Lab2 Report

Dayou Du(dd2645@nyu.edu)

1. General Description and Compilation Guide

    a. Code Structure
    gs.c contains reading input, distributing input, communications and write the result.
    gssol.c contains the updating procedure. i.e. calculate next iteration of x.

    b. Compilation Guide
    Simply,
    ```
    module load mpi/openmpi-x86_64
    ```
    Then, type
    ```
    make
    ```
    Should be fine on crunchy machines.

2. Algorithm Description
The whole program contains several procedures:

    a. read input from file (Only rank 0) ("read input" in the result table)
    Simply, read the input file. From result we can see that actually most of time this part take most of the time. i.e. a "disk bound" application :P
    Here we improved this part by reading line by line and use own parser to parse the input, instead of just fscanf each number (as given in the original gs.c file). This accelerate this part 3~5 times, but since the disk speed is merely about 100MB/s, the whole program will be ultimately bound by disk reading.

    b. distribute input ("dispatch input" in the result table)
    After rank 0 read the input file, we calculate the chunk size (how many unknowns a process response for) and distribute the X array, B array and corresponding A array to each process, using MPI_Bcast and MPI_Scatterv. Note that here we do not need to copy the whole A to each process, only the parts that corresponds to the chunk size is copied.

    c. update result ("calculation" in the result table)
    Just as introduced in the instruction, each process update the unknowns that it response for, and calculate whether within the precision limit, then exchange the updated X array and judge if we have the next iteration of updating.

    d. write out

Simply write the result.

3. Experimental Result

Note: If you are testing 100000 inputs, due to the precision limit of 32-bit floating point and the truncation when writing to the result file, several points(less than 5) will have different truncation when writing to the output file. E.g:

0.00000249999999 => 0.000002

0.00000250000000 => 0.000003

This will ONLY happen when we have too much inputs. Because at this time the accumulated error is large enough, also each result is small enough that the 6-decimal-digit result format cannot hold our result precisely.

a. Environment

All the tests are done on crunchy5, with mpi/openmpi-x86_64, optimization level -O2.
For 10,100,1000,10000 unknowns, the result is the average of 20 tests.
For 100000 unknowns, the result is the average of 2 tests (Well, reading 30GB of input each time is rrrrealy slow and I do not want to occupy the server too long)

b. Table1 : Run Time for EACH PARTS of the program

Time Unit: second.

"read input", "dispatch input", "calculation" are three steps described above in section2 – Algorithm Description. The "real" is the performance of the whole program by Linux time command.

#Unknown	#Processes	1	2	10	20	40
10	read input	0.000252	0.000262	0.000321		
	dispatch input	0.00002	0.000068	0.000207		
	calculation	0.000018	0.000166	0.000805		
	real(by time command)	0.405	0.417	0.529		
100	read input	0.001391	0.001435	0.001497	0.001566	0.001906
	dispatch input	0.000048	0.000178	0.000779	0.000913	0.001196
	calculation	0.000539	0.000419	0.001357	0.001954	0.003019
	real(by time command)	0.402	0.447	0.541	0.676	1.123
1000	read input	0.109623	0.110095	0.107325	0.117198	0.111083
	dispatch input	0.002686	0.003861	0.005749	0.006816	0.009017
	calculation	0.044014	0.022199	0.005944	0.004636	0.004685
	real(by time command)	0.563	0.555	0.658	0.805	1.245
10000	read input	10.49857	10.27655	10.28387	10.45463	11.26606
	dispatch input	0.17966	0.211484	0.267239	0.273925	0.32261
	calculation	5.731402	2.892514	0.583855	0.297476	0.173093
	real(by time command)	16.826	13.783	11.777	11.583	13.263
100000	read input	1060.477	1090.899	1046.435	1089.379	1095.822
	dispatch input	61.7399	56.51331	41.29682	39.54822	39.36894
	calculation	899.0416	529.0964	90.51458	45.61862	29.28738
	real(by time command)	2062.886	1687.391	1184.416	1187.104	1170.415

c. Speed up for each part (exclude reading inputs)

#Unknown	#Processes	1	2	10	20	40
10	dispatch input	1	0.294118	0.096618		
	calculation	1	0.108434	0.02236		
	real(by time command)	1	0.971223	0.765595		
100	dispatch input	1	0.269663	0.061617	0.052574	0.040134
	calculation	1	1.286396	0.3972	0.275844	0.178536
	real(by time command)	1	0.899329	0.743068	0.594675	0.35797
1000	dispatch input	1	0.695675	0.467212	0.394073	0.297882
	calculation	1	1.982702	7.404778	9.49396	9.394664
	real(by time command)	1	1.014414	0.855623	0.699379	0.452209
10000	dispatch input	1	0.849521	0.672282	0.655873	0.556895
	calculation	1	1.98146	9.816482	19.26677	33.11169
	real(by time command)	1	1.220779	1.428717	1.452646	1.268642
100000	dispatch input	1	1.092484	1.495028	1.56113	1.568289
	calculation	1	1.699202	9.932562	19.70778	30.69723
	real(by time command)	1	1.22253	1.74169	1.737747	1.762525

4. Result Analytic

a. General Findings

i. (From 3.b Table 1) In most of the cases, reading the input from file take most of the run time. This is in expected since the disk speed is only about 100MB/s, and CIMS system use a network file system - the directories are mount upon login, thus the input file has to be transfer through the network when they are being read (though transparent to users) The input file for 100000 unknowns is about 30GB, thus it's normal to take 1000s to read it.

ii. (From 3.b Table 1) The percentage/weight of calculation goes up with number of unknowns grows. This is because the algorithm is O(kn^2), where n is #unknowns, k is the number of iterations and grows with n (we can rewrite it as O(f(n)n^2)). While reading input is O(n^2), thus it grows slower than the calculation part.

b. When I cannot get speed up and why?

i. For only calculation part

When the work per process is small. Like all cases for 10 and 100 unknowns.

This is because the calculation part contains the updating procedure and the information exchanging between each iterations. The later part grows with the number processes grows because we have to let all processes know the updated X array and whether all Xs are within error rate. Thus if the work per process is small, the benefit of parallelizing the updating procedure will not worth the cost of communications between iterations.

ii. For the overall performance (“real time” by Linux time command)

When the work per process is small, or the percentage/weight of calculation part is not large enough.

The overall performance is influenced by more aspects.

a) The only part that we can gain from parallel processing is the calculation part, thus if the calculation part itself cannot get speed up, the whole program will certainly not benefit from parallel processing.
b) Reading input file could be fluctuate. Part of reason is the disk reading time itself is not that stable –other users are also using the server, and the disk scheduling is totally depends on the operating system. The reason could also be the CIMS file system is ultimately a network file system. The user’s directory is mounted upon login, and the files are actually read from file servers so that we can see a “uniform” home directory on each server. So the time of reading the files is also influenced by the network transfer.
c) The time of dispatching inputs could be fluctuate. It’s not only about transfer the data (though transfer indeed take a main role – thus why we call this step dispatching), the system also take time on allocating the memory for each process. For 100000 unknowns actually we are allocating a total amount of 80GB heap memory (accumulated for all processes, because they are essentially on the same machine), and the operating system has to kick of some buff/cache memory (The shared, buff/cache entry for Linux free command)

Because of reason b) and c), even if we can gain speed up from calculation part, the whole program may not gain or even become slower when the percentage of calculation part is small – Amdahl’s law. Like 10, 20, 40 processes on 1000 unknowns.

c. When can I get speed up and why?
When the work preprocess is large enough, we can see speed up from calculation part. When we can gain performance from calculation part AND the calculation part take a big part of the whole running time, we can gain performance for the whole program. Like 2 processes for 1000 unknowns, and all cases for 10000 and 100000 unknown. (Besides, we can see that the overall performance of 40 processes on 10000 unknowns is not as good as 20 processes on 10000 unknowns).

The reasons has already be stated above in b. Just a brief summarize:

i. For the calculation part, the work per process has to be large enough, so that the performance gain could cover the communication overhead between each iterations.
ii. For the overall program, a) The only benefit we can get from parallelizing is the calculation part thus we have to gain performance on it. b) The calculation part has to “weight” enough to bring speed up to the whole program.