UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Parallel Computing (CPA)

Lab 4: Matrix Computations with MPI

Year 2016/17

Contents

Introduction

This last laboratory exercise focuses on the parallel implementation using MPI of a more advanced numeric problem. The problem selected is solving a linear system of equations. In this laboratory exercise, the students will have to deal with a more complex problem with more opportunities to improve.

The student will find in the proper PoliformaT folder a file namely `sistbf.c`, which implements the parallel version of such problem using a block-row distribution. The objective of this exercise is to base on it to implement a version using a row-cyclic distribution.

This exercise will go through 2 sessions:

- Session 1: Study of the problem and implementation of the row-cyclic parallel version.
- Session 2: Improvement of the cyclic parallel version and experimental measure of its performance.

1 Solving a linear system of equations

A linear system of equations can be described using a matrix form as $Ax = b$, where A is the coefficient matrix and, b is the "right-hand vector" (the vector with the independent terms), and x is the solution vector. We will assume that A is square, being n its dimension:

$$A = \begin{bmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,n-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,n-1} \\ \vdots & \vdots & & \vdots \\ a_{n-1,0} & a_{n-1,1} & \cdots & a_{n-1,n-1} \end{bmatrix}, \quad x = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{bmatrix}, \quad b = \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_{n-1} \end{bmatrix}. \tag{1}$$

Therefore, the first equation will be:

$$a_{0,0}x_0 + a_{1,0}x_1 + \cdots + a_{n-1,0}x_{n-1} = b_0. \tag{2}$$

LU Factorization

```
U = A;
For k = 0, ..., n-2
  if ukk = 0 then exit
  lkk = 1
  For i = k+1, ..., n-1
    m = uik/ukk
    lik = m
    uik = 0
    For j = k+1, ..., n-1
      uij = uij - m*ukj
    End for
  End for
End for
lnn = 1
```

Forward elimination

```
% Ly = b
y = b
For i = 0, 1, ..., n-1
  For j = i+1, ..., n-1
    yj = yj - lji*yi
  End for
End for
```

Backward substitution

```
% Ux = b
x = b
For i = n-1, ..., 0
  xi = xi/uii
  For j = i-1, ..., 0
    xj = xj - uji*xi
  End for
End for
```

Figure 1: Algorithms for LU factorization and solution of triangular systems.

The objective is to compute the vector x that fulfils the n equations simultaneously. The system has a single solution (it is compatible and determined) when matrix A has a determinant different from 0. There are several direct and iterative approaches to solve system of equations. We select for this case a direct method based on the LU factorization.

The LU factorization consists in obtaining a pair of matrices, one of them unit lower-triangular matrix (all elements above the main diagonal are zero and the diagonal elements are one) and another one upper triangular (all elements below the main diagonal are zero), both of them of the same dimension as A, such that their product equals A.

Then, solving the linear system of equations is equivalent to solving two triangular systems:

$$\left.\begin{array}{l} A = LU \\ Ax = b \end{array}\right\} \quad \longrightarrow \quad LUx = b \quad \longrightarrow \quad \left\{\begin{array}{l} Ly = b \\ Ux = y \end{array}\right. . \tag{3}$$

The computation of the LU factorization can be performed by means of Gaussian elimination, using the diagonal elements of the matrix as pivots and making zeros below the diagonal in each column.

The triangular systems can be solved using forward elimination (for the lower-triangular matrix) and backward substitution (for the upper triangular one).

1.1 Sequential Algorithm

The algorithms needed to solve the problem are shown in figure 1. Each of the algorithms addresses each one of the stages in the process: LU factorization, solving the lower-triangular system and solving the upper-triangular system.

The LU algorithm performs $n-1$ iterations of the k loop. Each iteration zeros out entries below the diagonal in the kth column. Such column has $n-k-1$ elements that must be annihilated (i loop). Each iteration of loop i modifies row i completely, but since the first k elements are zero already, the j loop only traverses the elements starting from $k+1$. The update of row i involves row k (called pivot row) and uses a factor m called multiplier that is equal to the element that is being annihilated (a_{ik}) divided by the diagonal element of the pivot row (a_{kk}), also called pivot element.

To ease the checking of the algorithms, as well as the analysis of the performance, the student will use a set of special matrices called Toeplitz, characterized by having constant diagonals, that is, all entries in the main diagonal are equal, as well as in the case of other sub- and super-diagonals. A Toeplitz matrix is defined from two vectors (f and c), which contain the values from the first row and column (obviously, the first element must be the same). The diagonal is scaled to ensure that a positive-definite matrix is obtained. The algorithm shown in figure 2 generates a Toeplitz matrix from f and c, and also returns a vector b such

$$\begin{bmatrix} c_0 & f_1 & f_2 & f_3 & f_4 \\ c_1 & c_0 & f_1 & f_2 & f_3 \\ c_2 & c_1 & c_0 & f_1 & f_2 \\ c_3 & c_2 & c_1 & c_0 & f_1 \\ c_4 & c_3 & c_2 & c_1 & c_0 \end{bmatrix}$$

```
function [A, b] = createoep(c,f)
  n = length(c);
  for i=0:n-1
      A(i,i) = c(0);
      b(i) = c(0);
      for j=0:i-1
          A(i,j) = c(i-j);
          A(j,i) = f(i-j);
          b(i) = b(i) + A(i,j);
          b(j) = b(j) + A(j,i);
      end
  end
```

Figure 2: A sample Toeplitz matrix and the algorithm to obtain it from row (`f`) and column (`c`) vectors.

that the solution of the system $Ax = b$ is $x = [1, 1, \ldots, 1]^T$. In this way, it will be very easy to check that the algorithm is working correctly.

The algorithm for the LU factorization is normally implemented in a way that the output factors L and U overwrite the A matrix. At the end of the process, the lower triangle of matrix A contains L (except the diagonal of ones), and the upper triangular part contains U (including the diagonal). In the same way, the next algorithms for solving the triangular systems overwrite the vector with the independent terms with the solution of the system. Figure 3 shows those algorithms.

2 Parallel implementation

The parallel implementation for solving a linear system of equations will concurrently execute several independent operations.

The most evident parallelism is row-wise, considering that the update of each row can be done independently (i loop). In this way, the A matrix can be distributed by rows and the update can be made in parallel. This will require that the process holding the pivot row (k) will have to send it to all the processes involved before starting the i loop. Figure 4 shows and scheme of how all the i iterations can be entirely executed in parallel.

In order to solve the triangular systems of equations, vector b gets replicated in all processes and each of its elements is being updated in parallel by using the new solutions computed. The row-wise implementation enables using different data distribution that reach different performance. In the scope of this laboratory exercise, two approaches will be studied:

a) Distribution based on blocks of consecutive rows.

b) Row-cyclic distribution.

LU Decomposition

```
For k = 0, 1, ..., n-2
  if akk = 0 then exit
  For i = k+1, ..., n-1
    aik = aik/akk
    For j = k+1, ..., n-1
      aij = aij - aik*akj
    End for
  End for
End for
```

Forward Elimination

```
For i = 0, 1, ..., n-1
  For j = i+1, ..., n-1
    bj = bj - lji*bi
  End for
End for
```

Backward Substitution

```
For i = n-1, ..., 0
  bi = bi/uii
  For j = i-1, ..., 0
    bj = bj - uji*bi
  End for
End for
```

Figure 3: Algorithms for the LU decomposition and triangular solves with overwriting.

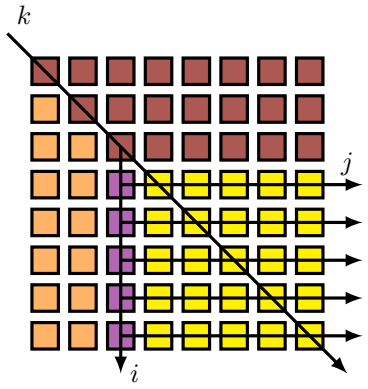

Figure 4: Scheme for the LU factorization showing the range of the three loops.

In each iteration of k, the process holding the pivot row will send it to the rest of the processes involved, which will receive it accordingly. After this, all the processes that still have rows to process will update the corresponding part.

After the factorization, triangular systems must be solved. Similarly, in each iteration there will be one process holding the row that contains the x_i element to solve. This process will compute the solution and send it to the rest of the processes that still have pending rows. In the end, all processes will contain the whole vector of unknowns.

At the end of the process, the algorithm should ensure that process P_0 has the final solution of the system. Considering the different distributions, the following factors must be taken into account:

- *Blocks of rows.* A balanced distribution considering the number of rows among the different processes and functional for any number of rows and processes. Each process, including P_0 will work on a block of consecutive rows.
- *Cyclic-row distribution.* A balanced distribution considering the number of rows among the different processes and functional for any number of rows and processes. Each process, including P_0 will work on a block of non-consecutive rows with a fixed stride.

3 Task to be implemented

The parallel version provided uses a block of rows approach, so only the cyclic distribution should be implemented. Once the cyclic version is implemented, experimental results from both approaches must be obtained. The results will compare the performance for different number of processes and different problem sizes (n). The student should justify the reasons for the different results obtained.

Appendix: Code of a block-row version

The main functions of the block-row parallel version are described in this section. This program implements the allocation and generation of matrix A and vector b, the scattering of the data following a row-wise block distribution, the LU factorization, the solution of the lower-triangular system and the solution of the upper-triangular one, the last three in parallel. The program also provide functions to allocate, free and show matrices and vectors on the screen.

The code, along with the main function, has the following functions implemented:

- `double **ppdGenMat(int nN, int nM)`. Function that given the dimensions of a matrix allocates a dynamic matrix following the descriptions above.

- `int nRellenaMat(double **ppdMat, int nN, int nM)`. Function that given an already allocated matrix and its dimensions, fills-it up with the values of a Toeplitz matrix.
- `double *pdGenTI(int nN)`. Function that allocates and computes an independent term vector such as the solution of the system with a Toeplitz matrix is always 1.
- `int nPrintMat(double **ppdMat, int nN, int nM)`. Function that prints in a tabular form the matrix provided.
- `int nSustReg(double **ppdU, double *pdB, int nN, int nId, int nP)`. Function that solves an upper triangular system of equations using backward substitution. Vector `pdB` is overwritten with the solution of the system. The function expects two additional arguments: `nId` with the rank of the process and `nP` with the number of processes.
- `int nLiberaMat(double **ppdMat)`. Función que libera la memoria reservada para una matriz dinámica.
- `int nPrintVec(double *ppdVec, int nN)`. Function that prints in a tabular form the vector provided.
- `int nElimProg(double **ppdL, double *pdB, int nN, int nId, int nP)`. Function that solves a lower triangular system of equations with diagonal 1, using the forward elimination. Vector `pdB` is overwritten with the solution of the system. The function expects two additional arguments: `nId` with the rank of the process and `nP` with the number of processes.
- `int nLU(double **ppdMat, int nN, int nId, int nP)`. Function that implements the LU factorization for a nonsingular matrix. L and U overwrite the lower and upper triangles of matrix A in the form explained above. The function expects two additional arguments: `nId` with the rank of the process and `nP` with the number of processes.

```
/* LU Factorization                                          */
/* Performs an LU decomposition on matrix A                  */
/* Leaves L on the lower triangle of A and U in the upper one */
/* Input: Matrix A and dimension                             */
/* Output: Matrix L / U
int nLU(double **ppdMat, int nN, int nId, int nP, int nBloque) {

  int i,j,k;
  double *pdPivote;

  if (ppdMat == NULL)
    return -2;

  if ( (nN<=0) || (nN>MAX_DIM) )
    return -3;

  pdPivote = (double *)malloc(sizeof(double)*nN);

  for (k=0;k<nN-1;k++) {
    if (k/nBloque == nId) {
      if (fabs(ppdMat[k%nBloque][k])<EPSILON) {
        return -1;
      }
      memcpy(pdPivote, ppdMat[k%nBloque], nN*sizeof(double));
    }

    MPI_Bcast(pdPivote, nN, MPI_DOUBLE, k/nBloque, MPI_COMM_WORLD);

    for (i=k+1;i<nN;i++) {
      if (i/nBloque == nId) {
        ppdMat[i%nBloque][k] = ppdMat[i%nBloque][k] / pdPivote[k];
        for (j=k+1;j<nN;j++) {
```

```
          ppdMat[i%nBloque][j] -= ppdMat[i%nBloque][k] * pdPivote[j];
        }
      }
    }
  }
  free (pdPivote);
  return 0;
}

/* Forward Elimination                                  */
/* Solves the system Lx = b                             */
/* Input: Matrix L, Vector B and Dimension              */
/* Output: Result Vector                                */
/* Note: Assumes that memory is already allocated       */
/* Note: Returns the solucion in vector B               */
int nElimProg(double **ppdL, double *pdB, int nN, int nId, int nP, int nBloque) {
  int i, j;

  if (ppdL == NULL)
    return -2;

  if ( (nN<=0) || (nN>MAX_DIM) )
    return -3;

  for(i=0;i<nN;i++) {
    MPI_Bcast(&pdB[i], 1, MPI_DOUBLE, i/nBloque, MPI_COMM_WORLD);
    for (j=i+1;j<nN;j++) {
      if (j/nBloque == nId) {
        pdB[j] -= ppdL[j%nBloque][i] * pdB[i];
      }
    }
  }
  return 0;
}

/* Backward Substitution                                */
/* Solves the system Ux = b                             */
/* Input: Matrix U, Vector B and Dimension              */
/* Output: Result Vector                                */
/* Note: Assumes that memory is already allocated       */
/* Note: Returns the solucion in vector B               */
int nSustReg(double **ppdU, double *pdB, int nN, int nId, int nP, int nBloque) {
  int i, j;

  if (ppdU == NULL)
    return -2;

  if ( (nN<=0) || (nN>MAX_DIM) )
    return -3;

  for(i=nN-1;i>=0;i--) {
    if (i/nBloque == nId) {
      if (fabs(ppdU[i%nBloque][i])<EPSILON) return -1;
      pdB[i] = pdB[i] / ppdU[i%nBloque][i];
    }
    MPI_Bcast(&pdB[i], 1, MPI_DOUBLE, i/nBloque, MPI_COMM_WORLD);
    for (j=i-1;j>=0;j--) {
      if (j/nBloque == nId) {
        pdB[j] -= ppdU[j%nBloque][i] * pdB[i];
      }
    }
  }
  return 0;
}
```