Syed Alle Mustafa

ID: A20519664

CS451
Assignment 3:

Algorithm's Correctness:

To prove my algorithms correctness, I have solved a 3x3 matrix with an online tool and then with my MPI program and results where exactly the same, the number in matrix is generated randomly form the program. The proof is as follow:

The Output from online tool:

Your matrix

	x_1	x_2	x_3	b
1	33041.722656	35557.308594	3865.022461	5679.871582
2	29967.964844	63327.148438	41622.5	44383.980469
3	52419.402344	2065.175293	17645.544922	30256.373047

Solution set:

x_1 = 0.196075149751344993
x_2 = -0.1474081040894252726
x_3 = 1.1494490278840734453

The Solution from my program:

```
smustafa2@fusion2:~$ mpirun -np 2 ./a.out 3
Input Vector:
        33041.722656 35557.308594 3865.022461 5679.871582
        29967.964844 63327.148438 41622.500000 44383.980469
        52419.402344 2065.175293 17645.544922 30256.373047

elapsed time: 0.000063 seconds
Processed Matrix:
        33041.722656 35557.308594 3865.022461 5679.871582
        0.000000 31077.615234 38117.027344 39232.488281
        0.000000 0.000000 78168.687500 89850.921875

Result: [0.196075 -0.147408 1.149449 ]
```

The output of my program also shows, that the gauss elimination works perfectly using mpi. I have tested few more matrices using the random seed, here's the proof for 4x4 matrix:

Your matrix

	X_1	X_2	X_3	X_4	b
1	54878.164062	52546.699219	33796.742188	58342.847656	14504.31543
2	49066.328125	65300.476562	38397.851562	56182.582031	41522.574219
3	11814.182617	10527.150391	4483.666992	7128.38623	45124.683594
4	38764.273438	60989.492188	10117.024414	44432.359375	19815.974609

Solution set:

$x_1 = 6.1742229894408295631$
$x_2 = 2.7126135190826476938$
$x_3 = 1.882805238348030046$
$x_4 = -9.0927559755586025158$

Solution from my program:

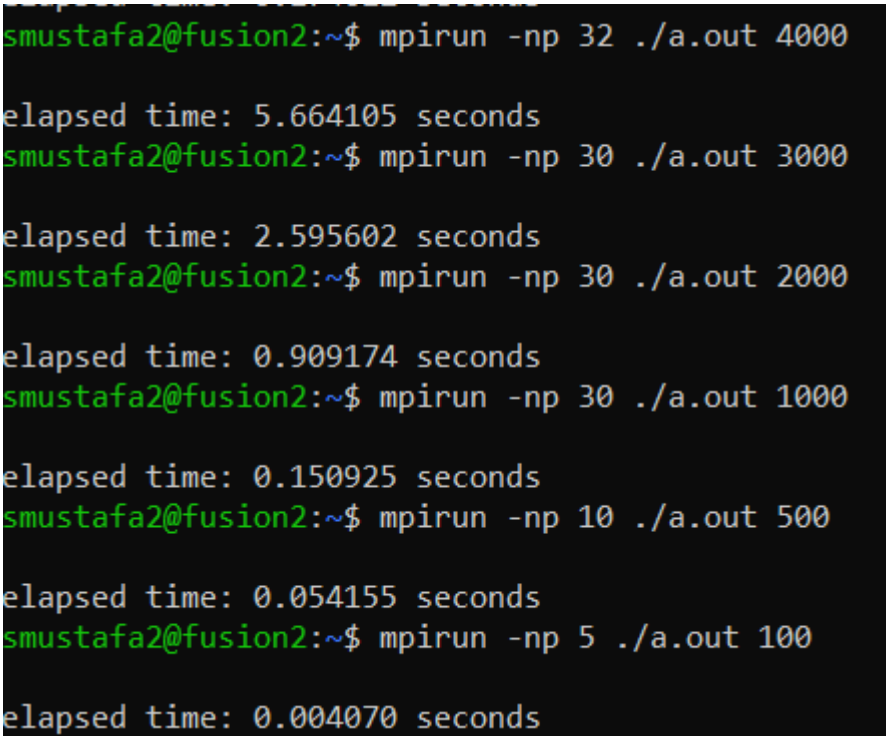
```
smustafa2@fusion2:~$ mpirun -np 3 ./a.out 4
Input Vector:
        54878.164062    52546.699219    33796.742188    58342.847656    14504.315430
        49066.328125    65300.476562    38397.851562    56182.582031    41522.574219
        11814.182617    10527.150391    4483.666992     7128.386230     45124.683594
        38764.273438    60989.492188    10117.024414    44432.359375    19815.974609

elapsed time: 0.000072 seconds
Processed Matrix:
        54878.164062    52546.699219    33796.742188    58342.847656    14504.315430
        0.000000        18318.699219    8180.332031     4018.496094     28554.328125
        0.000000        0.000061        -2441.504639    -5259.447266    43225.992188
        0.000000        0.000000        -0.001953       50580.984375    -459920.562500

Result: [6.174223 2.712614 1.882806 -9.092756 ]
smustafa2@fusion2:~$
```

Algorithm's scaling:

The given solution can solve more than 10000x10000 matrix Gaussian elimination, the preferred core size for such calculations in the given machine was 30. Following are the results:

```
smustafa2@fusion2:~$ mpirun -np 30 ./a.out 5000

elapsed time: 12.261236 seconds
smustafa2@fusion2:~$ mpirun -np 30 ./a.out 5500

elapsed time: 15.904650 seconds
smustafa2@fusion2:~$ mpirun -np 30 ./a.out 6000

elapsed time: 20.524132 seconds
smustafa2@fusion2:~$ mpirun -np 30 ./a.out 6500

elapsed time: 26.291223 seconds
smustafa2@fusion2:~$ mpirun -np 30 ./a.out 7000

elapsed time: 32.231176 seconds
smustafa2@fusion2:~$ mpirun -np 30 ./a.out 7500

elapsed time: 39.710416 seconds
smustafa2@fusion2:~$ mpirun -np 30 ./a.out 10000

elapsed time: 93.313788 seconds
```

Time taken by matrices with dimension equal to and lesser than 4000x4000, are found to be less than 6 seconds, we scale the core size with it as well:



If compared to the following result of serial program, we have achieved excellent performance:

```
smustafa2@fusion2:~/CS451-HW2$ ./serial 1000

Matrix dimension N = 1000.

Initializing...

Starting clock.
Computing Serially.
Stopped clock.

Elapsed time = 1117.03 ms.
(CPU times are accurate to the nearest 0.001 ms)
My total CPU time for parent = 0.111 ms.
My system CPU time for parent = 0 ms.
My total CPU time for child processes = 0 ms.
------------------------------------------
smustafa2@fusion2:~/CS451-HW2$ ./serial 2000

Matrix dimension N = 2000.

Initializing...

Starting clock.
Computing Serially.
Stopped clock.

Elapsed time = 7457.06 ms.
(CPU times are accurate to the nearest 0.001 ms)
My total CPU time for parent = 0.746 ms.
My system CPU time for parent = 0 ms.
My total CPU time for child processes = 0 ms.
------------------------------------------
```

The scale can also be increased to a very large value when working with multiple processes, depending upon the cores available.

Algorithm Running Time for Variable Processors:

A matrix of 2400x2400 dimension is used for comparing running time.

For 1 Processor:

```
smustafa2@fusion2:~$ mpirun -np 1 ./a.out 2400

elapsed time: 14.369978 seconds
```

For 2 Processors:

```
smustafa2@fusion2:~$ mpirun -np 2 ./a.out 2400

elapsed time: 7.340989 seconds
```

For 4 Processors:

```
smustafa2@fusion2:~$ mpirun -np 4 ./a.out 2400

elapsed time: 3.967948 seconds
```

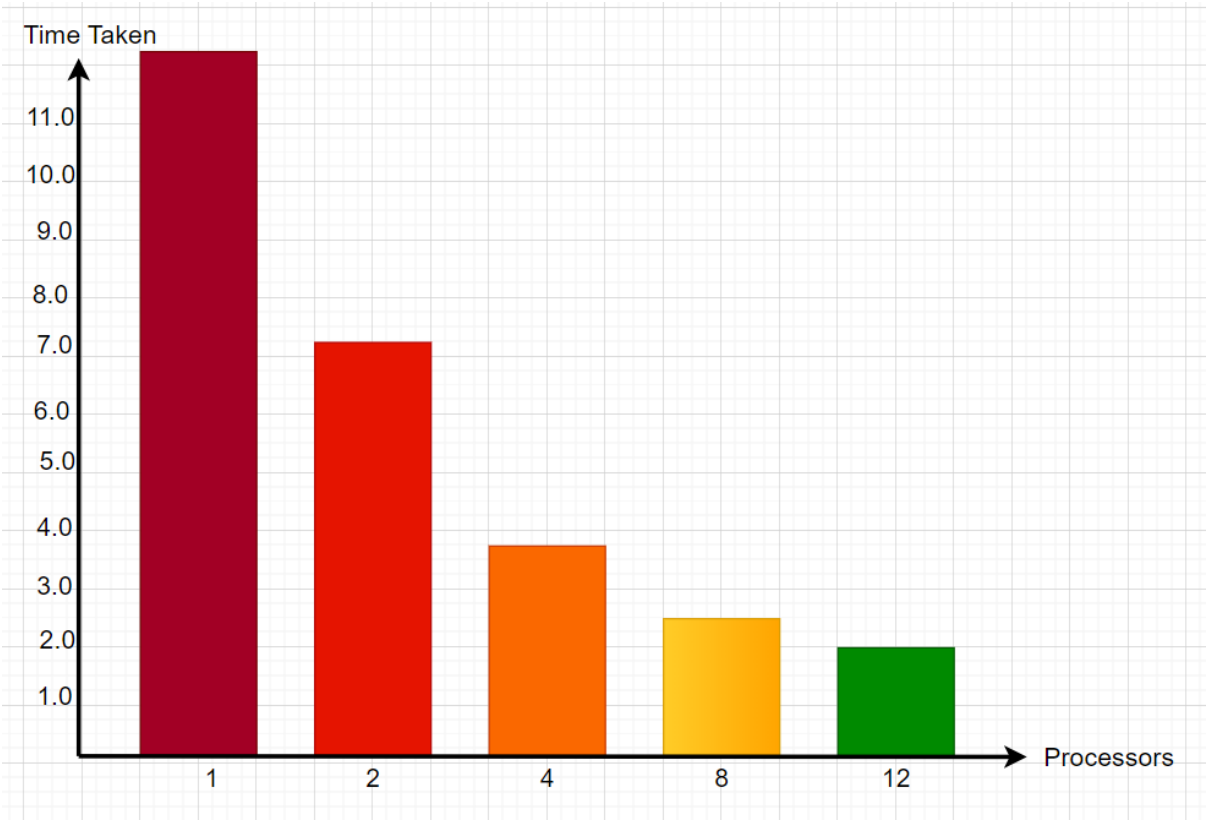
For 8 Processors:

```
smustafa2@fusion2:~$ mpirun -np 8 ./a.out 2400

elapsed time: 2.434218 seconds
```

For 12 Processors:

```
smustafa2@fusion2:~$ mpirun -np 12 ./a.out 2400

elapsed time: 2.073977 seconds
```

Performance Chart:



Implementation Details:

The parameters of seed and matrix dimensions are parsed just the way they were being parsed in the serial program, then we initialize the mpi by using the parameters of main. The size and current id of the process is saved in an in variable using mpi predefined APIs. The column size initialized to the row size + 1, one index is added which is taken as the B vector which will be calculated as vector X. After allocating the memory and initializing the matrix (Note that for ease of computation and passing the rows, a 1D array of float is used to store the 2D matrix) to random values using the first processor only, the rows are disturbed through processor 0 to the relevant processor depending upon the row index. The sending and receiving of rows are not async, thus synchronized in the loop. Then we start the gaussian elimination on each processor. We iterate from 0 to the Nth rows. If the row belongs to the current process, we share it with other processes (mpi broadcast) and find the multiplicator and apply elimination on each row and their columns respectively. The output matrix of this operation is perfect for the back substitution. Now we apply a mpi barrier, so that all processors gather before collecting data on the first process. This is done again with mpi send and mpi receive. Once all the data is received on processor 0, we perform back substitution and stop the timer which was started before data distribution step.