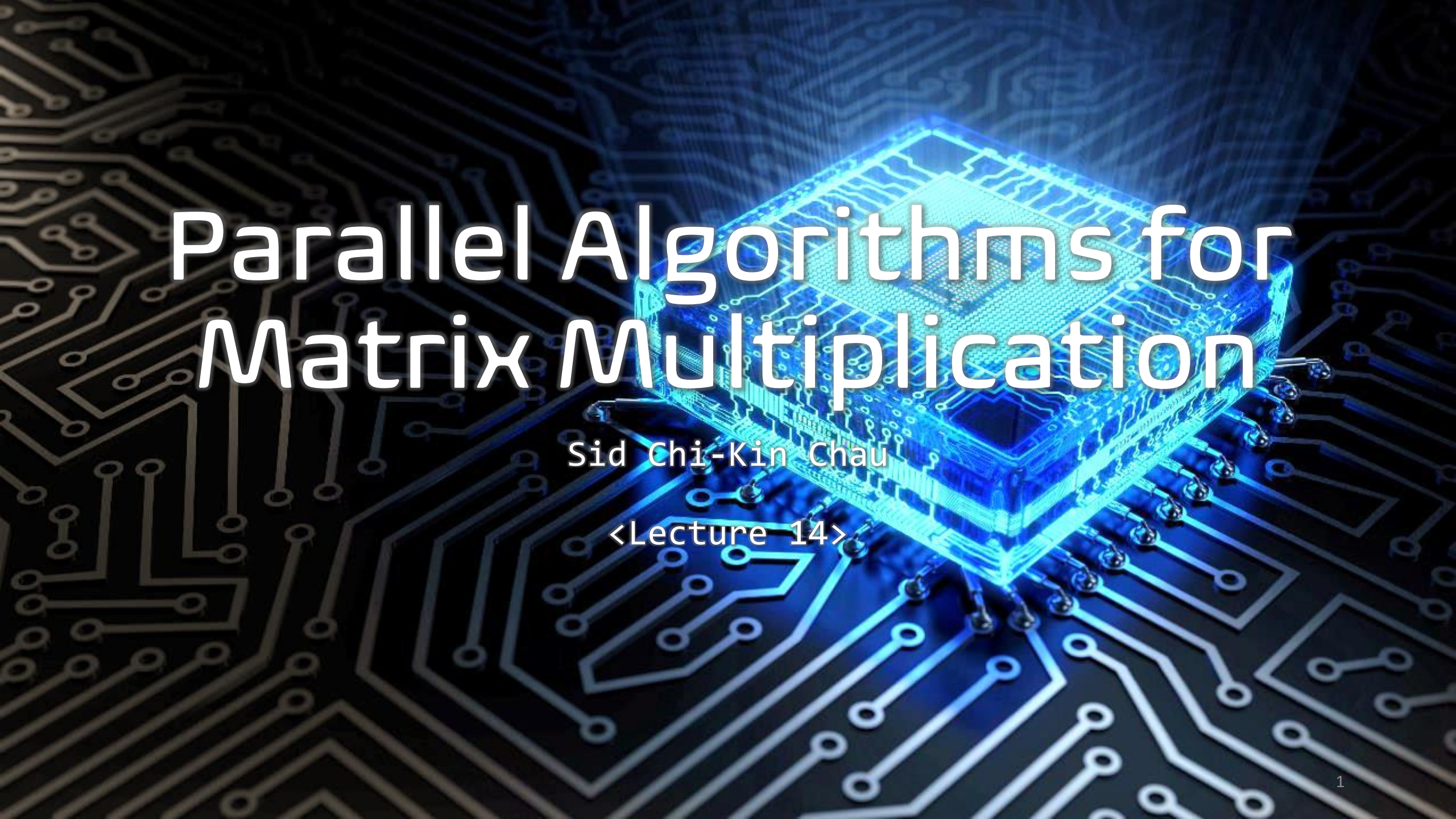


Parallel Algorithms for Matrix Multiplication

Sid Chi-Kin Chau

<Lecture 14>

Importance of Matrix Computations

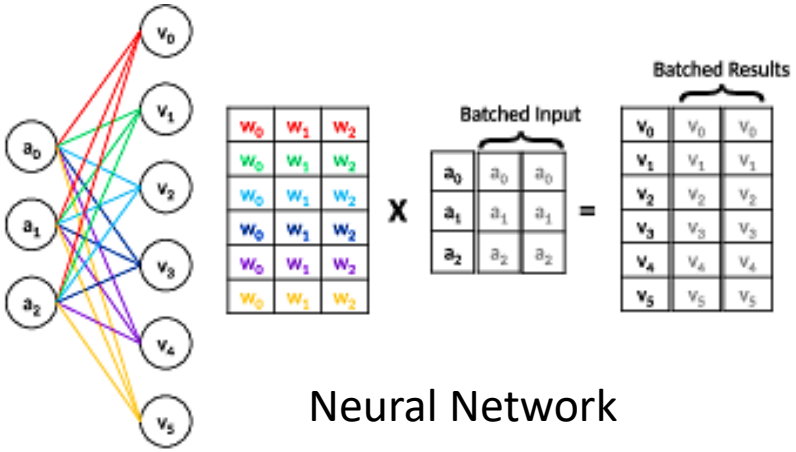
"What is the Matrix?"

💬 **Neo** (from a movie)

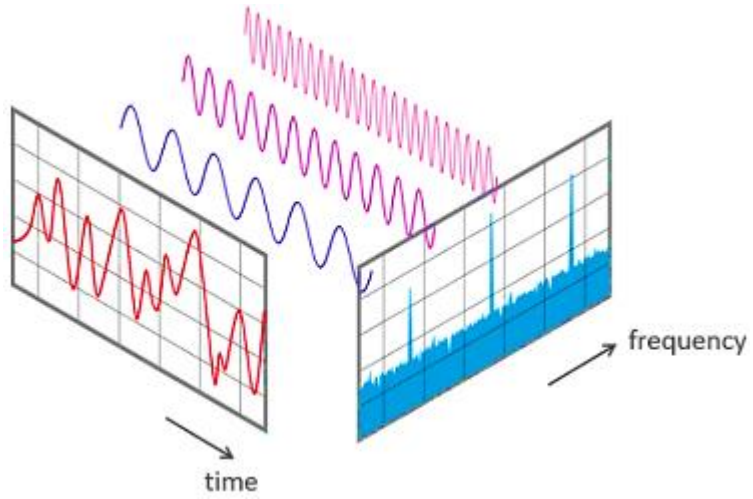
- The concept of matrix has existed, as long as the history of linear equations
- Lots of applications
  - Graphics, optimization, control systems
  - Solutions to many engineering, scientific and economic problems
- Goals
  - Parallel matrix multiplication algorithms
  - What makes (in)efficient matrix multiplication parallel algorithms?

Why Matrix Computation?

- Fundamental in many problems
  - Computer graphics
  - Linear algebra and multi-linear algebra
  - Machine learning (DNNs, CNNs)
  - Optimization and control
- Classes of matrix computations
  - Dense/Sparse matrix multiplication
  - Linear system solution, inverse matrix
  - Spectrum analysis (eigenvalues decomposition)
  - Singular values
  - Tensor Computations



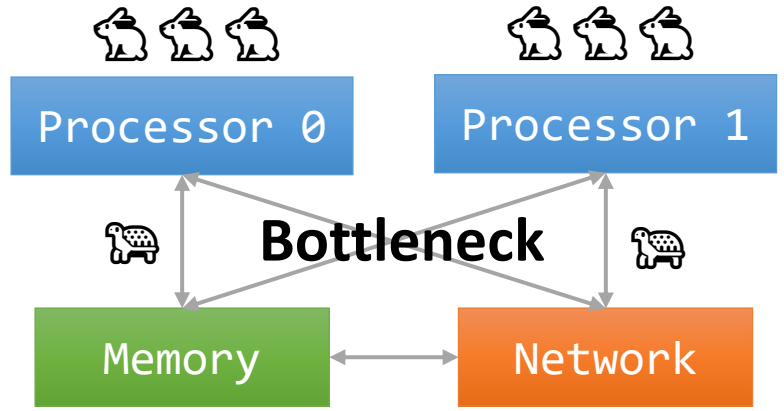
Neural Network



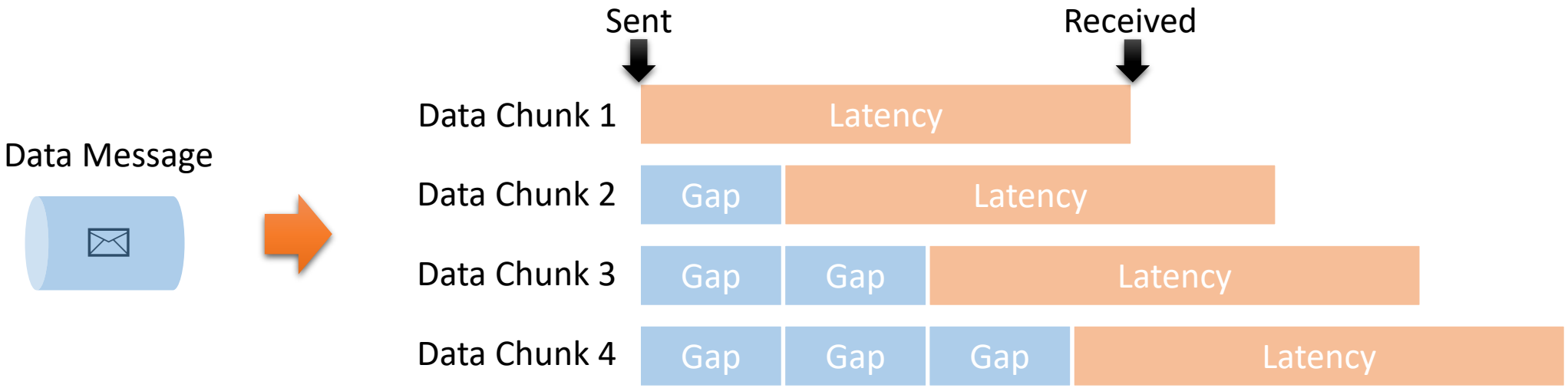
Spectrum analysis

Computational Performance

- Performance of computations is affected by two major factors:
  1. **Complexity** of arithmetic, execution in processors (e.g., FLOPs)
     - Need computationally efficient algorithms
  2. **Communication** of data and results within/among processors, and from/to memory
     - Transmitting data among multiple levels of memory hierarchy
     - Communicating among processors over a network
     - The major bottleneck is with communication
       - Processors can't execute unless data is loaded to register or cache
     - Need more than just computationally efficient algorithms, but also ***communication-avoiding*** algorithms (both serial and parallel)

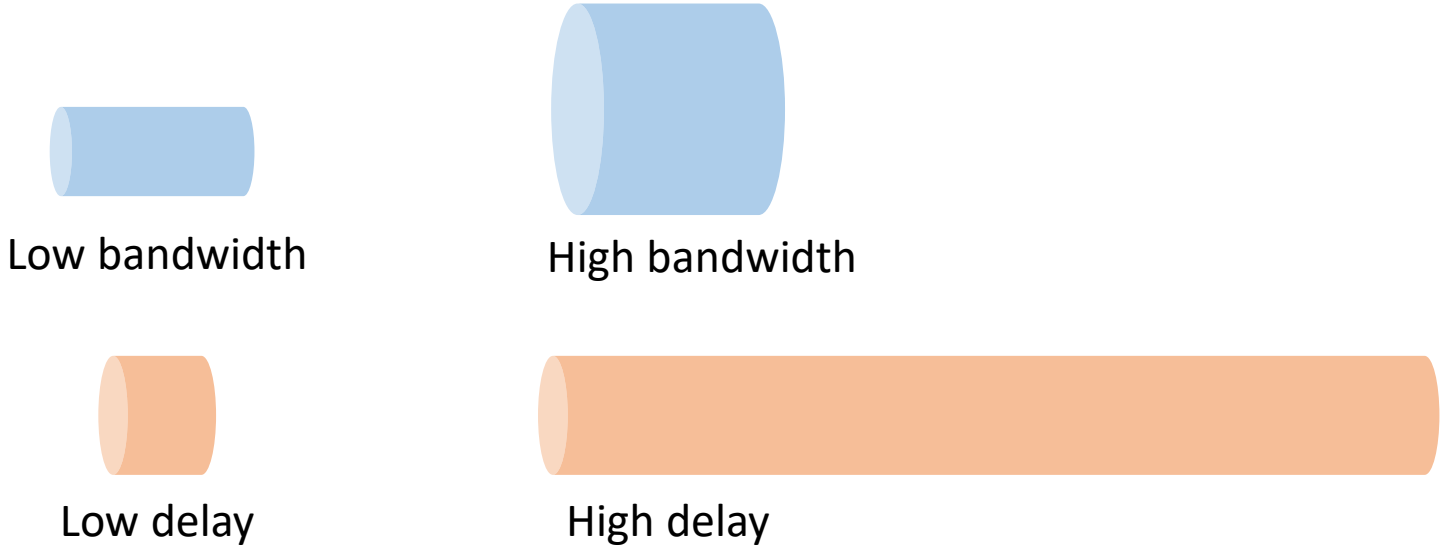


Communication Model



- A simple model of data communication
  - A message of data is transmitted through capacity-limited channel
  - It is broken into multiple chunks of data, each of which can be transmitted at a time
  - Bandwidth = Units of data can be transmitted at a time
    - Gap = 1/Bandwidth
    - Message latency ≈ Latency + #Chuncks × 1/Bandwidth

Bandwidth vs Latency

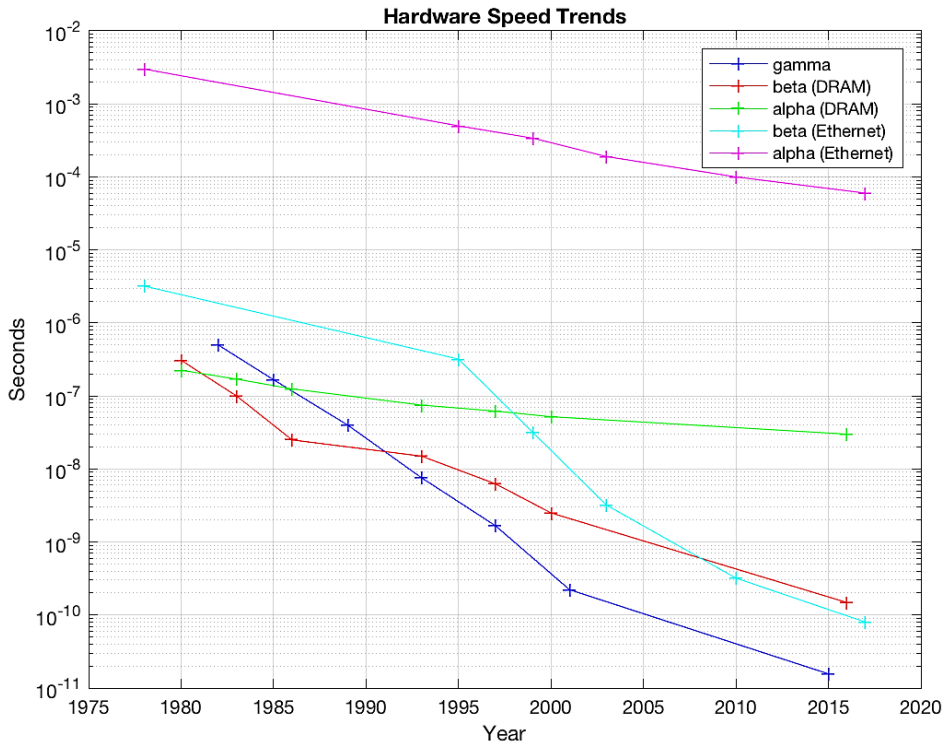


- Bandwidth
  - The amount of data can be moved simultaneously
  - Narrow pipe vs. Fat pipe
- Latency
  - The delivery time of transmitting data
  - Short pipe vs. Long pipe

Computational Performance

- γ (Time per FLOP) $\ll$ β (1/Bandwidth) $\ll$ α (Latency)
  - Gaps growing exponentially over time
- Running time of an algorithm
  - Running time $\approx$ Execution time + Waiting time + Delivery time
  - Running time $\approx$ #FLOPs $\times$ γ + #words_moved $\times$ β + #messages $\times$ α

	Capacity	Latency	Cost/GB
Register	1000 bits	20ps	Very Costly
SRAM	10KB-10MB	1-10ns	$1000
DRAM	10GB	80ns	$100
Flash	100GB	100µs	$1
Hard disk	1TB	10ms	$0.1



Computational Performance

- γ (Time per FLOP) $\ll$ β (1/Bandwidth) $\ll$ α (Latency)
  - Topology is assumed irrelevant
  - One long message is cheaper than many short ones
  - Can do hundreds or thousands of flops for cost of one message
- Computational intensity
  - m: the number of memory elements (#words_moved) moved between fast and slow memory, assuming #messages < #words_moved
  - f: the number of arithmetic operations (#FLOPs)
  - **(Computational intensity)**
    - **CI = f/m**
  - A large computational intensity means more efficiency

Common Matrix Computations

- Matrix-vector multiplication
  - $y = y + A \cdot x$

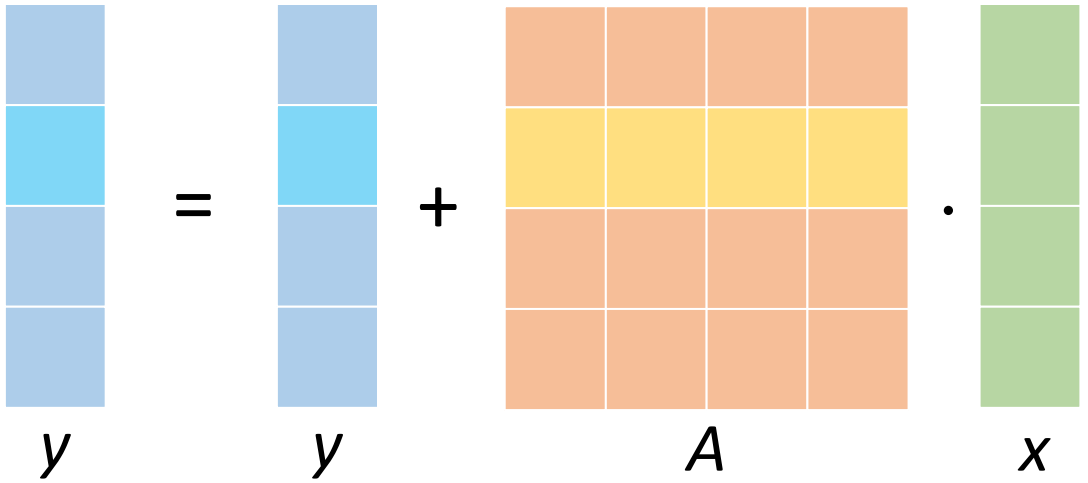
$$\begin{pmatrix} y_0 \\ \vdots \\ y_{n-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ \vdots \\ y_{n-1} \end{pmatrix} + \begin{pmatrix} A_{0,0} & \cdots & A_{0,n-1} \\ \vdots & \ddots & \vdots \\ A_{n-1,0} & \cdots & A_{n-1,n-1} \end{pmatrix} \cdot \begin{pmatrix} x_0 \\ \vdots \\ x_{n-1} \end{pmatrix}$$

- Matrix-matrix multiplication
  - $C = C + A \cdot B$

$$\begin{pmatrix} C_{0,0} & \cdots & C_{0,n-1} \\ \vdots & \ddots & \vdots \\ C_{n-1,0} & \cdots & C_{n-1,n-1} \end{pmatrix} = \begin{pmatrix} C_{0,0} & \cdots & C_{0,n-1} \\ \vdots & \ddots & \vdots \\ C_{n-1,0} & \cdots & C_{n-1,n-1} \end{pmatrix} + \begin{pmatrix} A_{0,0} & \cdots & A_{0,n-1} \\ \vdots & \ddots & \vdots \\ A_{n-1,0} & \cdots & A_{n-1,n-1} \end{pmatrix} \cdot \begin{pmatrix} B_{0,0} & \cdots & B_{0,n-1} \\ \vdots & \ddots & \vdots \\ B_{n-1,0} & \cdots & B_{n-1,n-1} \end{pmatrix}$$

Matrix-Vector Multiplication

- Setting:
  - Input:
    - y: n-D vector,
    - x: n-D vector,
    - A: $n \times n$ matrix
  - Compute:
    - $y = y + A \cdot x$




```
for i = 1:n
    for j = 1:n
        y[i] = y[i] + A[i,j]*x[j]
```

Where is the I/O with memory?

Matrix-Vector Multiplication

- Assumptions:
  - Constant computation rate
  - Fast memory can hold three vectors
  - The cost of fast memory access is zero
    - Nearly zero for registers
    - Small latency for cache
  - Memory latency is constant
- Computational intensity
  - $f = 2n^2$ (Two nested for-loops)
  - $m = 3n + n^2$ (Read/write *y, x, A*)
  - CI ≈ 2
    - Very inefficient, and limited by slow memory speed

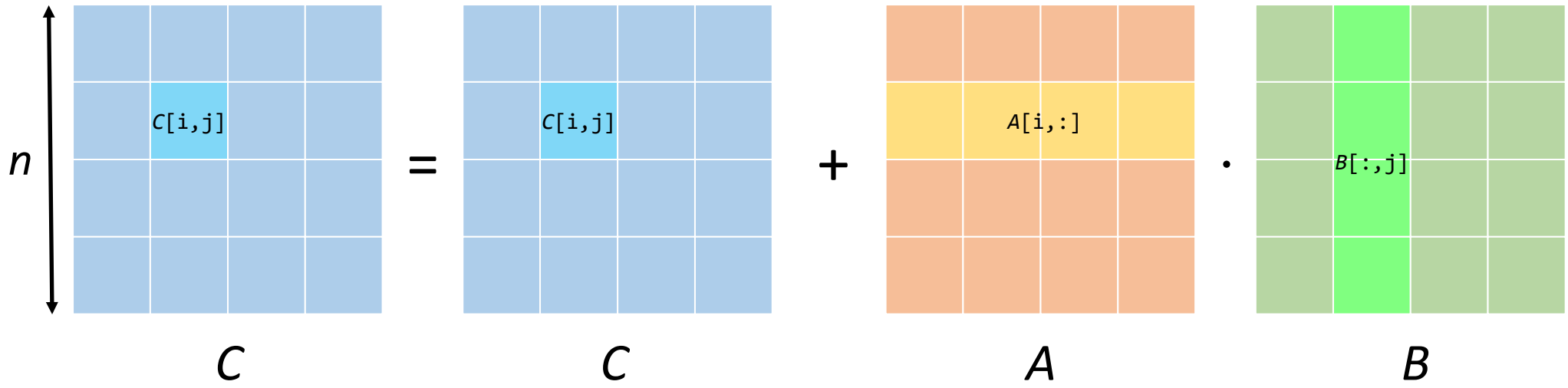
```
{read x[1:n] into fast memory}
{read y[1:n] into fast memory}
for i = 1:n
    {read i-th row of A into fast memory}
    for j = 1:n
        y[i] = y[i] + A[i,j]*x[j]

{write y[1:n] back to slow memory}
```

Matrix-Matrix Multiplication

- Input:
  - C, A, B: $n \times n$ matrices
- Compute:
  - $C = C + A \cdot B$

```
for i = 1 to n
    for j = 1 to n
        for k = 1 to n
            C[i,j] = C[i,j] + A[i,k] * B[k,j]
```

Where is the I/O with memory?

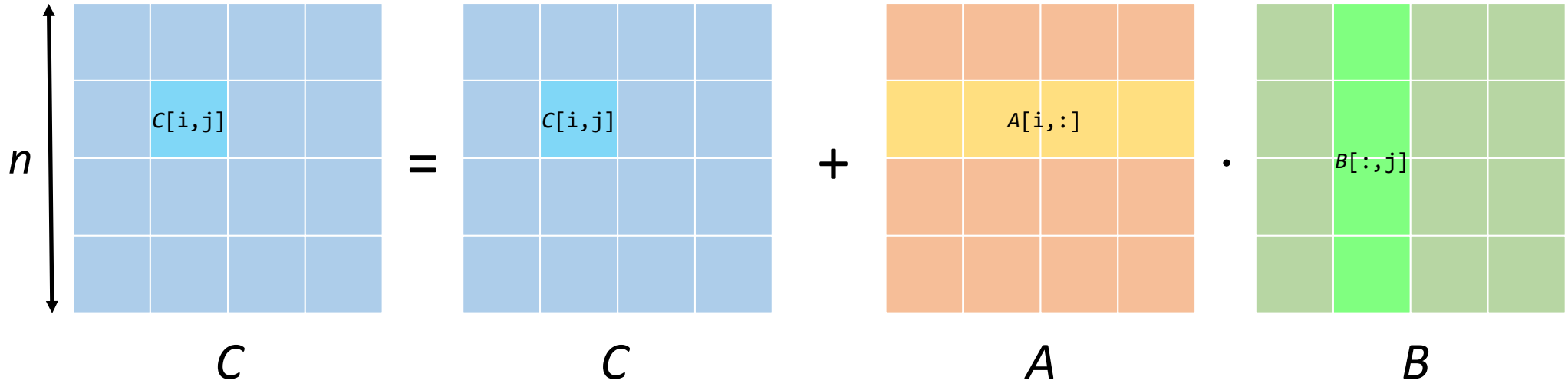


Matrix-Matrix Multiplication

- Computational intensity
  - $f = 2n^3$, $m = 3n^2$ (guess)
  - Can CI = O(n)?
- Reality:
  - Fast memory can hold only vectors, not matrices

```
for i = 1 to n
    for j = 1 to n
        for k = 1 to n
            C[i,j] = C[i,j] + A[i,k] * B[k,j]
```

Where is the I/O with memory?



Matrix-Matrix Multiplication

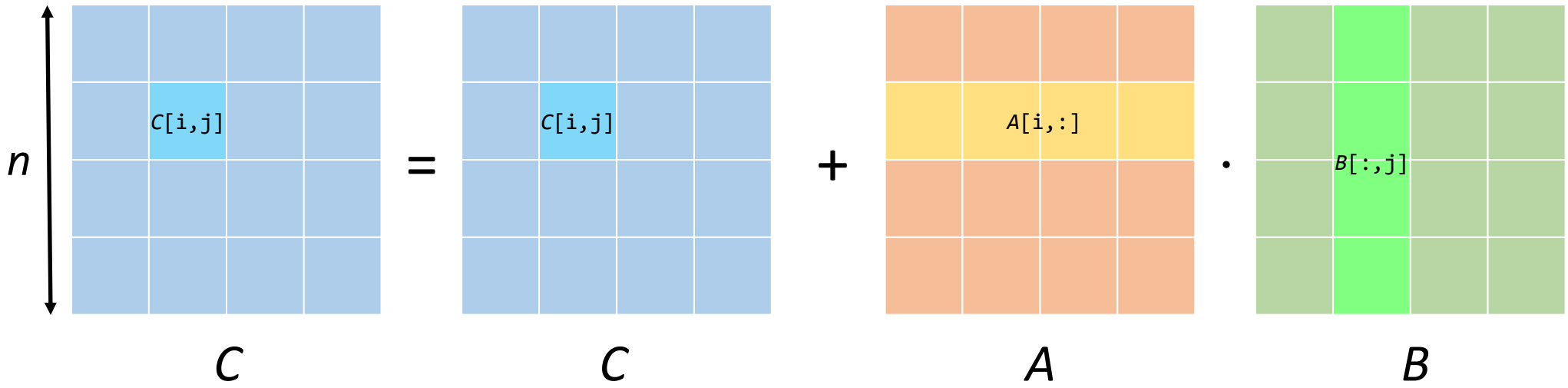
- Computational intensity
  - $m = n^2 + n^3 + 2n^2$

    (Read each row A

    +Read each column B n times

    + Read/write each cell C)
  - CI ≈ 2 (very inefficient)

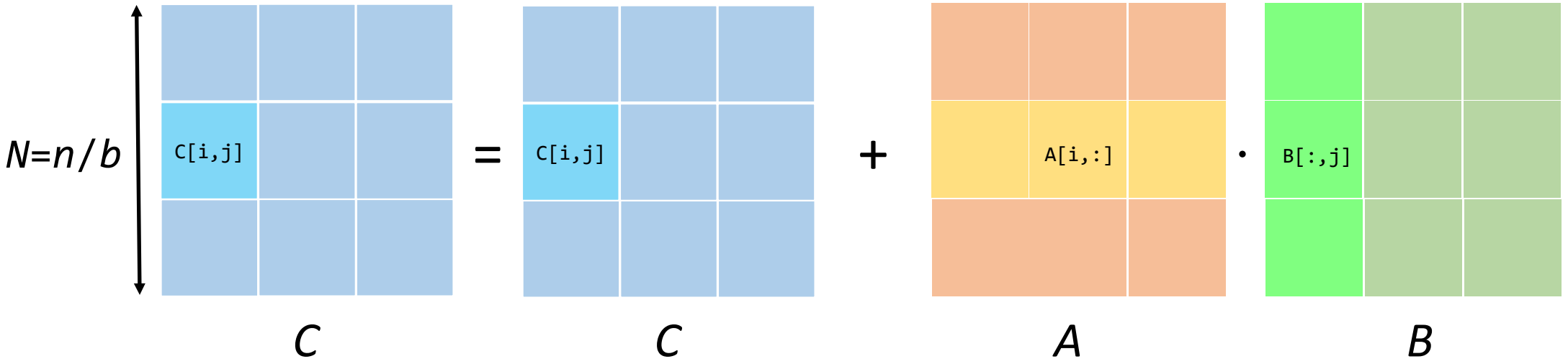
```
for i = 1 to n
    {read i-th row of A into fast memory}
    for j = 1 to n
        {read C[i,j] into fast memory}
        {read j-th column of B into fast memory}
        for k = 1 to n
            C[i,j] = C[i,j] + A[i,k] * B[k,j]
    {write C[i,j] back to slow memory}
```



Blocked Matrix-Matrix Multiplication

- Assumption:
  - Fast memory can hold the size of $O(b^2)$ elements
- Consider blocked matrix multiplication
  - `C[i,j]` is a $b \times b$ sub-matrix ⚠

```
for i = 1 to N
    for j = 1 to N
    {read block C[i,j] into fast memory}
    for k = 1 to N
        {read block A[i,k] into fast memory}
        {read block B[k,j] into fast memory}
            C[i,j] = C[i,j] + A[i,k] * B[k,j]
            /*do a matrix multiply on blocks*/
    {write block C[i,j] back to slow memory}
```

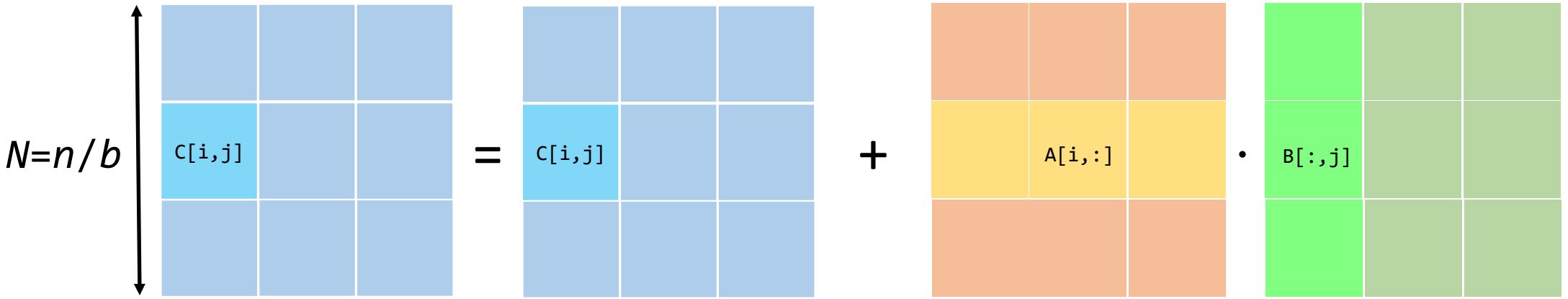


Blocked Matrix-Matrix Multiplication

- Computational intensity
  - $m = 2N^2 b^2 + N^3 b^2 + N^3 b^2$

    (Read/write each block C
    + Read each block A N times
    + Read each block B N times)
  - CI $\approx 2n^3 / (2N^3 b^2) \approx b$

```
for i = 1 to N
    for j = 1 to N
    {read block C[i,j] into fast memory}
    for k = 1 to N
        {read block A[i,k] into fast memory}
        {read block B[k,j] into fast memory}
            C[i,j] = C[i,j] + A[i,k] * B[k,j]
            /*do a matrix multiply on blocks*/
    {write block C[i,j] back to slow memory}
```



Matrix-Matrix Multiplication

- Simple matrix multiplication
  - Assumption: Fast memory can hold O(n) elements
  - CI ≈ 2
  - Very inefficient

```
for i = 1 to n
    {read i-th row of A into fast memory}
    for j = 1 to n
        {read C[i,j] into fast memory}
        {read j-th column of B into fast memory}
        for k = 1 to n
            C[i,j] = C[i,j] + A[i,k] * B[k,j]
    {write C[i,j] back to slow memory}
```

- Blocked matrix multiplication
  - Assumption: Fast memory can hold O(b^2) elements
  - CI $\approx b$
    - If $b \approx \sqrt{n}$, CI $\approx \sqrt{n}$
    - Much more efficient

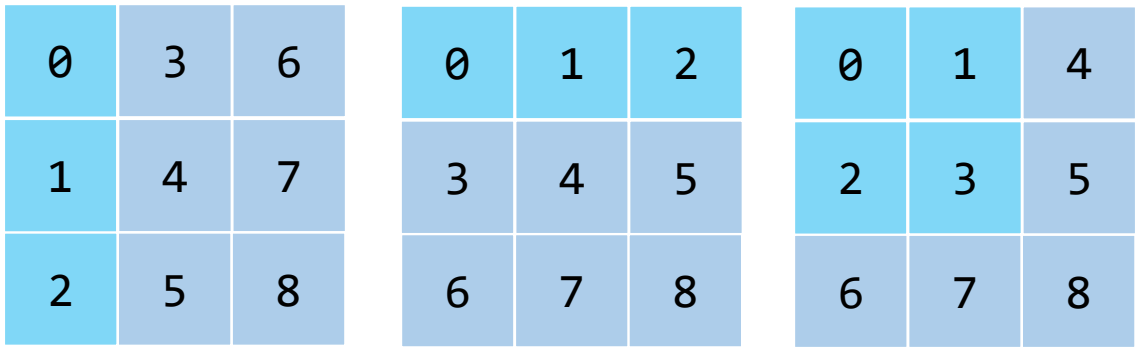
```
for i = 1 to N
    for j = 1 to N
    {read block C[i,j] into fast memory}
    for k = 1 to N
        {read block A[i,k] into fast memory}
        {read block B[k,j] into fast memory}
            C[i,j] = C[i,j] + A[i,k] * B[k,j]
            /*do a matrix multiply on blocks*/
    {write block C[i,j] back to slow memory}
```

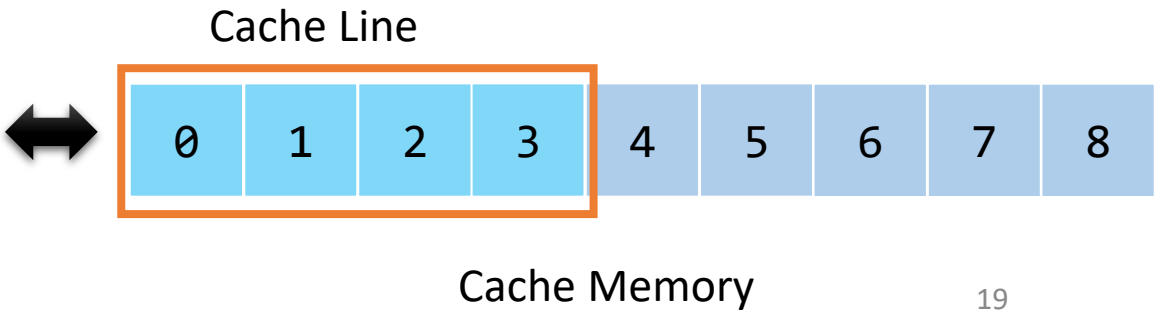
How to Implement in Practice?

- Tuning code can be tedious
  - Many optimizations besides blocking
  - Behavior of machine and compiler hard to predict
- Approach #1: Analytical performance models
  - Use model (of cache, memory costs, etc.) to select best code
  - But model needs to be both simple and accurate
- Approach #2: “Autotuning”
  - Let computer generate large set of possible code variations
  - Search for the fastest ones (may be matrix size dependent)
  - Sometimes all done “off-line,” sometimes at run-time

Data Layout and Caching

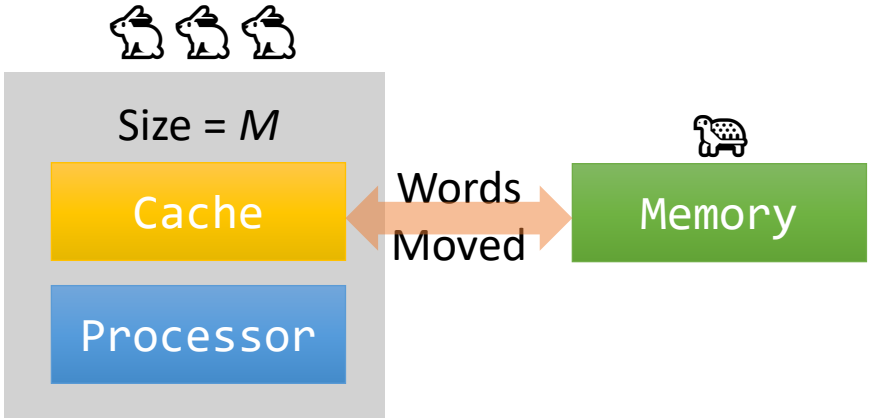
- A matrix is a 2-D array of elements, but is mapped to 1-D memory addresses
- Matrix data layouts:
  - By column, or "column major": `A[i,j]= A[i+j×n]`
  - By row, or "row major" (C default): `A[i,j] at A[i×n+j]`
  - Reorganized based on blocks
- Beware of cache line: caching based on spatial locality in memory addresses





Questions

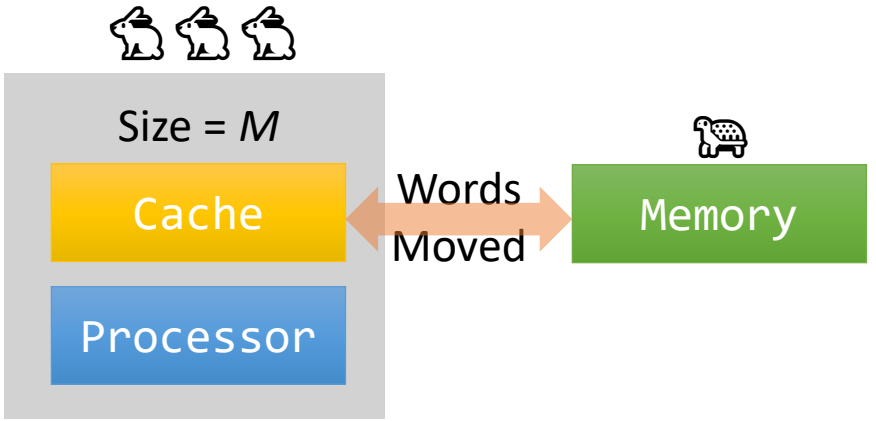
- Fundamental questions
  - Is blocked matrix multiplication optimal in communication with memory?
  - Is there any other way to achieve better communication?
  - How about parallel matrix multiplication with multiple processors?
  - Any lower bound for the communication costs of sequential and parallel computations?



- Let us apply some formal analysis and mathematics…

Lower Bound of Communication

- How much data must be transferred to/from fast memory to enable computation?
  - Size of fast memory is limited : M
  - Reduce communication to improve running time
  - Matrix-matrix multiplication
    - Execution takes $O(n^3)$, input data takes $O(n^2)$
    - How much data is required to transfer?
      - Need more than $O(n^2)$, if $M \ll n^2$
      - In practice, $M \sim n$
- Consider blocked matrix multiplication:
  - If $b^2 = O(M)$, then #words_moved = $\Omega(N^3 b^2)$
  - Blocked matrix multiplication is communication optimal



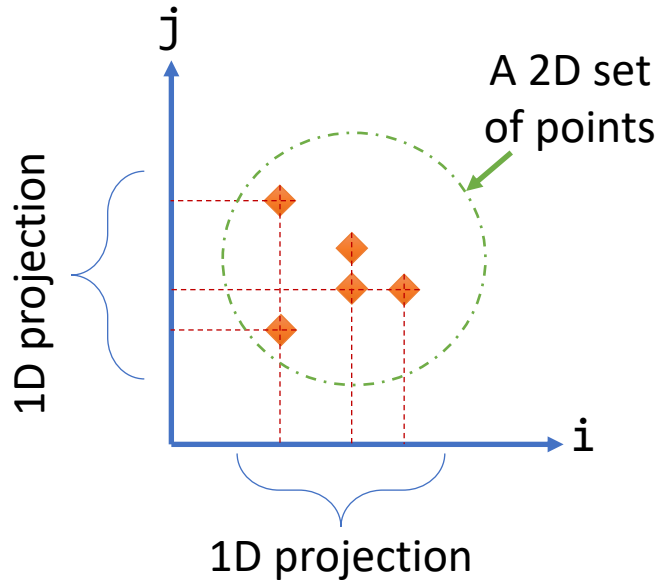
[Theorem]

#words_moved = $\Omega\left(\frac{n^3}{\sqrt{M}}\right)$

Proof Idea of Lower Bound

- Matrix-matrix multiplication has 3 nested loops
- First, get some intuitions
  - We consider a simpler setting with only 2 nested loops to compute `C[i,j]` via abstract function `f()`
  - We may reorganize data communication in different ways to provide inputs `A[i]` and `B[j]`
  - Assume we read a set {`A[i]`} and a set {`B[j]`} into fast memory
    - How many points of `C[i,j]` can be computed?
  - Equivalent question
    - Given two 1D projections of a 2D set of points, how many maximal points are in the 2D set?

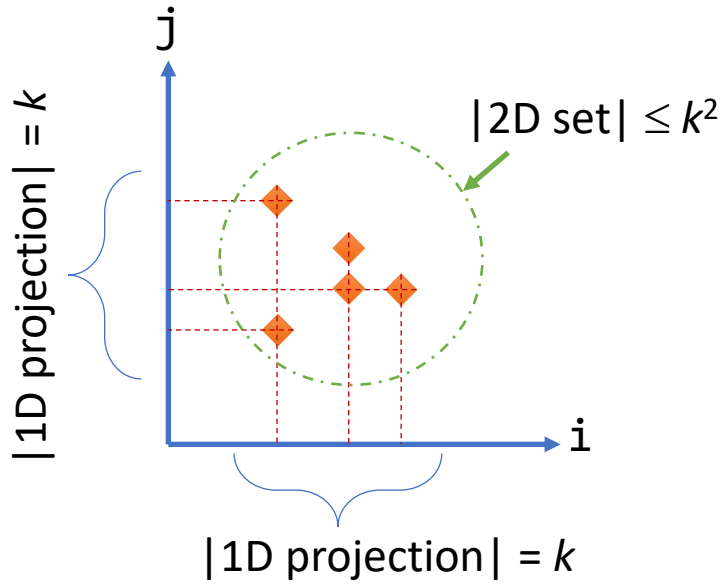
```
for i = 1 to n
    for j = 1 to n
      C[i,j] = f(A[i],B[j])
```



Proof Idea of Lower Bound

- How many maximal points are in the 2D set, given two 1D projections?
  - If there are k points in each 1D projection, then the maximal number of points in 2D $\leq k^2$
    - Maximum number of points when the 2D set is a square
- If size of fast memory $\leq M$, then how many reads are needed to compute `C[i,j]` for $(1 \leq i \leq n)$?
  - Totally, n^2 data points are needed for `C[i,j]`
  - Each time, only M points can be read into fast memory
    - Allow at most $M^2/2^2$ points to be computed for `C[i,j]`
    - Need at least $\frac{4n^2}{M^2}$ reads
      - Each reads M points of input into fast memory
  - Hence, #words_moved = $\Omega(n^2/M)$

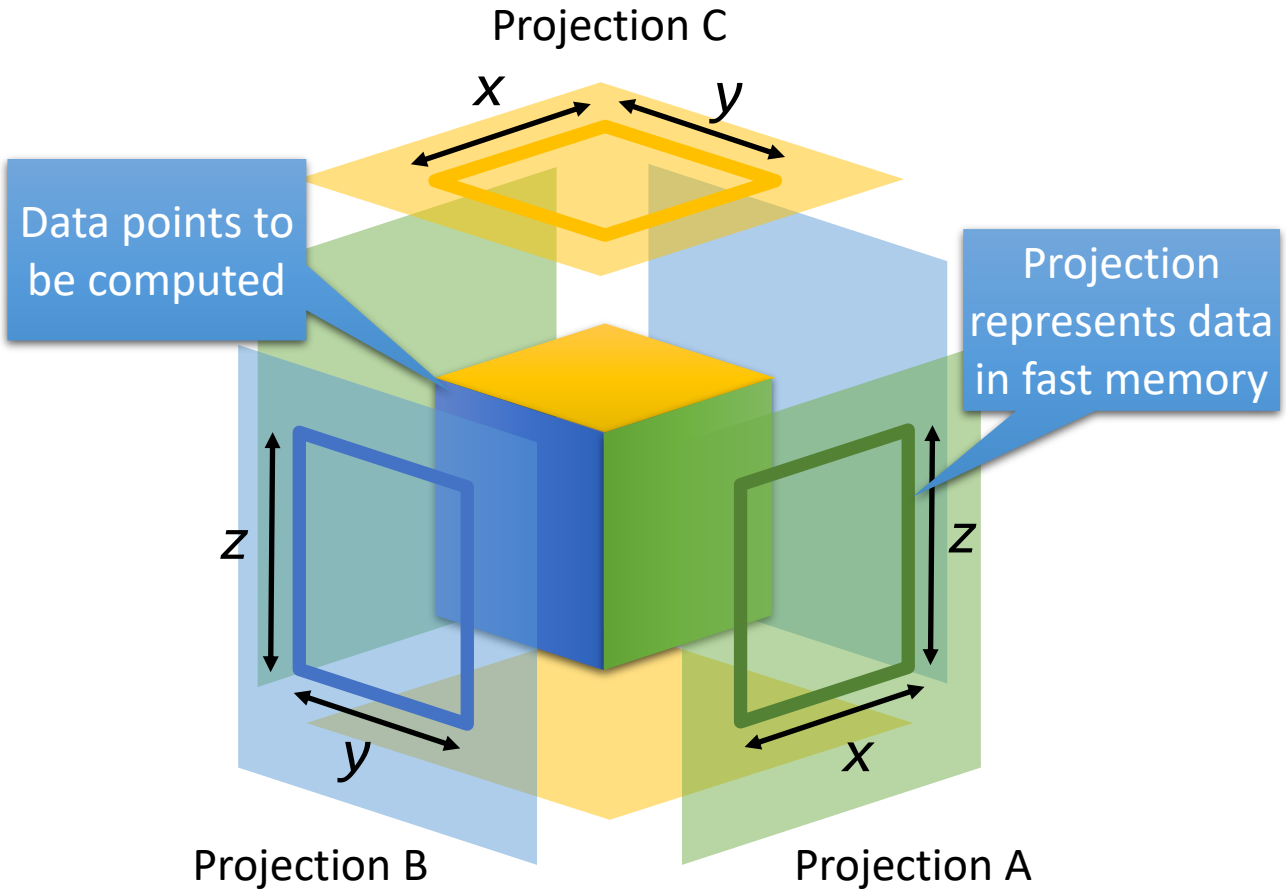
```
for i = 1 to n
    for j = 1 to n
      C[i,j] = f(A[i],B[j])
```



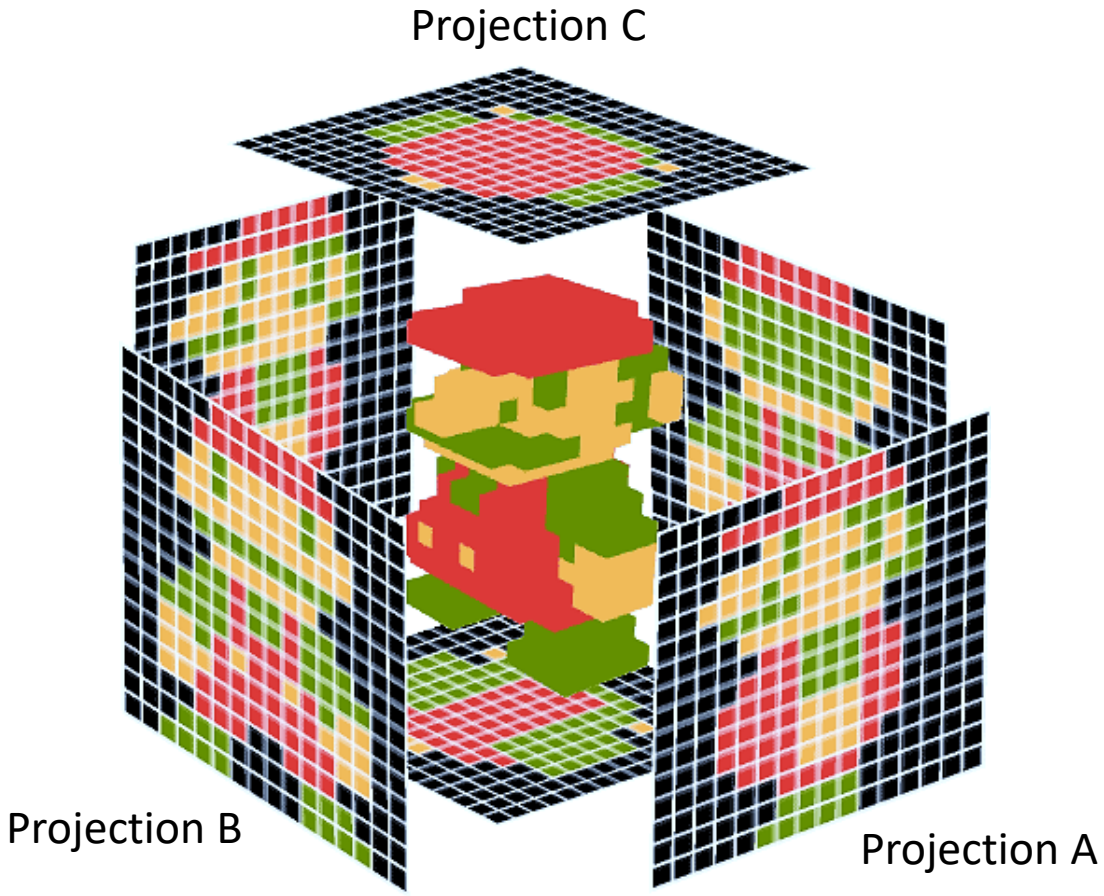
Proof Idea of Lower Bound

- Now consider matrix-matrix multiplication
  - Totally, n^3 data points for computing `C[i,j]+A[i,k]*B[k,j]`
  - Each time, only *M* points can be read into fast memory
  - How many points of (`C[i,j]+A[i,k]*B[k,j]`) can be computed by only *M* points in fast memory at a time?

```
for i = 1 to n
   for j = 1 to n
      for k = 1 to n
         C[i,j] = C[i,j] + A[i,k] * B[k,j]
```



A box with lengths *x, y, z*
Volume of the box
$= x \cdot y \cdot z = (xz \cdot zy \cdot yx)^{1/2}$
$= (|\text{Proj A}| \cdot |\text{Proj B}| \cdot |\text{Proj C}|)^{1/2}$



Loomis-Whitney Inequality:
Volume of 3D object
$\leq (|\text{Proj A}| \cdot |\text{Proj B}| \cdot |\text{Proj C}|)^{1/2}$

Proof Idea of Lower Bound

- Now consider matrix multiplication
  - Totally, n^3 data points are needed for computing `C[i,j]+A[i,k]*B[k,j]`
  - Each time, only M points of input can be read into fast memory
  - Allow at most $c\cdot(M\cdot M\cdot M)^{1/2}$ data points to be computed for `C[i,j]`
  - Need at least $\frac{n^3}{c\cdot M^{\frac{3}{2}}}$ reads
    - Each reads M points (i.e., words) into fast memory
  - Hence, #words_moved = $\Omega\left(\frac{n^3}{\sqrt{M}}\right)$

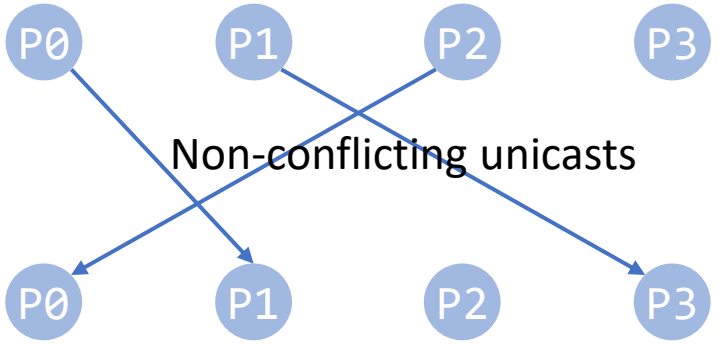
[Theorem]

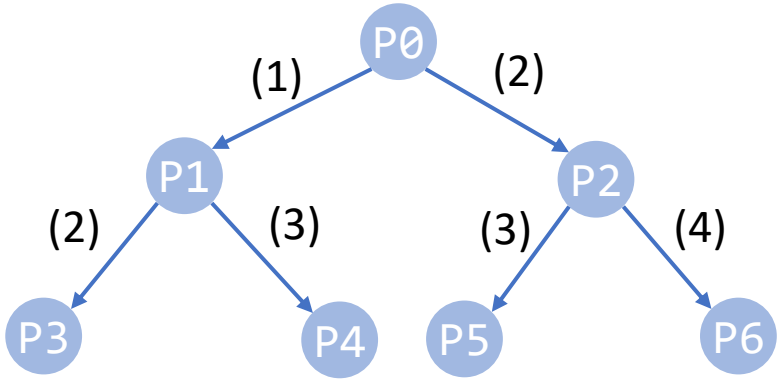
#words_moved = $\Omega\left(\frac{n^3}{\sqrt{M}}\right)$

```
for i = 1 to n
   for j = 1 to n
      for k = 1 to n
         C[i,j] = C[i,j] + A[i,k] * B[k,j]
```

Parallel Matrix Multiplication

- Parallel case: There are *P* processors
  - Each processor's fast memory has a limited capacity = *M*
  - Consider the following communication model of message-passing
    - Non-conflicting unicasts are allowed simultaneously
    - Broadcast is restricted
      - Only unicast is used at a time
      - Efficient broadcast has overhead of 2 log(*n*)



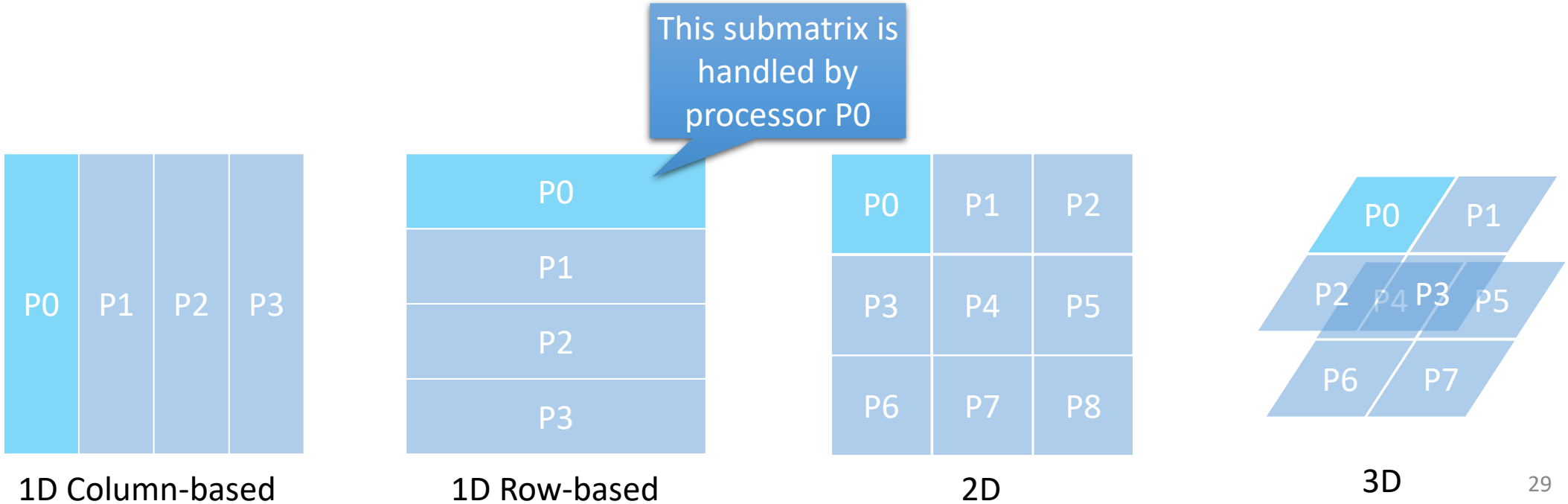


Parallel Matrix Multiplication

- Lower bound
  - The fast memory has a limited capacity = *M*
  - Sequential case: #words_moved = $\Omega\left(\frac{n^3}{\sqrt{M}}\right)$
- Parallel case: There are *P* processors
  - Inter-processor communication has a limited capacity = *M*
  - #words_moved per processor = $\Omega\left(\frac{n^3}{P\sqrt{M}}\right)$
  - Assume $M = \mathrm{O}\left(\frac{n^2}{P}\right)$
    - In distributed memory systems, inter-process communication is the bottleneck, rather than the cache
    - Then #words_moved per processor = $\Omega\left(\frac{n^2}{\sqrt{P}}\right)$ - *Can it be achievable?*

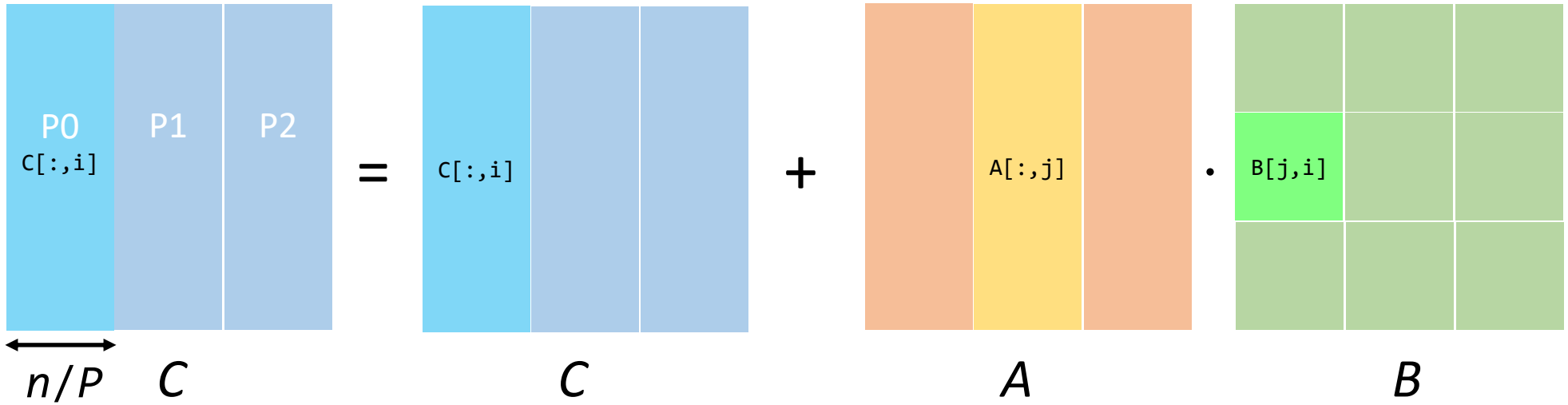
Data Layout

- Different data processes are handled by different processors
  - 1D: Each processor handles a column (or a row) of submatrix
  - 2D: Each processor handles a block of submatrix
  - 3D: Each processor handles a sub-operation of matrix



1D Data Layout

- We consider 1D column-based data layout
  - *P* processors; each processor handles a row of submatrix in *C*, *B*, *A*
  - `A[:,i]` is the (i+1)-th $n \times n/P$ column submatrix that processor *P*i handles
  - `B[i,j]` is the (j+1)-th $n/P \times n/P$ submatrix of `B[:,i]`
  - We use the formula
    - $\texttt{C[:,i]} = \texttt{C[:,i]} + \texttt{A} \cdot \texttt{B[:,i]} = \texttt{C[:,i]} + \sum_j \texttt{A[:,j]} \cdot \texttt{B[j,i]}$

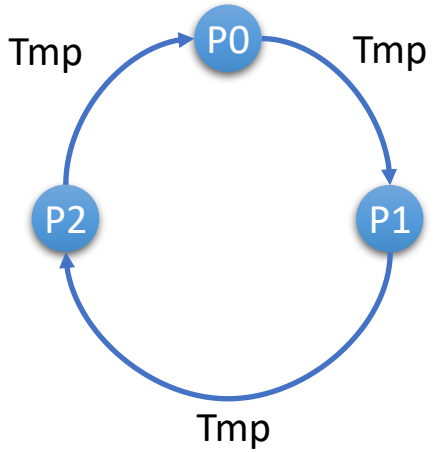


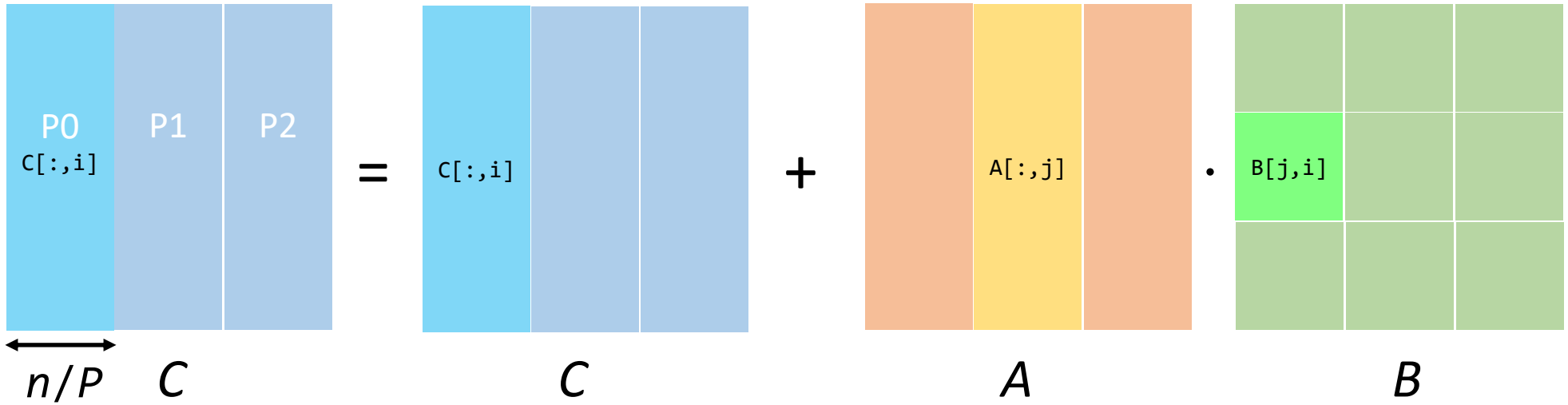
1D Data Layout

```
Execute in parallel with each processor me
    Copy A[:,me] into Tmp
    C[:,me] = C[:,me] + Tmp*B[me,me]

    for j = 1 to P-1
      Send Tmp to processor me+1 mod P
      Receive Tmp from processor me-1 mod P
      C[:,me] = C[:,me] + Tmp*B[me-j mod P, me]
```

Note Tmp is overwritten

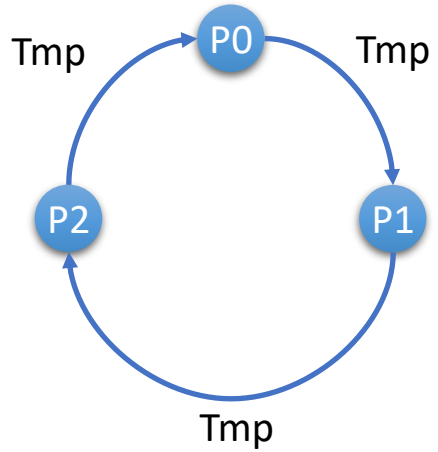
$C[:,i] = C[:,i] + \sum_j A[:,j] \cdot B[j,i]$

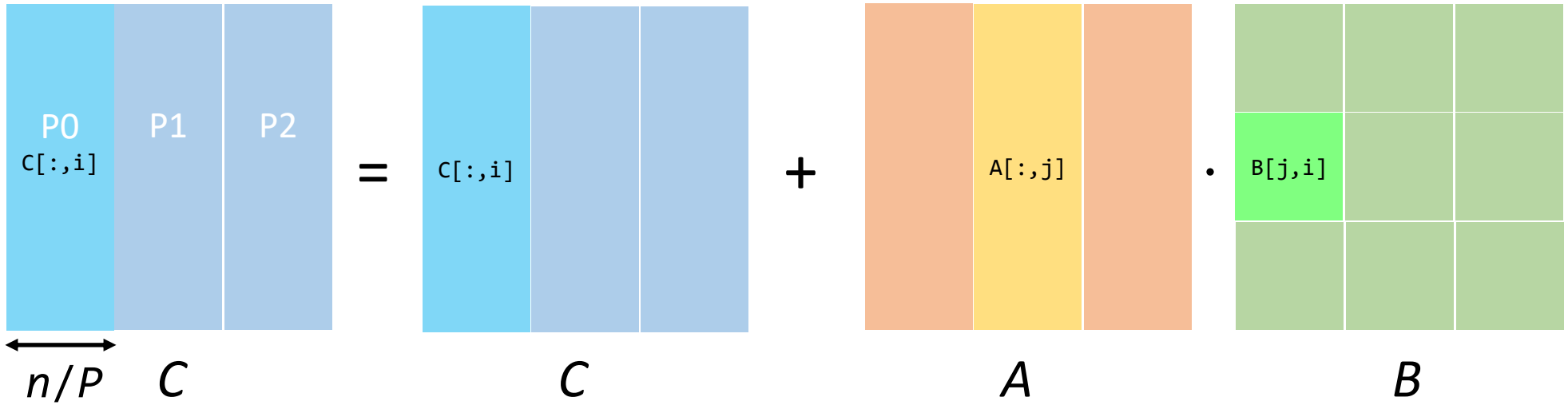




1D Data Layout

```
/* For example, processor P0 */
    Copy A[:,0] into Tmp
    C[:,0] = C[:,0] + Tmp*B[0,0]

    for j = 1 to P-1
      Send Tmp to processor P1
      Receive Tmp from processor P2
      C[:,0] = C[:,0] + Tmp*B[P-j, 0]
```

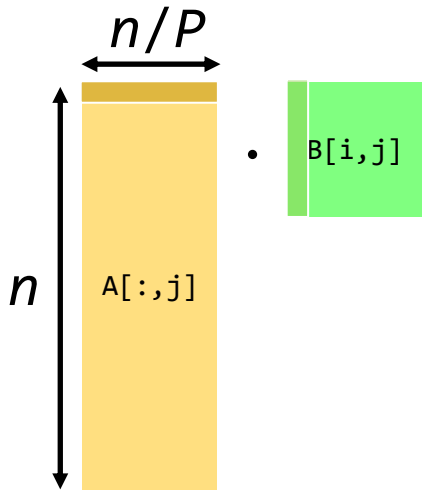




1D Data Layout

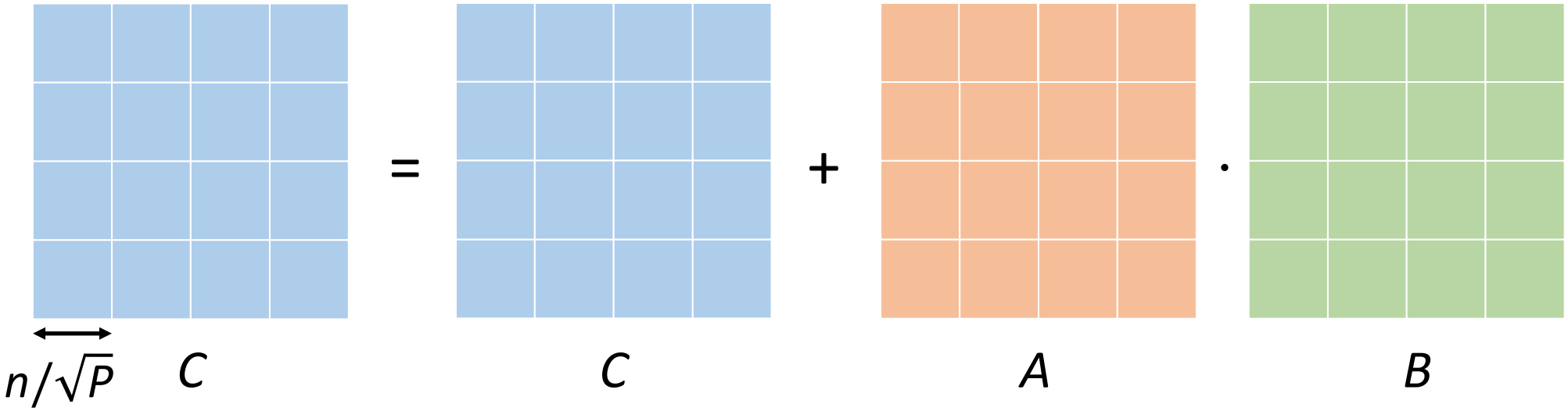
- We consider parallel execution time and communication per processor
  - Execution time = $P \times$ Multiplication_time = $O(P\,(n/P)^2\,n\,) = O(\,n^3/P\,)$
  - #Words_moved = $P \times$ Send_Tmp = $O(P\,(n/P)\,n\,) = O(\,n^2\,)$
  - Good execution time, but inefficient communication (note: lower bound = $\Omega(n^2/\sqrt{P})$)

```
Execute in parallel with each processor me
    Copy A[:,me] into Tmp
    C[:,me] = C[me,:] + Tmp*B[me,me]

    for j = 1 to P-1
      Send Tmp to processor me+1 mod P
      Receive Tmp from processor me-1 mod P
      C[:,me] = C[:,me] + Tmp*B[me-j mod P, me]
```



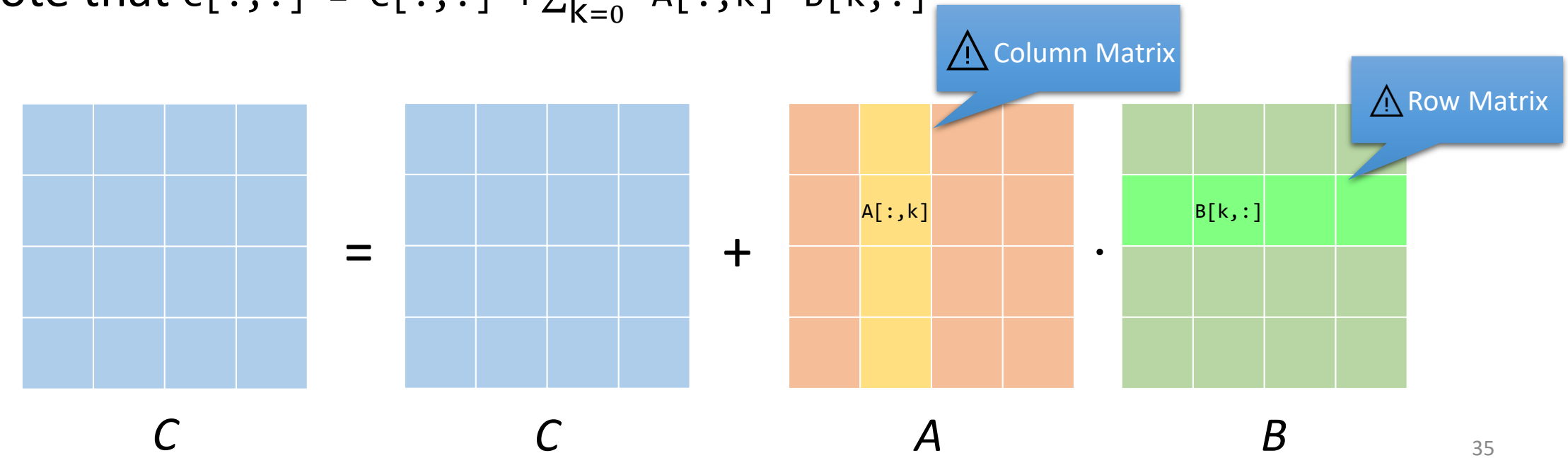
SUMMA

- SUMMA (Scalable Universal Matrix Multiply Algorithm)
  - Assume $\sqrt{P}$ is an integer
  - We consider 2D data layout with $\sqrt{P} \times \sqrt{P}$ processors
  - Each processor is labeled by `[i,j]`, where $0 \leq \texttt{i,j} \leq \sqrt{P}$
  - Each processor handles a $n/\sqrt{P} \times n/\sqrt{P}$ submatrix in C, B, A



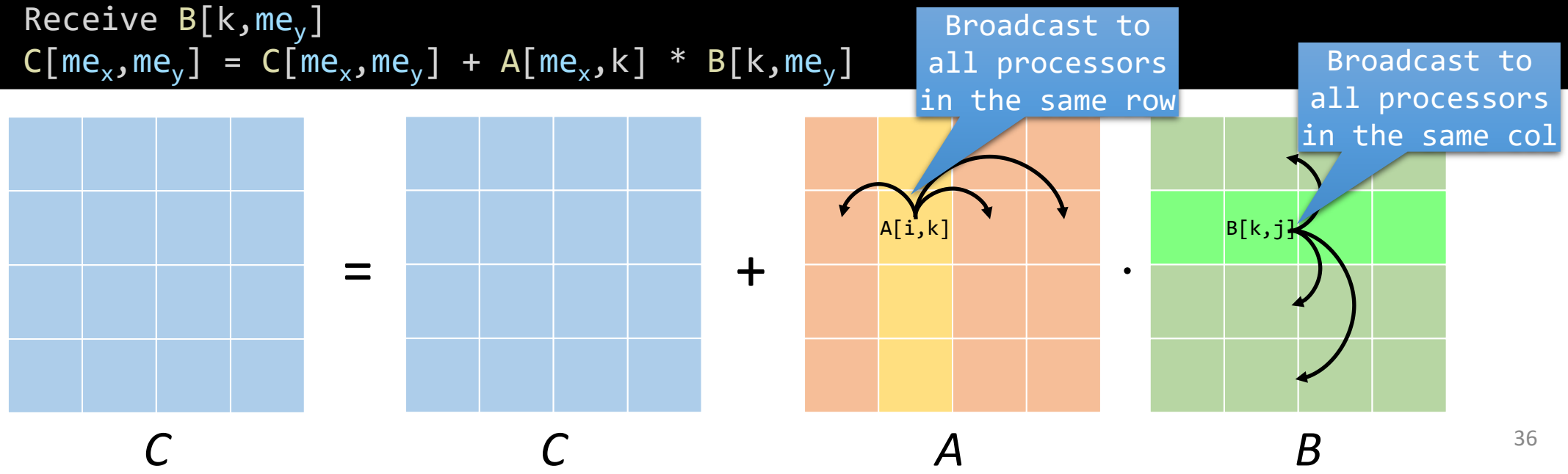
SUMMA

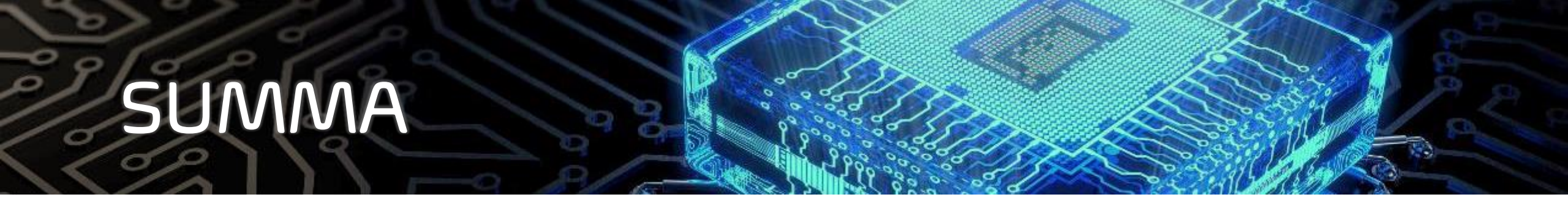
- ***Outer*** product
  - $\begin{pmatrix} u_0 \\ \vdots \\ u_{n-1} \end{pmatrix} \cdot \begin{pmatrix} v_0 & \cdots & v_{n-1} \end{pmatrix} = \begin{pmatrix} u_0 v_0 & \cdots & u_0 v_{n-1} \\ \vdots & \ddots & \vdots \\ u_{n-1} v_0 & \cdots & u_{n-1} v_{n-1} \end{pmatrix}$
- Note that `C[:,:] = C[:,:] +` $\sum_{k=0}^{\sqrt{P}-1}$ `A[:,k] · B[k,:]`



SUMMA

```
Execute in parallel with each processor (mex,mey)
for k = 0 to √P -1
     for all i = 0 to √P-1
            owner of A[i,k] broadcasts it to all processors in the same row
     for all j = 0 to √P-1
            owner of B[k,j] broadcasts it to all processors in the same column
     Receive A[mex,k]
     Receive B[k,mey]
     C[mex,mey] = C[mex,mey] + A[mex,k] * B[k,mey]
```





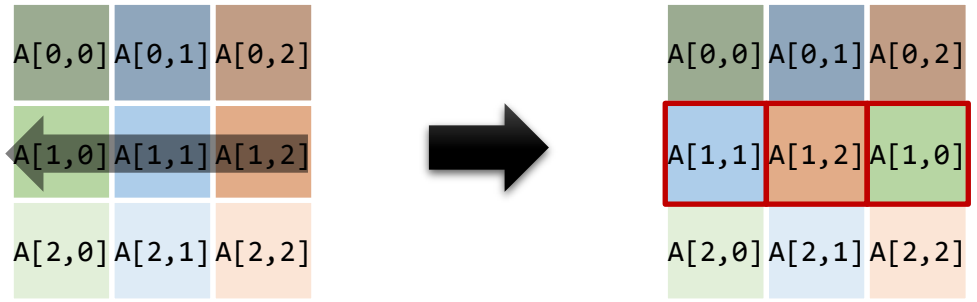
SUMMA

- We consider parallel execution time and communication per processor
  - Execution time = $\sqrt{P} \times$ Multiplication_time = $O(\sqrt{P}\,(n/\sqrt{P})^3\,) = O(\,n^3/P\,)$
  - Broadcast can be achieved with $O(\log(\sqrt{P}))$ communication overhead
  - #Words_moved =$O(\log(\sqrt{P}) \times \sqrt{P} \times |A[i,j]|) = O(\log(\sqrt{P})\sqrt{P}\,(n/\sqrt{P})^2) = O(\log(\sqrt{P})\,n^2/\sqrt{P})$
  - Good execution time, and quite efficient communication (but lower bound $\Omega(n^2/\sqrt{P})$)

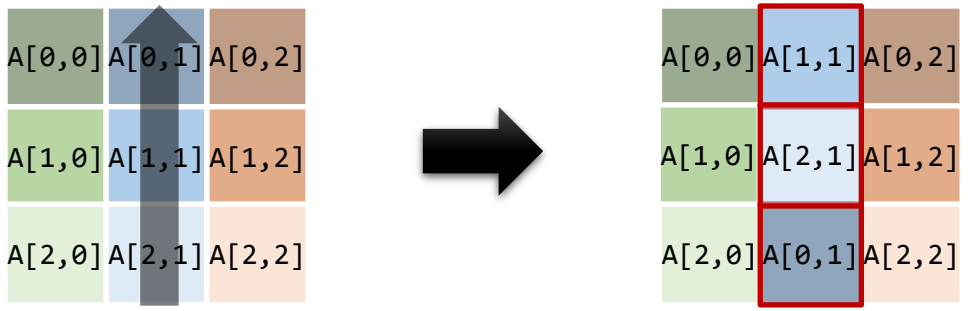
```
Execute in parallel with each processor (mex,mey)
for k = 0 to √P -1
     for all i = 0 to √P-1
            owner of A[i,k] broadcasts it to all processors in the same row
     for all j = 0 to √P-1
            owner of B[k,j] broadcasts it to all processors in the same column
     Receive A[mex,k]
     Receive B[k,mey]
     C[mex,mey] = C[mex,mey] + A[mex,k] * B[k,mey]
```

Cannon's Algorithm

- left-circular-shift by one:
  - Move the leftmost item to the rightmost position and shift other items to the left position



- up-circular-shift by one:
  - Move the topmost item to the bottom position and shift all other numbers to the lower position

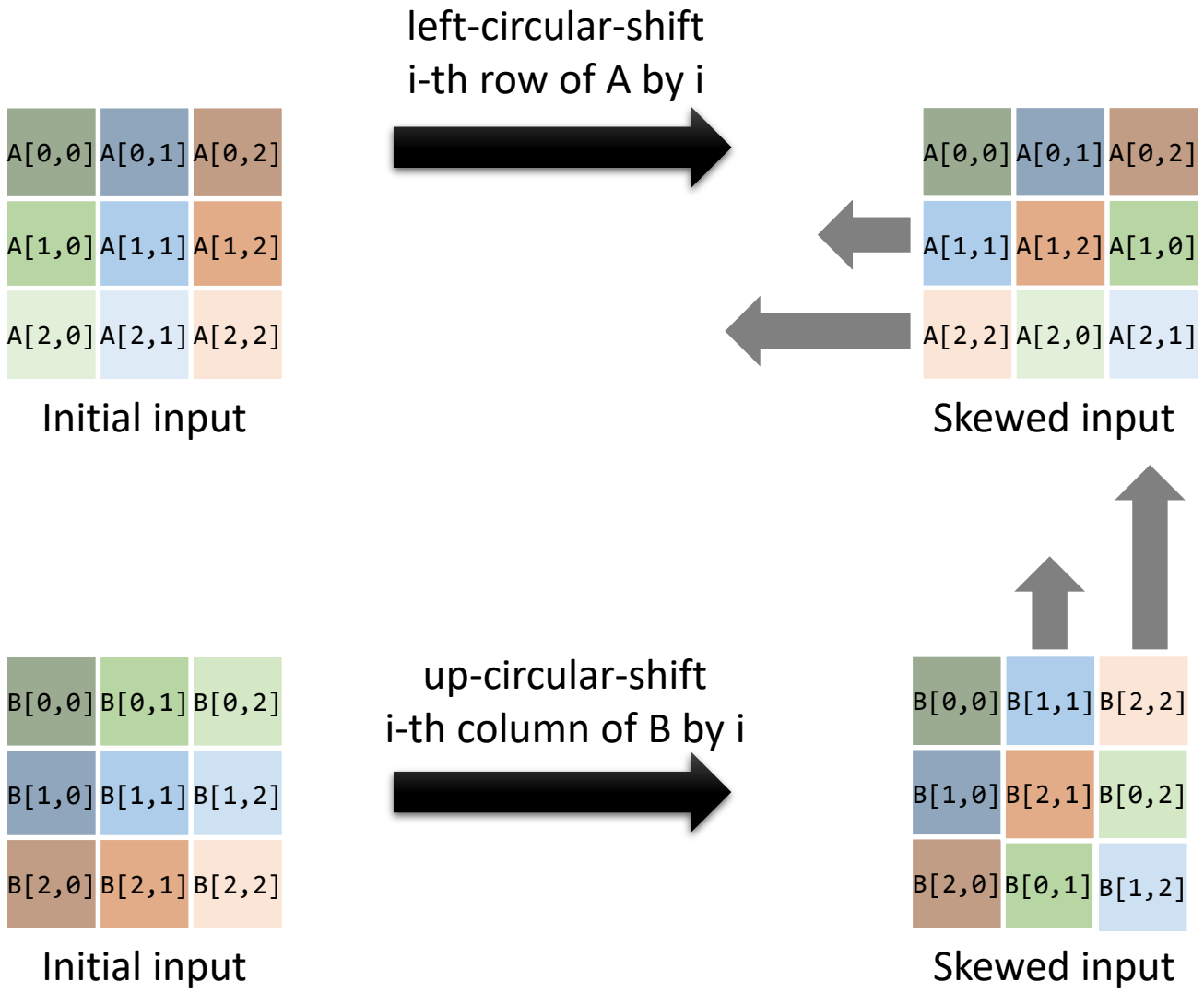


Cannon's Algorithm

```
for i = 0 to √P -1                    /* "skew" A  */
     left-circular-shift i-th row of A by i /* Communication within each row */
     Overwrite A[i,j] by A[i,(j+i) mod √P]
for i = 0 to √P -1                    /* "skew" B */
     up-circular-shift i-th column of B by i /* Communication within each col */
     Overwrite B[i,j] by B[(i+j) mod √P), j]
for k = 0 to √P -1
     Execute in parallel with each processor (i,j)
          C[i,j] = C[i,j] + A[i,j]*B[i,j]

     left-circular-shift each row of A by 1
     Overwrite A[i,j] by A[i,(j+1) mod √P]

     up-circular-shift each column of B by 1
     Overwrite B[i,j] by B[(i+1) mod √P), j]
```

Cannon's Algorithm

- Step 0:

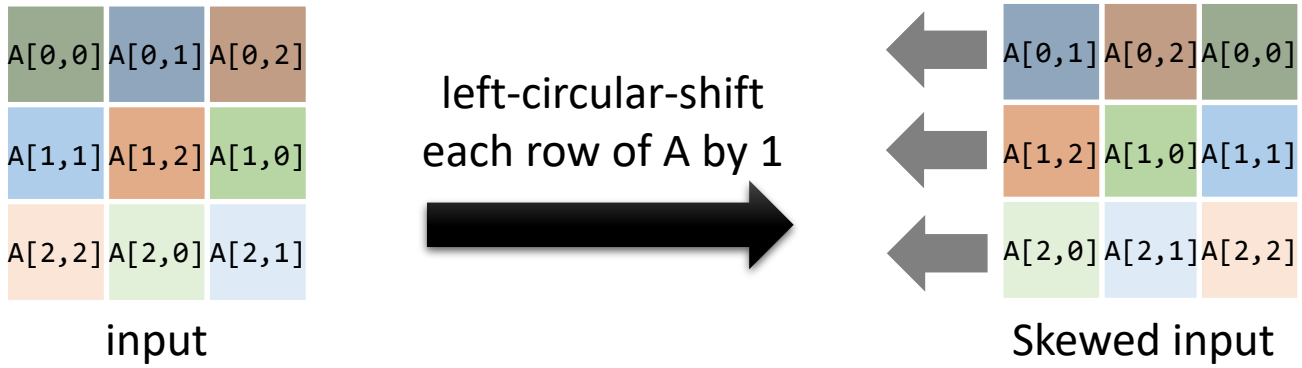


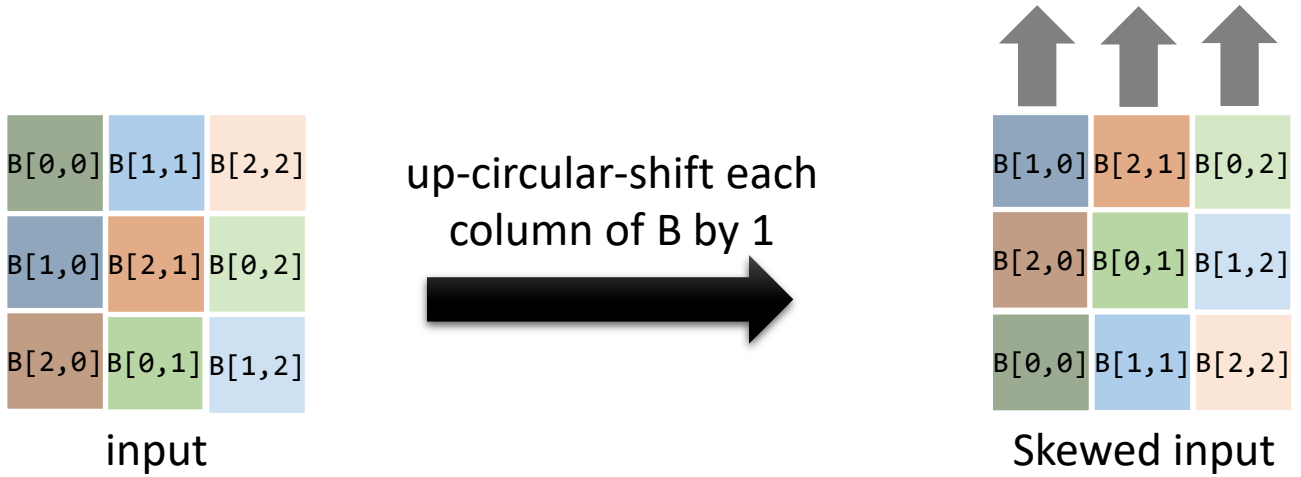
Cannon's Algorithm

- Step 1.1: Compute `C[i,j]=C[i,j]+A[i,j]*B[i,j]` at each processor `(i,j)`

- Step 1.2:

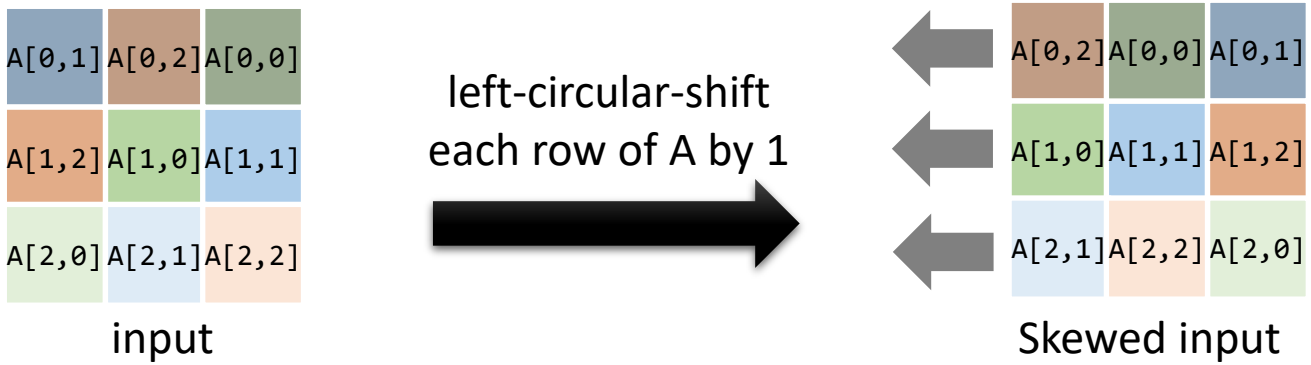
Ignore overwriting for clarity

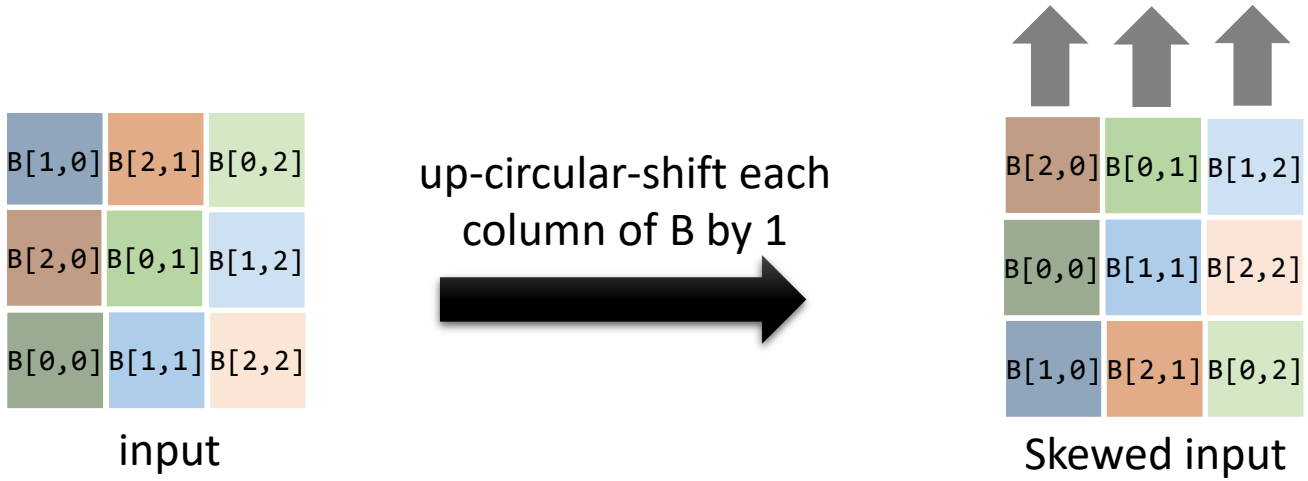




Cannon's Algorithm

- Step 2.1: Compute `C[i,j]=C[i,j]+A[i,j]*B[i,j]` at each processor `(i,j)`
- Step 2.2:





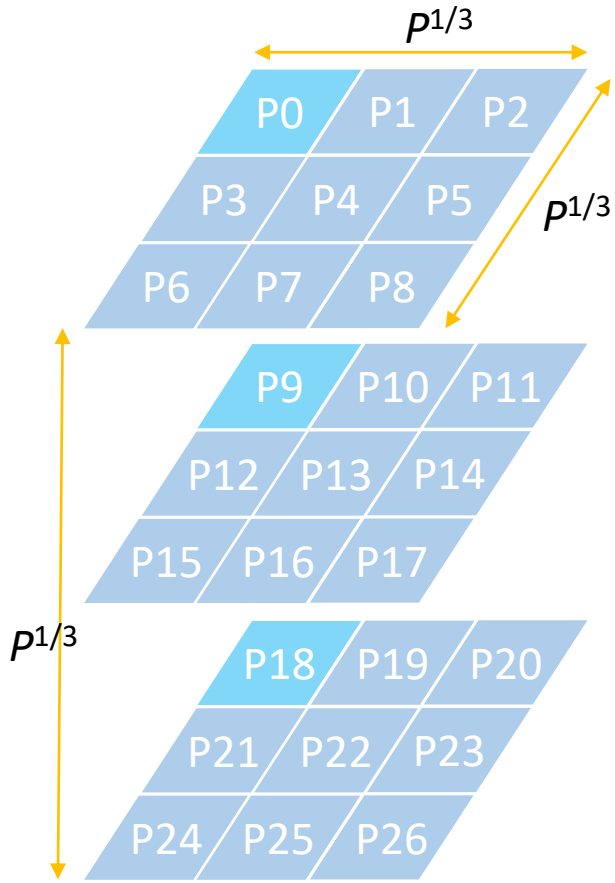
- Step 3.1: Compute `C[i,j]=C[i,j]+A[i,j]*B[i,j]` at each processor `(i,j)`

Cannon's Algorithm

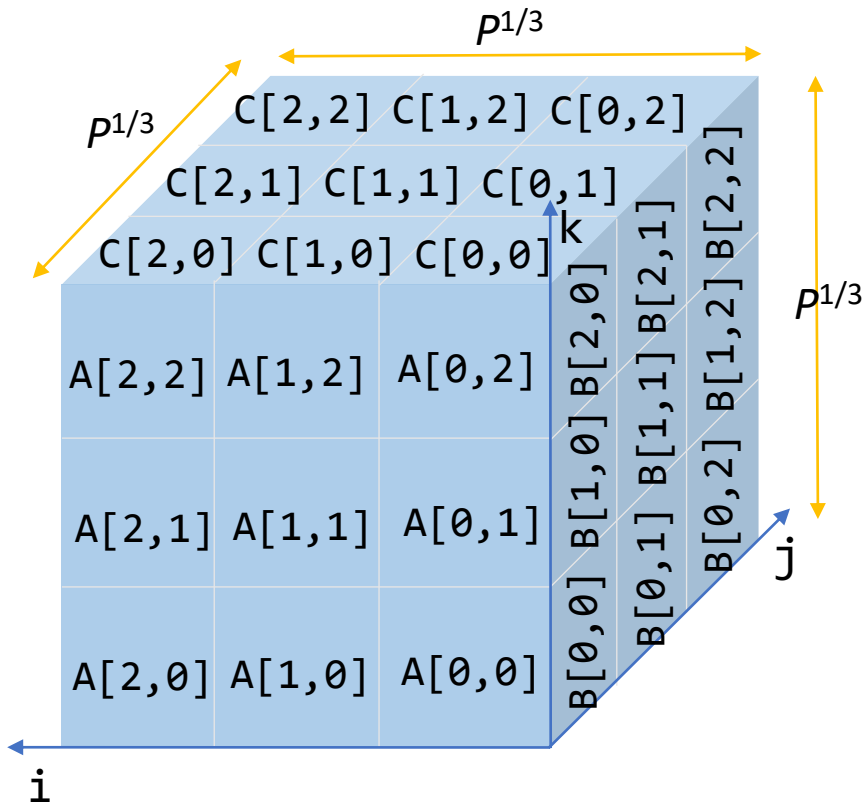
- We consider parallel execution time and communication per processor
  - Execution time = $\sqrt{P} \times$ Multiplication_time = $O(\sqrt{P}\,(n/\sqrt{P})^3) = O(n^3/P)$
  - #Words_moved = $O(\sqrt{P} \times |A[i,j]|) = O(\sqrt{P}\,(n/\sqrt{P})^2) = O(n^2/\sqrt{P})$
  - Good execution time, and efficient communication (lower bound = $\Omega(n^2/\sqrt{P})$)
- Pros 👍
  - Efficient communication
  - Close to theoretical lower bound
- Cons 👎
  - Difficult to handle with non-square matrices
  - Is it fast in practice?

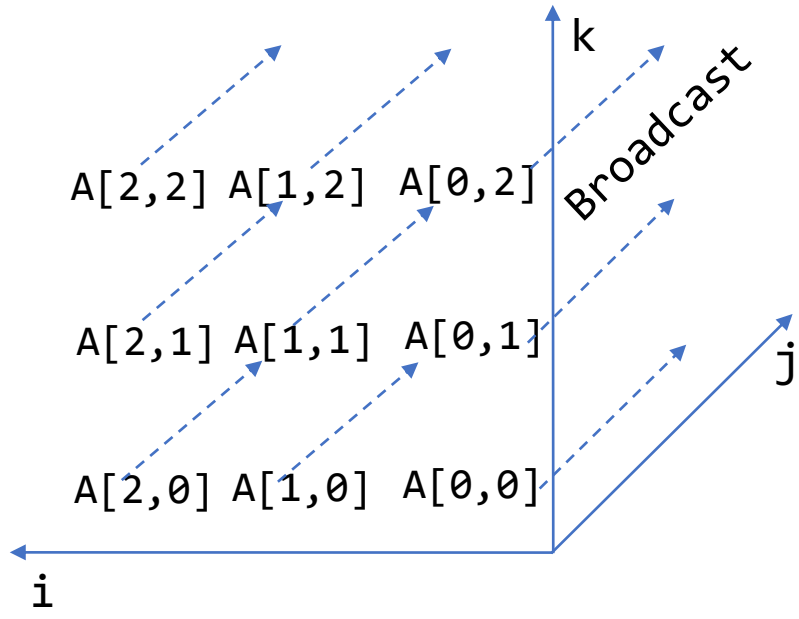
3D Data Layout

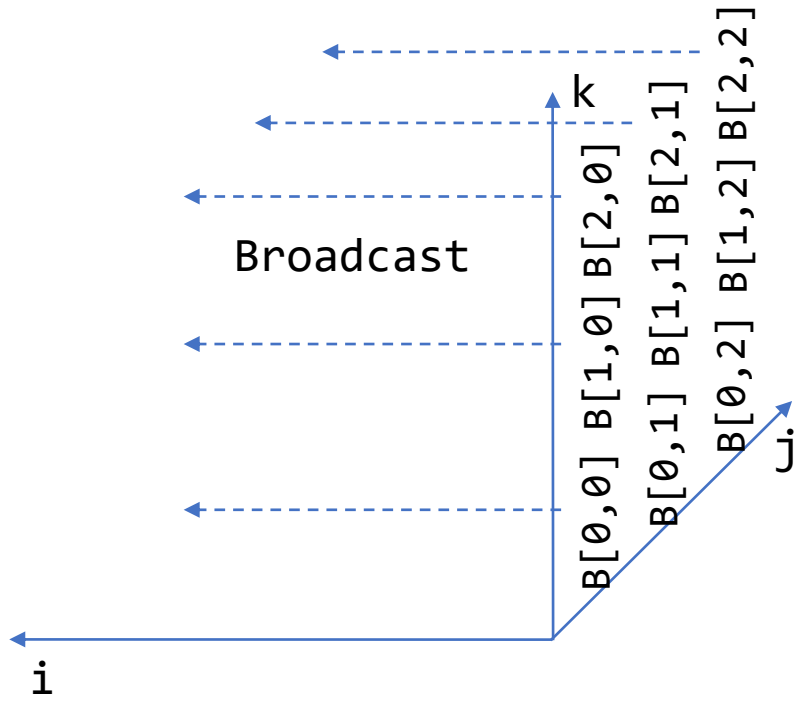
- #words_moved = $\Omega(n^3/(P\sqrt{M}))$
  - If $M = O(n^2/P)$, then words_moved = $O((n^2/P^{1/2}))$
  - Can we use more memory (larger M) to communicate less?
- 3D Matrix Multiply Algorithm on $P^{1/3}$ x $P^{1/3}$ x $P^{1/3}$ processor grid
  - Broadcast A in `j` direction ($P^{1/3}$ redundant copies)
  - Broadcast B in `i` direction ($P^{1/3}$ redundant copies)
  - Local multiplies
    - Processor (`i,j,k`) performs `C[i,j]=C[i,j]+A[i,k]*B[k,j]`
    - Each submatrix is $n/P^{1/3}$ x $n/P^{1/3}$, and $M = n^2/P^{2/3}$
    - Reduce (sum) in k direction: `C[i,j]`$+\sum_k$`A[i,k]B[k,j]`
- Communication volume = $O((n/P^{1/3})^2)$ - optimal
- Number of messages = O(log(P)) - optimal



3D Data Layout







Strassen's Matrix Multiplication

- The traditional algorithm has O(n^3) flops
  - Strassen discovered an algorithm with O($n^{2.81}$) flops
- Consider a 2x2 matrix multiplication, normally takes 8 multiplies, 4 adds
  - Strassen does it with 7 multiplies and 18 adds
  - Let $C = \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}$
  - Let $p_1 = (a_{12} - a_{22})(b_{21} + b_{22}), p_2 = (a_{11} + a_{22})(b_{11} + b_{22}), p_3 = (a_{11} - a_{21})(b_{11} + b_{12})$

    $p_4 = (a_{11} + a_{12})(b_{22}), p_5 = (a_{11})(b_{12} - b_{22}), p_6 = (a_{22})(b_{21} - b_{11}), p_7 = (a_{21} + a_{22})(b_{11})$
  - Then $c_{11} = p_1 + p_2 - p_4 + p_6, \quad c_{12} = p_4 + p_5$

    $c_{21} = p_6 + p_7, \quad c_{22} = p_2 - p_3 + p_5 - p_7$
- For $n \times n$ matrix multiplication, consider $\frac{n}{2} \times \frac{n}{2}$ sub-matrices

Strassen's Matrix Multiplication

- Let running time be $T(n)$
- By divide-and-conquer, and apply Strassen multiplication recursively
  - $T(n) = 7\ T(n/2) + 18\ (n/2)^2$

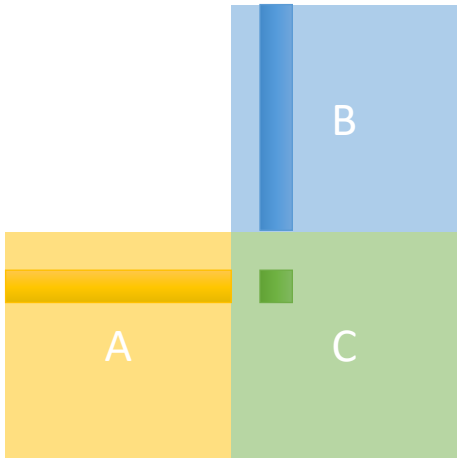
    $= O(n^{\log 7}) = O(n^{2.81})$ based on Master Theorem

- Possible to extend communication lower bound to Strassen
  - #words moved between fast and slow memory

    $= \Omega(n^{\log 7} / M^{(\log 7)/2 - 1}) \sim \Omega(n^{2.81} / M^{0.4})$
  - Attainable and parallelizable

Summary

- Matrix multiplication is a fundamental operation
  - Matrix-vector multiplication, matrix-matrix multiplication
- Serial matrix multiplication
  - Communication-avoiding, blocked matrix multiplication
- Parallel matrix multiplication
  - SUMMA, Cannon's algorithm, 3D matrix multiplication
- Lower bound on necessary communication
  - Achievable by serial/parallel matrix multiplication
- Strassen's algorithm
  - Faster recursive matrix multiplication



$$C = A \cdot B$$

References

- “Introduction to Parallel Computing ”, *Grama, Karypis, Kumar, Gupta,*
  - Chapter 8 (Dense Matrix Algorithms)
- “Communication lower bounds for distributed-memory matrix multiplication”, *Dror Irony, Sivan Toledo, Alexandre Tiskin*, Journal of Parallel & Distributed Computing, 2004