COMP/CS 605: Introduction to Parallel Computing
Topic : MPI: Matrix-Vector Multiplication Operations

Mary Thomas

Department of Computer Science
Computational Science Research Center (CSRC)
San Diego State University (SDSU)

Posted: 02/20/17
Updated: 02/21/17

Table of Contents

1. MPI Matrix-Vector Multiplication Operations
   - Basic Matrix Operations
     - Matrix Addition/Subtraction
     - Matrix Inverse
     - Matrix Transpose
   - Matrix-Vector Multiplication
     - Calculating account balances use Matrix-Vector Multiplication
     - Serial Matrix-Vector Multiplication
     - Parallel Matrix-Vector Multiplication - 1D Partitioning Example

Definition of a Matrix

Definition A matrix is a rectangular array of numbers.

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & a_{ij} & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$

element in ith row, jth column

Also written as $A=[a_{ij}]$

m rows

$m \times n$ matrix

n columns

When $m=n$, A is called a square matrix.

Ref: http://www.cse.msu.edu/~pramanik/teaching/courses/cse260/11s/lectures/matrix/Matrix.ppt

Matrix Operations: Addition/Subtraction

Let $A = [a_{ij}]$, $B = [b_{ij}]$ be $m \times n$ matrices. Then:
$A + B = [a_{ij} + b_{ij}]$, and $A - B = [a_{ij} - b_{ij}]$

$$\begin{bmatrix} 1 & -1 \\ 3 & 4 \\ 2 & 0 \end{bmatrix} + \begin{bmatrix} 3 & 4 \\ 1 & -4 \\ 2 & 3 \end{bmatrix} = \begin{bmatrix} 4 & 3 \\ 4 & 0 \\ 4 & 3 \end{bmatrix}$$

$$\begin{bmatrix} 1 & -1 \\ 3 & 4 \\ 2 & 0 \end{bmatrix} - \begin{bmatrix} 3 & 4 \\ 1 & -4 \\ 2 & 3 \end{bmatrix} = \begin{bmatrix} -2 & -5 \\ 2 & 8 \\ 0 & -3 \end{bmatrix}$$

Ref: http://www.cse.msu.edu/~pramanik/teaching/courses/cse260/11s/lectures/matrix/Matrix.ppt

Matrix: Inverse

- Let A be an nxn matrix (square).
- The Inverse of A is denoted by A^{-1}
- A is invertible only when $A \times A^{-1} = A^{-1} \times A = I$
- Where I_n is the identity matrix:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

- Similar but not same as numerical inverse of a number: $1/8 = 0.125$.
- Similar to numbers: $6 * \left(\frac{1}{6}\right) = 1$
- Not all square matrices are invertible.

Inverse: Used to Solve Equations of the form $AX = K$

$$\begin{bmatrix} a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \\ c_1 & c_2 & c_3 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} p \\ q \\ r \end{bmatrix}$$

$$AX = K$$
$$A^{-1}AX = A^{-1}K$$
$$IX = A^{-1}K$$

$$\begin{bmatrix} a_1x + a_2y + a_3z \\ b_1x + b_2y + b_3z \\ c_1x + c_2y + c_3z \end{bmatrix} = \begin{bmatrix} p \\ q \\ r \end{bmatrix}$$

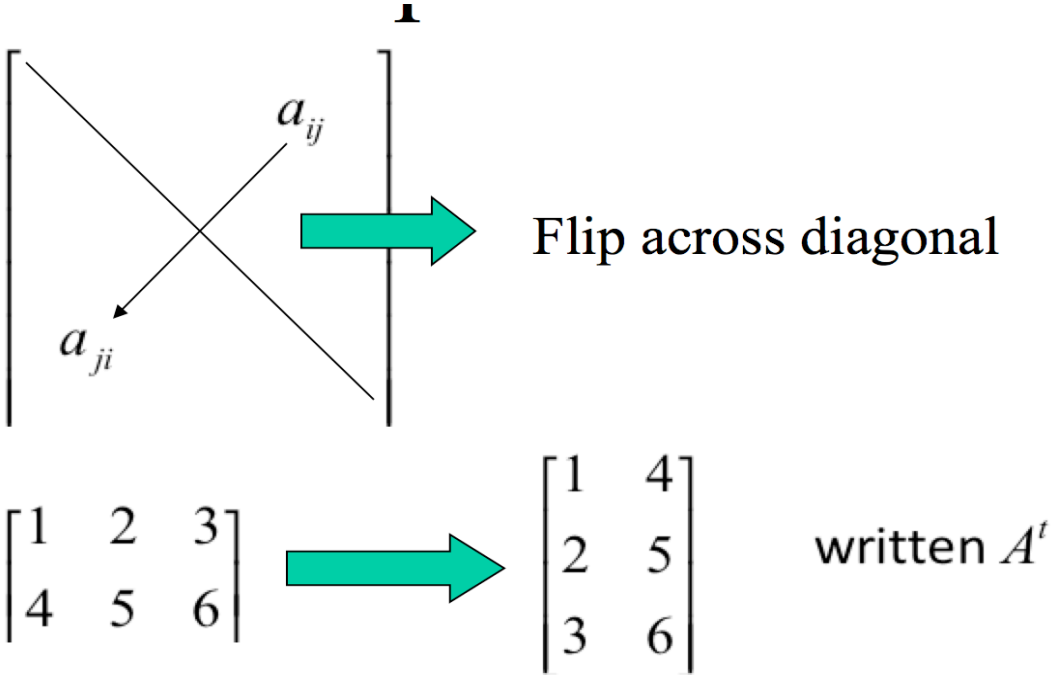
$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} a^{-1}_1 & a^{-1}_2 & a^{-1}_3 \\ b^{-1}_1 & b^{-1}_2 & b^{-1}_3 \\ c^{-1}_1 & c^{-1}_2 & c^{-1}_3 \end{bmatrix} \begin{bmatrix} p \\ q \\ r \end{bmatrix}$$

Please note that a^{-1}_j is NOT necessarily $(a_j)^{-1}$.

Ref:

http://www.cse.msu.edu/~pramanik/teaching/courses/cse260/11s/lectures/matrix/Matrix.ppt

Serial Matrix Transpose



Ref: http://www.cse.msu.edu/~pramanik/teaching/courses/cse260/11s/lectures/matrix/Matrix.ppt

Calculating Principal & Interest

- Members: The COMP605 Bank has M members.
  - Each Member has a 'user name' based on his/her ID
  - The Member ID is equal to the members row in the member accounts table
- Accounts: There are N account types:
  - checking; college (savings); vacation (savings); CD; IRA (investment).
  - Not all members have money in all accounts (sparse data).
  - The accounts are interest bearing:
    - Each account has its own interest rate: r_n
    - Assume all rates are positive
    - Monthly Interest Rate accrued is: $R\,[r_1, r_2, ..., r_n]$
- Account Matrix A stores all the members' (rows) account balances (cols)
- Each $Member(i)$ has a total account balance equal to the sum of the interest bearing account balances:

$$Acct_{total}(i) = \sum_{j=1}^{n} a_{ij} = a_{i1} + \cdots + a_{ij} + \cdots + a_{in}$$

Matrix-Vector Mult: Principal & Interest

- At the end of the month, each member earns interest based on the balance in each account.
- The total interest earned in each members account is computed as:
  $B_{int} = A * R$, where

$$B_{int} = \begin{bmatrix} b_1 \\ b_2 \\ \dots \\ b_i \\ \dots \\ b_m \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1j} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2j} & \dots & a_{2n} \\ & & \dots & & & \\ a_{i1} & a_{i2} & \dots & a_{ij} & \dots & a_{in} \\ & & \dots & & & \\ a_{m1} & a_{m2} & \dots & a_{mj} & \dots & a_{mn} \end{bmatrix} * \begin{bmatrix} r_1 \\ r_2 \\ \dots \\ r_i \\ \dots \\ r_n \end{bmatrix}$$

- Where
  $b_i = \sum_{j=1}^{n} a_{ji} = a_{i1} * r_1 + \cdots + a_{ij} * r_j + \cdots + a_{in} * r_n$
- The Members new total account balances will then be
  $Acct_{new} = Acct_{total} + B_{int}$

$A = (a_{ij})$ is an $m \times n$ matrix

$\mathbf{x}$ is a vector with n components

$\mathbf{y} = A\mathbf{x}$ is a vector with m components

$$y_i = a_{i0}x_0 + a_{i1}x_1 + a_{i2}x_2 + \cdots a_{i,n-1}x_{n-1}$$

i-th component of y

Dot product of the ith row of A with x.

Matrix-Vector Multiplication

$$\begin{bmatrix} a_{00} & a_{01} & \cdots & a_{0,n-1} \\ a_{10} & a_{11} & \cdots & a_{1,n-1} \\ \vdots & \vdots & & \vdots \\ a_{i0} & a_{i1} & \cdots & a_{i,n-1} \\ \vdots & \vdots & & \vdots \\ a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots a_{i,n-1}x_{n-1} \\ \vdots \\ y_{m-1} \end{bmatrix}$$

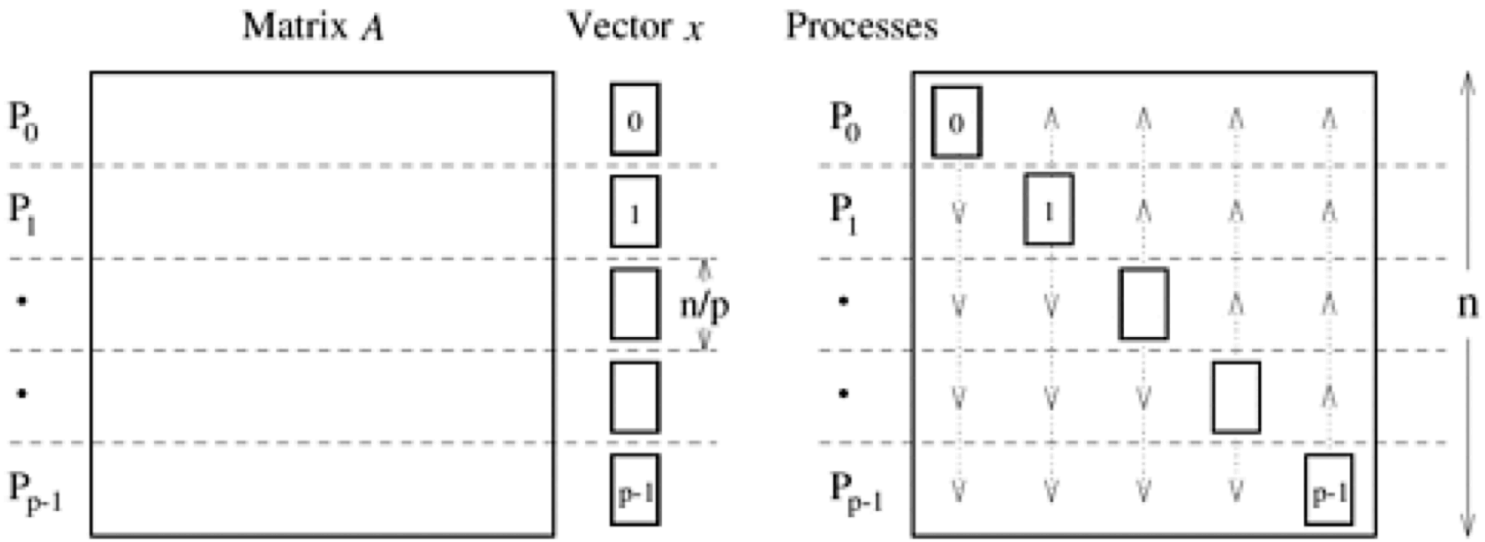
Matrix-Vector Multiplication - Serial Pseudo Code

```
/* For each row of A */
for (i = 0; i < m; i++) {
    /* Form dot product of ith row with x */
    y[i] = 0.0;
    for (j = 0; j < n; j++)
        y[i] += A[i][j]*x[j];
}
```

Serial matrix-vector multiplication is $\vartheta\left(n^2\right)$

Matrix-Vector Multiplication Partitioning: 1-D Rowwise

- The *nxn* matrix is partitioned among *p* processors; each processor stores n/p complete rows of the matrix.
- The *nx*1 vector x is distributed such that each process owns n/p of its elements.



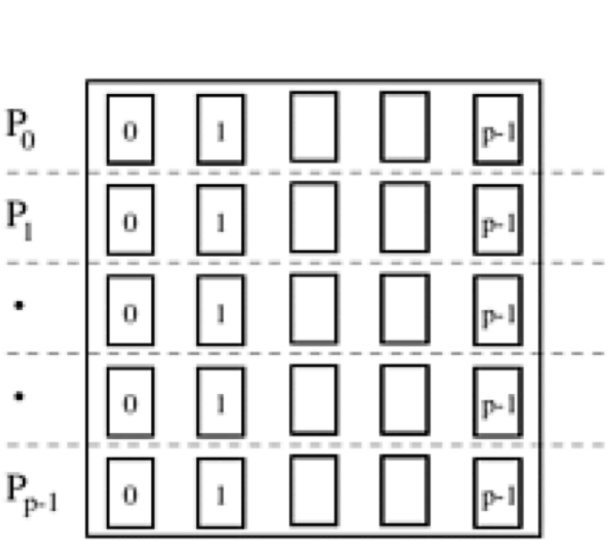
(a) Initial partitioning of the matrix and the starting vector *x*

(b) Distribution of the full vector among all the processes by all-to-all broadcast

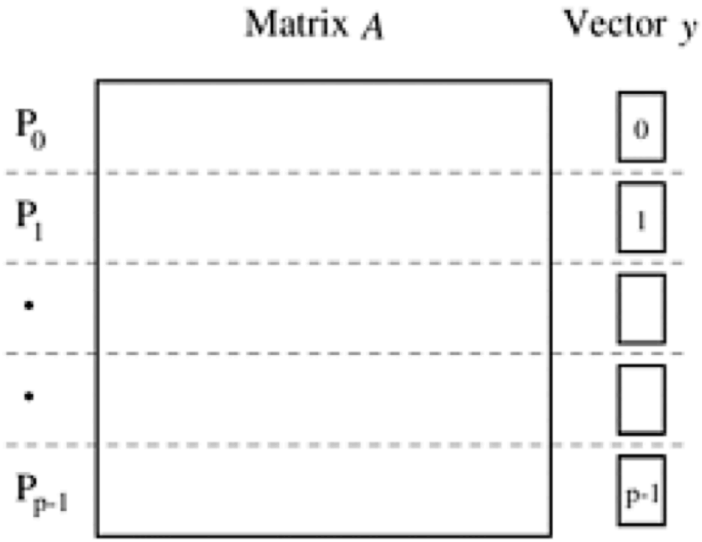
Source: Grama, et. al., Introduction to Parallel Computing, 2003

Matrix-Vector Multiplication Partitioning: 1-D Rowwise

- All-to-All broadcast of vector x is required.
- The $nx1$ vector x is distributed such that each process owns n/p of its elements.



(c) Entire vector distributed to each process after the broadcast

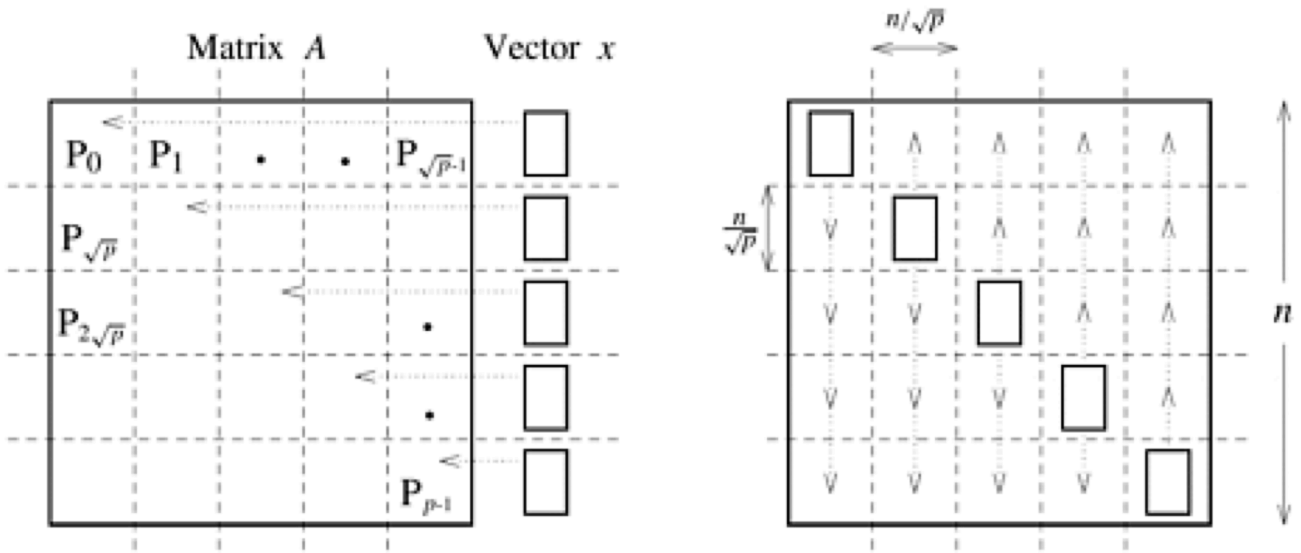


(d) Final distribution of the matrix and the result vector y

Source: Grama, et. al., Introduction to Parallel Computing, 2003

Matrix-Vector Multiplication Partitioning: 2-D Block-Block

- The *nxn* matrix is partitioned among *p* processors; each processor stores $n/\sqrt{p}$ rows x cols of the matrix.
- The *nx*1 vector x is distributed such that the processes along each row
  owns $n/\sqrt{p}$ of its elements.



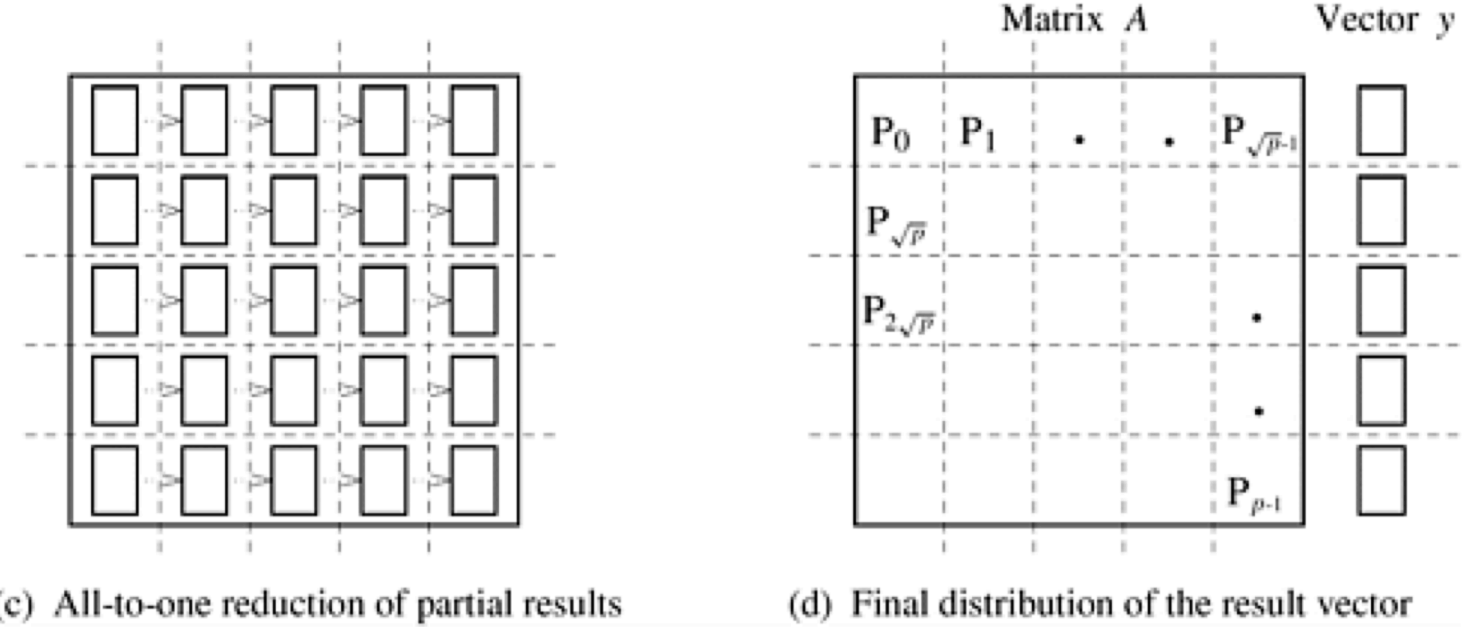
(a) Initial data distribution and communication steps to align the vector along the diagonal

(b) One-to-all broadcast of portions of the vector along process columns

Source: Grama, et. al., Introduction to Parallel Computing, 2003

Matrix-Vector Multiplication Partitioning: 2-D Block-Block

- All-to-All broadcast of vector x is required.
- The $nx1$ vector x is distributed such that each process owns n/p of its elements.



(c) All-to-one reduction of partial results (d) Final distribution of the result vector

Source: Grama, et. al., Introduction to Parallel Computing, 2003

Storage of C matrix

$$\begin{pmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \end{pmatrix}$$

stored as

0 1 2 3 4 5 6 7 8 9 10 11

Serial matrix-vector multiplication

```
void Mat_vect_mult(
      double  A[]  /* in  */,
      double  x[]  /* in  */,
      double  y[]  /* out */,
      int     m    /* in  */,
      int     n    /* in  */) {
   int i, j;

   for (i = 0; i < m; i++) {
      y[i] = 0.0;
      for (j = 0; j < n; j++)
         y[i] += A[i*n+j]*x[j];
   }
}  /* Mat_vect_mult */
```

An MPI matrix-vector multiplication function (1)

```
void Mat_vect_mult(
      double    local_A[]  /* in  */,
      double    local_x[]  /* in  */,
      double    local_y[]  /* out */,
      int       local_m    /* in  */,
      int       n          /* in  */,
      int       local_n    /* in  */,
      MPI_Comm  comm       /* in  */) {
   double* x;
   int local_i, j;
   int local_ok = 1;
```

An MPI matrix-vector multiplication function (2)

```
    x = malloc(n*sizeof(double));
    MPI_Allgather(local_x, local_n, MPI_DOUBLE,
          x, local_n, MPI_DOUBLE, comm);

    for (local_i = 0; local_i < local_m; local_i++) {
       local_y[local_i] = 0.0;
       for (j = 0; j < n; j++)
          local_y[local_i] += local_A[local_i*n+j]*x[j];
    }
    free(x);
}   /* Mat_vect_mult */
```

Pacheco MPI Matrix-Vector Multiplication (Pacheco IPP, Ch3

```
/* File:     mpi_mat_vect_mult.c
 * Purpose:  Implement parallel matrix-vector
 * one-dimensional arrays to store the vectors and the
 * matrix.  Vectors use block distributions and the
 * matrix is distributed by block rows.
  * IPP:      Section 3.4.9 (pp. 113 and ff.)
 */
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

void Check_for_error(int local_ok,char fname[],char message[],
      MPI_Comm comm);
void Get_dims(int* m_p,int* local_m_p,int* n_p,int* local_n_p,
      int my_rank,int comm_sz,MPI_Comm comm);
void Allocate_arrays(double** local_A_pp,double** local_x_pp,
      double** local_y_pp,int local_m,int n,int local_n,
      MPI_Comm comm);
void Read_matrix(char prompt[],double local_A[],int m,
     int local_m, int n,int my_rank,MPI_Comm comm);
void Read_vector(char prompt[],double local_vec[],int n,
      int local_n. int my_rank,MPI_Comm comm);
void Print_matrix(char title[],double local_A[],int m,
      int local_m, int n,int my_rank,MPI_Comm comm);
void Print_vector(char title[],double local_vec[],int n,
      int local_n,int my_rank,MPI_Comm comm);
void Mat_vect_mult(double local_A[],double local_x[],
      double local_y[],int local_m,int n,int local_n,
      MPI_Comm comm);
```

Pacheco MPI Matrix-Vector Multiplication (Pacheco IPP, Ch3

```
/*--------------------------------------------------*/
int main(void) {
   double* local_A, local_x,  local_y;
   int m,local_m,n,local_n, my_rank,comm_sz;
   MPI_Comm comm;

   MPI_Init(NULL,NULL);
   comm = MPI_COMM_WORLD;
   MPI_Comm_size(comm,&comm_sz);
   MPI_Comm_rank(comm,&my_rank);

   Get_dims(&m,&local_m,&n,&local_n,my_rank,
         comm_sz,comm);
   Allocate_arrays(&local_A,&local_x,&local_y,local_m,
             n,local_n,comm);
   Read_matrix("A",local_A,m,local_m,n,
              my_rank,comm);
#  ifdef DEBUG
   Print_matrix("A",local_A,m,local_m,n,my_rank,comm);
#  endif
   Read_vector("x",local_x,n,local_n,my_rank,comm);
#  ifdef DEBUG
   Print_vector("x",local_x,n,local_n,my_rank,comm);
#  endif

   Mat_vect_mult(local_A,local_x,local_y,local_m,n,local_n,comm);

   Print_vector("y",local_y,m,local_m,my_rank,comm);

   free(local_A); free(local_x); free(local_y);
   MPI_Finalize();
   return 0;
}  /* main */
```

```
/* PACHECO MAT-VEC-MULT (cont.) */
* Function:  Get_dims
 * Purpose:   Get the dimensions of the matrix and the vectors from stdin.
 * In args:   my_rank:   calling processes rank in comm
 *            comm_sz:   number of processes in comm
 *            comm:      communicator containing all processes calling Get_dims
 * Out args:  m_p:       global number of rows of A and components in y
 *            local_m_p: local number of rows of A and components of y
 *            n_p:       global number of cols of A and components of x
 *            local_n_p: local number of components of x
 *
 * Errors:    if either m or n isn't positive or if m or n isn't evenly  divisible by comm_sz,the program
 *            prints an error message and quits.
 * Note:
 *    All processes in comm should call Get_dims
 */
void Get_dims(
      int*      m_p        /* out */,
      int*      local_m_p  /* out */,
      int*      n_p        /* out */,
      int*      local_n_p  /* out */,
      int       my_rank    /* in  */,
      int       comm_sz    /* in  */,
      MPI_Comm  comm       /* in  */) {
      int local_ok = 1;

   if (my_rank == 0) {
      printf("Enter the number of rows\n");        scanf("%d",m_p);
      printf("Enter the number of columns\n");      scanf("%d",n_p);          }

   MPI_Bcast(m_p,1,MPI_INT,0,comm);
   MPI_Bcast(n_p,1,MPI_INT,0,comm);
   if (*m_p <= 0 || *n_p <= 0 || *m_p % comm_sz != 0
         || *n_p % comm_sz != 0) local_ok = 0;
   Check_for_error(local_ok,"Get_dims", "m and n must be positive and evenly divisible by comm_sz", comm);

   *local_m_p = *m_p/comm_sz;
   *local_n_p = *n_p/comm_sz;
}  /* Get_dims */
```

```
/* PACHECO MAT-VEC-MULT (cont.) */
/*--------------------------------------------------------------------
 * Function:  Check_for_error
 * Purpose:   Check whether any process has found an error.  If so, print message and terminate all processes.
 *                  Otherwise, continue execution.
 * In args:   local_ok:  1 if calling process has found an error, otherwise
 *            fname:     name of function calling Check_for_error
 *            message:   message to print if there's an error
 *            comm:      communicator containing processes calling
 *                       Check_for_error:  should be MPI_COMM_WORLD.
 *
 * Note:
 *    The communicator containing the processes calling Check_for_error
 *    should be MPI_COMM_WORLD.
 */
void Check_for_error(
      int       local_ok   /* in */,
      char      fname[]    /* in */,
      char      message[]  /* in */,
      MPI_Comm  comm       /* in */) {
   int ok;

   MPI_Allreduce(&local_ok,&ok,1,MPI_INT,MPI_MIN,comm);
   if (ok == 0) {
      int my_rank;
      MPI_Comm_rank(comm,&my_rank);
      if (my_rank == 0) {
         fprintf(stderr,"Proc %d > In %s,%s\n",my_rank,fname,
               message);
         fflush(stderr);
      }
      MPI_Finalize();
      exit(-1);
   }
}  /* Check_for_error */
```

```
/* PACHECO MAT-VEC-MULT (cont.) */
/*--------------------------------------------------------------------
 * Function:   Allocate_arrays
 * Purpose:    Allocate storage for local parts of A,x,and y
 * In args:    local_m:    local number of rows of A and components of y
 *             n:          global and local number of cols of A and global
 *                         number of components of x
 *             local_n:    local number of components of x
 *             comm:       communicator containing all calling processes
 * Out args:   local_A_pp: local storage for matrix (m/comm_sz rows,n cols)
 *             local_x_pp: local storage for x (n/comm_sz components)
 *             local_y_pp: local_storage for y (m/comm_sz components)
 *
 * Errors:     if a malloc fails,the program prints a message and all
 *             processes quit
 * Note:
 *    Communicator should be MPI_COMM_WORLD because of call to
 * Check_for_errors
 */
```

```
void Allocate_arrays(
      double**  local_A_pp  /* out */,
      double**  local_x_pp  /* out */,
      double**  local_y_pp  /* out */,
      int       local_m     /* in  */,
      int       n           /* in  */,
      int       local_n     /* in  */,
      MPI_Comm  comm        /* in  */) {

   int local_ok = 1;

      *local_A_pp = malloc(local_m*n*sizeof(double));
      *local_x_pp = malloc(local_n*sizeof(double));
      *local_y_pp = malloc(local_m*sizeof(double));

      if (*local_A_pp == NULL || local_x_pp == NULL ||
            local_y_pp == NULL) local_ok = 0;
      Check_for_error(local_ok,"Allocate_arrays",
            "Can't allocate local arrays",comm);
   }  /* Allocate_arrays */
```

```
/*--------------------------------------------------
 * Function:  Read_matrix
 * Purpose:   Read in the matrix and distribute among
 *           the processes using a block row distribution
 * In args:   prompt:  description of matrix (e.g.,"A")
 *            m:       global number of rows of A
 *            local_m: local number of rows of A
 *            n:       global and local number of cols of A
 *            my_rank: process rank in communicator comm
 *            comm:    communicator containing processes
 *                     calling Read_matrix
 * Out args:  local_A: the local matrix
 *
 * Errors:    if malloc of temporary storage fails on process
 *           0,the program prints a message and all processes quit
 * Note:
 * 1. Communicator should be MPI_COMM_WORLD
 *      because of call to Check_for_errors
 * 2. local_m and n should be the same on each process
 */
void Read_matrix(
      char      prompt[]   /* in  */,
      double    local_A[]  /* out */,
      int       m          /* in  */,
      int       local_m    /* in  */,
      int       n          /* in  */,
      int       my_rank    /* in  */,
      MPI_Comm  comm       /* in  */) {
   double* A = NULL;
   int local_ok = 1;
   int i,j;

   if (my_rank == 0) {
      A = malloc(m*n*sizeof(double));
      if (A == NULL) local_ok = 0;
      Check_for_error(local_ok,"Read_matrix",
            "Can't allocate temporary matrix",comm);
      printf("Enter the matrix %s\n",prompt);
      for (i = 0; i < m; i++)
         for (j = 0; j < n; j++)
            scanf("%lf",&A[i*n+j]);
      MPI_Scatter(A,local_m*n,MPI_DOUBLE,
            local_A,local_m*n,MPI_DOUBLE,0,comm);
      free(A);
   } else {
      Check_for_error(local_ok,"Read_matrix",
            "Can't allocate temporary matrix",comm);
      MPI_Scatter(A,local_m*n,MPI_DOUBLE,
            local_A,local_m*n,MPI_DOUBLE,0,comm);
   }
}  /* Read_matrix */
```

```
/*---------------------------------------------------------
 * Function:  Read_vector
 * Purpose:   Read a vector from stdin and distribute among the
 *            processes using a block distribution
 * In args:   prompt:  description of vector (e.g.,"x")
 *            n:       global order of vector
 *            local_n: local order of vector (n/comm_sz)
 *
 * Errors:    if malloc of temporary storage fails on process
 *          0, program prints a message and all processes quit
 * Notes:
 * 1. Communicator should be MPI_COMM_WORLD
 *        because of call to  Check_for_errors
 * 2. local_n should be the same on all processes
 */
void Read_vector(
      char      prompt[]     /* in  */,
      double    local_vec[]  /* out */,
      int       n            /* in  */,
      int       local_n      /* in  */,
      int       my_rank      /* in  */,
      MPI_Comm  comm         /* in  */) {
   double* vec = NULL;
   int i,local_ok = 1;

   if (my_rank == 0) {
      vec = malloc(n*sizeof(double));
      if (vec == NULL) local_ok = 0;
      Check_for_error(local_ok,"Read_vector",
            "Can't allocate temporary vector",comm);
      printf("Enter the vector %s\n",prompt);
      for (i = 0; i < n; i++)
         scanf("%lf",&vec[i]);
      MPI_Scatter(vec,local_n,MPI_DOUBLE,
            local_vec,local_n,MPI_DOUBLE,0,comm);
      free(vec);
   } else {
      Check_for_error(local_ok,"Read_vector",
            "Can't allocate temporary vector",comm);
      MPI_Scatter(vec,local_n,MPI_DOUBLE,
            local_vec,local_n,MPI_DOUBLE,0,comm);
   }
}  /* Read_vector */
```

```
/*--------------------------------------------------------------
 * Function:  Print_matrix
 * Purpose:   Print a matrix distributed by block rows to stdout
 * In args:   title:    name of matrix
 *            local_A:  calling process' part of matrix
 *            m:        global number of rows
 *            local_m:  local number of rows (m/comm_sz)
 *            n:        global (and local) number of cols
 *            my_rank:  calling process' rank in comm
 *            comm:     communicator containing all processes
 * Errors:    if malloc of local storage on process 0 fails,all
 *            processes quit.
 * Notes:
 * 1.  comm should be MPI_COMM_WORLD because of call
 *      to Check_for_errors
 * 2.  local_m should be the same on all the processes
 */
void Print_matrix(
      char      title[]    /* in */,
      double    local_A[]  /* in */,
      int       m          /* in */,
      int       local_m    /* in */,
      int       n          /* in */,
      int       my_rank    /* in */,
      MPI_Comm  comm       /* in */) {
   double* A = NULL;
   int i,j,local_ok = 1;

   if (my_rank == 0) {
      A = malloc(m*n*sizeof(double));
      if (A == NULL) local_ok = 0;
      Check_for_error(local_ok,"Print_matrix",
            "Can't allocate temporary matrix",comm);
      MPI_Gather(local_A,local_m*n,MPI_DOUBLE,
            A,local_m*n,MPI_DOUBLE,0,comm);
      printf("\nThe matrix %s\n",title);
      for (i = 0; i < m; i++) {
         for (j = 0; j < n; j++)
            printf("%f ",A[i*n+j]);
         printf("\n");
      }
      printf("\n");
      free(A);
   } else {
      Check_for_error(local_ok,"Print_matrix",
            "Can't allocate temporary matrix",comm);
      MPI_Gather(local_A,local_m*n,MPI_DOUBLE,
            A,local_m*n,MPI_DOUBLE,0,comm);
   }
}  /* Print_matrix */
```

```
/*--------------------------------------------------------------------
 * Function:  Print_vector
 * Purpose:   Print a vector with a block distribution
 * In args:   title:      name of vector
 *            local_vec:  calling process' part of vector
 *            n:          global number of components
 *            local_n:    local number of components (n/comm_sz)
 *            my_rank:    calling process' rank in comm
 *            comm:       communicator containing all processes
 * Errors:    if malloc of local storage on process 0 fails,all
 *            processes quit.
 * Notes:
 * 1.  comm should be MPI_COMM_WORLD because of call
 *             to Check_for_errors
 * 2.  local_n should be the same on all the processes
 */
void Print_vector(
      char      title[]     /* in */,
      double    local_vec[] /* in */,
      int       n           /* in */,
      int       local_n     /* in */,
      int       my_rank     /* in */,
      MPI_Comm  comm        /* in */) {
   double* vec = NULL;
   int i,local_ok = 1;

   if (my_rank == 0) {
      vec = malloc(n*sizeof(double));
      if (vec == NULL) local_ok = 0;
      Check_for_error(local_ok,"Print_vector",
            "Can't allocate temporary vector",comm);
      MPI_Gather(local_vec,local_n,MPI_DOUBLE,
            vec,local_n,MPI_DOUBLE,0,comm);
      printf("\nThe vector %s\n",title);
      for (i = 0; i < n; i++)
         printf("%f ",vec[i]);
      printf("\n");
      free(vec);
   }  else {
      Check_for_error(local_ok,"Print_vector",
            "Can't allocate temporary vector",comm);
      MPI_Gather(local_vec,local_n,MPI_DOUBLE,
            vec,local_n,MPI_DOUBLE,0,comm);
   }
}  /* Print_vector */
```

```
/*-------------------------------------------------------
 * Function:  Mat_vect_mult
 * Purpose:   Multiply a matrix A by a vector x.  The matrix
 *                  is distributed by block rows and the
 *                vectors are  distributed by blocks
 * In args:   local_A:  calling process' rows of matrix A
 *     local_x:  calling process' components of vector x
 *     local_m:  calling process' number of rows
 *     n:        global (and local) number of columns
 *     local_n:  calling process' number of components of x
 *     comm:     communicator containing all
 *                           calling processes
 * Errors:    if malloc of local storage on any process fails,
 *            all processes quit.
 * Notes:
 * 1.  comm should be MPI_COMM_WORLD because of call
 *         to Check_for_errors
 * 2.  local_m and local_n should be the same on all
 *         the processes
 */
```

```
void Mat_vect_mult(
      double    local_A[]  /* in  */,
      double    local_x[]  /* in  */,
      double    local_y[]  /* out */,
      int       local_m    /* in  */,
      int       n          /* in  */,
      int       local_n    /* in  */,
      MPI_Comm  comm       /* in  */) {
   double* x;
   int local_i,j;
   int local_ok = 1;

   x = malloc(n*sizeof(double));
   if (x == NULL) local_ok = 0;
   Check_for_error(local_ok,"Mat_vect_mult",
         "Can't allocate temporary vector",comm);
   MPI_Allgather(local_x,local_n,MPI_DOUBLE,
         x,local_n,MPI_DOUBLE,comm);

   for (local_i = 0; local_i < local_m; local_i++) {
      local_y[local_i] = 0.0;
      for (j = 0; j < n; j++)
         local_y[local_i] += local_A[local_i*n+j]*x[j];
   }
   free(x);
}  /* Mat_vect_mult */
```