DISTRIBUTED CALCULATOR

PROJECT REPORT

Submitted by

PRADEEP KUMAR (Roll No.: 11640680)
ANUJ SINGH (Roll No.: 11640210)

Guided by
Dr. RATAN K. GHOSH
Dr. SUBHAJIT SIDHANTA

INDIAN INSTITUTE OF TECHNOLOGY BHILAI
CS525
APRIL, 2020

Contents

Declaration

We declare that this written submission represents our ideas in our own words and where others' ideas or words have been included, we have adequately cited and referenced the original sources. We also declare that we have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source our submission. We understand that any violation of the above will be cause for disciplinary action by the Institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

Anuj Singh

11640210

Pradeep Kumar

11640680

Abstract

Our aim is to provide an easy to use Distributed Linear Algebraic computations by a complete abstraction of the end user to the distributed part of the computation. The distributed computation and MPI details remains hidden to the end user. The whole communication layer and process initialization is handled by the code write from scratch using multiprocessing and socket programming in python. Eight different distributed algorithms are implemented which are as follows: Eigenvalues, Gaussian Elimination,Inverse, Transpose, Multiplication, Eigenvectors, QR Decomposition, Determinant.

1 CHAPTER 1: INTRODUCTION

1.1 MPI Overview

1.1.1 What is MPI?

Message passing interface (MPI) is a communication protocol for parallel computers which supports both point-to-point and collective communication. It primarily addresses the message passing programming model in which data is moved from one address of a process more than equal to one process through coordinated operations on each process.

1.1.2 MPI Communicator

Communicator object connects group of processes in the MPI session which gives each process a unique identifier called rank.

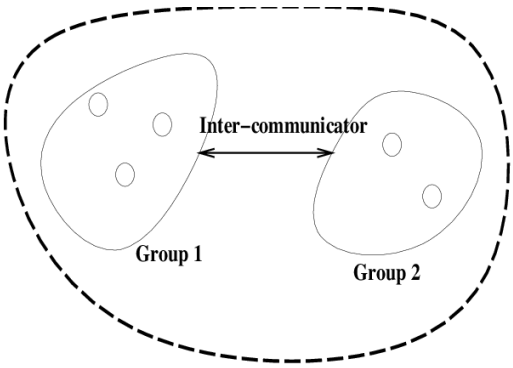

Figure 1: MPI Communication world [1]

1.1.3 Point to Point basic

MPI functions like **MPI_Send(...)** and **MPI_Recv(...)** enables communication between two different processes.

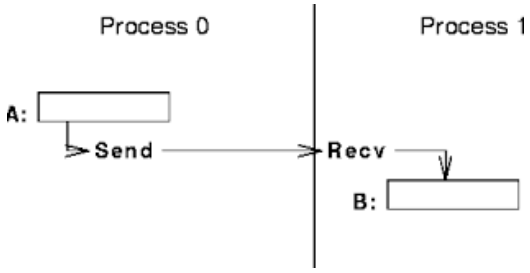

Figure 2: MPI Send Diagram [2]

1.1.4 Collective basics

Collective function employes communication among all processes in a communication group. Some MPI function are **MPI_Bcast(...)** which send the data to all the processes, **MPI_Reduce(...)** which collects data from all the processes in the group and do certain operation like summing and stores the results on one node, **MPI_Scatter(...)** and **MPI_Gather(...)**.

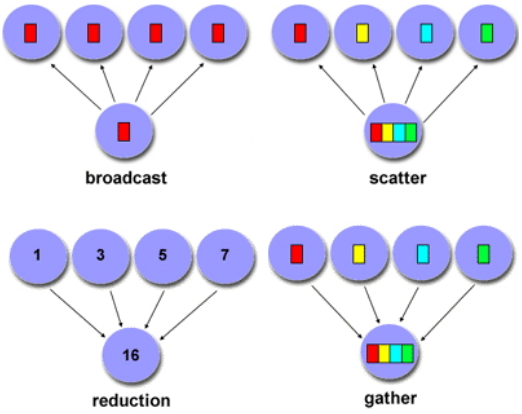

Figure 3: MPI Collective operations [3]

1.2 Socket programming

Sockets allows communication between two different processes on same or different machine. The communication between processes is machine independent. It mostly allows to create client-server environment.

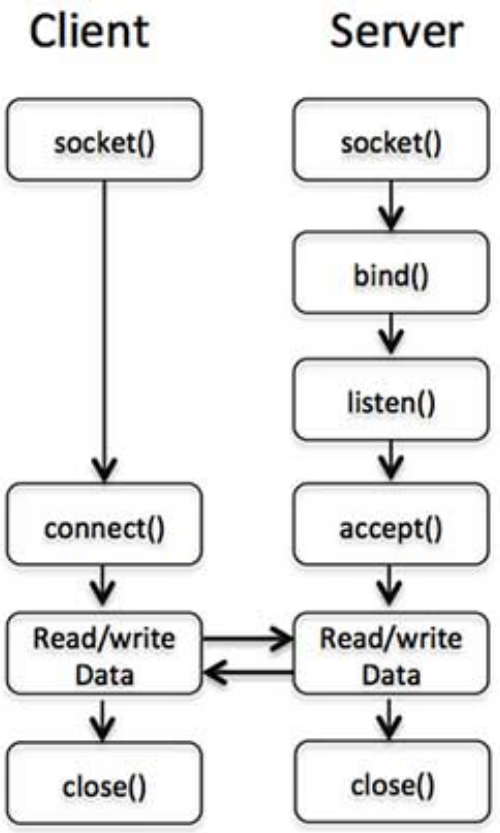

Figure 4: Server-client socket communication [4]

1.2.1 Functions used in the program

- **socket()**: This call creates an unnamed socket and returns a file descriptor to the calling process.
- **bind()**: It is a system call that binds a socket to an address. Here, the address would be the IP address of the current machine and the port number.
- **listen()**: This call allows a process to listen on socket for communication.
- **accept()**: It is a system call that causes the process to block until the client connects to the server. It returns a new descriptor and all communication should be carried out using the new file descriptor.
- **fd=open(buffer,O_RDONLY)** opens the file requested by the client in the read only mode.
- **read(fd,buffer,4096)**: This reads the contents of the file into the buffer. In socket programming, all communications happens using the buffer both at client and server side. With the completion of this read , the contents of the file is residing in the buffer and is ready to be sent to the client.
- **write(newsockfd,buffer,4096)**: This is the final command which writes the contents of the buffer(which has the file content) into the socket using the newsockfd which finally delivers it to the client process.
- **connect()** is a function used by the client to establish a connection to the server.

2 CHAPTER 2: LITERATURE SURVEY

There have been many attempts to compute matrix operations on a distributed environment. Most of them are built for expert use, where they leverage existing architecture to build state of the art libraries. Yu et al.[5] have presented scalable and efficient processing and communication schemes to optimize the communication cost. They implemented these techniques on Spark[6]. Ramamoorthy et al. [7] survey distributed matrix computation techniques using coding theory aimed to be straggler resistant. Qian et al.[8] created a library for large matrix computations on the cloud for the Alibaba group. Gu et al.[9] have created Marlin built on top of Spark for distributed matrix computations.

3 CHAPTER 3: METHODS DEVELOPED

3.1 Communication Layer

Here we discuss our process creation method, implemented in the lowest layer of our code. By the command line arguments each process receives, each one of them knows their own IP and the IP of the main process running on the users machine called the master. The processes executed in machines other than the master (slaves) send the number of virtual threads exposed by the system in each of their machines. Once the master receives this data from all processes, it distributes the total number of process across the machines to limit the number of process assigned to a machine by its virtual threads. If the total number of processes requested by the user exceeds the total number of virtual threads across all machines then the limit is bypassed.
After the number of processes to be created at every machine has been calculated, a mapping of ranks to socket addresses is computed and broadcast to all other machines. Using this mapping, the individual machines spawn the number of processes required and

assign ranks to them. The communication is handled using this rank to socket address mapping.
We now briefly discuss the various communication primitives developed for this layer:

- **Send:**
  Declaration: send(data, destination, tag), returns None
  We use a persistent blocking send, which uses the rank to socket mapping to identify the socket address of the destination. We also bind to our own address, so the receiver can identify which rank the data was received.
- **Receive:**
  Declaration: receive(destination, tag), returns data
  The receive is a blocking call which does not return until the requested data has been received. To handle multiple sends to the same rank, we implement a buffer which stores the data along with the source information. This buffer for data is checked before starting any TCP communications.
- **Broadcast:**
  Declaration: bcast(data, source, tag), returns data to the receiving Processes, None to the sender.
  The broadcast is a combination of send and receive, where the source sends data to all others.
- **Scatterv: 3**
  Declaration: scatterv(data, offset, root), returns variable chunks of the data provided in the offset to the receiving process, None to the sender.
  Scatterv send the variable chunks of data all the processes.
- **Gattherv: 3**
  Declaration: gatherv(data, offset, root), returns all merged variable chunks of data to the receiving process, None to the sender.
  Gatherv receives the merged variable chunks of data from all the other processes.
- **Reduce:**
  Declaration: reduce(data, destination, function, tag), returns data to the destination, None to others
  Reduce collects data from all processes sending the information and applies the function to reduce the result and return it to the destination rank.

For all the cost models discussed in the following subsections, assume a matrix size of NxN with p process. t_w is the communication time for a word and t_s is the time to create as single process.

3.2 Gaussian Elimination

Gaussian Elimination aims to solve a system of equations $Ax = b$ by creating a row reduced echelon form of the augmented matrix $A|b$. The single process method can be found at https://web.mit.edu/10.001/Web/Course_Notes/GaussElimPivoting.html We have implemented a distributed Gaussian Elimination method which with partial pivoting. The process is divided into three major parts:

1. **Matrix Distribution:**
   We begin by distributing the augmented matrix $A|b$ to all the processes available to us. We divide the rows of the augmented matrix by the number of processes

p that we have running. Hence every process get $\left\lfloor \frac{r}{p} \right\rfloor$ number of rows. In case if $r\%p = m$ where $m \neq 0$ we add the remaining rows among the top m ranks. The chunk calculation is done by all processes as each of them receives the number of rows r and number of columns c through an initial broadcast by rank 0 and they already have the number of process running.

2. **Reduction to row echelon form:**
   We then reduce the matrix to a row reduced echelon form using partial pivoting. We iterate through row 0 to row $r-1$. Every process is aware whether it has the current row in the iteration in its chunk as the chunk calculation was done by all processes. The process in charge divides the row by the diagonal element corresponding to the row as well as broadcasting the row to all processes which have rows below the current row. The other processes receiving the row update their rows accordingly.
   In case if the pivot element is 0, the process aims to find a row which is not zero by first going through other rows in its own chunk below the current row it swaps the rows. If it is unable to find a row with a non zero entry for the current row, it sends out a 'help' request to all other processes which have rows below the current for to send a row for swapping.
   The processes receiving the 'help' request then compute the row with the largest non zero pivot and send the pivot to the process which sent the 'help' request. The receiving process then computes the maximum of all the pivots received. To the process which sent the maximum pivot, it sends its own row for swapping, to the rest it sends the rejection message. After the swap, the rows are updated as in the earlier case.
   If however, there are no non zero pivots to be found, we see the entry of b. If the entry is non zero, there is no solution, otherwise there are infinite solutions. In either case, the algorithm sets the return value as a string indicating these possibilities and the algorithm terminates.

3. **Back Substitution:**
   Once the row reduced echelon matrix is generated, the chunks are collected and sent to process 0. The back substitution is handled by rank 0 entirely as there is no performance gain in distributing this operation.

Cost Model:

$$N^3/p + t_s p + 2t_w N(N+1)/p + t_w(N-1) + N^2 \tag{1}$$

3.3 Matrix Multiplication

Matrix multiplication aims to multiply two matrices A_{mxq} and B_{qxn} to get a result matrix C_{mxn}. The single process can be found at https://en.wikipedia.org/wiki/Matrix_multiplication. We simply distribute the naive algorithm by distributing the rows of A and broadcasting matrix B to all other processes. Hence the matrix multiplication can be broken into:

1. **Row distribution:**
   The row distribution is similar to the one discussed in Gaussian Elimination. The matrix A is divided into chunks and sent to all other processes with a near equal division of rows.

2. **Multiplication:**
   Every process carries out multiplication for its own chunk and produces the corre-

sponding rows of the result matrix. Finally the result of each chunk is sent to rank 0 and the resultant matrix is merged and returned as the output.

Cost Model:

$$N^3/p + t_s p + t_w N^2 p \tag{2}$$

3.4 Determinant

The determinant of a matrix A can be computed by computing the row reduced echelon form for matrix A. While computing the row reduced echelon form, we need to take account the following three points: (from https://en.wikipedia.org/wiki/Gaussian_elimination)

1. Swapping rows multiplies the determinant by -1. We track this by setting a variable swaps = 1 unique to every process, which is multiplied by -1 every time a swap of rows happens.
2. Multiplying a row with a non zero scalar multiplies the determinant by the same scalar. We track this by setting a variable divs unique to every process, which is a product of all the multiplicative factors by which a row is multiplied with.
3. Adding one row with a scalar multiple of another row does not change the determinant. We do not need to track anything here.

Hence when computing the row reduced echelon form by the process shown in Gaussian Elimination, we keep a track of the divisions and the swaps being done. Once the row reduction has been done, the processes send the values for divs and swaps variable for product reduction to rank 0. The sign is set using the product of swaps and the value is determined by the product of swaps. Note that during the row reduction, if any pivot element if 0 even after swapping, the algorithm terminates immediately and returns 0.

Cost Model:

$$N^3/p + t_s p + t_w N^2 p + t_w(N-1) \tag{3}$$

3.5 Inverse

The inverse of a matrix can be computed by row reduction. Consider a matrix A_{nxn} in order to compute its inverse we augment it with I_{nxn} which is the Identity matrix of size n and convert $A|I$ to its row reduced echelon form. Once the echelon form is computed, a reverse row reduction is done to reduce the left side of the augmented matrix $A|I$ to an Identity Matrix. The right half of the augmented matrix is the required inverse. Note we do not check before hand if the matrix A is invertible. We assume invertibility until we encounter a zero pivot element for a row even after all possible swaps have been considered. If this condition is reached we immediately stop the algorithm and set the return value to 'Not invertible'.

Cost Model:

$$2N^3 p + t_s p + 4t_w N^2/p + t_w(N-1) \tag{4}$$

3.6 Matrix Transpose

1. **Matrix Distribution:**
   The matrix to be transposed is distributed to all the process available to us. Every process get a matrix of size $\left\lfloor \frac{N}{p} \right\rfloor$xN. In case if $N\%p = m$ where $m \neq 0$ we add the

remaining rows among the last rank. The chunk calculation is done by all processes as each of them receives the number of rows r and number of columns c through an initial broadcast by rank 0.

2. **Computation**
   Matrix of size $\left\lfloor \frac{N}{p} \right\rfloor$xN is transposed by each of the processes and the new matrix is sent back to the master process (rank: 0) where all the resultant matrix is merged and returned as the output.

3. **Cost model:**
$$t_s p + t_w p(N^2/p) + p(N^2/p)$$

3.7 QR Decomposition

QR is decomposition is decomposition of a matrix A into a product A = QR of an orthogonal matrix Q and an upper triangular matrix R. QR decomposition technique using Using the Gram–Schmidt process can be viewed at https://en.wikipedia.org/wiki/QR_decomposition. The QR decomposition on multi-process can be broken into the following steps: **Note:** Equation notation is followed from the aforementioned wikipedia link.

1. **Column Distribution:** The matrix A for for u_i calculation is distributed to all the available process. Every process get a matrix of size Nx$\left\lfloor \frac{N}{p} \right\rfloor$. In case if $N\%p = m$ where $m \neq 0$ we add the remaining columns among the last rank. The chunk calculation is done by all processes as each of them receives the number of rows r and number of columns c through an initial broadcast by rank 0.

2. **Computation:** Starting from rank 0 till every other ranks, all u_is belonging to their column range is computed and e_is are sent back to the master process (rank 0). At last the Q and R matrices are formed using the e_is and a_is. See the matrix representation of Q and R in the Wikipedia page mentioned above.

3. **Cost model:**
$$\begin{aligned} T(N,p,t_s,t_w) = t_s p + \frac{(N^2/p^2)(N/2+1)^2}{8} N + \\ \frac{(N/p)(N/p+1)(2N/p+1)}{12} N + \\ \frac{(N/p)(N/p+1)(2N/p+1)}{3} t_w - \\ N/p(N/p+1)t_w \end{aligned}$$

3.8 Eigen Values & Eigen Vectors

1. **Computation:** It majorly uses distributed QR decomposition technique discussed in former section. The algorithm as follows:
   - Let X by a symmetric matrix.
   - Let X_1=X, and iterate the following:
   - Given X_k, compute QR decomposition $X_k = Q_k R_k$, and let $X_{k+1} = R_k Q_k$.

- The matrices sequence X_n converges to some diagonal matrix D with the eigenvalues on the diagonal.
- The corresponding eigenvectors can be retrieved from the columns of $\Pi_i Q_i$

2. **Cost model:** Let K be the number of iteration for diagonalization of matrix

$$KT(N, p, t_s, t_w)N^3$$

4 CHAPTER 4: RESULTS & CONCLUSIONS

All very well known computationally expensive linear algebraic computation are implemented by distributing the calculations among different processes, giving a perception of single coherent machine computation to the end user.

5 FUTURE SCOPE

We aim to package our files using Python package guidelines and make it available by pip. Also the algorithms being used can be modified to more modern implementations available in literature. Nonetheless, it would be worth introducing checkpointing for the fault tolerance.

References

[1] G. Fagg, J. Dongarra, and A. Geist, "Heterogeneous mpi application interoperation and process management under pvmpi.," pp. 91–98, 01 1997.

[2] "Point to Point message." http://hamilton.nuigalway.ie/teaching/AOS/NINE/MPI.htm.

[3] "MPI collective." https://pages.tacc.utexas.edu/~eijkhout/pcse/html/mpi-collective.html.

[4] "Server-Client." https://www.tutorialspoint.com/perl/perl_socket_programming.htm.

[5] Y. Yu, M. Tang, W. G. Aref, Q. M. Malluhi, M. M. Abbas, and M. Ouzzani, "In-memory distributed matrix computation processing and optimization," in *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pp. 1047–1058, IEEE, 2017.

[6] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, *et al.*, "Apache spark: a unified engine for big data processing," *Communications of the ACM*, vol. 59, no. 11, pp. 56–65, 2016.

[7] A. Ramamoorthy, A. B. Das, and L. Tang, "Straggler-resistant distributed matrix computation via coding theory," *arXiv preprint arXiv:2002.03515*, 2020.

[8] Z. Qian, X. Chen, N. Kang, M. Chen, Y. Yu, T. Moscibroda, and Z. Zhang, "Madlinq: Large-scale distributed matrix computation for the cloud," 11 2012.

[9] R. Gu, Y. Tang, Z. Wang, S. Wang, X. Yin, C. Yuan, and Y. Huang, "Efficient large scale distributed matrix computation with spark," in *2015 IEEE International Conference on Big Data (Big Data)*, pp. 2327–2336, 2015.