Lab 2 – MPI

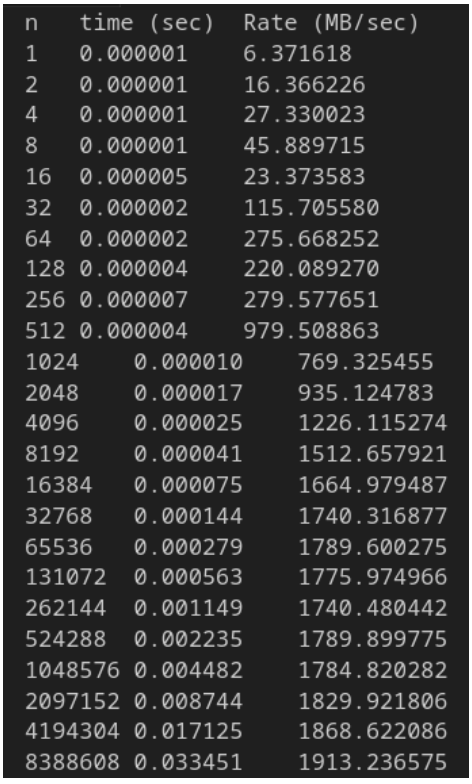
Name 1: Alejandro González _____NIA: 252658

Name 2: Luca Franceschi _____NIA: 253885

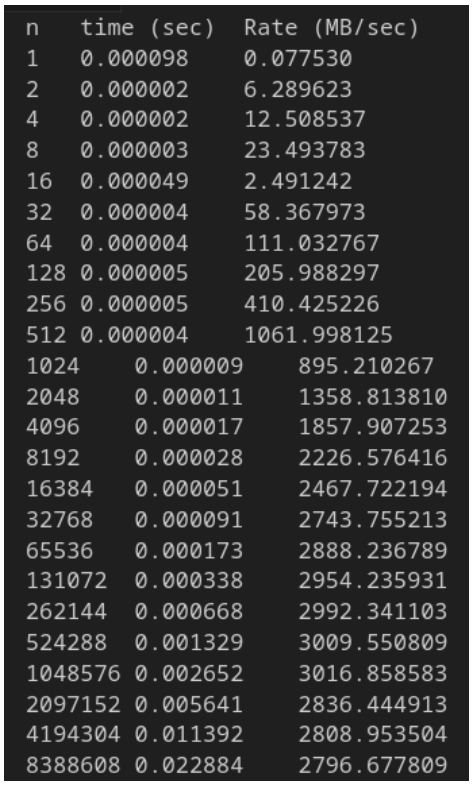
Send/Recv Rates:

1. **Obtain results when the two processes belong to different nodes. Explain them.**

Here are the outputs of running the program on one node (left image) and on two nodes (right image):



n	time (sec)	Rate (MB/sec)
1	0.000001	6.371618
2	0.000001	16.366226
4	0.000001	27.330023
8	0.000001	45.889715
16	0.000005	23.373583
32	0.000002	115.705580
64	0.000002	275.668252
128	0.000004	220.089270
256	0.000007	279.577651
512	0.000004	979.508863
1024	0.000010	769.325455
2048	0.000017	935.124783
4096	0.000025	1226.115274
8192	0.000041	1512.657921
16384	0.000075	1664.979487
32768	0.000144	1740.316877
65536	0.000279	1789.600275
131072	0.000563	1775.974966
262144	0.001149	1740.480442
524288	0.002235	1789.899775
1048576	0.004482	1784.820282
2097152	0.008744	1829.921806
4194304	0.017125	1868.622086
8388608	0.033451	1913.236575



n	time (sec)	Rate (MB/sec)
1	0.000098	0.077530
2	0.000002	6.289623
4	0.000002	12.508537
8	0.000003	23.493783
16	0.000049	2.491242
32	0.000004	58.367973
64	0.000004	111.032767
128	0.000005	205.988297
256	0.000005	410.425226
512	0.000004	1061.998125
1024	0.000009	895.210267
2048	0.000011	1358.813810
4096	0.000017	1857.907253
8192	0.000028	2226.576416
16384	0.000051	2467.722194
32768	0.000091	2743.755213
65536	0.000173	2888.236789
131072	0.000338	2954.235931
262144	0.000668	2992.341103
524288	0.001329	3009.550809
1048576	0.002652	3016.858583
2097152	0.005641	2836.444913
4194304	0.011392	2808.953504
8388608	0.022884	2796.677809

If we compare both scenarios, we see that for low n (n ranging from 1 to 32) the program on one node is much faster than when we run on two nodes. However, for larger n, the opposite happens, as the rate on two nodes is considerably larger than in one node.

It may seem like the case of n = 16 in two nodes is an outlier, but that's not the case, as we have tested many times the output of the code and it gives very similar results each time. We don't know exactly why this could be happening, but we think it is related to some type of overhead in the exchange of information or some storage-related reason such as memory hierarchy not being optimized for the size of the data.

2. **Check the documentation of the HPC DTIC cluster. Are we close to the maximum bandwidth for the case of different nodes?**

If we go to the documentation of the HPC DTIC cluster, we see that all nodes have a maximum bandwidth of 56 Gbps. This measure is in Gigabits per seconds, and the output of the code is given in Megabytes per second, so we need to convert 56 Gbps to MB/s. The result in MB/s is around 7000 MB/s. As seen in the previous images, when we run the program on two nodes, we achieve a connectivity of around 3000 MB/s, that is far from reaching the maximum bandwidth of each node.

We have not divided the time by 2 as we don't know whether we want to calculate the one corresponding to a send + receive operation or if we want to measure the time of each single operation. In the case we wanted to measure each single operation, the bandwidth achieved by the program would be around 6000 MB/s, so we would be closer to reaching the maximum connectivity of a node in the cluster.

Polling:

1. **Briefly explain how you have read the file in parallel. What functions have you used?**

To read the file in parallel, we have: created the needed MPI datatypes of tRecord named rectype; opened the file using an MPI call; divided the file into sections; depending on the number of processes that are running concurrently; read the information; and closed the file.

To read the information, each process has to first position its pointer to the one that corresponds to it due to its rank (using MPI_File_seek), then reads through a single call of MPI_File_read and writes all the data into a buffer with the needed capacity. Then it closes the file immediately since we don't need it anymore.

Then, we count all the votes locally, and when we have those local results ready, we make a call to MPI_Reduce to add all the counted votes to rank 0, which is the master process, and it prints the result in order most of the times.

We have some problems with printing the outputs of the file in order even when we flush the stdout buffers when we print anything and also creating some barriers (some of them seem to not work properly somehow).

Bugs:

1. **Explain what the bug is in each of the five cases.**

Bug 1:

In this file, the program has left hanging because the tags associated with the send/receive messages did not match, therefore it was not reaching its correct destination, and so the receive part was never completed as the message was sent wrongly, causing the program to be in a deadlock and hanging indefinitely.

Bug 2:

In this file, we were not receiving correctly the values sent by one of the processes because beta was wrongly declared. It was defined as a double, but in the send command it was being sent as an MPI_INT and being received as an MPI_DOUBLE. This caused an incompatibility that made the values incorrect.

Bug 3:

In this file, we were not obtaining the correct result on the final sum variable. This was due to the fact that a reduction was missing on the master part of the code. Also, there were some errors on the sending part, as we do not need the master to send anything to himself, just to the other process.

Bug 4:

In this file, we were getting a memory error due since we were trying to store some values on an array but with an index out of range. Also, secondary errors were the ordering of the send/receive operations and the wait all being done by a process that didn't need to.

Bug 5:

In this file, we were left with a hanging code because we were not doing the broadcast correctly. We need to change the value of the count variable to a fixed number.

2. **Explain how you have fixed each bug.**

Bug 1:

We simply changed the tag in both send/receive to the same one.

```
tag = 1234; // TAGS WERE DIFFERENT
        rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag,
MPI_COMM_WORLD);
        printf("Sent to task %d...\n",dest);
        rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag,
MPI_COMM_WORLD, &Stat);
        printf("Received from task %d...\n",source);
```

Bug 2:

We changed the definition of beta to be an int, and the receive operation with MPI_INT instead of MPI_DOUBLE.

```
    int beta; // Was declared as a double
```

```
            //Was receiving an MPI_DOUBLE
            MPI_Irecv(&beta, 1, MPI_INT, 0, tag, MPI_COMM_WORLD,
&reqs[i]);
```

Bug 3:

We added this reduce on the master part of the code:

```
        MPI_Reduce(&mysum, &sum, 1, MPI_DOUBLE, MPI_SUM, MASTER,
MPI_COMM_WORLD);
```

We also changed the starting indexes from 0 to 1:

```
        for (dest=1; dest<numtasks; dest++) {
```

```
        for (source=1; source<numtasks; source++) {
```

Bug 4:

We changed the offsets, so that for all processes it was 0:

```
    if (rank < 2) {
        nreqs = REPS*2;
        offset = 0;
        if (rank == 0) {
            src = 1;
        }
        if (rank == 1) {
            src = 0;
        }
        dest = src;
```

We did the Irecv command before the Isend command:

```
        for (i=0; i<REPS; i++) {
            MPI_Irecv(&buf, 1, MPI_INT, src, tag1, COMM,
&reqs[offset]);
            MPI_Isend(&rank, 1, MPI_INT, dest, tag1, COMM,
&reqs[offset+1]);
```

Lastly, we made process 3 (rank 2) not wait for all others as rank 2 is only making normal send commands:

```
    if (rank != 2) MPI_Waitall(nreqs, reqs, stats);
```

Bug 5:

We changed the value of the count variable from tasked to 1:

```
    count = 1;
```

This way, count has a constant value within all the threads.

Stencil:

1. **Explain how you have solved the problem.**

First of all, we had to create the variables that define the environment.

```
    /* determine my coordinates (x,y) -- rank=x*a+y in the 2d
processor array */
    rx = rank % px;
    ry = rank / px; // Integer division truncates result
```

For computing the rank in x, we can use the modulus by the number of processes in x, and for the one in y we can do a plain division also by px since it truncates the result.

Then we have to calculate the ranks to which we have to send information. We had previously initialized them to -1 in case a neighbor doesn't exist.

```
    /* determine my four neighbors */
    if ((rank - px) >= 0) north = rank - px;
    if ((rank + px) < (px*py)) south = rank + px;
    if ((rank - 1) / py == ry) west = rank - 1;
    if ((rank + 1) / py == ry) east = rank + 1;
```

We also have to create the datatypes, considering the appropriate type for each one of them:

```
    /* create north-south datatype */
    MPI_Type_contiguous(bx, MPI_DOUBLE, &typeNS);
    MPI_Type_commit(&typeNS);

    /* create east-west datatype */
    MPI_Type_vector(by, 1, bx+2, MPI_DOUBLE, &typeEW);
    MPI_Type_commit(&typeEW);
```

We chose Type contiguous for the North-South datatype because we are adding an "extra row" on top/bottom of our area, and we chose Type vector for East-West datatype because we are adding an "extra column" on the right/left of our area.

Then we prepare the arguments for the MPI_Alltoallw communication:

```
/* prepare arguments of alltoallw */ // DISPLACEMENTS IN BYTES
recv_counts = (int*) malloc(size*sizeof(int));
send_counts = (int*) malloc(size*sizeof(int));
rdispls = (int*) malloc(size*sizeof(int));
sdispls = (int*) malloc(size*sizeof(int));
types = (MPI_Datatype*) malloc(size*sizeof(MPI_Datatype));

for (i = 0; i < size; i++) {
    if(i == north) {
        send_counts[i] = 1;
        sdispls[i] = ind_f(1, 1, bx) * sizeof(double); // (first col, first row) of block
        rdispls[i] = ind_f(1, 0, bx) * sizeof(double); // north halo (first col, first row-1) of block
        types[i] = typeNS;
    }
    else if(i == south) {
        send_counts[i] = 1;
        sdispls[i] = ind_f(1, by, bx) * sizeof(double); // (first col, last row) of block
        rdispls[i] = ind_f(1, by+1, bx) * sizeof(double); // south halo (first col, last row+1) of block
        types[i] = typeNS;
    }
    else if(i == west) {
        send_counts[i] = 1;
        sdispls[i] = ind_f(1, 1, bx) * sizeof(double); // (first col, first row) of block
        rdispls[i] = ind_f(0, 1, bx) * sizeof(double); // west halo (first col-1, first row) of block
        types[i] = typeEW;
    }
    else if(i == east) {
        send_counts[i] = 1;
        sdispls[i] = ind_f(bx, 1, bx) * sizeof(double); // (last col, first row) of block
        rdispls[i] = ind_f(bx+1, 1, bx) * sizeof(double); // east halo (last col+1, first row) of block
        types[i] = typeEW;
    }
    else {
        send_counts[i] = 0;
        sdispls[i] = 0;
        rdispls[i] = 0;
        types[i] = typeNS; // does not send anything
    }
}
```

After the parameter preparation, we update the aold buffer using the alltoallw communication:

```
/* COMMUNICATION */
MPI_Alltoallw(aold, send_counts, sdispls, types, aold, recv_counts, rdispls, types, MPI_COMM_WORLD);
```

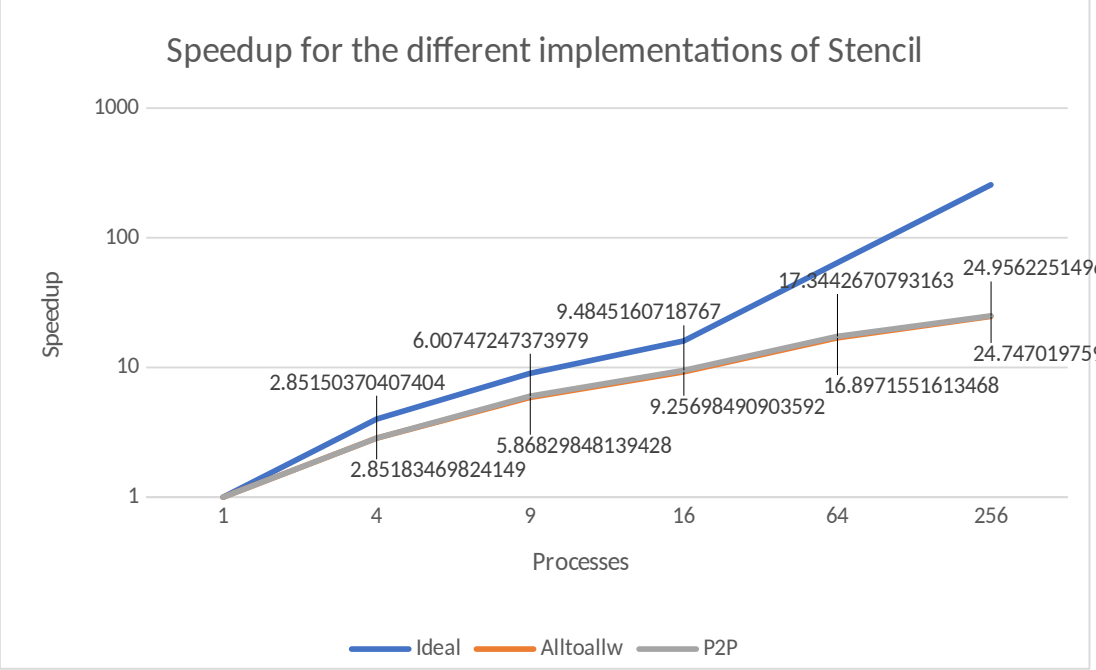
Finally, we free used pointers and reduce the result to obtain the final heat.

```
    /* free working arrays and communication buffers */
    free_bufs(aold, anew);
    free(send_counts);
    free(recv_counts);
    free(rdispls);
    free(sdispls);
    free(types);

    /* get final heat in the system */

    MPI_Reduce(&heat, &rheat, 1, MPI_DOUBLE, MPI_SUM, 0,
MPI_COMM_WORLD);
```

2. **Plot the speedup for the problem using 1, 4, 9, 16, 64, and 256 processes between the alltoallw version and the point-to-point case. Use n=4096, x=10, iter=1000 and px=py=sqrt(nprocs). Compare computation and communication times for both cases.**



We will consider that executions with 1 process are not parallelized to make the calculations.

We can see that the speedup of Alltoallw and Point-to-point are almost equal, so probably there are not many differences in the implementations. We can also see that the difference between the speedup of both implementations and the ideal speedup is considerable and it is more noticeable as we increase the size of the grid.