
Distributed-memory Algorithms for Dense Matrices, Vectors, and Arrays

John Mellor-Crummey

**Department of Computer Science
Rice University**

johnmc@rice.edu

Topics for Today

Distributed-memory Algorithms

- **Matrix-vector multiplication**
 - 1D and 2D partitionings
- **Matrix-matrix multiplication**
 - 2D partitioning and Cannon's algorithm
 - 2.5D matrix multiplication
- **Exotic distributions for dense arrays**
 - generalized multipartitioning for line sweep computations

Matrix Algorithms

- Regular structure
- Well-suited to static partitioning of computation
- Computation typically partitioned based on the data
 - input, output, intermediate
- Typical data partitionings for matrix algorithms
 - 1D and 2D block, cyclic, block-cyclic
- Exotic partitionings with special properties
 - multipartitioning

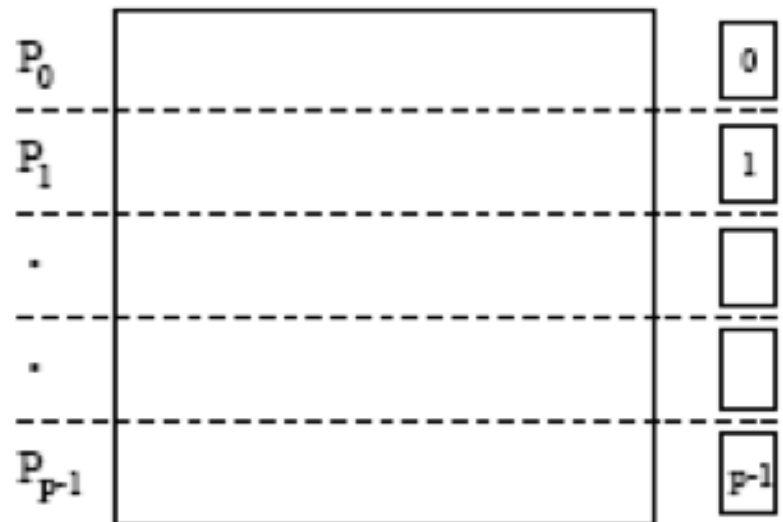
Matrix-Vector Multiplication

- **Problem:**
 - multiply a dense $n \times n$ matrix A with an $n \times 1$ vector x
 - yield the $n \times 1$ result vector y
- ***Serial algorithm requires n^2 multiplications and additions***

Matrix-Vector Multiplication: 1D Partitioning

First, consider simple case: $p = n$

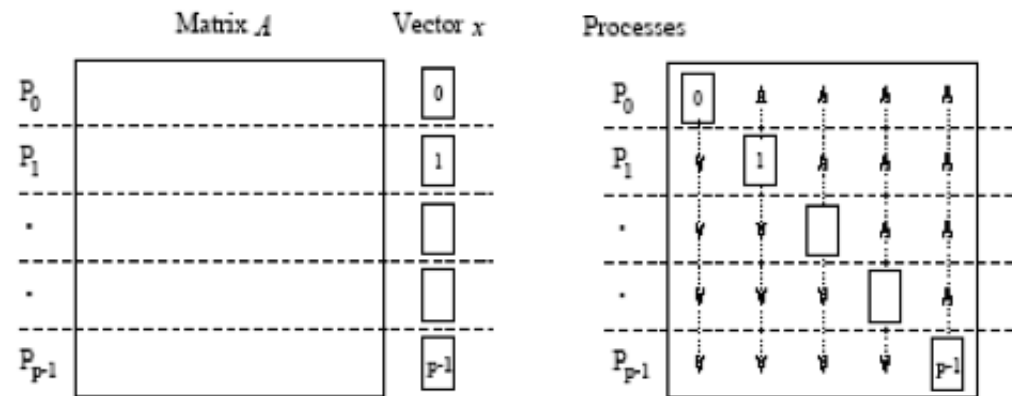
- $p \times p$ matrix A is partitioned among p processors
 - each processor stores a complete row of the matrix
- $p \times 1$ vector x is also partitioned
 - each process owns one element of x



Matrix-Vector Multiplication: 1D Partitioning

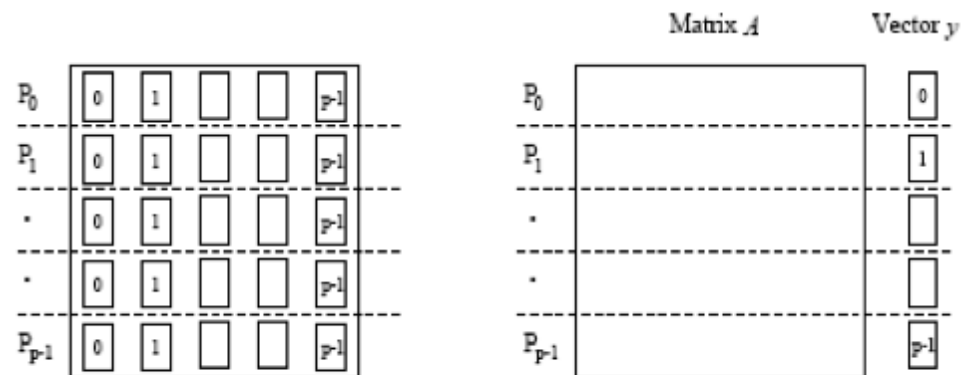
Computation

- Distribute all of x to each process using what primitive?



all-to-all
broadcast

- Process P_i computes $y[i] = \sum_{j=0}^{p-1} A[i,j] \times x[j]$



(c) Entire vector distributed to each process after the broadcast

(d) Final distribution of the matrix and the result vector y

Matrix-Vector Multiplication: 1D Partitioning

Analysis for $p = n$

- **All-to-all broadcast:** $\Theta(p) = \Theta(n)$ time
—each process must receive vector x , which is of length n
- **Computation of $y[i]$:** $\Theta(p) = \Theta(n)$ time
—each process performs a dot product of length n
- **Overall algorithm:** $\Theta(p) = \Theta(n)$ time
—both communication and computation are $\Theta(n)$
- **Total parallel work** = $\Theta(p^2) = \Theta(n^2)$

Matrix-Vector Multiplication: 1D Partitioning

Consider $p < n$

- **Initially, each process stores**
 - n/p complete rows of the matrix A
 - n/p elements of the vector x
- **Distribute all of vector x to each process**
 - all-to-all broadcast of among p processes, first msg of size n/p
- **On each processor: n/p local dot products on rows of length n**
- **Parallel run time $T_p = n^2/p + t_s \log p + t_w(n/p)(p-1)$ (hypercube)**

Distributing a Matrix of Rows

Originally: process 0 has a vector of rows in orig_A

Distribute the rows so that each process has their rows in A

Find the MPI bug!

```
for (i=0; i < n; i++) {  
    int dest = get_owner(i, n, npes);  
  
    if (pid == 0) {  
        MPI_Send(orig_A[i], n+1, MPI_DOUBLE, dest, i, MPI_COMM_WORLD);  
    }  
    if (pid == dest) {  
        MPI_Recv(A[local_row_index(i, n, npes, myrank)], n+1,  
                MPI_DOUBLE, 0, i, MPI_COMM_WORLD, NULL);  
    }  
}
```

Is there a better way?

Matrix-Vector Multiplication: 2D Partitioning

Consider when $p = n^2$

- $n \times n$ matrix A is partitioned among n^2 processors
—each processor owns a single element
- $n \times 1$ vector x is distributed in last column of n processors
- *What is a good algorithm given this distribution?*

Matrix-Vector Multiplication: 2D Partitioning

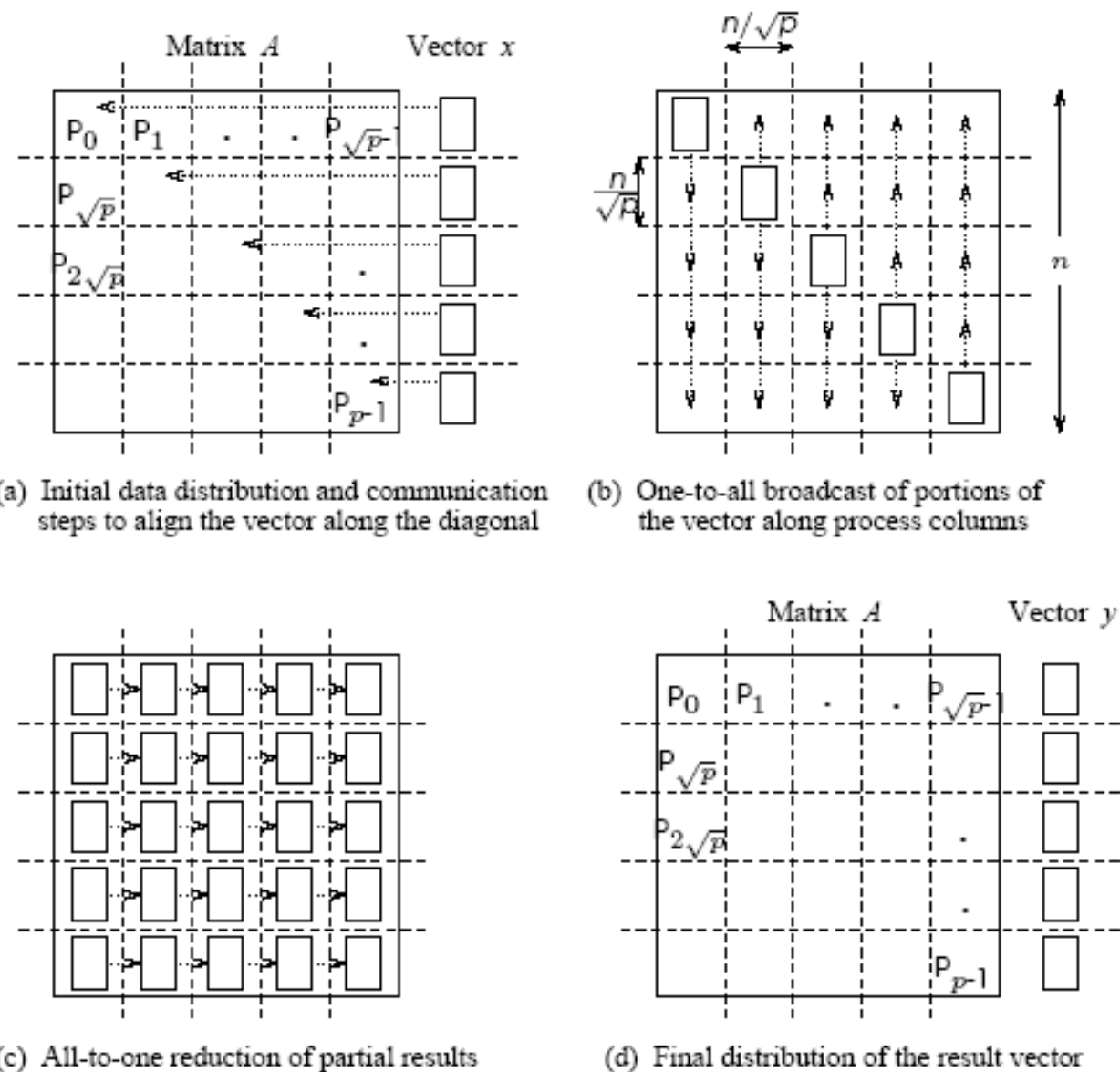
Computation sketch

- **Align the vector with the matrix**
 - first communication step
 - align the vector x along the main diagonal of the matrix
 - second communication step
 - copy vector element from each diagonal process to all processes in its corresponding column

use $n/p^{1/2}$ independent broadcasts

independent broadcast along each column of processes
- **Compute the result vector**
 - perform independent all-to-one reductions along each row

Matrix-Vector Multiplication: 2-D Partitioning



For the one-element-per-process case, $p = n^2$ if matrix size is $n \times n$

Matrix-Vector Multiplication: 2-D Partitioning

Analysis when $p = n^2$

- Three communication operations used in this algorithm
 - one-to-one communication: align vector along the main diagonal
 - one-to-all broadcast: vector element to n processes in column
 - all-to-one reduction in each row
- Time complexity of communication
 - the one-to-all and all-to-one operations take $\Theta(\log n)$ time
- Parallel time is $\Theta(\log n)$
- The cost (process-time product) is $\Theta(n^2 \log n)$ (recall $p = n^2$)

Matrix-Matrix Multiplication

- **Compute $C = A \times B$ for A, B, C $n \times n$ dense, square matrices**
- **Serial complexity is $O(n^3)$**
 - we don't explicitly consider better serial algorithms (e.g., Strassen)
 - they can be used as serial kernels in the parallel algorithms
- **Block operations are a useful concept**
 - an $n \times n$ matrix A can be regarded as a $q \times q$ array of blocks
 - $A_{i,j}$ ($0 \leq i, j < q$), each block is an $(n/q) \times (n/q)$ submatrix
- **Blocked computation**
 - q^3 matrix multiplications, each using $(n/q) \times (n/q)$ matrices

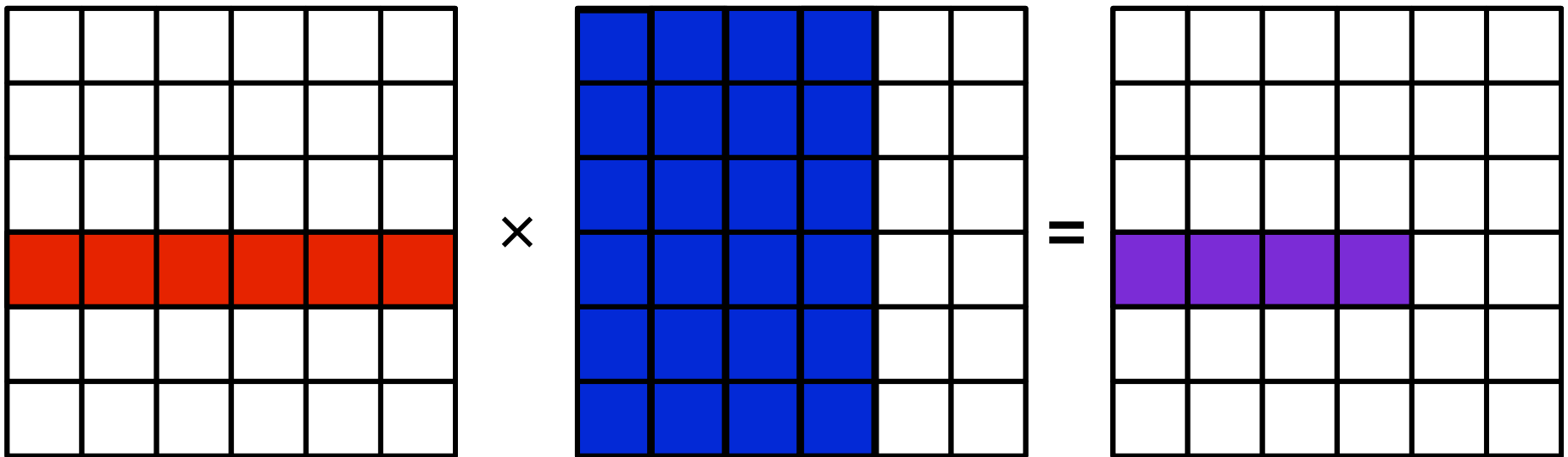
Matrix-Matrix Multiplication

Consider two $n \times n$ matrices A and B partitioned into p blocks

- $A_{i,j}$ and $B_{i,j}$ ($0 \leq i, j < p^{1/2}$) **of size $n/p^{1/2} \times n/p^{1/2}$ each**
- **Process $P_{i,j}$**
 - initially stores $A_{i,j}$ and $B_{i,j}$
 - computes block $C_{i,j}$ of the result matrix.
- **Computing submatrix $C_{i,j}$**
 - requires all submatrices $A_{i,k}$ and $B_{k,j}$ for $0 \leq k < p^{1/2}$

Matrix Multiplication

Consider data needed for output matrix block shown in purple



Matrix-Matrix Multiplication

- **Two all-to-all broadcasts**
 - broadcast blocks of A along rows
 - broadcast blocks of B along columns
- **Perform local submatrix multiplication**
- **Two all-to-all broadcasts take time** $2(t_s \log \sqrt{p} + t_w(n^2/p)(\sqrt{p} - 1))$
- **Computation requires \sqrt{p} multiplications of $(n/\sqrt{p}) \times (n/\sqrt{p})$ submatrices**
- **Parallel run time**

$$T_p = n^3/p + 2t_s \log \sqrt{p} + 2t_w(n^2/p)(\sqrt{p} - 1)$$

- **Major drawback of the algorithm: memory requirements**
 - each process needs space for a block of rows and a block of columns

Matrix-Matrix Multiplication: Cannon's Algorithm

Memory efficient algorithm idea

- **Approach**
 - schedule the computations of the \sqrt{p} processes of each row
 - at any given time, each process is using different blocks
- **Systematically rotate blocks among the processes**
 - rotate after every submatrix multiplication
 - each process get fresh blocks in each rotation

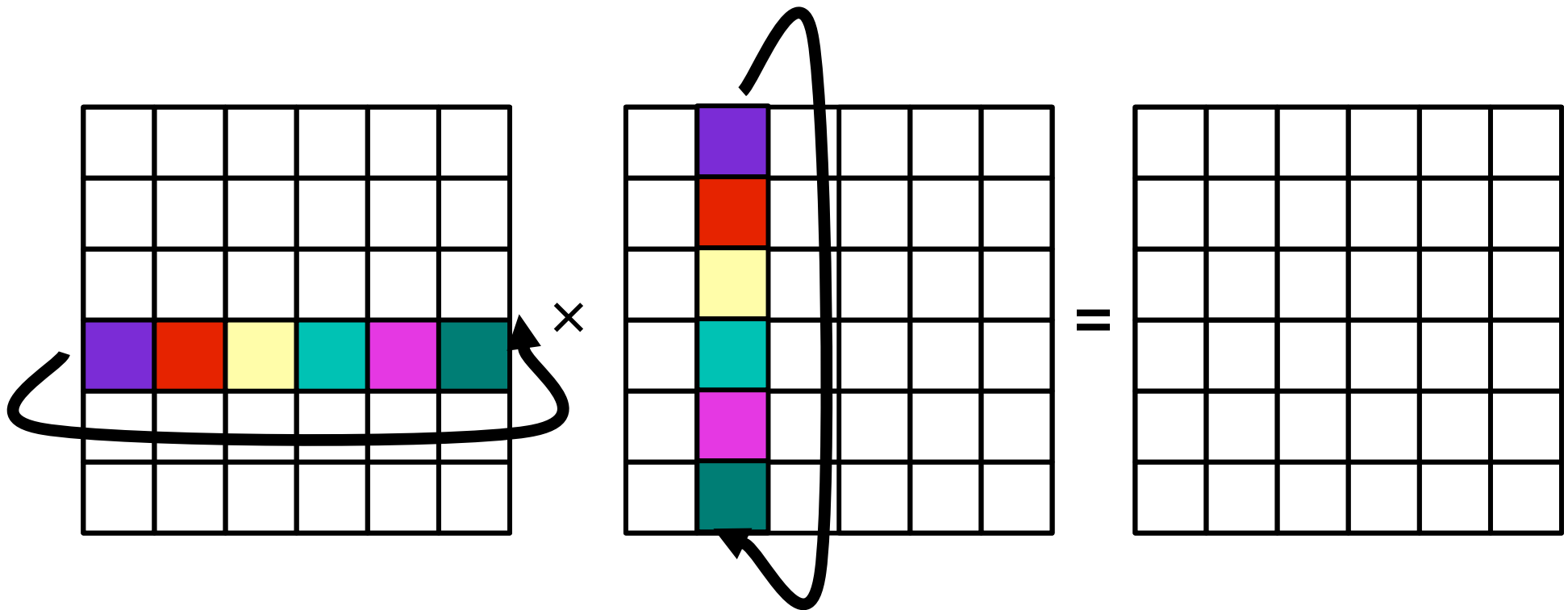
Matrix-Matrix Multiplication: Cannon's Algorithm

Aim: Circulate the blocks of A and B with each process multiplying local submatrices at each step

- **Initial alignment**
 - shift all submatrices $A_{i,j}$ to the left (with wraparound) by i steps
 - shift all submatrices $B_{i,j}$ up (with wraparound) by j steps
- **Perform local block multiplication**
- **Rotate**
 - move each block of A one step left (with wraparound)
 - move each block of B moves one step up (with wraparound)
- **Perform next block multiplication**
- **Add to partial result**
- **Repeat until all \sqrt{p} blocks have been multiplied**

Cannon's Matrix Multiplication

Initial State

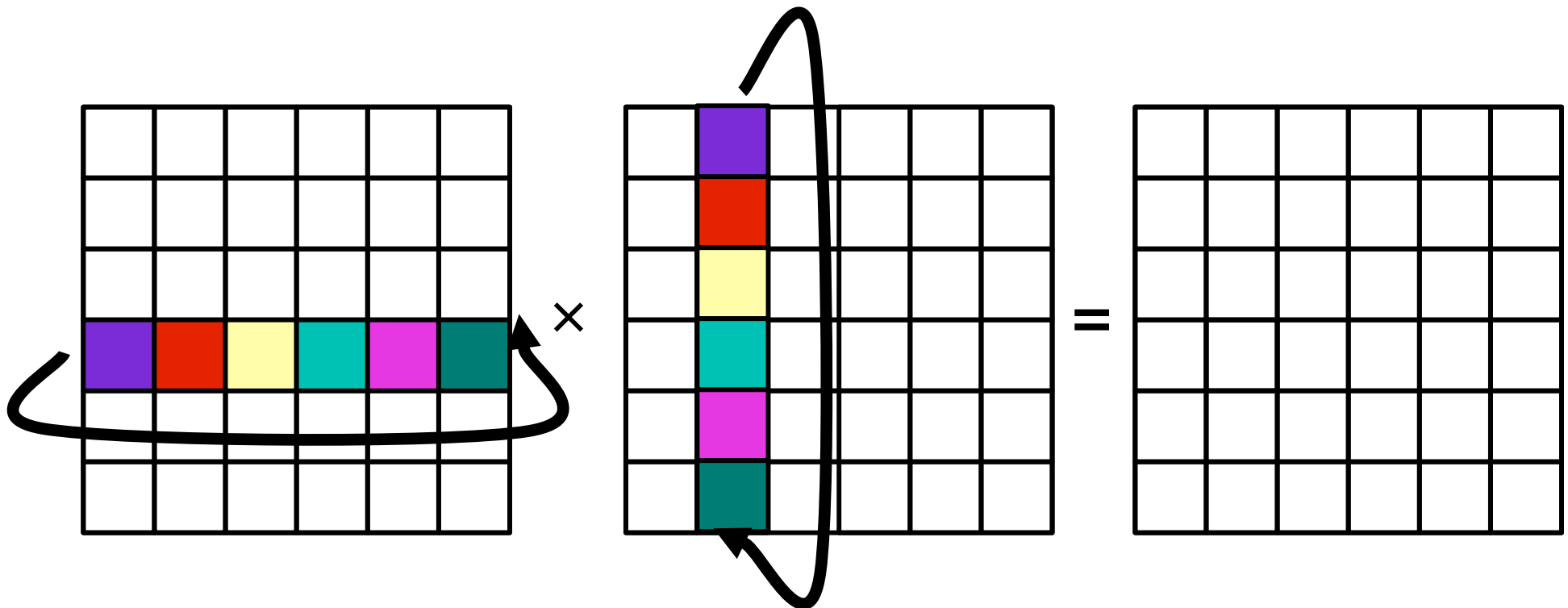


Cannon's Matrix Multiplication

Perform Alignment

shift A_{ij} left by i

shift B_{ij} up by j



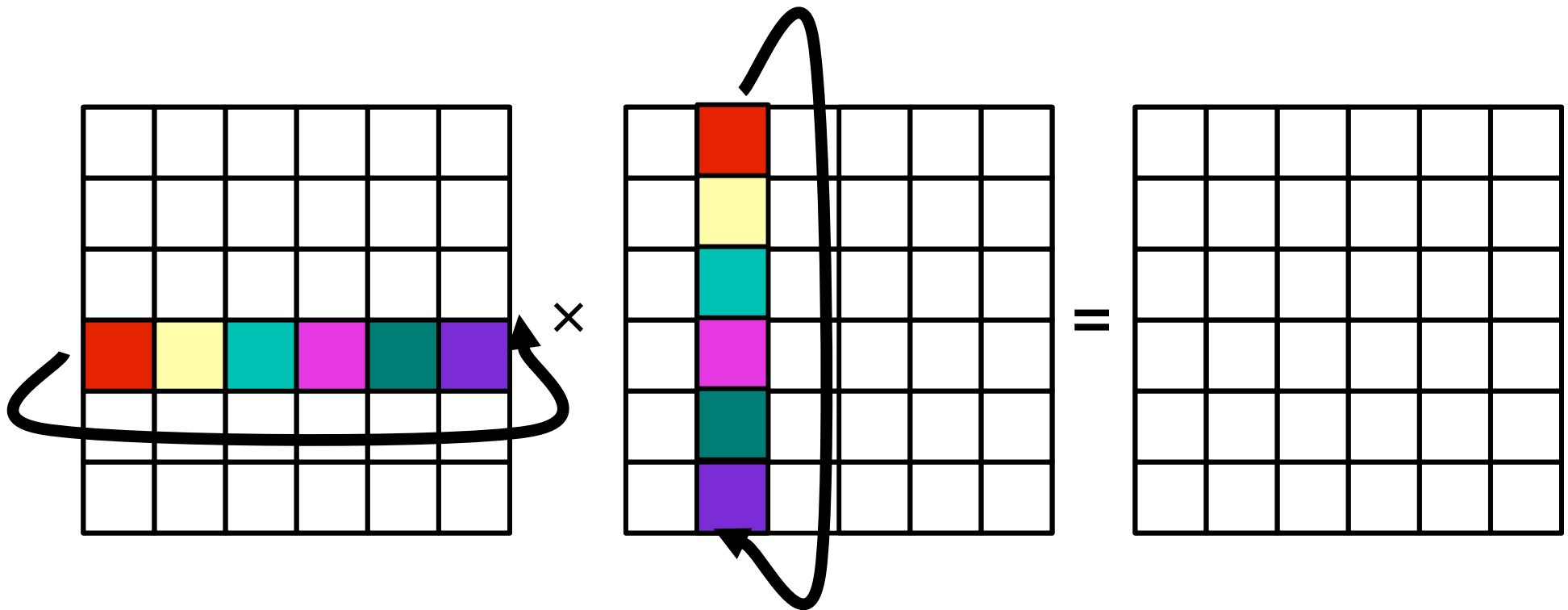
Alignment step 0

Cannon's Matrix Multiplication

Perform Alignment

shift A_{ij} left by i

shift B_{ij} up by j



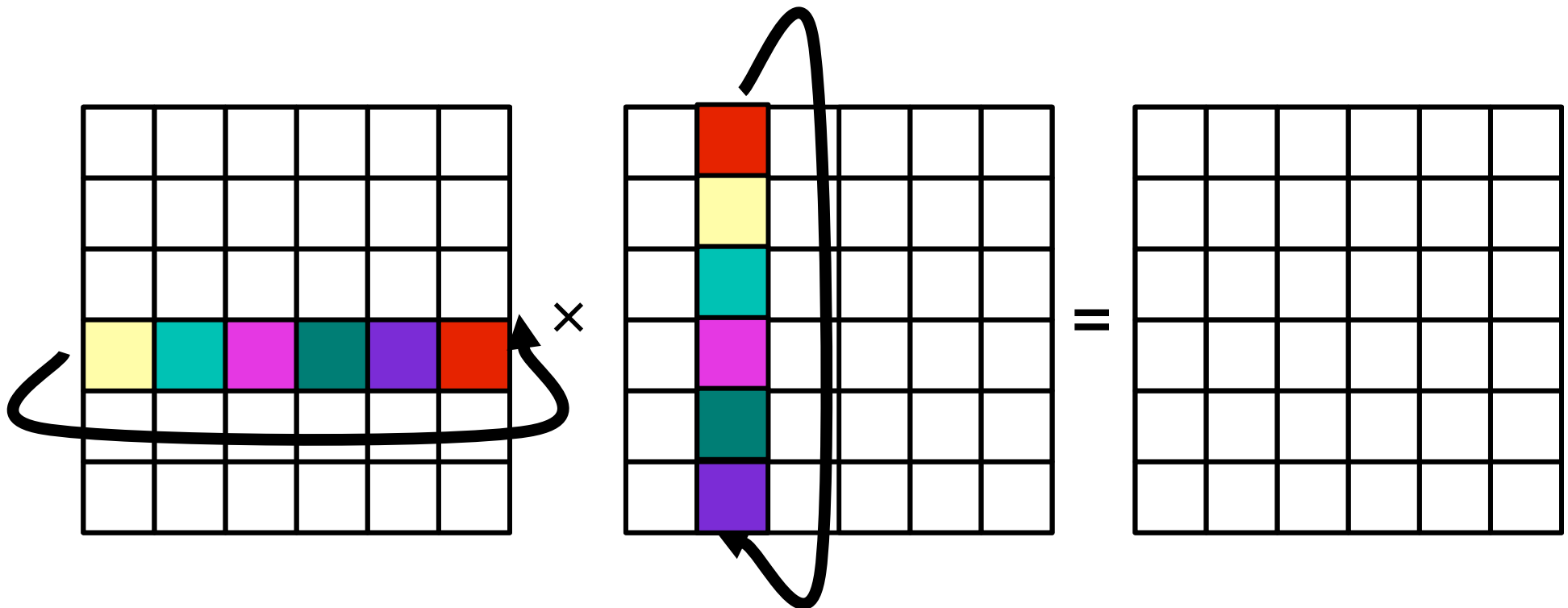
Alignment step 1

Cannon's Matrix Multiplication

Perform Alignment

shift A_{ij} left by i

shift B_{ij} up by j



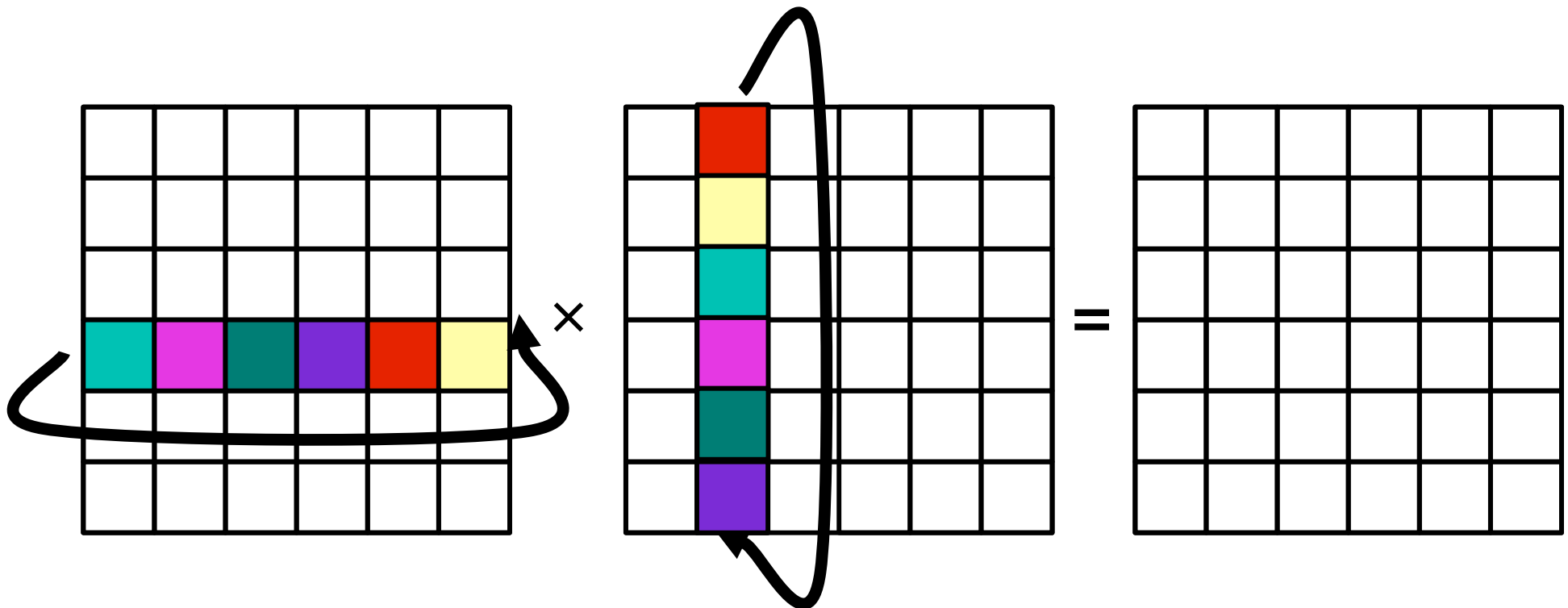
Alignment step 2

Cannon's Matrix Multiplication

Perform Alignment

shift A_{ij} left by i

shift B_{ij} up by j



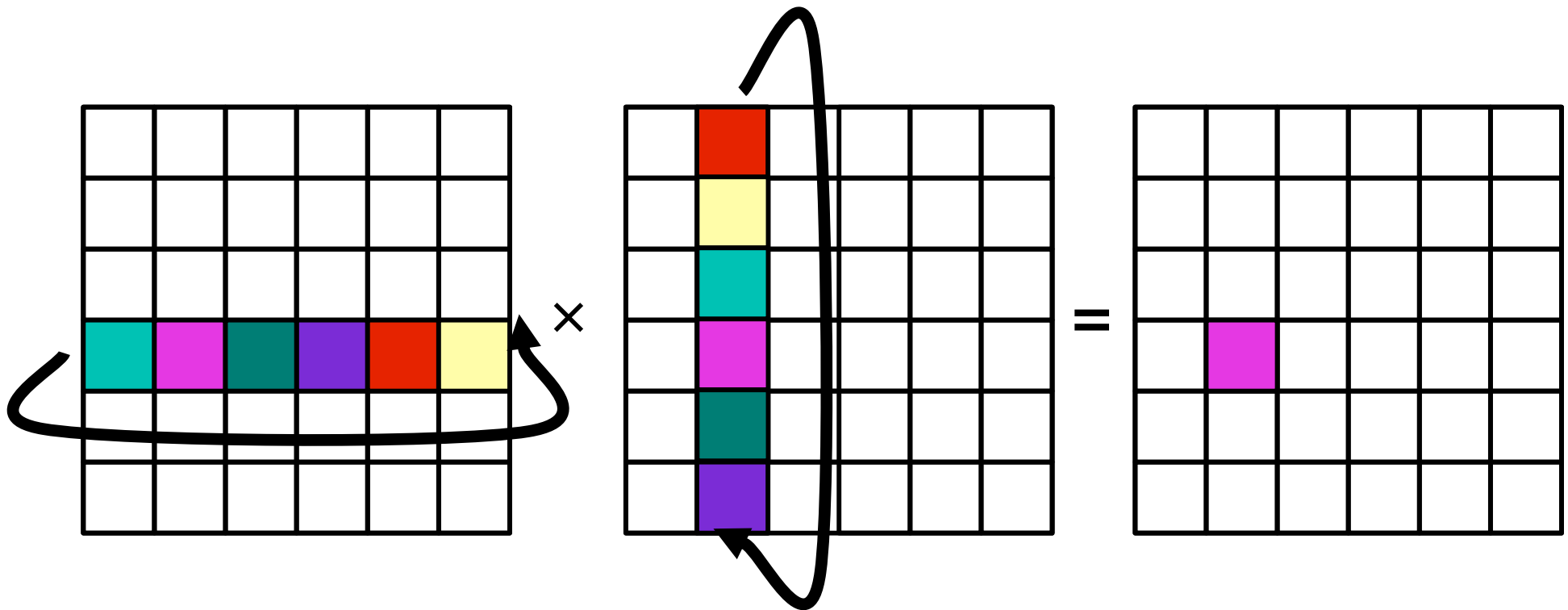
Alignment step 3

Cannon's Matrix Multiplication

Perform Multiplication; then

shift A_{ij} left by 1

shift B_{ij} up by 1



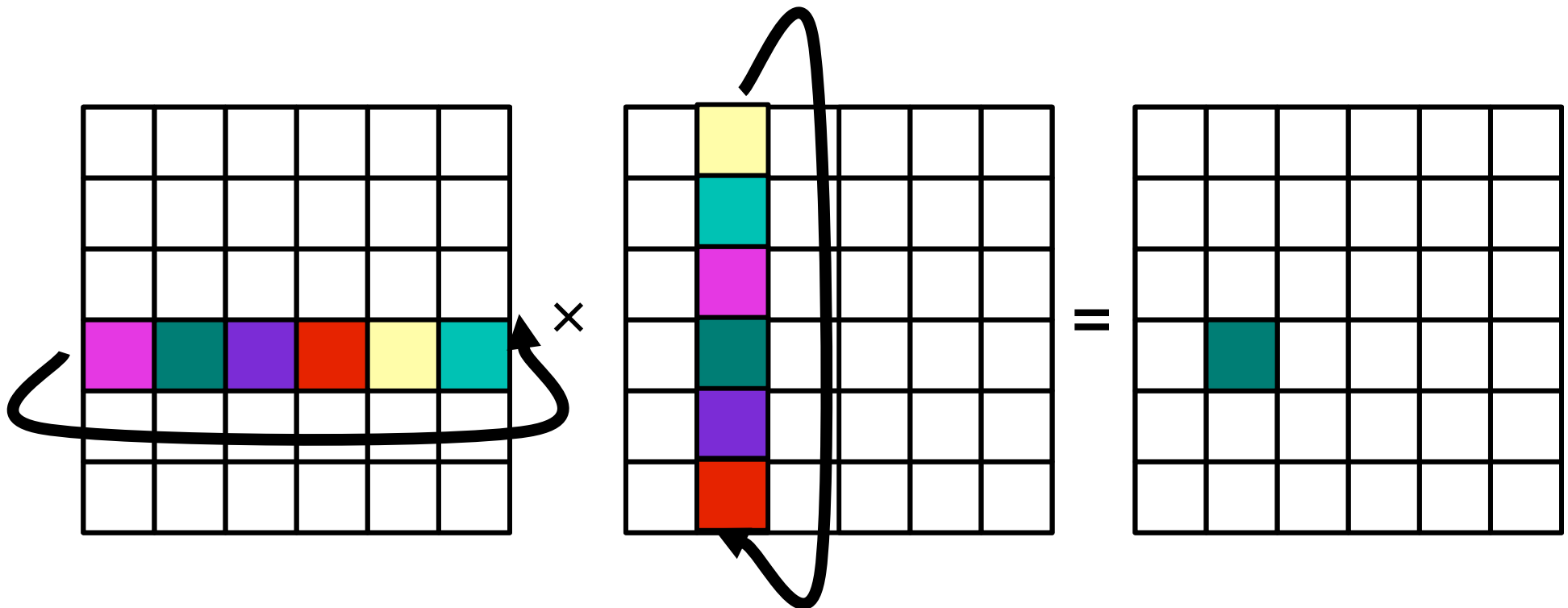
Multiplication step 1

Cannon's Matrix Multiplication

Perform Multiplication; then

shift A_{ij} left by 1

shift B_{ij} up by 1



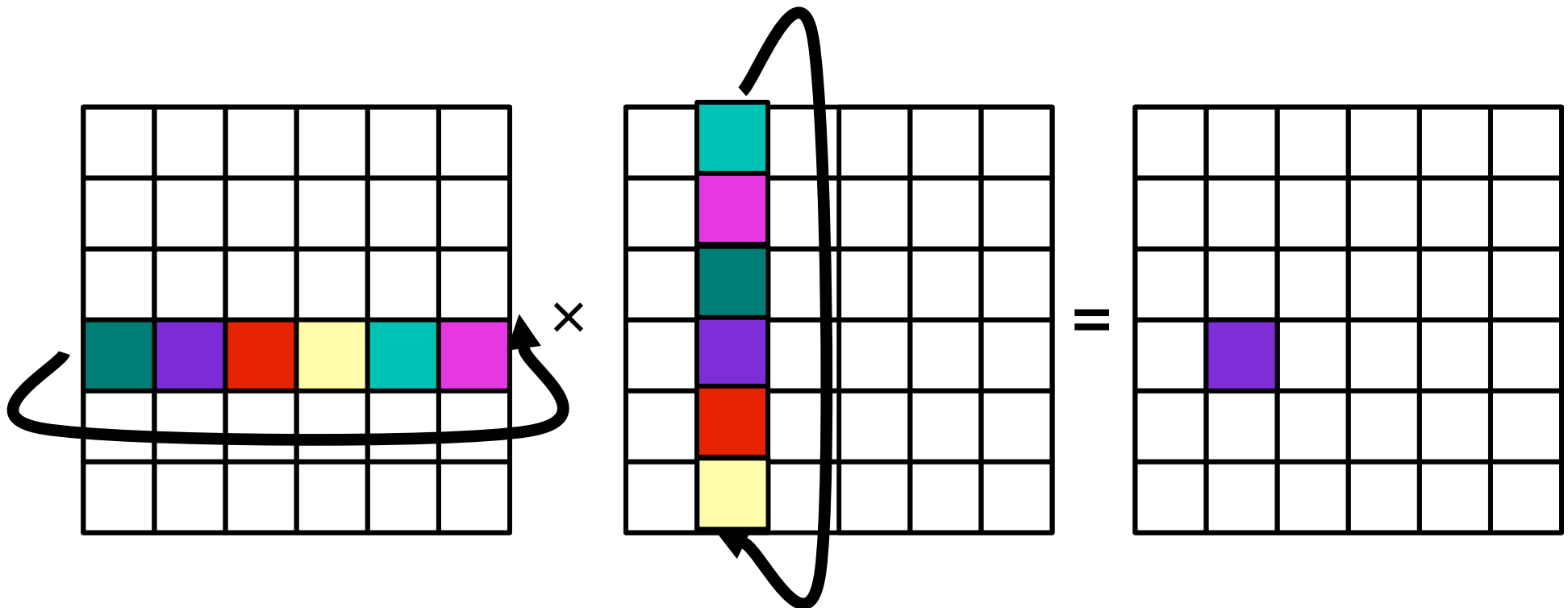
Multiplication step 2

Cannon's Matrix Multiplication

Perform Multiplication; then

shift A_{ij} left by 1

shift B_{ij} up by 1



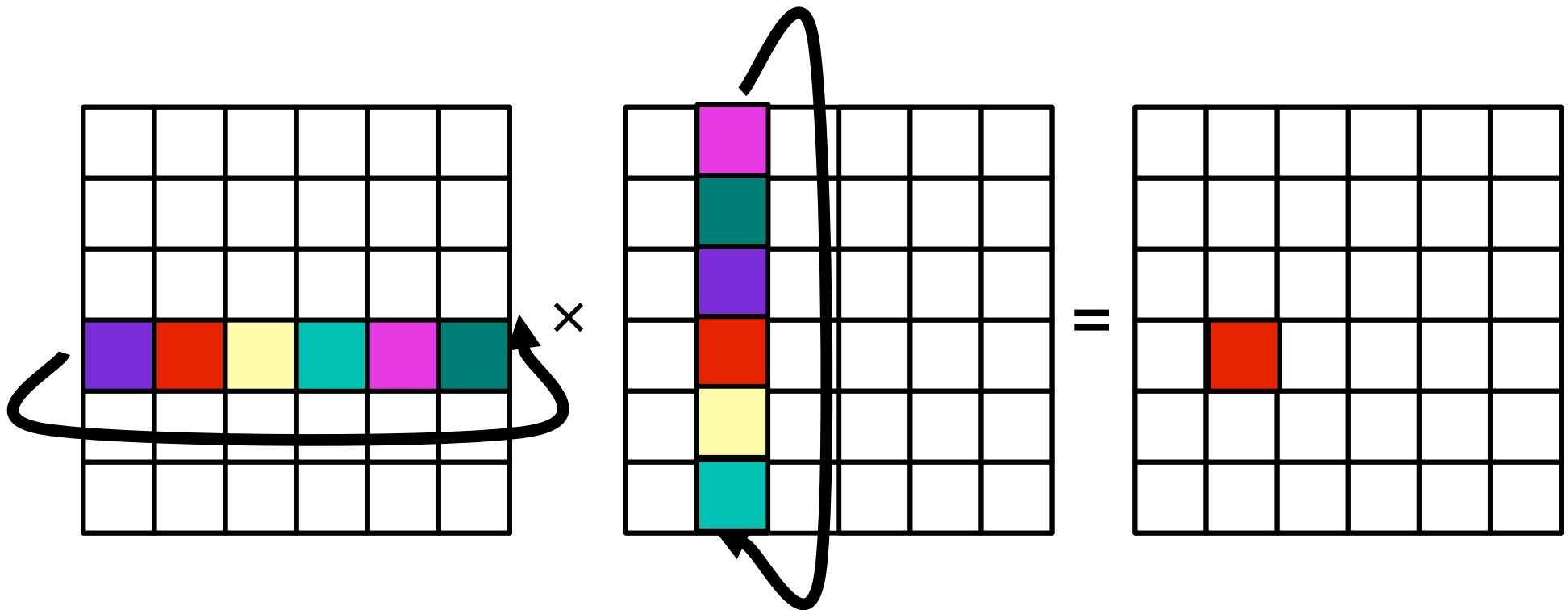
Multiplication step 3

Cannon's Matrix Multiplication

Perform Multiplication; then

shift A_{ij} left by 1

shift B_{ij} up by 1



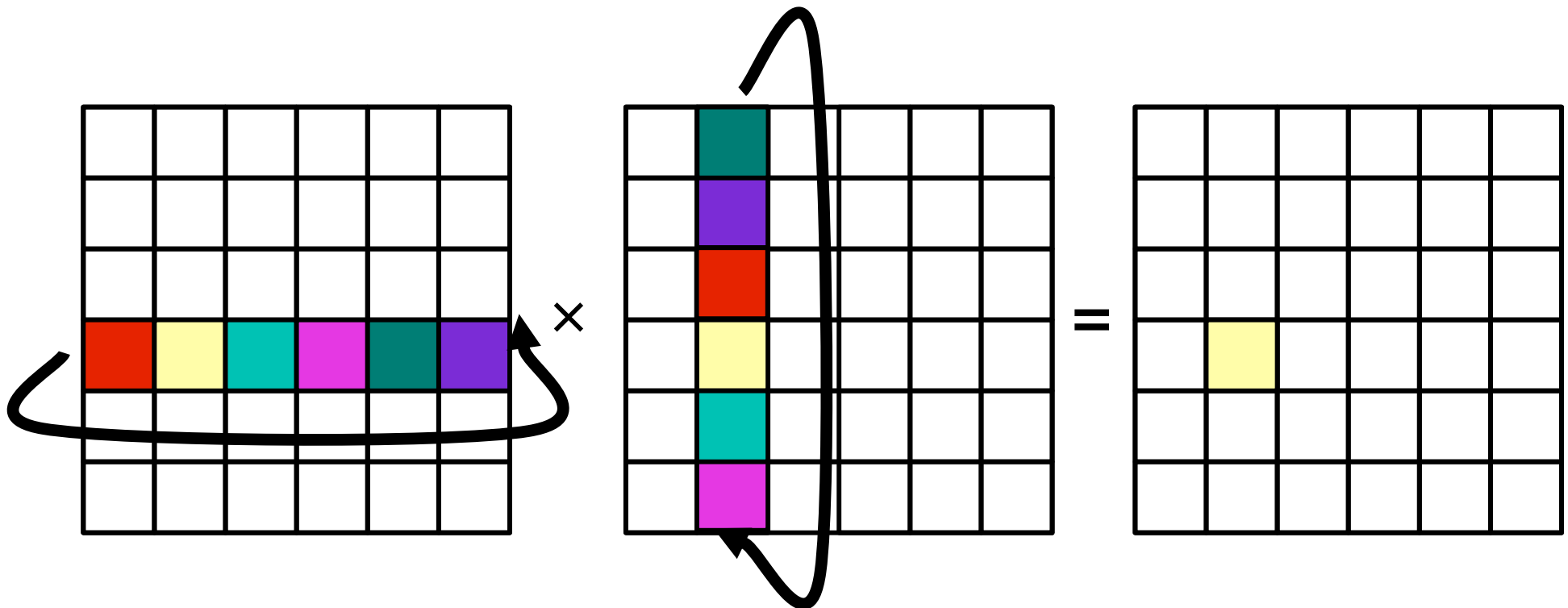
Multiplication step 4

Cannon's Matrix Multiplication

Perform Multiplication; then

shift A_{ij} left by 1

shift B_{ij} up by 1



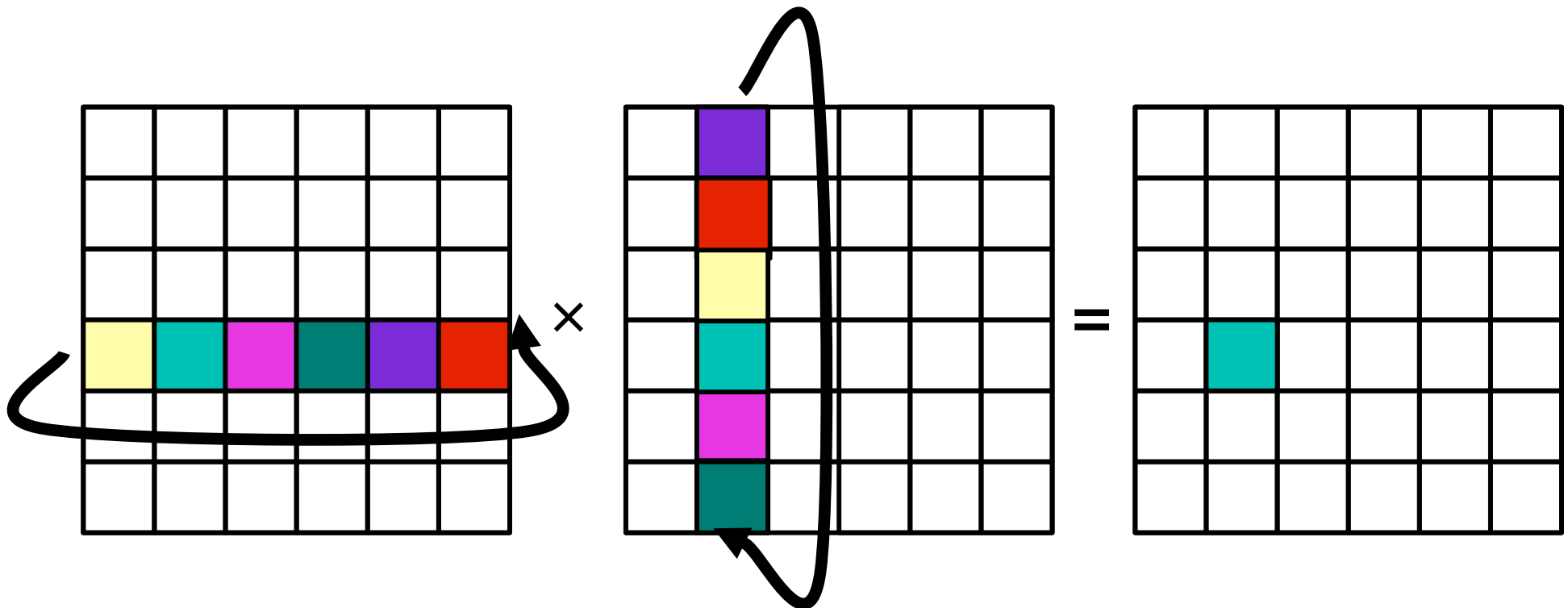
Multiplication step 5

Cannon's Matrix Multiplication

Perform Multiplication; then

shift A_{ij} left by 1

shift B_{ij} up by 1



Multiplication step 6

Matrix-Matrix Multiplication: Cannon's Algorithm

- **Alignment step**
 - maximum distance over which a block shifts is $\sqrt{p} - 1$
 - two shift operations require a total of $2(t_s + t_w n^2/p)$ time.
- **Compute-shift phase**
 - \sqrt{p} single-step shifts
 - each shift takes $t_s + t_w n^2/p$ time
- **Computation time**
 - multiplying \sqrt{p} matrices of size $(n/\sqrt{p}) \times (n/\sqrt{p})$ is n^3/p
- **The parallel time is approximately**

$$T_P = \frac{n^3}{p} + 2\sqrt{p}t_s + 2t_w \frac{n^2}{\sqrt{p}}.$$

- **More messages, but Cannon's is memory optimal**

Cannon's Algorithm in Use

G. Baumgartner et al. Synthesis of High-Performance Parallel Programs for a Class of Ab Initio Quantum Chemistry Models. Proceedings of the IEEE, Vol. 93, No. 2, Feb. 2005, pp. 276-292.

<http://www.csc.lsu.edu/~gb/TCE/Publications/SynthApproach-ProcIEEE05.pdf>

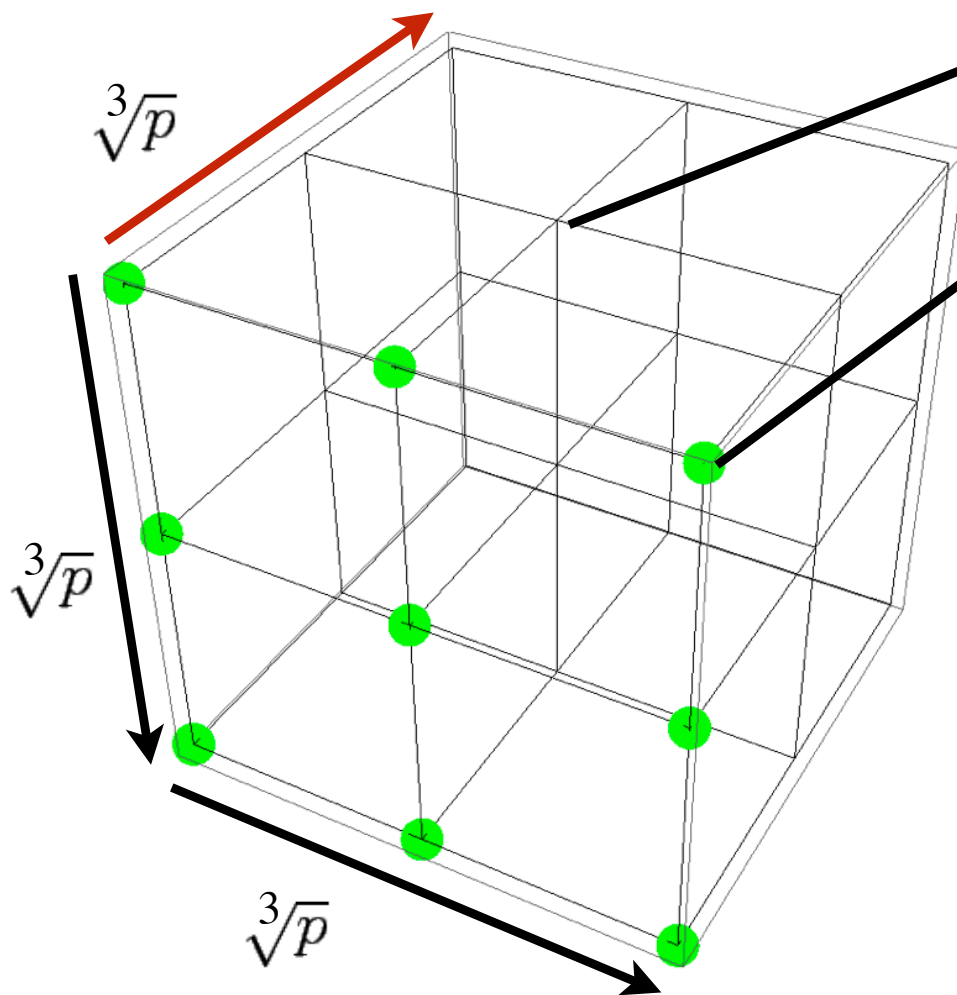
This paper provides an overview of a program synthesis system for a class of quantum chemistry computations. These computations are expressible as a set of tensor contractions and arise in electronic structure modeling. The input to the system is a high-level specification of the computation, from which the system can synthesize high-performance parallel code tailored to the characteristics of the target architecture. Several components of the synthesis system are described, focusing on performance optimization issues that they address.

Keywords—Communication minimization, compiler optimizations, data locality optimization, domain-specific languages, high-level programming languages, memory-constrained optimization, tensor contraction expressions

3D Matrix Multiplication

- What if a processor can hold a larger block of the matrix ?

$$(n/\sqrt{p}) \times (n/\sqrt{p}) \longrightarrow (n/\sqrt[3]{p}) \times (n/\sqrt[3]{p})$$



Each point on the cube is a processor

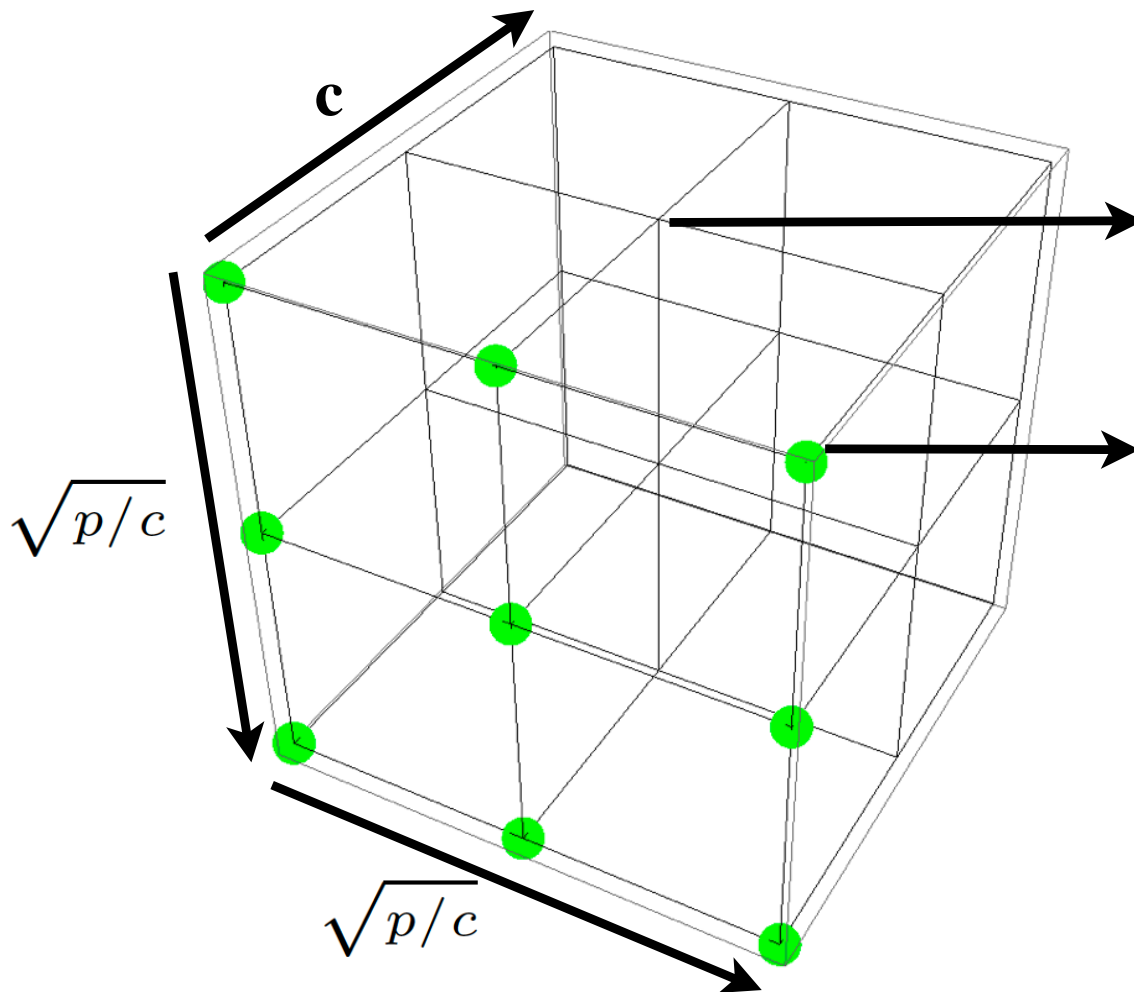
Each green circle represents pair of blocks of A and B of size:
 $(n/\sqrt[3]{p}) \times (n/\sqrt[3]{p})$

1. Broadcast A and B to back planes
2. Skew the blocks of A and B so that all $A_{i,k}$ and $B_{k,j}$ block pairs are along the red dimension
3. Multiply the blocks at each processor
4. Reduce along the red dimension to yield $C_{i,j}$ at the i,j processor in the front plane

2.5D Matrix Multiplication

- What if we can store a matrix larger than $(n/\sqrt{p}) \times (n/\sqrt{p})$

$$(n/\sqrt[3]{p}) \times (n/\sqrt[3]{p}) \longrightarrow \frac{n}{\sqrt{p/c}} \times \frac{n}{\sqrt{p/c}}$$



Each point on the cube is a processor

Each green circle represents a pair of blocks from A and B of size:

$$\frac{n}{\sqrt{p/c}} \times \frac{n}{\sqrt{p/c}}$$

Idea: compute a partial result in each plane by multiplying a subset of the block pairs for a $C_{i,j}$ result

2.5D Matrix Multiplication Algorithm

Algorithm 2: $[C] = \text{2.5D-matrix-multiply}(A, B, n, p, c)$

Input: square n -by- n matrices A, B distributed so that P_{ij0} owns $\frac{n}{\sqrt{p/c}}$ -by- $\frac{n}{\sqrt{p/c}}$ blocks A_{ij} and B_{ij} for each i, j

Output: square n -by- n matrix $C = A \cdot B$ distributed so that P_{ij0} owns $\frac{n}{\sqrt{p/c}}$ -by- $\frac{n}{\sqrt{p/c}}$ block C_{ij} for each i, j

```
/* do in parallel with all processors */
forall  $i, j \in \{0, 1, \dots, \sqrt{p/c} - 1\}, k \in \{0, 1, \dots, c - 1\}$  do
     $P_{ij0}$  broadcasts  $A_{ij}$  and  $B_{ij}$  to all  $P_{ijk}$  /* replicate input matrices */
     $s := \text{mod}(j - i + k\sqrt{p/c^3}, \sqrt{p/c})$  /* initial circular shift on A */
     $P_{ijk}$  sends  $A_{ij}$  to  $A_{\text{local}}$  on  $P_{isk}$ 
     $s' := \text{mod}(i - j + k\sqrt{p/c^3}, \sqrt{p/c})$  /* initial circular shift on B */
     $P_{ijk}$  sends  $B_{ij}$  to  $B_{\text{local}}$  on  $P_{s'jk}$ 
     $C_{ijk} := A_{\text{local}} \cdot B_{\text{local}}$ 
     $s := \text{mod}(j + 1, \sqrt{p/c})$ 
     $s' := \text{mod}(i + 1, \sqrt{p/c})$ 
    for  $t = 1$  to  $\sqrt{p/c^3} - 1$  do
         $P_{ijk}$  sends  $A_{\text{local}}$  to  $P_{isk}$  /* rightwards circular shift on A */
         $P_{ijk}$  sends  $B_{\text{local}}$  to  $P_{s'jk}$  /* downwards circular shift on B */
         $C_{ijk} := C_{ijk} + A_{\text{local}} \cdot B_{\text{local}}$ 
    end
     $P_{ijk}$  contributes  $C_{ijk}$  to a sum-reduction to  $P_{ij0}$ 
end
```

2.5D Matrix Multiplication Algorithm

Algorithm 2: $[C] = \text{2.5D-matrix-multiply}(A, B, n, p, c)$

Input: square n -by- n matrices A, B distributed so that P_{ij0} owns $\frac{n}{\sqrt{p/c}}$ -by- $\frac{n}{\sqrt{p/c}}$ blocks A_{ij} and B_{ij} for each i, j

Output: square n -by- n matrix $C = A \cdot B$ distributed so that P_{ij0} owns $\frac{n}{\sqrt{p/c}}$ -by- $\frac{n}{\sqrt{p/c}}$ block C_{ij} for each i, j

/ do in parallel with all processors*

forall $i, j \in \{0, 1, \dots, \sqrt{p/c} - 1\}, k \in \{0, 1, \dots, c - 1\}$ **do**

P_{ij0} broadcasts A_{ij} and B_{ij} to all P_{ijk} \longrightarrow

$s := \text{mod}(j - i + k\sqrt{p/c^3}, \sqrt{p/c})$

P_{ijk} sends A_{ij} to A_{local} on P_{isk}

$s' := \text{mod}(i - j + k\sqrt{p/c^3}, \sqrt{p/c})$

P_{ijk} sends B_{ij} to B_{local} on $P_{s'jk}$

$C_{ijk} := A_{\text{local}} \cdot B_{\text{local}}$

$s := \text{mod}(j + 1, \sqrt{p/c})$

$s' := \text{mod}(i + 1, \sqrt{p/c})$

for $t = 1$ **to** $\sqrt{p/c^3} - 1$ **do**

P_{ijk} sends A_{local} to P_{isk}

P_{ijk} sends B_{local} to $P_{s'jk}$

$C_{ijk} := C_{ijk} + A_{\text{local}} \cdot B_{\text{local}}$

end

P_{ijk} contributes C_{ijk} to a sum-reduction to P_{ij0}

end

Pass my blocks to the
back planes of the cube
(3D matrix multiplication step)

/ initial circular shift on B */*

/ rightwards circular shift on A */*

/ downwards circular shift on B */*

2.5D Matrix Multiplication Algorithm

Algorithm 2: $[C] = \text{2.5D-matrix-multiply}(A, B, n, p, c)$

Input: square n -by- n matrices A, B distributed so that P_{ij0} owns $\frac{n}{\sqrt{p/c}}$ -by- $\frac{n}{\sqrt{p/c}}$ blocks A_{ij} and B_{ij} for each i, j

Output: square n -by- n matrix $C = A \cdot B$ distributed so that P_{ij0} owns $\frac{n}{\sqrt{p/c}}$ -by- $\frac{n}{\sqrt{p/c}}$ block C_{ij} for each i, j

/ do in parallel with all processors */*

forall $i, j \in \{0, 1, \dots, \sqrt{p/c} - 1\}, k \in \{0, 1, \dots, c - 1\}$ **do**

P_{ij0} broadcasts A_{ij} and B_{ij} to all P_{ijk}

/ replicate input matrices */*

$s := \text{mod}(j - i + k\sqrt{p/c^3}, \sqrt{p/c})$

P_{ijk} sends A_{ij} to A_{local} on P_{isk}

$s' := \text{mod}(i - j + k\sqrt{p/c^3}, \sqrt{p/c})$

P_{ijk} sends B_{ij} to B_{local} on $P_{s'jk}$

Like Cannon's
Alignment Step

$C_{ijk} := A_{\text{local}} \cdot B_{\text{local}}$

$s := \text{mod}(j + 1, \sqrt{p/c})$

$s' := \text{mod}(i + 1, \sqrt{p/c})$

for $t = 1$ **to** $\sqrt{p/c^3} - 1$ **do**

P_{ijk} sends A_{local} to P_{isk}

P_{ijk} sends B_{local} to $P_{s'jk}$

$C_{ijk} := C_{ijk} + A_{\text{local}} \cdot B_{\text{local}}$

/ rightwards circular shift on A */*

/ downwards circular shift on B */*

end

P_{ijk} contributes C_{ijk} to a sum-reduction to P_{ij0}

end

2.5D Matrix Multiplication Algorithm

Algorithm 2: $[C] = \text{2.5D-matrix-multiply}(A, B, n, p, c)$

Input: square n -by- n matrices A, B distributed so that P_{ij0} owns $\frac{n}{\sqrt{p/c}}$ -by- $\frac{n}{\sqrt{p/c}}$ blocks A_{ij} and B_{ij} for each i, j

Output: square n -by- n matrix $C = A \cdot B$ distributed so that P_{ij0} owns $\frac{n}{\sqrt{p/c}}$ -by- $\frac{n}{\sqrt{p/c}}$ block C_{ij} for each i, j


```

/* do in parallel with all processors */
forall  $i, j \in \{0, 1, \dots, \sqrt{p/c} - 1\}, k \in \{0, 1, \dots, c - 1\}$  do
     $P_{ij0}$  broadcasts  $A_{ij}$  and  $B_{ij}$  to all  $P_{ijk}$                                 /* replicate input matrices */
     $s := \text{mod}(j - i + k\sqrt{p/c^3}, \sqrt{p/c})$                                 /* initial circular shift on A */
     $P_{ijk}$  sends  $A_{ij}$  to  $A_{\text{local}}$  on  $P_{isk}$ 
     $s' := \text{mod}(i - j + k\sqrt{p/c^3}, \sqrt{p/c})$                                 /* initial circular shift on B */
     $P_{ijk}$  sends  $B_{ij}$  to  $B_{\text{local}}$  on  $P_{s'jk}$ 
     $C_{ijk} := A_{\text{local}} \cdot B_{\text{local}}$ 
     $s := \text{mod}(j + 1, \sqrt{p/c})$ 
     $s' := \text{mod}(i + 1, \sqrt{p/c})$ 
    for  $t = 1$  to  $\sqrt{p/c^3} - 1$  do
        

$P_{ijk}$  sends  $A_{\text{local}}$  to  $P_{isk}$   

             $P_{ijk}$  sends  $B_{\text{local}}$  to  $P_{s'jk}$   

             $C_{ijk} := C_{ijk} + A_{\text{local}} \cdot B_{\text{local}}$


        Like Cannon's  
Multiply and Shift Step
    end
     $P_{ijk}$  contributes  $C_{ijk}$  to a sum-reduction to  $P_{ij0}$ 
end

```

Matrix Multiplication Costs

Algorithm	Bandwidth (W)	Latency (S)	Memory per processor(M)
Cannon's MM	$\theta(n^2/p)$	$\theta(\sqrt{p})$	$\cong 3n^2/p$
3D MM	$\theta(n^2/p^{2/3})$	$\theta(\log p)$	$\cong 3n^2/p^{2/3}$
2.5D MM	$\Omega(n^2/\sqrt{cp})$	$\theta(\sqrt{p}/\sqrt{c^3})$	$\cong 3cn^2/p$

W = number of words sent or received along critical path

S = number of messages sent or received along critical path

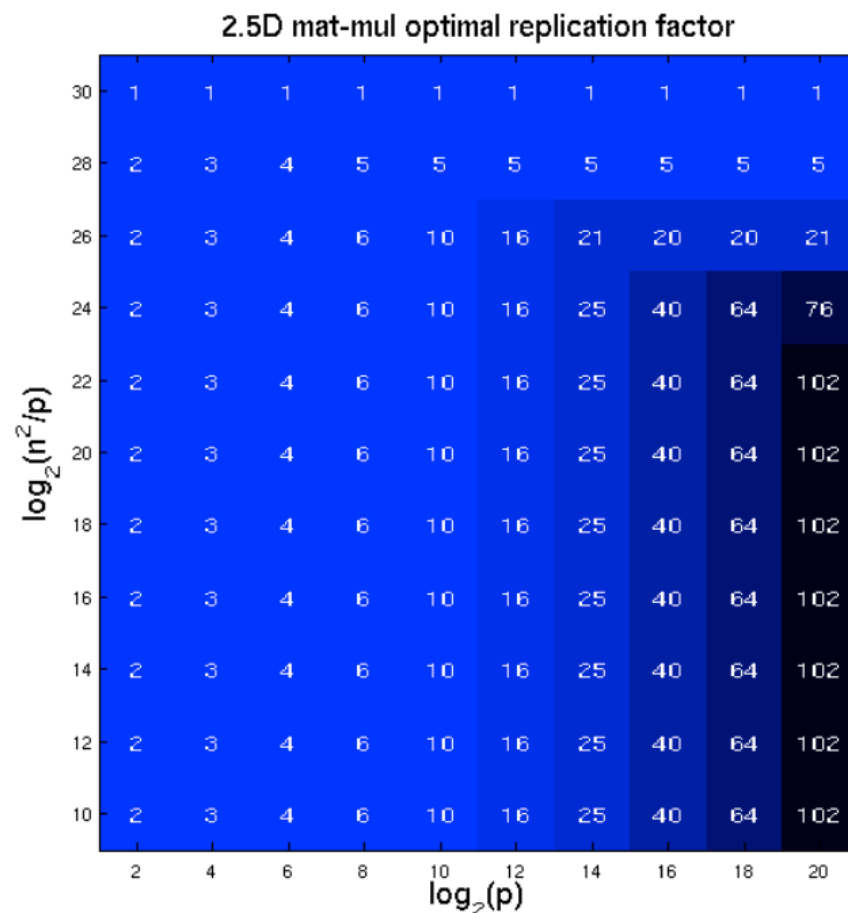
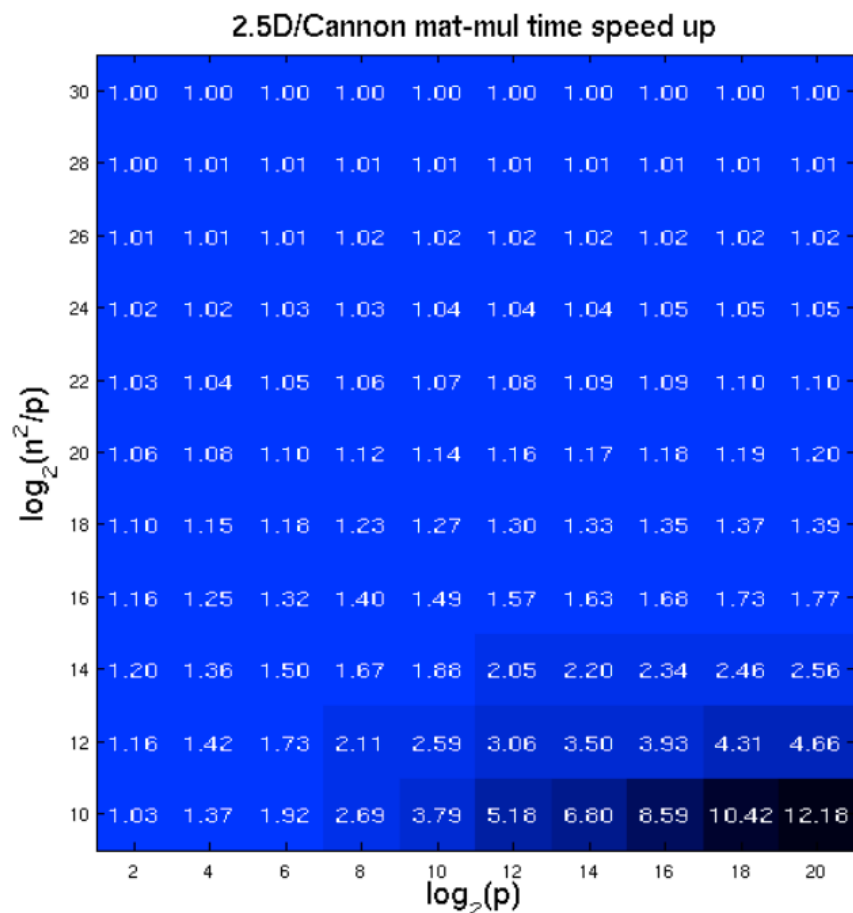
2.5dMM

Reduces bandwidth cost by \sqrt{c} compared to Cannon's algorithm

Reduces latency cost by $\sqrt{c^3}$ compared to Cannon's algorithm

*matrix size: $n \times n$, number of processors: p

2.5D Matrix Multiplication Algorithm



Speed up for best choice of c in comparison to Cannon's algorithm.

“On 16,384 nodes of BG/P, our 2.5D algorithm multiplies a small square matrix ($n = 8192$), 2.6X faster than Cannon's algorithm.”

Figure Credits: Technical Report No. UCB/EECS-2011-10,
Communication-optimal parallel 2.5D matrix multiplication and LU factorization algorithms

Linear Algebra for Multicore

- Multicore is a disruptive technology for software
- Must rethink and rewrite applications, algorithms and software
 - as before with cluster computing and message passing
- Numerical libraries need to change
- Recent research
 - event-driven DAG scheduled computations
 - direct solvers (LU) on distributed memory using UPC
 - PLASMA software framework dense linear algebra for multicore

PLASMA: Parallel Linear Algebra for Multicore

- **Objectives**

- parallel performance

- high utilization of each core
 - scaling to large numbers of cores

- any memory model

- shared memory: symmetric or non-symmetric
 - distributed memory
 - GPUs

- **Solution properties**

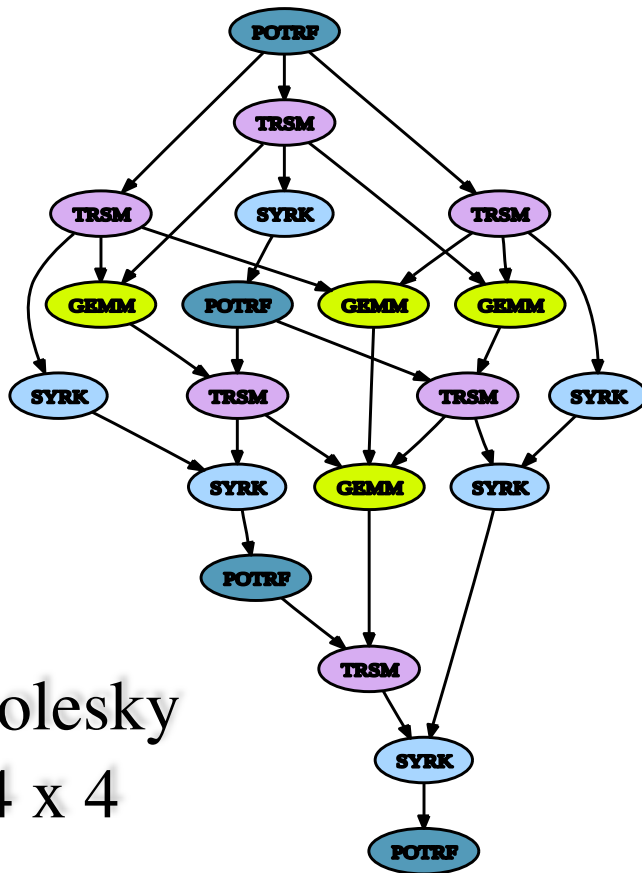
- asynchrony: avoid fork-join (bulk synchronous design)
 - dynamic scheduling: out-of-order execution
 - fine granularity: independent block operations
 - locality of reference: store data using block data layout

A community effort led by Tennessee and Berkeley

PLASMA Methodology

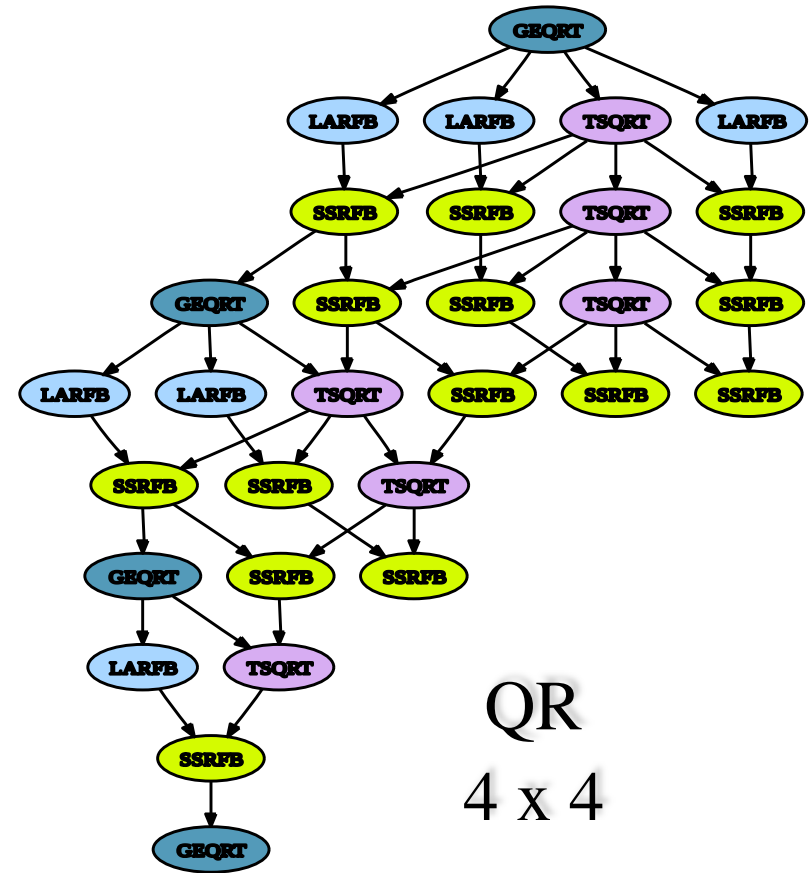
Computations as DAGs

Reorganize algorithms and software to work on tiles that are scheduled based on the directed acyclic graph of the computation



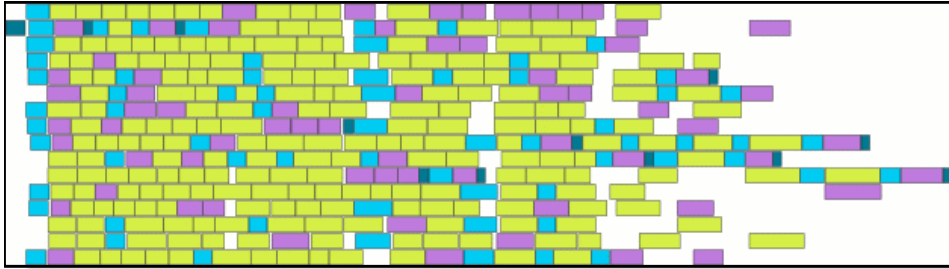
Cholesky

4 x 4



QR
4 x 4

Cholesky using PLASMA

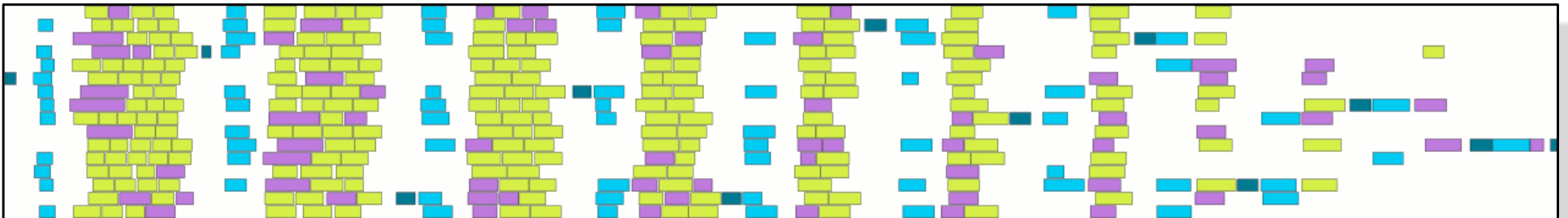


PLASMA

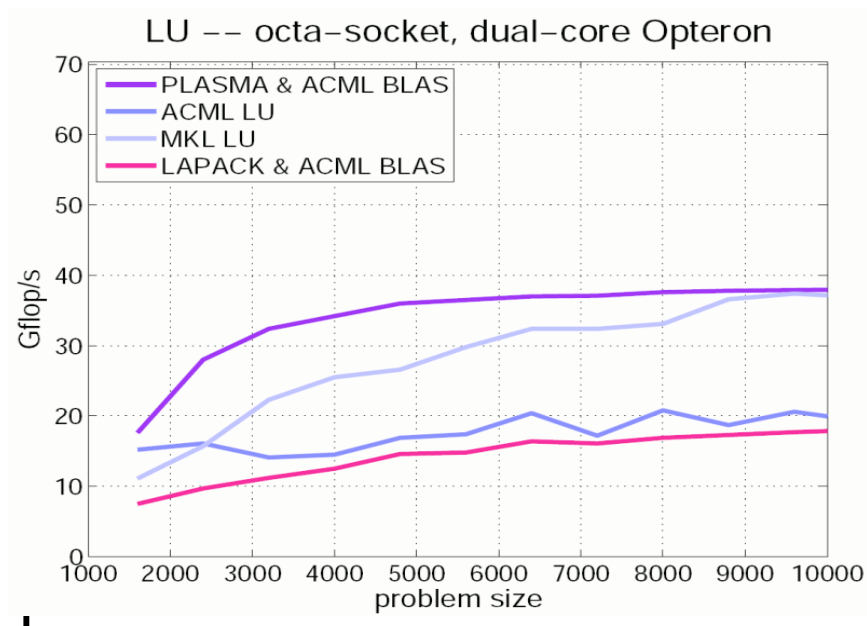
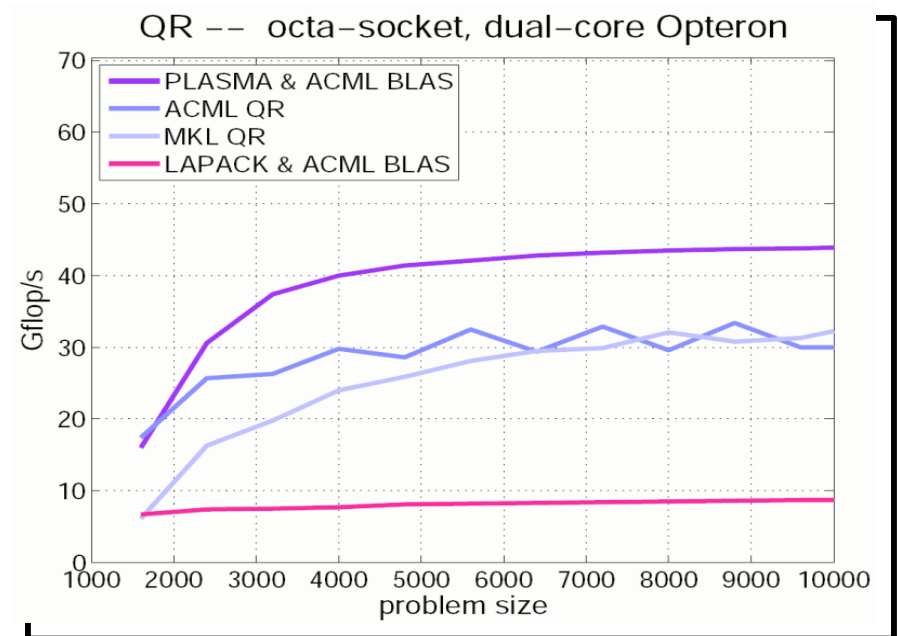
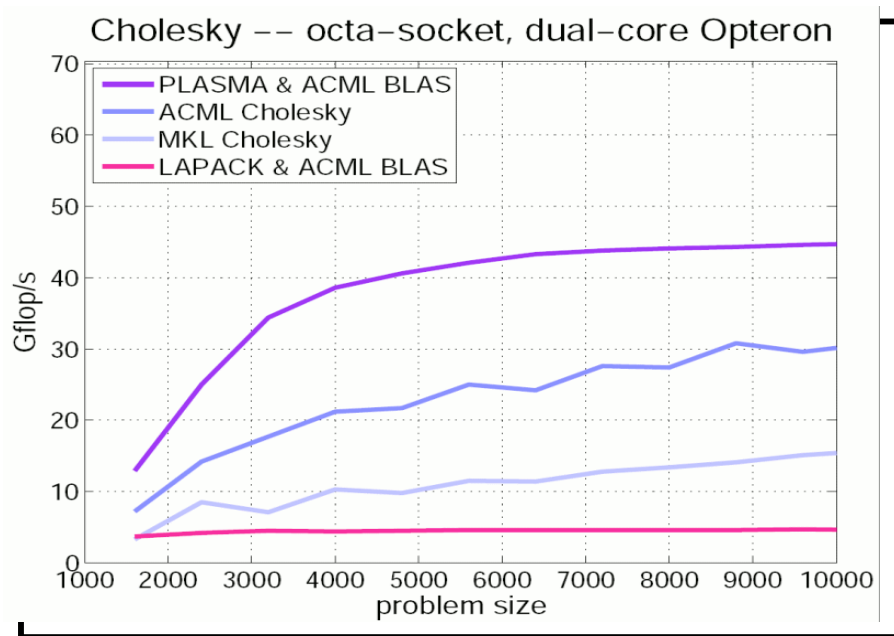
Arbitrary DAG

Fully dynamic scheduling

Nested fork-join parallelism (e.g., Cilk, TBB)



PLASMA Provides Highest Performance



Over 40 Years of Dense Matrix Algorithms

Approach	Reduce Communication	Length of Critical Path	Memory Footprint
Cannon 1969	communication intensive	\sqrt{p} shifts	optimal
Multipartitioning 2002	“just enough cuts” for balanced parallelism without excessive msg volume	full parallelism using skewed cyclic block distribution	
PLASMA 2009		DAG + priorities avoids waiting	
2.5D MM 2011	reduce message volume along critical path	reduce number of messages along critical path	maintain ‘c’ copies of data

References

- Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar. Chapter 8 “Dense Matrix Algorithms”, In Introduction to Parallel Computing, Addison Wesley, 2003.
- Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar. Slides on Dense Matrix Algorithms accompanying Introduction to Parallel Computing, Addison Wesley, 2003.
- G. Baumgartner et al. Synthesis of High-Performance Parallel Programs for a Class of Ab Initio Quantum Chemistry Models. Proceedings of the IEEE, Vol. 93, No. 2, February 2005, pp. 276-292. <http://www.csc.lsu.edu/~gb/TCE/Publications/SynthApproach-ProclEEE05.pdf>
- Alain Darte, John Mellor-Crummey, Robert Fowler, and Daniel Chavarria-Miranda. 2003. Generalized multipartitioning of multi-dimensional arrays for parallelizing line-sweep computations. *J. Parallel Distributed Computing* 63, 9 (September 2003), 887-911. <http://www.sciencedirect.com/science/article/pii/S0743731503001035>
- Edgar Solomonik and James Demmel. Communication-optimal parallel 2.5D matrix multiplication and LU factorization algorithms. In Euro-Par 2011 *Parallel Processing*, Springer-Verlag, Berlin, Heidelberg, 2011, 90-109. <http://www.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-72.pdf>
- <http://icl.cs.utk.edu/plasma/index.html>
- <http://www.cs.utexas.edu/~flame>