Lab 6 notes:

Matrix multiply in distributed memory

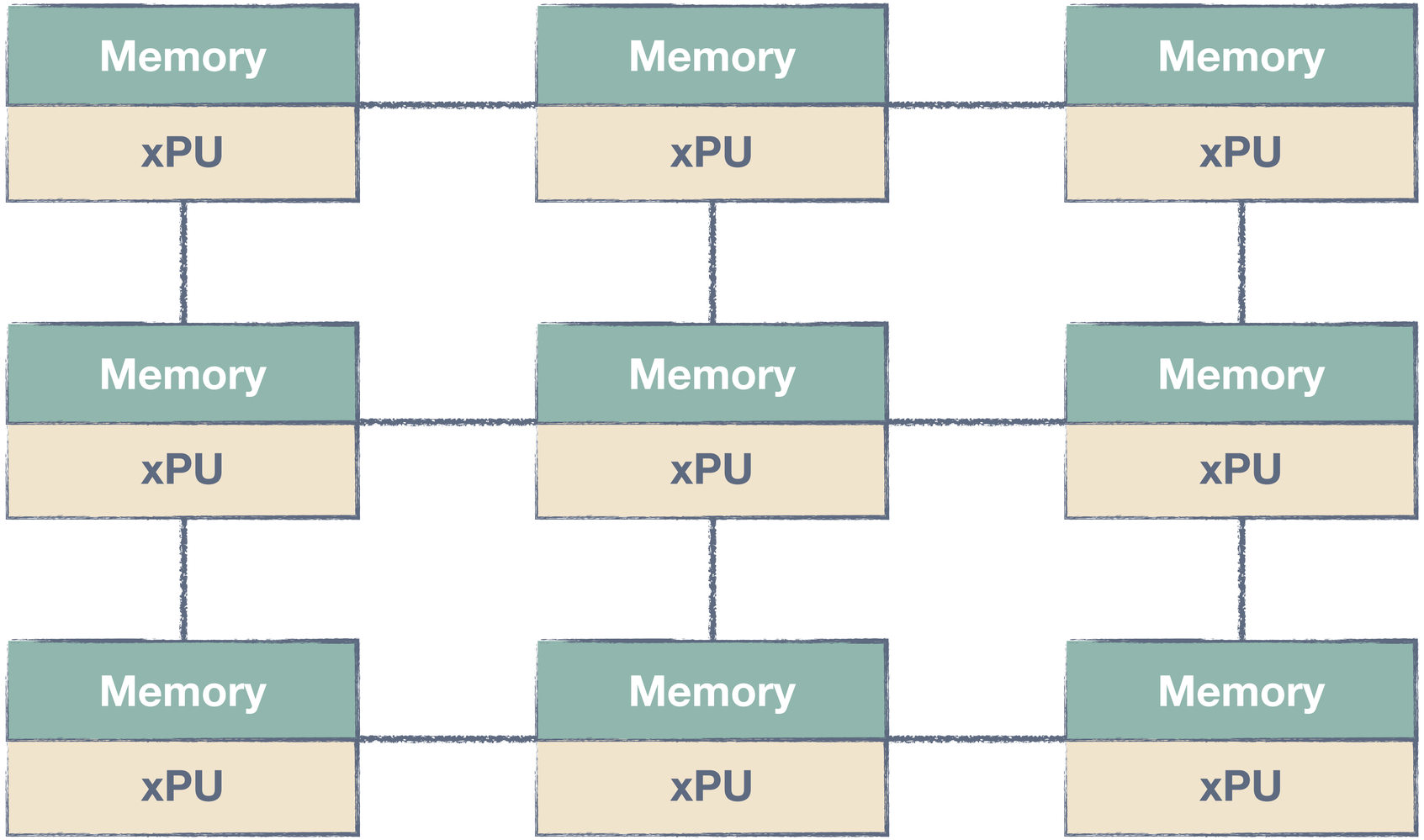
CSE 6230: HPC Tools & Applications
Fall 2014 — October 7 & 9

Georgia Tech College of Computing
Computational Science and Engineering

hpcgarage

Recall:

An abstract machine model for a *distributed memory system*:
Processing "nodes", each with its own private memory,
communicate over a network by "message passing."



“α-β” (latency-bandwidth) cost model

- Model time to send a message in terms of latency and bandwidth

$$t(n) = \alpha + \frac{n}{\beta}$$

- Usually have cost(flop) << 1/β << α
  - One long message cheaper than many short ones
  - Can do ~ thousands of flops for each message
  - Want large computation-to-communication ratio

Recall:

Matrix multiply

$$C \leftarrow C + A \cdot B$$

$$c_{i,j} \leftarrow c_{i,j} + \sum_{i=0}^{n-1} a_{i,k} \cdot b_{k,j}$$

Recall:

Matrix multiply

$$C \leftarrow C + A \cdot B$$

$$c_{i,j} \leftarrow c_{i,j} + \sum_{i=0}^{n-1} a_{i,k} \cdot b_{k,j}$$

0,0	0,1	0,2
1,0	1,1	1,2
2,0	2,1	2,2

B

A

0,0	0,1	0,2
1,0	1,1	1,2
2,0	2,1	2,2

0,0	0,1	0,2
1,0	1,1	1,2
2,0	2,1	2,2

C

Recall:

Matrix multiply

$$C \leftarrow C + A \cdot B$$

$$c_{i,j} \leftarrow c_{i,j} + \sum_{i=0}^{n-1} a_{i,k} \cdot b_{k,j}$$

0,0	0,1	0,2
1,0	1,1	1,2
2,0	2,1	2,2

B

A

0,0	0,1	0,2
1,0	1,1	1,2
2,0	2,1	2,2

C

0,0	0,1	0,2
1,0	1,1	1,2
2,0	2,1	2,2

```
for i ← 0 to n-1
  for j ← 0 to n-1
    for k ← 0 to n-1
      c(i,j) ← c(i,j) + a(i,k)*b(k,j)
```

Recall:

Matrix multiply

$$C \leftarrow C + A \cdot B$$

$$c_{i,j} \leftarrow c_{i,j} + \sum_{i=0}^{n-1} a_{i,k} \cdot b_{k,j}$$

0,0	0,1	0,2
1,0	1,1	1,2
2,0	2,1	2,2

B

A

0,0	0,1	0,2
1,0	1,1	1,2
2,0	2,1	2,2

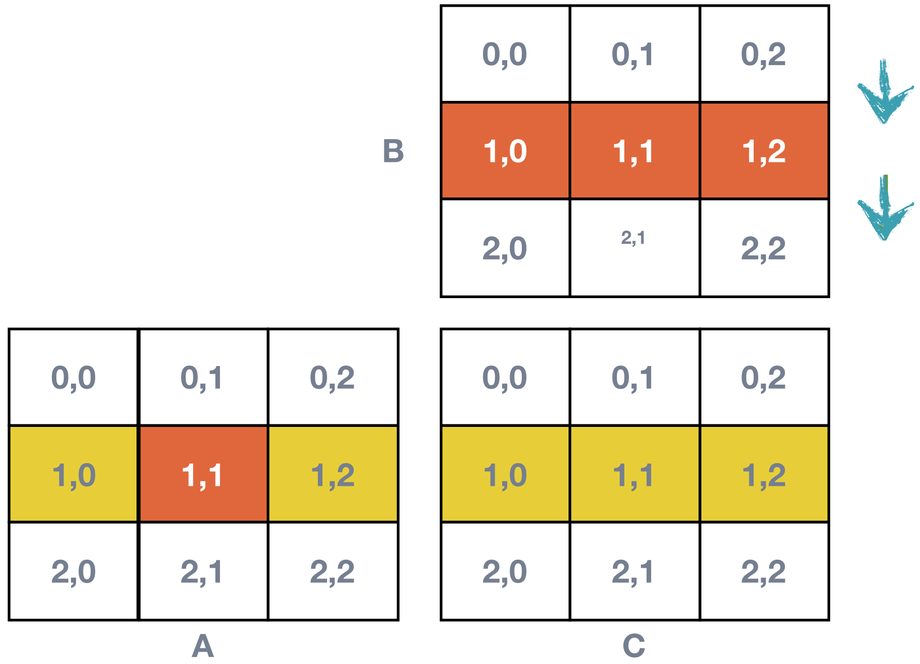
0,0	0,1	0,2
1,0	1,1	1,2
2,0	2,1	2,2

C

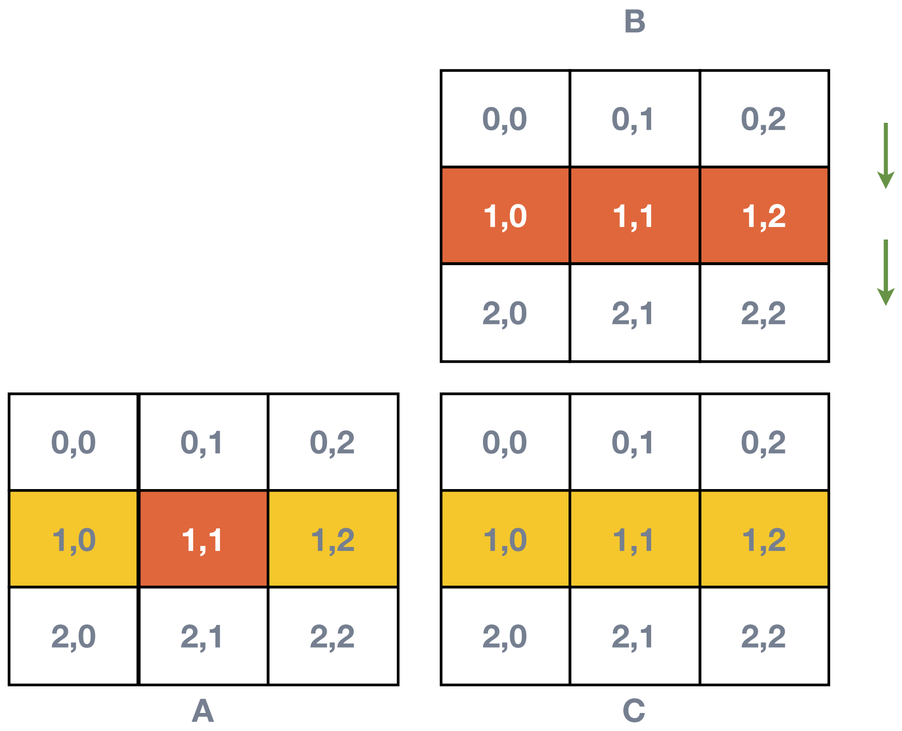
```
par-for i ← 0 to n-1
  par-for j ← 0 to n-1
    for k ← 0 to n-1
      c(i,j) ← c(i,j) + a(i,k)*b(k,j)
```

1D block row-based algorithm

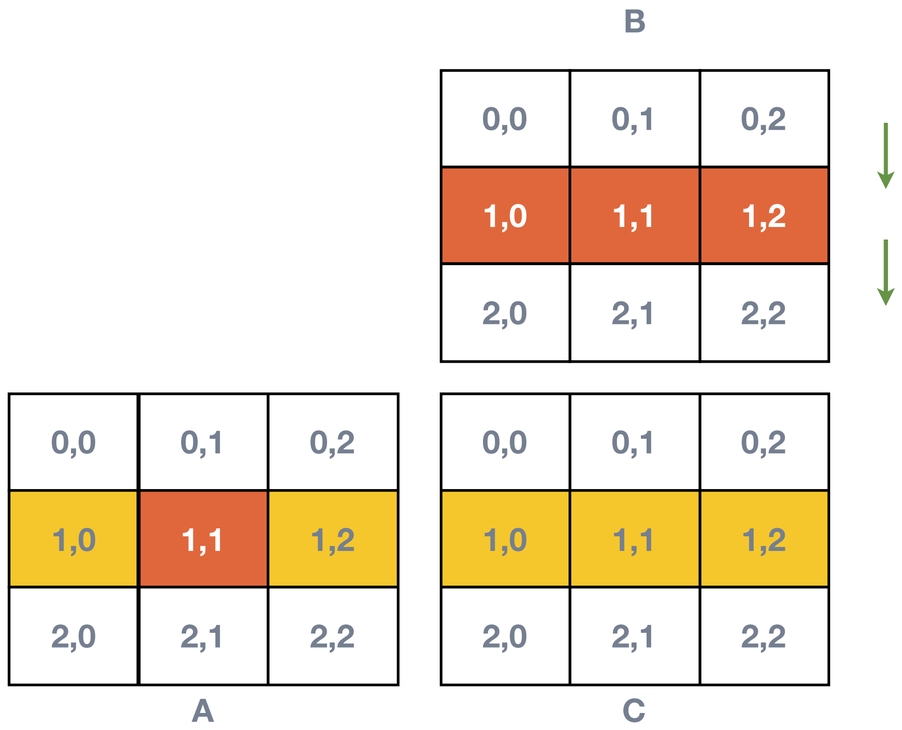
- $n \times n$ matrix multiply on P processors
- Block size $b = n / P$
- At any time, processor owns same block row of A, C
- Owns some block row of B (passed along)
- Must eventually see all of B



```
// A, B, and C are n x n
// P processes (tasks) total, where P divides n
// Local process id is r in {0, 1, …, P-1}

// Assume initial distribution:
//   A_local = A(r*n/P : (r+1)n/P-1, :)
//   B_local, C_local similarly
//   Temporary storage B_local'

// Code for process r:
r_prev ← (r+ P-1) % P
r_next ← (r+1) % P
for k = 0 to P-1 do:
  block ← …  // Compute block index
  C_local += A_local(block) * B_local;
  send_async (B_local → r_next)
  receive_async (B_local' ← r_prev)
  wait_all ()
  swap (B_local, B_local')
```



```
// A, B, and C are n x n
// P processes (tasks) total, where P divides n
// Local process id is r in {0, 1, …, P-1}

// Assume initial distribution:
//   A_local = A(r*n/P : (r+1)n/P-1, :)
//   B_local, C_local similarly
//   Temporary storage B_local'

// Code for process r:
r_prev ← (r+ P-1) % P
r_next ← (r+1) % P
for k = 0 to P-1 do:
  block ← …  // Compute block index
  C_local += A_local(block) * B_local;
  send_async (B_local → r_next)
  receive_async (B_local' ← r_prev)
  wait_all ()
  swap (B_local, B_local')
```



$$T_{\text{net}}(n) = P \times \left(\alpha + \frac{n^2/P}{\beta} \right)$$

Time, speedup, and efficiency of the 1D block row-based algorithm

$$T_{\text{comp}}(n;p) = \frac{2n^3}{p}$$

$$T_{\text{net}}(n;p) = \alpha \cdot p + \frac{n^2}{\beta}$$

$$S(n;p) = \frac{p}{1 + \alpha\frac{p^2}{n^3} + \frac{1}{\beta}\frac{p}{n}}$$

$$E(n;p) \equiv S(n;p)/p$$

“Perfect” speedup possible …

But n ~ p, so large flop and memory req’d to scale

Bonus materials

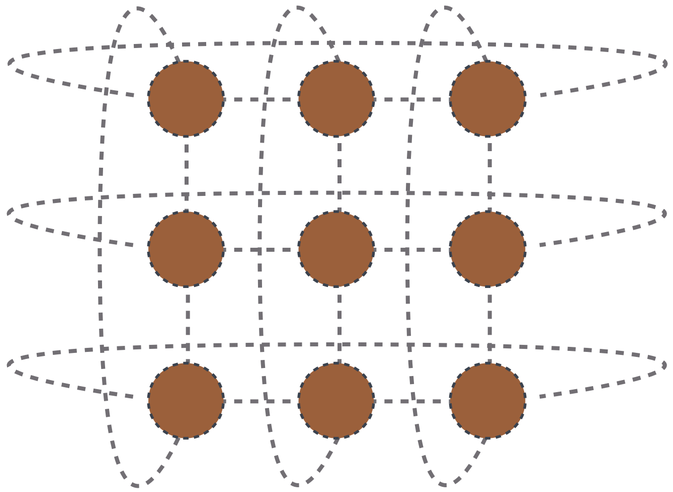
(These are really CSE 6220 topics — lucky you!)

SUMMA: A more scalable matrix multiply

By van de Geijn and Watts: http://www.netlib.org/lapack/lawns/lawn96.ps
Used in the Parallel BLAS: http://www.netlib.org/lapack/lawns/lawn100.ps

2D torus networks

- Assume square grid of P processes total
  - sqrt (P) = integer
- Links to nearest neighbors with wraparound links



Example:
P = 9
sqrt (P) = 3

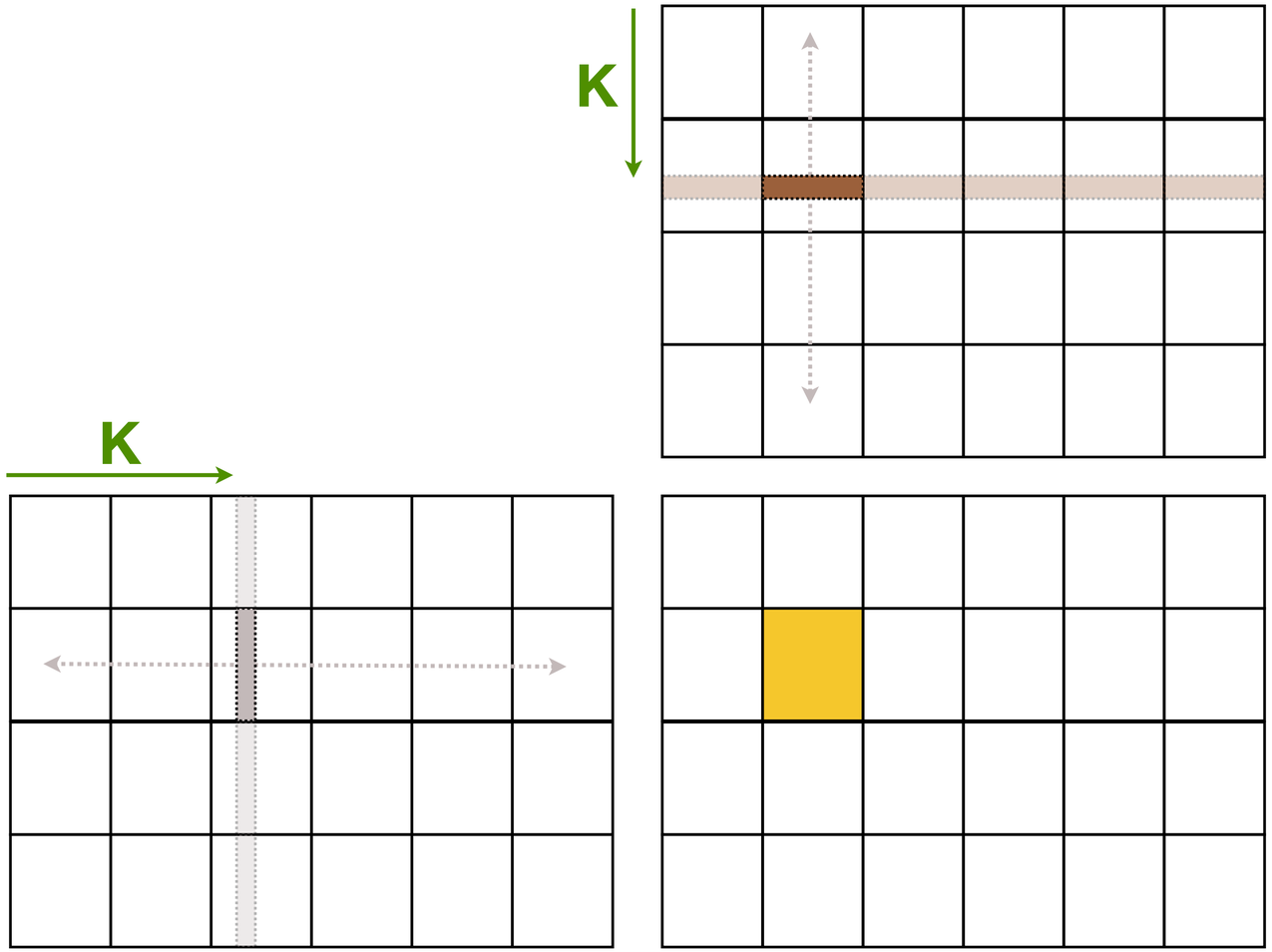
Equivalent formulations of matrix multiply

Dot products

$$
\begin{aligned}
C = A \cdot B &= \begin{pmatrix} \hat{a}_1^T \\ \vdots \\ \hat{a}_m^T \end{pmatrix} \cdot \begin{pmatrix} b_1 & \cdots & b_n \end{pmatrix} = \begin{pmatrix} \hat{a}_1^T \cdot b_1 & \cdots & \hat{a}_1^T \cdot b_n \\ \vdots & & \vdots \\ \hat{a}_m^T \cdot b_1 & \cdots & \hat{a}_m^T \cdot b_n \end{pmatrix} \\
\implies c_{ij} &= \hat{a}_i^T \cdot b_j
\end{aligned}
$$

Outer products

$$
C = A \cdot B = \begin{pmatrix} a_1 \cdots a_k \end{pmatrix} \cdot \begin{pmatrix} \hat{b}_1^T \\ \vdots \\ \hat{b}_k^T \end{pmatrix} = a_1 \hat{b}_1^T + \cdots + a_k \hat{b}_k^T = \sum_k a_k \hat{b}_k^T
$$

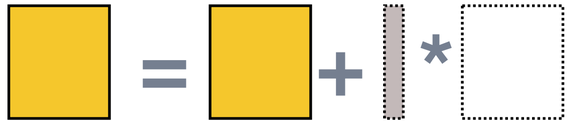


SUMMA:

for-all **K**:

Broadcast strip

Broadcast strip



Cost of SUMMA

- Assume:
  - Square matrix multiply, dim = ***n***
  - sqrt(***P***) x sqrt(***P***) processor grid
  - Panel block size ***b***
  - Tree-based broadcast

SUMMA:

for-all **K**:

Broadcast strip

Broadcast strip

Time, efficiency, and space of SUMMA

For $1 \leq b \leq \frac{n}{\sqrt{P}}$:

b to trades latency and storage

$$T_{\text{net}}(n;P) = \alpha \frac{n}{b} \log P + \frac{n^2}{\beta} \frac{\log P}{\sqrt{P}}$$

$$E(n;P) = \frac{1}{1 + \frac{\alpha}{2} \frac{P \log P}{n^2 b} + \frac{1}{2\beta} \frac{\sqrt{P}}{n} \log P}$$

$$M(n;P) = 3\frac{n^2}{P} + 2\frac{nb}{\sqrt{P}} \equiv \text{per proc. mem.}$$

Communication in matrix multiply

(so far, except Cannon)

1-D block row: $T_{\text{net}}(n;p) = \alpha \cdot p + \frac{1}{\beta} \cdot n^2$

2-D SUMMA: $T_{\text{net}}(n;p) = \alpha \cdot \frac{n}{b} \log p + \frac{n^2}{\beta} \cdot \frac{\log p}{\sqrt{p}}$

(*Tree-based*)

where $1 \leq b \leq \frac{n}{\sqrt{p}}$

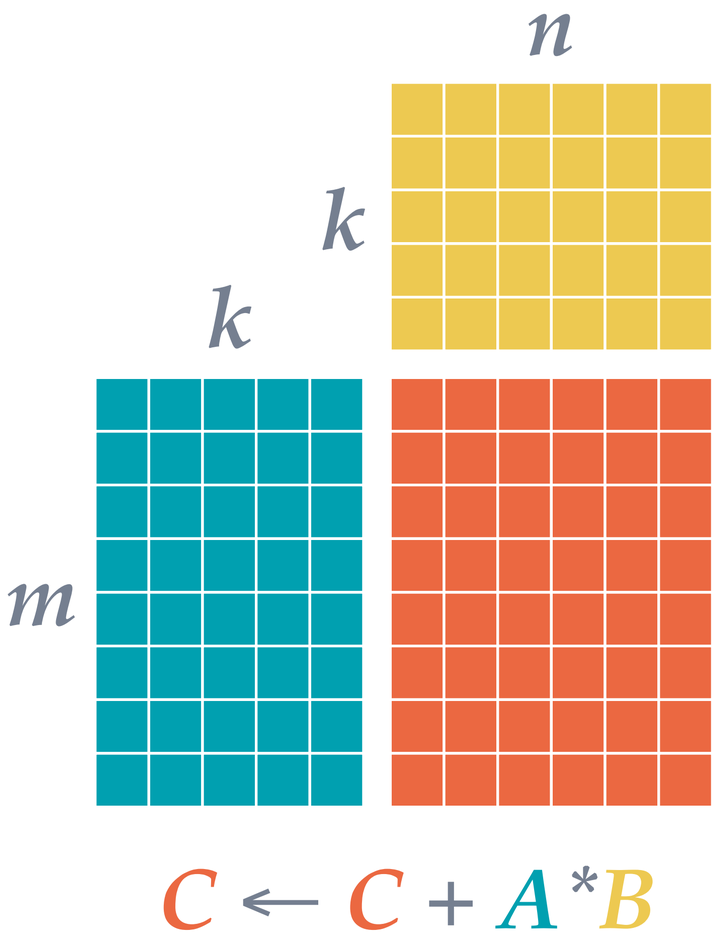
Cannon: $T_{\text{net}}(n;p) = \alpha\sqrt{p} + \frac{1}{\beta}\frac{n^2}{\sqrt{p}}$

Can we do better?

Cannon: $T_{\text{net}}(n;p) = \alpha\sqrt{p} + \frac{1}{\beta}\frac{n^2}{\sqrt{p}}$

D. Irony, S. Toledo, A. Tiskin. “Communication lower-bounds for distributed-memory matrix multiplication.” *JPDC*, **64**, p. 1017—1026, 2004.

Recall: Matrix multiply (non-square, non-Strassen)

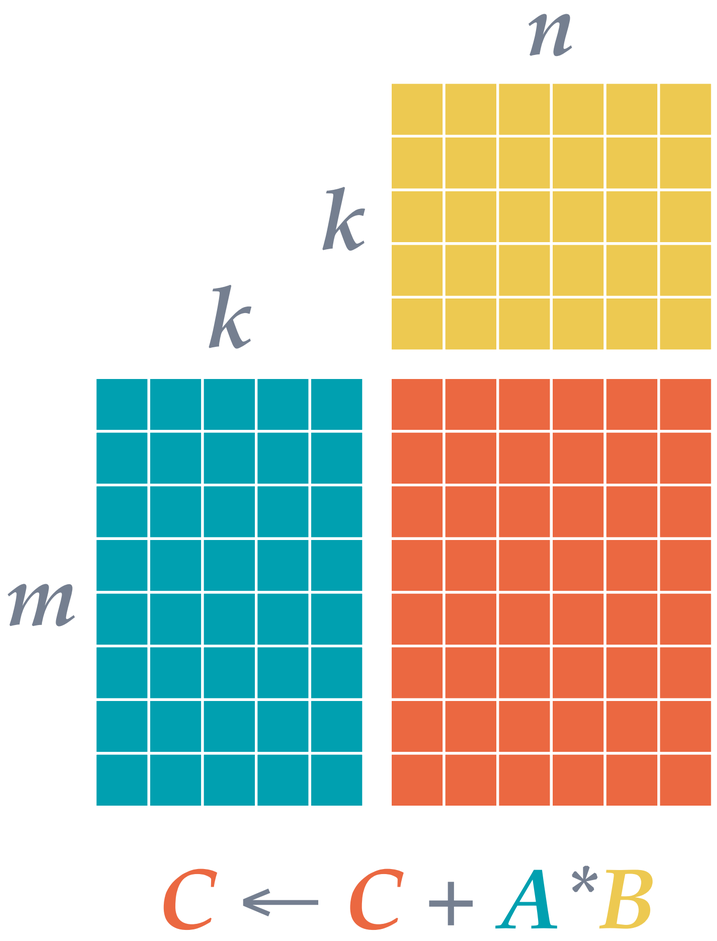


$$A: \quad m \times k$$
$$B: \quad k \times n$$
$$C: \quad m \times n$$

total multiplies $= \quad m \cdot k \cdot n$

D. Irony, S. Toledo, A. Tiskin. “Communication lower-bounds for distributed-memory matrix multiplication.” *JPDC*, **64**, p. 1017—1026, 2004.

Recall: Matrix multiply (non-square, non-Strassen)



total multiplies $=$ $m \cdot k \cdot n$

Let S_A $=$ some subset of the elements of A

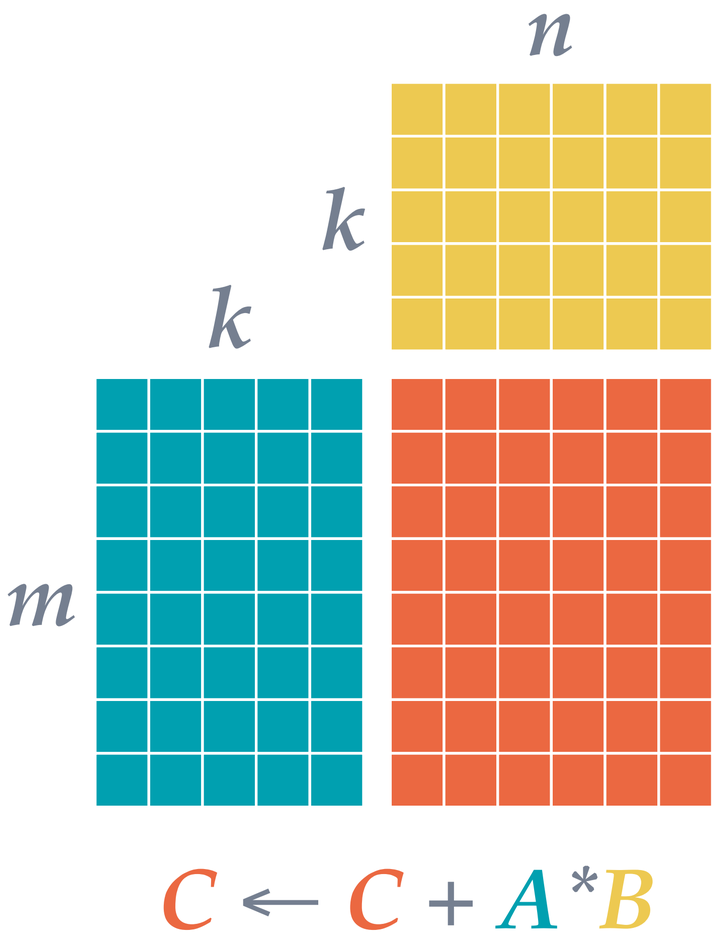
S_B $=$ some subset of the elements of B

S_C $=$ some subset of the elements of C

no. multiplies $=$??

D. Irony, S. Toledo, A. Tiskin. “Communication lower-bounds for distributed-memory matrix multiplication.” *JPDC*, **64**, p. 1017—1026, 2004.

Recall: Matrix multiply (non-square, non-Strassen)



total multiplies $= m \cdot k \cdot n$

Let S_A = some subset of the elements of A

S_B = some subset of the elements of B

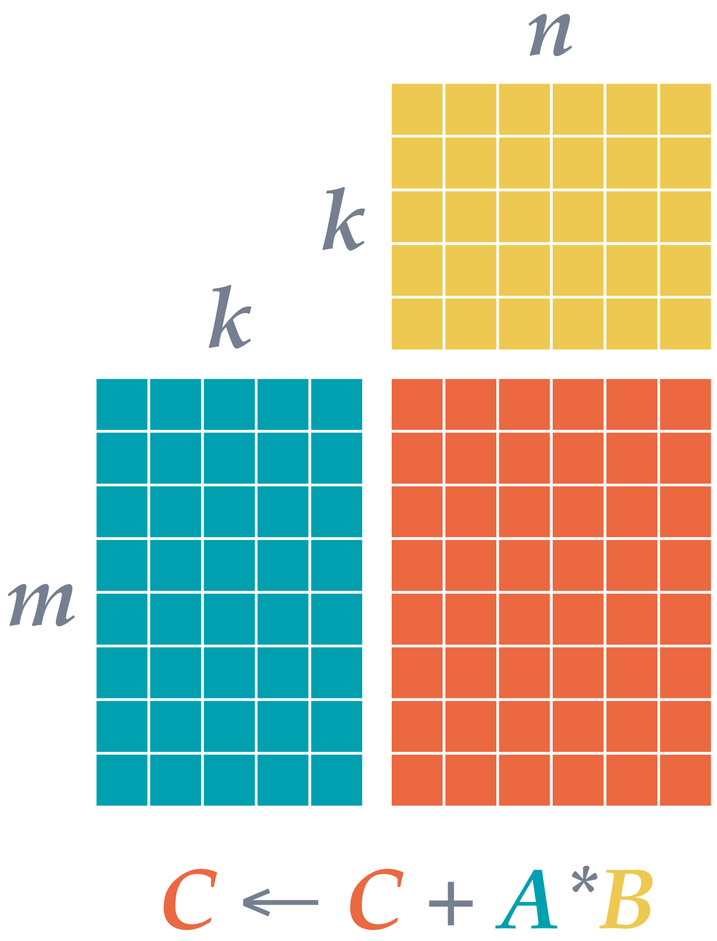
S_C = some subset of the elements of C

no. multiplies = ?? $\leq \sqrt{|S_A| \cdot |S_B| \cdot |S_C|}$

By *discrete Loomis-Whitney inequality*

D. Irony, S. Toledo, A. Tiskin. “Communication lower-bounds for distributed-memory matrix multiplication.” *JPDC*, **64**, p. 1017—1026, 2004.

Recall: Matrix multiply (non-square, non-Strassen)



Consider one of the processors. Suppose it has M words of local memory and executes W multiplies.

How many words ***must*** it communicate? (i.e., send & receive)

D. Irony, S. Toledo, A. Tiskin. “Communication lower-bounds for distributed-memory matrix multiplication.” *JPDC*, **64**, p. 1017—1026, 2004.

Consider a schedule of operations …

= computation = communication

…

D. Irony, S. Toledo, A. Tiskin. “Communication lower-bounds for distributed-memory matrix multiplication.” *JPDC*, **64**, p. 1017—1026, 2004.

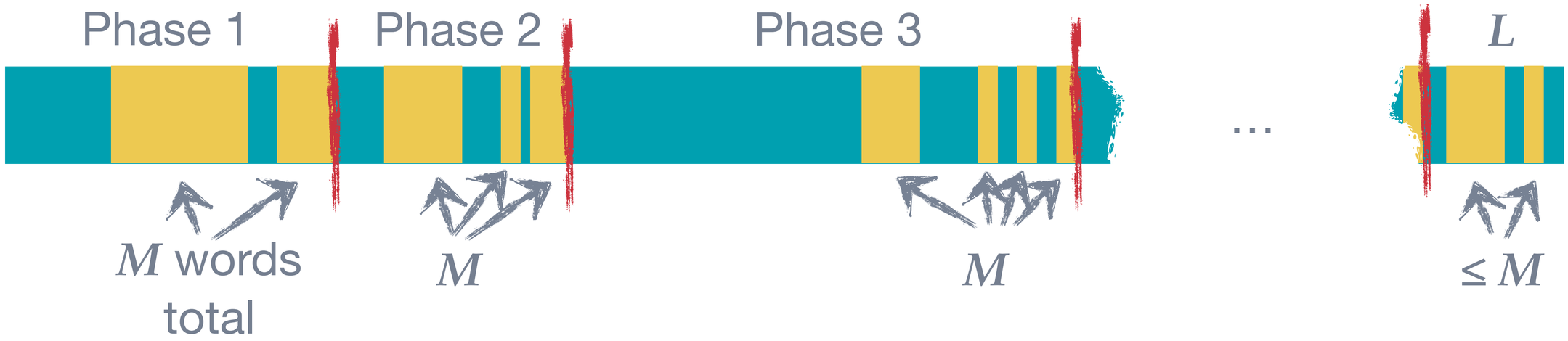
Consider a schedule of operations …

= computation = communication

D. Irony, S. Toledo, A. Tiskin. “Communication lower-bounds for distributed-memory matrix multiplication.” *JPDC*, **64**, p. 1017—1026, 2004.

Consider a schedule of operations …

= computation = communication

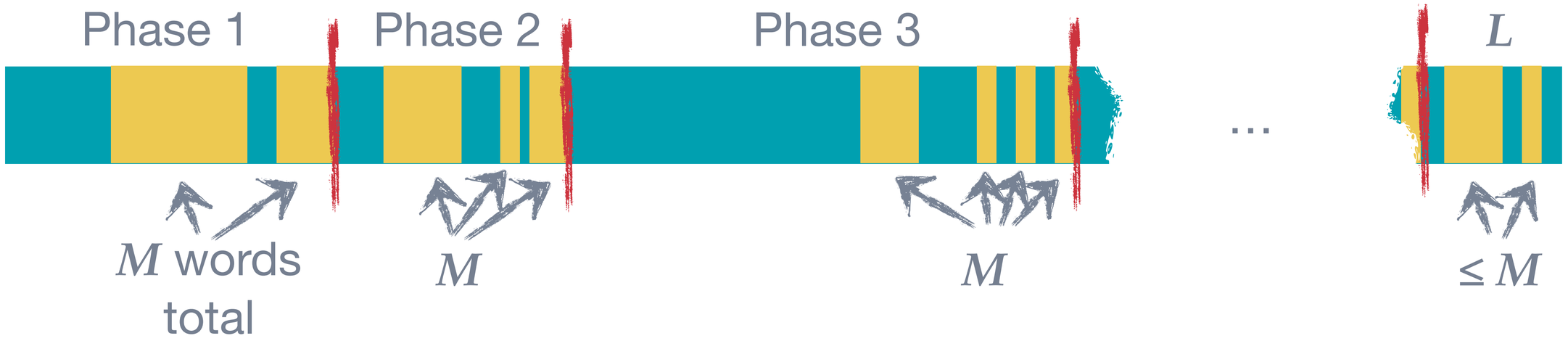


… broken into L phases, where each phase sends or receives exactly M words, except possibly the last.

D. Irony, S. Toledo, A. Tiskin. “Communication lower-bounds for distributed-memory matrix multiplication.” *JPDC*, **64**, p. 1017—1026, 2004.

Consider a schedule of operations …

= computation = communication

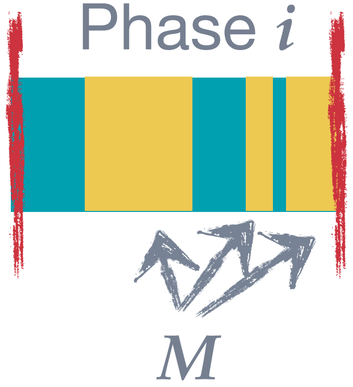


What is the **maximum** number of **multiplies** each phase can perform?

D. Irony, S. Toledo, A. Tiskin. “Communication lower-bounds for distributed-memory matrix multiplication.” *JPDC*, **64**, p. 1017—1026, 2004.

Consider a schedule of operations …

= computation = communication



Recall:

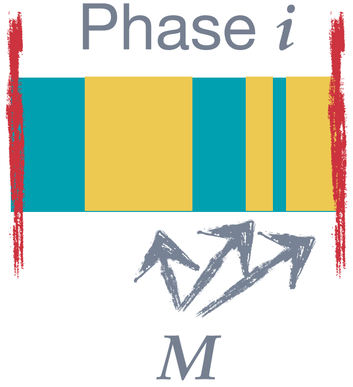
$$\#\text{ mults} \leq \sqrt{|S_A| \cdot |S_B| \cdot |S_C|}$$

$$|S_A| \leq ??$$
$$|S_B| \leq ??$$
$$|S_C| \leq ??$$

D. Irony, S. Toledo, A. Tiskin. “Communication lower-bounds for distributed-memory matrix multiplication.” *JPDC*, **64**, p. 1017—1026, 2004.

Consider a schedule of operations …

= computation = communication



Recall:

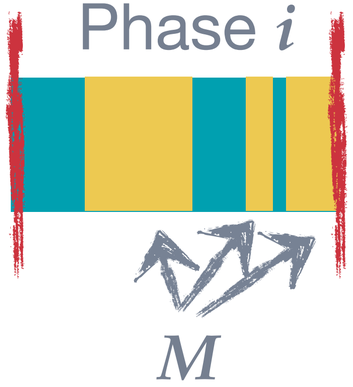
$$\#\text{ mults} \leq \sqrt{|S_A| \cdot |S_B| \cdot |S_C|}$$

$$|S_A| \leq 2M$$
$$|S_B| \leq 2M$$
$$|S_C| \leq 2M$$

D. Irony, S. Toledo, A. Tiskin. “Communication lower-bounds for distributed-memory matrix multiplication.” *JPDC*, **64**, p. 1017—1026, 2004.

Consider a schedule of operations …

= computation = communication



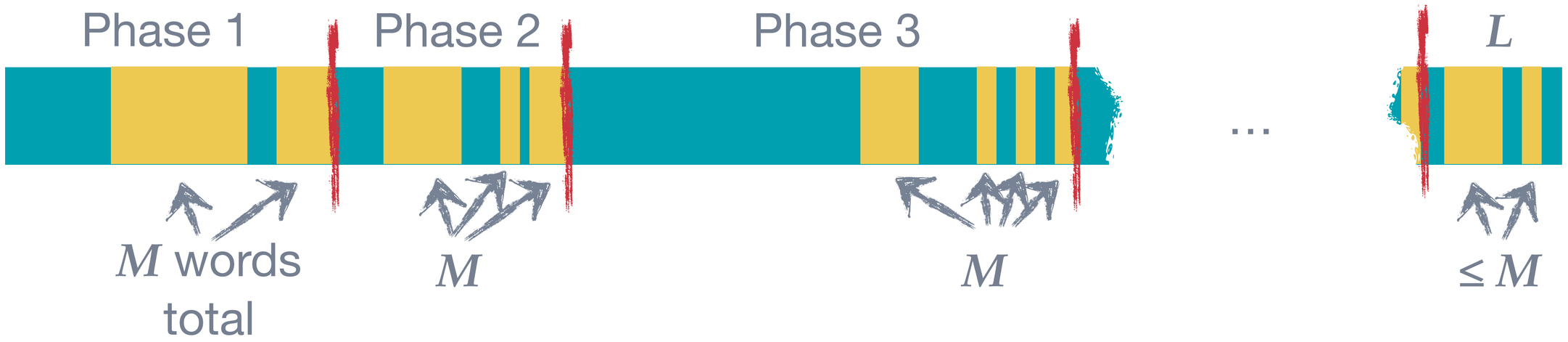
Recall:

$$\#\text{ mults} \leq \sqrt{|S_A| \cdot |S_B| \cdot |S_C|}$$
$$\leq 2\sqrt{2} \cdot M^{3/2}$$

D. Irony, S. Toledo, A. Tiskin. “Communication lower-bounds for distributed-memory matrix multiplication.” *JPDC*, **64**, p. 1017—1026, 2004.

Consider a schedule of operations …

= computation = communication



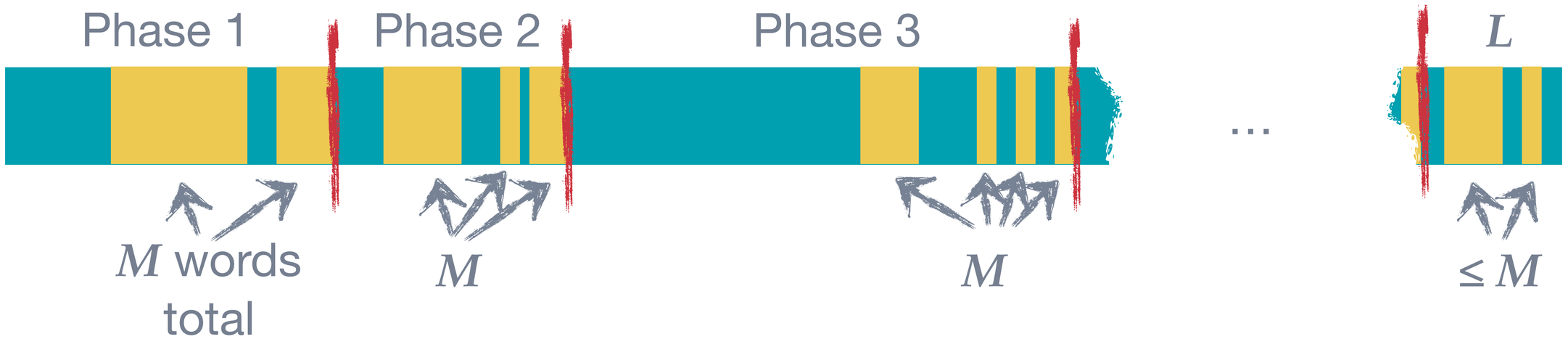
$$\text{\# mults in phase } i \quad \leq \quad 2\sqrt{2} \cdot M^{3/2}$$

$$\implies L \quad \geq \quad ??$$

D. Irony, S. Toledo, A. Tiskin. “Communication lower-bounds for distributed-memory matrix multiplication.” *JPDC*, **64**, p. 1017—1026, 2004.

Consider a schedule of operations …

= computation = communication



$$\text{\# mults in phase } i \quad \leq \quad 2\sqrt{2} \cdot M^{3/2}$$

$$\implies L \quad \geq \quad \left\lfloor \frac{W}{2\sqrt{2} \cdot M^{3/2}} \right\rfloor \geq \frac{W}{2\sqrt{2} \cdot M^{3/2}} - 1$$

D. Irony, S. Toledo, A. Tiskin. “Communication lower-bounds for distributed-memory matrix multiplication.” *JPDC*, **64**, p. 1017—1026, 2004.

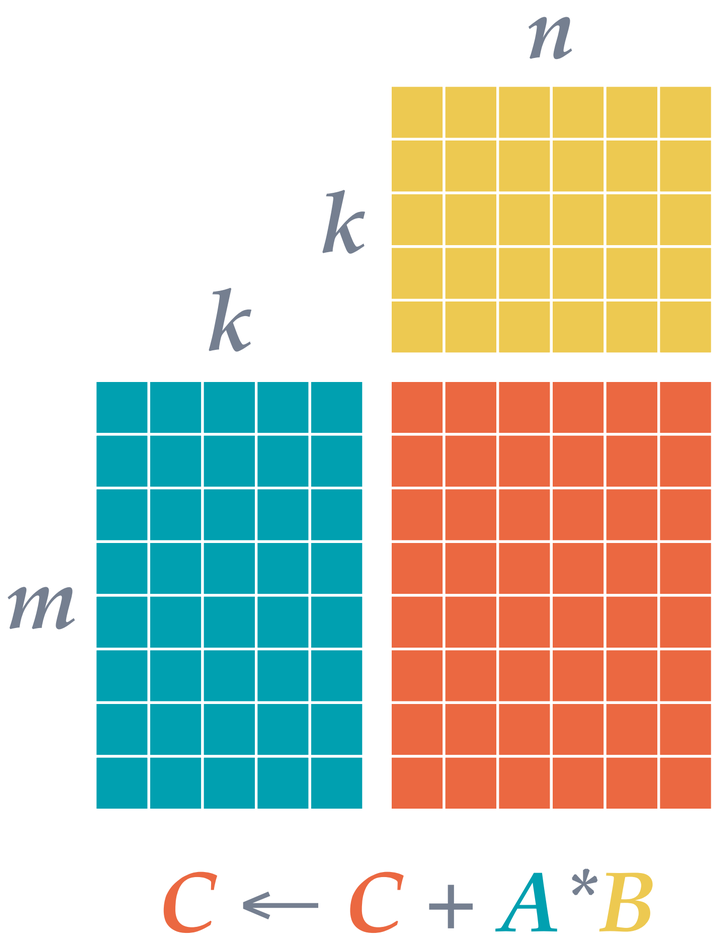
Consider a schedule of operations …

= computation = communication

$$\therefore \text{ total \# words moved } \gtrsim L \cdot M \geq \frac{W}{2\sqrt{2}\sqrt{M}} - M$$

D. Irony, S. Toledo, A. Tiskin. “Communication lower-bounds for distributed-memory matrix multiplication.” *JPDC*, **64**, p. 1017—1026, 2004.

Analysis for the full computation

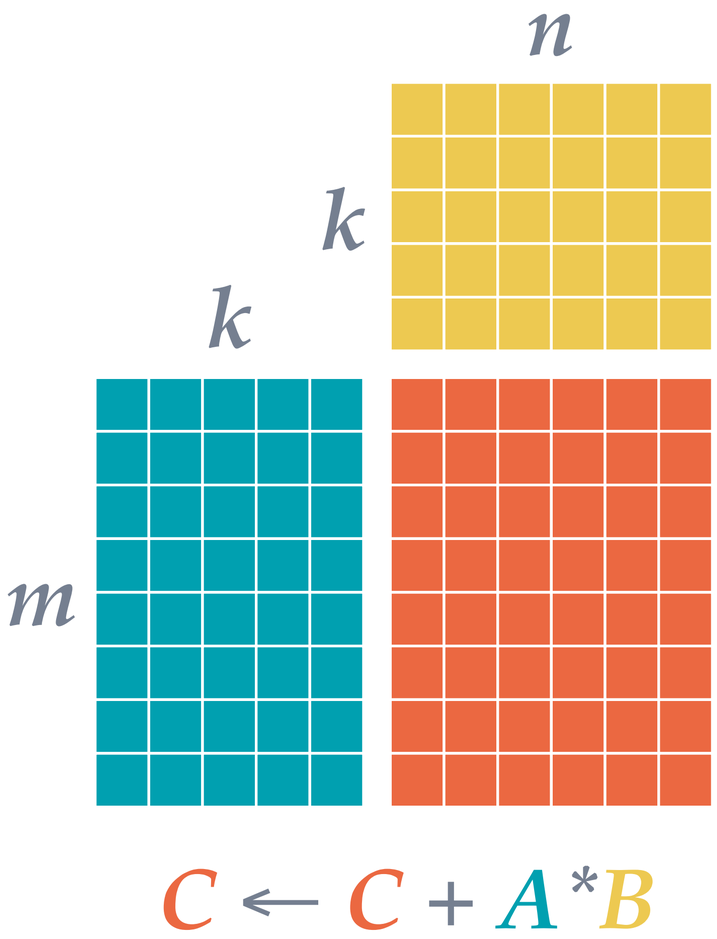


If there are p processors, at least one of them must perform $m \cdot k \cdot n \ / \ p$ multiplies. Thus, the total number of words moved by the entire computation is:

$$\frac{mkn}{2\sqrt{2} \cdot p\sqrt{M}} - M$$

D. Irony, S. Toledo, A. Tiskin. “Communication lower-bounds for distributed-memory matrix multiplication.” *JPDC*, **64**, p. 1017—1026, 2004.

Analysis for the full computation



Recall:

Cannon: $T_{\text{net}}(n;p) = \alpha\sqrt{p} + \frac{1}{\beta}\frac{n^2}{\sqrt{p}}$

$$\frac{mkn}{2\sqrt{2}\cdot p\sqrt{M}} - M$$

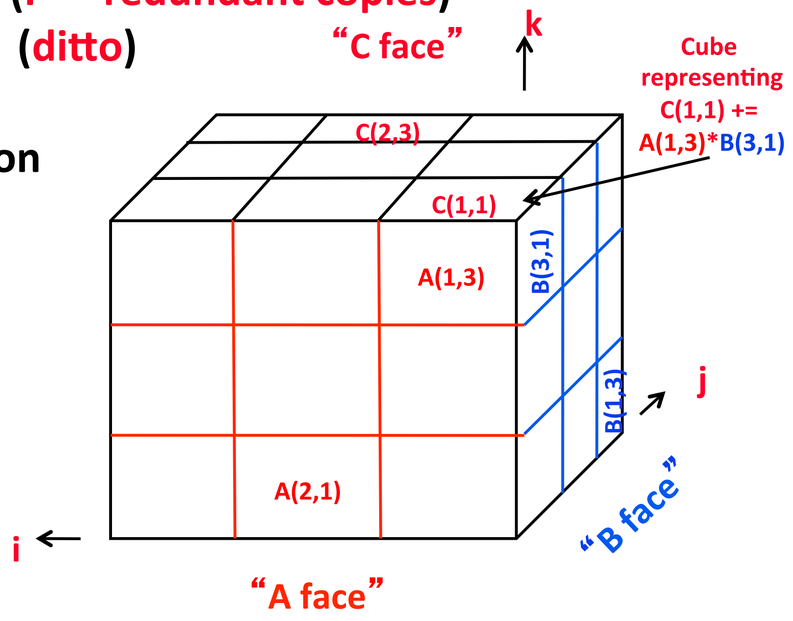
Thus, Cannon is optimal in the bandwidth term. What about latency?

D. Irony, S. Toledo, A. Tiskin. “Communication lower-bounds for distributed-memory matrix multiplication.” *JPDC*, **64**, p. 1017—1026, 2004.

Cannon is optimal, but we assumed we only had enough local memory to store one copy, i.e., $M = O(n^2/p)$. If we have more memory, we can further reduce communication using so-called **"3D"** algorithms.

Beating #words_moved = $\Omega(n^2/P^{1/2})$

- #words_moved = $\Omega((n^3/P)/local_mem^{1/2})$
- If one copy of data, local_mem = n^2/P
- Can we use more memory to communicate less?
- "3D Matmul" 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 (ditto)
  - Local multiplies
  - Reduce (sum) in k direction



- Communication volume = $O((n/P^{1/3})^2)$ - optimal
- Number of messages = O(log(P)) – optimal

- What if we don't have memory for $P^{1/3}$ copies?

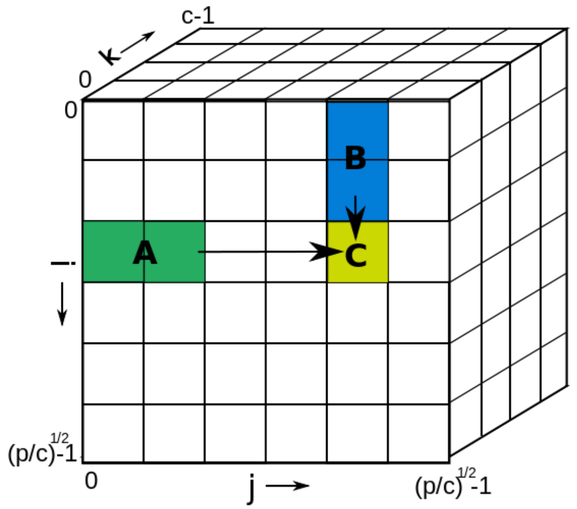
Slide: Jim Demmel (UCB) — CS 267
http://www.cs.berkeley.edu/~demmel/cs267_Spr11/

D. Irony, S. Toledo, A. Tiskin. "Communication lower-bounds for distributed-memory matrix multiplication." *JPDC*, **64**, p. 1017—1026, 2004.

If the 3D algorithm uses too much memory, there is a spectrum of algorithms in between the 2D Cannon and 3D, called **“2.5D”** algorithms.

2.5D matrix multiply

- **Interpolate between Cannon and 3D matmul**
  - **Replicate A, B c-1 times**
  - **Do $p^{1/2}/c^{3/2}$ steps of Cannon on each copy of A,B**
  - **Sum contributions to C from all c layers**



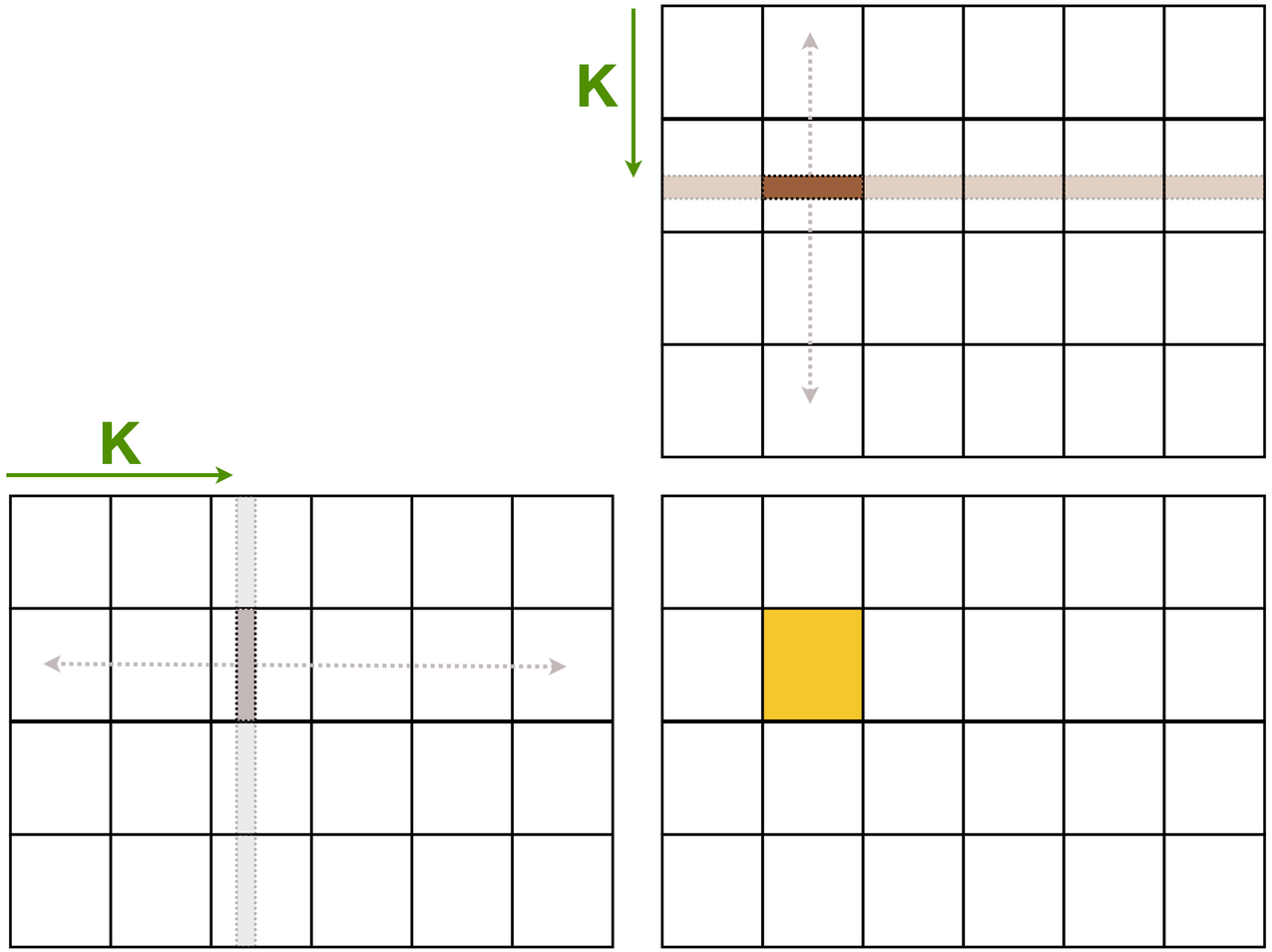
- **#words_moved = $O(n^2 / (cp)^{1/2})$**
- **#messages = $O(p^{1/2}/c^{3/2} + \log(c))$**

Source: Edgar Solomonik

Slide: Jim Demmel (UCB) — CS 267
http://www.cs.berkeley.edu/~demmel/cs267_Spr11/

E. Solomonik and J. Demmel. “Communication-optimal parallel 2.5D matrix multiplication and LU factorization algorithms.” Technical report, U.C. Berkeley, UCB/EECS-2011-10, 2011. http://www.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-10.html

- **Note on SUMMA implementation: Cartesian Communicators in MPI**



SUMMA:

for-all **K**:

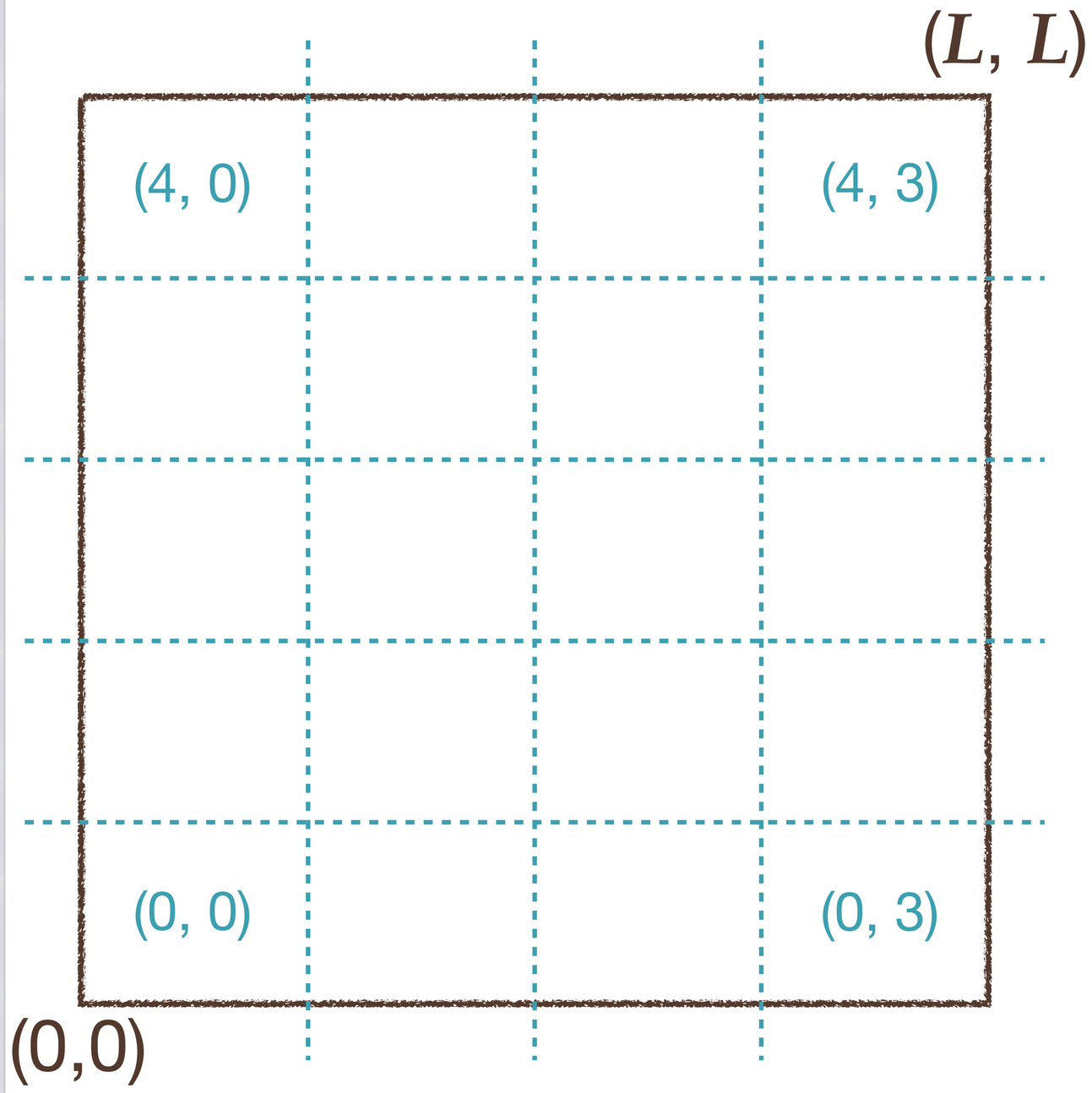
Broadcast strip

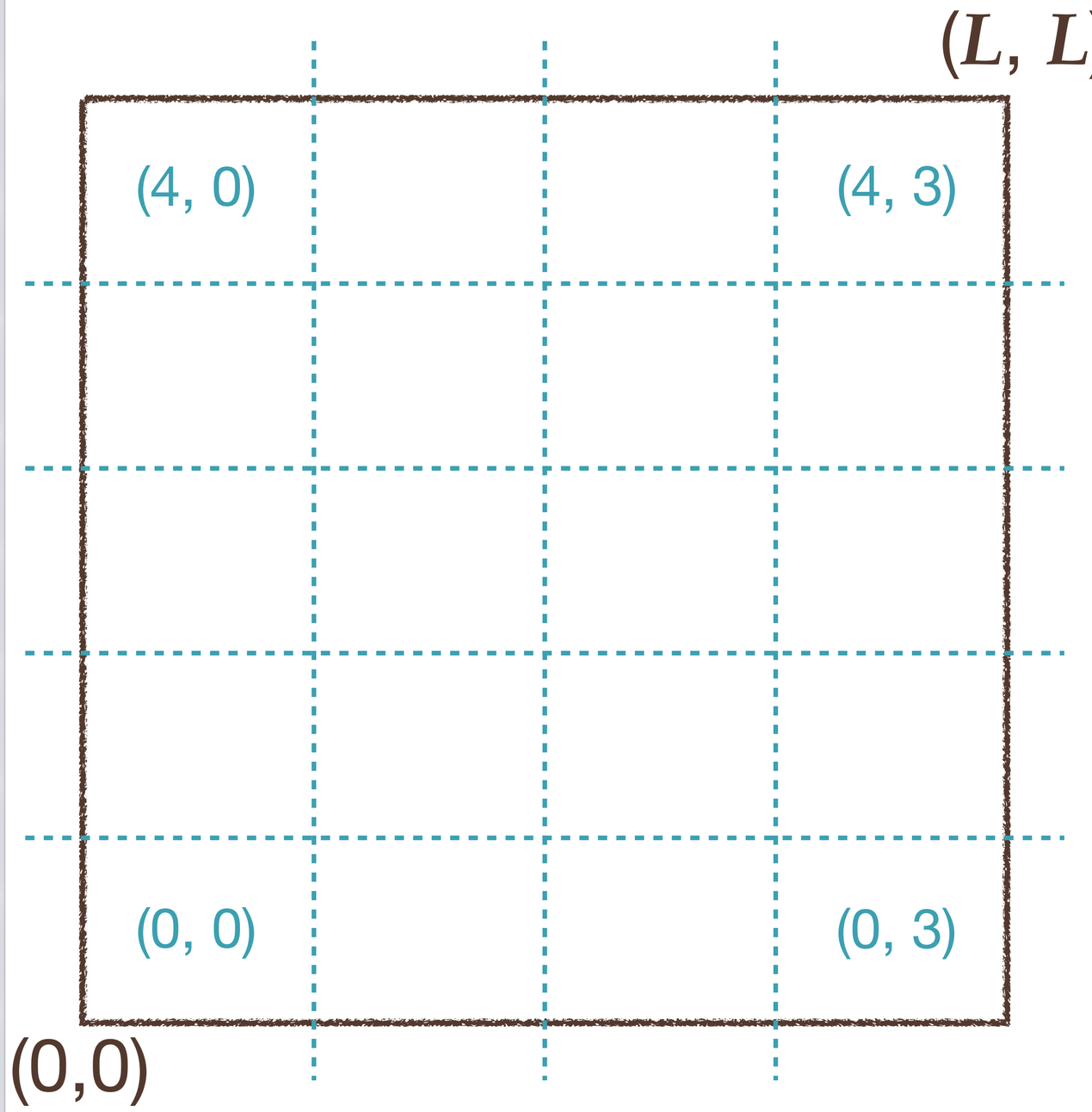
Broadcast strip

MPI_Bcast (…, comm) → comm != MPI_COMM_WORLD

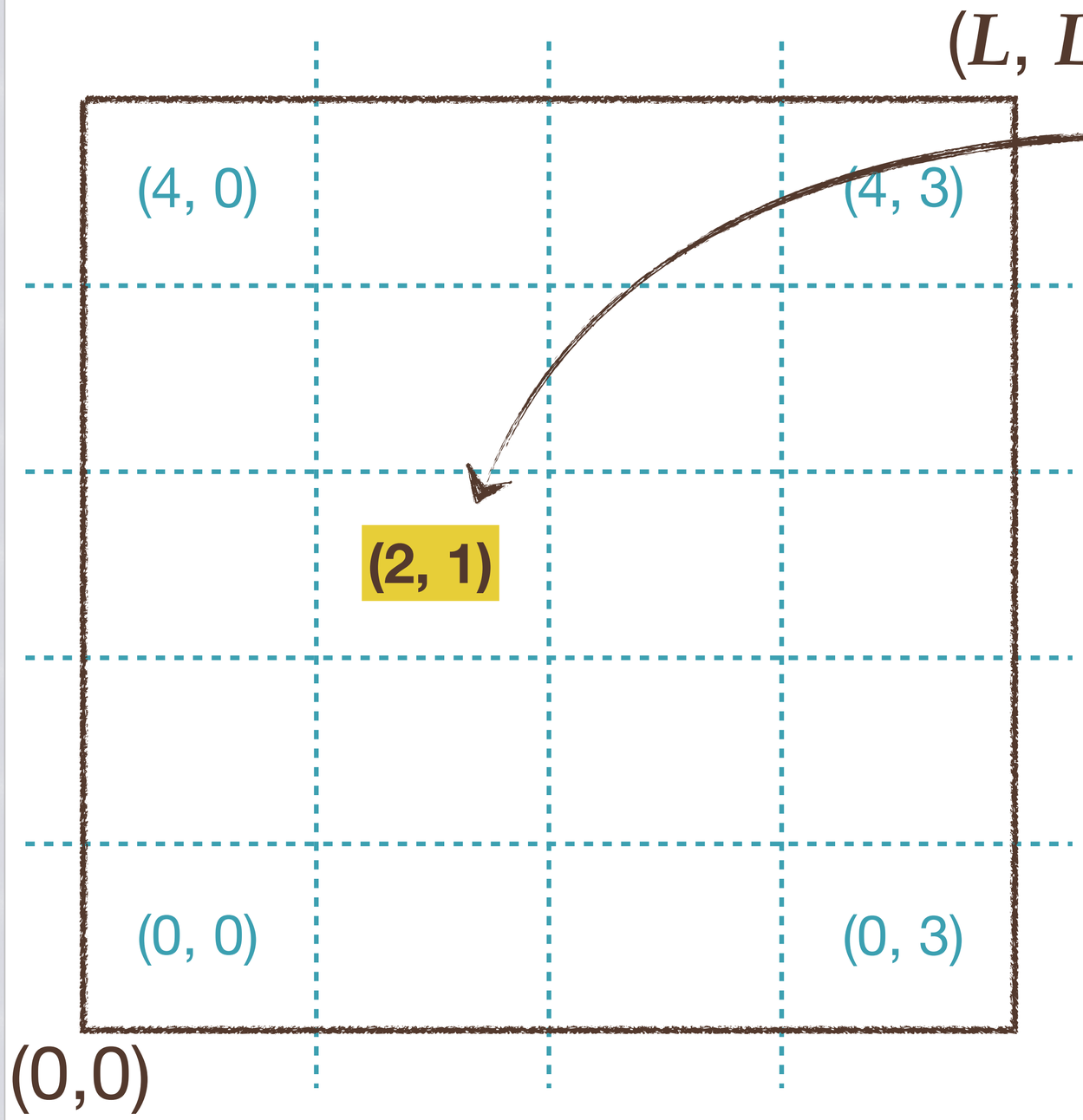
(L, L)

(0,0)

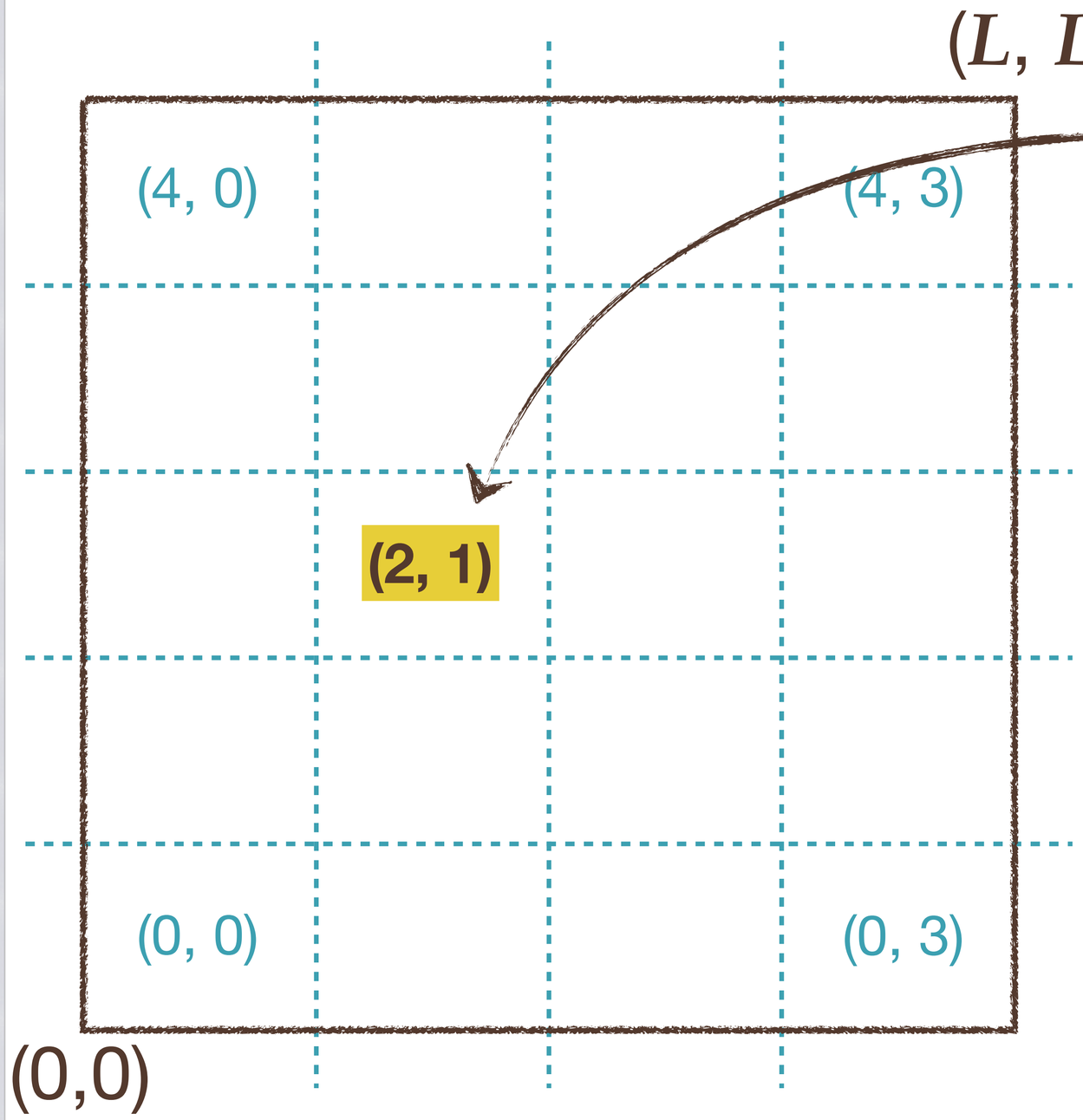

(L, L)
(4, 0)
(4, 3)
(0, 0)
(0, 3)
(0,0)



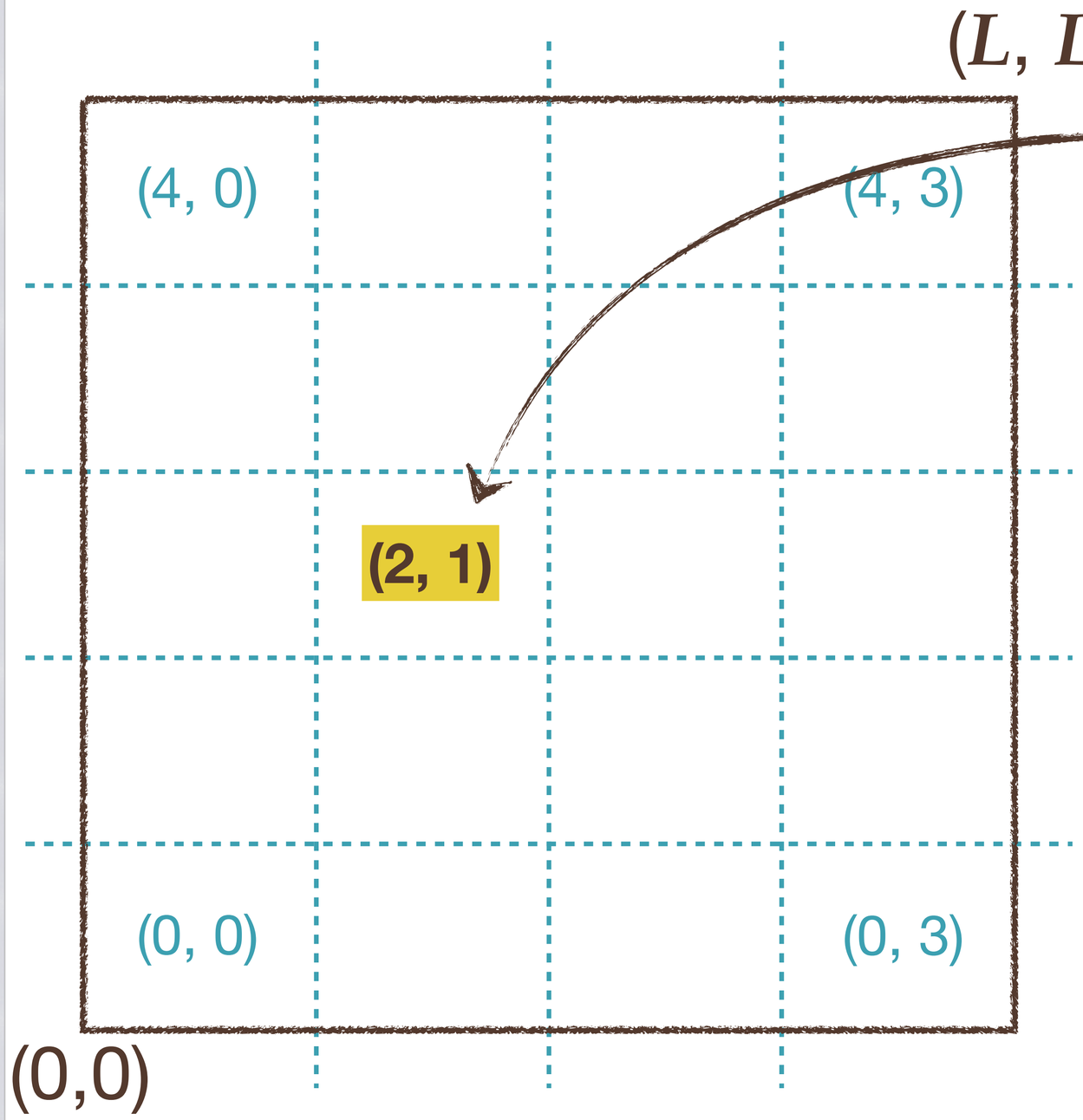

```
// Create 2-D communicator, ‘comm2d’,
// for a 4 x 3 logical process grid,
// from ‘comm’ = MPI_COMM_WORLD = {0, …, 11}.
//
int dims[2] = {4, 3};
int periods[2] = {0, 0};   // no wrap-around
int reorder = 1;           // 1 (true) or 0 (false)
MPI_Comm comm2d = MPI_COMM_NULL;

MPI_Cart_create (comm, 2, dims, periods,
                 reorder, &comm2d);
```



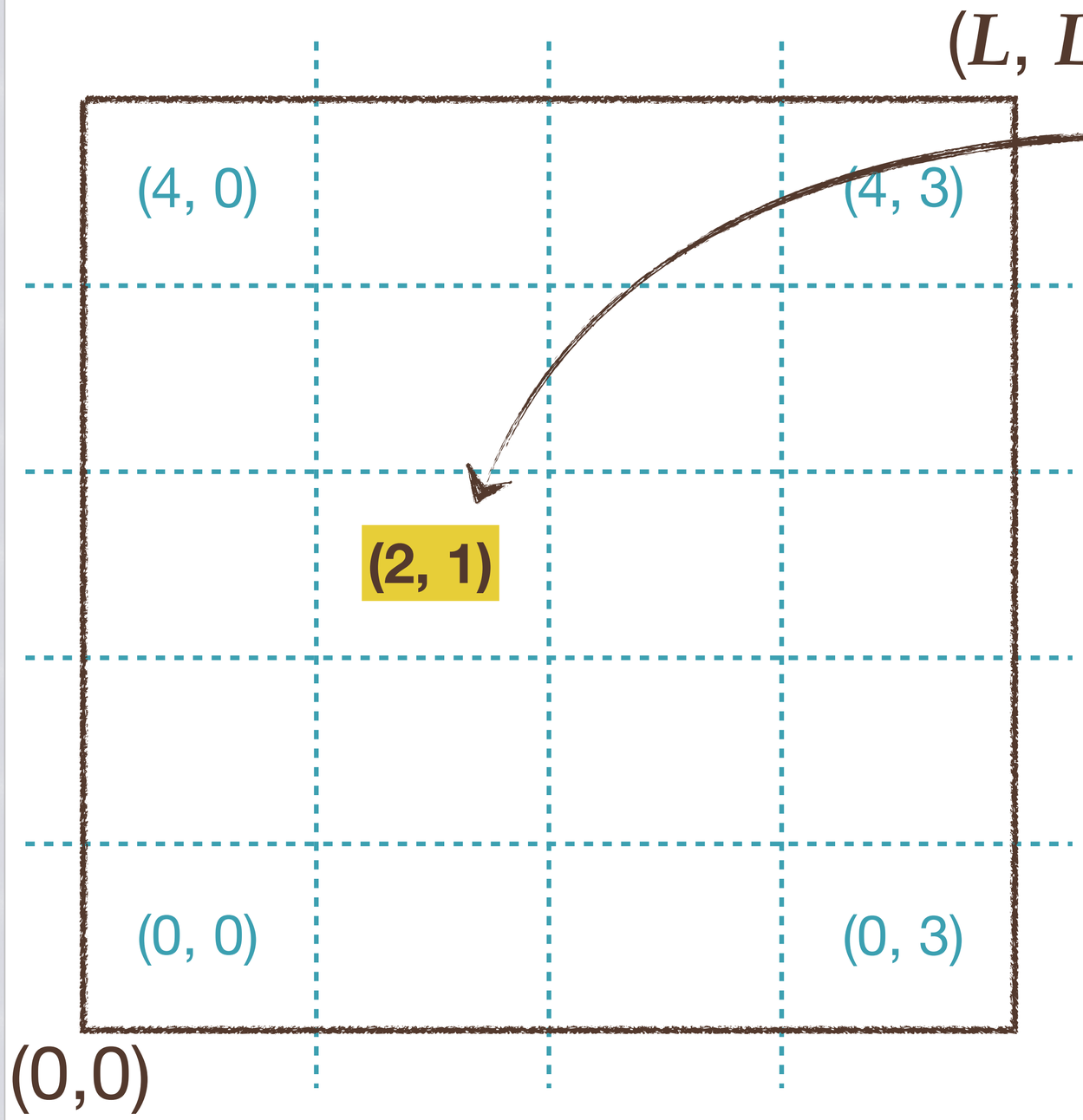

```
// Get my process coordinates from comm2d
//
int dims[2];
int periods[2];
int coords[2];

MPI_Cart_get (comm2d, 2, dims, periods, coords);

assert (dims[0] == 4  &&  dims[1] == 3);
assert (periods[0] == 0  &&  periods[1] == 0);
assert (coords[0] == 2  &&  coords[1] == 1);
```



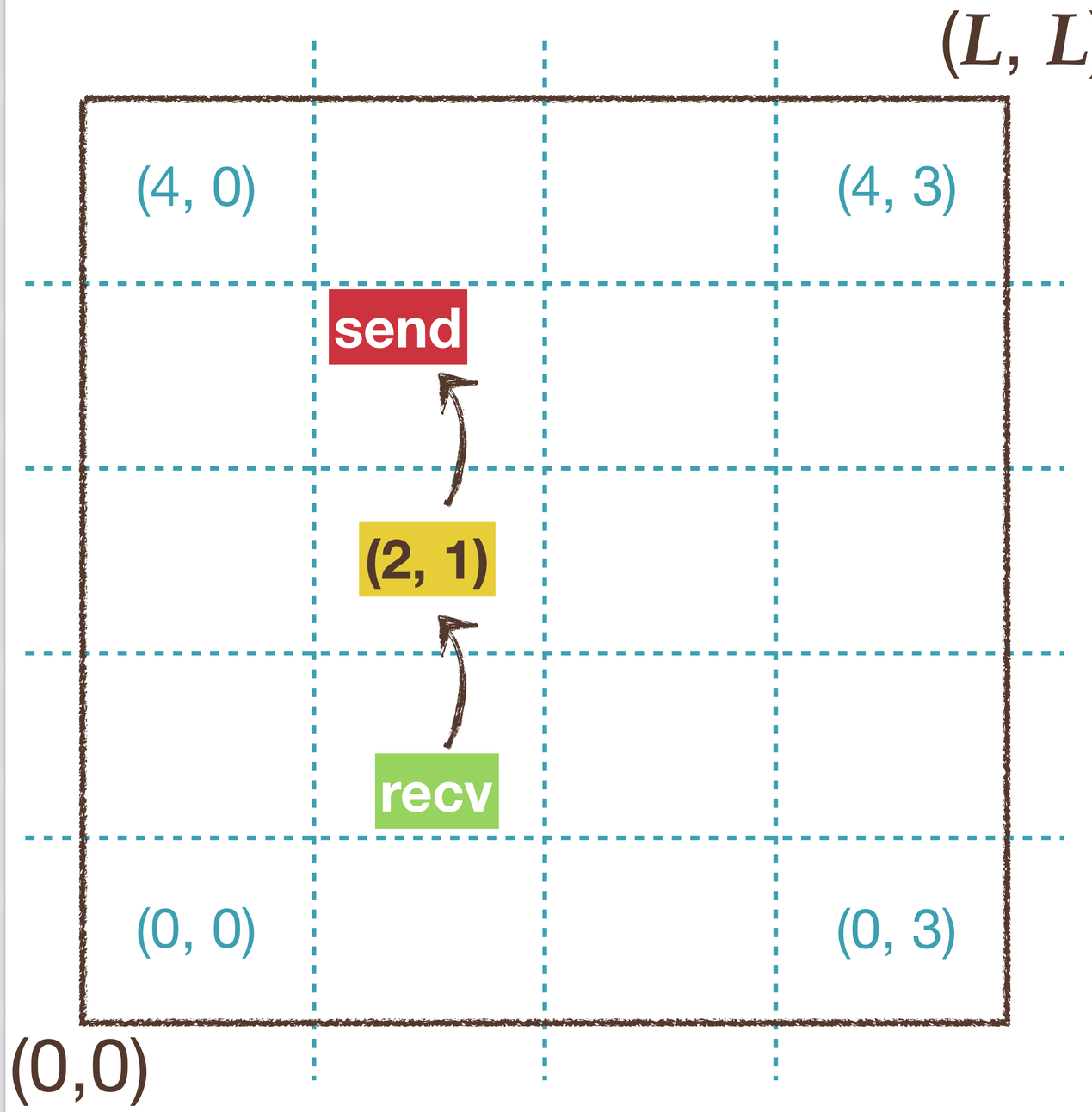

```
// Convert coords to global rank, and vice-versa

// Get rank by the usual method
int rank_usual;
MPI_Comm_rank (comm2d, &rank_usual);
```



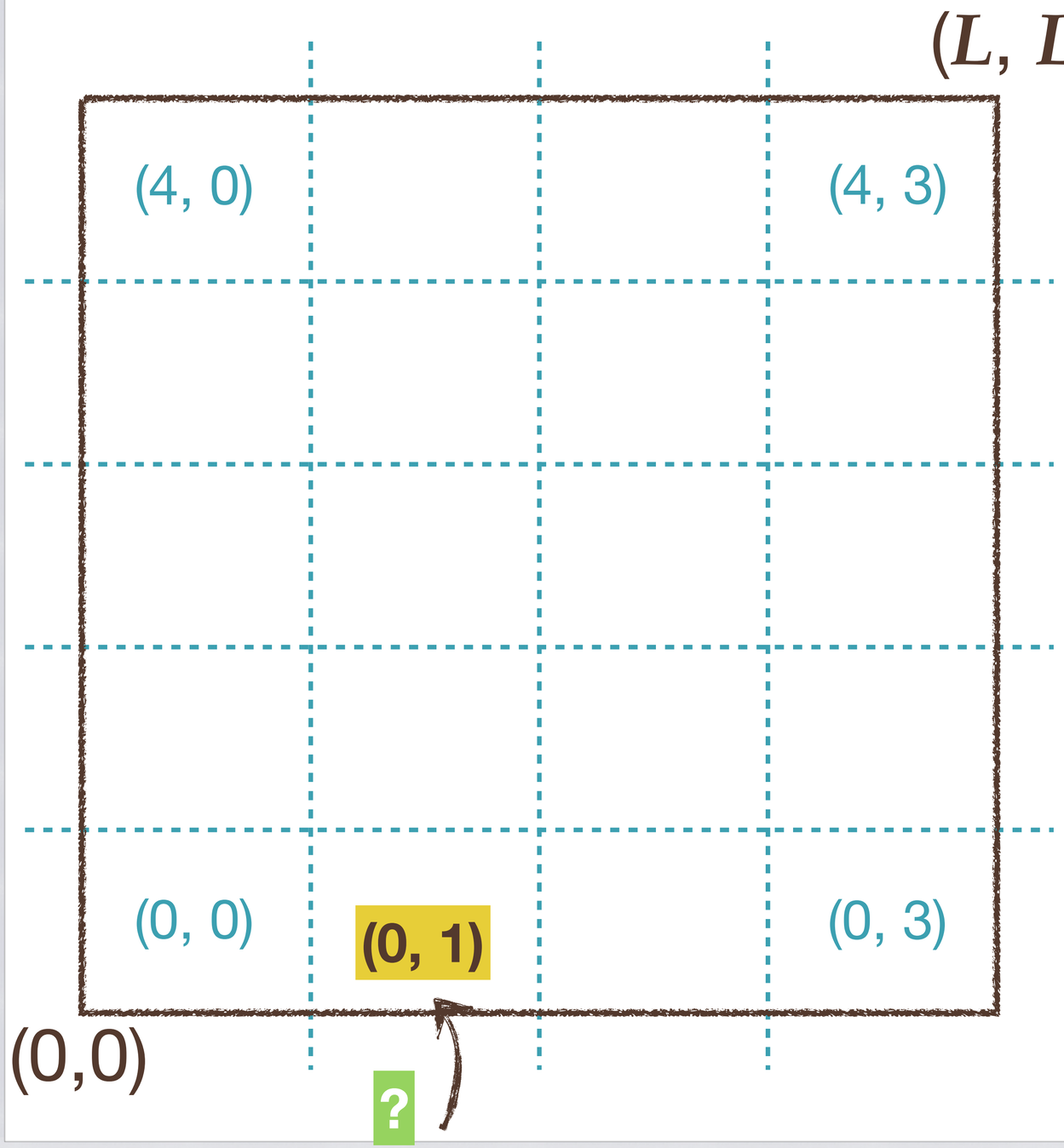

```
// Convert coords to global rank, and vice-versa

// Get rank by the usual method
int rank_usual;
MPI_Comm_rank (comm2d, &rank_usual);

// Convert any rank to process coordinates
int coords[2];
MPI_Cart_coords (comm2d, rank_usual, 2, coords);
assert (coords[0] == 2  &&  coords[1] == 1);
```




```
// Convert coords to global rank, and vice-versa

// Get rank by the usual method
int rank_usual;
MPI_Comm_rank (comm2d, &rank_usual);

// Convert any rank to process coordinates
int coords[2];
MPI_Cart_coords (comm2d, rank_usual, 2, coords);
assert (coords[0] == 2  &&  coords[1] == 1);

// Can also convert any coords back to a rank
int rank;
MPI_Cart_rank (comm2d, coords, &rank);
assert (rank == rank_usual);
```



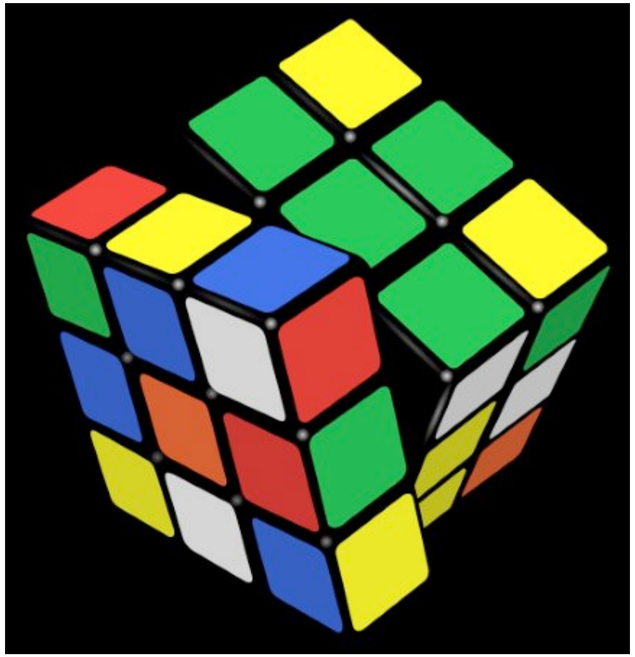

```
// Get ranks of (+/-1, 0) neighbors
int rank_src, rank_dest;
MPI_Cart_shift (comm2d, 0, +1, &rank_src, &rank_dest);

// Send "314". Should receive same value.
int msg_send = 314, msg_recv = -1;
const int MTAG = 100;  // some message tag
MPI_Status stat;
MPI_Sendrecv (&msg_send, 1, MPI_INT, rank_dest, MTAG,
                         &msg_recv, 1, MPI_INT, rank_src, 2, MTAG,
                         comm2d, &stat);

assert (coords[0] == 2  &&  coords[1] == 1);
assert (msg_recv == 314);
```



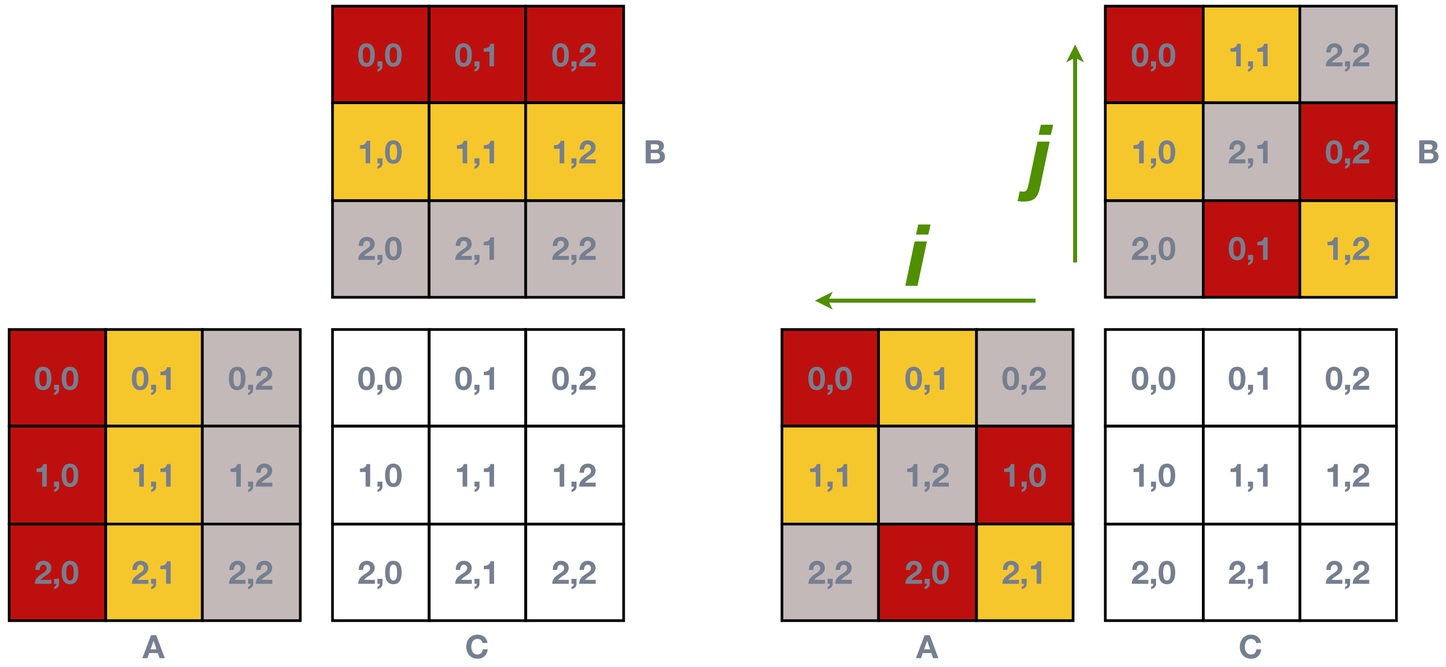

```
// Get ranks of (+/-1, 0) neighbors
int rank_src, rank_dest;
MPI_Cart_shift (comm2d, 0, +1, &rank_src, &rank_dest);

// Send "314". Should receive same value.
int msg_send = 314, msg_recv = -1;
const int MTAG = 100;  // some message tag
MPI_Status stat;
MPI_Sendrecv (&msg_send, 1, MPI_INT, rank_dest, MTAG,
                         &msg_recv, 1, MPI_INT, rank_src, 2, MTAG,
                         comm2d, &stat);

assert (coords[0] == 0  &&  coords[1] == 1);
assert (msg_recv == -1 && rank_src == MPI_PROC_NULL);
```

Cannon’s algorithm

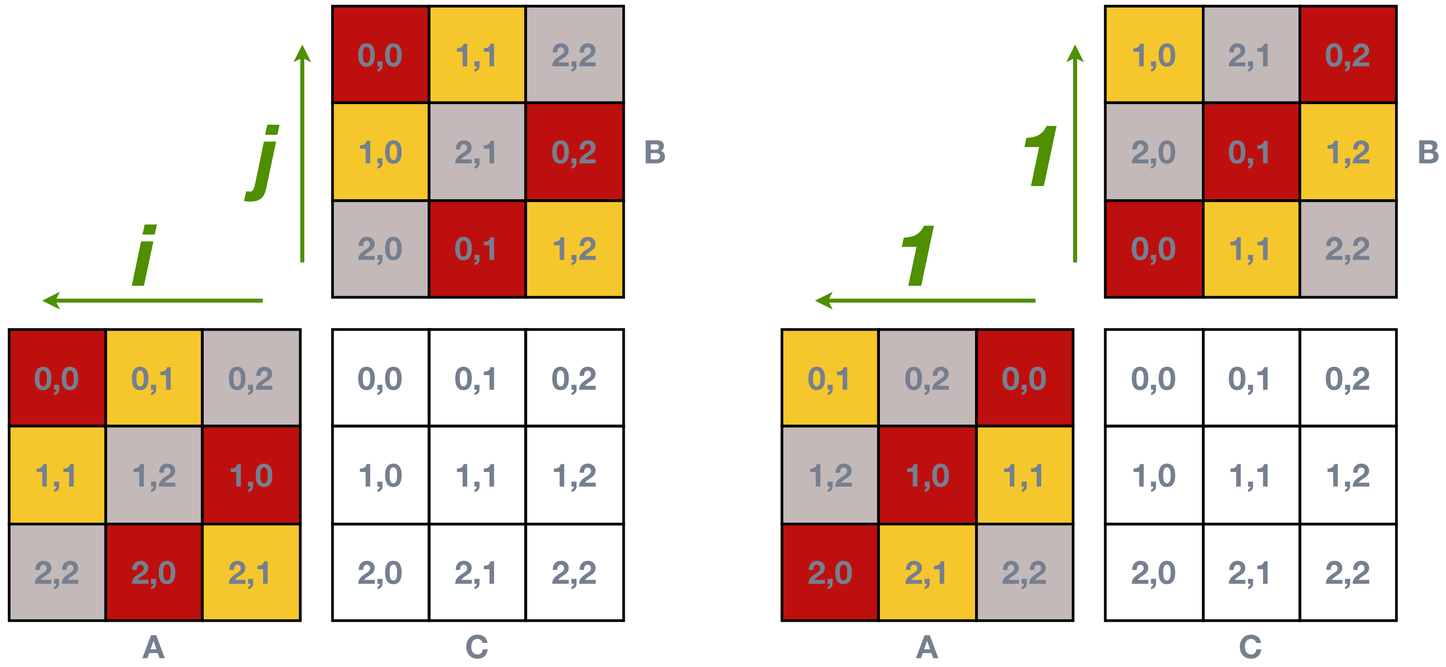
Cannon’s Algorithm



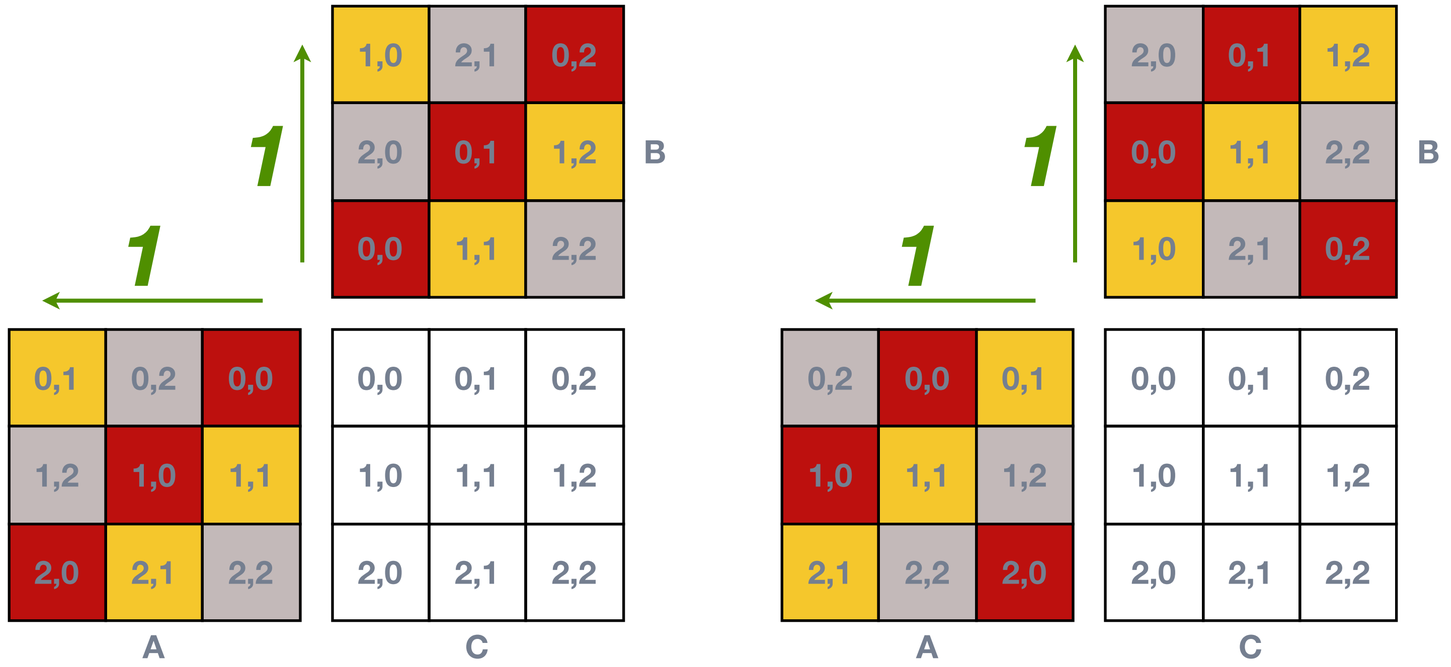
Cannon’s algorithm, initial step: “Skew” A & B



Cannon’s algorithm, iteration step: Local multiply + circular shift (1)

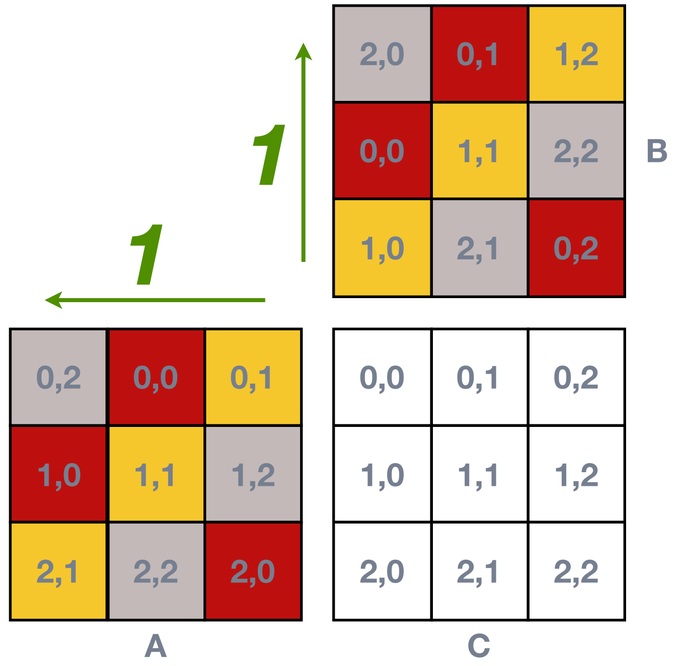


Cannon’s algorithm, iteration step: Local multiply + circular shift (2)



Cannon's algorithm

```
// Skew A & B
for i = 0 to s-1     // s = sqrt (p)
   left-circular-shift row i of A by i
for i = 0 to s-1
   up-circular-shift column i of B by i

// Multiply and shift
for k = 0 to s-1
   local-multiply
   left-circular-shift each row of A by 1
   up-circular-shift each column of B by 1
```



Analysis

*// **Skew A & B***
for i = 0 **to** s-1 *// s = sqrt (p)*
  left-circular-shift row i of A by i // ***cost = s*(α + n^2/p/β)***
for i = 0 **to** s-1
  up-circular-shift column i of B by i // ***cost = s*(α + n^2/p/β)***

*// **Multiply and shift***
for k = 0 **to** s-1
  local-multiply // ***cost = 2*(n/s)^3 = 2*n^3/$p^{3/2}$***
  left-circular-shift each row of A by 1 // ***cost = α + n^2/p/β***
  up-circular-shift each column of B by 1 // ***cost = α + n^2/p/β***

Time, speedup, and efficiency of Cannon's algorithm

$$T_{\text{comp}}(n;p) = \frac{2n^3}{p}$$

$$T_{\text{net}}(n;p) = 4\alpha\sqrt{p} + \frac{4n^2}{\beta\sqrt{p}}$$

Better latency term

$$S(n;p) = \frac{p}{1 + 2\alpha\left(\frac{\sqrt{p}}{n}\right)^3 + \frac{2}{\beta}\frac{\sqrt{p}}{n}}$$

"Perfect" speedup

n ~ sqrt(p)

$$E(n;p) = S(n;p)/p$$

Cannon's Algorithm: Pros and Cons

- Pros: Improved comp-to-comm ratio; optimal in bandwidth term
- Cons: Hard to generalize for …
  - Non-square $\boldsymbol{A}$ & $\boldsymbol{B}$, processor grid ($\boldsymbol{p}$)
  - Dimensions not divisible by sqrt ($\boldsymbol{p}$)
  - $\boldsymbol{A}$ & $\boldsymbol{B}$ not aligned in storage