Parallel Dense Linear Algebra

1. Algorithm Description

1.1 Matrix-vector multiplication

In this section, a brief overview of the matrix-vector multiplication algorithm implemented will be introduced. Next, MPI functions used in the algorithm will be discussed before the pseudo code is given.

Noticing the data layout of the matrix is inconsistent with the vector, the first step is to redistribute the vector to keep the dimension in each process consistent. There is generally two ways for redistribution, one of which is to gather all child vectors at the first process while the other is local gathering with respect to the local dimension of the sub-matrix. Since the first method may cause some problems of memory, the latter is used in this report. After multiplying in each process, the format of outputs should be changed into the same as inputs.

To begin with, if there exists only one process, do the multiplication directly. Else, split the whole process by row of the 2D block layout, which means blocks in the same row are split in one subgroup. In this manner, data in the vector can be all-gathered in the same subgroup so that it can multiply with the child matrix. Then, after sending the gathered data of the vector to each column of the grid corresponding to the rank in each subgroup via blocking Point-to-Point messaging with global communicator *MPI_COMM_WORLD*, *MPI_Allreduced* is used to sum the local product in each subgroup. Finally, scatter the result to maintain the same format as inputs. Pseudo code is shown below.

```
MPI_Comm_rank(comm,&rank);
MPI_Comm_size(comm,&size);
if size == 1 : Multiply directly;
else:
    MPI_Comm_split(comm, rank/A->np, rank, &conn_comm); // split by row
    Allgather x->data in each subgroup with communicator `conn_comm`, where x is the input vector;
    Send the gathered data to each corresponding column of grid, meaning to send the data to its transposed position of the grid;
    Do the local multiplication; Allreduce the local result in each subgroup;
    Scatter the reduced result;
```

1.2 Cannon's algorithm

The key point of Cannon's algorithm is block shifting. Denote the total process is a prefect square number $p = s^2$, and $A(i, j)$, $B(i, j)$ is the sub-matrix in the grid with row i and column j. In order to do multiplication, every row of matrix A has to be left-circular-shifted while every column of B has to be up-circular-shifted. It would be useful to shift by splitting the whole process by row and by column so that shifting can be done in subgroups. Pseudo code is shown below.

```
for all i in 0 to s-1:
    Left-circular-shift row i of A by i;
    Up-circular-shift column i of B by i;
    // MPI_Sendrecv is applied to avoid deadlock of message sending
for all k in 0 to s-1:
    for all i, j in 0 to s-1:
        C(i, j) += A(i, j) * B(i, j);
        Left-circular-shift each row of A by 1;
        Up-circular-shift each column of B by 1;
        // MPI_Sendrecv_replace is applied to update A(i, j) and B(i, j)
```

1.3 SUMMA

SUMMA is a more general algorithm compared with the Cannon's. Shifting is not required in this algorithm, so SUMMA looks more straightforward. Splitting processes by row and by column is also effective for communication. Using the same notations as Chapter 1.2, pseudo code of SUMMA is shown below. One thing should be noticed that SUMMA is also fixed for situations where p is not a perfect square number, and where the local sizes of sub-matrices are not consistent.

```
for all k in 0 to s-1:
    for all i in 0 to s-1:
        broadcast A(i, k) to row;
    for all j in 0 to s-1:
        broadcast B(k, j) to column;
    C(i, j) += A(i, j) * B(i, j);
```

2. Performance Analysis

All of the algorithms in Chapter 1 will be simulated on Hamilton with strong scaling and weak scaling. Analytically, time cost for strong scaling

should be proportional to $1/p$, while that for weak scaling should be a constant. To guarantee the programme runs successfully from one to 256 processes, 16 nodes of Hamilton are used and 24 tasks per node is set in the *SLURM* file. Additionally, fix $N = 25600$ for strong scaling while setting $N = p$ for weak scaling.

Since the number of process should be perfectly squared and be divided exactly by N, only $[1,4,16,64,100,256]$ processes were simulated for strong scaling of *MATMULT*. It is found that the programme came across a deadlock only on the condition that $p = 4$. The reason might be that *MPI_Send* and *MPI_Recv* are used for communication and the size of message to send is too large. After *MPI_Sendrecv* takes the place of them to avoid deadlocks and *OpenBLAS* paralleling is avoided, a figure describing the performance is plotted as Fig. 1 using average time cost data, with the number of process being x-axis and time cost being y-axis.

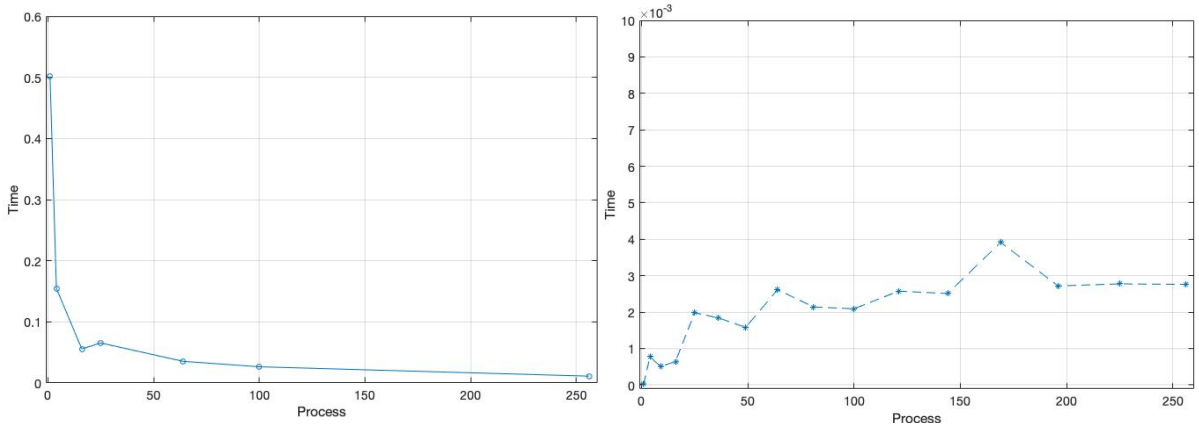

Fig. 1 Performance of *MAT_MULT*

The left-hand-side of Fig. 1 describes strong scaling and the time cost begins with around 0.5s at only one process. Then, it decreases dramatically as process grows. When process reaches 256, time cost is close to 0.01. According to the plot, time cost does not converge at 256 processes, and theoretically, it could still decline when process continues increasing.

On the right-hand side of weak scaling, time cost fluctuates slightly after 25 processes and the curve looks stable. Strictly speaking, time cost of weak scaling in this simulation will not be a constant because loops in *OpenBLAS* are not paralleled by *MPI*. Since the size of local matrix in every trail is not divided equally, the proportion of series and parallel is unbalanced, and it is hard to maintain the balance due to inconstancy of the size of local matrices and vectors. This is why time cost increases at the beginning. However, the curve

shows the disproportion makes little difference with process growing.(see the last three points)

Same parameters are applied to simulate *MAT_MAT_MULT* with respect to strong scaling while local dimension is maintained as 16-by-16 in weak scaling. For example, global matrix is 16-by-16 when conducting one process whereas 32-by-32 when conducting 4. The performance is plotted as Fig. 2.

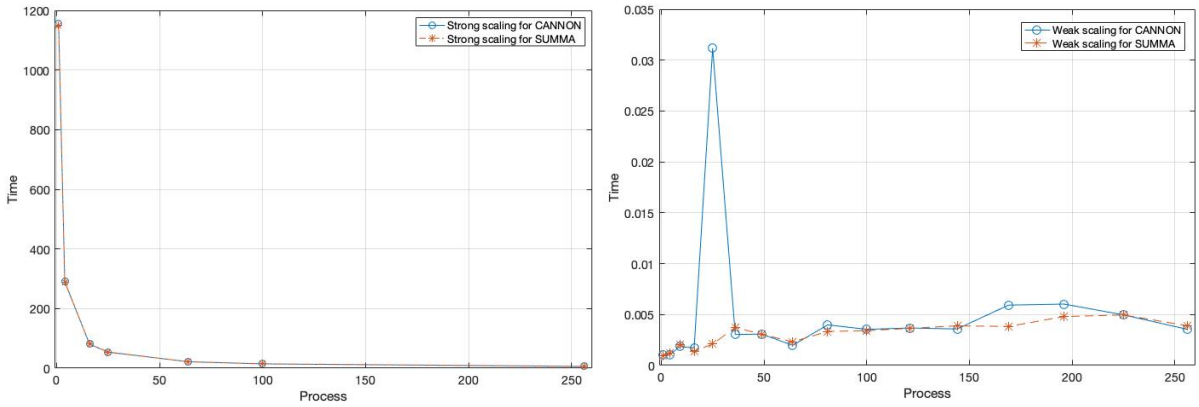

Fig. 2 Performance of *MAT_MAT_MULT*

With the same axises as Fig. 1, there are two curves in each subplot of Fig. 2, blue for CANNON and orange for SUMMA. Fig. 2 also shows similar trend to Fig. 1. In the strong scaling(left-hand-side subplot of Fig. 2), both curves, starting with around 1200s, drop significantly with process increasing, ending up with around 6s. But it is interesting to concentrate on the difference of the two algorithms. Unfortunately, the plot is not able to demonstrate their dissimilarities while *BENCH* data show CANNON costs slightly less time than SUMMA.

Details of the two algorithms can be detected in weak scaling(right-hand-side subplot of Fig. 2). Generally, two curves troop together and nearly stay stable. However, there is an outlier at 25 processes on blue curve. This might be because latency between nodes in this trail as 24 tasks are set per node. The subplot demonstrates tiny difference of time cost for the two algorithms. SUMMA costs slightly less time than CANNON sometimes while it also overrides occasionally. Thus, it is hard to judge which algorithm performs better in the trail.

3. Conclusion

Parallel computing can greatly save time for running big programmes. Theoretically, the more process is used in MPI programs, the more time value can be saved. However, there also exists latency for inner-nodes and nodes communication, which might cause some bias for performance analysis.