Serial to Parallel Matrix Inverse Algorithm using Gauss – Jordan Method

Ramón O. Fabery
Electrical Engineering
Luis Vicente, Ph.D.
Electrical and Computer Engineering & Computer Science Department
Polytechnic University of Puerto Rico

***Abstract* — In the past decade computers have become faster and more efficient, utilizing the technology available to put more transistors inside the same space; but when they can't shrink the transistor any more is time to put more processors together to reduce the time of execution and have more than one processor making different instructions at the same time. This increment in throughput is possible using Message Passing Interface (MPI) a portable message-passing communication protocol that allows breaking apart the code and thus perform many instructions assigning to each processor the specific instruction to be executed making the process much faster. We are using a serial C programming language code program that calculates the inverse of a matrix using the Gauss-Jordan Method; then, we parallelize the same program using MPI.*

***Key Terms* — Matrix Inverse, Message Passing Interface, Multiples Cores, Parallel Computing.*

INTRODUCTION

Message Passing Interface is a standardized and portable message-passing system. The advantages of this protocol is that it can provide the programmer a collection of functions for the design and implementation, without necessarily having to know the particular hardware on which is going to be executed, or the way in which they have implemented the used functions. This is possible because MPI works between the application and the software layer as we see in the Figure 1. MPI has been developed by the MPI Forum, a group of researchers from universities, laboratories and companies involved in High Performance Computing (HPC). The fundamental objectives of MPI Forum are:

- Define a single programming environment that ensures the portability of parallel applications.
- Fully define the programing interface, without specifying how I going to be implemented.
- Offer quality implementations, of public domain, to favor the extension of the standard.
- Convince the parallel computer manufacturers to offer optimize MPI versions for their machines (which have already manufacturers such as IM and Silicon Graphics)[1].

The Message-passing systems are used especially on distributed machines with separate memory for executing parallel applications as shown in Figure 3. With this system, each executing process will communicate and share its data with others by sending and receiving messages using different commands.

MPI FUNCTIONS

To use the MPI features in the code the #include "mpi.h" library must be included. The statement needed in every program before any other MPI code is MPI_Init(&argc, &argv); and the last statement of MPI code must be MPI_Finalize; The program will not terminate without this last statement.

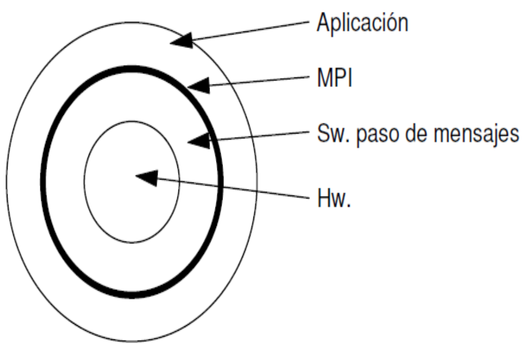
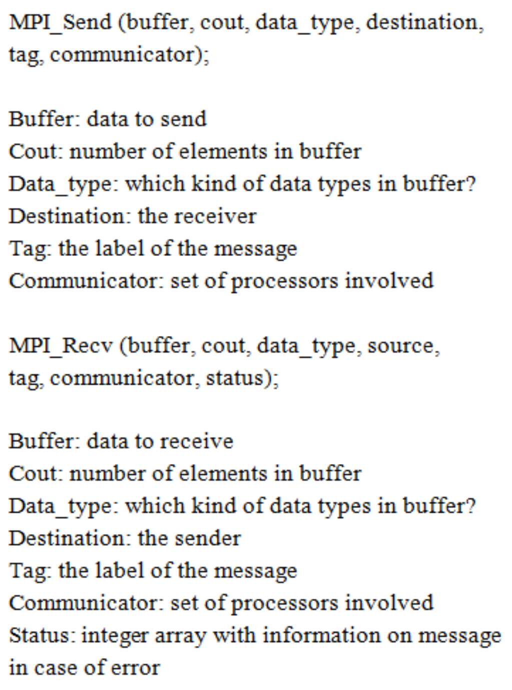


Figure 1
Location of MPI in the Programming of Parallel Applications.

Sending messages is straightforward. The source (the identity of the sender) is determined implicitly, but the rest of the message (envelope and body) is given explicitly by the sending process. To receive a message, a process specifies a message envelope that MPI compares to the envelopes of pending messages. If there is a match, a message is received. Otherwise, the receive operation cannot be completed until a matching message is sent. The MPI send and receive functions are discuss in the Figure 2.



MPI_Send (buffer, cout, data_type, destination, tag, communicator);

Buffer: data to send
Cout: number of elements in buffer
Data_type: which kind of data types in buffer?
Destination: the receiver
Tag: the label of the message
Communicator: set of processors involved

MPI_Recv (buffer, cout, data_type, source, tag, communicator, status);

Buffer: data to receive
Cout: number of elements in buffer
Data_type: which kind of data types in buffer?
Destination: the sender
Tag: the label of the message
Communicator: set of processors involved
Status: integer array with information on message in case of error

Figure 2
MPI Send and Receive Functions

These are the basic point-to-point communication routines in MPI [2]. Table 1 shows the MPI data types used in the send receive functions:

Table 1
Basic MPI Data Types – C Programming

MPI Datatype	Type
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short
MPI_UNSIGNED	int
MPI_UNSIGNED_LONG	unsigned int
MPI_FLOAT	unsigned long
MPI_DOUBLE	int
MPI_LONG_DOUBLE	float
MPI_BYTE	double
MPI_PACKED	long double

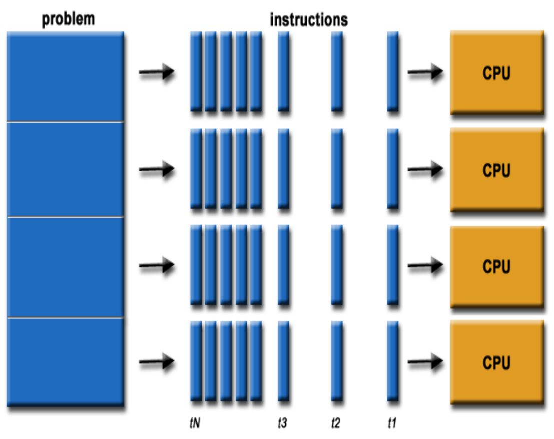
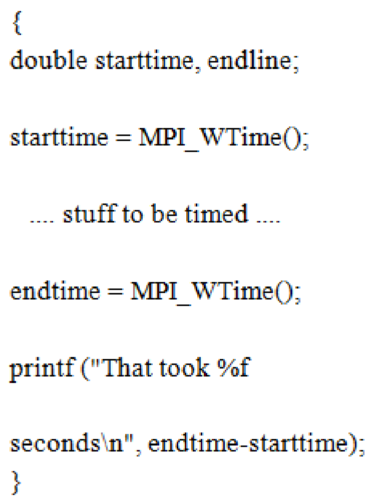


Figure 3
Parallel Computing Example

MPI provide a function very practical to have quantitative information using a timer. A timer is specified even though it is not “message-passing,” because timing parallel programs is important in “performance debugging” and that function is MPI_WTIME (). MPI_WTIME returns a floating-point number of seconds, representing elapsed wall-clock time since some time in the past. This function is portable (it returns seconds, not "ticks"), it allows high-resolution, and carries no unnecessary baggage [3]. It can be implemented like we see in Figure 4.

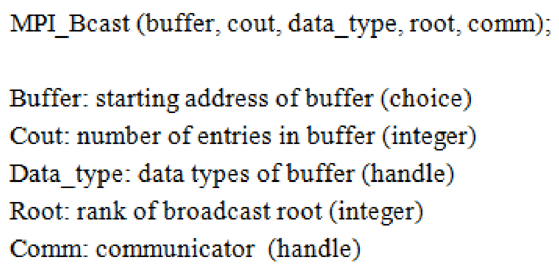


```
{
double starttime, endline;

starttime = MPI_WTime();

   .... stuff to be timed ....

endtime = MPI_WTime();

printf ("That took %f

seconds\n", endtime-starttime);
}
```

Figure 4
Implementation of MPI_WTime() Function

All MPI communication is based on a Communicator which contains a context. A context allows different libraries to co-exist, define a safe communication space for message-passing and can be viewed as system-managed tags. The group is just a set of processes that are always referred to by unique rank in group. The MPI_COMM_WORLD contains all processes available at the time the program was started and provides initial safe communication space. Inside the MPI area a processor could determine its rank in a communicator by using a call to MPI_COMM_RANK. Also, the processor can determine the size, or number of processors, of any communicator to which it belongs with a call to MPI_COMM_SIZE.

The MPI_BCAST function broadcasts a message from the process with rank root to all processes of the group, itself included. It is called by all members of group using the same arguments for comm, root. On return, the content of root's communication buffer has been copied to all processes. The type signature of count, datatype on any process must be equal to the type signature of count, datatype at the root. This implies that the amount of data sent must be equal to the amount received, pairwise between each process and the root. MPI_BCAST and all other data-movement collective routines make this restriction. Distinct type maps between sender and receiver are still allowed [4]. This function is described in Figure 5.



```
MPI_Bcast (buffer, cout, data_type, root, comm);


Buffer: starting address of buffer (choice)
Cout: number of entries in buffer (integer)
Data_type: data types of buffer (handle)
Root: rank of broadcast root (integer)
Comm: communicator  (handle)
```

Figure 5
Implementation of MPI_Bcast Function

There are more functions in the MPI protocol that allows the programmer to exploit all the available resources. This method of message passing is practical but brings some complexity to the serial code because we need to understand and know all the options that offer the MPI. Another added problem is the way we are going to divide the code, so each processor will perform the task at the same time as the other processes (as it is shown in Figure 3) depending how much one process need the output of other process. The final objective to optimize the algorithm is to get all the processors working the most of the execution time.

The exercise introduced in this work is to calculate the matrix inverse of a square matrix using the Gauss–Jordan elimination. This is a known algorithm that is performed by augmenting the square matrix $[A]$ with the identity matrix of the same dimensions $[C]$(1) and applying the following matrix operations:

$$[C] = [A|I] \qquad (1)$$

Explicit representation of the workspace is(2):

$$[C] = [A|I] = \begin{bmatrix} a_{11}\ a_{12}\ a_{13}\ a_{1n}\ 1\ 0\ 0\ 0 \\ a_{21}\ a_{22}\ a_{23}\ a_{2n}\ 0\ 1\ 0\ 0 \\ a_{31}\ a_{32}\ a_{33}\ a_{3n}\ 0\ 0\ 1\ 0 \\ a_{41}\ a_{42}\ a_{43}\ a_{nn} 0\ 0\ 0\ 1 \end{bmatrix} \qquad (2)$$

Elementary row operations are used then to reduce the left half of [C](3) occupied by [A] into the identity matrix. Each iteration (*i*) for this step aims to reduce the a_{ii}element to 1

$$[C] = \begin{bmatrix} \ddots & \cdots & \cdots & \cdots & \cdots & ⋰ \\ \vdots & 1 & a_{i-1,i}^{i-1} & a_{i-1,i+1}^{i-1} & a_{i-1,i+2}^{i-1} & \vdots \\ \vdots & 0 & 1 & \frac{a_{i,i+1}^{i-1}}{a_{i,i}^{i-1}} & \frac{a_{i,i+2}^{i-1}}{a_{i,i}^{i-1}} & \vdots \\ \vdots & 0 & a_{i+1,i}^{i-1} & a_{i+1,i+1}^{i-1} & a_{i+1,i+2}^{i-1} & \vdots \\ ⋰ & \cdots & \cdots & \cdots & \cdots & \ddots \end{bmatrix} \qquad (3)$$

where a(*k*) is the result of the iteration *k*. Further operation is to zero all a_{ji} coefficients except a_{ii} by replacing row j with a properly chosen linear combination between row i and row j(4).

$$[C] = \begin{bmatrix} \ddots & \cdots & \cdots & \cdots & \cdots & ⋰ \\ \vdots & 1 & 0 & a_{i-1,i+1}^{(i)} & a_{i-1,i+2}^{i-1} & \vdots \\ \vdots & 0 & 1 & a_{i,i+1}^{(i)} & \frac{a_{i,i+2}^{i-1}}{a_{i,i}^{i-1}} & \vdots \\ \vdots & 0 & 0 & a_{i+1,i+1}^{i-1} - \frac{a_{i,i+1}^{(i)}}{a_{i+1,i}^{i-1}} & a_{i+1,i+2}^{i-1} - \frac{a_{i,i+2}^{(i)}}{a_{i+1,i}^{i-1}} & \vdots \\ ⋰ & \cdots & \cdots & \cdots & \cdots & \ddots \end{bmatrix} (4)$$

After all iterations are over, the right half of [C] will contain the inverse matrix or $[A]^{-1}$(5).

$$[C] = \begin{bmatrix} 1 & 0 & \cdots & 0 & a_{1,1}^{\{inv\}} & a_{1,2}^{\{inv\}} & \cdots & a_{1,n}^{\{inv\}} \\ 0 & 1 & \cdots & 0 & a_{2,1}^{\{inv\}} & a_{2,2}^{\{inv\}} & \cdots & a_{2,n}^{\{inv\}} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \cdots & \vdots \\ 0 & 0 & \cdots & 1 & a_{1,1}^{\{inv\}} & a_{1,2}^{\{inv\}} & \cdots & a_{1,n}^{\{inv\}} \end{bmatrix} \qquad (5)$$

The iterations have to be performed sequentially, i.e. iteration *k*+1 have to be performed after iteration *k* is over. However, the algorithm has some opportunities for parallel processing as we will explain next.

The first step of the iteration *i* means dividing all elements of row *i* with a_{ii} and this can be executed in parallel. However, the simple division is an operation too simple compared to the overhead implied by parallelism introduction and the gains would be rather small.

The second step of iteration *i* is performed over all rows *j* with *j* = *i* and within each row *j* we have to perform one multiplication and one addition for each column of the rows *i* and *j*. Processing one row *j* is an operation complex enough to allow parallel processing despite parallelization overhead. Moreover, particular programming techniques allow us to reduce the overhead to one equivalent fork/join operation per iteration *i* [5].

PROBLEM STATEMENT

In this project we have a serial program that calculates the matrix inverse using Gauss Jordan method and we want to optimize the time of execution by using the advantage of parallel computing. This can be achieved by the knowing and understanding of the MPI Protocol. The challenge of this project is to transfer the code to a parallel programing without losing the capabilities of the serial program and be able to reduce the time of execution.

METHODOLOGY

We start the project with a serial program (as we see in Figure 6) in C language that can calculate the matrix inverse using Gauss Jordan method.

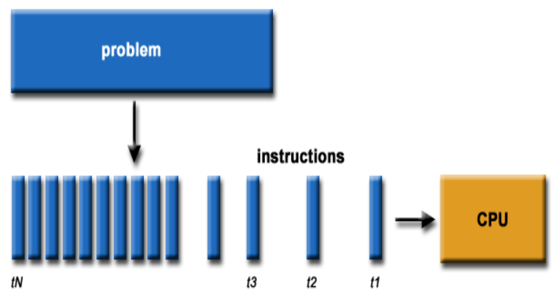


Figure 6
Serial Computing Example

For the conversion to parallel we analyze the best way to implement the code in the MPI, this parallel code is made for four processors parallel computing. This can be done in a computer with less than four processors but does not guaranteed the optimization that we are looking for. In the Figure 7 we see the task made for each processor.

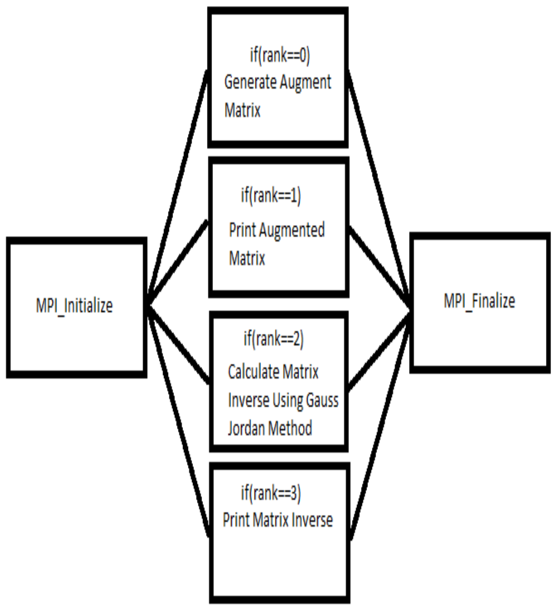

MPI_Initialize
if(rank==0) Generate Augment Matrix
if(rank==1) Print Augmented Matrix
if(rank==2) Calculate Matrix Inverse Using Gauss Jordan Method
if(rank==3) Print Matrix Inverse
MPI_Finalize

Figure 7
The Task Done by Each Processor

The first processor is appending the columns of the matrix that we want the inverse and the identity matrix to create the augmented matrix (Figure 8). After that this processor sends the order of the matrix to the other three processors and sends the augmented matrix to the second and third processor.

The second processor is receiving the order of the matrix and the augmented matrix, then the processor print the augmented matrix so that the operator see in the terminal that the matrix inverse calculation is in progress.

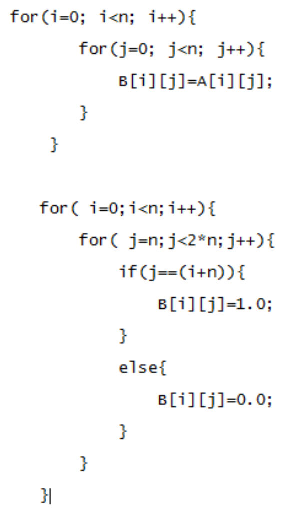

```
for(i=0; i<n; i++){
        for(j=0; j<n; j++){
             B[i][j]=A[i][j];
        }
    }

    for( i=0;i<n;i++){
        for( j=n;j<2*n;j++){
             if(j==(i+n)){
                 B[i][j]=1.0;
             }
             else{
                 B[i][j]=0.0;
             }
        }
    }
```

Figure 8
Processor 0 Task

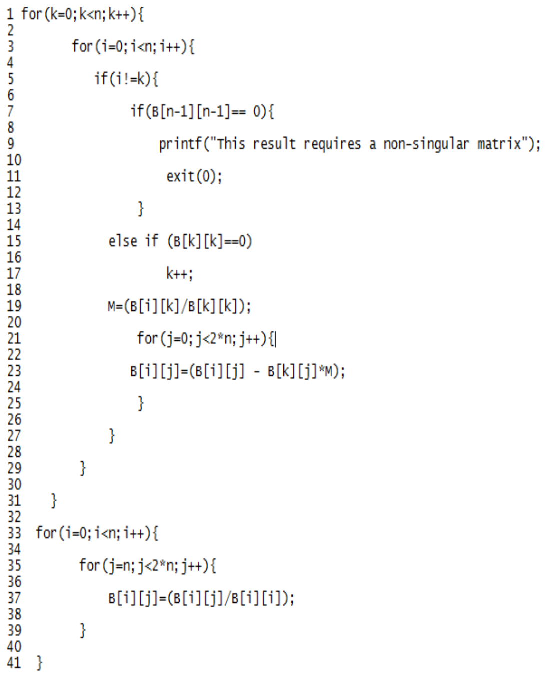

```
for(k=0;k<n;k++){
      for(i=0;i<n;i++){
         if(i!=k){
              if(B[n-1][n-1]== 0){
                   printf("This result requires a non-singular matrix");
                    exit(0);
               }
           else if (B[k][k]==0)
                    k++;
           M=(B[i][k]/B[k][k]);
                for(j=0;j<2*n;j++){
              B[i][j]=(B[i][j] - B[k][j]*M);
                }
           }
       }
  }
 for(i=0;i<n;i++){
       for(j=n;j<2*n;j++){
           B[i][j]=(B[i][j]/B[i][i]);
       }
 }
```

Figure 9
Gauss-Jordan Operation in the code

The third processor performs the calculation of the matrix inverse using Gauss-Jordam method as seen in Figure 9. Between the line 1 and 31 this for loop is checking if the pivot is not zero. If the matrix is singular doesn't have inverse and the program print out "This result requires a non-singular matrix" and the program is terminated. The next step after verifying if the matrix is non-singular the program divides by the diagonal element to have a 1 in this position. Then the program performs the operation that leave us zero in the desired position, after that we send the result to the processor number four. The fourth processor prints the result in the terminal.

LAB RESULTS

In this work we could find that the conversion from serial to parallel can be done, and the processors can communicate with each other and perform the tasks as we designed. The time of execution wasn't the expected because the program has simple arithmetic operations making the program too light and the computer runs the program to fast and the time is so little that can't be measured in 6 significant figures. We made a loop so the process can last longer but for small matrices the thread synchronization overhead is overwhelming and for our design a 5 by 5 matrix is the maximum matrix we can calculate the inverse because due to buffer size limits, we are limited in the amount of actual data that can be sent with each call to MPI_SEND.

CONCLUSION

The conversion could be done with no complications this part of the goals that we establish in the problem statement was achieved. But the part of the execution time was not accomplished because of two factors:

- For small matrices the thread synchronization overhead is overwhelming. We need to handle bigger matrixes to make the program process more information and the optimization in execution time could be significant.
- The limit in the buffer for the MPI_Send and the MPI_Receive holds the program and the large matrixes can't be done.

REFERENCES

[1] Alonso, J., M., "Programación de aplicaciones paralelas con MPI (Message Passing Interface)", Facultad de Informática UPV/EHU, May 7, 2012, http://www.sc.ehu.es/acwmialj/edumat/mpi.pdf

[2] The PACS Training Group, "Introduction to MPI", Board of Trustees of the University of Illinois, May 2, 2012, http://people.sc.fsu.edu/~jburkardt/pdf/mpi_course.pdf

[3] Ross, R, "MPITypes : MPITypes: A Library for Processing MPI Datatypes outside MPI", Argone National Laboratory, May 1, 2012, http://www.mcs.anl.gov/research/projects/mpi/mpi-standard/mpi-report-1.1/node150.htm

[4] The group of representatives that defines and maintain the MPI Standard, "4.4. Broadcast" MPI Forum, May 4, 2012, http://www.mpi-forum.org/docs/mpi-11-html/node67.html

[5] C. Vancea, "Parallel Algorithm for Computing Matrix Inverse by Gauss-Jordan Method", Department of Electrical Measurements and Usage of Electric Energy, April 23, 2012, http://electroinf.uoradea.ro/reviste%20CSCS/documente/JCSCS_2008/JCSCS_2008_22_VanceaC_1.pdf