Final report

Distributed matrix multiplication

March 2019

The work has been done by MSc. 1st year students Artem Vasilev and Mahmud Allahverdiyev

1 Introduction

Matrix multiplication is one the central operations in a number of numerical algorithms. Up to now, several different approaches have been proposed to improve the efficiency of matrix multiplication algorithms. If we apply the original mathematical definition of matrix multiplication, it gives an algorithm that takes time on the order of n^3 to multiply two $n \times n$ matrices ($\Theta(n^3)$ in big $\mathcal{O}$ notation). Better asymptotic bounds on the time required to multiply matrices have been known since the work of Strassen in the 1960s, but it is still unknown what the optimal time is. The definition of matrix (1) multiplication is that if $C = AB$ for an $n \times m$ matrix A and an $m \times p$ matrix B, then C is an $n \times p$ matrix with entries

$$c_{ij} = \sum_{k=1}^{m} a_{ik} b_{kj} \tag{1}$$

2 General and distributed matrix multiplication algorithms

Strassen's algorithm which runs in $\mathcal{O}(N^{2.81})$ time was an improvement to the naive algorithm. In the following decades a number of theoretical and practical improvements have been made, including CopperSmith-Winograd algorithm which has a complexity of $\mathcal{O}(n^{2.3754})$. This algorithm is based on approaching the problem via Arithmetic progressions. Despite of the smaller exponent in the complexity, this algorithm has a large constant; therefore, is not used in practice.

In general, algorithms with better asymptotic running time than the Strassen algorithm are rarely used in practice, due to the same factor in their running times which make them impractical. Theoretically, it is possible to improve the exponent of the algorithm further; however, the exponent must be at least 2.

The reason for the latter is obvious; the matrices contains $\mathcal{O}(n^2)$ entries and all of them have to be read at least once to calculate the exact result.

Strassen's algorithm improves the naive multiplication by grouping the operations in a clever manner. For convenience, let's assume that matrices A, B have sizes $2^n \times 2^n, n \geq 0$; if it's not the case, we can pad the matrices with zeros to get them have the sizes equal to 2-s powers. Let's partition matrices A, B and C into block form

$$A = \begin{vmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{vmatrix}$$

,

$$B = \begin{vmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{vmatrix}$$

and

$$C = \begin{vmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{vmatrix}$$

where the smaller matrices have dimensionality of $2^{n-1} \times 2^{n-1}$. The naive algorithm does not exploit the grouping possibilities for submatrices to reduce the number of possible matrix multiplications. Strassen introduces the result via new matrices:

$$\begin{aligned} c_{11} &= f_1 + f_4 - f_5 + f_7, \\ c_{12} &= f_3 + f_5, \\ c_{21} &= f_2 + f_4, \\ c_{22} &= f_1 - f_2 + f_3 + f_6, \end{aligned} \tag{2}$$

where

$$\begin{aligned} f_1 &= (a_{11} + a_{22})(b_{11} + b_{22}), \\ f_2 &= (a_{21} + a_{22})b_{11}, \\ f_3 &= a_{11}(b_{12} - b_{22}), \\ f_4 &= a_{22}(b_{21} - b_{11}), \\ f_5 &= (a_{11} + a_{12})b_{22}, \\ f_6 &= (a_{21} - a_{11})(b_{11} + b_{12}), \\ f_7 &= (a_{12} - a_{22})(b_{21} + b_{22}). \end{aligned} \tag{3}$$

Using only 7 multiplications and 18 matrix additions (2), (3) it has the lower complexity than the naive algorithm. In order to get a reasonable speedup via Strassen's algorithm, large sized matrices are needed since the constant factor in the naive algorithm is comparably smaller.

Block algorithms have become building blocks for distributed algorithm routines, including the world-ruling BLAS-1, BLAS-2 and BLAS-3 modules are crucial factors in fast developing parallel programming world.

The three loops in iterative matrix multiplication can be arbitrarily swapped with each other without an effect on correctness or asymptotic running time. However, the order can have a considerable impact on practical performance due to the memory access patterns and cache use of the algorithm; which order is best also depends on whether the matrices are stored in row-major order, column-major order, or a mix of both.

In a perfect setting, we can distribute the data of matrix in blockwise fashion between p servers (1).

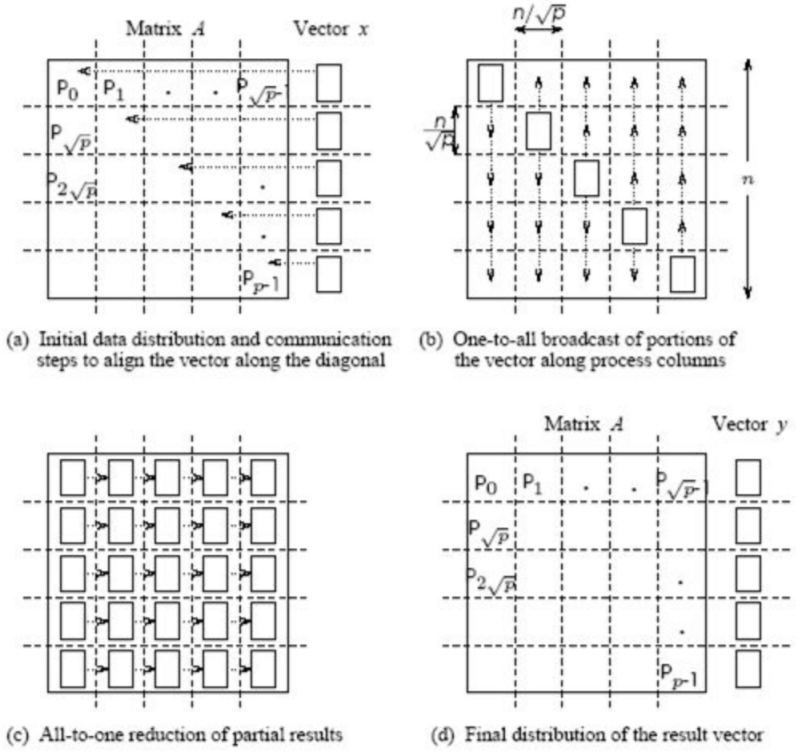

Figure 1: Distributing operations between servers

3 GASP codes for Distributed Matrix Multiplication

Now, consider a setting where two users have matrices $A \in \mathcal{F}_q^{r \times s}$ and $B \in \mathcal{F}_q^{r \times s}$, respectively. The common goal is to calculate AB in such a distributed way that the operations are done by N independent servers while assuming that no information is about A or B is revealed to any of the servers. Among all

servers, it is guaranteed that at most T of them may collude. Moreover, the communication costs, thus total amount of work should be minimized. From the previous formulations it's obvious that the total number of operations that the naive algorithm would take is $O(rst)$. In the liteature, a few generic methods can be found to apply polynomial codes for the information security problems [3], [4], [2]. These polynomials can also be useful for preserving the secure way of calculating matrix multiplication in several servers setting. GASP codes [1] are especially designed to beat the performance of the previous proposed approaches.

It's explicitly stated that not all of the polynomial are fit for the outperforming performance of the algorithm. The input matrices A and B are divided into row and column vectors, respectively. These vector sets can be denoted as $A_1, A_2, \ldots, A_r$ and $B_1, B_2, \ldots, B_t$. Then, each value of the product $(AB)_{ij} = A_i B_j$. The polynomial codes to retrieve the product are:

$$\begin{aligned} f(x) &= A_1 + A_2 x^{p_1} + \cdots + A_r x^{p_{K-1}} + R_1 x^{p_r} + \cdots + R_r x^{p_{K+T-1}} \\ f(x) &= B_1 + B_2 x^{q_1} + \cdots + B_r x^{q_{L-1}} + S_1 x^{q_r} + \cdots + S_r x^{q_{L+T-1}} \\ h(x) &= f(x)g(x) = A_1 B_1 + (A_1 B_2 + A_2 B_1)x + \ldots \end{aligned} \tag{4}$$

The powers p_i and q_i should be carefully chosen; otherwise, it's not possible to retrieve all elements of the correct output. Interestingly, not all of the powers do exist in the representation of $h(x)$, in turn, it indicates gaps in degrees. Moreover, we need lesser points to interpolate h. We can construct a degree table where the coefficient of the term with degree i in the product $f(x)g(x)$ has a nonzero value. The idea proposed is to minimize the number of such values so that the decoding becomes more efficient and security aspect are preserved.

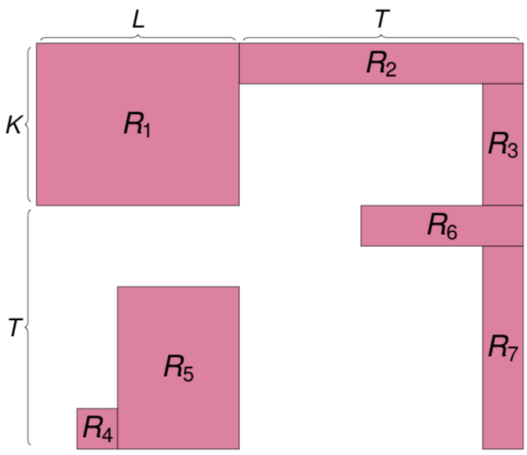

Figure 2: GASP active terms

For K, L, T parameter triples, they show that the number of big GASP terms

fall under the marked area (2) for all $L \leq K$. Apparently, the similar derivation has been made for $K < L$ case, where the number of nonzero coefficients in the product is named as small GASP.

The performance of small and big GASP have also been addressed; [1] claims that $GASP_{small}$ outperforms $GASP_{big}$ for $T < min\{K, L\}$.

4 Conclusion

In this report, we summarized the generic techniques for distributed matrix multiplication in secure environment. The findings from the recent GASP method have been discussed and the demo implementation is designed to assert their results.

References

[1] Rafael G. L. D'Oliveira, Salim El Rouayheb, and David A. Karpuk. Gasp codes for secure distributed matrix multiplication. *CoRR*, abs/1812.09962, 2018.

[2] Jaber Kakar, Seyedhamed Ebadifar, and Aydin Sezgin. Rate-efficiency and straggler-robustness through partition in distributed two-sided secure matrix computation. *CoRR*, abs/1810.13006, 2018.

[3] Ravi Tandon. The capacity of cache aided private information retrieval. In *55th Annual Allerton Conference on Communication, Control, and Computing, Allerton 2017*, 2018.

[4] Heecheol Yang and Jungwoo Lee. Secure distributed computing with straggling servers using polynomial codes. *IEEE Transactions on Information Forensics and Security*, 2018.