Lab2 Report

Ke Xu 604761427

The experimental result and discussion. Please report and quantify the impact of all experiments you have done such as block size and number of processors. Please express your performance in GFlop/s.

My final results:95 GFlop/s. The techniques I utilized are loop tiling, loop change (just as done in lab1) and the block size is 64.
In my experiments, when the input size is less than 2048*2048 and the number of processors is 16, we can get about 80-90 GFlop/s without loop tiling, while the results become down to 30 GFlop/s when the size is 4096*4096 without loop tiling.
When adding loop tiling and the block size is 64. I get about 95 GFlop/s when the input size is 4096*4096 and about 80-90 GFlop/s when the size is less than 2048*2048.
The above results are all with loop change. The results without loop change while with loop tiling are 75 GFlop/s under 4096*4096 and 65 GFlop/s under 2048*2048.The results without both two techniques above are 27 GFlop/s under 4096*4096 and 59 GFlop/s under 2048*2048.
It can be obviously seen that when the input size is large, the loop tiling can be determinative.
The above results are based on 16 processors.
The results under 4096*4096 input size based on different numbers of processors are as following: (With loop tiling and loop change)
np=8: 100.52 GFlop/s
np=4: 57.65 GFlop/s
np=32: 82.27 GFlop/s

Please briey explain how you partition the data and the computation among the processors. Also, briey explain how the communication among processors is done.

For matrix A, I just partition it based on row. Assume we have p processors and the input size is N*N and every processor gets (N/p)*N-size block of A (Row strip). And for matrix B, each one gets N*(N/p)-size block of B (Col strip).
For the computation and communication, I just firstly scatter the A and B to p processors and do multiply and addition for the “row strip” and “col strip”. Then, I move the B “col strips” among processors for p times, every time each one send the its current “col strip” to the left neighbor and receive the strip from its right neighbor. Finally, each processor gets the result of one (N/p)*N-size block of C (Row strip) and do gather to gather all the results and get the whole C matrix.
I tried to use Cannon Algorithm but there are errors of floating computing result from partition k among processors.

Please provide the runtime using blocking (MPI Send, MPI Recv), buffered blocking (MPI Bsend, MPI BRecv),and non-blocking (MPI Isend, MPI Irecv) communication. Which type of communication gives best result? Please explain why.

For 4096*4096 inputs and 16 processors.

Blocking: 3.78 seconds and 79.3 GFlop/s
non-blocking: 3.3 seconds and 95 GFlop/s
buffered blocking: 3.75 seconds and 81.2 GFlop/s

Non-blocking is fastest because it returns immediately without waiting for the sending or receiving to complete. But I have to add MPI_wait so that I can use the received data correctly or change the sent data after it has been correctly sent. Before I implement MPI_wait I can do some other computings.

Please report the scalability of your algorithm using mpirun np 4, 8, 16, 32. Do you get linear speedup? If not, why?

np=8: 100.52 GFlop/s
np=4: 57.65 GFlop/s
np=32: 82.27 GFlop/s
np=16: 95 GFlop/s

Definitely not. Especially when the np is relatively large. Because when the np becomes larger, we have more and more parallel overhead including communication. Although the parallel computing time can reduce, the overhead will prevent the speedup from increasing linearly.

How is this MPI implementation when comparing to your OpenMP implementation in lab 1 in terms of the throughput and the programming effort?

For the final performance in GFlop/s, it is better than that of OpenMP. For the lab2, the result is about 95 GFlop/s, and it is about 55 GFlop/s for lab1. But the time I spent is much more than lab1 because I tried some different partitioning algorithms to improve the performance and try different MPI APIs.