Marlin: Efficient Large-Scale Distributed Matrix Computation with Spark

顾荣、唐云
南京大学 PASA大数据实验室

About Me

- Ph.D Student in Nanjing University
  - Homepage Link
  - Contact: gurongwalker@gmail.com
  - Github:
    - https://github.com/RongGu
    - https://github.com/PasaLab

Outline

- **Background & Related Work**
- Overview of Marlin
- Distributed Matrix Multiplication on Spark
- Evaluation
- Conclusion

Background

- Matrix computation is the core of many massive data-intensive scientific applications[1]
  - Such as large-scale numerical analysis, data mining, and computational physics

- In the Big Data era, as the scale of the matrix grows, traditional single-node systems can hardly solve the problem

1. G. Stewart, "The decompositional approach to matrix computation," Computing in Science & Engineering, vol. 2, no. 1, pp. 50–59, 2000.

Parallel Matrix Multiplication Algorithms

- Grid-based approach
  - regard processors as residing on a two- or three-dimensional grid.
  - The computation is iterative with several rounds.
  - For example, SUMMA[1](the most widely-used parallel matrix multiplication algorithm).
  - achieve good performance on grid or torus-based topologies. May not perform as well in more general topologies.

1. 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.

Parallel Matrix Multiplication Algorithms

- BFS/DFS approach
  - view the processor layout as a hierarchy rather than a grid
  - based on sequential recursive algorithms.
  - For example, CARMA[1].

Algorithm 1 CARMA, in brief

Input: A is an $m \times k$ matrix, B is a $k \times n$ matrix
Output: $C = AB$ is $m \times n$

1: Split the largest of m, n, k in half, giving two subproblems
2: **if** Enough memory **then**
3: Solve the two problems recursively with a BFS
4: **else**
5: Solve the two problems recursively with a DFS

J. Demmel, D. Eliahu, A. Fox, S. Kamil, B. Lipshitz, O. Schwartz, and O. Spillinger, "Communication-optimal parallel recursive rectangular matrix multiplication," in IPDPS. IEEE, 2013, pp. 261–272.

Parallel Matrix Multiplication Algorithms

- Demmel J et al. have proved that SUMMA is only communication-optimal for certain matrix dimensions, while CARMA can minimize communication for all matrix dimensions cases.[1]

- The data layout requirement of CARMA is quite different from any existing linear algebra library and it cannot work with the widely-used linear algebra libraries well, CARMA is limited in practical use.

J. Demmel, D. Eliahu, A. Fox, S. Kamil, B. Lipshitz, O. Schwartz, and O. Spillinger, “Communication-optimal parallel recursive rectangular matrix multiplication,” in IPDPS. IEEE, 2013, pp. 261–272.

Related Work(in brief)

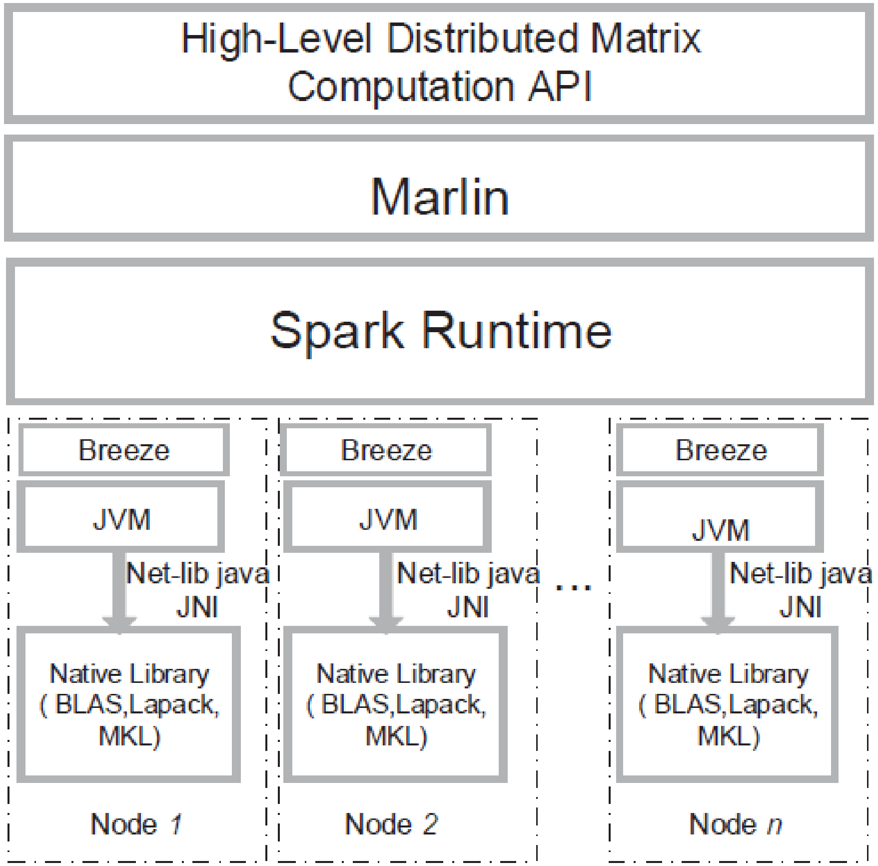
- ScaLAPACK[1] (MPI)
  - Fast, not easy to use, not good robustness.
- HAMA [2](MapReduce)
  - Easy to use, not efficient
- MadLINQ [3](Dryad)
  - Also DAG Execution, Not exploit efficient memory, Not open source. Is Dryad ended?

1. J. Choi, J. J. Dongarra, R. Pozo, and D. W. Walker, "Scalapack: A scalable linear algebra library for distributed memory concurrent computers," in Frontiers of Massively Parallel Computation, 1992., Fourth Symposium on the. IEEE, 1992, pp. 120–127.
2. S. Seo, E. J. Yoon, J. Kim, S. Jin, J.-S. Kim, and S. Maeng, "Hama: An efficient matrix computation with the mapreduce framework," in CloudCom. IEEE, 2010, pp. 721–726.
3. Z. Qian, X. Chen, N. Kang, M. Chen, Y. Yu, T. Moscibroda, and Z. Zhang, "Madlinq: large-scale distributed matrix computation for the cloud," in EuroSys. ACM, 2012, pp. 197–210.

Outline

- Background & Related Work
- **Overview of Marlin**
- Distributed Matrix Multiplication on Spark
- Evaluation
- Conclusion

The system stack of Marlin and its related systems



Features of Marlin

- Native Linear Algebra Library Acceleration
  - Marlin takes a divide-and-conquer strategy to deal with the large scale matrix computation.
  - For each sub-problem, instead of performing linear algebra computations on JVM, Marlin offloads the CPU-intensive operation from JVM to the native linear algebra library (e.g. BLAS,Lapack,MKL)

Features of Marlin

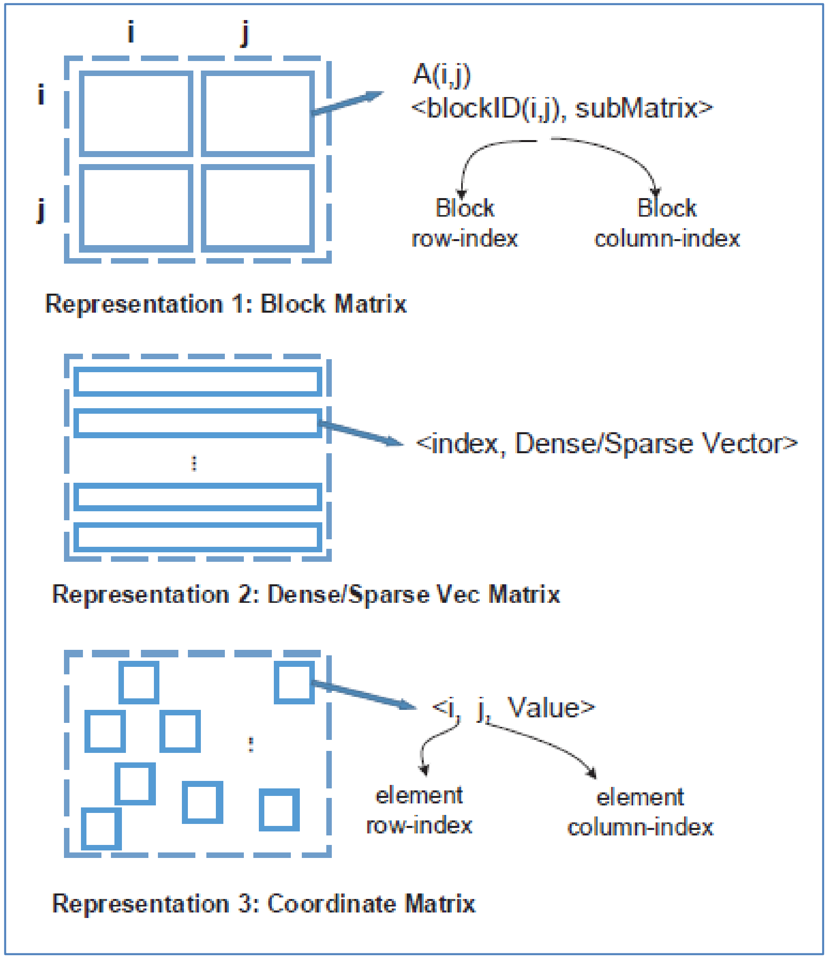
- Fine-grained Fault Tolerance and Ease to Use
  - achieves the fine-grained fault tolerance which is extended from Spark.
  - offers developers with high level matrix computation interfaces in Scala/Java which can accelerate the development of big data applications

- Efficient Distributed Matrix Operations
  - Has quite a few distributed matrix computing operations.
  - This talk focuses on distributed matrix multiplication

Outline

- Background & Related Work
- Overview of Marlin
- **Distributed Matrix Multiplication on Spark**
- Evaluation
- Conclusion

Representing Large Scale Matrices on Spark RDD

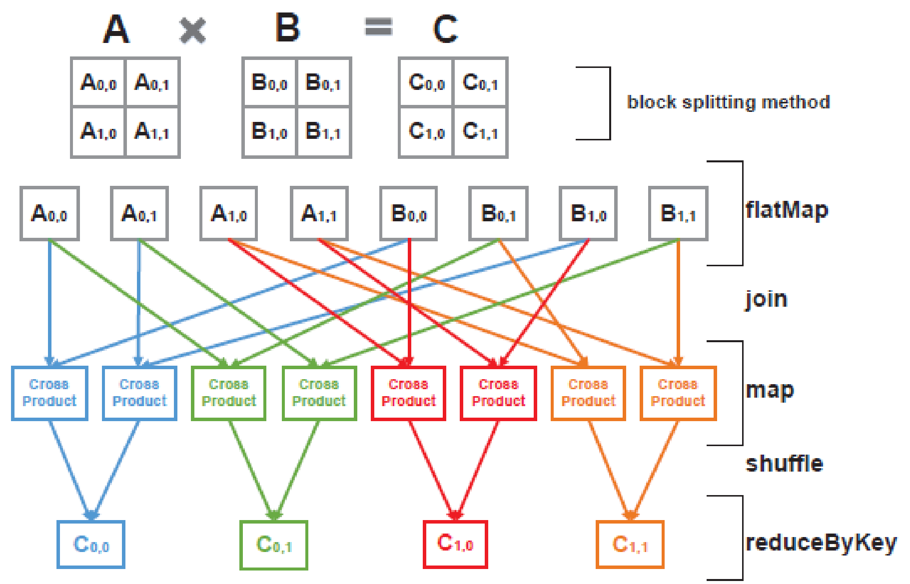


Distributed Matrix Multiplication in Marlin

- proposed three distributed matrix multiplication algorithms which are suitable for different situations.
- Based on this, we designed an adaptive model to choose the best approach for different problems.
- Instead of naively using Spark, we put forward some optimization methods.

Approach 1: Block-splitting matrix multiplication

- Similar to the blocking-approach in HAMA [4], Split two original matrices into blocked matrices and executes the multiplication of submatrices in parallel.
- This approach is suitable for multiplying two square matrices.



Approach 2: CARMA matrix multiplication

- When two input matrices are not square, the above dimension-splitting method is no longer suitable.

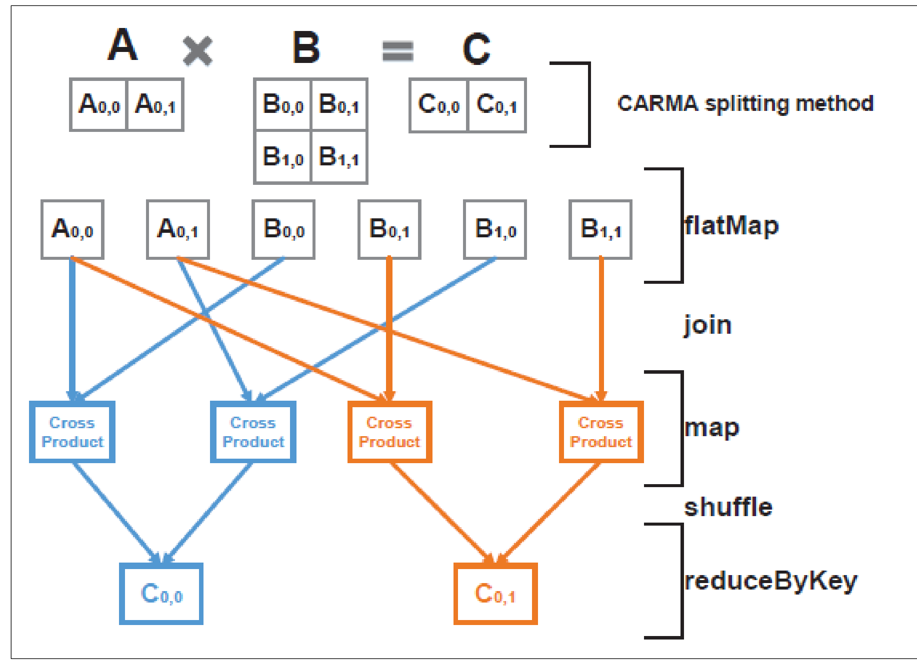
- To solve this problem, we refer to the equal representation of dimension-splitting in BFS steps of CARMA and design a dimension-splitting method similar to CARMA.

Approach 2: CARMA matrix multiplication

Algorithm 1: $CARMA(A, B, C, m, k, n, P)$ the splitting method

Input: A is an $m \times k$ matrix, B is a $k \times n$ matrix, P is the total core num of the cluster.
Output: $C = AB$ is $m \times n$

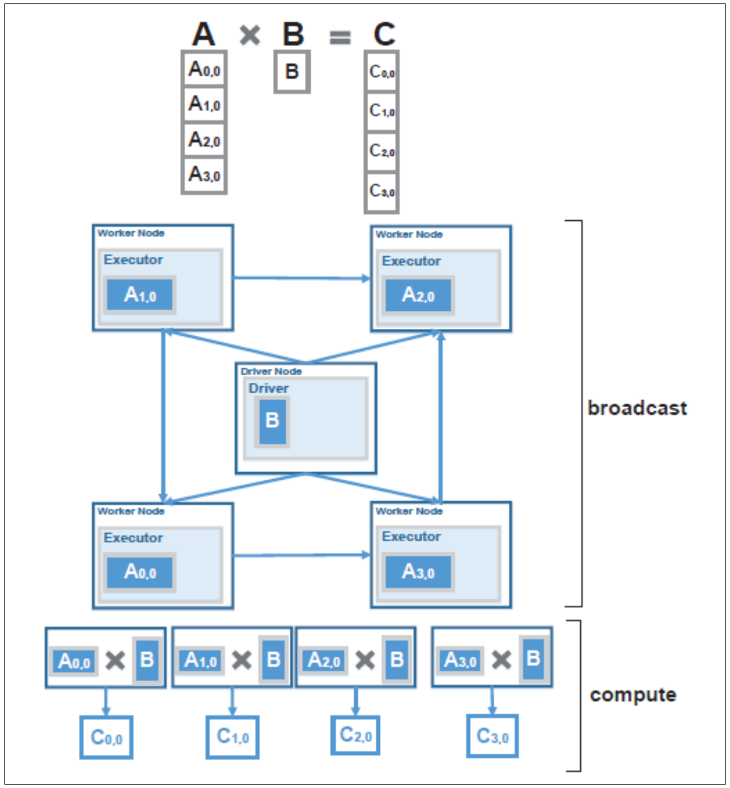
```
begin
    if P = 1 then
        return SequentialMultiply( A, B, C, m, k, n )
    if n is the largest dimension then
        Parallel do
            CARMA(A, B_left, C_left, m, k, n/2, P/2)
            CARMA(A, B_right, C_right, m, k, n/2,P/2)
    if m is the largest dimension then
        Parallel do
            CARMA(A_top, B, C_top, m/2, k, n, P/2)
            CARMA(A_bot, B, C_bot, m/2, k, n, P/2)
    if k is the largest dimension then
        Parallel do
            CARMA(A_left, B_top, C, m, k/2, n, P/2)
            CARMA(A_right, B_bot, C, m, k/2, n, P/2)
end
```



The workflow of CARMA approach matrix multiplication on Spark programming mode, here r = 1; s = 2; t = 2

Approach 3: Broadcast matrix multiplication

- If matrix B is quite small, broadcast it to each executor to avoid shuffling the large scale matrix A across network



Adaptive Approaches Selection

- Based on the time cost model analyzed above, we put forward an algorithm for selecting the appropriate matrix multiplication approach when given two distributed matrices.

Algorithm 2: $ApproachSelection(A, B, m, k, n, P)$

Input: A is an $m \times k$ matrix, B is a $k \times n$ matrix, P is the real num of cores across the cluster
Output: $C = AB$ is $m \times n$

begin
    if A or B is under broadcast threshold **then**
        C = BroadcastMultiply$(A, B, \lambda P)$
    else if m, k, n is close equal **then**
        C = BlockingMultiply(A, B, b)
    else
        C = CarmaMultiply$(A, B, 2P)$
    return C
end

Outline

- Background & Related Work
- Overview of Marlin
- Distributed Matrix Multiplication on Spark
- **Evaluation**
- Conclusion

Experimental Setup

- A local cluster with 17 nodes.

- Each node has two Xeon Quad 2.4 GHz processors altogether 16 logical cores, 24 GB memory and two 2 TB 7200 RPM SATA hard disks.

- The Marlin contains three matrix multiplication approaches. Thus, the block-splitting approach is denoted as ***Marlin-Blocking***, the CARMA approach is denoted as ***Marlin-CARMA***, and the broadcast approach is denoted as ***Marlin-Broadcast***.

Effects of Adopting Native Linear Algebra Library

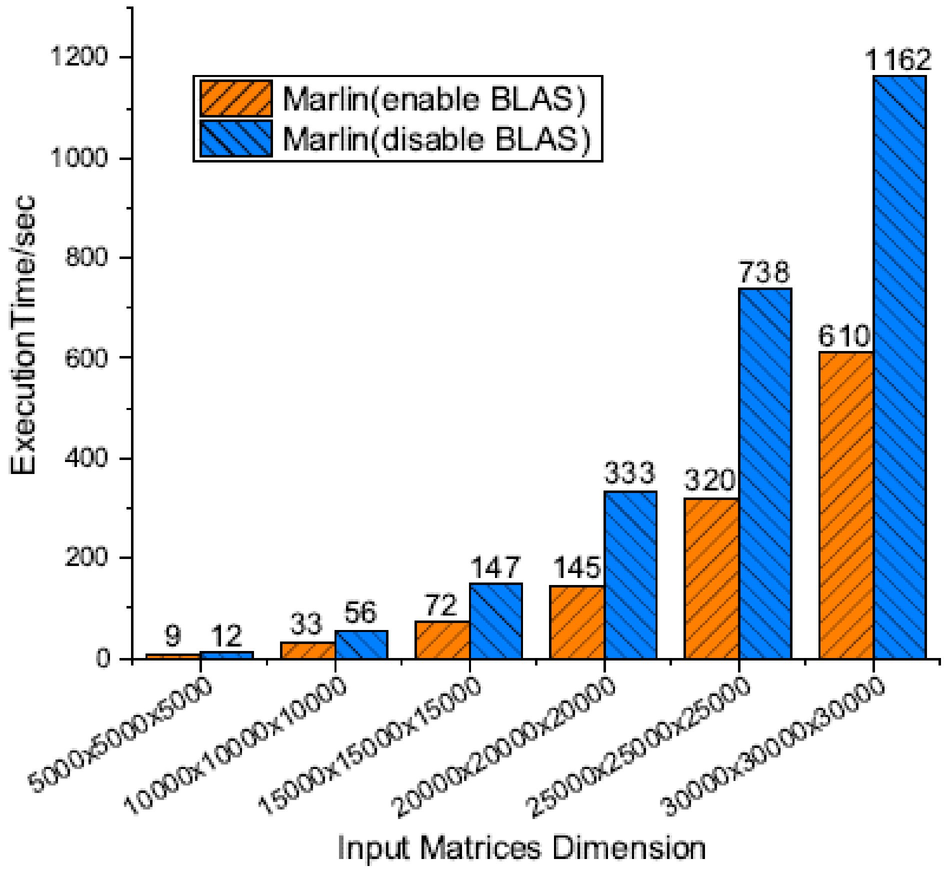


Figure 6. Performance comparison between Marlin enabling BLAS and disabling BLAS

Effects of Adaptive Approach Selection

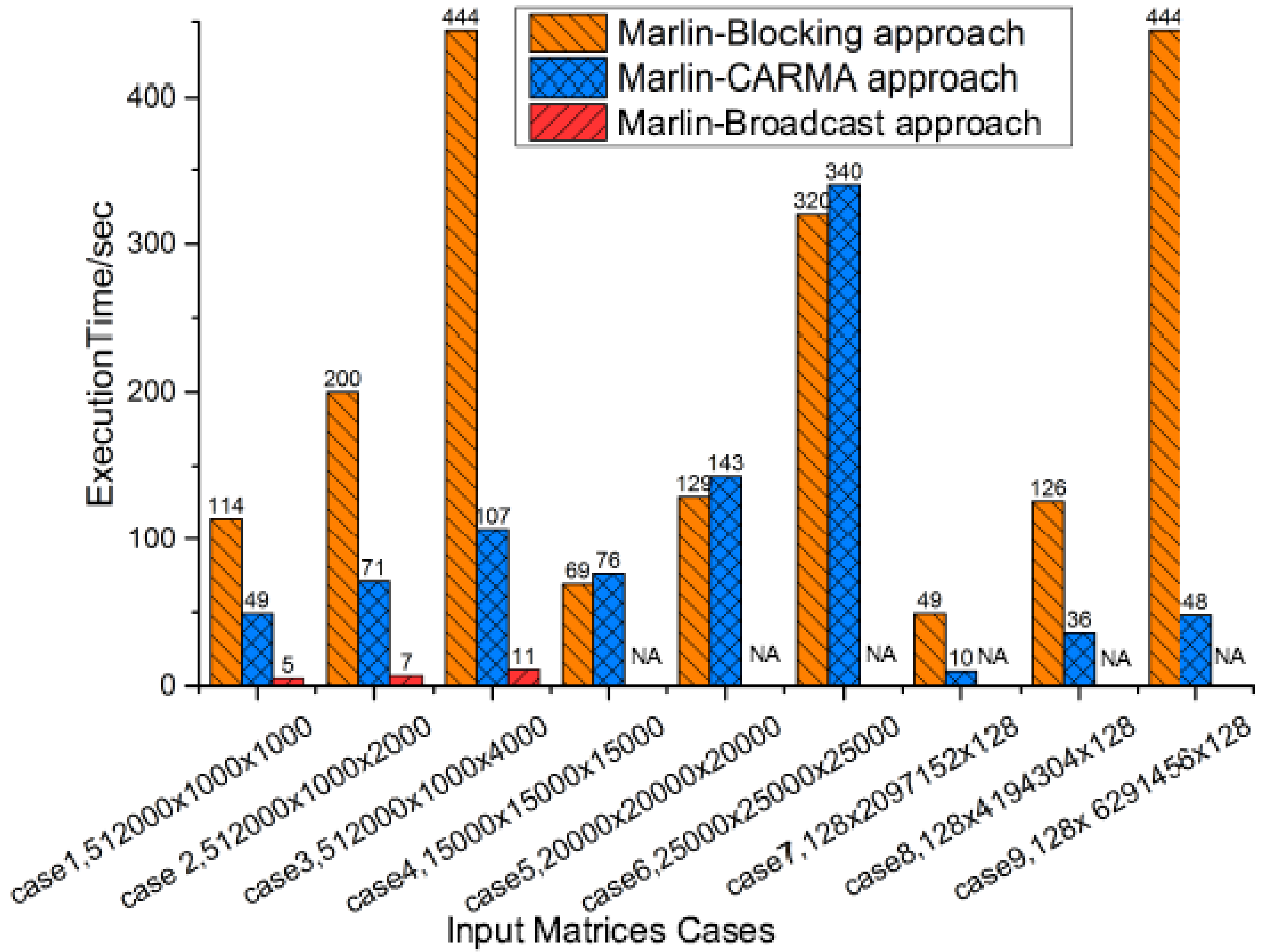


Figure 7. Performance comparison of three multiplication approaches in Marlin

Effects of Tuning the Matrix Split Granularity

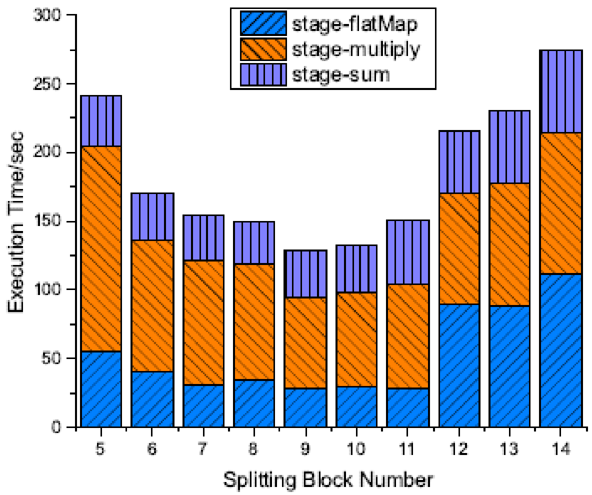


Figure 8. Execution time of two square matrices with different split granularity

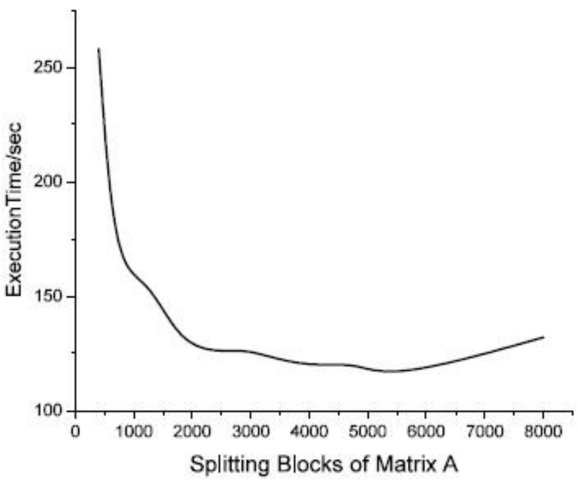


Figure 9. Execution time of two matrices with different split granularity, using broadcast-approach

Performance Comparison With Other Systems

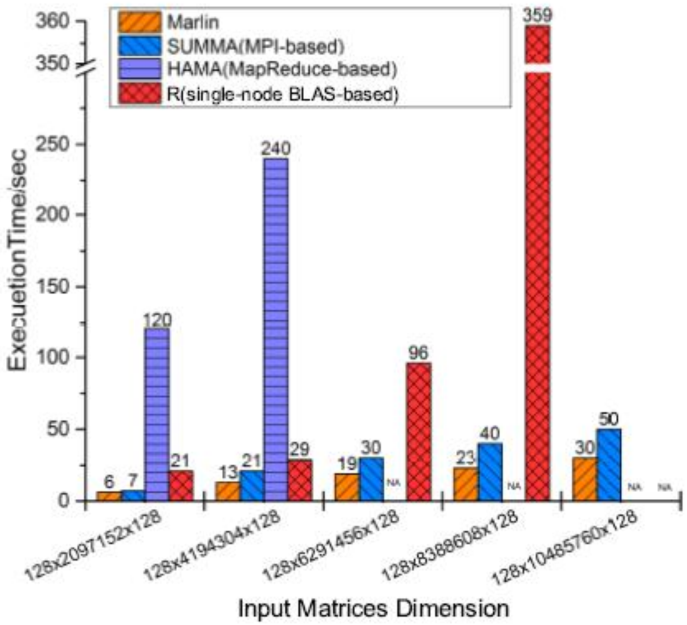


Figure 10. Performance comparison of the four systems, in the cases that two large-scale matrices with one large dimension

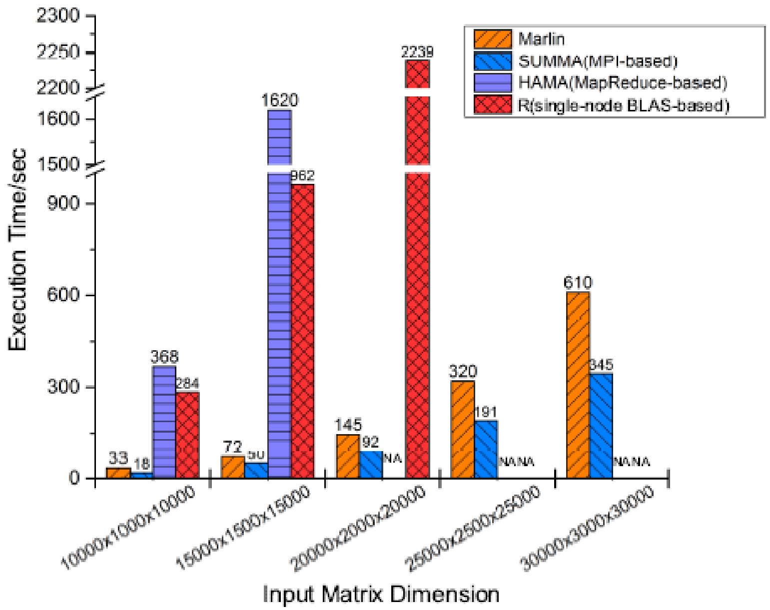


Figure 11. Performance comparison of the four systems, in the cases that two large-scale square matrices

Performance Comparison With Other Systems

Table I

PERFORMANCE COMPARISON OF THE FOUR SYSTEMS, IN THE CASES THAT ONE OF THE MATRICES IS NOT SO LARGE, MARLIN ADOPTS BROADCAST APPROACH (THE UNIT OF EXECUTION TIME IS SECOND).

Matrix dimension	Marlin	SUMMA	HAMA	R
512000x1000x1000	5	10.6	1250	148
1024000x1000x1000	10	20.3	3000	297
1536000x1000x1000	12	29	NA	906
2048000x1000x1000	13	39	NA	3302
2048000x1000x1000	16	79	NA	NA
65536x128x65536	7	7	NA	1163
81920x128x81920	8	9.7	NA	NA
98304x128x98304	10	15	NA	NA
114688x128x114688	13	17	NA	NA
131072x128x131072	16	21.4	NA	NA

Scalability Performance Analysis

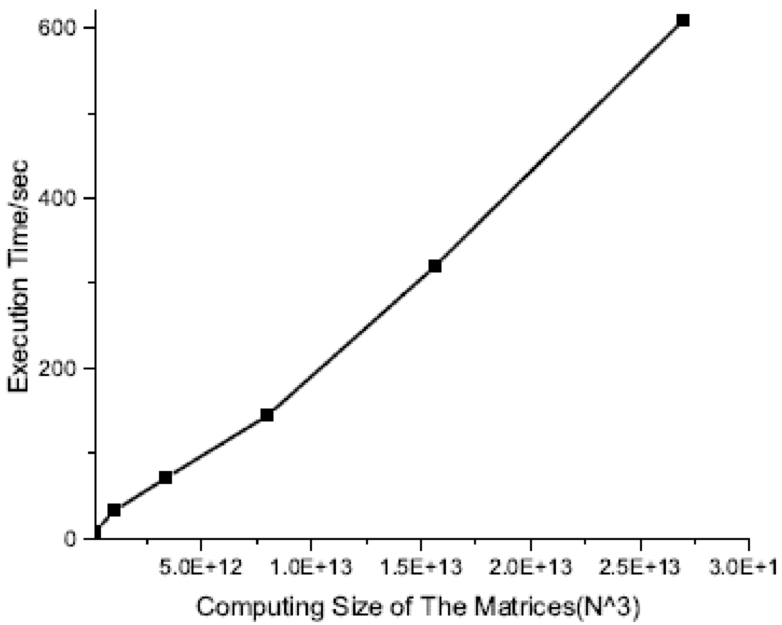


Figure 12. Data scalability

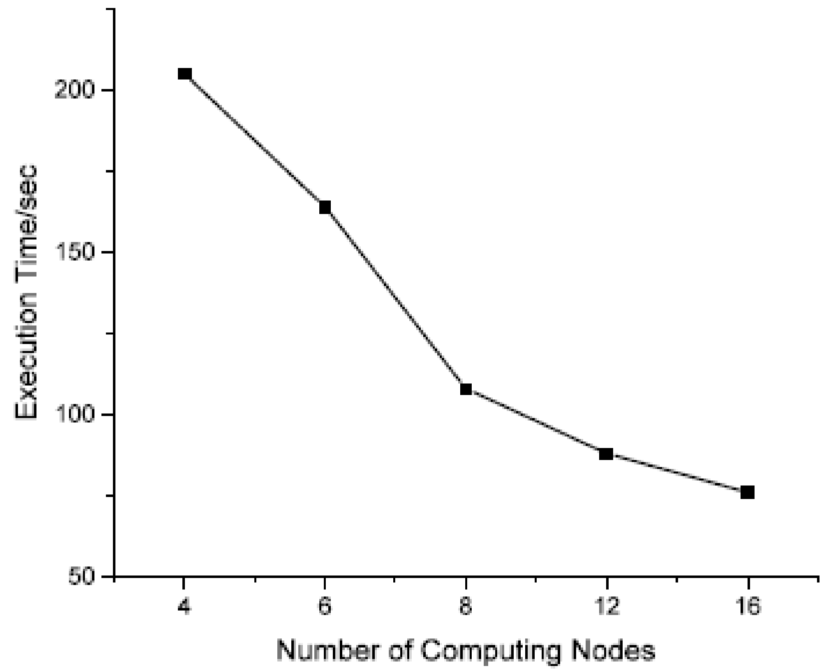
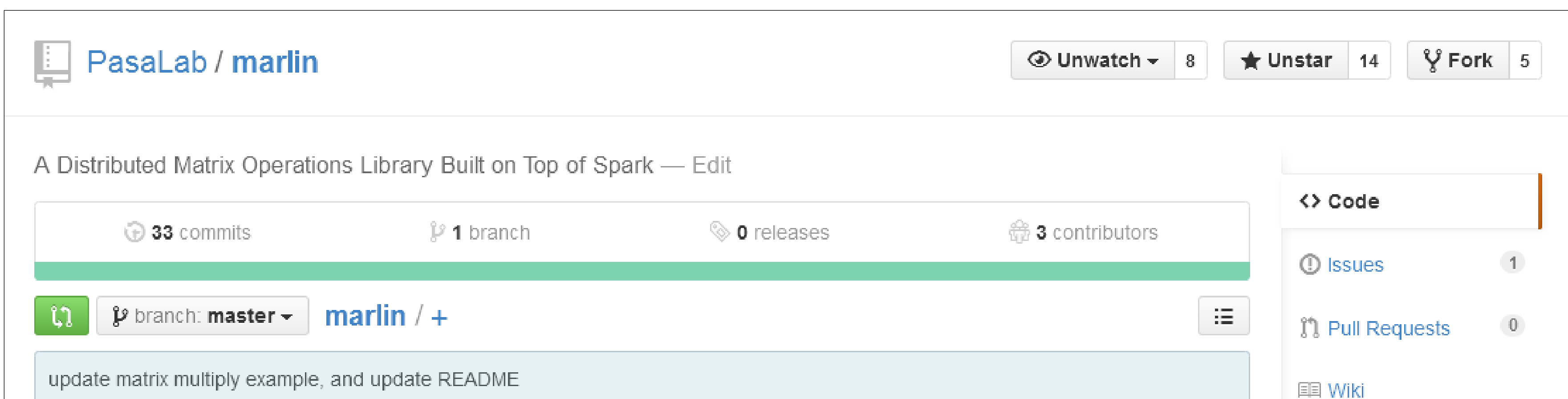


Figure 13. Node scalability

Outline

- Background & Related Work
- Overview of Marlin
- Distributed Matrix Multiplication on Spark
- Evaluation
- **Conclusion**

- We propose Marlin, an efficient distributed matrix computation library built on top of Spark. Three distributed matrix multiplication algorithms, suitable for different scenarios, are designed in Marlin. Also, an adaptive model is proposed to select the best matrix multiplication approach.
- Marlin is currently open-sourced at https://github.com/PasaLab/marlin



Thanks!

QA.

Backups

并行矩阵乘法概况

- 约定要进行的运算是 C = AB
  - A: m× k， B: k × n， C: m×n
- 算法计算复杂度
  - 经典算法 $O(mnk)$
  - 新的快速算法 $O(n^{2.977})$， 目前还没有并行实现
- 负载均衡（将计算平均平摊到各个处理器上）
- 通信复杂度*
  - P < d3/d2 — 1 large dimension — CARMA
  - d3/d2 < P < $(d2d3)/d1^2$ — 2 large dimensions — {2,3}D SUMMA/CARMA
  - $(d2d3)/d1^2$ < P — 3 large dimensions — 3D SUMMA/CARMA

能取到通信复杂度下限的算法

* d1,d2,d3分别是m,n,k三者中的最小，次大，最大者；更多细节见
J. Demmel, et. al ,“Communication-Optimal Parallel Recursive Rectangular Matrix Multiplication,” in 2013 IEEE 27th International Symposium on Parallel Distributed Processing (IPDPS), 2013, pp. 261–272.

2D SUMMA ——数据分布

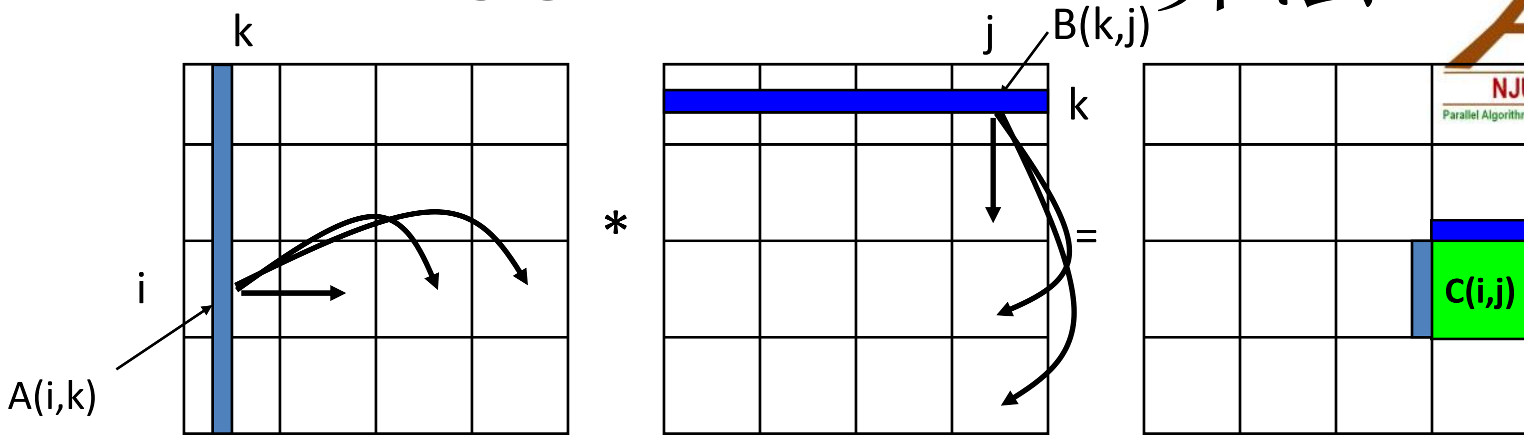
- 这是在PBLAS库中采用的算法
- 首先定义处理器网格
  - 假设有6个处理器
  - 分布在一个2×3的网格上
- 将A,B,C三个矩阵按照处理器网格指定的形式分布存储在这些处理器的内存里
  - 见右侧X矩阵的分块方案
  - X_{ij}分块将被分配给处理器P_{ij}

0	1	2
3	4	5

P_{12}

$$X = \begin{pmatrix} X_{00} & X_{01} & X_{02} \\ X_{10} & X_{11} & X_{12} \end{pmatrix}$$

2D SUMMA ——算法



For k=0 to n/b-1 … where b is the block size

… b = # cols in A(i,k) and # rows in B(k,j)

for all i = 1 to p_r … in parallel. in this example pr = 4

owner of A(i,k) broadcasts it to whole processor row

for all j = 1 to p_c … in parallel. in this example pc = 4

owner of B(k,j) broadcasts it to whole processor column

Receive A(i,k) into Acol

Receive B(k,j) into Brow

C_myproc = C_myproc + Acol * Brow

本页PPT来自 James Demmel ,Dense Linear Algebra:History and Structure, Parallel Matrix Multiplication

基于递归的CARMA算法

- 该算法是Network-Oblivious的，在任何情况下都能取到最优的通信复杂度下界
- 基于局部分块矩阵乘法
- 基于递归分治的思想
- 假设计算任务中m = 32, k = 8, n = 16，一共有8个处理器，见下例

CARMA算法

Algorithm 2 CARMA(A,B,C,m,k,n,P)

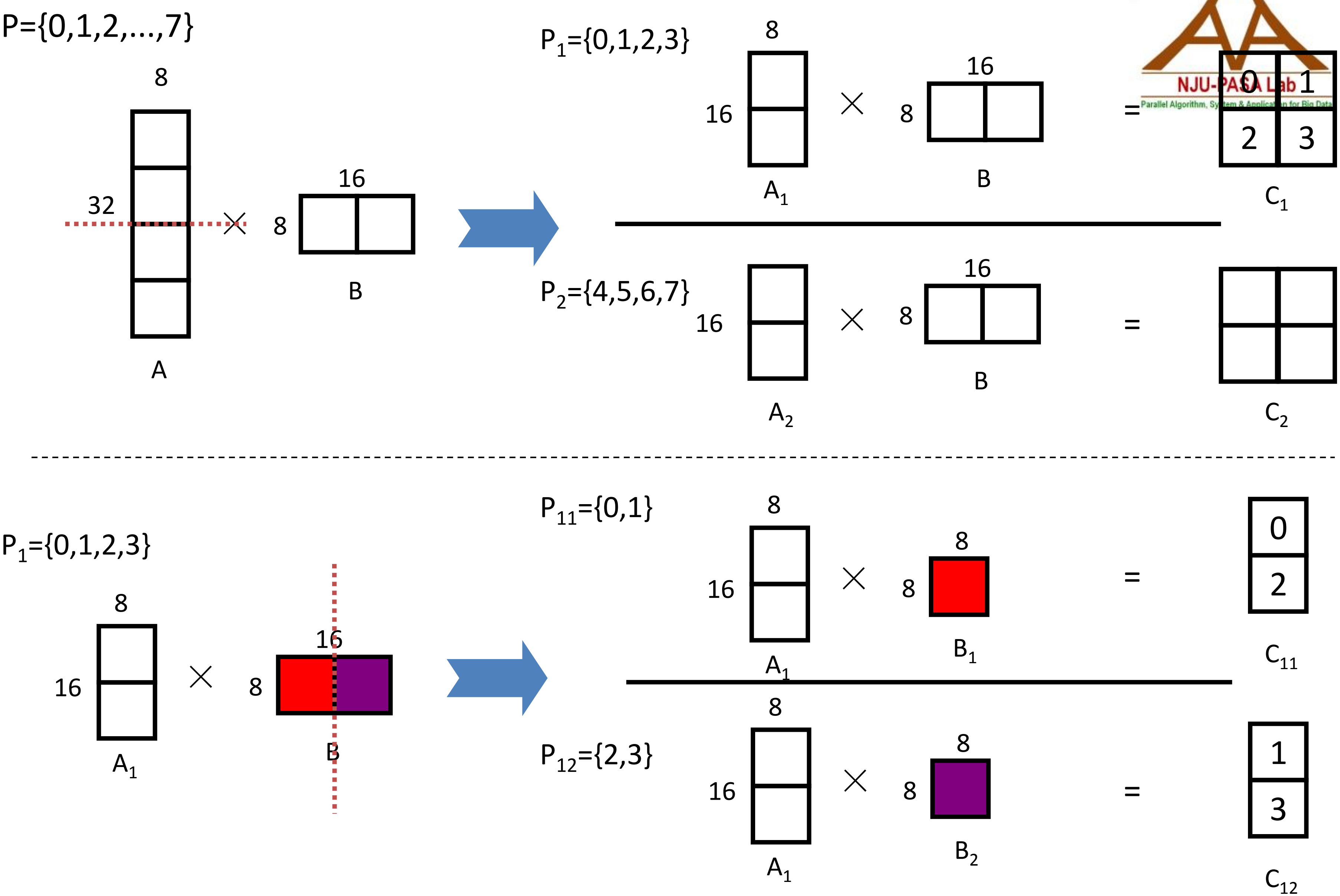
Input: A is an $m \times k$ matrix and B is a $k \times n$ matrix
Output: $C = AB$

- **if** $P = 1$ **then**
  - SequentialMultiply(A, B, C, m, k, n)
- **if** Enough Memory **then** ▷ Do a BFS
  - **if** n is the largest dimension **then**
    - Copy A to disjoint halves of the processors. Processor i sends and receives local A from processor $i \pm P/2$
    - **Parallel do**
      - CARMA(A, B_{left}, C_{left}, m, k, $n/2$, $P/2$)
      - CARMA(A, B_{right}, C_{right}, m, k, $n/2$, $P/2$)
  - **if** m is the largest dimension **then**
    - Copy B to disjoint halves of the processors. Processor i sends and receives local B from processor $i \pm P/2$
    - **Parallel do**
      - CARMA(A_{top}, B, C_{top}, $m/2$, k, n, $P/2$)
      - CARMA(A_{bot}, B, C_{bot}, $m/2$, k, n, $P/2$)
  - **if** k is the largest dimension **then**
    - **Parallel do**
      - CARMA(A_{left}, B_{top},C, m, $k/2$, n, $P/2$)
      - CARMA(A_{right}, B_{bot},C, m, $k/2$, n, $P/2$)
    - Gather C from disjoint halves of the processors. Processor i sends C and receives C' from processor $i \pm P/2$
    - $C \leftarrow C + C'$
- **else** ▷ Do a DFS
  - **if** n is the largest dimension **then**

更多细节见文献：

J. Demmel, D. Eliahu, A. Fox, S. Kamil, B. Lipshitz, O. Schwartz, and O. Spillinger, "Communication-Optimal Parallel Recursive Rectangular Matrix Multiplication," in 2013 IEEE 27th International Symposium on Parallel Distributed Processing (IPDPS), 2013, pp. 261–272.

CARMA算法示例



P={0,1,2,...,7}
8
32
A
16
8
B
P1={0,1,2,3}
8
16
A1
16
8
B
=
0
1
2
3
C1
P2={4,5,6,7}
16
A2
16
8
B
=
C2
P1={0,1,2,3}
8
16
A1
16
8
B
P11={0,1}
8
16
A1
8
8
B1
=
0
2
C11
P12={2,3}
8
16
A1
8
8
B2
=
1
3
C12

CARMA算法 —— 数据分解

- 基于递归的算法一大问题是：数据分解不具有连贯性
- 同一个矩阵在不同情况下有不同的分解方式

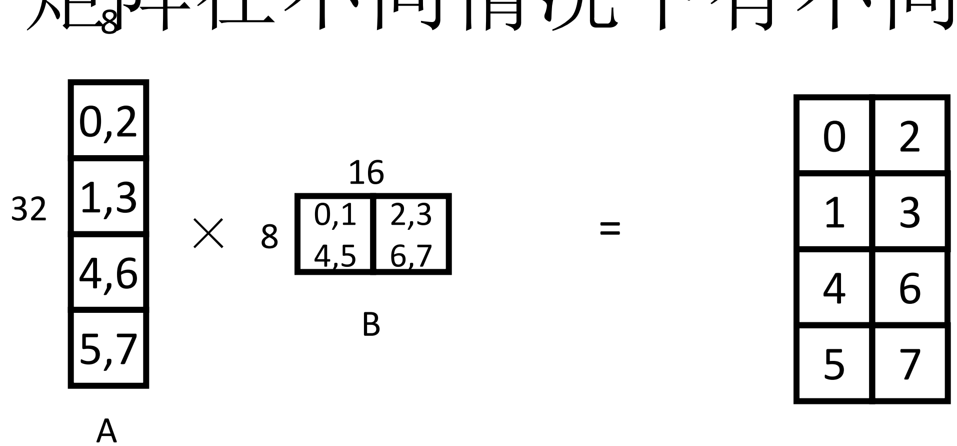
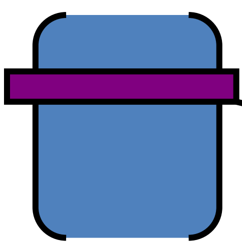


图 **CARMA**算法将数据分解到**8**个处理器上示例

HAMA的实现 —— 数据存储

- HAMA早期致力于实现Spark中MLLib所完成的功能，但2012年后专心做BSP框架
- 在HAMA的早期论文中，基于MapReduce框架完成了一个密集矩阵乘法的实现
- 它的矩阵是保存在HBase的表中



	column_vector
row index （行号）	elements in that row

HAMA的实现 —— 算法

- 利用的矩阵乘法计算公式
  - $C(i,j) = \Sigma_k A(i,k) * B(k,j)$
  - 该公式对于分块矩阵也是适用的，只要A矩阵在列方向的分块方式和B矩阵在行方向的分块方式相同
- 它使用两趟MapReduce Job实现乘法功能
  - 第一趟：由HBase中的存储表生成CollectionTable
  - 第二趟：从CollectionTable计算分块矩阵乘积，并汇合成整个结果矩阵C

HAMA的实现 —— 算法（续）

性能问题

生成CollectionTable的时候，大矩阵A和B需要完全地在网络上传输一次，而且传输的数据量是O(mnk/b)，b是分块矩阵的宽度

- Pass 1 : 由HBase中的表生成CollectionTable

```
CollectionTable:
                        matrix A        matrix B
-----------------------+-----------------------------
block(0, 0)-0          block(0, 0)    block(0, 0)
block(0, 0)-1          block(0, 1)    block(1, 0)
 ...                                   ...
block(i, j)-k          block(i, k)    block(k, j)
 ...                    ...
block(N-1, n-1)-(N^3-1)    block(N-1, N-1)  block(N-1, N-1)
```

- Pa
  –
    - 接收block(i,j)-k，进行分块矩阵乘A(i,?)*B(?,j)，发射C(i,j)的部分和C’(i,j)-k
  – Reducer:
    - 接受C(i,j)的部分和，进行累加求和C(i,j) = Σ_k C’ (i,j)-k
- 因为Spark支持表达MapReduce框架，因此我们现在先考虑使用HAMA的实现