Dense Matrix Algorithms

1 Matrix-Vector Multiplication

1.1 Serial Situation

Suppose we need to multiply a dense $n \times n$ matrix A with an $n \times 1$ vector x to yield $n \times 1$ result vector y, so:

$$Ax = y \tag{1}$$

Then the serial algorithm will require n^2 multiplications and additions, so the runtime is $O(n^2)$.

1.2 1D Partitioning

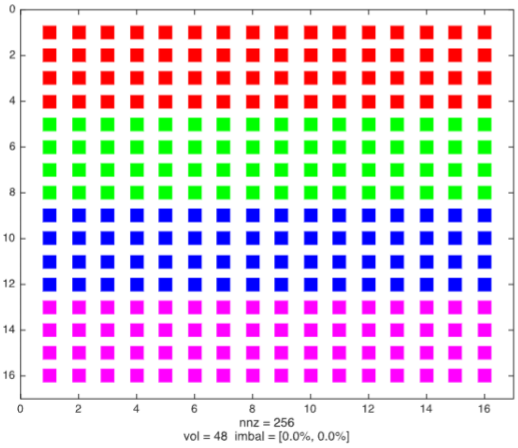

Figure 1: 1D Partitioning

1.2.1 When $n = p$

Now matrix A is partitioned among p processors, each processor stores **complete row of the matrix**. Vector x is also partitioned, each process owns one element of x. The algorithm includes the following steps:

1. **Step 1**: Use AllGather to distribute all of x to each processor. Recall the communication primitive knowledge:

$$T_{comm} = O(\tau \log n + \mu n) \tag{2}$$

2. **Step 2**: Now each processor will compute:

$$y[i] = \sum_{j=0}^{n-1} A[i,j] \cdot x[j] \tag{3}$$

The computation runtime will be $O(n)$

1.2.2 When $n > p$

Now each processor stores n/p complete rows of the matrix A, and n/p elements of the vector x

The algorithm includes the following steps:

1. **Step 1**: Distribute all of x vector to each processor. This will use AllGather operation among p processors, including messages of size n/p. Therefore:

$$T_{comm} = O(\tau \log p + \mu \cdot \frac{n}{p} \cdot p) = O(\tau \log p + \mu n) \tag{4}$$

2. **Step 2**: Then on each processor, n/p local dot products on vectors of length n. So:

$$T_{comp} = O(\frac{n^2}{p}) \tag{5}$$

1.3 2D Partitioning

1.3.1 $p = n^2$

Suppose $n \times n$ matrix is partitioned among n^2 processors, so each processor owns a single element. In addition, we have $n \times 1$ vector x distributed in the last column of n processors.

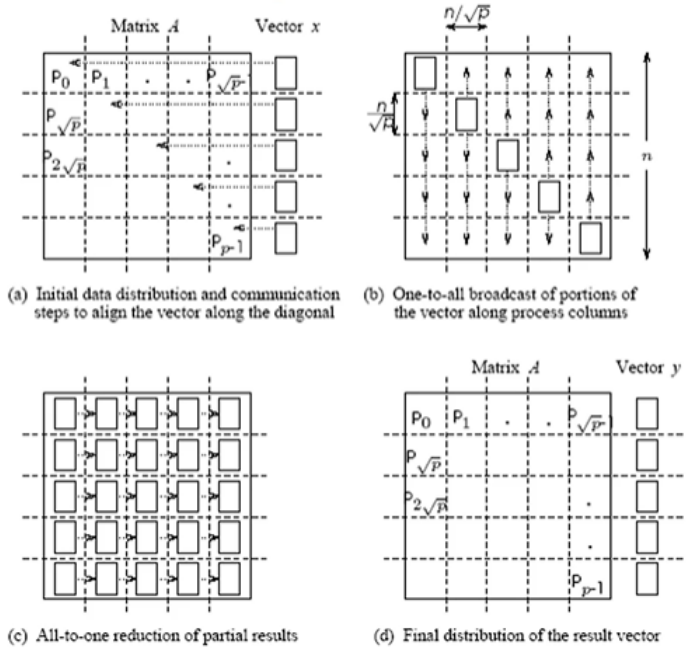

Figure 2: 2D Partition Steps

For step a:

1. The $n \times n$ matrix A is divided into $\sqrt{p} \times \sqrt{p}$ blocks. At this case, $\sqrt{p} = n$. Each processor P_{ij} holds $(n/\sqrt{p})$ messages.

2. The vector x is also partitioned and distributed along the diagonal processors P_{ii} because each processor requires the corresponding element of vector x to multiply with its block of matrix A.

For step b:

1. After aligning the vector x on the diagonal, a **one-to-all broadcast** is performed within each column of processors.

2. Each diagonal processor P_{ii} sends its portion of the vector to all other processors in the same column. This step ensures that every processor in a column has the necessary part of the vector x to proceed with the multiplication with its local block of the matrix A.

For step c:

1. Each processor multiplies its block of the matrix A with the corresponding elements of vector x to compute a partial result of the resulting vector y.

2. Once all processors have computed their partial results, an **all-to-one reduction** is performed within each row of processors to sum up the partial results. This reduction step is necessary to construct the elements of the resulting vector y as each processor in a row holds a part of the sum needed for a single element of y.

For step d:

1. After the reduction step, each end of row processor will have a complete element of the resulting vector y. This is ready for final distribution.

All the steps could be summarized as:

1. Align vector along the main diagonal: **one-to-one communication**
2. Broadcast vector element to n processors in column: **one-to-all broadcast**
3. Local multiplication
4. Sum partial y values in each row: **all-to-one reduction**

Recall that the expressions of the communication time for **one-to-all broadcast** and **all-to-one reduction** are both $O(\tau \log p + \mu m \log p)$. Here p is should be the number of processors in row and column, they are both n. Each processor only has one element, so $m = 1$. The local multiplication does not need communication, and the **one-to-one communication** takes $O(\tau + \mu)$. Therefore:

$$T_{Comm} = O(\tau \log n + \mu \log n) \tag{6}$$

For the computation time, we need to add each partial results to get single element in y using **all-to-one reduction**. For each level, it requires 1 computation, and it has $\log n$ levels in this case. Therefore:

$$T_{Comp} = O(\log n) \tag{7}$$

1.3.2 $p < n^2$

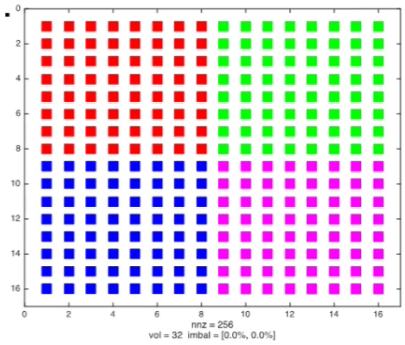

Figure 3: p less than n^2 case

Basically the same as the general form in the last subsection. Now:

1. Each processor owns an $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$ block of the matrix
2. The vector is distributed in portions of $\frac{n}{\sqrt{p}}$ elements in the last processor-column
3. The message sizes for the alignment, broadcast, and reduction are all $\frac{n}{\sqrt{p}}$. For the reduction, after local computation it will just be a $\frac{n}{\sqrt{p}}$ vector.

4. The local computation is a product of an $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$ submatrix with a vector of length $\frac{n}{\sqrt{p}}$

The algorithm is the same as previous case, but different runtime.

1. To align vector along the main diagonal, we use **one to one communication**, the runtime is $O(\tau + \mu\frac{n}{\sqrt{p}})$
2. Broadcast vector elements to $\sqrt{p}$ processors in column, this will take $O(\tau \log \sqrt{p} + \mu\frac{n}{\sqrt{p}} \log \sqrt{p})$
3. Local multiplication will take $O(\frac{n^2}{p})$
4. Sum partial y values in each row using **reduce** in $\sqrt{p}$ processors, this will take $O(\tau \log \sqrt{p} + \mu\frac{n}{\sqrt{p}} \log \sqrt{p})$

Therefore, the communication time at this case will be:

$$T_{Comm} = O(\tau \log \sqrt{p} + \mu\frac{n}{\sqrt{p}} \log \sqrt{p}) \tag{8}$$

And the computation will be:

$$T_{Comp} = O(\frac{n^2}{p}) \tag{9}$$

Therefore the total runtime will be:

$$T = O(\frac{n^2}{p} + (\tau + \mu\frac{n}{\sqrt{p}}) \log \sqrt{p}) = O(\frac{n^2}{p} + \frac{n}{\sqrt{p}} \log \sqrt{p}) \tag{10}$$

To achieve maximum efficiency, we have:

$$\frac{O(n^2)}{p \cdot O(\frac{n^2}{p} + \frac{n}{\sqrt{p}} \log \sqrt{p})} = 1 \tag{11}$$

$$\frac{O(n^2)}{O(n^2 + n\sqrt{p} \log \sqrt{p})} = 1 \tag{12}$$

$$O(n\sqrt{p} \log \sqrt{p}) = O(n^2) \tag{13}$$

$$O(\sqrt{p} \log \sqrt{p}) = O(n) \tag{14}$$

Therefore, we get:

$$p = O(\frac{n^2}{\log^2 n}) \tag{15}$$

So this algorithm is efficient up to $O(\frac{n^2}{\log^2 n})$ processors.

2 Matrix-Matrix Multiplication

2.1 Serial Situation

Suppose we need to multiply a dense $n \times n$ matrix A with an $n \times n$ matrix B to yield $n \times n$ result matrix C, so:

$$AB = C \tag{16}$$

The implementation is shown below:

```
for (i = 0; i < n; ++i)
    for (j = 0; j < n; ++j) {
        double sum = 0;
        for (k = 0; k < n; ++k)
            sum += A[i][k] * B[k][j];
        C[i][j] = sum;
    }
}
```

Figure 4: Serial Matrix-Matrix Multiplication

Then the serial algorithm will require n^3 multiplications and additions, so the runtime is $O(n^3)$.

2.2 Normal Block Algorithm

2.2.1 Algorithm

In parallel case, we can use **block** operations to decompose the problem:

1. Divide $n \times n$ matrix A can be recomposed into $q \times q$ array of blocks
2. $A_{i,j}(0 \leq i, j \leq q)$ will be used for block notation, each block contains $(\frac{n}{q} \times \frac{n}{q})$ submatrix.
3. With this decomposition, we need to perform q^3 matrix multiplications, each involving $(\frac{n}{q} \times \frac{n}{q})$ matrices, as shown below.

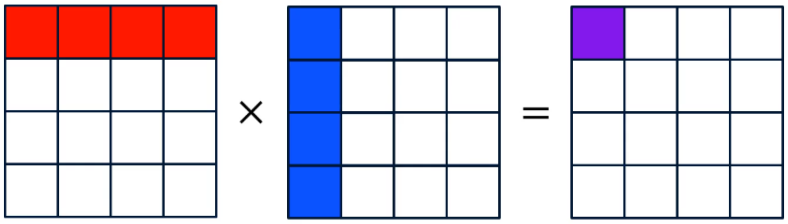

Figure 5: Matrix-Matrix Multiplication Example

The algorithm details are shown below:

1. Suppose now the decomposition is done, matrices A and B partitioned into p blocks A_{ij} and $B_{ij}(0 \leq i, j \leq \sqrt{p})$, each block has a $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$ submatrix
2. Processor P_{ij} initially stores A_{ij} and B_{ij}. In order to compute submatrix C_{ij} requires all A_{ik} and B_{kj}, where $0 \leq k < \sqrt{p}$.
3. This is done by **AllGather** operation, gather blocks of A along rows and B along columns. Notice that the function of **AllGather** is assembling, not adding the matrix.
4. After **AllGather**, the local multiplication will be applied, and C_{ij} will be calculated.

2.2.2 Runtime Analysis

1. First we need to do the **AllGather** operations in row and column, within $\sqrt{p}$ processors. Each processor has $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}} = \frac{n^2}{p}$ elements. Therefore the communication time is $O(\tau \log \sqrt{p} + \mu \frac{n^2}{p} \sqrt{p})$.
2. For each processor, the computation requires $\sqrt{p}$ (number of processors in row) multiplications of two $(\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}})$ sized submatrices, this will require $O(\sqrt{p} \times (\frac{n}{\sqrt{p}})^3) = O(\frac{n^3}{p})$.
3. Notice that here each processor already stores one row in A and one column in B, so the local computation could just get the corresponding value (same position) in C.
4. Therefore the total runtime is $O(\frac{n^3}{p} + \tau \log \sqrt{p} + \mu \frac{n^2}{p} \sqrt{p})$
5. This will be efficient for $p = O(n^2)$
6. This algorithm has a very high memory requirement because each processor need to store the whole row or column's message. **So it is not memory optimal.**

2.3 Cannon's Algorithm

2.3.1 Algorithm

1. First align the blocks depending on their positions in the matrix. Suppose we have matrix A and matrix B as shown below:

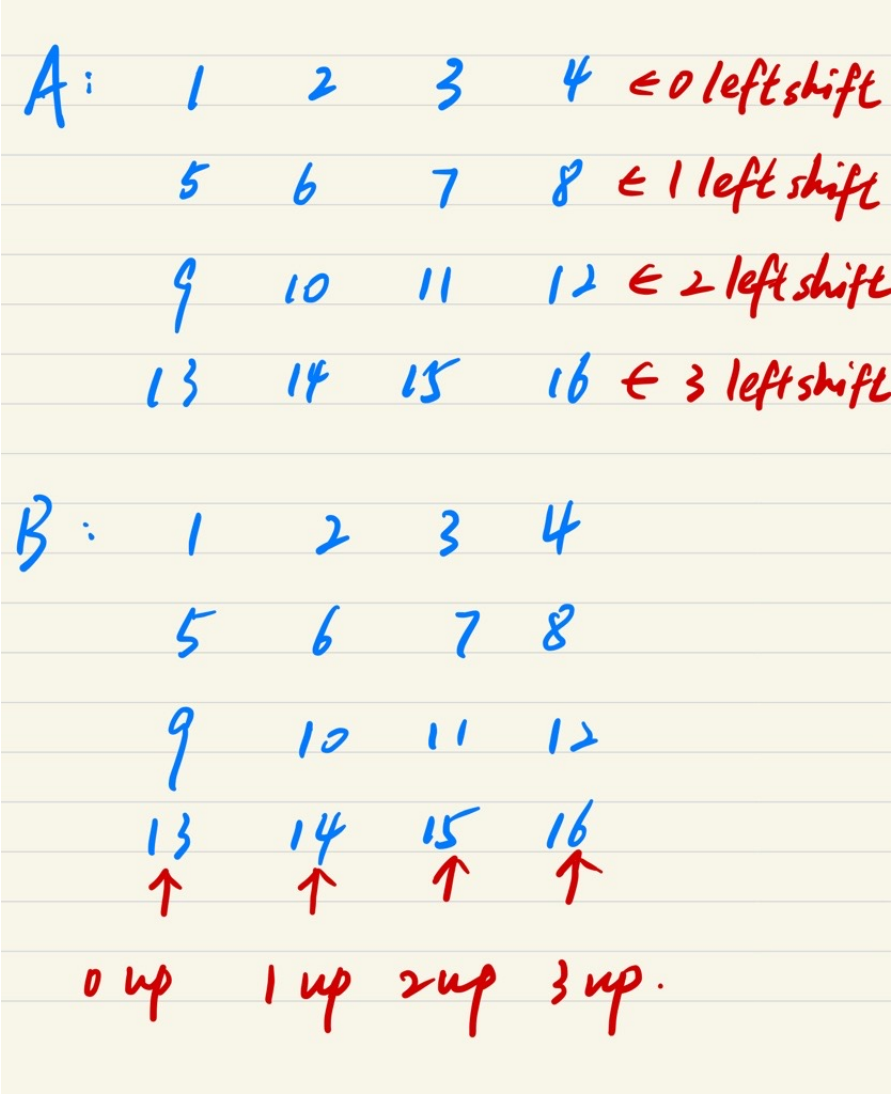

Figure 6: Matrix A and Matrix B

Notice that for each row and column, the shifting distances are different. After the alignment process, $P_{i,j}$ will have $A_{i,(j+1)mod\sqrt{p}}$ and $B_{(i+j)mod\sqrt{p},j}$. After the alignment, the matrices are shown below, and we calculate the C matrix using the formula:

$$C_{ij} = A_{ij} \cdot B_{ij} \tag{17}$$

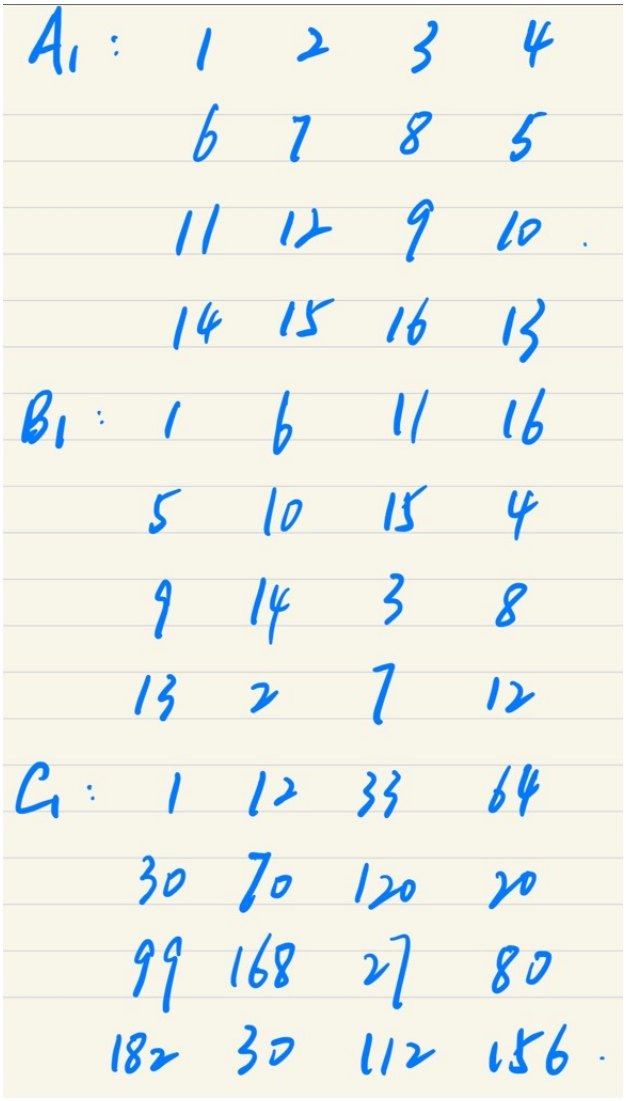

Figure 7: After Alignment

2. Then do the 1 left shift in matrix A and 1 up shift in matrix B, and calculate C_2:

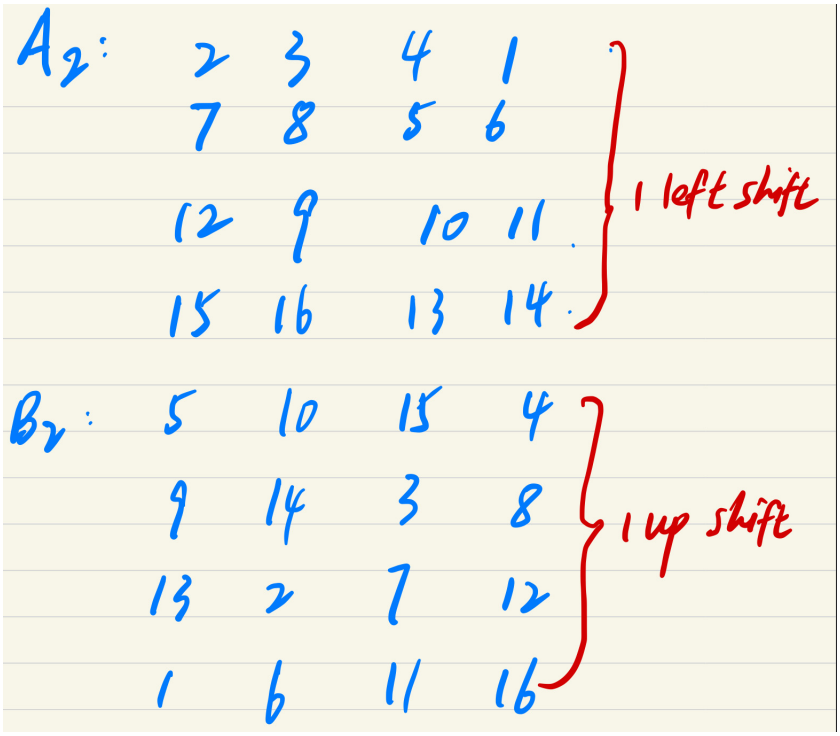

Figure 8: Shifting process

3. Then repeat these steps for $\sqrt{p}-1$ times, and the final C matrix will be:

$$C = C_1 + C_2 + C_3 + C_4 \tag{18}$$

2.3.2 Runtime Analysis

1. In the alignment step, the maximum distance for a block to shift is $\sqrt{p}-1$, so two shift operations require $O(2*(\tau(\sqrt{p}-1)+\mu\frac{n^2}{p}(\sqrt{p}-1))) = O(\tau\sqrt{p}+\mu\frac{n^2}{p}\sqrt{p})$

2. The compute and shift phase has $\sqrt{p}-1$ steps. In each step, the computation requires the multiplications of two $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$ submatrices, so the computation time is $O((\sqrt{p}-1) \times (\frac{n}{\sqrt{p}})^3) = O(\frac{n^3}{p})$

3. The communication time for $\sqrt{p}-1$ steps shifts will take $O(\tau(\sqrt{p}-1)+\mu\frac{n^2}{p}(\sqrt{p}-1)) = O(\tau\sqrt{p}+\mu\frac{n^2}{p}\sqrt{p})$

4. So the final total time is $O(\frac{n^3}{p}+\tau\sqrt{p}+\mu\frac{n^2}{\sqrt{p}})$. Notice that this runtime is larger than the block algorithm, but Cannon's algorithm is more memory optimal.