Project 1: Parallel Programming with MPI

Yunfei Liu
School of Computer Science, Shanghai Jiao Tong University, Shanghai, Minhang, 800 Dongchuan RD.
liuyunfei@sjtu.edu.cn

***Abstract*—This is the first project report of final assignments of Course Parallel Computing and Parallel Programming Autumn, 2020.**

I. MPI_AllGather

A. Method

Given a set of elements distributed across all processes, MPI Allgather will gather all of the elements to all the processes. Note that the number of elelments in each process should be same.

In this task, we use Use MPI_Send and MPI_Recv to implement the MPI_ALLGATHER function, and compare the performance of our implementation and the original MPI implementation. The details are as follows.

- First, allocate a big arrary for each process to store the data from other processes.
- Second, each process sends its data to other processes with the help of MPI_Send.
- Third, each process receives data from other processes with the help of MPI_Recv, and places the data into the corresponding position of the big array.

B. Experiments

We implement this algorithm with C and MPI. The MPI ALLGATHER experiments are conducted on HUAWEI Cloud with different number of hosts and processes. We set the total number of elements as 4000. Each experiment is repeated for 100 times. And Table I shows the average results.

TABLE I
TIME COST COMPARISION FOR MPI_ALLGATHER.

Hosts	Procs	Average Time (s)	
		my_allgather	MPI_Allgather
1	4	0.032024	0.021326
1	8	0.064938	0.060837
1	16	0.161980	0.178068
1	32	2.222867.	0.485575
2	4	0.013461	0.009947
2	8	0.032132	0.034218
2	16	0.220833	0.083750
2	32	1.121776	0.235945
4	4	0.017813	0.010518
4	8	0.086170	0.052541
4	16	0.443156	0.085482
4	32	1.673738	0.117404

As Table I shows, we increase the number of hosts or procs, the time cost will also increase, which is because the times of communication between processes increases. Compared to the MPI_Allgather, the function we implement is more inefficient. We give an example to explain that. Suppose Proc-1 sends its rank to other processes, the allgather function we implement will let Proc-1 send the value to other processes one by one. But in the MPI_Allgather, once Proc-1 has send the value to Proc-2, Proc-2 will also send this value to other processes except Proc-1 and Proc-2, which is more efficient.

II. Gemm

A. Method

Cannon algorithm for matrix multiplication. we parallelize the program in the tile unit using cannon algorithm. Suppose we have two $n \times n$ square matrices A and B, we need to calculate $C = A \times B$. And there are p processes and p is a square number, and n can be divided by $\sqrt{p}$ totally. The details of cannon algorithm are as follows.

- Step 1. Master process partitions A and B into p blocks, and divides them into each process. Process p(i,j) has two $n/\sqrt{p} \times n/\sqrt{p}$ matrices A(i, j) and B(i, j).
- Step 2. Shift all submatrices A(i , j) to the left (with wraparound) by i steps and all submatrices B(i , j) up (with wraparound) by j steps.
- Step 3. Perform local block multiplication in each process.
- Step 4. Each block of A moves one step left and each block of B moves one step up (again with wraparound).
- Step 5. Perform next block multiplication, add to partial result, repeat until all blocks have been multiplied.
- Step 6. Master process collects results from each processes.

Convolution and Pooling. Given a $n \times n$ matrix A and a 4×4 kernel B, we need to achieve the convolution and pooling operations in parallel, which is $C = A * B$. We choose to implement sum pooling, which is the same as convolution operation, only that sum pooling is to use the kerenl whose elements are all 1 to do convolution. We set stride as 1. And we use padding operation to make sure the size of matrix C is the same as matrix A.

In this task, we cannot partition matrix A, so we just calculate the staring position and ending position of each convolution operation on matrix A, and parition the position to each process. Thus, each process can do convolution independently. Details are as follows. In this way, each process should store matrix A and kernel B, which is a waste of space.

- First, each process get the whole matrix A and kernel B.

- Second, each process calculate the starting position of rows of matrix A based on its rank id.
- Each process do convolution from starting position to ending position.
- Master process collects the results.

B. Experiments

We implement this algorithm with C and MPI. The experiments are conducted on HUAWEI Cloud with different number of hosts and processes. We set the matrix dimension n as 1,024. Each experiment is repeated for 5 times. Table II, Table III, Table IV show the experiment results.

host=4, pro=16 1024// proc =4 //

TABLE II
TIME COST FOR MATRIX MULTIPLICATION.

Hosts	Procs	Average Time Cost (s)
1	4	14.270980
2	4	10.987764
4	4	9.010034

TABLE III
TIME COST FOR MATRIX CONVOLUTION.

Hosts	Procs	Average Time (s)
1	4	0.032002
1	16	0.120003
2	4	0.025631
2	16	0.067892
4	4	0.015679
4	16	0.021893

TABLE IV
TIME COST FOR MATRIX POOLING.

Hosts	Procs	Average Time (s)
1	4	0.063301
1	16	0.085479
2	4	0.024220
2	16	0.041198
4	4	0.008766
4	16	0.019812

From the above tables, we can ses that the time cost decrease with the number of hosts increases. But under the same host number, as the process number gets more, the performance gets worse. This might because that there is no such number of physical process in one server. And the communication time between hosts and processes cost most time.

III. WORDCOUT

A. Method

For this task, instead dividing each small file into each process or dividing each row of a big file into each process, we read the size in byte of each file, make the sum, and split by the number of workers. In this way each workers read and perform the words count on the same size of problem, reaching a perfect division of the problem between the processes.

To achieve that, at first each process must calculate the part processed by previous processes and then start to process its part. We consider all conditions: when it starts from beginning of a file, inside a word, on a space, on a new line, on an EOF, etc. The main steps of this algorithm are as follows.

- First, generate the master file which contains the path of all the files that are to be counted. Master process reads the master file. Then each of the processes should receive their portion of the file from the master process.
- Second, once a process has received its list of files to process, it should then read in each of the files and perform a word counting, keeping track of the frequency each word found in the files occurs. We will call the histogram produced the local histogram. This is similar to the map stage or map-reduce.
- Third, master process gathers frequencies of words across processes. Each process sends its local histogram to the master process. The master process just needs to gather up all this information and print the results. For example the word 'cat' might be counted in multiple processes and we need to add up all these occurrences. This is similar to the reduce stage of map-reduce.

In our designed algorithm, we split the problem based on the size of text, not by the number of files or by the numer of lines. Thus, there is no difference between the situation of several small files and the situation of one large file.

TABLE V
WORD COUNT RESULTS.

Small Files	Big File
502,817	146,521

TABLE VI
TIME COST FOR WORDCOUN OF 100 SMALL FILES.

Hosts	Procs	Average Time (s)
1	4	0.134651
1	16	0.223993
1	32	0.540004
2	4	0.044649
2	16	0.115963
2	32	0.239995
4	4	0.046426
4	16	0.087999
4	32	0.123951

B. Experiments

We also implement this algorithm with C and MPI. The experiments are conducted on HUAWEI Cloud with different number of hosts and processes. The words count results are shown in Table V. There are 502,817 words in those small files

and 146,521 words for the big file. The running time results are shown in Table VI and Table VII.

TABLE VII
TIME COST FOR WORDCOUN OF ONE BIG FILE.

Hosts	Procs	Average Time (s)
1	4	0.138615
1	16	0.647994
1	32	1.304004
2	4	0.063227
2	16	0.297363
2	32	0.711950
4	4	0.062360
4	16	0.148277
4	32	0.308154

From Table VI and Table VII, we can find similar trends with GEMM experiments. More hosts will decrease the running time, but under the same number of hosts, too many processes will increase average running time. This also might because that there is no such number of physical process in one server. And the communication time between hosts and processes cost most time.

IV. DEMO LINK

The demo for this project is upload to jbox, you can follow the link below and check the running process.

- The demo link:
  https://jbox.sjtu.edu.cn/l/3Jv9so.