

CSE 260

Lecture 14

Parallel Matrix Multiplication
Working with communicators

Announcements

Today's lecture

- Cannon's Parallel Matrix Multiplication Algorithm
- Communication Avoiding Matrix Multiplication
- Working with communicators

Matrix Multiplication

- An important core operation in many numerical algorithms
- Given two *conforming* matrices A and B , form the matrix product $A \times B$
 - A is $m \times n$
 - B is $n \times p$
- Operation count: $O(n^3)$ multiply-adds for an $n \times n$ square matrix
- See Demmel

www.cs.berkeley.edu/~demmel/cs267_Spr99/Lectures/Lect02.html

Serial Matrix Multiplication

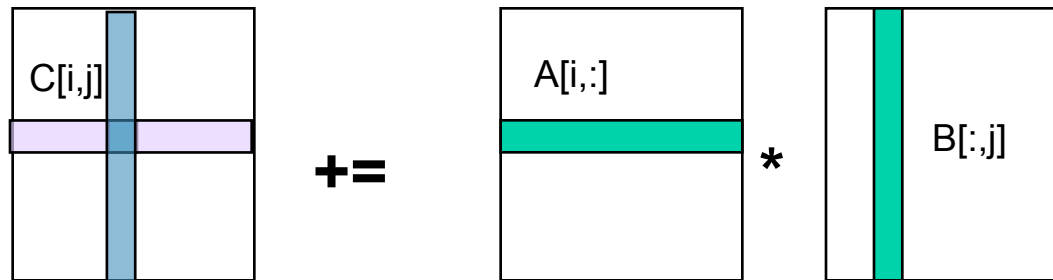
“ijk” variant

for i := 0 **to** n-1

for j := 0 **to** n-1

for k := 0 **to** n-1

$C[i,j] += A[i,k] * B[k,j]$



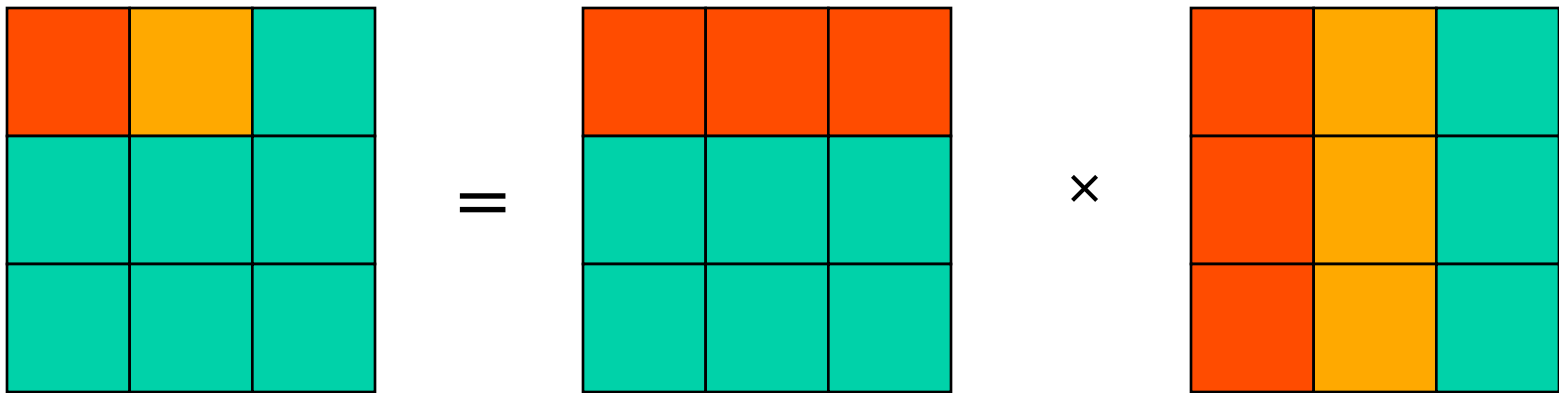
Parallel matrix multiplication

- Organize processors into rows and columns
 - ♦ Process rank is an ordered pair of integers
 - ♦ Assume p is a perfect square
- Each processor gets an $n/\sqrt{p} \times n/\sqrt{p}$ chunk of data
- Assume that we have an efficient serial matrix multiply (dgemm, sgemm)

$p(0,0)$	$p(0,1)$	$p(0,2)$
$p(1,0)$	$p(1,1)$	$p(1,2)$
$p(2,0)$	$p(2,1)$	$p(2,2)$

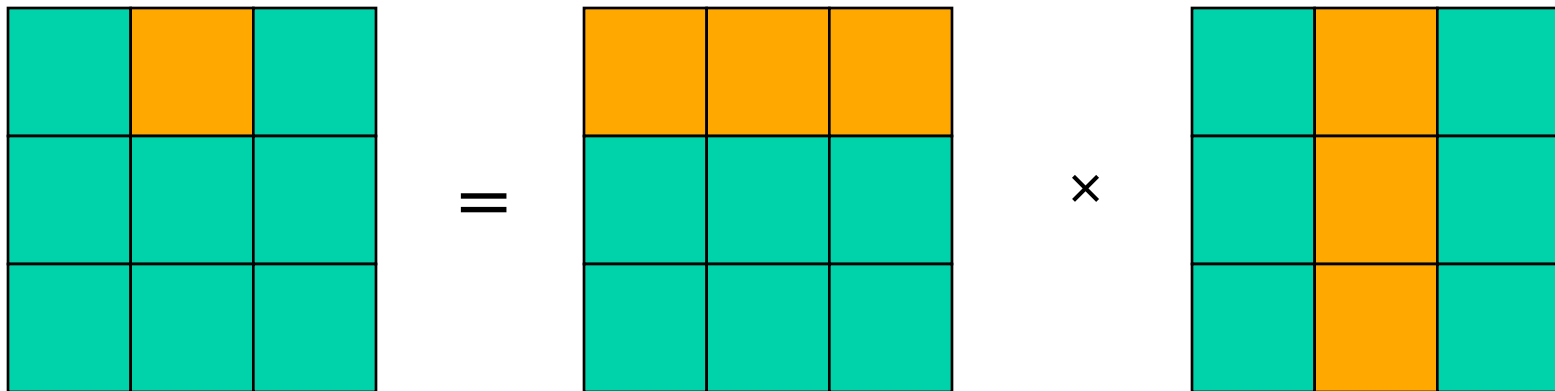
A simple parallel algorithm

- Apply the basic algorithm but treat each element $A[i,j]$ as a block rather than a single element
- Thus, $A[i,k] * B[k,j]$ is a matrix multiplication in $C[i, j] += A[i, k] * B[k, j]$



Cost

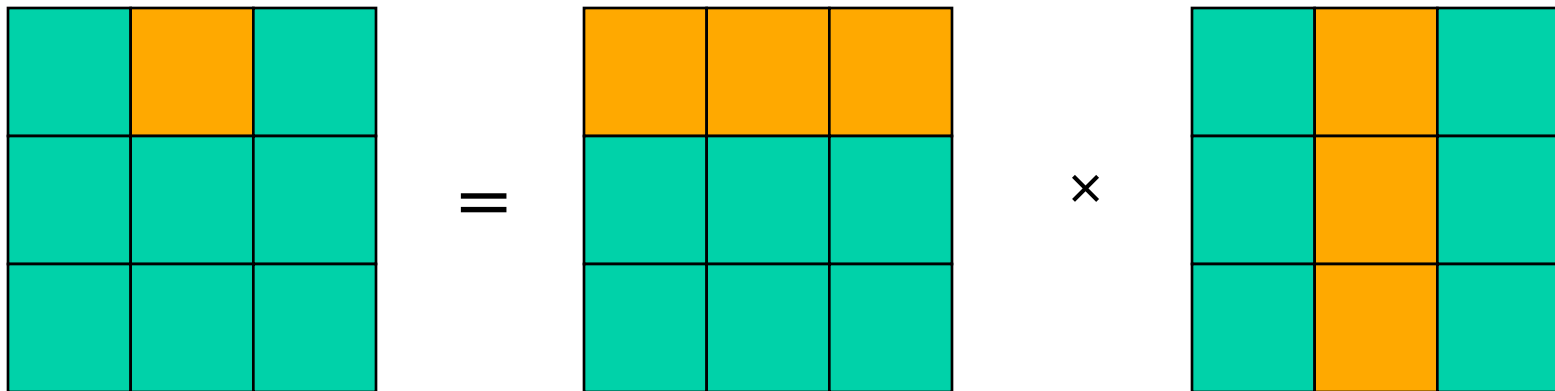
- Each processor performs n^3/p multiply-adds
- But a significant amount of communication is needed to collect a row and a column of data onto each processor
- Each processor broadcasts a chunk of data of size n^2/p within a row and a column of $\sqrt{p}$ processors
- Disruptive - distributes all the data in one big step
- High memory overhead
 - ♦ needs $2\sqrt{p}$ times the storage needed to hold A & B



Observation

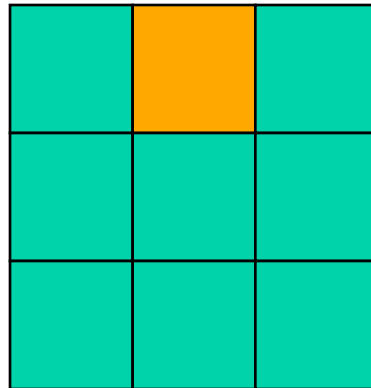
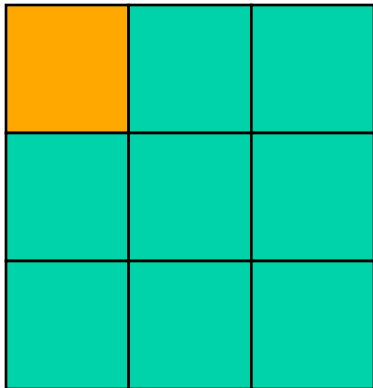
- In the broadcast algorithm each processor multiplies two skinny matrices of size $n^2/\sqrt{p}$
- But we can form the same product by computing $\sqrt{p}$ separate matrix multiplies involving $n^2/p \times n^2/p$ matrices and accumulating partial results
for $k := 0$ to $n - 1$

$$C[i, j] += A[i, k] * B[k, j];$$



A more efficient algorithm

- Take advantage of the organization of the processors into rows and columns
- Move data incrementally in $\sqrt{p}$ phases, using smaller pieces than with the broadcast approach
- Circulate each chunk of data among processors within a row or column
- In effect we are using a ring broadcast algorithm
- Buffering requirements are $O(1)$



Canon's algorithm

- Implements the strategy

- Consider block $C[1,2]$

$$C[1,2] = A[1,0]*B[0,2] + A[1,1]*B[1,2] + A[1,2]*B[2,2]$$

A(0,0)	A(0,1)	A(0,2)
A(1,0)	A(1,1)	A(1,2)
A(2,0)	A(2,1)	A(2,2)

B(0,0)	B(0,1)	B(0,2)
B(1,0)	B(1,1)	B(1,2)
B(2,0)	B(2,1)	B(2,2)

Canon's algorithm

- Move data incrementally in $\sqrt{p}$ phases
- Circulate each chunk of data among processors within a row or column
- In effect we are using a ring broadcast algorithm
- Consider iteration $i=1, j=2$:

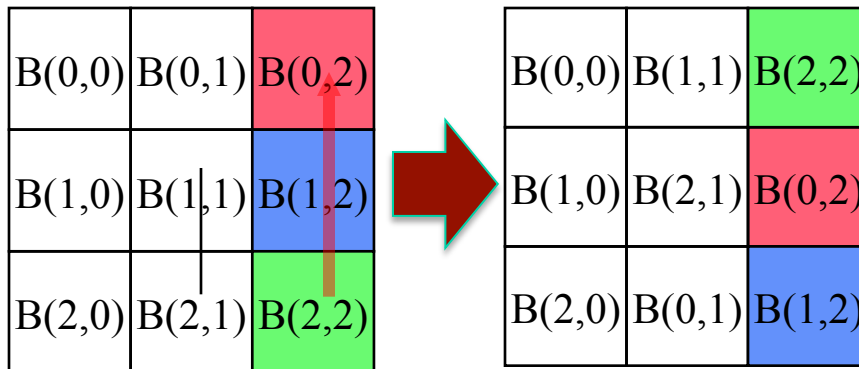
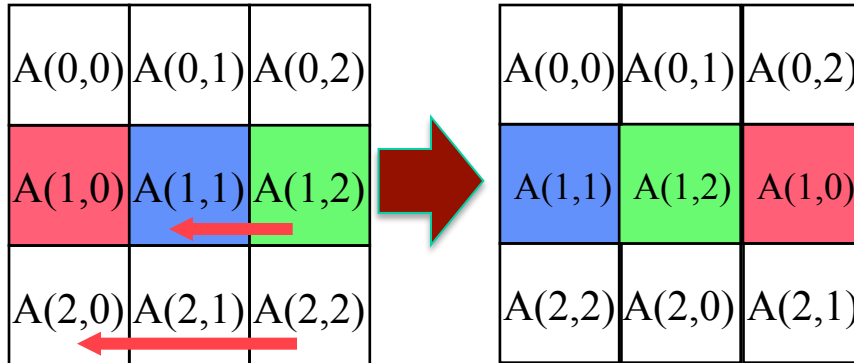
$$C[1,2] = A[1,0]*B[0,2] + A[1,1]*B[1,2] + A[1,2]*B[2,2]$$

A(0,0)	A(0,1)	A(0,2)
A(1,0)	A(1,1)	A(1,2)
A(2,0)	A(2,1)	A(2,2)

B(0,0)	B(0,1)	B(0,2)
B(1,0)	B(1,1)	B(1,2)
B(2,0)	B(2,1)	B(2,2)

Skewing the matrices

$$C[1,2] = A[1,0]*B[0,2] + A[1,1]*B[1,2] + A[1,2]*B[2,2]$$

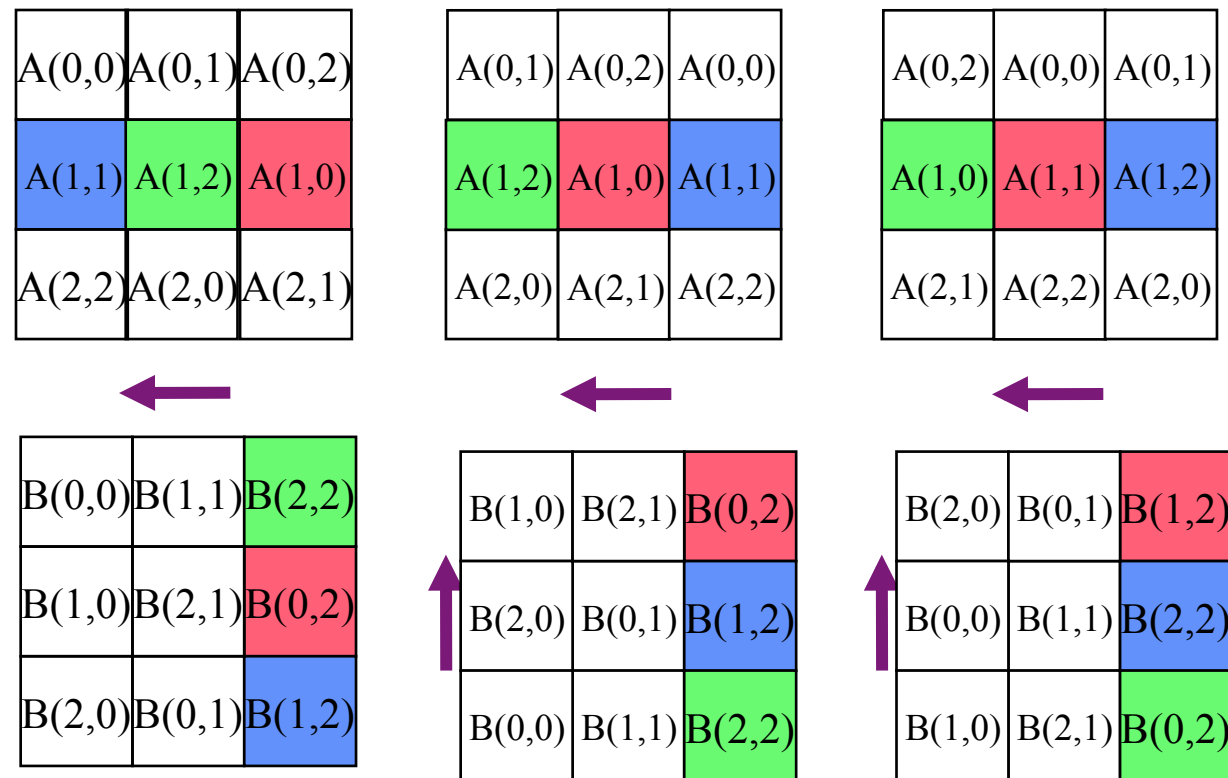


- Before we start, we *preskew* the matrices so everything lines up
- Shift row i by i columns to the left using sends and receives
 - ♦ Do the same for each column
 - ♦ Communication wraps around
- Ensures that each partial product is computed on the same processor that owns $C[I,J]$, using only shifts

Shift and multiply

$$C[1,2] = A[1,0]*B[0,2] + A[1,1]*B[1,2] + A[1,2]*B[2,2]$$

- $\sqrt{p}$ steps
- Circularly shift Rows 1 column to the left, columns 1 row up
- Each processor forms the partial product of local A & B and adds into the accumulated sum in C



Cost of Cannon's Algorithm

forall $i=0$ to $\sqrt{p}-1$

 CShift-left $A[i; :]$ by i

// $T = \alpha + \beta n^2/p$

forall $j=0$ to $\sqrt{p}-1$

 Cshift-up $B[:, j]$ by j

// $T = \alpha + \beta n^2/p$

for $k=0$ to $\sqrt{p}-1$

forall $i=0$ to $\sqrt{p}-1$ **and** $j=0$ to $\sqrt{p}-1$

$C[i,j] += A[i,j]*B[i,j]$

// $T = 2*n^3/p^{3/2}$

 CShift-left $A[i; :]$ by 1

// $T = \alpha + \beta n^2/p$

 Cshift-up $B[:, j]$ by 1

// $T = \alpha + \beta n^2/p$

end forall

end for

$$T_p = 2n^3/p + 2(\alpha(1+\sqrt{p}) + \beta n^2/(1+\sqrt{p})/p)$$

$$E_p = T_1/(pT_p) = (1 + \alpha p^{3/2}/n^3 + \beta \sqrt{p}/n)^{-1}$$

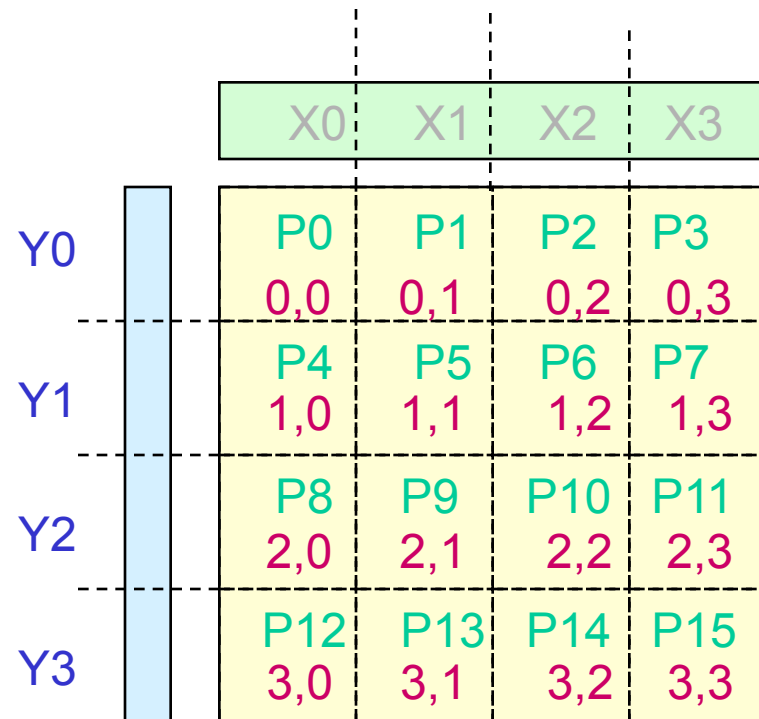
$$\approx (1 + O(\sqrt{p}/n))^{-1}$$

$$E_p \rightarrow 1 \text{ as } (n/\sqrt{p}) \text{ grows [sqrt of data / processor]}$$

Implementation

Communication domains

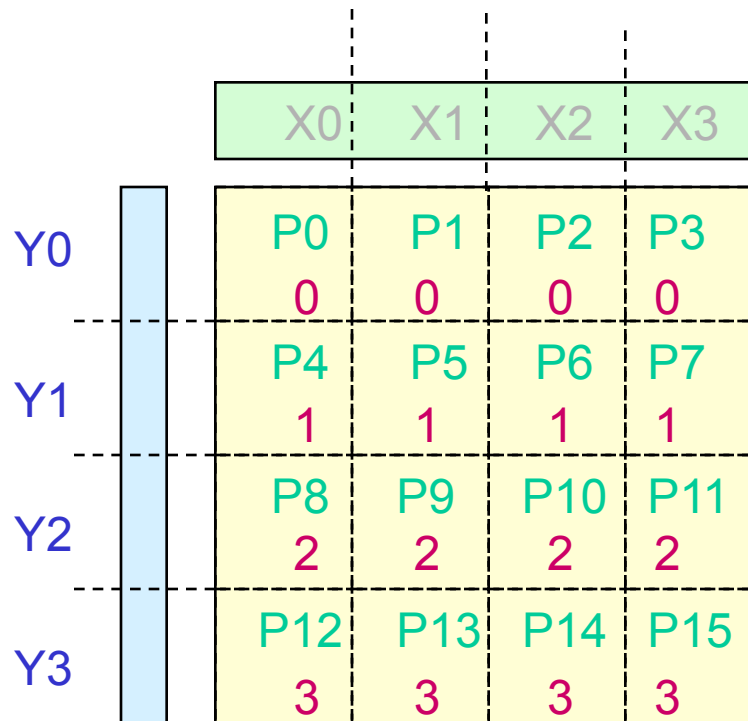
- Cannon's algorithm shifts data along rows and columns of processors
- MPI provides communicators for grouping processors, reflecting the communication structure of the algorithm
- An MPI communicator is a name space, a subset of processes that communicate
- Messages remain within their communicator
- A process may be a member of more than one communicator



Establishing row communicators

- Create a communicator for each row and column
- By Row

$$\text{key} = \text{myRank} \div \sqrt{P}$$



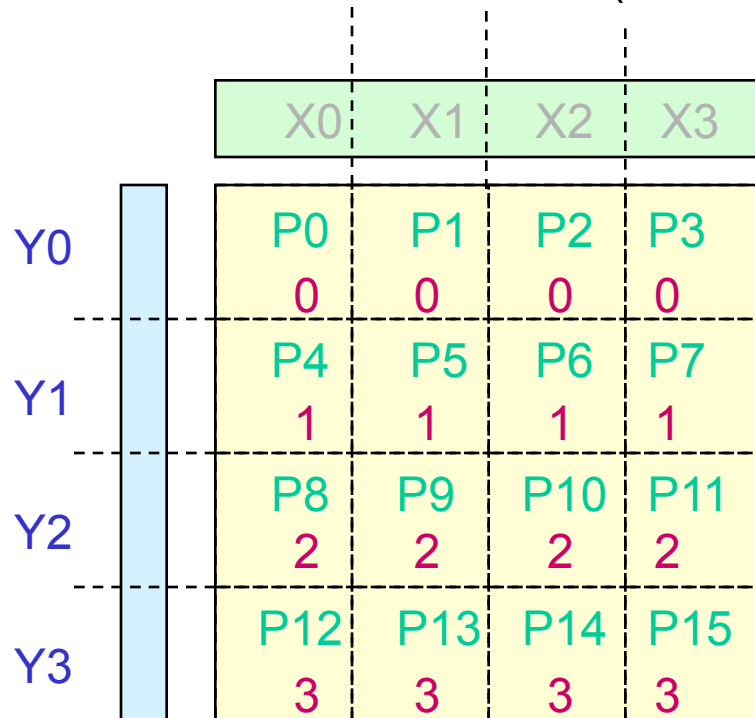
Creating the communicators

- Create a row communicator $\text{key} = \text{myRank} \div \sqrt{P}$

```
MPI_Comm rowComm;
```

```
MPI_Comm_split( MPI_COMM_WORLD,  
               myRank /  $\sqrt{P}$ , myRank, &rowComm);
```

```
MPI_Comm_rank(rowComm,&myRow);
```

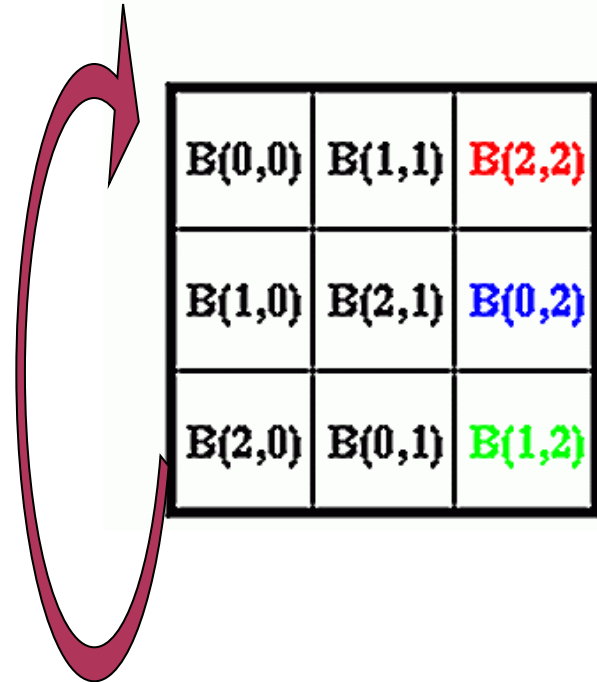


- Each process obtains a new communicator
- Each process' rank relative to the new communicator
- Rank applies to the respective communicator only
- Ordered according to **myRank**

Circular shift

- Communication with columns (and rows)

$p(0,0)$	$p(0,1)$	$p(0,2)$
$p(1,0)$	$p(1,1)$	$p(1,2)$
$p(2,0)$	$p(2,1)$	$p(2,2)$



More on Comm_split

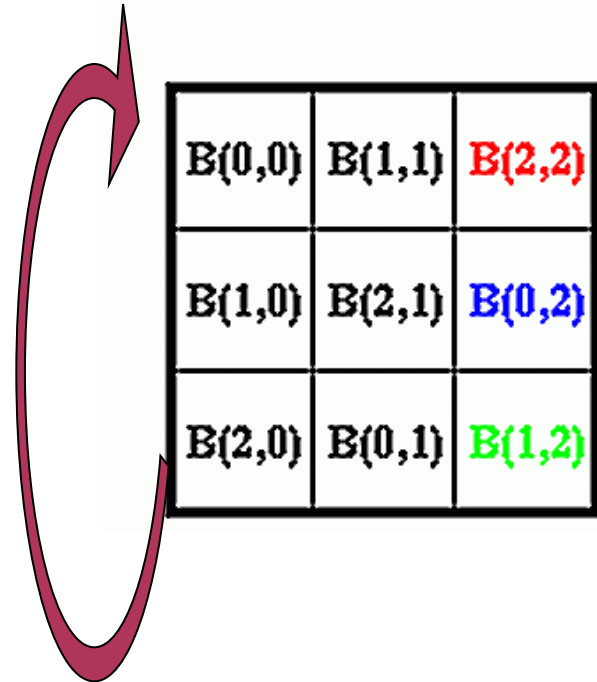
**MPI_Comm_split(MPI_Comm comm, int splitKey,
int rankKey, MPI_Comm* newComm)**

- Ranks assigned arbitrarily among processes sharing the same **rankKey** value
- May exclude a process by passing the constant **MPI_UNDEFINED** as the **splitKey**
- Return a special **MPI_COMM_NULL** communicator
- If a process is a member of several communicators, it will have a rank within each one

Circular shift

- Communication with columns (and rows)

$p(0,0)$	$p(0,1)$	$p(0,2)$
$p(1,0)$	$p(1,1)$	$p(1,2)$
$p(2,0)$	$p(2,1)$	$p(2,2)$

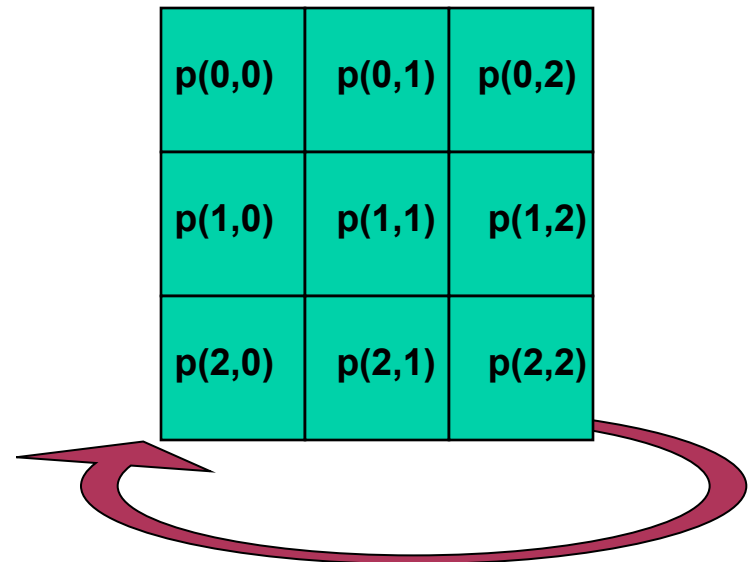


Circular shift

- Communication with columns (and rows)

```
MPI_Comm_rank(rowComm,&myidRing);  
MPI_Comm_size(rowComm,&nodesRing);  
int next = (myidRng + 1 ) % nodesRing;  
MPI_Send(&X,1,MPI_INT,next,0, rowComm);  
MPI_Recv(&XR,1,MPI_INT,  
        MPI_ANY_SOURCE,  
        0, rowComm, &status);
```

- Processes 0, 1, 2 in one communicator because they share the same key value (0)
- Processes 3, 4, 5 are in another (key=1), and so on



Today's lecture

- Cannon's Matrix Multiplication Algorithm
- 2.5D “Communication avoiding”
- SUMMA

Motivation

- Relative to arithmetic speeds, communication is becoming more costly with time
- Communication can be data motion on or off-chip, across address spaces
- We seek algorithms that increase the amount of work (flops) relative to the amount of data they move

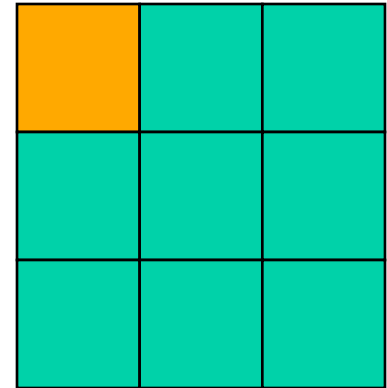
Communication lower bounds on Matrix Multiplication

- Assume we are using an $O(n^3)$ algorithm
- Let M = Size fast memory (cache/local memory)
- Sequential case: # slow memory references
 $\Omega(n^3 / \sqrt{M})$ [Hong and Kung '81]
- Parallel, p = # processors,
 μ = Amount of memory needed to store matrices
 - ♦ Refs to remote memory
 $\Omega(n^3 / (p\sqrt{\mu}))$ [Irony, Tiskin, Toledo, '04]
 - ♦ If $\mu = 3n^2/p$ (one copy of A, B, C) $\Rightarrow$
 lower bound = $\Omega(n^2 / \sqrt{p})$ words
 - ♦ Achieved by Cannon's algorithm ("2D algorithm")
 - ♦ $T_p = 2n^3/p + 4\sqrt{p}(\alpha + \beta n^2/p)$

Canon's Algorithm - optimality

- General result
 - ♦ If each processor has M words of local memory ...
 - ♦ ... at least 1 processor must transmit $\Omega (\# \text{ flops} / M^{1/2})$ words of data
- If local memory $M = O(n^2/p)$...
 - ♦ at least 1 processor performs $f \geq n^3/p$ flops
 - ♦ ... lower bound on number of words transmitted by at least 1 processor

$$\begin{aligned}\Omega ((n^3/p) / \sqrt{(n^2/p)}) &= \Omega ((n^3/p) / \sqrt{M}) \\ &= \Omega (n^2 / \sqrt{p})\end{aligned}$$

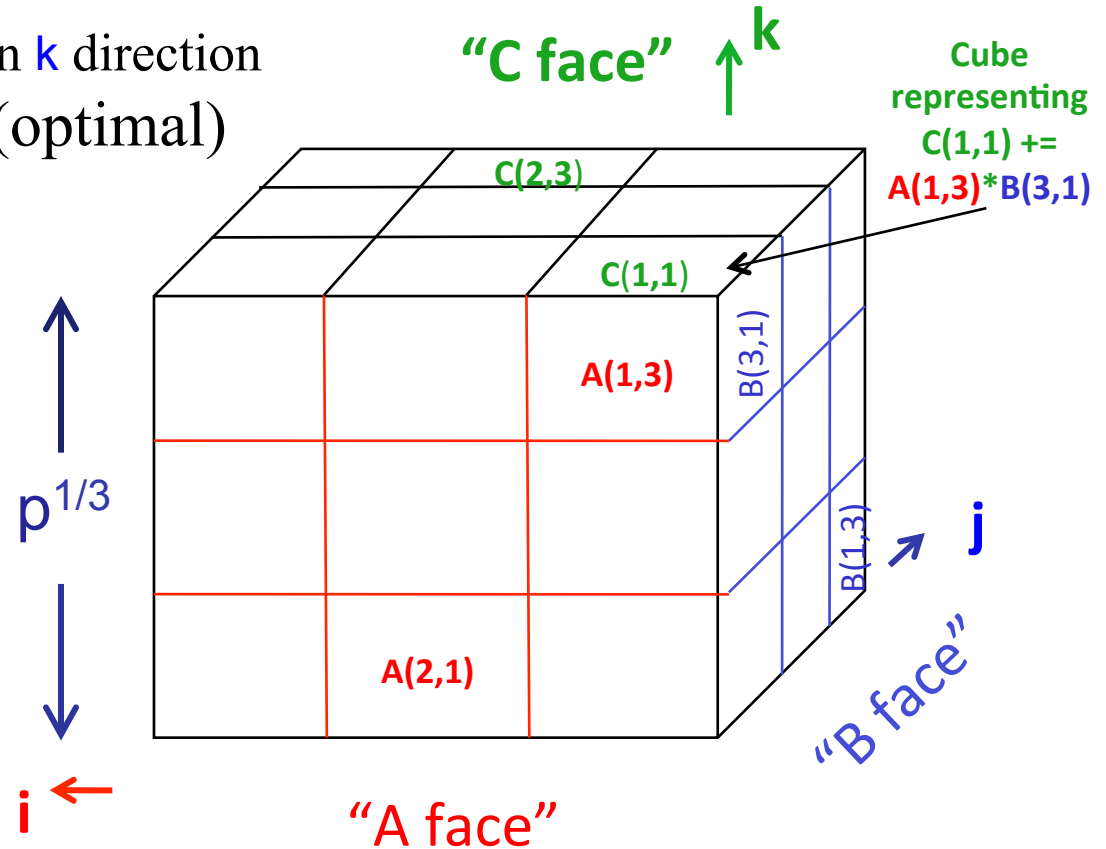
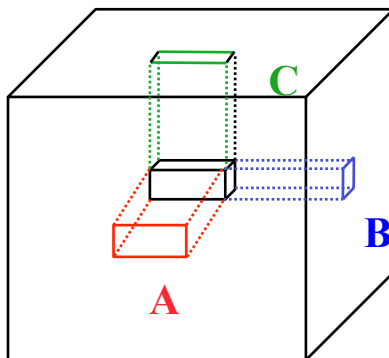


New communication lower bounds – direct linear algebra [Ballard & Demmel '11]

- Let M = amount of fast memory per processor
- Lower bounds
 - ♦ # words moved by at least 1 processor
 $\Omega(\# \text{ flops} / M^{1/2})$
 - ♦ # messages sent by at least 1 processor
 $\Omega(\# \text{ flops} / M^{3/2})$
- Holds not only for Matrix Multiply but many other “direct” algorithms in linear algebra, sparse matrices, some graph theoretic algorithms
- Identify 3 values of M
 - ♦ 2D (Cannon’s algorithm)
 - ♦ 3D (Johnson’s algorithm)
 - ♦ 2.5D (Ballard and Demmel)

Johnson's 3D Algorithm

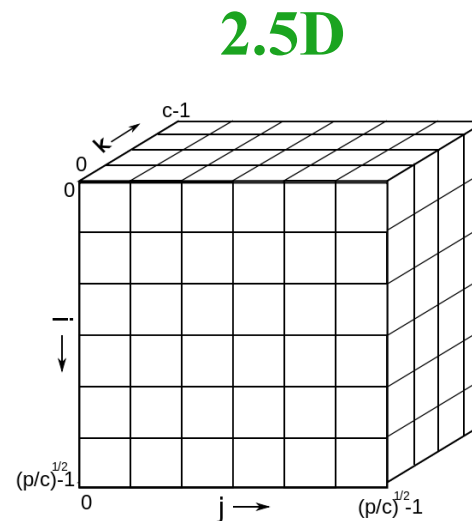
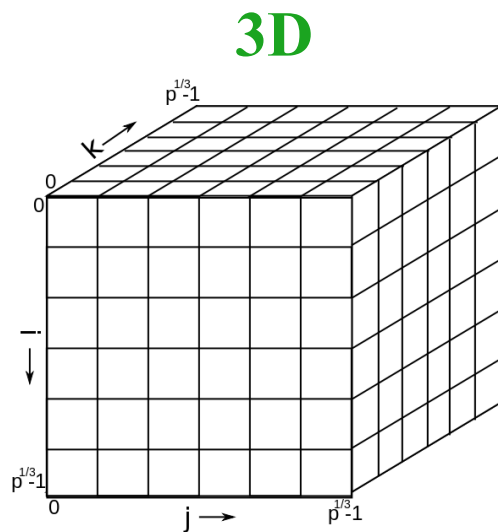
- 3D processor grid: $p^{1/3} \times p^{1/3} \times p^{1/3}$
 - Bcast A (B) in j (i) direction ($p^{1/3}$ redundant copies)
 - Local multiplications
 - Accumulate (Reduce) in k direction
- Communication costs (optimal)
 - Volume = $O(n^2/p^{2/3})$
 - Messages = $O(\log(p))$
- Assumes space for $p^{1/3}$ redundant copies
- Trade memory for communication



Source: Edgar Solomonik

2.5D Algorithm

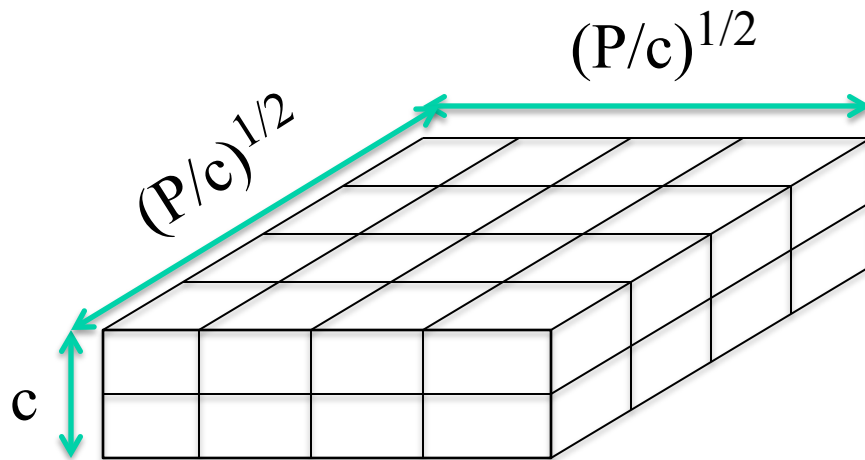
- What if we have space for only $1 \leq c \leq p^{1/3}$ copies ?
- $M = \Omega(c \cdot n^2/p)$
- Communication costs : lower bounds
 - ♦ Volume = $\Omega(n^2 / (cp)^{1/2})$; Set $M = c \cdot n^2/p$ in $\Omega(\# \text{ flops} / M^{1/2})$
 - ♦ Messages = $\Omega(p^{1/2} / c^{3/2})$; Set $M = c \cdot n^2/p$ in $\Omega(\# \text{ flops} / M^{3/2})$
- 2.5D algorithm “interpolates” between 2D & 3D algorithms



Source: Edgar Solomonik

2.5D Algorithm

- Assume can fit cn^2/P data per processor, $c > 1$
- Processors form $(P/c)^{1/2} \times (P/c)^{1/2} \times c$ grid

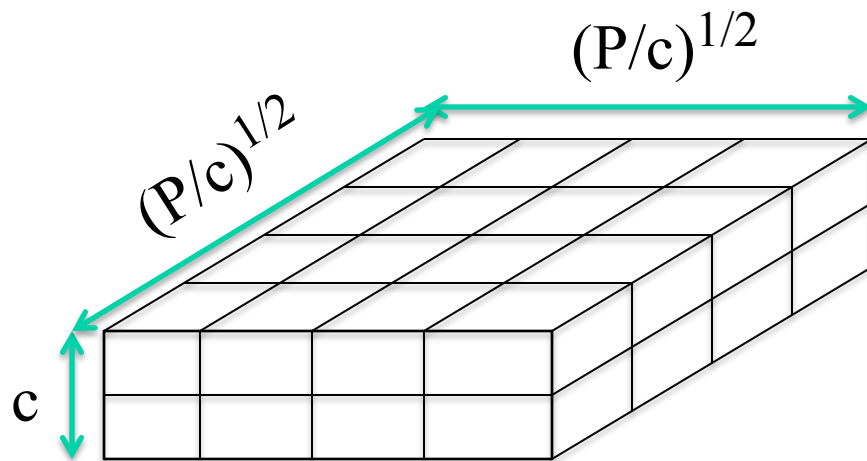


Example: $P = 32$, $c = 2$

Source Jim Demmel

2.5D Algorithm

- Assume can fit cn^2/P data per processor, $c > 1$
- Processors form $(P/c)^{1/2} \times (P/c)^{1/2} \times c$ grid



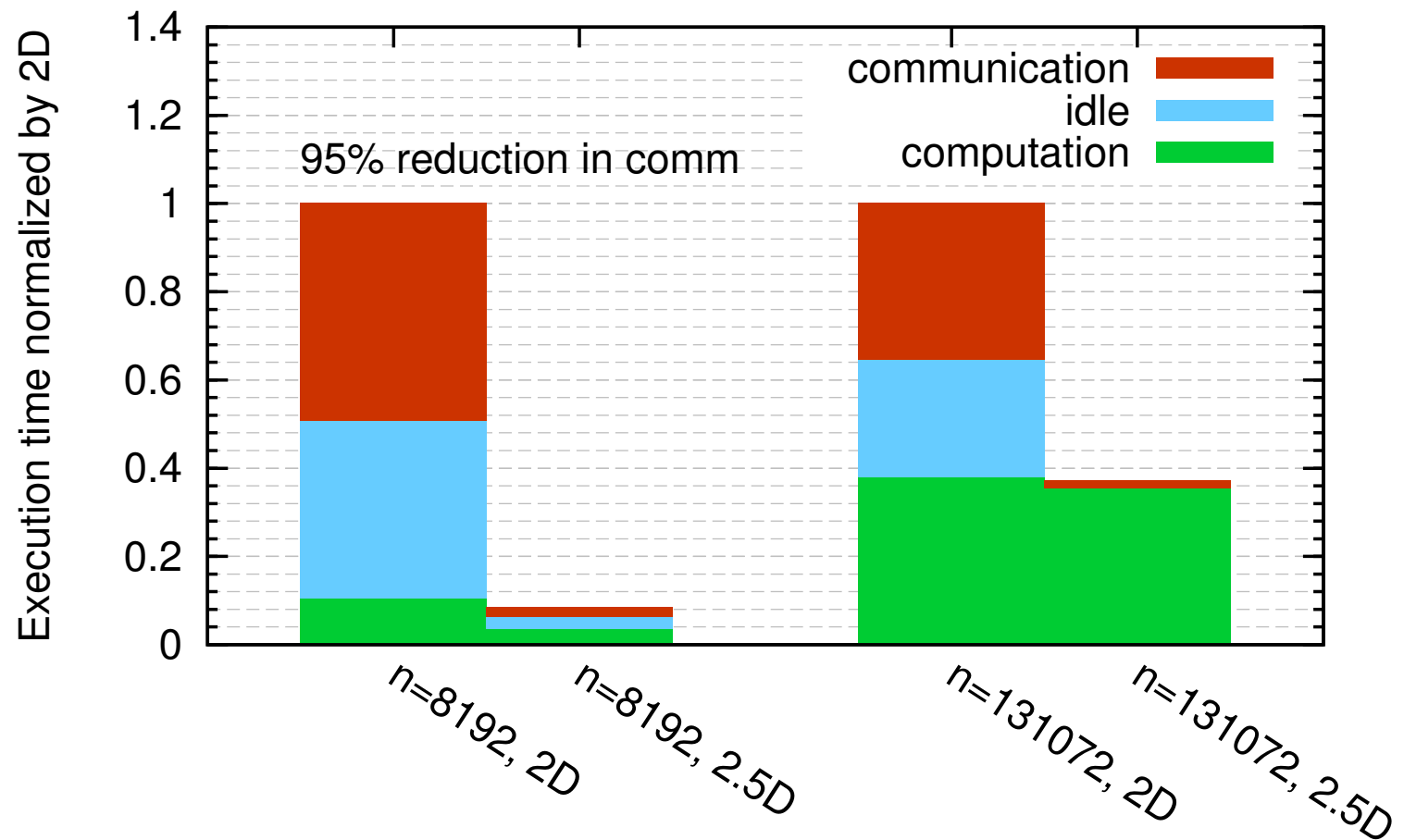
Initially $P(i,j,0)$ owns $A(i,j)$ & $B(i,j)$
each of size $n(c/P)^{1/2} \times n(c/P)^{1/2}$

- (1) $P(i,j,0)$ broadcasts $A(i,j)$ and $B(i,j)$ to $P(i,j,k)$
- (2) Processors at level k perform $1/c$ -th of SUMMA,
i.e. $1/c$ -th of $\sum_m A(i,m) * B(m,j)$
- (3) Sum-reduce partial sums $\sum_m A(i,m) * B(m,j)$ along k -axis so that $P(i,j,0)$ owns $C(i,j)$

Performance on Blue Gene P

C=16

Matrix multiplication on 16,384 nodes of BG/P



2.5D Algorithm

- Interpolate between 2D (Cannon) and 3D
 - c copies of A & B
 - Perform $p^{1/2}/c^{3/2}$ Cannon steps on each copy of A&B
 - Sum contributions to C over all c layers
- Communication costs (not quite **optimal**, but not far off)

- Volume:

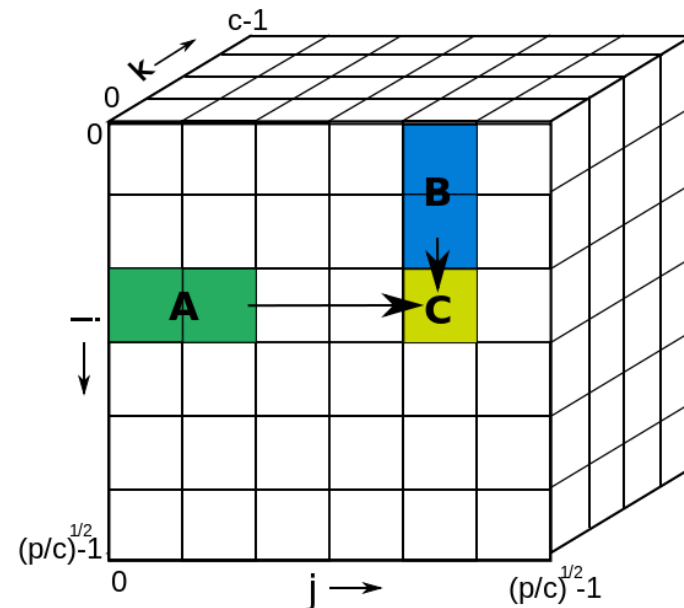
$$O(n^2 / (cp)^{1/2})$$

$$[\Omega(n^2 / (cp)^{1/2})]$$

- Messages:

$$O(p^{1/2} / c^{3/2} + \log(c))$$

$$[\Omega(p^{1/2} / c^{3/2})]$$



Source: Edgar Solomonik

Today's lecture

- Cannon's Matrix Multiplication Algorithm
- 2.5D "Communication avoiding"
- SUMMA

Outer product formulation of matrix multiply

- Limitations of Cannon's Algorithm
 - ♦ P must be a perfect square
 - ♦ A and B must be square, and evenly divisible by $\sqrt{p}$
- Interoperation with applications and other libraries difficult or expensive
- The SUMMA algorithm offers a practical alternative
 - ♦ Uses a shift algorithm to broadcast
 - ♦ A variant used in SCALAPACK by Van de Geijn and Watts [1997]

Formulation

- The matrices may be non-square (kij formulation)

for k := 0 to n3-1

 for i := 0 to n1-1

 for j := 0 to n2-1

 C[i,j] += A[i,k] * B[k,j]

C[i,:] += A[i,k] * B[k,:]

- The two innermost loop nests compute

n3 outer products

for k := 0 to n3-1

 C[:,:] += A[:,k] • B[k,:]

where • is outer product

Outer product

- Recall that when we multiply
an $m \times n$ matrix by an $n \times p$ matrix...
we get an $m \times p$ matrix
- Outer product of *column vector* a^T and *vector* b = matrix C
an $m \times 1$ times a $1 \times n$

$$a[1,3] \cdot x[3,1]$$

$$(a,b,c) * (x,y,z)^T \equiv \begin{pmatrix} ax & ay & az \\ bx & by & bz \\ cx & cy & cz \end{pmatrix}$$

*	1	2	3
10	11	20	30
20	20	40	60
30	30	60	90

Multiplication table with rows formed by $a[:]$ and the columns by $b[:]$

- The SUMMA algorithm computes n partial outer products:

```
for k := 0 to n-1
    C[:,:] += A[:,k] • B[k,:]
```

Outer Product Formulation

- The new algorithm computes **n** partial outer products:

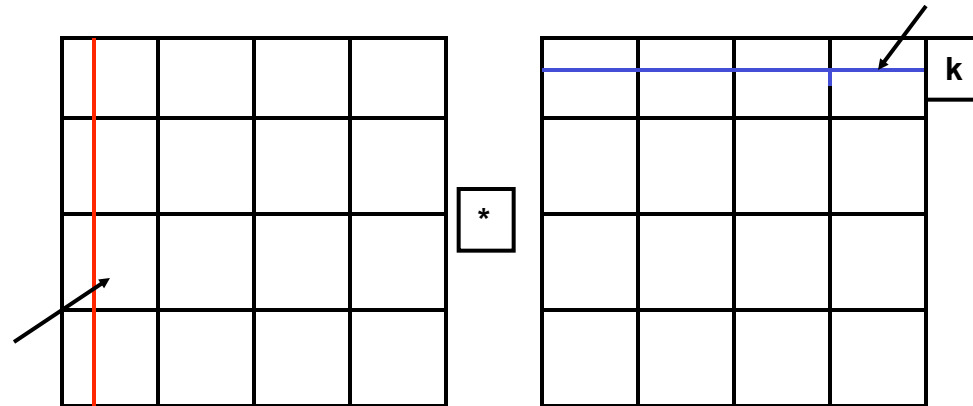
for $k := 0$ to $n-1$
 $C[:, :] += A[:, k] \cdot B[k, :]$

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

“Inner product” formulation:
for $i := 0$ to $n-1$, $j := 0$ to $n-1$
 $C[i, j] += A[i, :] * B[:, j]$

Serial algorithm

- Each row k of B contributes to the n partial n partial outer products
for $k := 0$ to $n-1$
 $C[:, :] += A[:, k] \cdot B[k, :]$

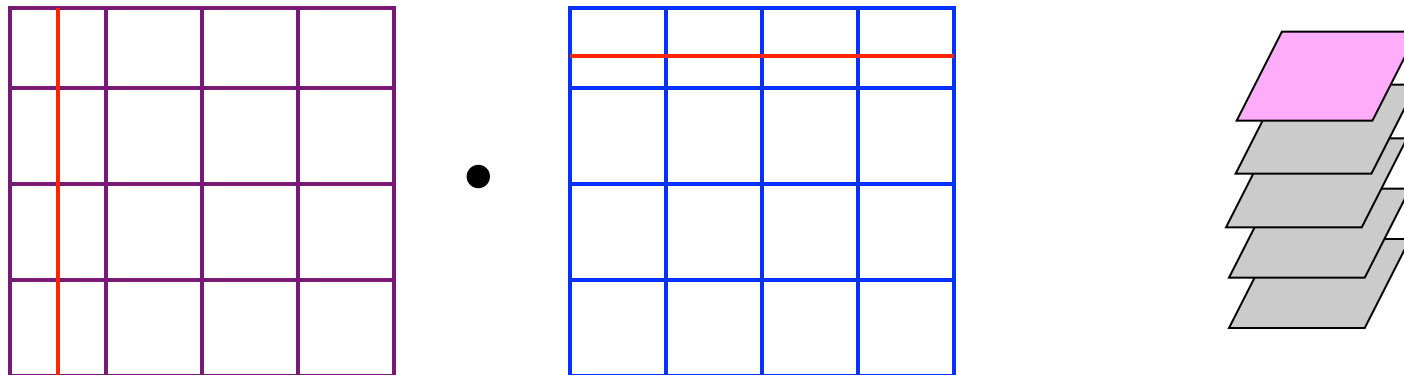


Animation of SUMMA

- Compute the sum of **n** outer products
- Each row & column **(k)** of A & B generates a single outer product
 - ♦ Column vector **A[:,k]** ($n \times 1$) & a vector **B[k,:]** ($1 \times n$)

for k := 0 to n-1

C[:,:] += A[:,k] • B[k,:]

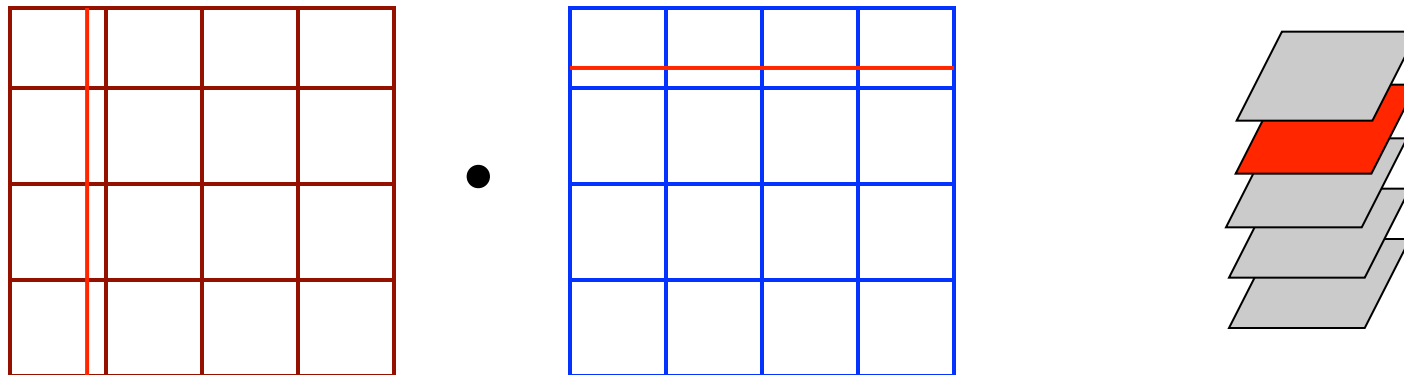


Animation of SUMMA

- Compute the sum of **n** outer products
- Each row & column **(k)** of A & B generates a single outer product
 - ♦ $A[:,k+1] \bullet B[k+1,:]$

for $k := 0$ to $n-1$

$C[:,:] += A[:,k] \bullet B[k,:]$

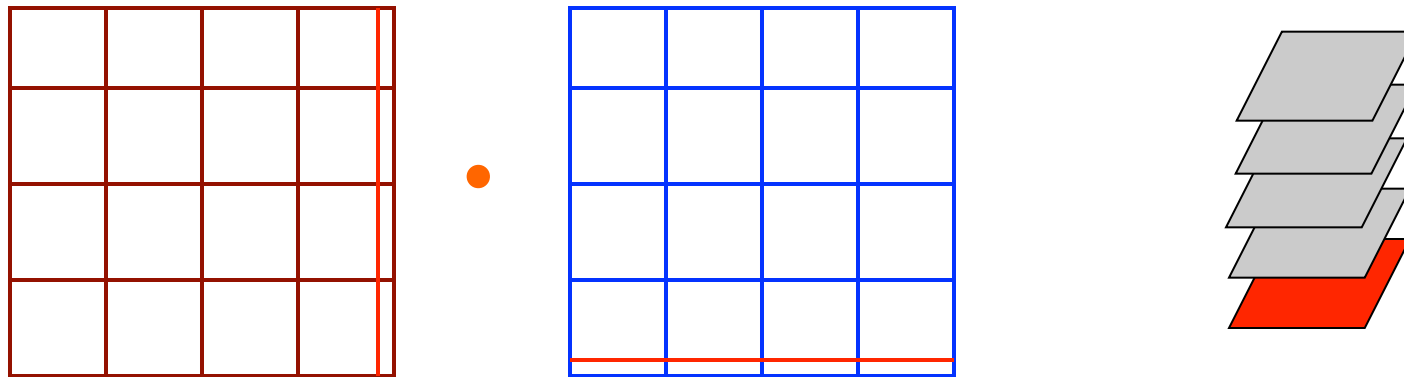


Animation of SUMMA

- Compute the sum of **n** outer products
- Each row & column **(k)** of A & B generates a single outer product
 - ♦ $A[:,n-1] \bullet B[n-1,:]$

for $k := 0$ to $n-1$

$C[:,:] += A[:,k] \bullet B[k,:]$



Parallel algorithm

- Processors organized into rows and columns, process rank an ordered pair
- Processor geometry $P = p_x \times p_y$
- Blocked (serial) matrix multiply, panel size = $b \ll N/\max(p_x, p_y)$

for $k := 0$ to $n-1$ by b

Owner of $A[:,k:k+b-1]$ Bcasts to ACol // Along processor rows

Owner of $B[k:k+b-1,:]$ Bcasts BRow // Along processor columns

$C += \text{Serial Matrix Multiply}(\text{ACol}, \text{BRow})$

- Each row and column of processors independently participate in a panel broadcast
- Owner of the panel (Broadcast root) changes with k , shifts across matrix

