Parallel Computing

Assignment #2 - Message Passing

Adrian Boghean
a11742914

1. Performance measurements

1.1 Raw performance data

Using MPI_Sendrecv

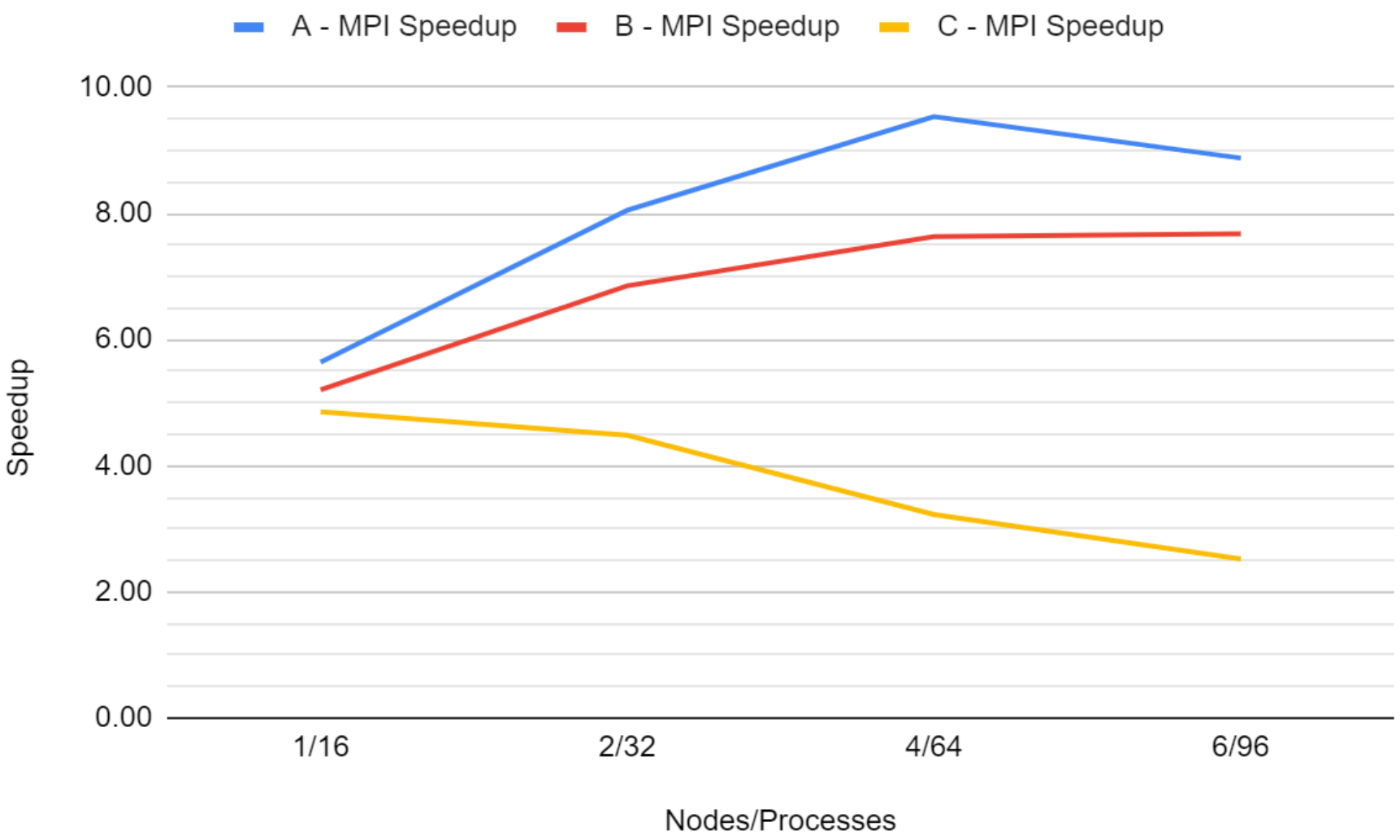
Nodes/Processes	A - Sequential	A - MPI	B - Sequential	B - MPI	C - Sequential	C - MPI	A - MPI Speedup	B - MPI Speedup	C - MPI Speedup
1/16	46.28	8.2	100.23	19.26	5.68	1.17	5.64	5.20	4.85
2/32	46.39	5.76	100.48	14.66	5.65	1.26	8.05	6.85	4.48
4/64	46.25	4.85	100.72	13.19	5.58	1.73	**9.54**	7.64	3.23
6/96	46.17	5.2	100.17	13.05	5.65	2.24	8.88	7.68	2.52

Using MPI_Send and MPI_Recv

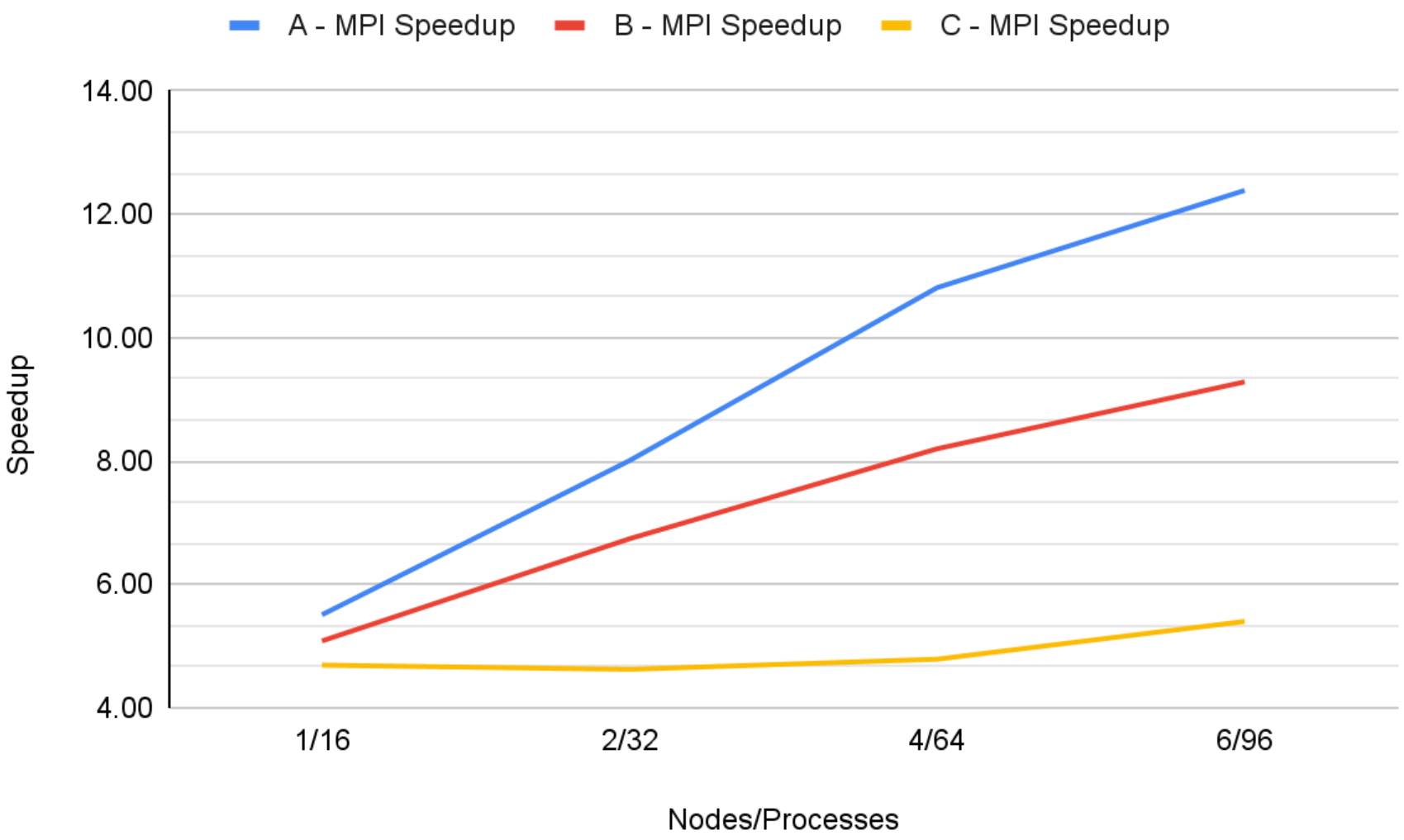
Nodes/Processes	A - Sequential	A - MPI	B - Sequential	B - MPI	C - Sequential	C - MPI	A - MPI Speedup	B - MPI Speedup	C - MPI Speedup
1/16	46.27	8.4	100.15	19.7	5.63	1.2	5.51	5.08	4.69
2/32	46.36	5.79	100.63	14.93	5.64	1.22	8.01	6.74	4.62
4/64	46.05	4.26	99.94	12.19	5.65	1.18	10.81	8.20	4.79
6/96	46.2	3.73	100.35	10.81	5.67	1.05	**12.39**	9.28	5.40

A -> --m 2688 --n 4096 --epsilon 0.001 --max-iterations 1000
B -> --m 2688 --n 4096 --epsilon 0.001 --max-iterations 2000
C -> --m 1152 --n 1152 --epsilon 0.001 --max-iterations 1000

1.2 Performance speedup using MPI_Sendrecv



1.3 Performance speedup using MPI_Send and MPI_Recv



2. Source code discussion

Firstly, I will describe briefly what my code does. I use two variables to describe the height of a local matrix, one includes the buffers, the other not. So basically there is always a difference of 2 between them, 2 being the number of buffers I have in each process.

```
// describes the height of the local matrix
int M_local = (M / size) + 2; //+2 are the buffer zones, one top one bottom.
int M_local_real = M / size; //height without buffers
```

Then I have started initializing the matrix, everything is 0 except the last row in the last local matrix. That would be the M_local-1 row of the matrix of the last process.

After that I have calculated the number of elements I have in each local matrix since I will need that value for the Gatherv. (counts and displacements) Mo

```
int receive_counts[size];
int receive_displs[size];

for (int i = 0; i < size; ++i)
{
    receive_counts[i] = local_elements_real;
    receive_displs[i] = local_elements_real *i;
}
```

And that's where the initialization finishes.

Then I had to deal with the communication. Firstly, I have used MPI_Sendrecv since it was mentioned in the slides that it avoids deadlocks but as it can be seen in the raw performance data and also in the graphs, the speedup was smaller than 12, and it also was worsening after using 4 nodes with 64 processes.

```
if (rank < size - 1){
    MPI_Sendrecv(&U_local(M_local_real, 0), N, MPI_DOUBLE, rank + 1, 0, &U_local(M_local_real + 1, 0), N, MPI_DOUBLE, rank + 1, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}

if (rank > 0){
    MPI_Sendrecv(&U_local(1, 0), N, MPI_DOUBLE, rank - 1, 1, &U_local(0, 0), N, MPI_DOUBLE, rank - 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

Since I had no ideas how to improve this, I have decided to try the same communication strategy, but using MPI_Send and MPI_Recv. As seen in the raw data and also in the graph, the results are better. When I have used 6 nodes and 96 processes on a 2688x4096 matrix with a maximum number of 1000 interactions, I have achieved a speedup of 12.39.

```
if(rank < size - 1){
    MPI_Send (&U_local(M_local_real,0), N, MPI_DOUBLE, rank + 1, 0, MPI_COMM_WORLD);
}

if(rank > 0){
   MPI_Recv (&U_local(0, 0), N, MPI_DOUBLE, rank - 1, 0, MPI_COMM_WORLD,  MPI_STATUS_IGNORE);
}


if(rank > 0){
    MPI_Send (&U_local(1,0), N, MPI_DOUBLE, rank - 1, 1, MPI_COMM_WORLD);
}

if(rank < size - 1){
    MPI_Recv (&U_local(M_local_real+1, 0), N, MPI_DOUBLE, rank + 1, 1, MPI_COMM_WORLD,  MPI_STATUS_IGNORE);

}
```

To calculate the diffnorm in each process, I have used MPI_Allreduce with MPI_SUM.

```
// computed the diffnorm from all processes
MPI_Allreduce(&diffnorm, &mpidiffnorm, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
mpidiffnorm = sqrt(mpidiffnorm);
```

To calculate the time needed for the MPI part, I have used MPI_Reduce exactly the same as in the example from MOPED.

```
// calculates elapsed time in the MPI part
double rank_elapsed_time = MPI_Wtime() - time_mpi_1;
double rank_time_max = -1;
MPI_Reduce(&rank_elapsed_time, &rank_time_max, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);
```

Then, I have used a Gatherv to gather the local matrix from each process into a new “large” matrix. My Gatherv also ignores the buffer rows since I have calculated the receive_counts and the receive_displs using the number of local elements excluding the elements on the buffer rows.

```
// gathers the local matrix from each process into a new "large" matrix
// ignores the buffers
Mat U(M_local_real*size, N);
MPI_Gatherv(&U_local(1, 0), local_elements_real, MPI_DOUBLE, &U(0, 0), receive_counts, receive_displs, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

An aspect worth mentioning is that when using the Gatherv like that, the resulting matrix will have the first two rows full of 0. That's why, at the very end, I had to move all the elements one row above.

3. Communication discussion

In P0 the process has to send the last "real" row to the P1 process. At the same time, P0 has to receive from P1, P1's first "real row. P0's first "real" row is not sent above, since there is no process above P0, only below. That's why all the sends of the first "real" row happen only if the rank is higher than 0, and they have as destination the process below (rank + 1).

In the very last process (Pn-1), the last "real" row can't be sent anywhere, since there are no processes below. That's why the last "real" row is sent only if the rank is smaller than size-1.

3.1 Performance

As already mentioned, the combination of MPI_Send and MPI_Recv has a better speedup than MPI_Sendrecv, but I am not sure why. Most likely, it is related to the fact that when using MPI_Sendrecv send and receive happen simultaneously and that means two blocking operations happen simultaneously. That will also explain why after using 4 nodes, the performance is no longer getting better but worse. The amount of simultaneously blockings affects the performance when the number of processes gets even higher.

With the combination of MPI_Send and MPI_Recv, as seen on the graph and raw data, the performance gets better every time the number of processes get higher. With a 2688x4096 matrix and 1000 iterations the speedup is the highest, but after using 4 nodes and 64 the speedup gain is a bit smaller but still far from flattening. When using the same matrix but with 2000 iterations, the speedup gain looks to be more constant between 1 and 6 nodes, around x1.4 for each extra node.

With a 1152x1152 matrix, and 1000 iterations, the speedup is almost the same when using 1,2 or 3 nodes. It gets noticeable better when using 4 nodes. But, out of the all three cases this one has the smallest speedup.

A conclusion might be that the bigger the matrix is, the higher the speedup will be.

4. Bottlenecks

As mentioned before, the number of blocking communications are a bottleneck in my implementation.

5. Improvements

Using non-blocking communication would speedup things a bit and would slow the speedup much later.

Another improvement in my implementation would be to make it possible to use an M number that is not a multiple of the process number. I have tried something, but I have failed to implement this.

```
// this doesn't work and I can't figure out why
// might have been useful if M%size != 0
/*
if (rank == size - 1){
    M_local = M_local + (M % size);
    M_local_real = M / size + (M % size);
}
*/
```