          {
            ptr1=ptr1+Ncols[i];
            ptr2=ptr2+Ncols[i];
            ptr3++;
          }
       i=last_row;
       lth=Ncols[i-1];
```

```
          fprintf(myfile," Myrow= %d, Row= %d, Diag Coef= %f, lth= %d\n",
                i-1,Row[i-1],*ptr3,lth,Ncols[i-1]);
          ptr3++;
          for( j=0; j<Ncols[i-1]; j++)
             {
               fprintf(myfile,"  column= %d, Coef= %f\n",*ptr1,*ptr2);
               ptr1++;
               ptr2++;
             }
        }
  }
  else
    printf(" me= %d, cannot understand print option %s\n",me,option);
}



/* ---------------------------------------- */
/*                                          */
/* ---------------------------------------- */

void pr_int_array( name, array, lth )
int *array,
    lth;

char *name;
{
  int i;

  fprintf(myfile," Integer Array %s follows: lth=%d\n" ,name,lth);

  for( i=0; i<lth; i++ )
     {
       fprintf(myfile," %s[%d]=%d\n",name,i,array[i]);
       fflush(myfile);
     }

}



/* ---------------------------------------- */
/*                                          */
/* ---------------------------------------- */

void pr_float_array( name, array, lth )
float *array;
int    lth;

char *name;
{
  int i;

  fprintf(myfile," Float Array %s follows: lth=%d\n" ,name,lth);

  for( i=0; i<lth; i++ )
     {
       fprintf(myfile," %s[%d]=%f\n",name,i,array[i]);
       fflush(myfile);
     }

}
```

```
/* ------------------------------------------------ */
/* sparse matrix vector multiply function           */
/* require that the schedule be built and passed in */
/* ------------------------------------------------ */


void spmvm(sr,n_off_proc,x,y)
SCHED *sr;  /* <- communication schedule */
float *x,
      *y; /* <- input and result vectors */
int n_off_proc;
{
  int count,
      i,
      j;


  /*
   * ------------------------------------------------------------
   * Gather data using previously computed communication schedule.
   * ------------------------------------------------------------
   */

  fgather(sr,&x[Myrows],x);
  count = 0;

  for( i=0; i<Myrows+n_off_proc; i++ )
    y[i]=0.0;

  for( i=0; i<Myrows; i++ ){
    y[i] += Diags[i]*x[i];
    for( j=0; j<Ncols[i]; j++ ){
      y[i] += x[LocalCols[count]]*Vals[count];
/*      y[LocalCols[count]] += x[i]*Vals[count]; */
      count++;
    }
  }

  fscatter_add(sr,&y[Myrows],y);

}



/* ------------------------------------------------------ */
/* Calculate dot product of distributed vectors x[i],y[i] */
/* Every processor gets copy of the gloabl result         */
/* ------------------------------------------------------ */

float dot(x,y,n)

float *x;
float *y;

int n;
{
  int i;
  float result,wrk;

  result = 0;

  for(i=0;i<n;i++)
    {
     result += x[i]*y[i];
    }

  gssum(&result,1,&wrk); /* global sum */

  return(result);
}
/* ------------------------------------------ */
/* return a*x[i] + y[i] (saxpy vector operation) */
/* ------------------------------------------ */

void saxpy(a,x,y,result,n)

float a;
float *x;
float *y;
float *result;
int n;
{
  int i;

  for(i=0;i<n;i++)
    {
      result[i] = a*x[i]+y[i];
    }
}
/*-------------------------------------*/
float *pcg(sr,n_off_proc,rhs,epsilon,max_iters, n_iter)


SCHED *sr;   /* <-communications schedule for sparse matrix-vector multiply */
int n_off_proc; /* number of fetchs (communications) required */
float *rhs;  /* <- rhs for rhs=Ax */
float epsilon; /* <- tolerance */
int max_iters; /* <- max # of iterations to go through before giving up */
int *n_iter;
{
  float *temp;
  float *z;
  float *r;
  float *x;
  float *p;
  float *Kp;
  float alpha;
  float beta=0.0;
  float old_zTr;
  float new_zTr;
  float pKp;
  float rTr;

  int i;
  int j;

  temp  = (float *) malloc(sizeof(float)*Myrows);
  z  = (float *) malloc(sizeof(float)*Myrows);
  r  = (float *) malloc(sizeof(float)*Myrows);
  x  = (float *) malloc(sizeof(float)*Myrows);
  p  = (float *) malloc(sizeof(float)*(Myrows+n_off_proc));
  Kp = (float *) malloc(sizeof(float)*(Myrows+n_off_proc));

  /* choose initial guess */
  for( i=0; i<Myrows; i++)
```

```
    {
      x[i] = 0;
      r[i] = rhs[i];
    }
      precond(z, r );
 /* - - - - - - - - - - - - - - - */
 /* check if the rhs was null */
 /* - - - - - - - - - - - - - - - */
  new_zTr = dot(z,r,Myrows); /* global dot  stored in each processor */
  rTr     = dot(r,r,Myrows); /* global dot  stored in each processor */
  if( rTr < epsilon)
    {
      *n_iter = 0;
      return(x); /* processor wise */
    }

/* - - - - - - - - - - - - - - - - - - */
/* Initialize the search direction */
/* - - - - - - - - - - - - - - - - - - */
for( i=0; i<Myrows; i++)
    {
      p[i] = z[i]; /* processor wise */
    }

/* - - - - - - - - - - - - - - - - */
/* PCG loop in every processor */
/* - - - - - - - - - - - - - - - - */
if(mynode == 0) fprintf(stderr, "iteration  alpha  beta  (r,r)\n");

for( i =0; i<max_iters; i++)
    {
      spmvm(sr,n_off_proc,p,Kp);  /* K has already been stored don't worry */

      pKp = dot(p,Kp,Myrows);
      alpha = new_zTr / pKp;  /* -- G quantity -- */

      saxpy(alpha,p,x,x,Myrows);   /* new x */
      saxpy(-alpha,Kp,r,r,Myrows); /* new r */

      precond(z, r);

      old_zTr = new_zTr;                /* -- G quantity -- */
      new_zTr = dot(z,r,Myrows);        /* -- G quantity -- */
      rTr     = dot(r,r,Myrows);        /* -- G quantity -- */

      /* print out for checking intermediate results */
      if( mynode()==0 )
        {
          fprintf(stderr,
                  "%d %f  %f  %f %f\n",
                   i,alpha,beta,rTr, pKp);
        }


      if( rTr < epsilon)
        {
          *n_iter = i;
          return(x);
        }

      beta = new_zTr / old_zTr; /* -- G quantity -- */

      saxpy(beta,p,z,p,Myrows);  /* new p */


    }
  *n_iter = i;
  return(x);

}
/*
 * preconditionig with the diagonal elements of the stiffness matrix
 */
void precond( z, r )
float z[];
float r[];
{
  int count,
      i,
      index,
      j,
      k;

  float sum;

 for( i=0; i<Myrows; i++)
    {
       z[i] = 0.0;
    }

/* for( k=0; k<2; k++)    */
    {
      /* - - - - - - - - - - - -  */
      /* forward gauss-siedel */
      /* - - - - - - - - - - - -  */
      count = 0;
      sum = 0.0;
      for( i=0; i<Myrows; i++ )
         {
           for( j=0; j<Ncols[i]; j++ )
              {
                index = LocalCols[count];
                if( index < Myrows )
                  {
                    sum -= z[index]*Vals[count];
                  }
                count++;
              }
           z[i] = ( sum + r[i]) / Diags[i];
         }
      /* - - - - - - - - - - - - - */
      /* backward gauss-siedel */
      /* - - - - - - - - - - - - - */
      count--;
      for( i=Myrows-1; i>=0; i-- )
         {
           sum = 0.0;
           for( j=0; j<Ncols[i]; j++ )
              {
                index = LocalCols[count];
                if( index < Myrows )
                  {
                    sum -= z[index]*Vals[count];
                  }
                count--;
              }
           z[i] = ( sum + r[i]) / Diags[i];
         }
    }
```