COL380 - Introduction to Parallel and Distributed Programming

Assignment 2

Matrix Multiplication using OpenMPI

Pradyumna Meena(2016CS10375)

Manav Rao(2016CS10523)

Algorithm

```
for(i = 0;i<n;i++){
    for(j = 0;j<n;j++){
        val = 0;
        for(k = 0;k<m;k++){
            val+=(A[i*m + k] * B[k*n + j]);
        }
        prod[i*n + j] = val;
    }
}
```

- An important thing is that computation of any position (i,j) in the product matrix is unaffected by any other position in the same matrix. So we split the work in the threads row-wise i.e. each thread will execute the multiplication for a certain number of rows of A matrix.

Difference in MPI calls

1. *P2P blocking send and receive*
    a. If a process is sending certain data to another process then it cannot proceed further unless the other process has received the data.
    b. Since sending data to two different processes is independent of each other, therefore this will lead to an increase in the running time of the algorithm.

```
for(int i = 1;i<=num_workers;i++){
   if(i==num_workers && num_workers>1){
      load = n%num_workers;
   }
   MPI_Send(&load,1,MPI_INT,i,0,MPI_COMM_WORLD);
   MPI_Send(&A[idx*m],load*m,MPI_FLOAT,i,0,MPI_COMM_WORLD);
   MPI_Send(&B[0],n*m,MPI_FLOAT,i,0,MPI_COMM_WORLD);
   idx+=load;
}

idx = 0;
load = n/num_workers;
for(int i = 1;i<=num_workers;i++){
   if(i==num_workers && num_workers>1){
      load = n%num_workers;
   }
   MPI_Recv(&prodM[idx*m],load*n,MPI_FLOAT,i,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
   idx+=load;
}
```

```
if(rank==0){ ...

}else{
   MPI_Recv(&load,1,MPI_INT,0,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
   MPI_Recv(&A[0],load*m,MPI_FLOAT,0,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
   MPI_Recv(&B[0],n*m,MPI_FLOAT,0,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);

   double val;
   for(int i = 0;i<load;i++){
      for(int j = 0;j<n;j++){
         val = 0;
         for(int k = 0;k<m;k++){
            val+=(A[i*m + k]*B[j*m + k]);
         }
         C[i*n+j] = val;
      }
   }

   MPI_Send(&C[0],load*n,MPI_FLOAT,0,0,MPI_COMM_WORLD);
}
```

2. *P2P non-blocking send and receive*
    a. This helps us relax the restriction on the order in which the MPI_Send and MPI_Recv requests will be executed.
    b. Sending and receiving data to and from different threads is independent of each other. Hence we can provide all requests at once and then wait for all of them to complete and then proceed further.

```
for(int i = 1;i<=num_workers;i++){
   if(i==num_workers && num_workers>1){
      load = n%num_workers;
   }
   MPI_Isend(&load,1,MPI_INT,i,0,MPI_COMM_WORLD,&sends[3*(i-1)]);
   MPI_Isend(&A[idx*m],load*m,MPI_FLOAT,i,0,MPI_COMM_WORLD,&sends[3*(i-1) + 1]);
   MPI_Isend(&B[0],n*m,MPI_FLOAT,i,0,MPI_COMM_WORLD,&sends[3*(i-1) + 2]);
   idx+=load;
}

MPI_Waitall(3*num_workers,sends,send_status);

idx = 0;
load = n/num_workers;
for(int i = 1;i<=num_workers;i++){
   if(i==num_workers && num_workers>1){
      load = n%num_workers;
   }
   MPI_Irecv(&prodM[idx*m],load*n,MPI_FLOAT,i,0,MPI_COMM_WORLD,&receive[i-1]);
   idx+=load;
}

MPI_Waitall(num_workers,receive,receive_status);
```

```
if(rank==0){ ...

}else{
   MPI_Irecv(&load,1,MPI_INT,0,0,MPI_COMM_WORLD,&proc_receive1);
   MPI_Wait(&proc_receive1,&proc_receive_status1);

   MPI_Irecv(&A[0],load*m,MPI_FLOAT,0,0,MPI_COMM_WORLD,&proc_receive2[0]);
   MPI_Irecv(&B[0],n*m,MPI_FLOAT,0,0,MPI_COMM_WORLD,&proc_receive2[1]);
   MPI_Waitall(2,proc_receive2,proc_receive_status2);

   double val;
   for(int i = 0;i<load;i++){
      for(int j = 0;j<n;j++){
         val = 0;
         for(int k = 0;k<m;k++){
            val+=(A[i*m + k]*B[j*m + k]);
         }
         C[i*n+j] = val;
      }
   }

   MPI_Isend(&C[0],load*n,MPI_FLOAT,0,0,MPI_COMM_WORLD,&send_req);
   MPI_Wait(&send_req,&send_stat);
}
```

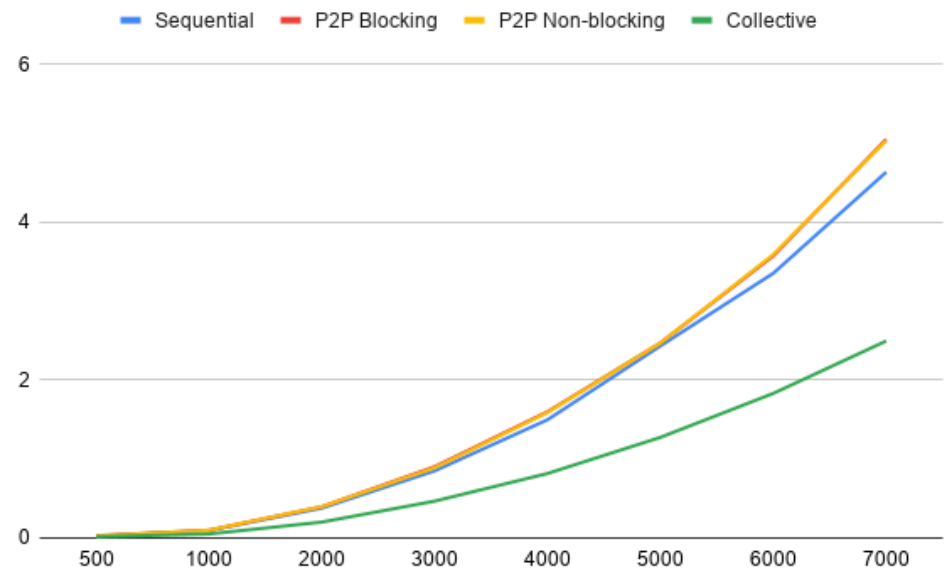
3. *Collective Communication*
    a. Here instead of process to process communication we have a process communicating with all other processes present in the MPI_COMM_WORLD.
    b. MPI_Bcast, MPI_Gather and MPI_Scatter were used for this communication.

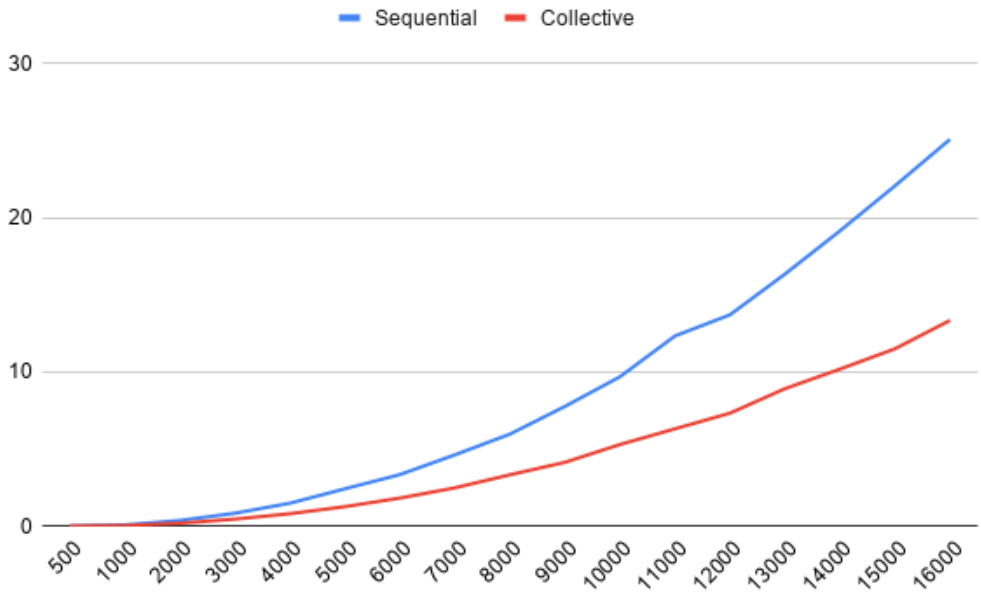
```
MPI_Bcast(&B[0],m*n,MPI_FLOAT,0,MPI_COMM_WORLD);
MPI_Scatter(&A[0],load*m,MPI_FLOAT,&A[0],load*m,MPI_FLOAT,0,MPI_COMM_WORLD);

double val;
for(int i = 0;i<load;i++){
   for(int j = 0;j<n;j++){
      val = 0;
      for(int k = 0;k<m;k++){
         val+=(A[i*m + k] * B[j*m + k]);
      }
      prodS[i*n + j] = val;
   }
}

MPI_Gather(&prodS[0],load*n,MPI_FLOAT,&prodM[0],load*n,MPI_FLOAT,0,MPI_COMM_WORLD);
```

Results

Matrix size	Sequential	P2P Blocking	P2P Non-blocking	Collective
500	0.024026	0.028946	0.025479	0.016159
1000	0.09467	0.099185	0.099144	0.051352
2000	0.379217	0.397048	0.397102	0.202269
3000	0.851331	0.904046	0.890437	0.467924
4000	1.49984	1.601073	1.591869	0.818463
5000	2.43496	2.472743	2.47024	1.276173
6000	3.3561	3.576685	3.594184	1.833411
7000	4.638081	5.054387	5.037784	2.497872

- Running time vs matrix size





Discussion

- *Should N be multiple of number of processors?*

  This restriction helps to distribute the work evenly between all the processes. In other cases the last process will have a lesser amount of work compared to other processes. This will lead to insufficient exploitation of parallel computing available at our hands.