High-Performance Computing

1. Description of algorithms

1.1 Matrix-Vector Multiplication

The whole N*N matrix will be divided into N*N vectors, and the matrix loop operation is prepared. Firstly, the buf will send the vectors and multiply them with the vectors of each row and column in the matrix. At the same time, each sendbuf operation will back up the data, and finally the matrix values of each row will be multiplied to get the final result.

1.2 Cannon's algorithm

1.2.1 Algorithm principle

Cannon algorithm is an algorithm to optimize matrix block multiplication, and it is a storage efficient algorithm. Different from parallel block multiplication, it is not the multiple broadcasting of rows and columns of the array, but the cyclic shifting of rows and columns to meet the multiplication requirements. This method can reduce the total memory requirements of the processor.

The following matrices, matrices A and B are sums with n blocks($A_{i,j}$ and $B_{i,j}$), and we allocate $\sqrt{n}$ x $\sqrt{n}$ processors in calculation.

$$A=\begin{bmatrix} A_{11} & A_{12} & A_{13} & \cdots \\ A_{21} & A_{22} & A_{23} & \cdots \\ \vdots & \vdots & \vdots & \ddots \\ A_{\sqrt{n}1} & A_{\sqrt{n}2} & A_{\sqrt{n}3} & A_{\sqrt{n}\sqrt{n}} \end{bmatrix} \times B=\begin{bmatrix} B_{11} & B_{12} & B_{13} & \cdots \\ B_{21} & B_{22} & B_{23} & \cdots \\ \vdots & \vdots & \vdots & \ddots \\ B_{\sqrt{n}1} & B_{\sqrt{n}2} & B_{\sqrt{n}3} & B_{\sqrt{n}\sqrt{n}} \end{bmatrix}$$

1.2.2 Algorithm description

At the beginning, the processor stores $A_{i,j}$ and $B_{i,j}$, and then calculates $C_{i,j}$, and the algorithm continues to perform the following steps:

1. Circulate $A_{i,j}$ by I steps to the left and move $B_{i,j}$ by J steps to the upper ring;
2. Execute multiply-add operation, circularly move the block $A_{i,j}$ to the left by one step, and circularly move the block $B_{i,j}$ up by one step;
3. Repeat the second step, and perform $\sqrt{n}$ times of multiply-add operation and $\sqrt{n}$ times of cyclic single-step shift of blocks $A_{i,j}$ and blocks $B_{i,j}$.

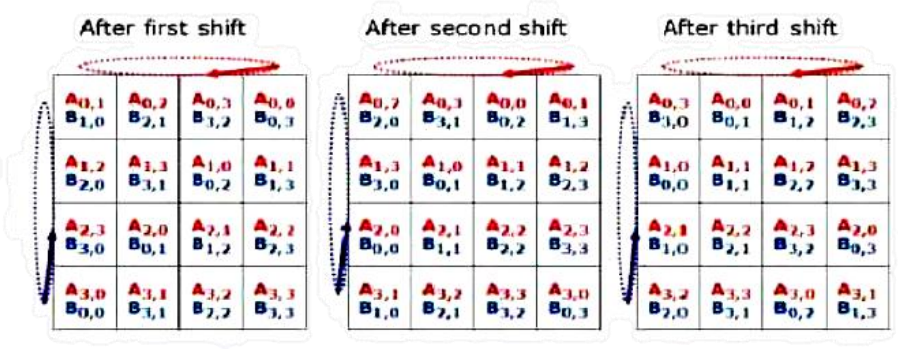

Figure 1.1 Schematic diagram of execution steps

1.3 SUMMA algorithm

The core idea of SUMMA algorithm is that each processor collects all columns of A matrix subblock in the same row processor and all rows of B matrix subblock in the same row processor, and then multiplies the rows and columns and accumulates them to form a block matrix of C matrix. Finally, the processor with rank=0 collects the data of other processors to form the final matrix c.

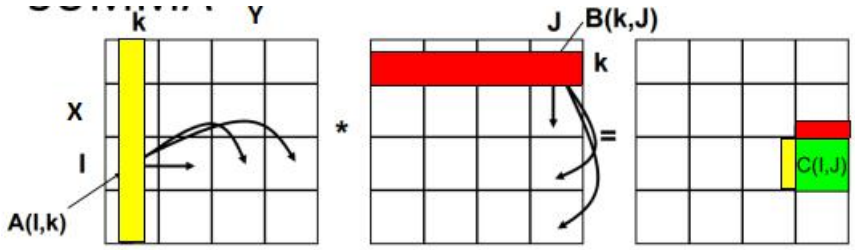

Figure 1.2 Schematic diagram of algorithm analysis

As shown in the figure as $X \times Y$ two-dimensional grid graph, it can be seen that there are X processor rows and Y processor columns in the graph, and each processor number is (S_I,S_J). Matrix A, matrix B and matrix result C are all divided into row and column blocks according to $X \times Y$.

$C(I,J)$ is defined as $C(I,J) = C(I,J) + \sum kA(I,k) * B(k,J)$

1.4 Compared SUMMA with Cannon algorithm

Compared with cannon algorithm, SUMMA algorithm's advantage lies in that SUMMA algorithm can calculate the multiplication of m*l's A matrix and l*n's B matrix (M, L and N can not be equal), while cannon algorithm can only realize the multiplication of n*n's A matrix and n*n's B matrix, which has great limitations. However, due to the limited level, and in order to partition the matrix well, the SUMMA algorithm implemented by MPI requires that the input matrix dimensions (m, l, n) must be integral multiples of the number of processors (defined by the user), otherwise an error will be prompted.

2. Analysis and presentation of benchmarking data

2.1 presentation of benchmarking data

The following are presentations of benchmarking data.

```
Timing data for MatMatMult[SUMMA] on 1 processes, matrix size 1
All data in seconds. Min, Mean, Max, Standard deviation.
1.3113e-05 1.3113e-05 1.3113e-05 0
Timing data for MatMatMult[SUMMA] on 4 processes, matrix size 4
All data in seconds. Min, Mean, Max, Standard deviation.
3.50475e-05 4.19617e-05 3.77297e-05 2.9611e-06
Timing data for MatMatMult[SUMMA] on 16 processes, matrix size 16
All data in seconds. Min, Mean, Max, Standard deviation.
4.79221e-05 0.000114918 8.56519e-05 2.1507e-05
Timing data for MatMatMult[SUMMA] on 25 processes, matrix size 25
All data in seconds. Min, Mean, Max, Standard deviation.
0.000622034 0.00127196 0.000694838 0.000123082
Timing data for MatMatMult[SUMMA] on 36 processes, matrix size 36
All data in seconds. Min, Mean, Max, Standard deviation.
0.000464916 0.000757933 0.00064928 0.000115931
Timing data for MatMatMult[SUMMA] on 49 processes, matrix size 49
All data in seconds. Min, Mean, Max, Standard deviation.
0.000681877 0.000895977 0.00077326 5.56046e-05
Timing data for MatMatMult[SUMMA] on 64 processes, matrix size 64
All data in seconds. Min, Mean, Max, Standard deviation.
0.000192881 0.00054121 0.000415776 0.000111903
```

Figure2.1 Part of The Data Results

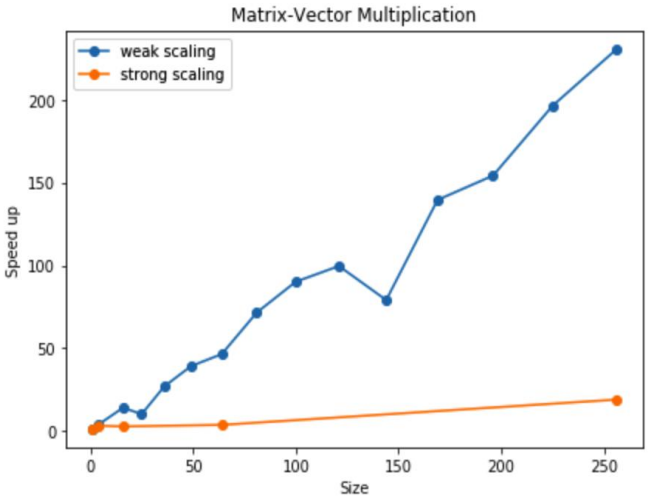

Figure2.2 Matrix-Vector Multiplication

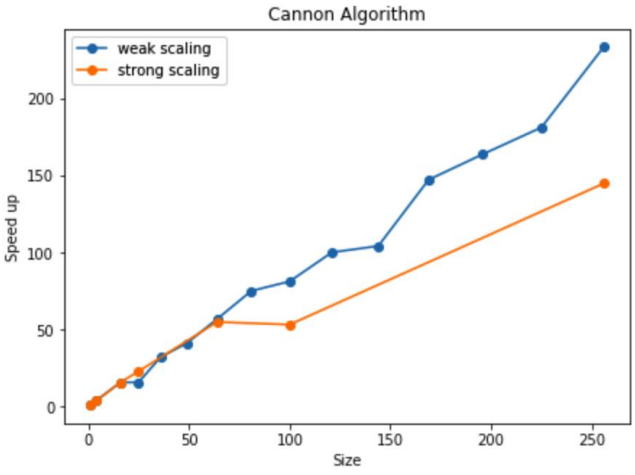

Figure2.3 Cannon Algorithm

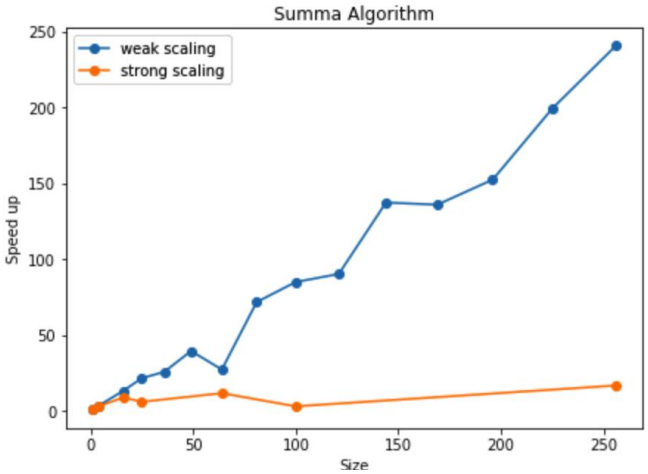

Figure2.4 Summa Algorithm

2.2 Analysis of benchmarking data

The above picture shows the size and speed up images of matrix-vector multiplication, cannon algorithm and Summa Algorithm. The blue line shows the trend of weak scaling, and the orange line shows the trend of strong scaling.

2.2.1 Strong Scaling and Amdashl's Law

Strong Scaling is a method to measure how time changes with the increase or decrease of the number of processors under the condition that the total scale of the

problem is constant. According to Amdahl's law, for a fixed-scale problem, the acceleration ratio s that can be achieved by increasing the number of processors can be expressed by the following formula:

$S = \frac{1}{(1-P)+\frac{P}{N}}$, P is the ratio of the execution time of parallelizable partial substitute code to the total execution time, and N represents the total number of processors executing parallelizable code.

In this figure, size is equivalent to N in the formula, and S corresponds to speed. It can be seen from the figure that the overall trend of Weak scaling is increasing, but the growth trend is relatively stable compared with the Strong scaling curve, and the Speed increases slowly with the increase of size. It can also be seen from the formula that N has little influence on S and is mostly influenced by P, so the experimental data is in good agreement with the theoretical data.

2.2.2 Weak Scaling and Gustafson's Law

Weak scaling is a method to measure how time changes with the increase or decrease of the number of processors when the scale of problems that each processor can handle is constant. Weak scaling is often calculated by Gustafson's law. According to the definition of weak scale, the scale of problems handled by the system will increase with the increase of processors. Therefore, the acceleration ratio S can be expressed by the following formula:

$S = N + (1-P)(1-N)$.

P represents the ratio of the parallelizable part running time to the total running time in the serial execution problem, and N represents the number of processors.

In this figure, size is equivalent to N and S corresponding to Speed in the formula. It can be seen from the image that speed increases with the increase of size. Although the data is unstable, the overall situation is still rising. And it can be seen from the figure that with the increase of the absolute value of size, the upward trend of speed up becomes steeper. The results in the image are in good agreement with the theoretical results of the formula.

2.2.3 compare the algorithm

The speed of the three algorithm will increase with the increase of size, but in different methods, the speed is affected by size differently. By comparison, it can be found that Summa algorithm has a faster growth response than the other two when the number of processes is large. The growth trend of Cannon algorithm is relatively stable. Therefore, all three algorithm can improve the best performance of quantitative analysis.