COMPARISON BETWEEN M.P.I & MAPREDUCE PARALLEL PROGRAMMING TECHNIQUES

-ARINDAM SEN & MINTU BISWAS.
ROLL-33 & 14
DEPT. OF INFORMATION TECHNOLOGY
JADAVPUR UNIVERSITY.

INTRODUCTION:

In this project report, we try to present our observations while trying to compare the performance of **M.P.I** (Message Passing Interface) which is a standardized and portable message-passing system designed by a group of researchers from academia and industry to function on a wide variety of parallel computers and **MAPREDUCE** which is a programming model and a proprietary *Google* technology for processing large data sets with a parallel, distributed algorithm in a cluster.

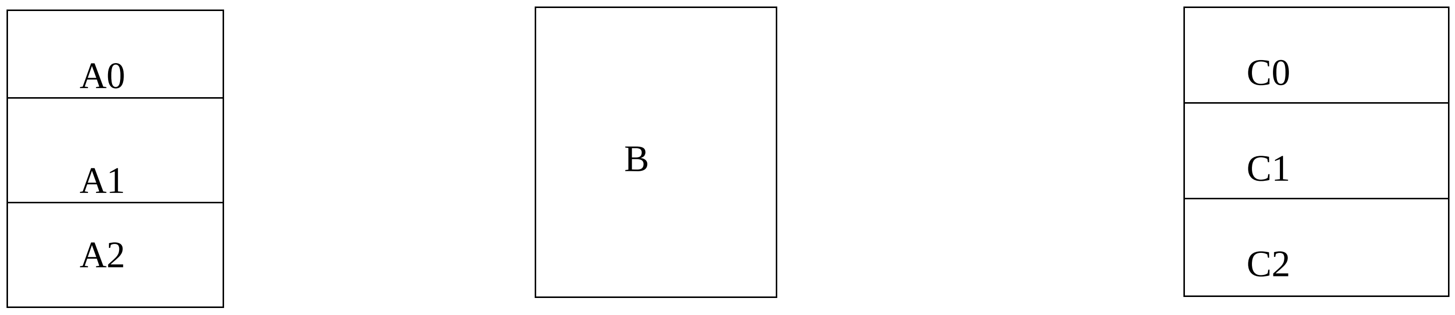
For our performance comparison purpose, we have chosen Matrix Multiplication as our reference . Computing matrix multiplication for matrices with very large number of rows/columns intuitively seems to be less time consuming if computed in parallel using multiple nodes.

Since we are bothered only about the time required to compute the matrix multiplication and not so much about the multiplication result in particular, we have only dealt with square matrix multiplications in this project report. However the codes presented in this report for the matrix multiplications can be easily modified to include non-square matrices.

M.P.I STRATEGY:

We shall use the Direct Row Stripped Partitioning to parallelize the matrix multiplication. It is a master-slave Programming approach.

- One master, several slaves.
- One node will run the master program, the rest will run the slave program.



Matrix A is split equally according to number of slave processes and then sent to the slaves.

Matrix B is sent to all slaves as it is.

Each slave computes each row of Matrix C and sends it back to the master.

We use the default **MPI_COMM_WORLD** communicator and specify a ***hostfile*** named my_hostfile which contains all the IP addresses of the nodes we use for the parallel operations. The top most IP address in the hostfile is the master node and the others are the slaves.
Using **MPI_SEND** and **MPI_RECV** calls we communicate and pass over matrix values and receive results.

CODE:

MASTER :

```
if (taskid == MASTER) {
printf(" Numberofslaves tasks = %d%n",numslaves );
for (i=0; i<N; i++)
for (j=0; j<N; j++)
a[i][j]= i+j;
for (i=0; i<N; i++)
for (j=0; j<N; j++)
b[i][j]= i*j;
/* sending matrix data to the slaves tasks */
averow = N/numslaves ;
extra = N%numslaves ;
offset = 0;
mtype = FROM_MASTER;
for (dest=1; dest<=numslaves ; dest++) {
rows = (dest <= extra) ? averow+ 1 : averow;
printf("sending %d rows to
task %d n",rows ,dest);
MPI_Send( &offset, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);
MPI_Send( &rows , 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);
MPI_Send( &a[offset][0],rows*N,PI_DOUBLE dest,mtype,MPI_COMM_WORLD);
MPI_Send(&b, N*N, MPI_DOUBLE, dest, mtype, MPI_COMM_WORLD);
offset = offset + rows;
}
/* waiting for results from all slaves tasks */
mtype = FROM_SLAVES;
for (i=1; i<=numslaves; i++) {
source = i;
MPI_Recv(&offset, 1, MPI_INT, source, mtype, MPI_COMM_WORLD, &status);
MPI_Recv(& rows , 1, MPI_INT, source, mtype, MPI_COMM_WORLD, &status);
MPI_Recv(&c[offset][0],rows*N,MPI_DOUBLE, source,mtype,MPI_COMM_WORLD, &status);
}
```

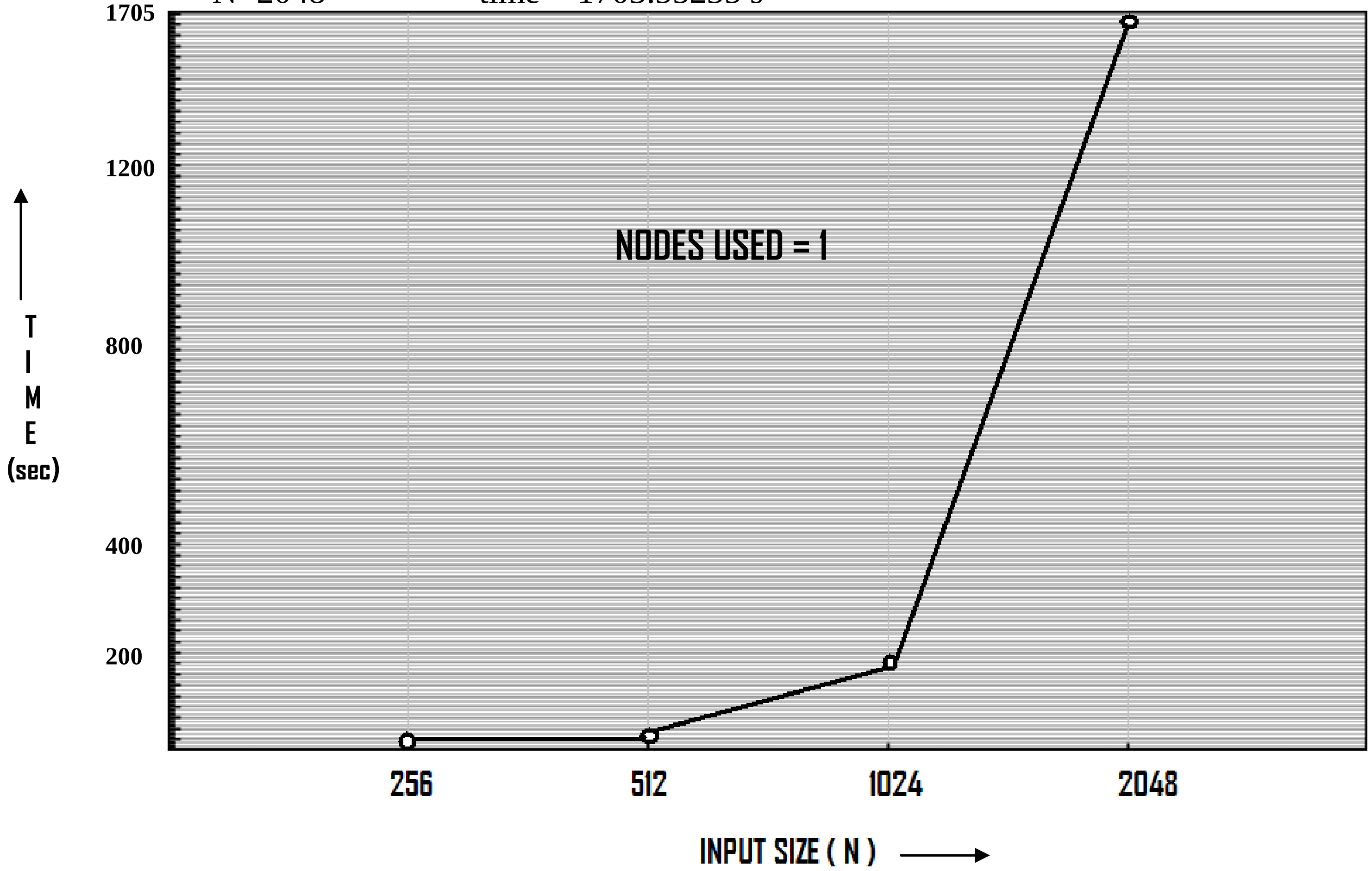
SLAVES:

```
if (taskid > MASTER) {
mtype = FROM_MASTER;
MPI_ Recv(&offset,1,MPI_INT,MASTER, mtype, MPI_COMM_WORLD, &status);
MPI_Recv(&rows ,1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD, &status);
MPI_Recv(&a,rows *N,MPI_DOUBLE, MASTER, mtype,MPI_COMM_WORLD,&status);
MPI_Recv(&b,N*N,MPI_DOUBLE,MASTER, mtype,MPI_COMM_WORLD, &status);
for (k=0; k<N; k++)
for (i=0; i<rows; i++) {
c[i][k] = 0.0;
for (j=0; j<N; j++)
c[i][k] = c[i][k] + a[i][j] * b[j][k];
}
mtype = FROM_SLAVES;
MPI_Send(&offset,1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD);
MPI_Send(&rows , 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD);
MPI_Send( &c,rows*N,MPI_DOUBLE, MASTER, mtype, MPI_COMM_WORLD);
```

OBSERVATIONS:

1. TIME TAKEN FOR SINGLE NODE vs VARYING INPUT SIZE:

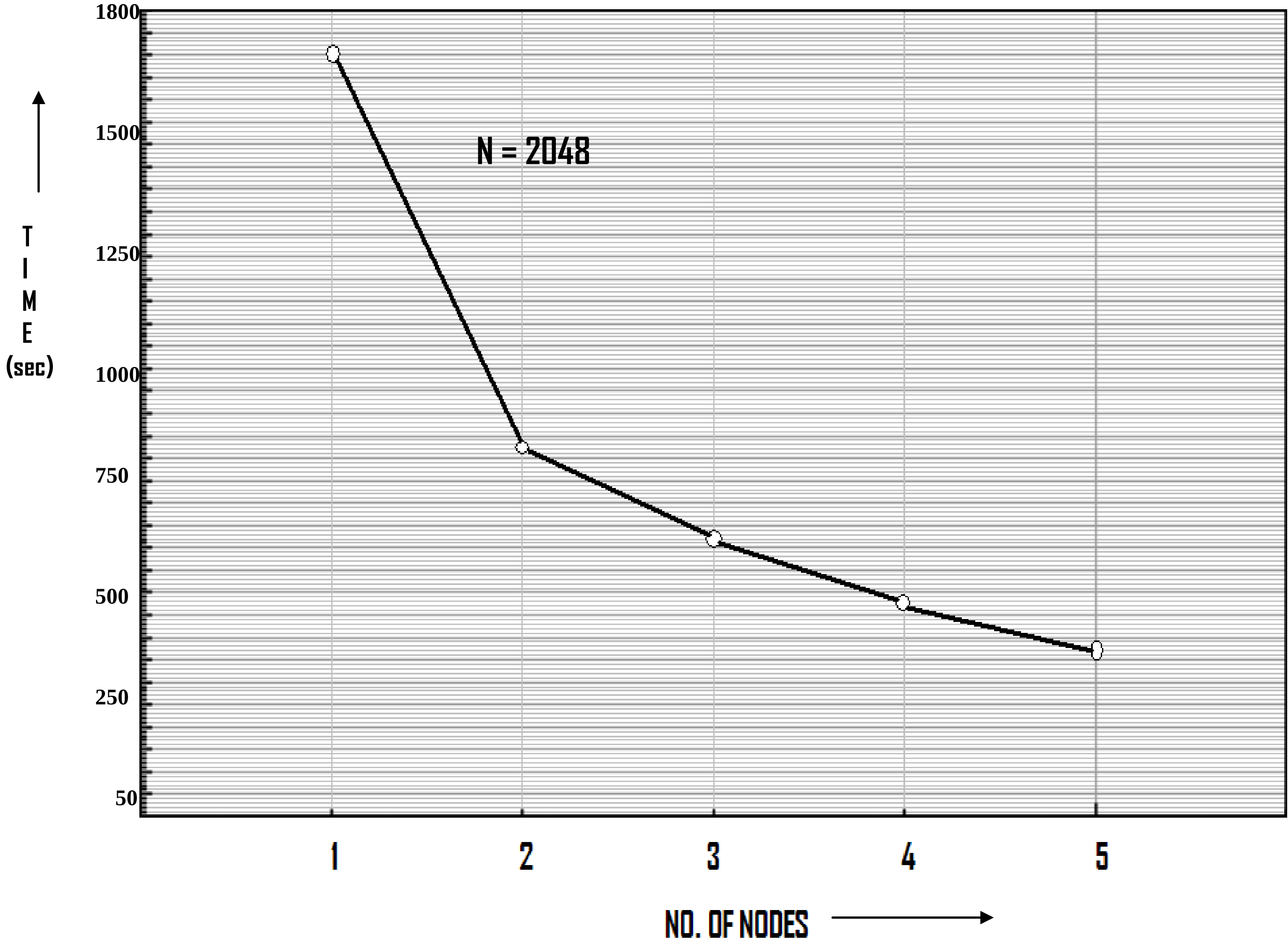
• N=256	time = 3.0007113 s
• N=512	time = 23.523900 s
• N=1024	time = 205.393565 s
• N=2048	time = 1703.55235 s



We see that as input size increases, the time required for computation also increases greatly as expected and it increases more or less proportionately as observed from the curve of time vs input size. For an input size of N=2048, we had to wait for as long as 28 minutes to get the result!! This indicates the poor performance of single processor for large input size.

2. TIME TAKEN FOR MULTIPLE NODES vs CONSTANT INPUT SIZE:

• N=2048	No. Of Nodes = 2	time = 847.604325 s
• N=2048	No. Of Nodes = 3	time = 622.724525 s
• N=2048	No. Of Nodes = 4	time = 495.601600 s
• N=2048	No. Of Nodes = 5	time = 397.294800 s

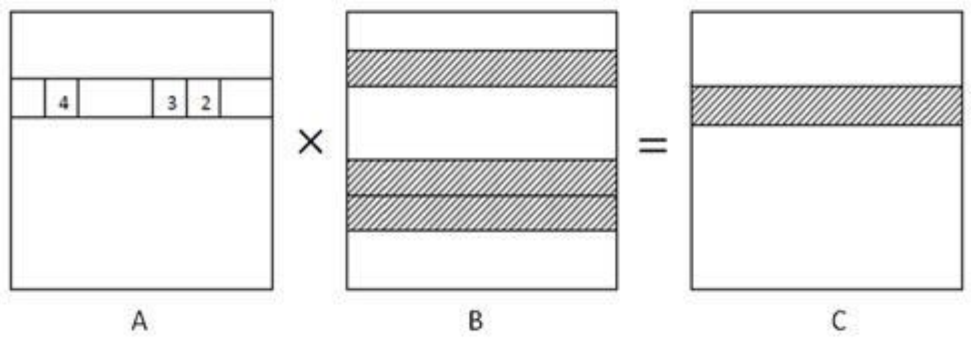


As the number of nodes increases for a given input size (N = 2048) we see that the running times decrease drastically which is what we expect. Thus, as the number of nodes increases, substantial speedup is achieved as indicated by the curve of time vs number of nodes.

Ideally if we double the number of nodes, the running times should be halved but this is not the case because there are overheads like the time wasted by blocking operations such as MPI_Send, MPI_Recv, MPI_Barrier, and the actual communication itself.

MAP REDUCE STRATEGY:

The first matrix **A** is partitioned into **partsize X partsize** square blocks, and the second matrix **B** into **partsize** row strips. A map task multiplies a block from the first matrix and a row strip from the second, and produces partial results for a row strip of the output matrix **C**. We are using sparse matrices and hence this is a good strategy.



Every block matrix multiplication is handled by one **MAP** task.
Since the total amount of multiplication is fixed, when the number of map slots is also fixed.

Matrix data is stored in a file on Hadoop with **line numbers at the beginning of each line in the file.**

For example this represents a 3X3 matrix on Hadoop.

0 1 1 1
1 1 1 1
2 1 1 1

Matrix **A** is stored in a file named "**left**".
Matrix **B** is stored in a file named "**right**".
Both these files are in a folder named "**input**".
Results are placed in a folder named "**output**" inside the Hadoop File System (**H.D.F.S**)
Every file in the "**output**" folder will contain one row of matrix **C.**

CODE:

```
/*

 hadoop jar matrixmul.jar MatrixMul input output 2048 32


*/


public class MatrixMul {

  public static class MyMapper extends Mapper<LongWritable, Text, IntWritable, Text>{

    private String filename=null;
    private boolean isLeftMatrix=false;
    private int totalSize, partSize, npart;

    private boolean isLeft(){return isLeftMatrix;}
    protected void setup(Context context) throws IOException, InterruptedException{
        //get filename
        FileSplit fileSplit = (FileSplit)context.getInputSplit();
        filename = fileSplit.getPath().getName();
        if("left".equalsIgnoreCase(filename))
                isLeftMatrix=true;
        else
                isLeftMatrix=false;

        //getting  size and partition information
        Configuration conf=context.getConfiguration();
        totalSize=conf.getInt("matrix-mul-totalsize", -1);
        partSize=conf.getInt("matrix-mul-partsize", -1);
        npart=conf.getInt("matrix-mul-npart", -1);
        if(totalSize<0 || partSize<0 || npart<0){
                System.out.println("Error in setup of MyMapper.");
                System.exit(1);
        }
    }
```

MAPPER:

```
    public void map(LongWritable key, Text value, Context context
                    ) throws IOException, InterruptedException {
        String line=value.toString();
        String[] strs=line.split(" ");
        if(strs.length!=totalSize+1){
                System.out.println("Error in map of Mapper.");
                System.out.println(strs.length+"___"+totalSize);
                System.out.println("line is: "+line);
                System.exit(1);
        }
        int linenum=Integer.parseInt(strs[0]);
        int[] numbers=new int[totalSize];
        for(int i=0;i<totalSize;i++)
                numbers[i]=Integer.parseInt(strs[i+1]);
        int part_hor=linenum/partSize; //horizontal partitioned id
        int prev_part_ver=-1;
        String msg=null;
        for(int i=0;i<totalSize;i++){
                int part_ver=i/partSize; //vertical partition number
                if(part_ver!=prev_part_ver){
                        if(msg!=null){
                                int baselinenum = part_hor * partSize;
                                int old=part_ver;
                                part_ver=prev_part_ver;
                                if(isLeft()){
                                        String toSend="l:"+(linenum -
baselinenum)+":"+part_ver+"#"+msg;
                                        System.out.println("left
"+linenum+","+part_ver+" "+msg);
                                        for(int k=0;k<npart;k++){
                                                int dest=part_hor * npart + k;
                                                context.write(new IntWritable(dest),
new Text(toSend));
                                        }
                                }else{
                                        String toSend="r:"+(linenum -
baselinenum)+":"+part_hor+"#"+msg;
                                        System.out.println("right
"+part_ver+":"+linenum+" "+msg);
                                        for(int k=0;k<npart;k++){
                                                int dest=k * npart + part_ver;
                                                context.write(new IntWritable(dest),
new Text(toSend));
                                        }
                                }
                                part_ver=old;
                        }
                        msg=null;
                        prev_part_ver=part_ver;
                }
                if(msg==null)
                        msg=""+strs[i+1];
                else
                        msg+=" "+strs[i+1];
        }
        if(msg!=null){
                int part_ver=npart-1;
                int baselinenum = part_hor * partSize;
                if(isLeft()){
                        String toSend="l:"+(linenum -
baselinenum)+":"+part_ver+"#"+msg;
                        System.out.println("left "+linenum+","+part_ver+" "+msg);
                        for(int k=0;k<npart;k++){
                                int dest=part_hor * npart + k;
                                context.write(new IntWritable(dest), new
Text(toSend));
                        }
                }else{
                        String toSend="r:"+(linenum -
baselinenum)+":"+part_hor+"#"+msg;
                        System.out.println("right "+part_ver+":"+linenum+" "+msg);
                        for(int k=0;k<npart;k++){
                                int dest=k * npart + part_ver; //has to be the last
part
                                context.write(new IntWritable(dest), new
Text(toSend));
                        }
                }
        }
    }
  }
```

REDUCERS:

```
public static class MyReducer extends Reducer<IntWritable, Text, Text, Text> {

    private int totalSize, partSize, npart;
    int[][] left=null;
    int[][] right=null;
    protected void setup(Context context) throws IOException, InterruptedException{

        //getting the number of partitions

        Configuration conf=context.getConfiguration();
        totalSize=conf.getInt("matrix-mul-totalsize", -1);
        partSize=conf.getInt("matrix-mul-partsize", -1);
        npart=conf.getInt("matrix-mul-npart", -1);
        if(totalSize<0 || partSize<0 || npart<0){
                System.out.println("Error in setup of MyReducer.");
                System.exit(1);
        }
        left=new int[partSize][totalSize];
        right=new int[totalSize][partSize];
    }
    public void reduce(IntWritable key, Iterable<Text> values, Context context
                    ) throws IOException, InterruptedException {
        int sum = 0;
        for (Text val : values) {
                String line=val.toString();
                String[] meta_val=line.split("#");
                String[] metas=meta_val[0].split(":");
                String[] numbers=meta_val[1].split(" ");

                int baselinenum=Integer.parseInt(metas[1]);
                int blkindex=Integer.parseInt(metas[2]);
                if("l".equalsIgnoreCase(metas[0])){ //from left matrix
                        int start=blkindex * partSize;
                        for(int i=0;i<partSize; i++)

left[baselinenum][start+i]=Integer.parseInt(numbers[i]);
                }else{
                        int rowindex=blkindex*partSize + baselinenum;
                        for(int i=0;i<partSize; i++)
                                right[rowindex][i]=Integer.parseInt(numbers[i]);
                }
        }
    }
    protected void cleanup(Context context) throws IOException, InterruptedException {

        int[][] res=new int[partSize][partSize];
        for(int i=0;i<partSize;i++)
                for(int j=0;j<partSize;j++)
                        res[i][j]=0;
        for(int i=0;i<partSize;i++){
                for(int k=0;k<totalSize;k++){
                        for(int j=0;j<partSize;j++){
                                res[i][j]+=left[i][k]*right[k][j];
                        }
                }
        }
        for(int i=0;i<partSize;i++){
                String output=null;
                for(int j=0;j<partSize;j++){
                        if(output==null)
                                output=""+res[i][j];
                        else
                                output+=" "+res[i][j];
                }
                context.write(new Text(output), null);
        }
    }
  }
```

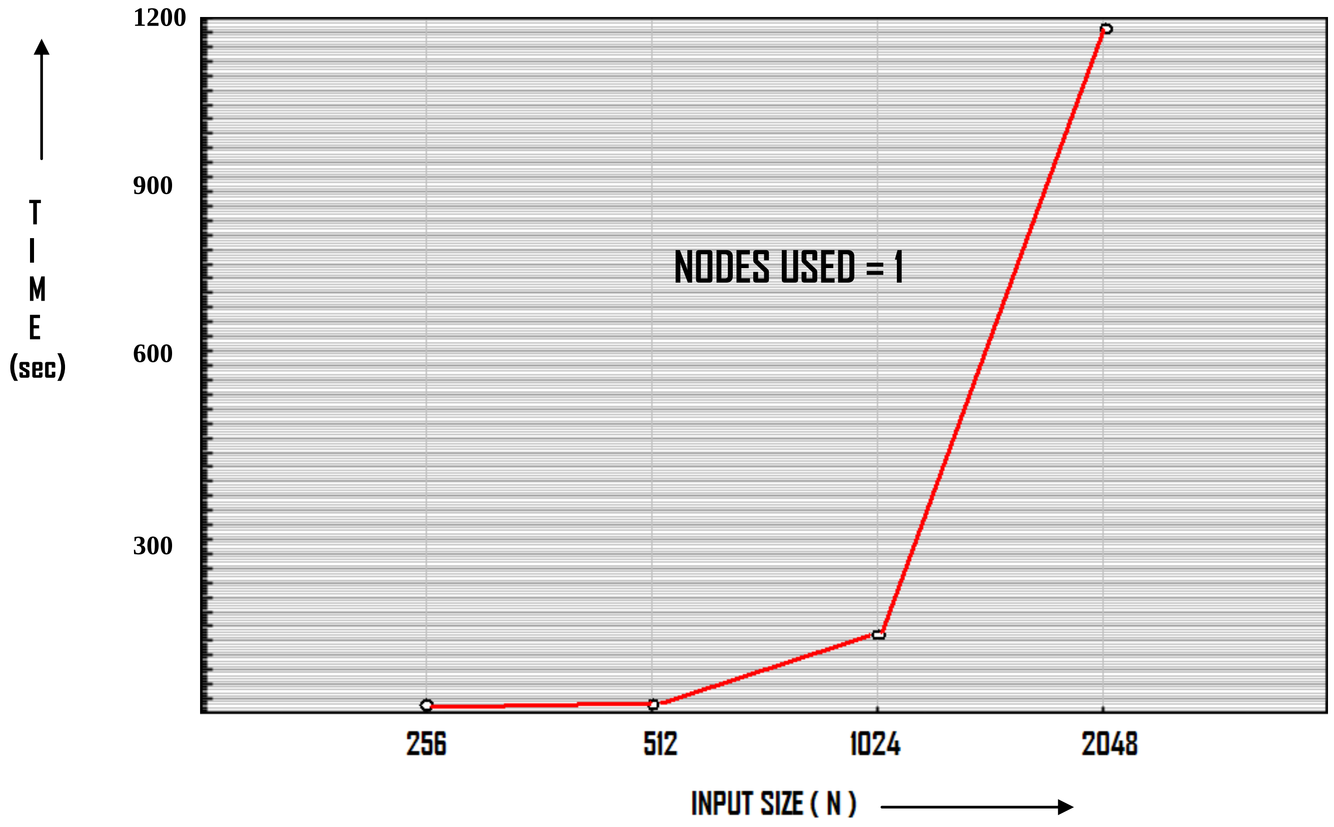
DRIVER PROGRAM:

```
  public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    if (args.length != 4) {
        System.err.println("Usage: MatrixMul input-dir output-dir total-size part-
size");
        System.exit(2);
    }
    int totalsize=Integer.parseInt(args[2]);
    int partsize=Integer.parseInt(args[3]);
    if(totalsize==0 || partsize==0 || partsize>totalsize){
        System.out.println("Invalid total-size or part-size");
        System.exit(1);
    }
    conf.setInt("matrix-mul-totalsize", totalsize); //the matrix is 'totalsize' by
'totalsize'
    conf.setInt("matrix-mul-partsize", partsize); //every block is 'partsize' by
'partsize'
    int npart=totalsize/partsize;
    if(npart*partsize<totalsize)
        npart++;
    conf.setInt("matrix-mul-npart", npart); //number of parts on one dimension
    Job job = new Job(conf, "matrix-mul");
    job.setJarByClass(MatrixMul.class);
    job.setMapperClass(MyMapper.class);
    job.setReducerClass(MyReducer.class);
    job.setNumReduceTasks(npart*npart);

    job.setOutputKeyClass(IntWritable.class);
    job.setOutputValueClass(Text.class);

        TextInputFormat.addInputPath(job, new Path(args[0]));
FileOutputFormat.setOutputPath(job, new Path(args[1]));
    job.waitForCompletion(true) ;
  }
}
```

OBSERVATIONS:

1. TIME TAKEN FOR SINGLE NODE vs VARYING INPUT SIZE:

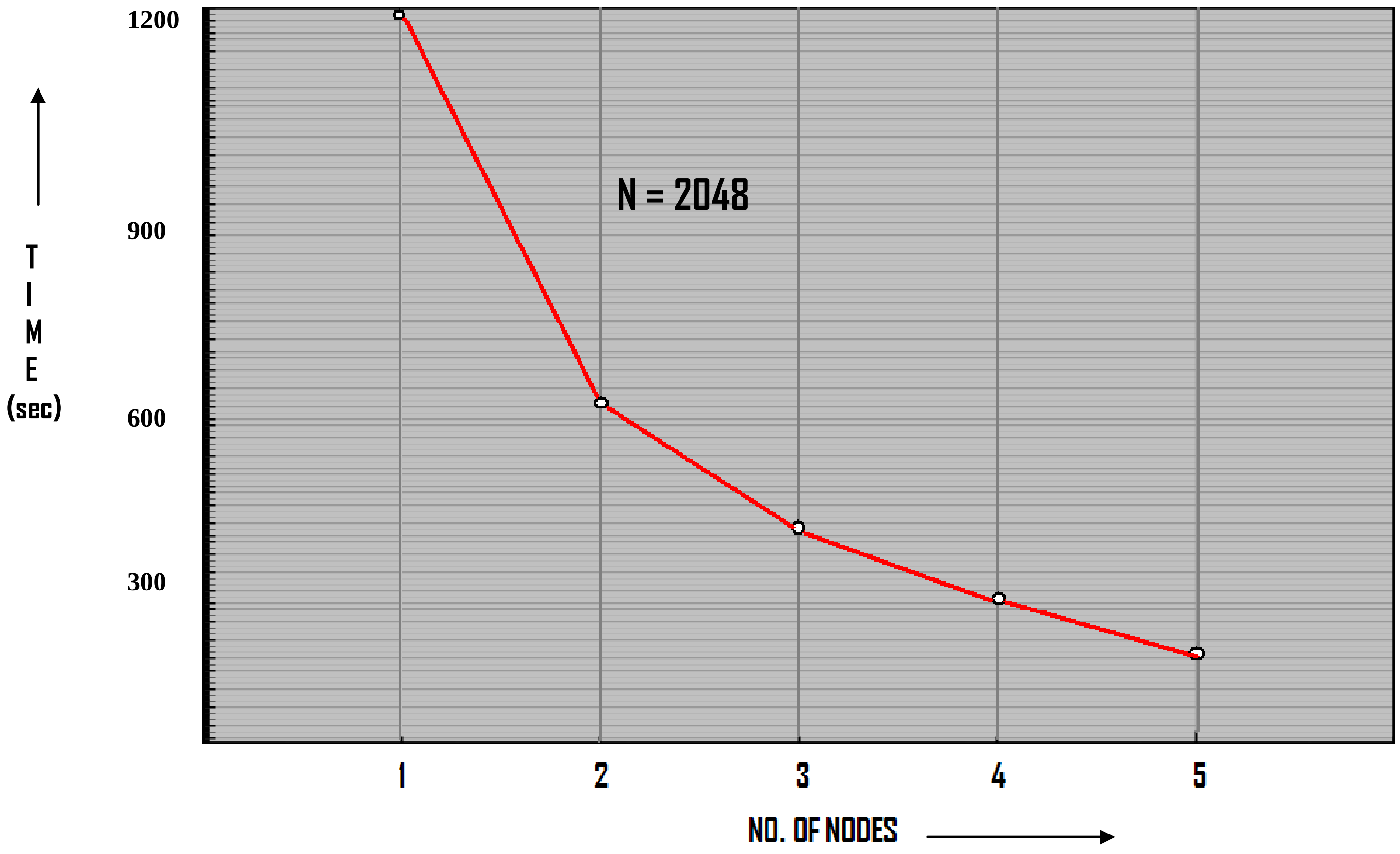
N=256	time = 2.87341232 s
N=512	time = 10.3352390 s
N=1024	time = 133.935650 s
N=2048	time = 1191.65650 s



Like M.P.I, here also we get increasing execution times with increasing input size. However we found a marked improvement over the performance of M.P.I. For an input size of N=2048 the execution time was around 20 mins, whereas it was around 28 mins for M.P.I

2. TIME TAKEN FOR <u>MULTIPLE NODES</u> vs CONSTANT INPUT SIZE:

• N=2048	No. Of Nodes = 2	time = 605.423615 s
• N=2048	No. Of Nodes = 3	time = 385.252325 s
• N=2048	No. Of Nodes = 4	time = 251. 006153 s
• N=2048	No. Of Nodes = 5	time = 161.429800 s



The execution time decreases as number of nodes in the cluster increases but as compared to M.P.I the performance is vastly improved. The characteristic curve of Mapreduce is lower than that of the corresponding curve of MPI, indicating a larger speedup.

Ideally, when we double the cluster size, the running time should be halved. However, there are **overheads** such as **HDFS I/O cost**, **shuffle cost**, **map startup cost**, etc.

WHY MAPREDUCE IS PERFORMING BETTER THAN M.P.I?

From the results that we obtained it was evident that Mapreduce performed much better than M.P.I for our given matrix multiplication.

MPI is best suited for problems that require a lot of inter-process communication. When Data becomes large and there is little inter-process communication, MPI becomes a pain. This is so because the processes will spend all the time sending data to each other (bandwidth becomes a limiting factor) and the CPUs will remain idle. Perhaps an even bigger problem is reading all that data.

This is the fundamental reason behind having something like Hadoop. The Data also has to be distributed - Hadoop Distributed File System! The inter-process communications required in MPI are inherently built in Hadoop. So we don't have to explicitly manage these communications, rather the Hadoop framework does this for us. Thus in general Mapreduce using Hadoop yields better performance than MPI.

To say all this in short, MPI is good for task parallelism and Hadoop is good for Data Parallelism.

CONCLUSION:

In this project we implemented and analyzed the parallel matrix multiplication implementations on a MPI cluster using master/slaves algorithm and on a Hadoop cluster using Map/Reduce algorithm to achieve high performance. It has been shown that the performance of parallel model is reliable and saves more time than the serial model. It has also been shown that MAPREDUCE works better than MPI for our given problem.