CS546 “Parallel and Distributed Processing”

Homework 4
First name: Guillermo
Last Name: de la Puente
CWID: A20314328
Due date: 4/182014

Also included:
- 1/get_data.c
- 1/run_get_data.bash
- 2/gauss.c
- 2/run_gauss.bash

Aclaration about qsub and working directory

When submitting a job using the qsub command, the path where the executable file is searched is the home directory. This can be easily changed using the command cd. This is the code of my run_get_data.bash:

```
# Modifying working directory
# I did this to adapt the script to my custom working directory structure

cd parallel-processing/hw4/1
echo "Working directory is:"
pwd

# Run job
mpirun -npernode 8 ./get_data
```

Problem 1

The program get_data.c has been modified so the last process is the one collecting the data and printing the output. The output is:

```
Process number 8 of 8 says:
With n = 100 trapezoids, our estimate
of the integral from 0.000000 to 1.000000 = 0.294928
```

Problem 2

The algorithm used to implement the Gaussian Elimination in parallel is the same that I used in the previous assignments.

For every iteration of the main loop, the workload is divided in rows and distributed to the processes. There aren't any dependencies at this moment, and this is why processes can operate in different data. For every iteration, the workload per process decreases. The rows in every iteration are divided between the number of processes, trying to assign the same amount of work to all of them.

While in shared memory programming models, the speedup achieved is very high, using MPI to parallelize gaussian elimination isn't very efficient. The problem is that for every iteration, the process #0 must distribute data to all the rest, and then gather the results.

Different tests using Send, Recv, Isend, Bcast, Scatterv and Gatherv operations have been performed, concluding that the fastest way is to distribute the work using Scatterv and then collect it using Isend and Recv. The program has other alternatives commented to show how they were implemented.

MPI works because the number of rows that processes handle in every iteration can be calculated both by process #0 and by the others.

Even though many different alternatives have been tested, **I haven't been able to achieve an improvement over the sequential algorithm**. The problem is the communication, and I'm sure my code can be improved more although I can't think about how. These are the results that have been obtained running the code in the Jarvis cluster:

N	Sequential	MPI with 8 CPUs
50	0.5 ms	6.2 ms
100	2.5 ms	14.3 ms
200	12 ms	37 ms
400	80 ms	180 ms
800	630 ms	1,110 ms
1600	5,200 ms	9,500 ms
2400	17 s	28 s
4800	136 s	211 s

And this is the comparison of times for N=2000 modifying the number of CPUs. It can be seen that due to the communication costs, the time increases when we increment the number of processes.

CPUs	Time
1 (seq)	9.8 s
2	14.6 s
4	16.2 s
8	18.1 s
16	30.4 s