A Cloud-Friendly Communication-Optimal Implementation for Strassen’s Matrix Multiplication Algorithm

Jie ZHOU†, Nonmember and Feng YU†(a), Member

SUMMARY  Due to its on-demand and pay-as-you-go properties, cloud computing has become an attractive alternative for HPC applications. However, communication-intensive applications with complex communication patterns still cannot be performed efficiently on cloud platforms, which are equipped with MapReduce technologies, such as Hadoop and Spark. In particular, one major obstacle is that MapReduce’s simple programming model cannot explicitly manipulate data transfers between compute nodes. Another obstacle is cloud’s relatively poor network performance compared with traditional HPC platforms. The traditional Strassen’s algorithm of square matrix multiplication has a recursive and complex pattern on the HPC platform. Therefore, it cannot be directly applied to the cloud platform. In this paper, we demonstrate how to make Strassen’s algorithm with complex communication patterns “cloud-friendly”. By reorganizing Strassen’s algorithm in an iterative pattern, we completely separate its computations and communications, making it fit to MapReduce programming model. By adopting a novel data/task parallel strategy, we solve Strassen’s data dependency problems, making it well balanced. This is the first instance of Strassen’s algorithm in MapReduce-style systems, which also matches Strassen’s communication lower bound. Further experimental results show that it achieves a speedup ranging from 1.03× to 2.50× over the classical Θ(n^3) algorithm. We believe the principle can be applied to many other complex scientific applications.

key words: parallel algorithms, communication-optimal, Strassen’s matrix multiplication, cloud computing, MapReduce

1. Introduction

As one of the most fundamental operations in linear algebra, matrix multiplication is applied in many scientific and engineering areas such as signal processing [1] and computer graphics [2]. Optimizations of matrix multiplication have been extensively studied in distributed computing, scientific computing and high-performance computing [3]–[8]. In recent studies, it has been argued that communication costs are and will continue to be the limiting factor in the design of efficient parallel algorithms [9]. As a result, some significant communication-avoiding matrix multiplication algorithms have been proposed to address the bottleneck in HPC domain [3], [9]–[12]. The state-of-the-art one is the Communication-Avoiding Parallel Strassen (CAPS) algorithm which reportedly performs asymptotically better than any previous classical or Strassen-based parallel algorithm and obtains 24% to 184% speedups on Cray XT4 [10].

Cloud computing [13] is emerging as an alternative computing platform for HPC applications with the benefits of virtualization, on-demand provision of compute resources and elimination of cluster setup time. These benefits are attracting more and more HPC applications users to migrate their work to cloud platforms. However, not all HPC applications can perform well on the cloud [14]. In order to make HPC applications “cloud-friendly” [15], three requirements should be satisfied. First, communication overhead should be reduced for communication-intensive applications, otherwise we would suffer severe performance degradation on the cloud platform [16], [17]. Second, we should express our algorithms in a fairly easy way to fit in the cloud technologies such as Hadoop and Spark. Third, we should design topology-oblivious algorithms, since the optimizations based on specific topologies will not take effect any more, due to the virtualization technology provided by the cloud [18].

For matrix multiplication, Strassen’s algorithm requires less communications and computations compared with the classical Θ(n^3) algorithm. It has been successfully implemented in HPC platforms [10], where the properties of high bandwidth and low latency are guaranteed. In particular, they adopt a BFS/DFS recursive approach to parallel Strassen’s algorithm and then synchronous, dedicated data movements between BFS/DFS steps are required to address data dependency problems. However, these technologies cannot be directly applied to the cloud platform. In particular, they are tedious and inefficient to express Strassen’s algorithm in the same recursive way as that on the HPC platform, since dedicated point-to-point communications between compute nodes are not supported. To date, the conventional matrix multiplication implementations on the cloud such as HAMA [19], Marlin [20], are all based on the classical-2D [21] algorithm.

In this paper, we present the first “cloud-friendly” implementation (MR-Strassen) of Strassen’s matrix multiplication algorithm on the cloud platform. It takes full advantages of Strassen’s algorithm in the cloud environment. Our paper makes the following two contributions.

1. We reorganize Strassen’s algorithm in an iterative pattern. Computations and communications can thus be completely separated to fit the MapReduce programming model.

2. A novel data/task parallel strategy is adopted to solve Strassen’s data dependency problems.
Our experiments are conducted on Amazon EC2 using Spark runtime. The results show that a speedup ranging from 1.03× to 2.50× is achieved over the classical $\Theta(n^3)$ algorithm for square matrix multiplication.

The rest of the paper is organized as follows. In Sect. 2, we detail the properties of cloud platforms and MapReduce programming model. In Sect. 3, we propose our MR-Strassen algorithm. In Sect. 4, we present a communication model accompanied with computation and communication analyses of MR-Strassen. In Sect. 5, we evaluate MR-Strassen on Amazon EC2 using Spark runtime. In Sect. 6, we describe prior efforts relevant to our work. Finally, in Sect. 7, we make conclusions and discuss our future work.

2. Preliminaries

2.1 Cloud Platforms

Cloud computing provides three categories of service models: Infrastructure-as-a-service (IaaS), Platform-as-a-service (PaaS) and Software-as-a-service (SaaS). Up to now, scientific community mainly relies on IaaS to setup their clusters [22]. A cluster typically consists of a master node and a number of work nodes, each of which has a few compute cores, a fast primary memory, a slower secondary memory and is connected via an underlying network to the rest of the cluster. As Qiming He [23] pointed out, poor network capability is the main reason for performance degradation when migrating HPC applications to the cloud. Virtualization technology used by cloud computing adds little performance overhead. However, when we allocate compute nodes from the cloud, the allocation cannot be predicted. It is an arbitrary subset of the cloud and the network topology information is hidden by virtualization from the users. Furthermore, MapReduce’s simple programming model hides the details of parallelization and data distribution, making it difficult to match algorithms according to the network topology. As a result, general topology-oblivious algorithms are preferred in MapReduce-style systems.

2.2 Cloud Technologies

To simplify data processing on large clusters provided by the cloud, cluster computing frameworks such as MapReduce [24], Hadoop [25] and Spark [26], [27] are used. All these frameworks share the following properties.

These frameworks operate in a way similar to the BSP model [28]. Computations are divided into basic rounds. In each round, processors receive input at the beginning, do some computations independently and write output to storage at the end. When all of the compute nodes finish their work in a basic round, synchronization is carried out. Only during synchronization are compute nodes allowed to exchange data between each other. All these frameworks operate on data in terms of key-value pair, and use keys as indices to do shuffling. Key-value pairs sharing the same key will be routed to a single compute node. Direct point-to-point communication is not supported.

In MapReduce-style frameworks, computations in the basic rounds are split into a number of tasks. In the input round, the task number equals to the number of input file splits. In the other rounds, the task number equals to the number of partitions produced by its previous round. Task numbers determine the maximum parallel degree we can achieve for each round. As a result, for applications with no intrinsic keys such as matrix multiplication, we can use our self-defined keys to control partition numbers and further control parallel degree of the applications.

Task assignment is automatically managed by the scheduler. When a processor finishes its task, a new task will be assigned to it by the scheduler. Typically, the number of tasks in each basic round exceeds the number of processors in the cluster. So it is common for a single processor to handle a number of tasks in sequence. Because of the task management layer, the strict one-to-one correspondence between task numbers and processor numbers does not exist anymore. Parallel degree is determined by task number instead.

3. MR-Strassen Algorithm

3.1 Overview

Strassen’s algorithm [29] is a recursive algorithm for matrix multiplication which has a lower computational complexity than the classical $\Theta(n^3)$. In practice, we prefer to use its variant, Strassen-Winograd algorithm [30] since it requires fewer additions. Suppose we do matrix multiplication of $C = A \times B$. The recursion tree in Fig. 1 shows details of the algorithm and specific terms are listed in Table 1.

For Strassen-Winograd algorithm, matrix $A$, $B$ and $C$ each own an individual tree called tree $A$, tree $B$ and tree $C$ respectively. They share the same tree structure shown in Fig. 1(a). The tree height represents number of recursion levels of the algorithm; the root represents matrix $A$, $B$ or $C$; the tree nodes represent submatrices of matrix $A$, $B$ or $C$. Figure 1(b) shows two kinds of tree operations:

![Fig. 1](image-url)  
**Fig. 1** Strassen’s recursion tree. $P$ represents parent matrices, $Q$ represents child matrices, $T$ represents temporary matrices.
positive operation and negative operation. For positive operation, we divide the “parent matrix” into four submatrices which we name as “decompose blocks”, and perform linear combinations of these submatrices to produce 7 “child matrices”. For negative operation, we use 7 “child matrices” to calculate their “parent matrix”. It should be noticed that in positive and negative operations, the \((i,j)\)th element of a decompose block is only added to or subtracted from the same \((i,j)\)th element of another decompose block.

As shown in Fig. 1 (c), we initially get matrix \(A\) and \(B\). Then we perform positive tree operations recursively to calculate all the nodes of tree \(A\) and \(B\). When the “leaf matrices” of tree \(A\) and \(B\) are reached, we multiply them pair wise to get the “leaf matrices” of tree \(C\). These “leaf matrices” are then used to perform negative tree operations to finally get the output matrix \(C\).

According to the above descriptions, we categorize the operations of Strassen’s algorithm into two kinds: recursive additions/subtractions and one time multiplications of the “leaf matrices”. In our algorithm, we turn recursive additions/subtractions into iterative pattern and combine all the positive operations in one single round as well as the negative operations. As a result, the algorithm can be divided into three rounds. In round 1, we perform iterative operations of matrix \(A\) and \(B\) locally; in round 2, we do matrix multiplications of the “leaf matrices”; and in round 3, we do iterative operations of matrix \(C\) locally. All the data transfers between these three rounds are achieved by two shuffle operations provided by MapReduce-style frameworks. Through this simple implementation, we can express Strassen’s algorithm in a fairly easy way. The number of rounds is also fixed. A simplified version of pseudocode appears as Algorithm 1 (for more details, see Table 2). In Sect. 3.2, we detail the way to do local computations; in Sect. 3.3, we show our task parallel strategy; in Sect. 3.4, we detail the way to perform iterative operations and in Sect. 3.5, we show how to settle all the data transfers in two total shuffles. We will prove that our algorithm is topology-oblivious and communication-optimal in later sections. As a result, we consider it cloud-friendly.

### 3.2 Data Parallel Strategy

To do iterative additions/subtractions of matrices \(A\), \(B\) and \(C\) locally and in balance, the data layout of the matrices satisfies the following three properties:

1. In each recursion level, the data layouts of “decompose blocks” of \(A\), \(B\) and \(C\) must match among compute nodes.
2. Each of the “decompose blocks” must be distributed equally among compute nodes for load balancing.
3. The \((i,j)\)th element and \((i + nk/2^r, j + nm/2^r)\)th element in \(A\) or \(B\) should reside in the same compute node, \(i, j = 0, 1, \ldots, n/2^r - 1, k, m = 1, 2, \ldots, 2^r - 1\).

As mentioned in Sect. 3.1, during positive or negative operations, the \((i,j)\)th element of a decompose block is only

| Terms | Meanings |
|-------|----------|
| \(n\) | matrix dimension |
| \(R\) | recursion level of Strassen’s algorithm |
| \(M\) | maximum memory size user codes can use |
| \(p\) | task number of round 2 |
| \(p^*\) | task number of round 1 |
| \(p''\) | task number of round 3 |
| decompose block | four submatrices derived from parent matrices to generate 7 child matrices in each recursion level |
| basic block | decompose block in the last recursion level |
| \(q\) | the segment number that the row and column of each basic block are divided into. As a result there are \(q^2\) sub-blocks in each basic block. |
| corresponding element | corresponding sub-blocks among all the basic blocks |
| \(P\) | parent matrices |
| \(Q\) | child matrices |
| \(T\) | temporary matrices |

### Table 1  Glossary.

| Steps | Actions | Key Transformations |
|-------|---------|--------------------|
| 1 | Determine parameter \(p\) and the block length of block-cyclic layout according to memory size of the compute node and physical processor numbers. | [column index, dense vector of \(A\) or \(B\)] |
| 2 | Do repartitions of \(A\) and \(B\) to aggregate corresponding elements of each “basic block”, and organize them in Morton order according to \(i\). | [column index, dense vector of \(A\) or \(B\)] \(\rightarrow\) [\(i, (j, submatrices \text{ of } A \text{ and } B)\)] |
| 3 | Do iterative addition/subtraction operations to compute 7 “leaf matrices” locally. | [\(i, (j, submatrices \text{ of } A \text{ and } B)\)] \(\rightarrow\) [\(i, (k, leaf \text{ matrices \text{ of } A \text{ and } B})\)] |
| 4 | Perform total shuffle using \(k\) as the key. | [\(k, (i, leaf \text{ matrices \text{ of } A \text{ and } B})\)] \(\rightarrow\) [\(k, (i, leaf \text{ matrices \text{ of } C})\)] |
| 5 | Do iterative addition/subtraction operations to compute \(C\) locally. | [\(k, (i, leaf \text{ matrices \text{ of } C})\)] \(\rightarrow\) [\(i, (j, submatrices \text{ of } C)\)] |
| 6 | Perform total shuffle using \(i\) as the key. | [\(k, (i, leaf \text{ matrices \text{ of } C})\)] \(\rightarrow\) [\(i, (k, leaf \text{ matrices \text{ of } C})\)] |
| 7 | Do iterative addition/subtraction operations to compute \(C\) locally. | [\(i, (j, submatrices \text{ of } C)\)] \(\rightarrow\) [column index, dense vector of \(C\)] |

### Algorithm 1: MR-Strassen

**Input**: \(A\), \(B\), \(n\), where \(A\) and \(B\) are two \(n \times n\) matrices

**Output**: \(C = A \times B\)

**Procedure**

1. do iterative additions/subtractions of \(A\) and \(B\) locally \(\Rightarrow\) round 1
2. total shuffle of the “leaf matrices” of \(A\) and \(B\) \(\Rightarrow\) round 2
3. local matrix multiplications of the “leaf matrices” of \(A\) and \(B\) \(\Rightarrow\) round 3
4. total shuffle of the “leaf matrices” of \(C\) \(\Rightarrow\) round 4
5. do iterative additions/subtractions of its “leaf matrices” to get \(C\) \(\Rightarrow\) round 5

End
added to or subtracted from the same \((i, j)\)th element of another decompose block. As we turn recursive operations into iterative pattern and finish them in one round, the \((i, j)\)th element of the original matrix \(A\) or \(B\) is only operated with the \((i + nk/2^R, j + nm/2^R)\)th element when considered iteratively. Block-cyclic data layout introduced by Q. Luo and J.B. Drake [4] just satisfies this property. For recursion level \(R = 2\) in Fig. 2, \(k = 1, 2, 3, m = 1, 2, 3\), the whole matrix is divided into 16 basic blocks. As shown in Fig. 2, corresponding elements with deep color should be located in the same compute node. The block length selection strategy of block-cyclic data layout will be discussed in the next section.

### 3.3 Task Parallel Strategy

In MapReduce-style frameworks, task numbers determine the parallel degree of the algorithm, physical processor numbers determine the parallel degree that can be actually achieved. Task numbers impact the performance from three perspectives:

1. It determines the data split size each task should process, and thus determines the memory usage of each task. In MapReduce-style frameworks, tasks are executed by the underlying java virtual machine (JVM). Memory usage affects JVM’s garbage collection (GC) frequency, which has significant influence on the overall performance [31].
2. Its ratio to physical processor numbers decides whether we can fully utilize actual parallelism without bringing too much overhead.
3. It has a close relationship to the communication cost. More tasks lead to higher parallel degree but in turn bring more communication cost.

In our implementation, we adopt different parallel strategies for each round. Suppose \(A\) and \(B\) are square matrices of dimension \(n\). Let \(R\) be the recursion level of Strassen’s algorithm.

For matrix multiplication in round 2, \(R\) levels of recursion lead to \(7^R\) matrix multiplications of the “leaf matrices”. We treat each matrix multiplication as a single task. As a result, there are totally \(7^R\) tasks. Let \(p\) be the task number of round 2, \(p = 7^R\). After \(R\) recursion levels, matrix size is reduced by a factor of \((1/4)^R\). We assume that no extra memory is required for local multiplications. Thus, memory usage of each task is given by

\[
Mem_{\text{round}2}(n, p) = \frac{3n^2(1^R)}{4} = \frac{3n^2}{p^{\log_4 7}} \quad (1)
\]

For iterative additions/subtractions in round 1, memory usage increases by a factor of \(7/4\) in each recursion level. We adopt data level parallelism, separating “basic blocks” into equal partitions to meet memory requirements. As shown in Fig. 2, we divide the row and column of each “basic blocks” equally into \(q\) segments to form \(q^2\) elements. Corresponding elements of each “basic block” are stored in the same compute node, and operated by the same task to do iterative additions/subtractions. Thus, there are totally \(2q^2\) tasks for matrices \(A\) and \(B\). Let \(p'\) be task numbers of round 1, \(p' = 2q^2\). For inplace operations, memory usage of each task is given by

\[
Mem_{\text{round}1}(n, p') = \frac{2n^2}{p'} \left(\frac{1}{4}\right)^R = \frac{2n^2}{p'} \frac{p}{p^{\log_4 4} p'} \quad (2)
\]

Round 3 is just an inverse of round 1. As we only operate on matrix \(C\), memory usage of each task is given by

\[
Mem_{\text{round}3}(n, p'') = \frac{n^2}{p''} \left(\frac{1}{4}\right)^R = \frac{n^2}{p''} \frac{p}{p^{\log_4 4} p''} \quad (3)
\]

Parameter settings of \(p, p', p''\) are nontrivial in MapReduce-style frameworks. To achieve best performance, not only framework configuration parameters such as physical processor numbers, memory size of each compute node, shuffle buffer size, but also JVM GC performance should be considered. Moreover, some parameters correlate with each other [32] and the number of tasks in one round may affect the performance of the next round [33]. There exist a body of work focusing on automatic MapReduce and black-box optimizations which relies on profiling and specific performance models to search best parameter settings through the whole parameter space [32]–[34].

Auto tuning is not within the scope of this paper, instead we use a simple manual tuning method to achieve best performance under certain constraints. Considering parameter \(p, p', p''\) as a parameter space, we search the best parameters on the basis of the following rule:

**Tuning rule:** task numbers should be as small as possible when physical parallel degree is achieved. Fewer tasks means less overhead and communication cost.

Unfortunately, fewer tasks also means more memory cost per compute node. When dealing with large matrices, memory is easily to become a bottleneck. Just achieving physical parallel degree is not enough, easily leading to
OutOfMemory error. In such a situation, our implementation is memory-bound. Suppose $M$ to be the maximum memory size we can use. All the three rounds fully utilize memory of size $M$ to reduce task numbers in their own round, resulting equal memory usage among these rounds. With equal memory usage in each round, relations among $p$, $p'$, $p''$ are fixed and the search space is also shrunk.

According to Eqs. (1), (2), (3), we just need to set $p' = \frac{2}{3}p$ and $p'' = \frac{1}{3}p$ to make memory usage equal. Consequently, $q = \lceil \sqrt{p'/2} \rceil = \lceil \sqrt{p/3} \rceil$ and the block length of block-cyclic layout is

$$
n = \frac{n}{2^kq} = \frac{n}{2^{k\lceil \sqrt{p/3} \rceil}} = \frac{n}{2^{\log_2 p \lceil \sqrt{p/3} \rceil}}$$

We notice that all the parameters can be expressed by $p$, once $p$ is determined, all the other parameters are determined. Due to large space of framework configurations and memory management mechanism of JVM, the actual memory usage is hard to estimate [31]. We need to tune parameter $p$ to make task numbers as close to physical parallel degree as possible and to meet the actual memory requirements at the same time. Fortunately, as $p$ increases by a factor of 7, there exist only a few choices to tune. Details of this part are described in Sect. 5, and we will show that the rule we follow is consistent with experiments. Overall, the parallel strategies described above organize Strassen’s algorithm in a flat structure, taking no dependency on any special topologies. Thus we regard it as topology-oblivious.

### 3.4 Iterative Additions/Subtractions

To save memory usage in rounds 1 and 3, we do inplace operations for iterative additions/subtractions. We directly allocate a memory with the required size of the last recursion level and do all levels of iterative operations on it. The process details of matrix $A$ are shown in Fig. 3. Matrix $B$ can be operated in the same way.

In recursion level 0, corresponding elements of each “basic block” are organized in Morton [35] order at the bottom of the memory. High address area is not used yet. In recursion level 1, we treat small, deep color “z”s as the “decompose blocks” to compute 7 child matrices $Q_0$-$Q_6$. In recursion level 2, $Q_0$-$Q_6$ are used to compute 7 child matrices each, selecting blocks with numeric labels as “decompose blocks”. They are processed in an inverse order to prevent memory overriding. Due to Morton order organization, additions/subtractions can always be performed in a way similar to continuous memory copy. After $R$ levels of such operations, we complete round 1, filling up the memory with $7^R$ “leaf matrices”. Round 3 is just an inverse process of round 1.

### 3.5 Key Management and Shuffle Operations

We use a square bracket to denote a key-value pair, and use a round bracket to denote the value. The index of corresponding element is denoted by $i$, the index of basic block is denoted by $j$, the index of leaf matrix is denoted by $k$. Suppose input matrices $A$ and $B$ are square matrices of dimension $n$. They are stored in column major order as <column index, dense vector>. Table 2 shows the execution flow of MR-Strassen as well as details of key management and shuffle operations.

### 4. Communication Model

In MapReduce-style frameworks, computations are divided into basic rounds. In each round, processors receive input at the beginning, do some computations independently and write output to storage at the end. We define communication complexity similar to [36]. Let $L$ be the number of rounds an algorithm uses. We define $BW_i$, the communication complexity of round $i$, to be the total size of inputs and outputs in round $i$, and the communication complexity of the whole algorithm, $BW$, to be the sum of communication cost in each round.

$$BW = \sum_{i=1}^{L} BW_i$$

We model MapReduce-style systems as $MR(\alpha, \beta, \gamma)$, where $\alpha$ denotes the constant overhead brought by each round. It is mainly the cost to set up communication during shuffle phase. $\beta$ denotes the transfer ability of the network. $\beta$ is defined such that it takes $\beta h$ time for $h$ words of data to complete shuffling under continuous traffic conditions. $\gamma$ denotes the FLOPS of the system.

We make the following assumptions for our model:

1. For algorithms with good load balancing, all the compute nodes process synchronously. They do communication or computation at the same time.
2. For each compute node, communication and computation cannot be overlapped.

Suppose an algorithm has $BW_i$ to transfer and $F_i$ float point operations to perform in round $i$. Running time of round $i$ is given by

$$\alpha + \beta BW_i + \gamma F_i$$

Total running time of the entire algorithm is given by
Computation cost of the algorithm is defined as $F$. Our algorithm has a factor of

Table 3 compares MR-Strassen with the Classical-2D algorithm. As a result, total running time is given by

$$T = \sum_{i=1}^{L} [\alpha + \beta BW_i + \gamma F_i] = \alpha L + \beta BW + \gamma F$$

Computation cost of the algorithm is defined as $F = \sum_{i=1}^{L} F_i$.

For our implementation, only "leaf matrices" of matrix $A, B$ and $C$ need to be shuffled. In the last recursion level, "leaf matrices" increase by a factor of $(7/4)^R$. Besides, matrix $A, B$ and $C$ may be repartitioned to meet input, output formats. Thus, computation cost is given by

$$BW = 2n^2 + 2n^2(7/4)^R + n^3(7/4)^R + n^2$$

$$= 3n^2 + 3n^2 \frac{p}{p^2 + 0.069}$$

$$\approx 3n^2 + 3n^2 p^{0.288}$$

$\omega_0 = \log_2 7$. As $p, p', \omega''$ are proportional to each other, here we use $p$ to calculate average communication cost. It only affects the analysis by a constant factor.

$$BW_{avg} = \frac{BW}{p} = \frac{3n^2}{p} + \frac{3n^2}{p^2 + 0.069} \approx \Theta \left( \frac{n^2}{p^{2/0.069}} \right)$$

The communication cost matches Strassen’s lower bound described in [11]. So we consider MR-Strassen as communication-optimal.

For computation, we only consider multiplication operations. Impacts of additions/subtractions will be discussed in Sect. 5.3. Thus computation cost is given by

$$F = (7/8)^R n^3 = (7/8)^{log_2 p} n^3 \approx \frac{n^3}{p^{0.069}}$$

As a result, total running time is given by

$$T = 3\alpha + \beta \left( 3n^2 + 3n^2 p^{0.288} \right) + \gamma \frac{n^3}{p^{0.069}}$$

Table 3 compares MR-Strassen with the Classical-2D algorithm. Our algorithm has a factor of $p^{0.212}$ less communication and a factor of $p^{0.069}$ less computation than Classical-2D. Detailed performance comparisons will be shown in Sect. 5.

5. Performance Results

5.1 Experimental Setup

We have implemented MR-Strassen using Scala in Spark framework and benchmark it on Spark standalone clusters on Amazon EC2 [37]. We compared the performance of MR-Strassen with Classical-2D. The type of each cluster node is m1.large, with 2 Intel Xeon E5-2650 cores and 7.5GB of main memory. Spark version is v1.0.0, executor memory is set to 6GB. We use HDFS for storage, with 128MB block size. We use Breeze library to do local matrix multiplication, which internally calls optimized single-threaded OpenBLAS [38] for core linear algebra routines.

5.2 Breakdown of Execution Time

Figure 4 shows the breakdown of execution time for both MR-Strassen and Classical-2D with $p = 49$ and core = 16. For MR-Strassen, Fig.4 illustrates the execution time of three rounds and two shuffle phase which are described in Algorithm 1. For Classical-2D, we divide input matrix $A$ and $B$ into $k \times k$ blocks and implement the algorithm in two rounds. In round 1, each block is copied $k$ times using a transformation operation called FlatMap [27]. In round 2, corresponding blocks are gathered to do multiply-accumulate operations to get the output matrix $C$. For all the experiments below, we tune parameter $p, q$ for MR-Strassen and parameter $k$ for Classical-2D to make their parallel degree equal when comparing with each other.

5.3 Performance under Various Parallel Degrees

We tested the performance of MR-Strassen and Classical-2D under various parallel degrees with 8, 16, 32, 64 cores. The inputs are square matrices of fixed dimension 14112. For MR-Strassen, we set $p$ to 7, 49, 343, 2401, parameter $q$ is set to 2, 4, 7, 14 correspondingly. For Classical-2D, we set $k$ to 2, 7, 18, 49 to gain approximately equal parallel degrees. We discuss three factors that influence the performance of matrix multiplication.

**Communication:** Figure 5 (a) and Fig. 5 (b) show the communication percentage of MR-Strassen and Classical-2D under various parallel degrees, respectively. With the increment of parallelism, communication gradually becomes the bottleneck of the performance. As mentioned in Sect. 4, MR-Strassen is communication-optimal and has a factor of $p^{0.212}$ less communication than Classical-2D. As a result, communication advantage of MR-Strassen becomes more
significant when \( p \) increases from 7 to 2401. Figure 5(c) shows the ratio between communication time of MR-Strassen and communication time of Classical-2D and just proves this point.

**Computation:** we use less DGEMM time than Classical-2D. But with the increment of parallelism, the cost of extra additions gradually could not be ignored. Figure 5(d) shows the ratio between computation time of MR-Strassen and computation time of Classical-2D. When \( p = 2401 \), we spend more computation time for all core numbers. That’s because our addition codes are written in pure high level language without further optimization. By careful optimization, we believe costs of extra additions can be ignored finally.

**Best \( p/core \) Ratio:** as shown in Fig. 5(e), we achieve the best performances when \( p = 7 \) for \( core = 8 \), \( p = 49 \) for \( core = 16 \), 32 and \( p = 343 \) for \( core = 64 \). With parallel degree close to physical processor numbers, the experimental results are consistent to the tuning rule described in Sect. 3.3. For other cases, either the power of physical cores is not fully utilized (e.g., \( p = 7, core = 64 \) in Fig. 5(e)), or extra overheads are brought about by too high degree of parallelism (e.g., \( p = 2401, core = 8 \) in Fig. 5(e)). Currently, best \( p/core \) ratios are mainly obtained by experiments and experiences. Auto-tuning of this part is one of our main jobs in the future.

**Put them all together:** finally, we compare the overall performance between MR-Strassen and Classical-2D in Fig. 5(f). For various combinations of parallel degree and core number, a speedup ranging from 1.03× to 2.50× is achieved. For best \( p/core \) ratios, a speedup ranging from 1.23× to 1.31× is achieved.

### 5.4 Performance under Various Matrix Dimensions

In this part, performances of different matrix dimensions are presented. We tested input matrices of dimension 3528, 7056, 10584, 14112, 17640 and 21168 on clusters with 8, 16, 32, 64 cores. For all matrix dimensions and core numbers, \( p \) is manually tuned to achieve best performances. Results are shown in Fig. 6.

From Fig. 6(a) to Fig. 6(f), we notice that the line of execution time tends to be more linear with the increment of matrix dimensions. When parallelizing algorithms, communication computation ratio is an important factor we must take into account. For small matrix dimensions such as \( n = 3528, 7056, 10584 \), it is not feasible to increase core numbers from 32 to 64, which brings more communication cost and leaves each core less work to do. Improper communication and computation ratio finally leads to performance degradations.

From Fig. 6(a) to Fig. 6(f), we also notice that MR-Strassen gradually outperforms Classical-2D with the increment of matrix dimensions. When matrix dimension is small, advantages of MR-Strassen are counteracted by more complex implementation than Classical-2D (e.g., \( n = 3528 \)). When \( n \geq 10584 \), our algorithm shows better performances over Classical-2D.
From Fig. 6 (e) and Fig. 6 (f), we notice that for large matrix dimensions (e.g., \( n = 17640, 21168 \)), the advantage of MR-Strassen becomes more significant with the increment of cluster size. That’s because for large cluster size, parallel degree should be increased correspondingly to achieve best performance. High parallel degree leads to more communication advantages over Classical-2D.

As a result, we consider our algorithm more competitive for large matrix dimensions and cluster sizes.

6. Related Work

In HPC domain, the most widely used algorithm for matrix multiplication is SUMMA [6]. In 2012, Ballard [10] proposed a communication-optimal Strassen’s algorithm which performs better than any previous classical or Strassen-based parallel algorithm for square matrix multiplication. Later, Demmel [39] proposed a communication-optimal recursive algorithm for rectangular matrix multiplication, applying communication-optimal theory to all matrix dimensions.

In MapReduce systems, Norstad [40] first proposed three strategies for classical \( \Theta(n^3) \) algorithm in 2009. Seo [19] presented an efficient matrix computation framework with MapReduce called HAMA. Recently, RongGu [20] presented a distributed matrix computation library built on top of Spark framework called Marlin.

Marlin achieved great performance improvements for rectangular matrix, but performs a little worse than classical-2D for square matrix multiplication.

7. Conclusions

In this paper, we have proposed a cloud-friendly communication-optimal implementation for Strassen’s matrix multiplication algorithm and achieved a speedup from 1.03\( \times \) to 2.50\( \times \) over the Classical-2D algorithm. To be cloud-friendly, we reorganize Strassen’s algorithm in an iterative pattern, making it easily expressed by cloud technologies such as Hadoop and Spark. A novel data/task parallel strategy is adopted to reduce communication cost. We hope the experiences we obtained are applicable to other complex scientific applications.

In future work, we plan to focus on three areas:

1. Fully study performance and scalability of our algorithm for large matrix dimensions and cluster sizes.
2. Realize auto-tuning of parameter \( p, q \) for different matrices and cluster sizes.
3. Design more efficient addition codes.

References

[1] M.H. Hayes, Statistical digital signal processing and modeling, John Wiley & Sons, 2009.
[2] J.D. Foley, A. Van Dam, S.K. Feiner, J.F. Hughes, and R.L. Phillips, Introduction to computer graphics, Addison-Wesley, Reading, 1994.

[3] D. Irony, S. Toledo, and A. Tiskin, “Communication lower bounds for distributed-memory matrix multiplication,” J. Parallel. Distrib. Com., vol.64, no.9, pp.1017–1026, 2004.

[4] Q. Luo and J.B. Drake, “A scalable parallel strassen’s matrix multiplication algorithm for distributed-memory computers,” in Proceedings of the 1995 ACM symposium on Applied computing, pp.221–226, ACM, 1995.

[5] Y. Yang and S. Kimura, “The optimal architecture design of two-dimension matrix multiplication jumping systolic array,” IEICE Trans. Fundamentals, vol.E91-A, no.4, pp.1101–1111, 2008.

[6] R.A. Van De Geijn and J. Watts, “Summa: Scalable universal matrix multiplication algorithm,” Concurrency-Practice and Experience, vol.9, no.4, pp.255–274, 1997.

[7] J. Deijang and S.G. Ziavras, “A super-programming technique for large sparse matrix multiplication on pc clusters,” IEICE Trans. Inf. & Syst., vol.E87-D, no.7, pp.1774–1781, July 2004.

[8] Q.-L. Kao and C.-R. Lee, “Design fast matrix algorithms on high-performance cloud platforms,” in Proceedings of the 2012 IEEE 4th International Conference on Cloud Computing and Science (CloudCom), pp.771–776, IEEE Computer Society, 2012.

[9] C. Baransehl, “Strassen’s communication-avoiding parallel matrix multiplication algorithm for all-port 2d torus networks,” in Parallel computing technologies, Lecture Notes in Computer Science, vol.7979, pp.1–12, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[10] G. Ballard, J. Demmel, O. Holtz, B. Lipshitz, and O. Schwartz, “Communication-optimal parallel algorithm for strassen’s matrix multiplication,” in Proceedings of the 24th ACM symposium on Parallelism in algorithms and architectures, pp.193–204, ACM, 2012.

[11] G. Ballard, J. Demmel, O. Holtz, B. Lipshitz, and O. Schwartz, “Strong scaling of matrix multiplication algorithms and memory-independent communication lower bounds,” arXiv preprint arXiv:1202.3177, 2012.

[12] B. Lipshitz, G. Ballard, J. Demmel, and O. Schwartz, “Communication-avoiding parallel strassen: Implementation and performance,” Proc. the International Conference on High Performance Computing, Networking, Storage and Analysis, p.101, IEEE Computer Society Press, 2012.

[13] M. Armbrust, O. Fox, R. Griffith, A.D. Joseph, Y. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, and I. Stoica, “M.: Above the clouds: A Berkeley view of cloud computing,” 2009.

[14] A. Gupta and D. Milojicic, “Evaluation of hpc applications on cloud,” in Open Cirrus Summit (OCS), 2011 Sixth, pp.22–26, IEEE, 2011.

[15] J. Ekanayake and G. Fox, “High performance parallel computing with clouds and cloud technologies,” in Cloud computing, Lecture Notes of the Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering, vol.34, pp.115–131, Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[16] J. He, S. Zhou, B. Kobler, D. Duffly, and T. McGlynn, “Case study for running hpc applications in public clouds,” in Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing, pp.395–401, ACM, 2010.

[17] J. Dean and S. Ghemawat, “Mapreduce: Simplified data processing on large clusters,” Commun. ACM, vol.51, no.1, pp.107–113, 2008.

[18] T. White, Hadoop: The definitive guide: The definitive guide, O’Reilly Media, 2009.

[19] M. Zaharia, M. Chowdhury, M.J. Franklin, S. Shenker, and I. Stoica, “Spark: Cluster computing with working sets,” in Proceedings of the 2nd USENIX conference on Hot topics in cloud computing, p.10, 2010.

[20] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M.J. Franklin, S. Shenker, and I. Stoica, “Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing,” Proc. 9th USENIX conference on Networked Systems Design and Implementation, p.2, USENIX Association, 2012.

[21] L.G. Valiant, “A bridging model for parallel computation,” Commun. ACM, vol.33, no.8, pp.103–111, 1990.

[22] Y. Strassen, “Gaussian elimination is not optimal,” Numer. Math., vol.13, no.4, pp.354–356, 1969.

[23] S. Winograd, “On multiplication of 2 × 2 matrices,” Linear. Algebra. Appl., vol.4, no.4, pp.381–388, 1971.

[24] L. Xu, J. Liu, and J. Wei, “Fmem: A fine-grained memory estimator for mapreduce jobs,” ICAC, pp.65–68, 2013.

[25] K. Wang, X. Lin, and W. Tang, “Predator—an experience guided configuration optimizer for hadoop mapreduce,” in Proceedings of the 2012 IEEE 4th International Conference on Cloud Computing Technology and Science (CloudCom), pp.419–426, IEEE Computer Society, 2012.

[26] Z. Zhang, L. Cherkasova, and B.T. Loo, “Getting more for less in optimized mapreduce workflows,” Integrated Network Management (IM), 2013 IFIP/IEEE International Symposium on, pp.93–100, IEEE, 2013.

[27] H. Herodotou and S. Babu, “Profiling, what-if analysis, and cost-based optimization of mapreduce programs,” Proc. VLDB Endowment, vol.4, no.11, pp.1111–1122, 2011.

[28] S. Chatterjee, A.R. Lebeck, P.K. Patnala, and M. Thottethodi, “Recursive array layouts and fast matrix multiplication,” IEEE Trans. Parallel Distrib. Syst., vol.13, no.11, pp.1105–1123, 2002.

[29] M.T. Goodrich, N. Sitchinava, and Q. Zhang, “Searching, simulation and in the mapreduce framework,” in Algorithms and computation, Lecture Notes in Computer Science, vol.7074, pp.374–383, Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[30] “Amazon elastic compute cloud (amazon ec2),” http://aws.amazon.com/ec2/, accessed May 9, 2014.

[31] Z. Xianyi, W. Qian, and Z. Chothia, “Openblas,” https://github.com/xianyi/OpenBLAS, accessed May 12, 2014.

[32] J. Denmel, D. Eliahu, A. Fox, S. Kamil, B. Lipshitz, O. Schwartz, and O. Spillinger, “Communication-optimal parallel recursive rectangular matrix multiplication,” in Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on, pp.261–272, IEEE, 2013.

[33] J. Norstad, “A mapreduce algorithm for matrix multiplication,” http://www.norstad.org/matrix-multiply/index.html, accessed Feb. 12 2014.
Jie Zhou was born in Jiaxing city, Zhejiang province, China in 1987. He received the B.S. degree in the Department of Instrumentation Science & Technology from Zhejiang University, Hangzhou, China. Now he is pursuing the Ph.D. degree in Instrumentation Science & Technology Department, Zhejiang University. He majors in signal processing and parallel and distributed computing.

Feng Yu is presently a professor of Instrumentation Science & Technology Department, Zhejiang University, Hangzhou, China. He has been engaged in researches on high speed signal processing system and embedded communication system.