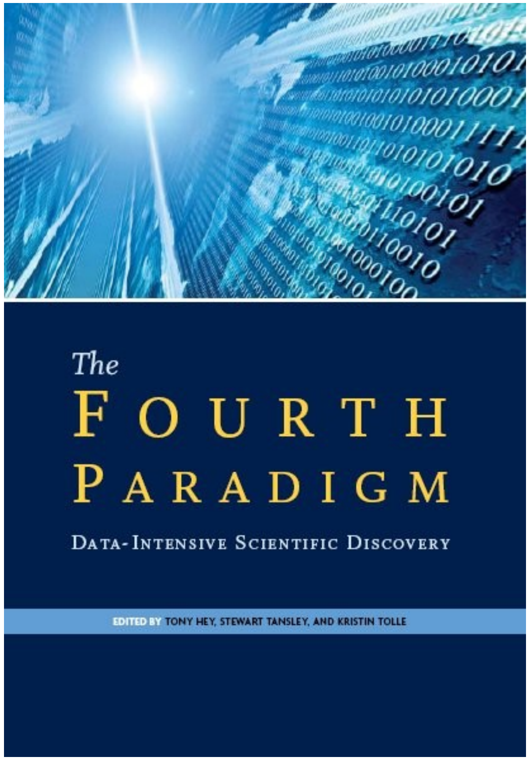
Matrix multiplication

Computational Science

Paradigms in science

1. Theory
2. Experiments
3. Large scale computer simulations
    - Construct mathematical models
    - Implement as computer programs
    - Simulate on computers
4. Massive data sets



All types of computational science sooner or later boil down to numerical computations

Elementary computations

BLAS: **B**asic **L**inear **A**lgebra **S**ubprograms	Data	Work
Level 1: $\mathbf{y} = \alpha\mathbf{x} + \mathbf{y}$ where **x** and **y** are n–dimensional vectors	$O(n)$	$O(n)$
Level 2: $\mathbf{y} = \alpha\mathbf{A}\mathbf{x} + \beta\mathbf{y}$ where **A** is an n x n matrix	$O(n^2)$	$O(n^2)$
Level 3: $\mathbf{C} = \alpha\mathbf{A}\mathbf{B} + \beta\mathbf{C}$ where **B** and **C** are n x n matrices	$O(n^2)$	$O(n^3)$

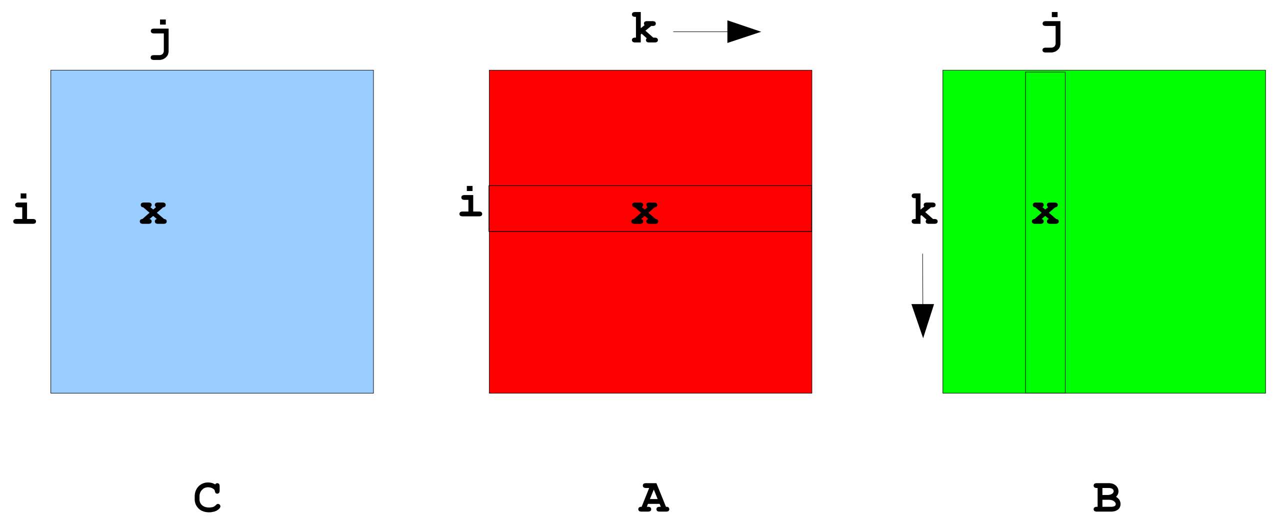
Building blocks when solving various problems in numerical linear algebra:

- Systems of linear equations
- Linear least squares
- Eigenvalue problems
- Singular value decompositions

Also similar computations for *sparse* data sets

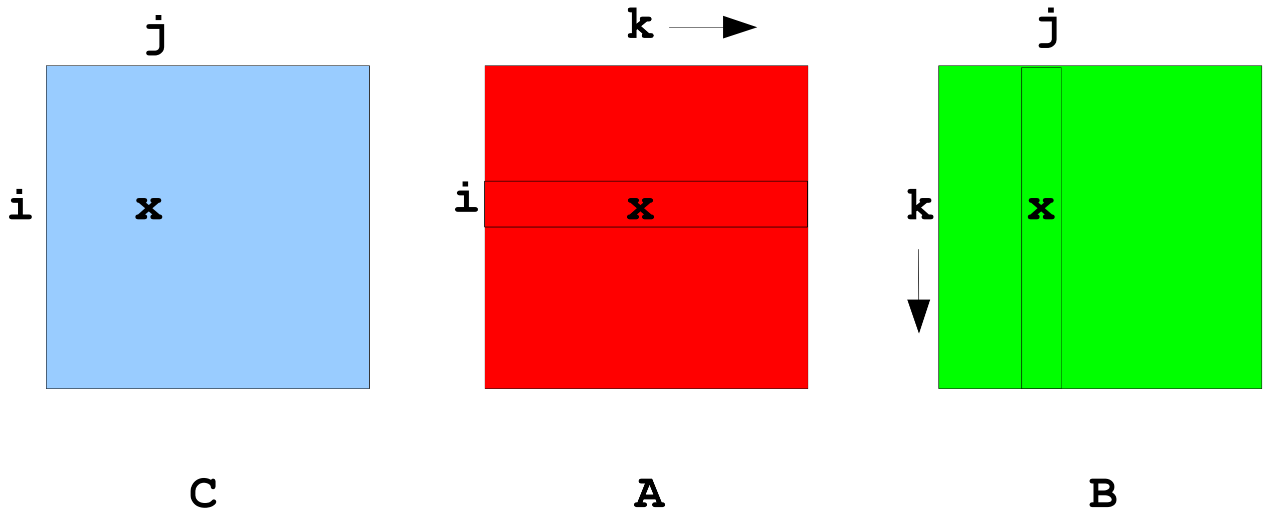
BLAS 3: Matrix Multiplication

Compute **`C = A*B`** where **`A`**, **`B`**, and **`C`** are **`nxn`** real valued matrices



BLAS 3: Matrix Multiplication

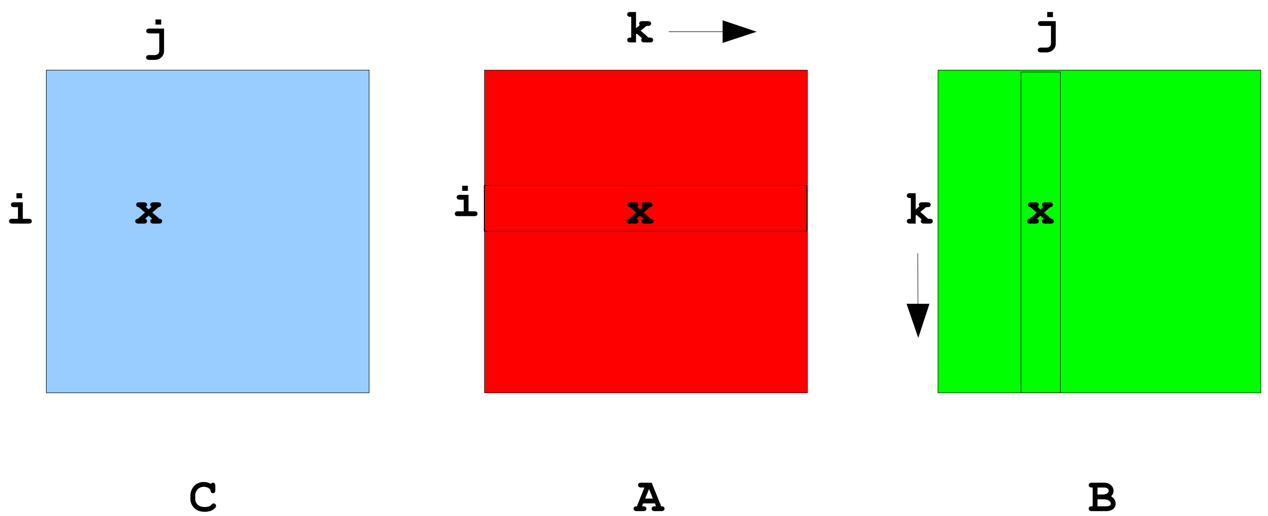
Compute `C = A*B` where `A`, `B`, and `C` are `nxn` real valued matrices




```
for i=0,...,n-1
  for j=0,...,n-1 {
    c[i][j] = 0.0
    for k=0,...,n-1
      c[i][j] += a[i][k] * b[k][j]
  }
```

BLAS 3: Matrix Multiplication

Compute **`C = A*B`** where **`A`**, **`B`**, and **`C`** are **`nxn`** real valued matrices




```
for i=0,...,n-1
  for j=0,...,n-1 {
    c[i][j] = 0.0
    for k=0,...,n-1
      c[i][j] += a[i][k] * b[k][j]
  }
```

Requires $3n^2$ data elements and $n^2(2n - 1)$ flops

Running time: $\Theta(n^3)$

The evolution of matrix multiplication algorithms

- Straightforward: $\mathbf{n^3}$
- Strassen (1968): $\mathbf{n^{lg(7)}} \approx \mathbf{n^{2.8074}}$
- Coppersmith–Winograd (1990): $\mathbf{n^{2.375477}}$
- Stothers (2010): $\mathbf{n^{2.3736897}}$
  - Williams (2011): $\mathbf{n^{2.3728642}}$
  - Le Gall (2014): $\mathbf{n^{2.3728639}}$

The evolution of matrix multiplication algorithms

- Straightforward: n^3
- Strassen (1968): $n^{\lg(7)} \approx n^{2.8074}$
- Coppersmith–Winograd (1990): $n^{2.375477}$
- Stothers (2010): $n^{2.3736897}$
  - Williams (2011): $n^{2.3728642}$
  - Le Gall (2014): $n^{2.3728639}$

In practice:

- Algorithms with lower complexity than Strassen are slow
- Sub-cubic algorithms require much memory

Strassen's algorithm

$$\begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} \times \begin{bmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{bmatrix} = \begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix}$$

Algorithm Strassen(**A**,**B**,**n**) {

    if (**n==1**) return **A*B**
    P_1 = Strassen $(A_{00}+A_{11}, B_{00}+B_{11}, n/2)$
    P_2 = Strassen $(A_{10}+A_{11}, B_{00}, n/2)$
    P_3 = Strassen $(A_{00}, B_{01}-B_{11}, n/2)$
    P_4 = Strassen $(A_{11}, B_{10}-B_{00}, n/2)$
    P_5 = Strassen $(A_{00}+A_{01}, B_{11}, n/2)$
    P_6 = Strassen $(A_{10}-A_{00}, B_{00}+B_{01}, n/2)$
    P_7 = Strassen $(A_{01}-A_{11}, B_{10}+B_{11}, n/2)$
    $C_{00} = P_1+P_4-P_5+P_7$
    $C_{01} = P_3+P_5$
    $C_{10} = P_2+P_4$
    $C_{11} = P_1-P_2+P_3+P_6$
    return **C**

}

Strassen's algorithm

$$\begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} \times \begin{bmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{bmatrix} = \begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix}$$

Algorithm Strassen(**A,B,n**) {

  if (**n==1**) return **A*B**

  P_1 = Strassen($A_{00}+A_{11}, B_{00}+B_{11}, n/2$)

  P_2 = Strassen($A_{10}+A_{11}, B_{00}, n/2$)

  P_3 = Strassen($A_{00}, B_{01}-B_{11}, n/2$)

  P_4 = Strassen($A_{11}, B_{10}-B_{00}, n/2$)

  P_5 = Strassen($A_{00}+A_{01}, B_{11}, n/2$)

  P_6 = Strassen($A_{10}-A_{00}, B_{00}+B_{01}, n/2$)

  P_7 = Strassen($A_{01}-A_{11}, B_{10}+B_{11}, n/2$)

  $C_{00} = P_1+P_4-P_5+P_7$

  $C_{01} = P_3+P_5$

  $C_{10} = P_2+P_4$

  $C_{11} = P_1-P_2+P_3+P_6$

  return **C**

}

Running time: $f(n)$ =

  number of additions and multiplications

Recursion: $f(n) = 7f(n/2) + k*n^2$

Yields: $f \in \Theta(n^{\lg 7})$

Straightforward algorithm in OpenMP

```
for i=0,...,n-1
  for j=0,...,n-1 {
    c[i][j] = 0.0
    for k=0,...,n-1
      c[i][j] += a[i][k] * b[k][j]
  }
```

Straightforward algorithm in OpenMP

```
#pragma omp parallel for private(j,k)
for i=0,...,n-1
  for j=0,...,n-1 {
    c[i][j] = 0.0
    for k=0,...,n-1
      c[i][j] += a[i][k] * b[k][j]
  }
```

Straightforward algorithm in OpenMP

```
#pragma omp parallel for private(j,k)
for i=0,...,n-1
  for j=0,...,n-1 {
    c[i][j] = 0.0
    for k=0,...,n-1
      c[i][j] += a[i][k] * b[k][j]
  }
```

Speedup:

- Wrt sequential straightforward: $n^3/(n^3/p) = p$ (perfect speedup!)

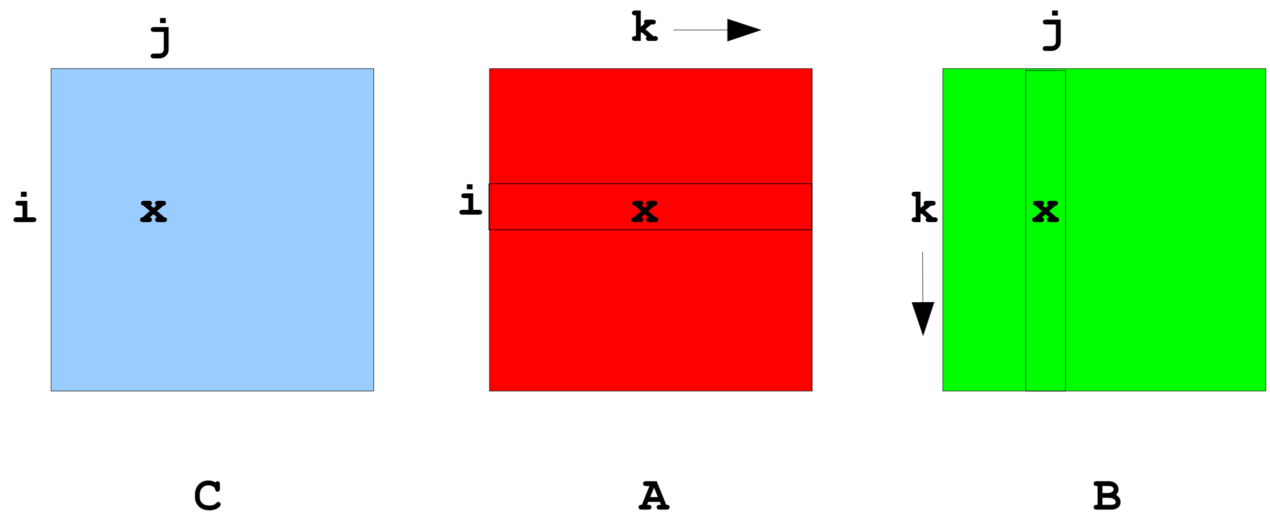
Straightforward algorithm in OpenMP

```
#pragma omp parallel for private(j,k)
for i=0,...,n-1
  for j=0,...,n-1 {
    c[i][j] = 0.0
    for k=0,...,n-1
      c[i][j] += a[i][k] * b[k][j]
  }
```

Speedup:

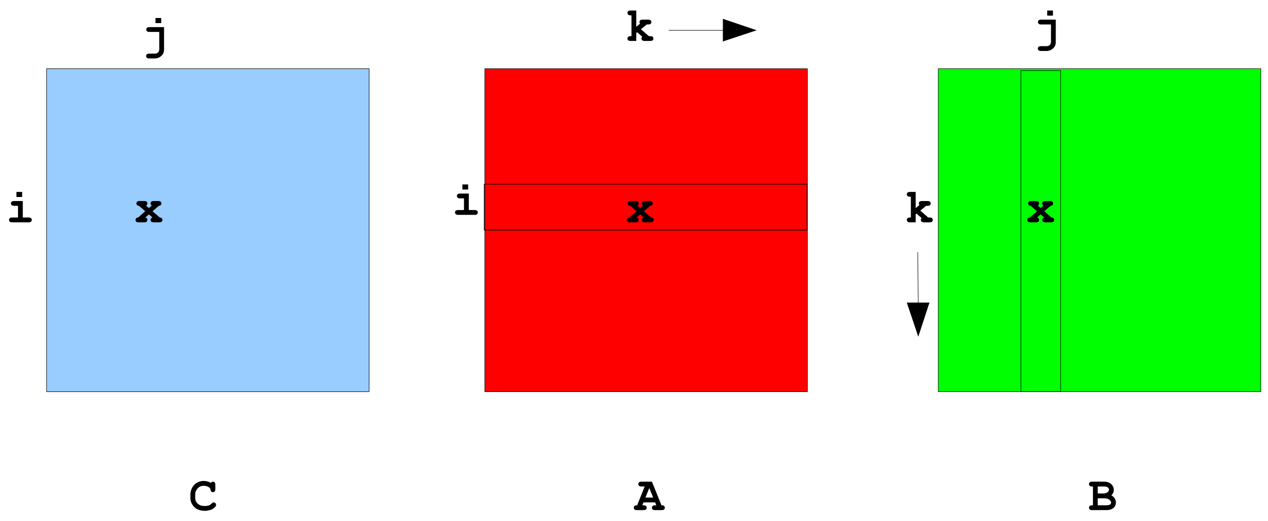
- Wrt sequential straightforward: $n^3/(n^3/p) = p$ (perfect speedup!)
- Wrt sequential Strassen: $n^{\lg 7}/(n^3/p) = pn^{\lg 7-3} \rightarrow 0$ as $n \rightarrow \infty$

Matrix multiplication in OpenMP




```
#pragma omp parallel for private(j,k)
for i=0,...,n-1
  for j=0,...,n-1 {
    c[i][j] = 0.0
    for k=0,...,n-1
      c[i][j] += a[i][k] * b[k][j]
  }
```

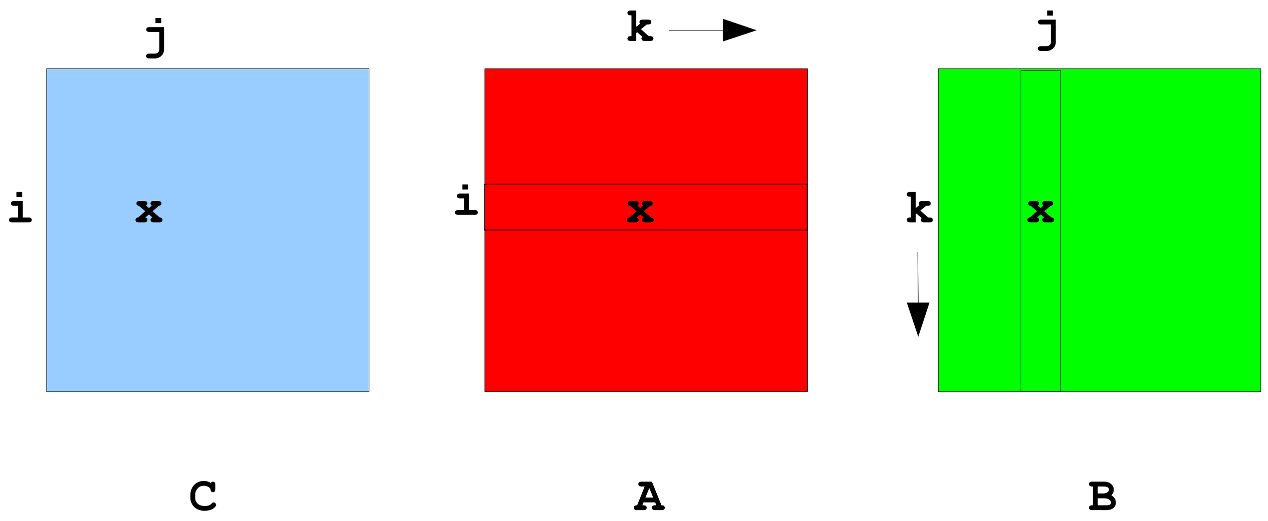
Matrix multiplication in OpenMP




```
#pragma omp parallel for private(j,k)
for i=0,...,n-1
  for j=0,...,n-1 {
    c[i][j] = 0.0
    for k=0,...,n-1
      c[i][j] += a[i][k] * b[k][j]
  }
```

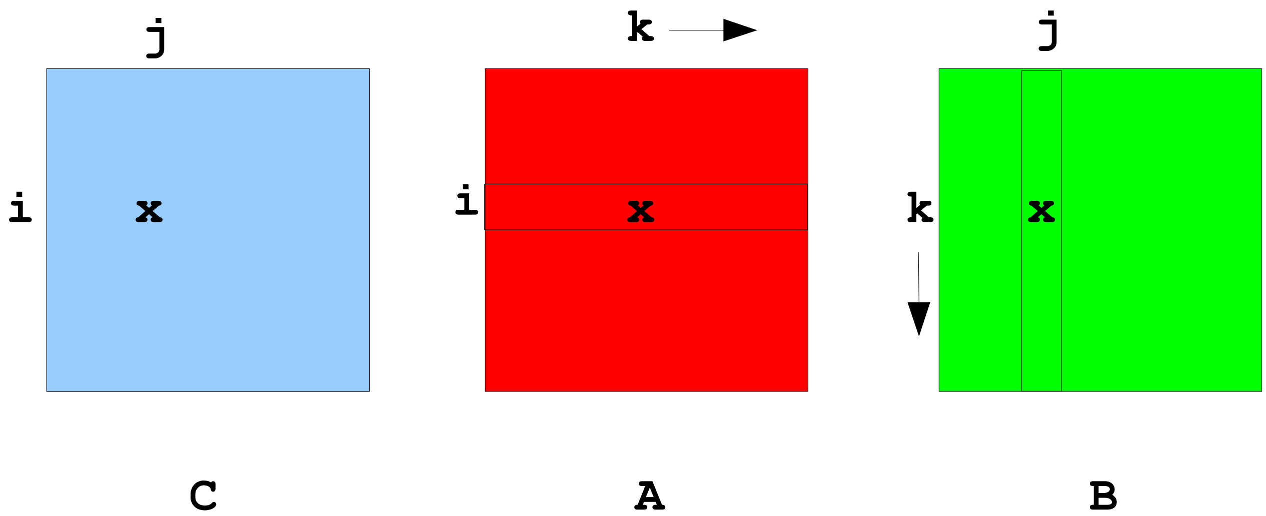
Rows of B move in and out of cache!

Matrix multiplication in OpenMP




```
#pragma omp parallel for private(j,k)
for i=0,...,n-1 {
  for j=0,...,n-1
    c[i][j] = 0.0
  for k=0,...,n-1
    for j=0,...,n-1
      c[i][j] += a[i][k] * b[k][j]
}
```

Matrix multiplication in OpenMP




```
#pragma omp parallel for private(j,k)
for i=0,...,n-1 {
  for j=0,...,n-1
    c[i][j] = 0.0
  for k=0,...,n-1
    for j=0,...,n-1
      c[i][j] += a[i][k] * b[k][j]
}
```

Finish all work with one row in B!

How fast can two matrices be multiplied?

- Straightforward version with **n** threads: $\Theta(n^2)$

```
for i=0,...,n-1
  for j=0,...,n-1 {
    c[i][j] = 0.0
    for k=0,...,n-1
      c[i][j] += a[i][k] * b[k][j]
  }
```

How fast can two matrices be multiplied?

- Straightforward version with **`n`** threads: $\Theta(n^2)$
- Straightforward version with n^2 threads:
  - Thread **`(i,j)`** computes **`c[i][j]`**:
  - Run through the **`i`**th row of **`A`** and the **`j`**th column of **`B`**: $\Theta(n)$

```
for i=0,...,n-1
  for j=0,...,n-1 {
    c[i][j] = 0.0
    for k=0,...,n-1
      c[i][j] += a[i][k] * b[k][j]
  }
```

How fast can two matrices be multiplied?

- Straightforward version with `n` threads: $\Theta(n^2)$
- Straightforward version with n^2 threads:
  - Thread `(i,j)` computes `c[i][j]`:
  - Run through the `i`th row of `A` and the `j`th column of `B`: $\Theta(n)$
- Straightforward version with n^3 threads:
  - Thread `(i,j,k)` computes `product = a[i][k] * b[k][j]`
  - n^2 reductions of `product` in parallel : `(i,j,k)`→`(i,j,0)`
  - $\Theta(\lg(n))$

```
for i=0,...,n-1
  for j=0,...,n-1 {
    c[i][j] = 0.0
    for k=0,...,n-1
      c[i][j] += a[i][k] * b[k][j]
  }
```

How fast can two matrices be multiplied?

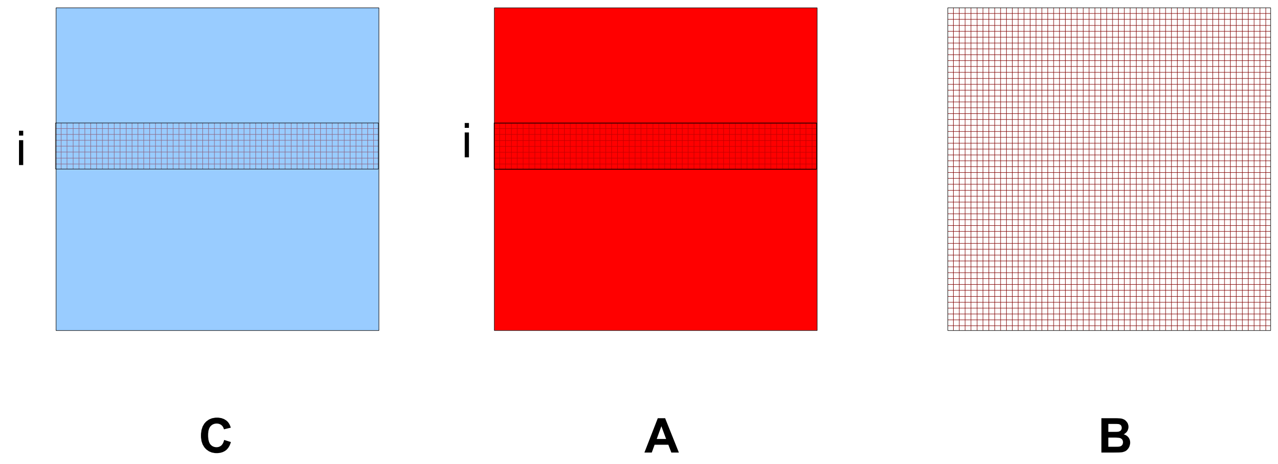
- Straightforward version with `n` threads: `Θ(n`^{2}`)`
- Straightforward version with `n`^{2} threads:
  - Thread `(i,j)` computes `c[i][j]`:
  - Run through the `i`th row of `A` and the `j`th column of `B`: `Θ(n)`
- Straightforward version with `n`^{3} threads:
  - Thread `(i,j,k)` computes `product = a[i][k] * b[k][j]`
  - `n`^{2} reductions of `product` in parallel : `(i,j,k)`→`(i,j,0)`
  - `Θ(lg(n))`
- Theorem (Moldovan, 1993): Cannot multiply faster than `Θ(lg(n))`

```
for i=0,...,n-1
  for j=0,...,n-1 {
    c[i][j] = 0.0
    for k=0,...,n-1
      c[i][j] += a[i][k] * b[k][j]
  }
```

Matrix multiplication in MPI (or BSP)

Row-wise decomposition (assume `p=n`):

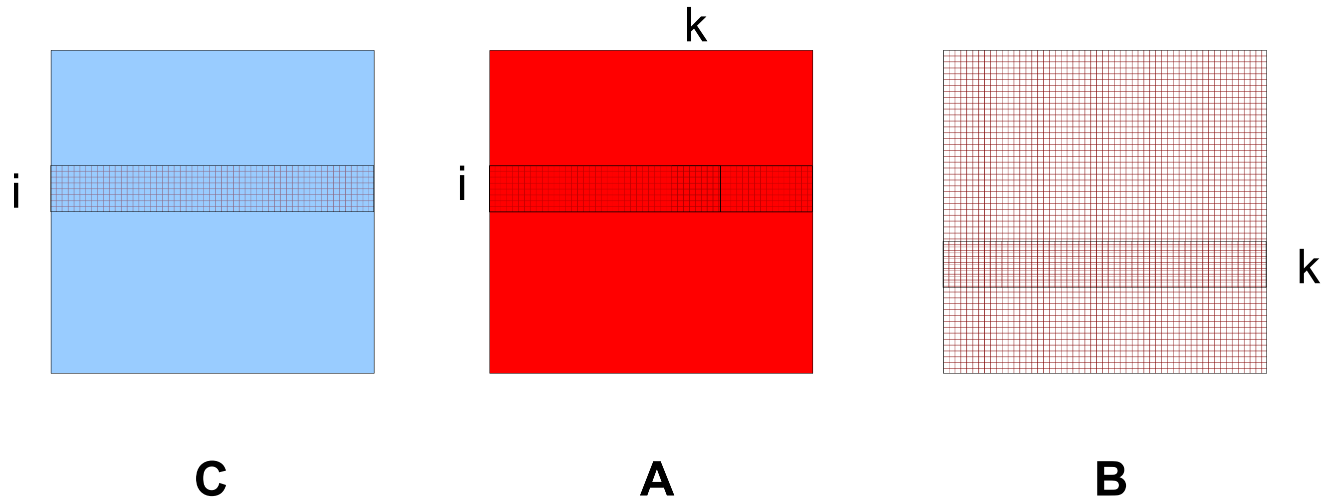
- Process `i` holds row `i` of matrices `A` and `B`
- Process `i` computes the `i`th row of `C`
  - Needs access to row `i` of `A`
  - Needs access to all of `B`
- Alternative 1: Broadcast `B` to all processes



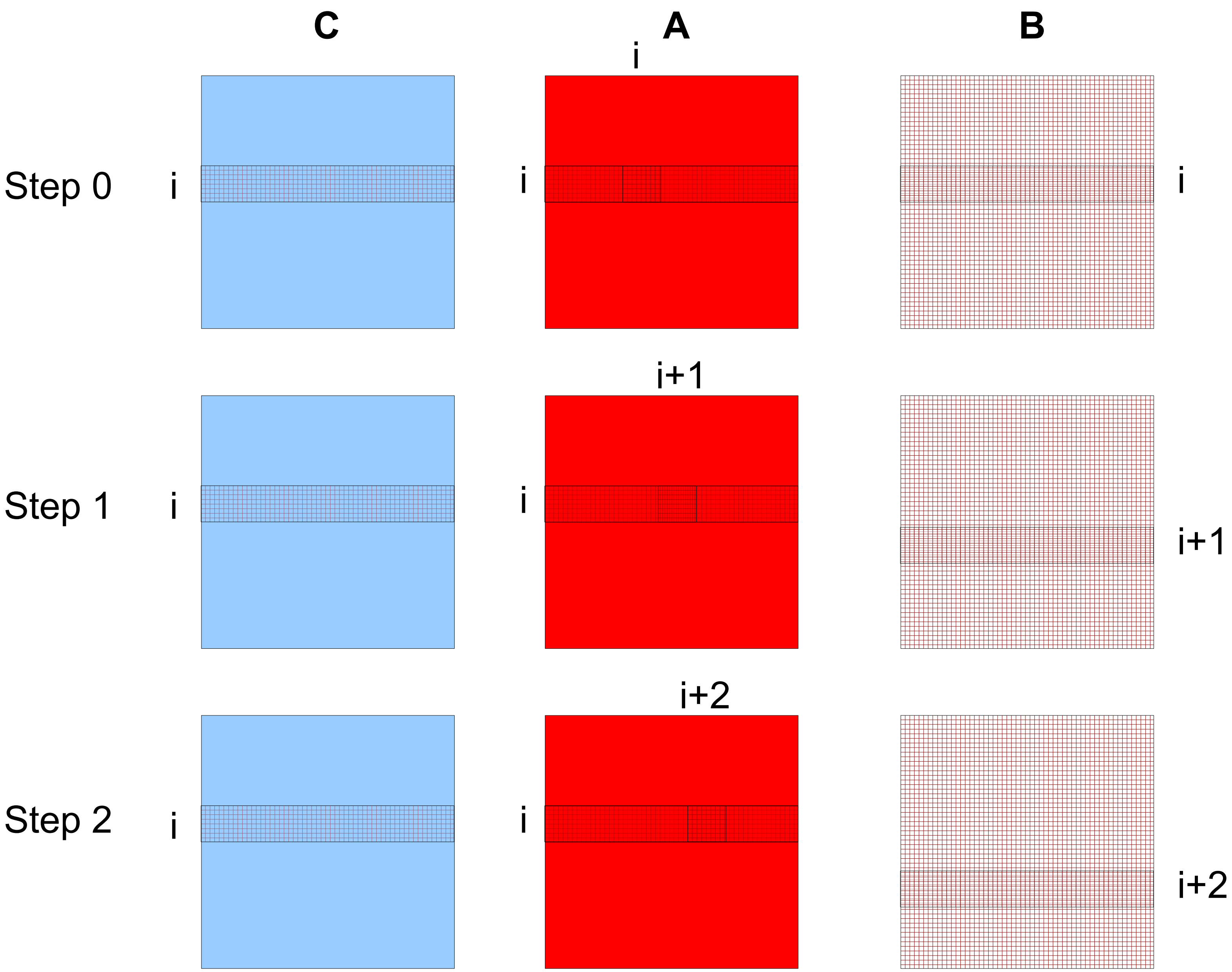
Matrix multiplication in MPI (or BSP)

Row-wise decomposition (assume `p=n`):

- Process `i` holds row `i` of matrices `A` and `B`
- Process `i` computes the `i`th row of `C`
  - Needs access to row `i` of `A`
  - Needs access to all of `B`
- Alternative 1: Broadcast `B` to all processes
- Alternative 2: Rotate the rows of `B` on the processes
  - Reduces memory usage in each process



Matrix multiplication in MPI (or BSP)



Matrix multiplication in MPI (or BSP)

Idea: Row `i` of `C = SUM(a[k] * row k of B)` over $0 \le k < n$

```
Algorithm MM(A,B,C) {              // Process i initially holds row i of A and B
  c[0..n-1] = 0.0; i = process id; k = i
  repeat
    for j=0,...,n-1
      c[j] += a[k]*b[j]
    k = (k+1)%n
    if (k!=i) {
      send b to process (i-1)%n
      receive b from process (i+1)%n
    }
  until k==i
}
```

Matrix multiplication in MPI (or BSP)

When **`p<n`**:

- Process **`i`** holds **`m=n/p`** rows of **`A`**
- Process **`i`** computes **`m`** rows of **`C`**

Matrix multiplication in MPI (or BSP)

When $p<n$:

- Process i holds $m=n/p$ rows of A
- Process i computes m rows of C

Running time analysis:

- Computation:
  - each process computes a submatrix with m rows and n columns
  - n^2/p elements in total
  - each element takes $\Theta(n)$ time to compute
  - $T_{comp} \in \Theta(n^3/p)$

Matrix multiplication in MPI (or BSP)

When $p<n$:

- Process i holds $m=n/p$ rows of A
- Process i computes m rows of C

Running time analysis:

- Computation:
  - each process computes submatrix with m rows and n columns
  - n^2/p elements in total
  - each element takes $\Theta(n)$ time to compute
  - $T_{comp} \in \Theta(n^3/p)$
- Communication:
  - p point-to-point communications of length n^2/p: $pt_0+pt_1n^2/p = pt_0+t_1n^2$
  - $T_{comm} \in \Theta(p+n^2)$

Matrix multiplication in MPI (or BSP)

When $p<n$:

- Process i holds $m=n/p$ rows of A
- Process i computes m rows of C

Running time analysis:

- Computation:
  - each process computes submatrix with m rows and n columns
  - n^2/p elements in total
  - each element takes $\Theta(n)$ time to compute
  - $T_{comp} \in \Theta(n^3/p)$
- Communication:
  - p point-to-point communications of length n^2/p: $pt_0+pt_1n^2/p = pt_0+t_1n^2$
  - $T_{comm} \in \Theta(p+n^2)$
- $T_{comp}/T_{comm} \in \Theta(n/p)$

Matrix multiplication in MPI (or BSP)

When $p<n$:

- Process i holds $m=n/p$ rows of A
- Process i computes m rows of C

Running time analysis:

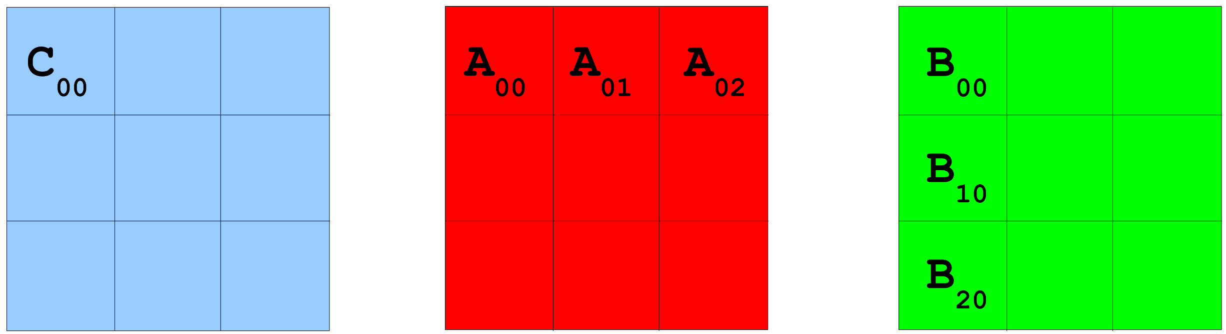
- Computation:
  - each process computes submatrix with m rows and n columns
  - n^2/p elements in total
  - each element takes $\Theta(n)$ time to compute
  - $T_{comp} \in \Theta(n^3/p)$
- Communication:
  - p point-to-point communications of length n^2/p: $pt_0+pt_1n^2/p = pt_0+t_1n^2$
  - $T_{comm} \in \Theta(p+n^2)$ (BSP: $l=t_0$ $g=t_1$)
- $T_{comp}/T_{comm} \in \Theta(n/p)$

Observation:

- Process i needs
  - n^2/p elements from A
  - all n^2 elements from B

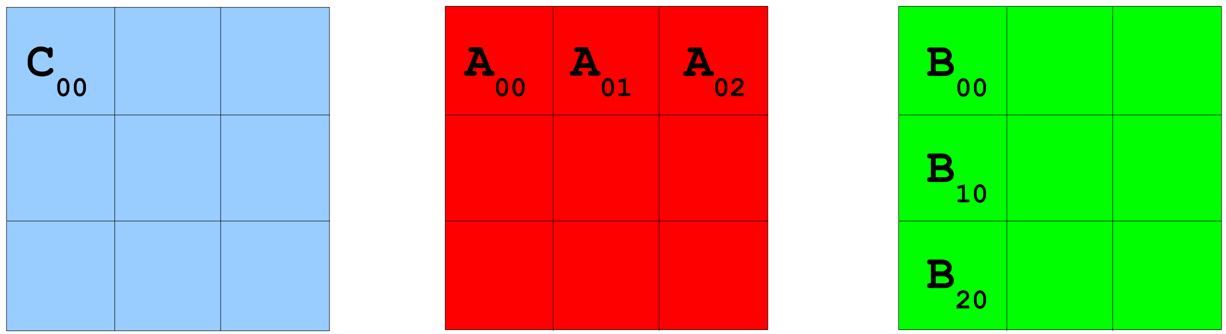
  to compute only n^2/p elements in C
- Can we compute n^2/p elements in C with access to fewer elements from B?
  - $\Rightarrow$ reduced communication

Block-wise decomposition



$$C_{00} = (A_{00} * B_{00}) + (A_{01} * B_{10}) + (A_{02} * B_{20})$$

Block-wise decomposition

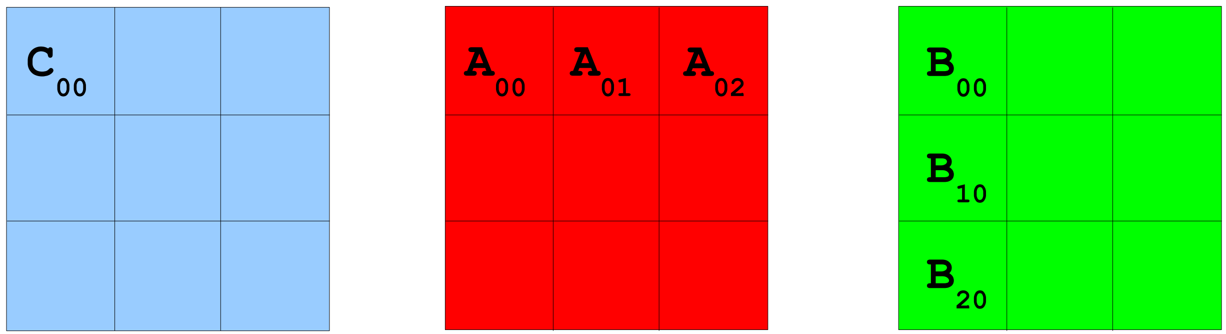


$C_{00} = (A_{00} * B_{00}) + (A_{01} * B_{10}) + (A_{02} * B_{20})$

s = number of row and column blocks

```
for i = 0,...,s-1
   for j = 0,...,s-1
      Cij = 0.0
      for k = 0,...,s-1
         Cij += Aik * Bkj // Matrix multiplication
```

Block-wise decomposition

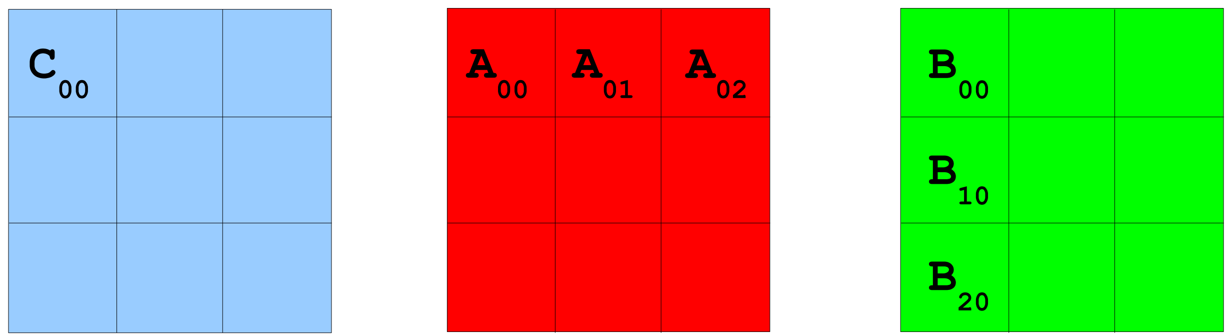


$$C_{00} = (A_{00} * B_{00}) + (A_{01} * B_{10}) + (A_{02} * B_{20})$$

```
s = number of row and column blocks
for i = 0,...,s-1
   for j = 0,...,s-1
      Cij = 0.0
      for k = 0,...,s-1
         Cij += Aik * Bkj // Matrix multiplication
```

- assume $p=s^2$ is a square number
- process (i,j) to compute C_{ij}
- $m = n/s =$ rows and columns in a block
- each block multiplication requires $\sim 2m^3$ flops.

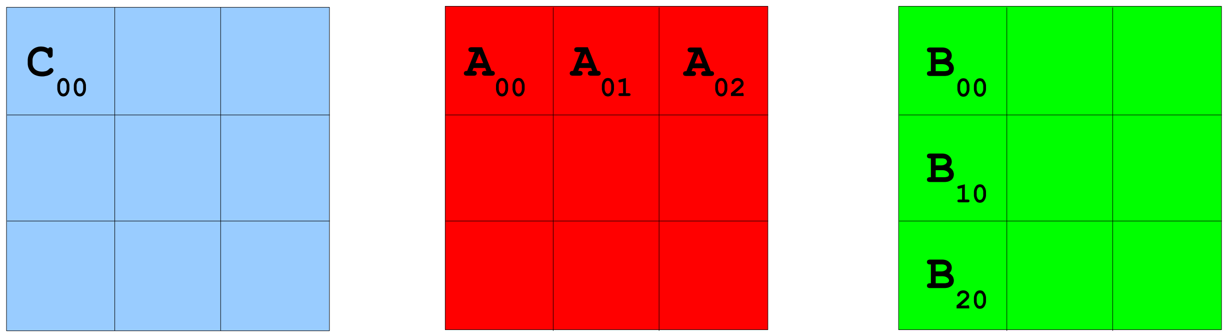
Block-wise decomposition



$$C_{00} = (A_{00} * B_{00}) + (A_{01} * B_{10}) + (A_{02} * B_{20})$$

- Idea (Canon's algorithm): Rotate **A**- and **B**-blocks between the processes

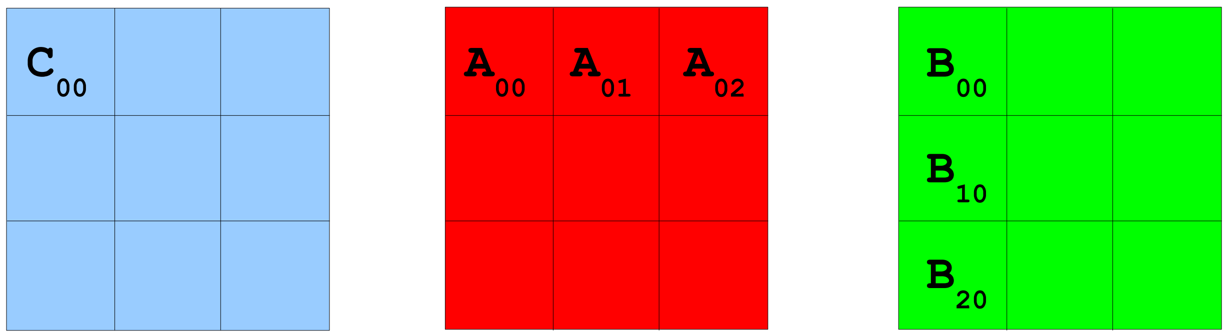
Block-wise decomposition



$$C_{00} = (A_{00} * B_{00}) + (A_{01} * B_{10}) + (A_{02} * B_{20})$$

- Idea (Canon's algorithm): Rotate **A**- and **B**-blocks between the processes
- To what **A**- and **B**-blocks does process **(i,j)** need access?
  - **A**-blocks with row index **i**: $A_{i0}, A_{i1}, \ldots, A_{i,s-1}$
  - **B**-blocks with column index **j**: $B_{0j}, B_{1j}, \ldots, B_{s-1,j}$
  - Synchronization: Get A_{ik} and B_{kj} simultaneously

Block-wise decomposition

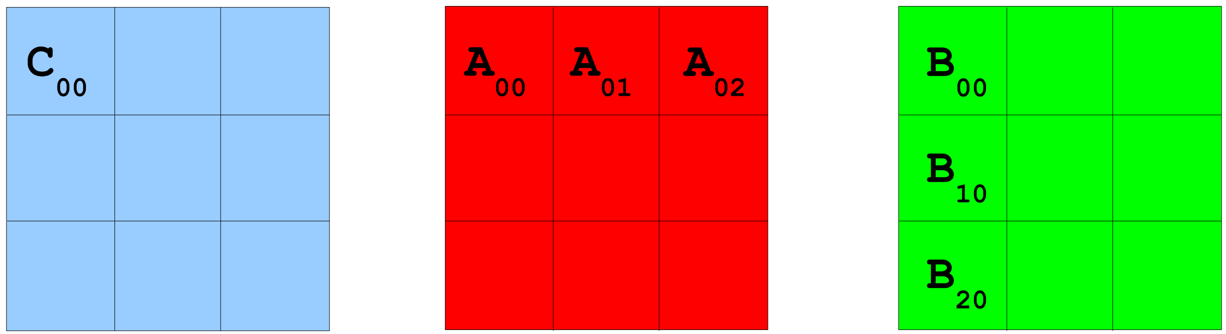


$$C_{00} = (A_{00} * B_{00}) + (A_{01} * B_{10}) + (A_{02} * B_{20})$$

- Idea (Canon's algorithm): Rotate **A**- and **B**-blocks between the processes
- To what **A**- and **B**-blocks does process **(i,j)** need access?
  - **A**-blocks with row index **i**: $A_{i0}, A_{i1}, \ldots, A_{i,s-1}$
  - **B**-blocks with column index **j**: $B_{0j}, B_{1j}, \ldots, B_{s-1,j}$
  - Synchronization: Get A_{ik} and B_{kj} simultaneously

- What processes need access to A_{ij}?
  - C_{i*}: All processes with row id **i**: **(i,0), (i,1),...,(i,s-1)**
- What processes need access to B_{ij}?
  - C_{*j}: All processes with column id **j**: **(0,j), (1,j),...,(s-1,j)**

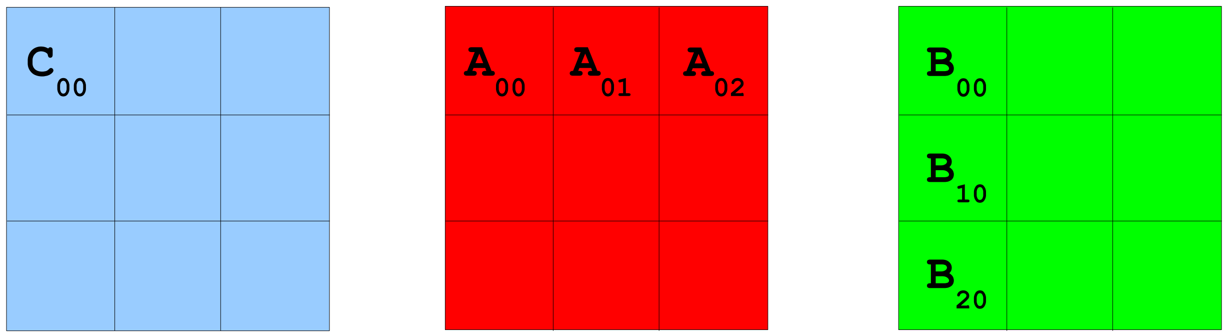
Block-wise decomposition



$$C_{00} = (A_{00} * B_{00}) + (A_{01} * B_{10}) + (A_{02} * B_{20})$$

- Idea (Canon's algorithm): Rotate **A**- and **B**-blocks on the processes
- Order is irrelevant for summation $C_{ij} = \Sigma_k(A_{ik} * B_{kj})$

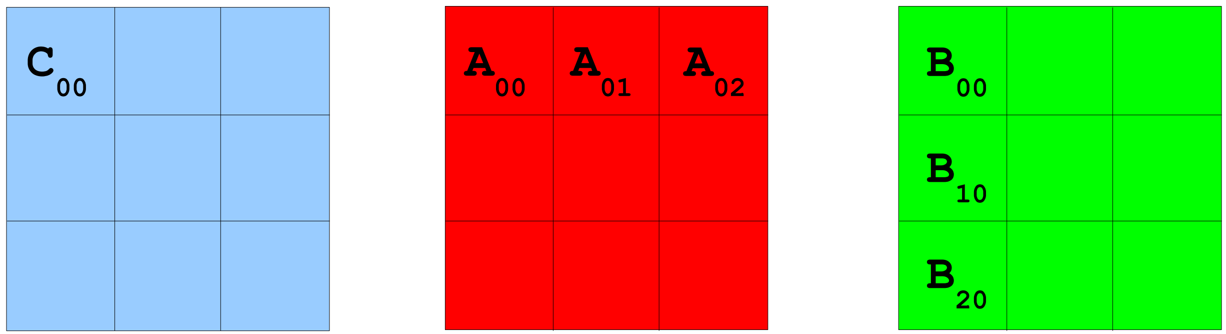
Block-wise decomposition



$$C_{00} = (A_{00} * B_{00}) + (A_{01} * B_{10}) + (A_{02} * B_{20})$$

- Idea (Canon's algorithm): Rotate **A**- and **B**-blocks on the processes
- Order is irrelevant for summation $C_{ij} = \Sigma_k (A_{ik} * B_{kj})$
- Start at **k=0**: $C_{ij} = A_{i0} * B_{0j} + A_{i1} * B_{1j} + ... + A_{i,s-1} * B_{s-1,j}$
  - Bad! All procs **(i,0),(i,1),(i,2),...,(i,s-1)** need A_{i0} early!

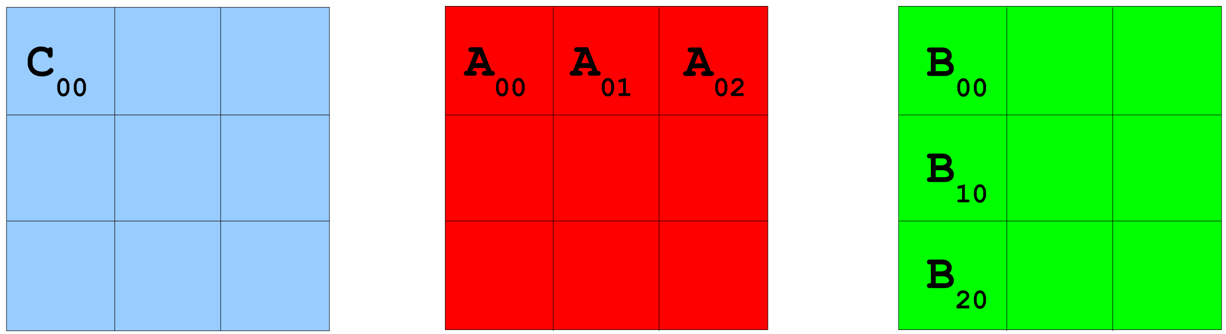
Block-wise decomposition



$$C_{00} = (A_{00} * B_{00}) + (A_{01} * B_{10}) + (A_{02} * B_{20})$$

- Idea (Canon's algorithm): Rotate **A**- and **B**-blocks on the processes
- Order is irrelevant for summation $C_{ij} = \Sigma_k (A_{ik} * B_{kj})$
- Start at **k=0**: $C_{ij} = A_{i0}*B_{0j} + A_{i1}*B_{1j}+...+A_{i,s-1}*B_{s-1,j}$
  - Bad! All procs **(i,0),(i,1),(i,2),...,(i,s-1)** need A_{i0} early!
- Start at **k=i**: $C_{ij} = A_{ii}*B_{ij} + A_{i,i+1}*B_{i+1,j}+...+A_{i,s-1}*B_{s-1,j}$
  $+ A_{i0}*B_{0j} + A_{i1}*B_{1j}+...+A_{i,i-1}*B_{i-1,j}$
  - Bad! All procs **(i,0),(i,1),(i,2),...,(i,s-1)** need A_{ii} early!

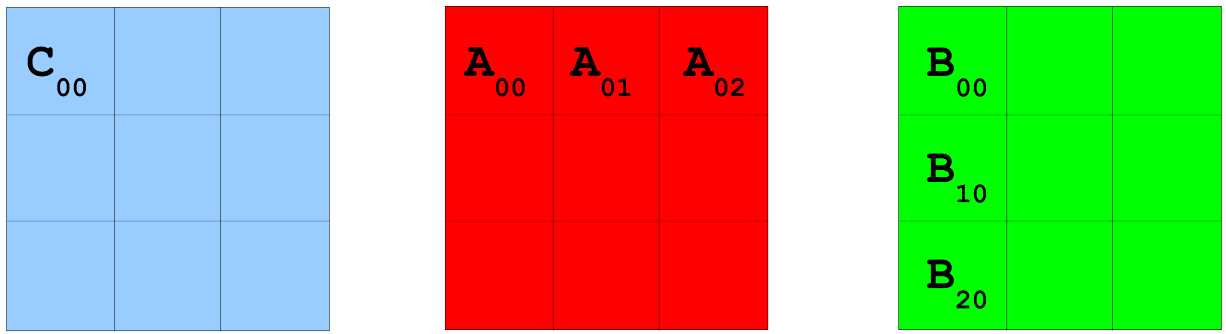
Block-wise decomposition



$$C_{00} = (A_{00} * B_{00}) + (A_{01} * B_{10}) + (A_{02} * B_{20})$$

- Idea (Canon's algorithm): Rotate **A**- and **B**-blocks on the processes
- Order is irrelevant for summation $C_{ij} = \Sigma_k (A_{ik} * B_{kj})$
- Start at **k=0**: $C_{ij} = A_{i0} * B_{0j} + A_{i1} * B_{1j} + \ldots + A_{i,s-1} * B_{s-1,j}$
  - Bad! All procs **(i,0),(i,1),(i,2),...,(i,s-1)** need A_{i0} early!
- Start at **k=i**: $C_{ij} = A_{ii} * B_{ij} + A_{i,i+1} * B_{i+1,j} + \ldots + A_{i,s-1} * B_{s-1,j}$
  $+ A_{i0} * B_{0j} + A_{i1} * B_{1j} + \ldots + A_{i,i-1} * B_{i-1,j}$
  - Bad! All procs **(i,0),(i,1),(i,2),...,(i,s-1)** need A_{ii} early!
- Summation order must depend on both **i** and **j**

Block-wise decomposition



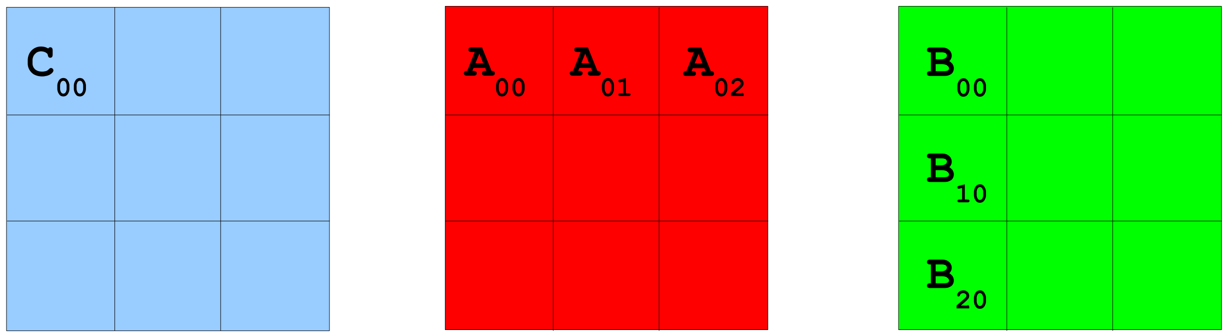
$$C_{00} = (A_{00} * B_{00}) + (A_{01} * B_{10}) + (A_{02} * B_{20})$$

- Idea (Canon's algorithm): Rotate **A**- and **B**-blocks on the processes
- Order is irrelevant for summation $C_{ij} = \Sigma_k (A_{ik} * B_{kj})$

First summation index value

k=0	k=1	k=2
k=1	k=2	k=0
k=2	k=0	k=1

Block-wise decomposition



$$C_{00} = (A_{00} * B_{00}) + (A_{01} * B_{10}) + (A_{02} * B_{20})$$

- Idea (Canon's algorithm): Rotate **A**- and **B**-blocks on the processes
- Order is irrelevant for summation $C_{ij} = \Sigma_k(A_{ik} * B_{kj})$

First summation index value

k=0	k=1	k=2
k=1	k=2	k=0
k=2	k=0	k=1

Start at **k=(i+j)%s**

Block-wise decomposition

C_{00}		

A_{00}	A_{01}	A_{02}

B_{00}		
B_{10}		
B_{20}		

$$C_{00} = (A_{00} * B_{00}) + (A_{01} * B_{10}) + (A_{02} * B_{20})$$

- Idea (Canon's algorithm): Rotate **A**- and **B**-blocks on the processes
- Order is irrelevant for summation $C_{ij} = \Sigma_k(A_{ik} * B_{kj})$

First summation index value

k=0	k=1	k=2
k=1	k=2	k=0
k=2	k=0	k=1

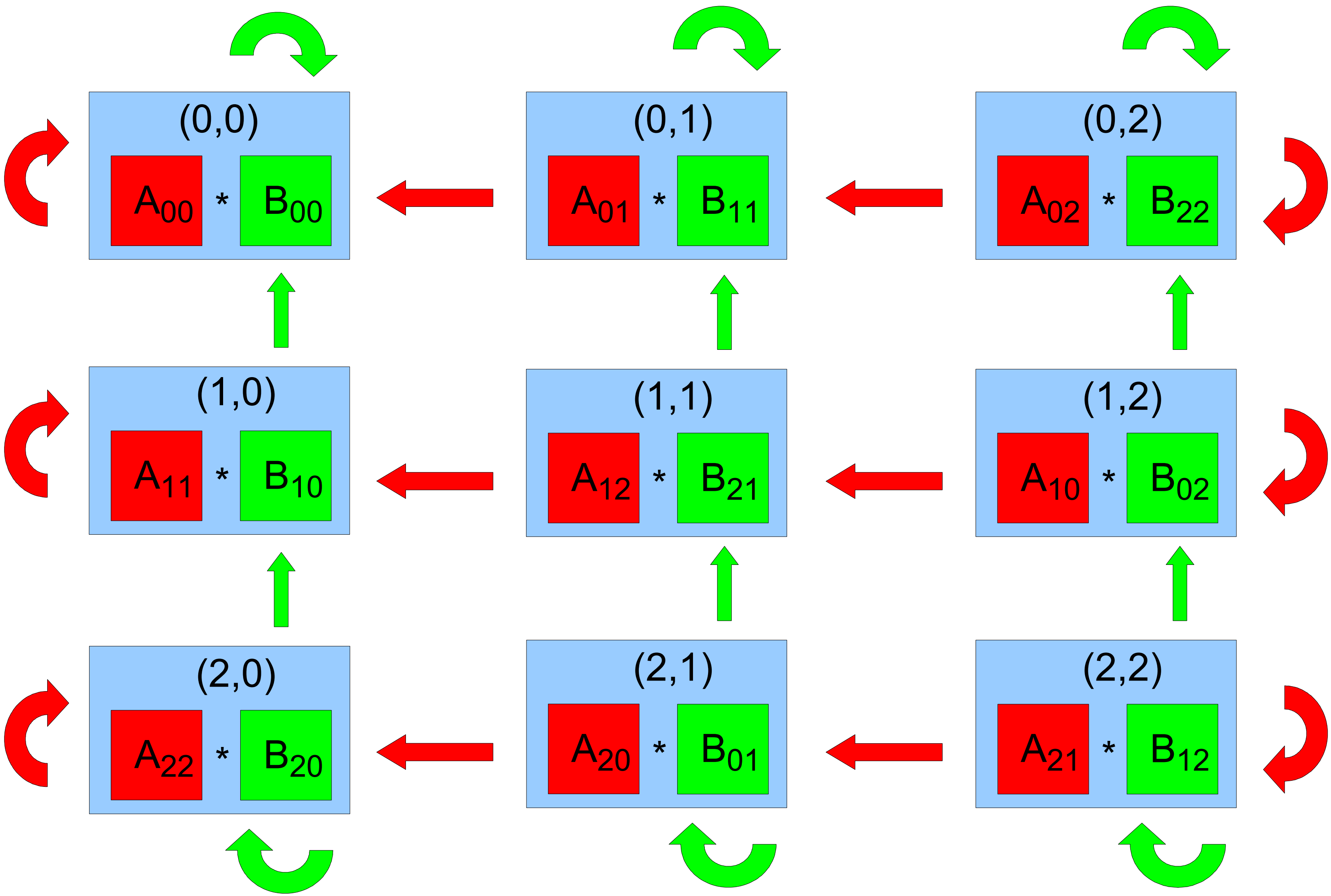
Start at `k=(i+j)%s`

Second summation index value

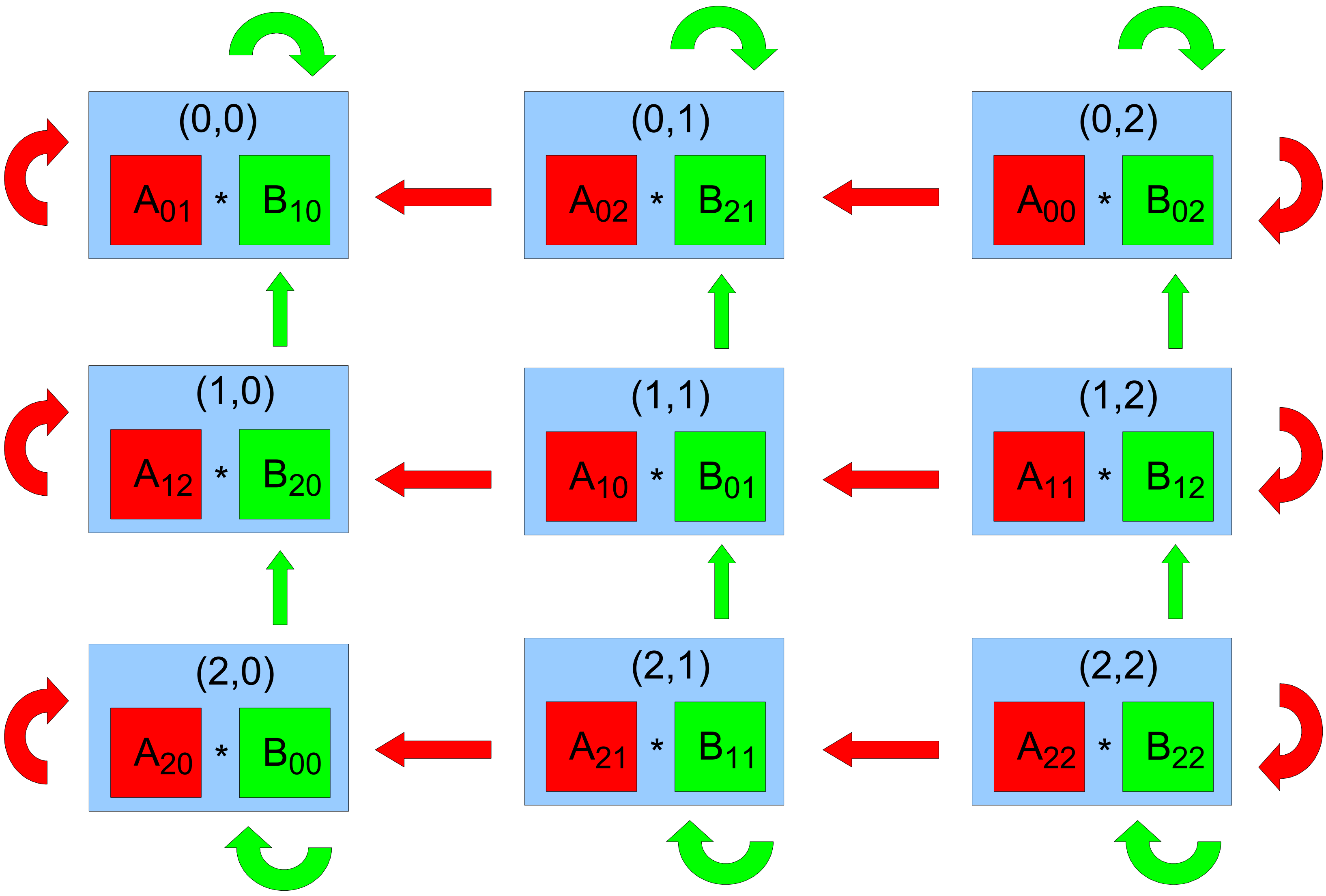
k=1	k=2	k=0
k=2	k=0	k=1
k=0	k=1	k=2

`k=(k+1)%s`

Cannon's algorithm: Step 0



Cannon's algorithm: Step 1



Canon's algorithm

Assumption: Process `(i,j)` holds $\mathbf{A_{ij}}$ and $\mathbf{B_{ij}}$

Canon's algorithm

Assumption: Process (i,j) holds A_{ij} and B_{ij}

Initialize: Rotate A_{ij} i positions left

Rotate B_{ij} j positions up

Canon's algorithm

Assumption: Process `(i,j)` holds A_{ij} and B_{ij}

Initialize: Rotate A_{ij} `i` positions left

Rotate B_{ij} `j` positions up

Compute: Local matrix multiplication: `C = A*B`

Repeat $\sqrt{p}-1$ times:

- Send `A` to the left
- Send `B` up
- Local matrix multiplication: `C += A*B`

Canon's algorithm

Assumption: Process `(i,j)` holds A_{ij} and B_{ij}

Initialize: Rotate A_{ij} i positions left
Rotate B_{ij} j positions up

Compute: Local matrix multiplication: `C = A*B`
Repeat $\sqrt{p}-1$ times:
    Send **A** to the left
    Send **B** up
    Local matrix multiplication: `C += A*B`

Running time analysis:

- Computation: $T_{comp} \in \Theta((n/\sqrt{p})^3\sqrt{p}) = \Theta(n^3/p)$
- Communication: $T_{comm} = 2\sqrt{p}(t_0+(n^2/p)t_1) \in \Theta(n^2/\sqrt{p})$
- $T_{comp}/T_{comm} \in \Theta(n/\sqrt{p})$

Canon's algorithm

Assumption: Process (i,j) holds A_{ij} and B_{ij}

Initialize: Rotate A_{ij} i positions left
Rotate B_{ij} j positions up

Compute: Local matrix multiplication: $C = A*B$
Repeat $\sqrt{p}-1$ times:
Send A to the left
Send B up
Local matrix multiplication: $C += A*B$

Running time analysis:

- Computation: $T_{comp} \in \Theta((n/p)^3 p) = \Theta(n^3/p)$
- Communication: $T_{comm} = 2\sqrt{p}(t_0+(n^2/p)t_1) \in \Theta(n^2/\sqrt{p})$
- $T_{comp}/T_{comm} \in \Theta(n/\sqrt{p})$

Compared to row-wise decomposition:

- Communication reduced by a factor $\sqrt{p}$

Canon's algorithm

Technical:

- Process **`(i,j)`** has id **`i*s+j`**
- Process **`q`** computes block **`(q/s,q%s)`**
- Process **`q`**'s upstairs neighbor has id **`(q-s)%p`**
- Process **`q`**'s downstairs neighbor has id **`(q+s)%p`**
- Process **`q`**'s right neighbor has id **`q+1`** or **`q-s+1`**
- Process **`q`**'s left neighbor has id **`q-1`** or **`q+s-1`**