

Lab 2 Report

Weiqiang Li

<https://github.com/bambrow>

Compile & Run

To compile the program on CIMS machines, first simply do the following:

```
module load mpi/openmpi-x86_64
```

Then compile it with:

```
mpicc -std=c99 -o gs gs.c -lm
```

Finally, to run the program, simply type:

```
mpirun -n x ./gs <filename>
```

Where x is the number of processes and filename is the path of input file.

(Please note in the above command, `./gs` is needed instead of `gs` because CIMS machines already contain a program called `gs`. Therefore, `./gs` guarantees that we're using the right program that we just compiled.)

Results & Analysis

1. Environment

For the CIMS servers, the server used in this lab is **crunchy1**. Checking the file **proc/cpuinfo**, the important specifications are listed below:

- CPU: Four AMD Opteron 6272 (2.1GHz) (64 cores)
- Memory: 256 GB
- OS: CentOS 7

The program will be tested on problem sizes 10, 100, 1000 and 10000, with error rate 0.001, and number of processes 1, 2, 10, 20 and 40. All tests were carried out for 5 times and the average time costs are included in the following tables.

2. Table 1 - Running Time

Problem size		1 process	2 processes	10 processes	20 processes	40 processes
10 unknowns	real time	0.3972	0.3948	0.5134	N/A	N/A
	read/write	0.0016778	0.0029732	0.0031042	N/A	N/A
	calculation	0.0000598	0.0002634	0.0018746	N/A	N/A
100 unknowns	real time	0.378	0.394	0.5302	0.6832	1.1416
	read/write	0.0048108	0.0076846	0.0053374	0.0115702	0.0083598
	calculation	0.0019962	0.0014036	0.0057704	0.0054588	0.0085376
1000 unknowns	real time	0.8366	0.7722	0.8562	0.9624	1.4664
	read/write	0.2972922	0.2935306	0.3327338	0.299291	0.374094
	calculation	0.1622642	0.0843664	0.0237252	0.0190436	0.0183606
10000 unknowns	real time	51.9956	40.3722	32.1686	31.2662	33.2936
	read/write	29.576131	28.7718954	29.1394332	28.896862	31.1426144
	calculation	22.016819	11.158429	2.5098262	1.700256	1.0172624

3. Table 2 - Speed-up

Problem size		1 process	2 processes	10 processes	20 processes	40 processes
10 unknowns	real time	1	1.006	0.774	N/A	N/A
	read/write	1	0.564	0.540	N/A	N/A


	calculation	1	0.227	0.032	N/A	N/A
100 unknowns	real time	1	0.959	0.713	0.553	0.331
	read/write	1	0.626	0.901	0.416	0.575
	calculation	1	1.422	0.346	0.366	0.234
1000 unknowns	real time	1	1.083	0.977	0.869	0.571
	read/write	1	1.013	0.893	0.993	0.795
	calculation	1	1.923	6.839	8.521	8.838
10000 unknowns	real time	1	1.288	1.616	1.663	1.562
	read/write	1	1.028	1.015	1.024	0.950
	calculation	1	1.973	8.772	12.949	21.643

4. When there is no speed-up and why?

As we can see from the tables, the speed-up highly depends on the problem size. For better clarifications, the running time and speed-up tables are extended: not only the real time is provided, the read/write time (time for reading input file and writing output file) and the calculation time (time cost purely on calculations) are also provided for each problem size and each number of process. This can give us a better view of the performance under different number of processes.

When there is no speed-up, it is because of one of the following to reasons, or both: a) the problem size is too small; and b) the read/write time is too large compared to the computation time. For example, we did not get any speed-up on problem size of 10, and could not get any speed-up on problem size of 100, except for the calculation time of 2 processes; for problem size of 1000, we could only get speed-up on 2 processes, but for calculation time the speed-up can be seen on all number of processes; finally, for problem size of 10000, we could get speed-up for all number of processes.

It has been stated that the reasons are mainly two. For the first reason, it is obvious that if the problem size is very small, having too many processes will increase overhead, because the cost of creating processes and communications between processes will become more significant. For



the second reason, it is because that the read/write disk access is much slower than CPU calculations. The read/write part in the program is serial (only finished by process 0), therefore the time cost is almost fixed here and cannot benefit from parallelism. If this part takes a large amount of time, our parallelized program will not behave much better given more processes. On the contrary, if we check only the calculation time (which is the only parallelized part), we can see that the speed-up is much more significant for almost all cases, if not influenced by the first reason (overhead of parallelism).

5. When there is speed-up and why?

Based on our discussion in 4), the conclusion here is clearer. When there is speed-up, it is because of one of the following reasons, or both: a) the problem size is large enough; and b) the read/write time does not occupy a large proportion of the whole running time.

As we can see from the table, for the real time, we can only get speed-up on problem size of 10000; however, for only the calculation part, we can get speed-up starting from problem size of 100, if not limited by the overhead given too much processes. As stated before, the best number of processes highly depends on the problem size. For small problems (10 and 100 unknowns), 1 process or 2 processes are already enough, and given more processes will make the performance even worse. For medium problems (1000 unknowns), calculation part can be greatly benefited from parallelism, but because of the serial part is too large, there is no performance gain in total. For large problems (10000 unknowns), we could finally see the performance enhancement given more processes, and the calculation part is greatly benefited from parallelism. Therefore, given the problem size is big enough (and the serial part is small enough relatively), we can expect to see better performance for more number of processes, only if the overhead of parallelism does not hurt the performance too much.

