A1Q2 Report

Name: Viswa Nihar Nukala
UB Name: viswanih
UB Number: 50414392

This code is to implement isort which sorts the elements rank-wise such that elements in i^{th} rank is greater than $(i-1)^{th}$ rank and $x_i = x_j$ then both the elements need to be in the same rank.

Since the numbers are in between range [-(p-1), (p-1)] where p is the number of ranks, we can assume that each rank can have only 2 distinct elements (the first rank will have only one distinct element).

First, we initiate 2 arrays which maintains counts of different elements in one rank which we will be using to send it to other ranks. The first array is to maintain the count of the first element which belong to one rank. The second array is maintaining the count of the second element which belong to one rank.

Then we iterate each array and send it to the respective ranks the counts of different elements it contains.

After send we make sure that the ranks are in sync, just to make sure that all the sends are completed before we start receiving. To differentiate between the elements, we use different tags (I have taken 111 and 222).

We clear Xi to add the counts of elements.

After the ranks are in sync, we start receiving the elements with tag 111 and add then in Xi. Then we do the same for 222.

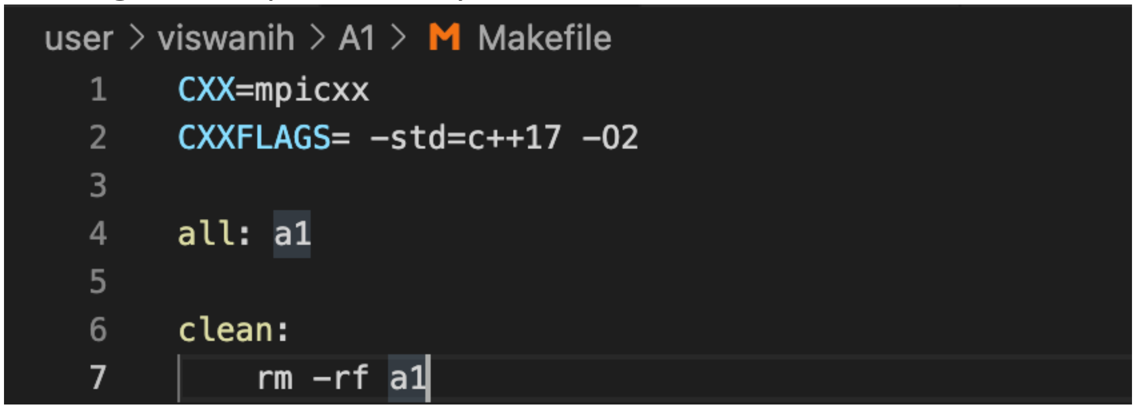
After this, we can be sure that the elements are sorted.

I have also added code to view elements.

Bottleneck:

All the sends and receives need to be executed one after the other and this takes time due to the sync issue. This can be overcome by using MPI_Alltoall, but I have chosen this method to use different MPI tags. I have profiled using both the methods and found this method to be more scalable, debug easy and understandable.

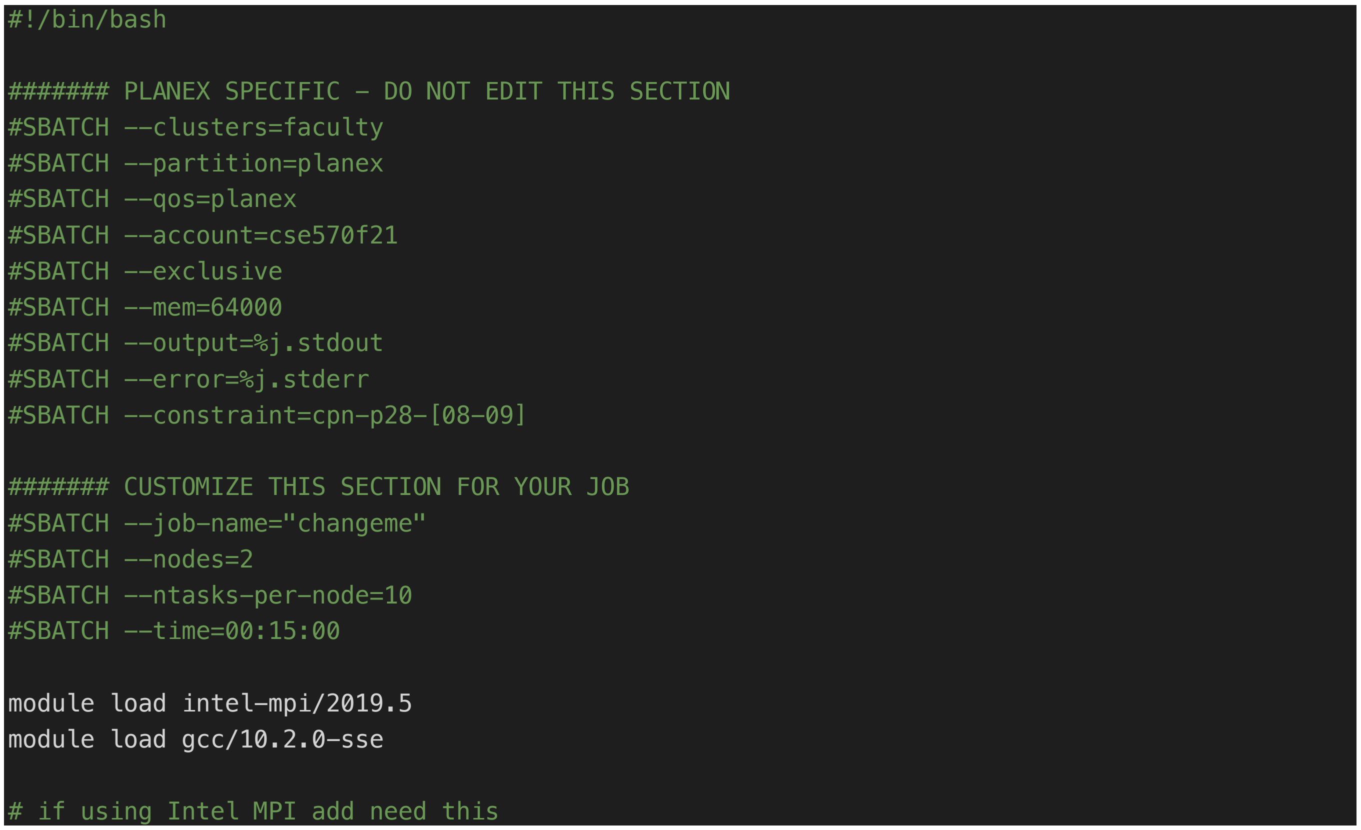
The following is a snapshot of my Makefile:



user > viswanih > A1 > Makefile

```
CXX=mpicxx
CXXFLAGS= -std=c++17 -O2

all: a1

clean:
	rm -rf a1
```

The following is a snapshot of my slurm.sh:



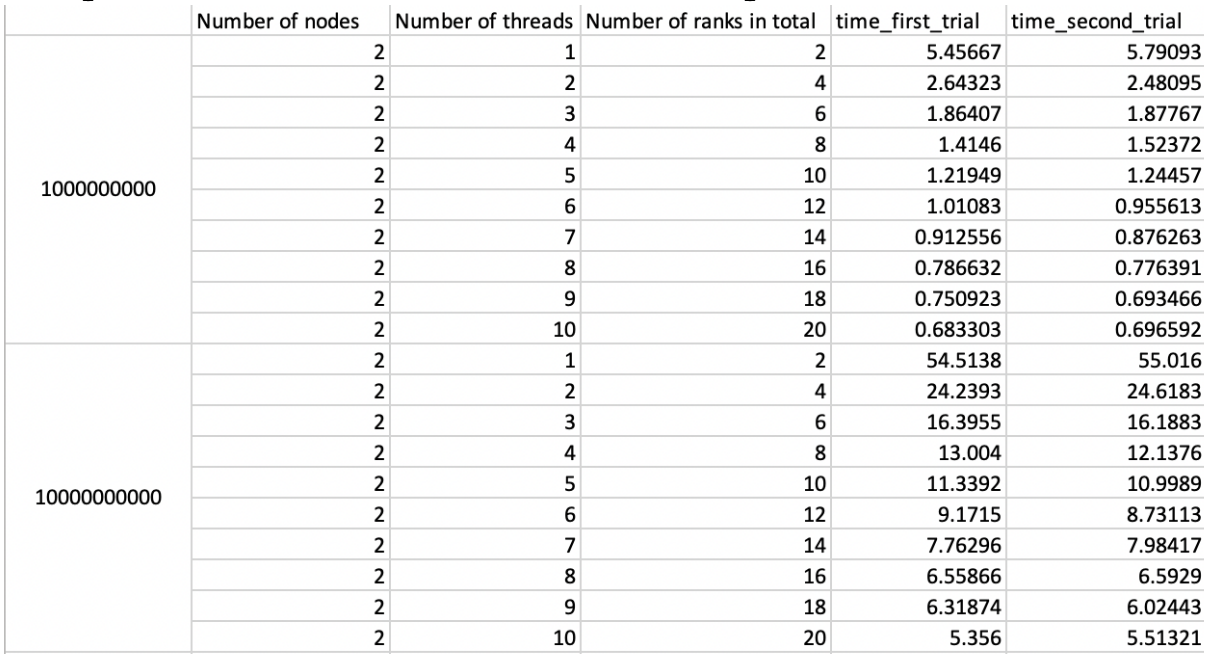
```
#!/bin/bash

####### PLANEX SPECIFIC - DO NOT EDIT THIS SECTION
#SBATCH --clusters=faculty
#SBATCH --partition=planex
#SBATCH --qos=planex
#SBATCH --account=cse570f21
#SBATCH --exclusive
#SBATCH --mem=64000
#SBATCH --output=%j.stdout
#SBATCH --error=%j.stderr
#SBATCH --constraint=cpn-p28-[08-09]

####### CUSTOMIZE THIS SECTION FOR YOUR JOB
#SBATCH --job-name="changeme"
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=10
#SBATCH --time=00:15:00

module load intel-mpi/2019.5
module load gcc/10.2.0-sse

# if using Intel MPI add need this
```

```
export I_MPI_PMI_LIBRARY=/usr/lib64/libpmi.so

make clear

make

srun --mpi=pmi2 ./a1 1000000000
srun --mpi=pmi2 ./a1 10000000000

make clear
```

I have given different ranks ranging from 2 to 20 which run only on 2 nodes. The reason I have chosen all the data size to be 2 nodes is to maintain consistency of data, as the communication overhead when there are 3 nodes and when there are 2 nodes is very different and the speed up graph which we need won't be consistent.

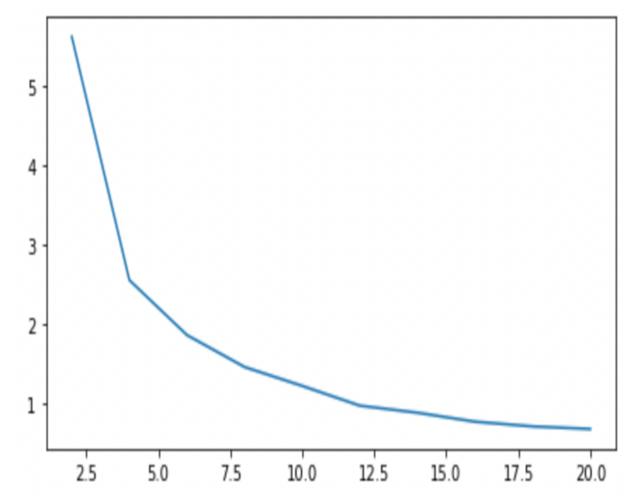
Why Strong Scaling:
I have chosen strong scaling as this gives a clearer idea of how the processors are impacting the speed on how the program is improving for a fixed data size.

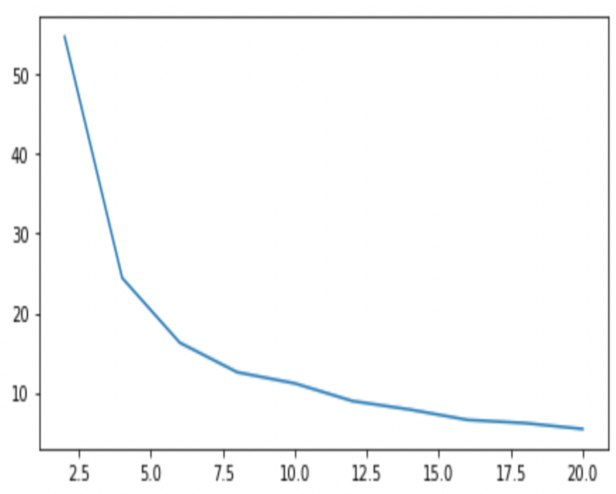
Following are the times I have obtained running different matrix sizes:



	Number of nodes	Number of threads	Number of ranks in total	time_first_trial	time_second_trial
1000000000	2	1	2	5.45667	5.79093
	2	2	4	2.64323	2.48095
	2	3	6	1.86407	1.87767
	2	4	8	1.4146	1.52372
	2	5	10	1.21949	1.24457
	2	6	12	1.01083	0.955613
	2	7	14	0.912556	0.876263
	2	8	16	0.786632	0.776391
	2	9	18	0.750923	0.693466
	2	10	20	0.683303	0.696592
10000000000	2	1	2	54.5138	55.016
	2	2	4	24.2393	24.6183
	2	3	6	16.3955	16.1883
	2	4	8	13.004	12.1376
	2	5	10	11.3392	10.9989
	2	6	12	9.1715	8.73113
	2	7	14	7.76296	7.98417
	2	8	16	6.55866	6.5929
	2	9	18	6.31874	6.02443
	2	10	20	5.356	5.51321

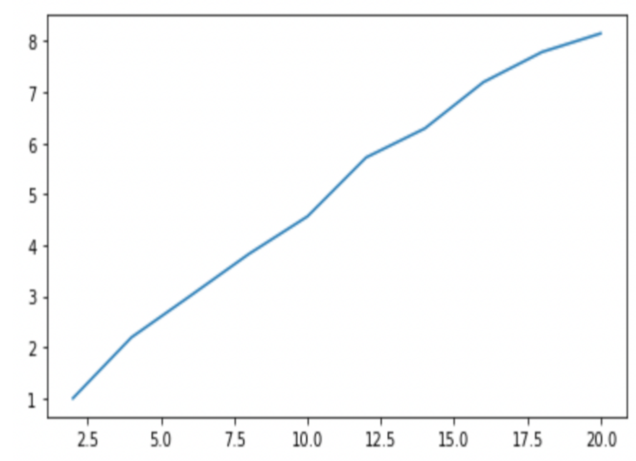
The following is the time graph obtained by running the above slurm.sh:

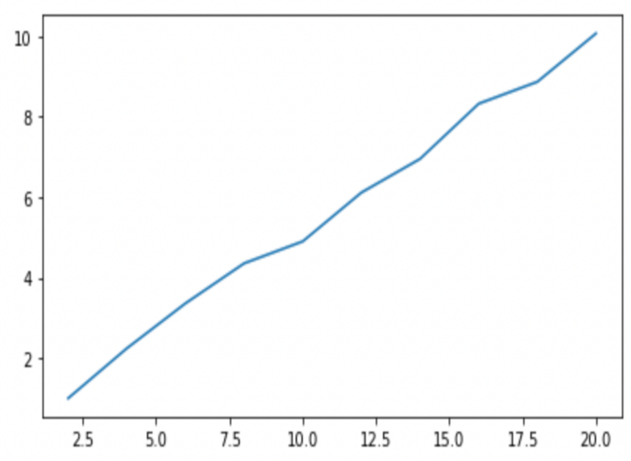




We can infer that as the number of processors increase, the time taken for the program to complete decreases but after a number of processors it is tending to become linear.

The following is the speed up graph obtained by running the above slurm.sh:





The graph on the left is the speed up for matrix of size 1000000000 and the graph on the right is for 10000000000.

Conclusion:

We can infer from the graph that as number of processors increase, the speed up decreases and after some number of processors the speed up remains constant. The algorithm is scalable at least till 20 processors. But as we increase the number of processors, the speed up may become linear as the number of send and receives increase and the threads need to sync up.