A brief review on

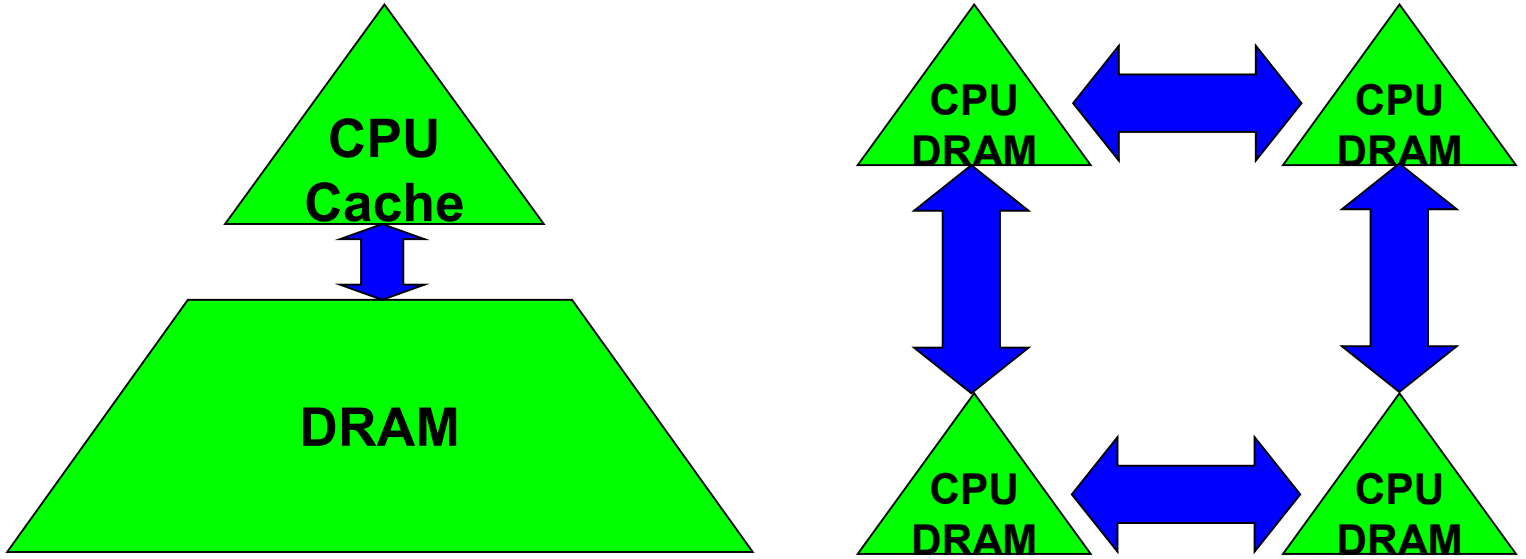
Parallel Matrix Multiplication

- A basic but very important operation from linear algebra
- Building blocks for training of neural networks

Why avoiding communication is important (1/2)

Algorithms have two costs:

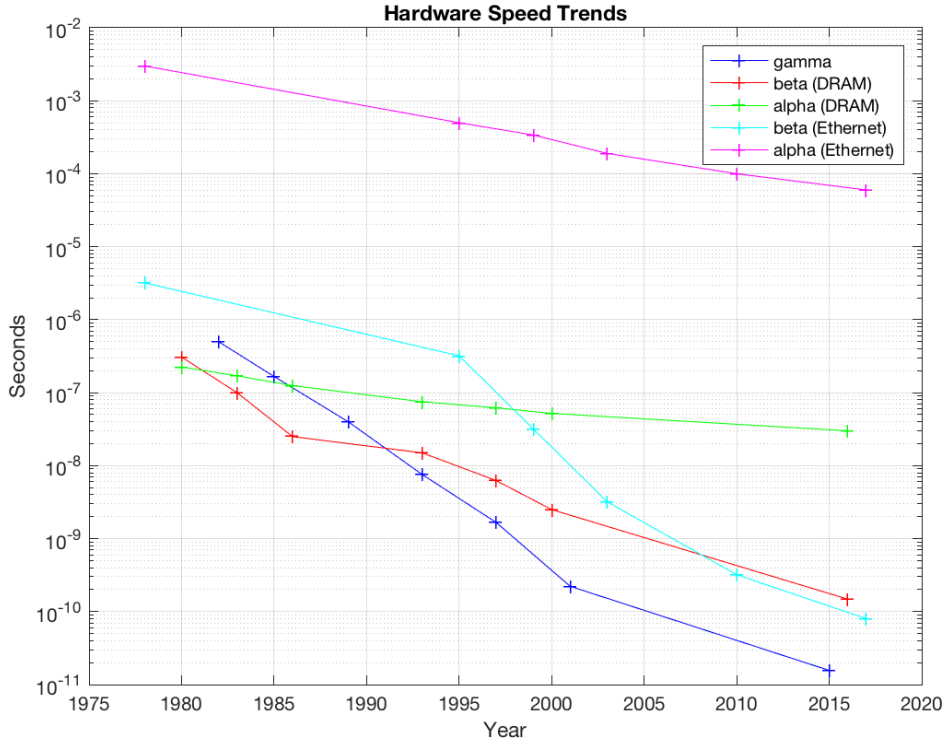
1. Arithmetic (FLOPS)
2. Communication: moving data between
   - levels of a memory hierarchy (sequential case)
   - processors over a network (parallel case).



Why avoiding communication is important (2/2)

- Running time of an algorithm is sum of 3 terms:
  - # flops * time_per_flop (γ)
  - # words moved / bandwidth
  - # messages * latency

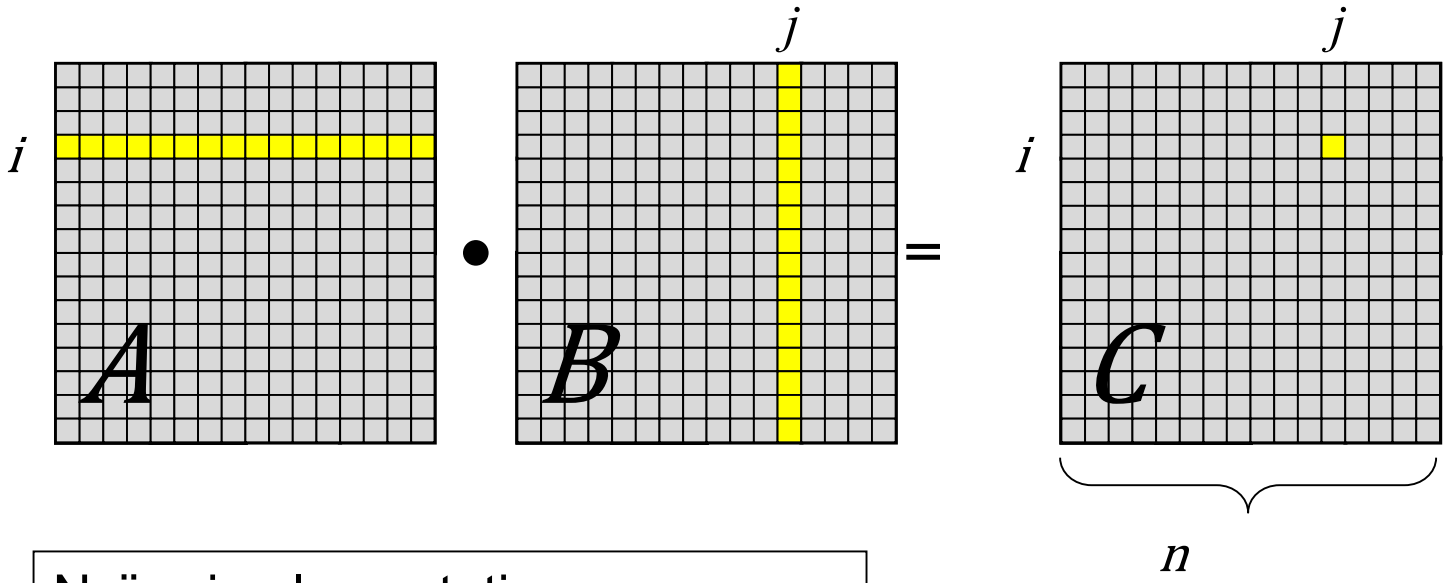
communication



- Time_per_flop (γ)
  << 1/ bandwidth (β)
  << latency (α)
- Gaps growing exponentially with time

- **Minimize communication to save time**

Example: Nested I,J,K-Loop Matrix Multiplication



Naïve implementation

For $i = 1$ to n

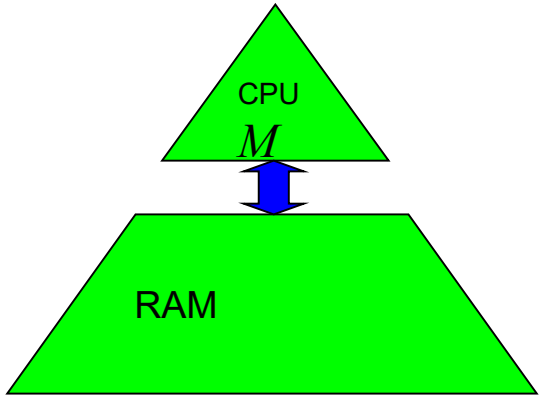
  For $j = 1$ to n

    For $k = 1$ to n

    $C(i, j) = C(i, j) + A(i, k)B(k, j)$

Bandwidth cost: $BW = \Theta(n^3)$

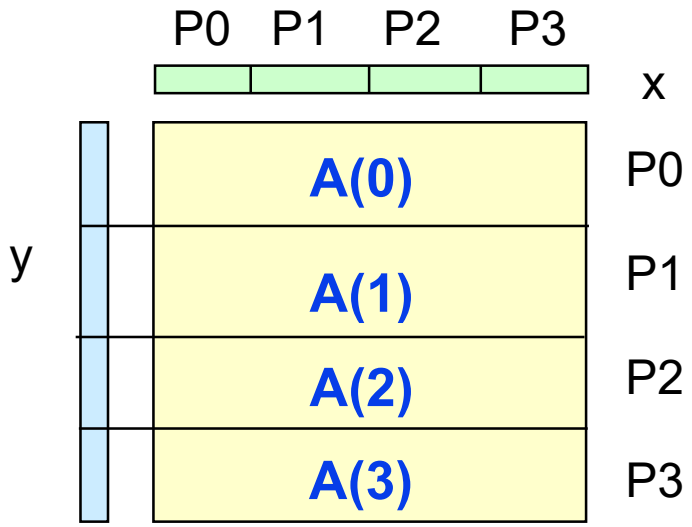
Sequential



Parallel Matrix-Vector Product

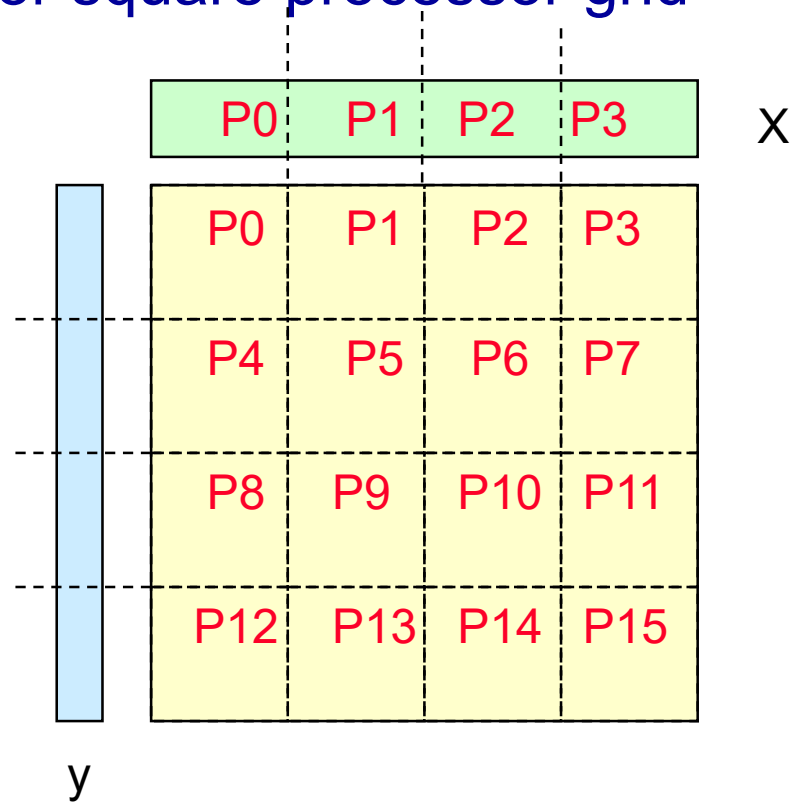
- Compute y = y + A*x, where A is a dense matrix
- Layout:
  - **1D row blocked**
- A(i) refers to the n by n/p block row that processor i owns,
- x(i) and y(i) similarly refer to segments of x,y owned by i
- **Algorithm:**
  - **Foreach processor i**
  - **Broadcast x(i)**
  - **Compute y(i) = A(i)*x**
- Algorithm uses the formula

$$y(i) = y(i) + A(i)*x = y(i) + \Sigma_j A(i,j)*x(j)$$



Matrix-Vector Product y = y + A*x

- A column layout of the matrix eliminates the broadcast of x
  - But adds a reduction to update the destination y
- A 2D blocked layout uses a broadcast and reduction, both on a subset of processors
  - sqrt(p) for square processor grid



Parallel Matrix Multiply

- Computing C=C+A*B
- Using basic algorithm: $2*n^3$ Flops
- Variables are:
  - Data layout: 1D? 2D? Other?
  - Topology of machine: Ring? Torus?
  - Scheduling communication

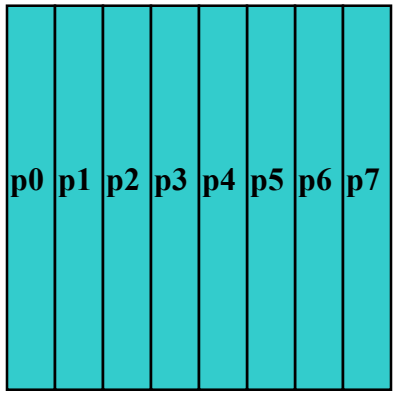
- Use of performance models for algorithm design
  - **Message Time = "latency" + #words * time-per-word**

$$= \alpha + n*\beta$$

- Efficiency (in any model):
  - serial time / (p * parallel time)
  - perfect (linear) speedup $\leftrightarrow$ efficiency = 1

Matrix Multiply with 1D Column Layout

- Assume matrices are n x n and n is divisible by p



May be a reasonable assumption for analysis, not for code

- A(i) refers to the n by n/p block column that processor i owns (similiarly for B(i) and C(i))
- B(i,j) is the n/p by n/p sublock of B(i)
  - in rows j*n/p through (j+1)*n/p - 1
- Algorithm uses the formula

  $C(i) = C(i) + A*B(i) = C(i) + \Sigma_j A(j)*B(j,i)$

Matrix Multiply: 1D Layout on Bus or Ring

- Algorithm uses the formula

  $C(i) = C(i) + A*B(i) = C(i) + \Sigma_j A(j)*B(j,i)$

- First consider a bus-connected machine without broadcast: only one pair of processors can communicate at a time (ethernet)

- Second consider a machine with processors on a ring: all processors may communicate with nearest neighbors simultaneously

MatMul: 1D layout on Bus without Broadcast

Naïve algorithm:

```
C(myproc) = C(myproc) + A(myproc)*B(myproc,myproc)
for i = 0 to p-1
  for j = 0 to p-1 except i
    if (myproc == i) send A(i) to processor j
    if (myproc == j)
      receive A(i) from processor i
      C(myproc) = C(myproc) + A(i)*B(i,myproc)
    barrier
```

Cost of inner loop:

computation: $2*n*(n/p)^2 = 2*n^3/p^2$

communication: $\alpha + \beta*n^2/p$

Naïve MatMul (continued)

Cost of inner loop:

computation: $2*n*(n/p)^2 = 2*n^3/p^2$

communication: $\alpha + \beta*n^2/p$

Data locality ratio ≈$2*n/(\beta*p)$, OK as long as n>>p, BUT

Only 1 pair of processors (i and j) are active on any iteration, and of those, only i is doing computation

=> the algorithm is almost entirely serial

Running time:

= (p*(p-1) + 1)*computation + p*(p-1)*communication

≈ $2*n^3 + p^2*\alpha + p*n^2*\beta$

This is worse than the serial time and grows with p.

Matmul for 1D layout on a Processor Ring

- Pairs of adjacent processors can communicate simultaneously

```
Copy A(myproc) into Tmp
C(myproc) = C(myproc) + Tmp*B(myproc , myproc)
for j = 1 to p-1
    Send Tmp to processor myproc+1 mod p
    Receive Tmp from processor myproc-1 mod p
    C(myproc) = C(myproc) + Tmp*B( myproc-j mod p , myproc)
```

- Same idea as for gravity in simple sharks and fish algorithm
  - May want double buffering in practice for overlap
  - Ignoring deadlock details in code
- Time of inner loop = $2*(\alpha + \beta*n^2/p) + 2*n*(n/p)^2$

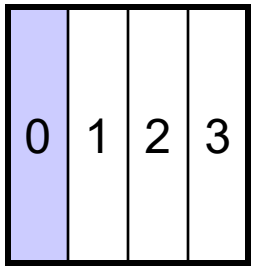
Matmul for 1D layout on a Processor Ring

- Time of inner loop = $2*(\alpha + \beta*n^2/p) + 2*n*(n/p)^2$
- Total Time = $2*n*(n/p)^2 + (p-1) * \text{Time of inner loop}$
- $\approx 2*n^3/p + 2*p*\alpha + 2*\beta*n^2$
- (Nearly) Optimal for 1D layout on Ring or Bus, even with Broadcast:
  - Perfect speedup for arithmetic
  - A(myproc) must move to each other processor, costs at least
    (p-1)*cost of sending n*(n/p) words

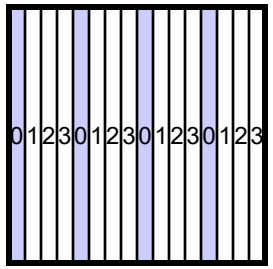
- Parallel Efficiency = $2*n^3 / (p * \text{Total Time})$
  $= 1/(1 + \alpha * p^2/(2*n^3) + \beta * p/(2*n))$
  $= 1/ (1 + O(p/n))$
- Grows to 1 as n/p increases (or α and β shrink)

- But far from communication lower bound

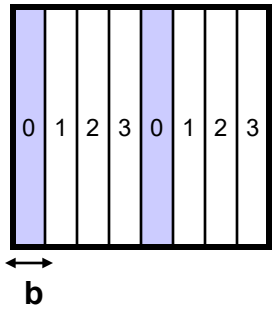
Need to try 2D Matrix layout



1) 1D Column Blocked Layout

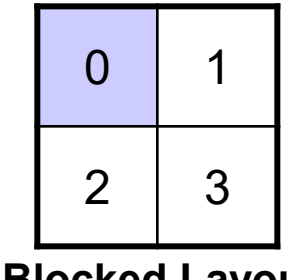


2) 1D Column Cyclic Layout

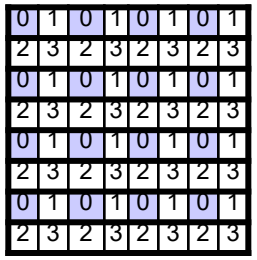


3) 1D Column Block Cyclic Layout

4) Row versions of the previous layouts



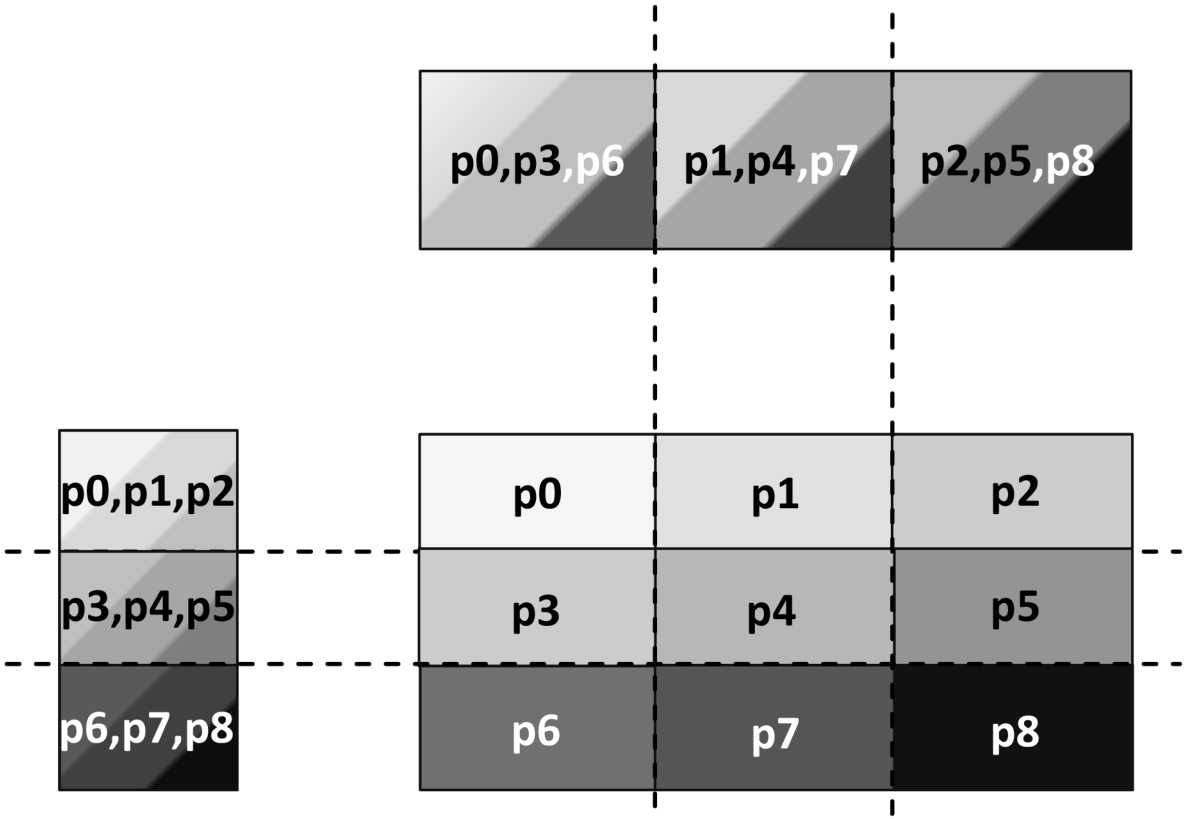
5) 2D Row and Column Blocked Layout



Generalizes others

6) 2D Row and Column Block Cyclic Layout

Parallel Matrix Multiplication: 2D Scheme



E.g., Systolic Array [Hong, Kung 1981]

SUMMA Algorithm

- SUMMA = Scalable Universal Matrix Multiply
- Presented by van de Geijn and Watts
  - www.netlib.org/lapack/lawns/lawn96.ps
- Used in practice in PBLAS = Parallel BLAS
  - www.netlib.org/lapack/lawns/lawn100.ps

SUMMA uses Outer Product form of MatMul

- C = A*B means $C(i,j) = \Sigma_k A(i,k)*B(k,j)$

- Column-wise outer product:

  C = A*B

  $= \Sigma_k A(:,k)*B(k,:)$

  $= \Sigma_k$ (k-th col of A)*(k-th row of B)

- Block column-wise outer product

  (block size = 4 for illustration)

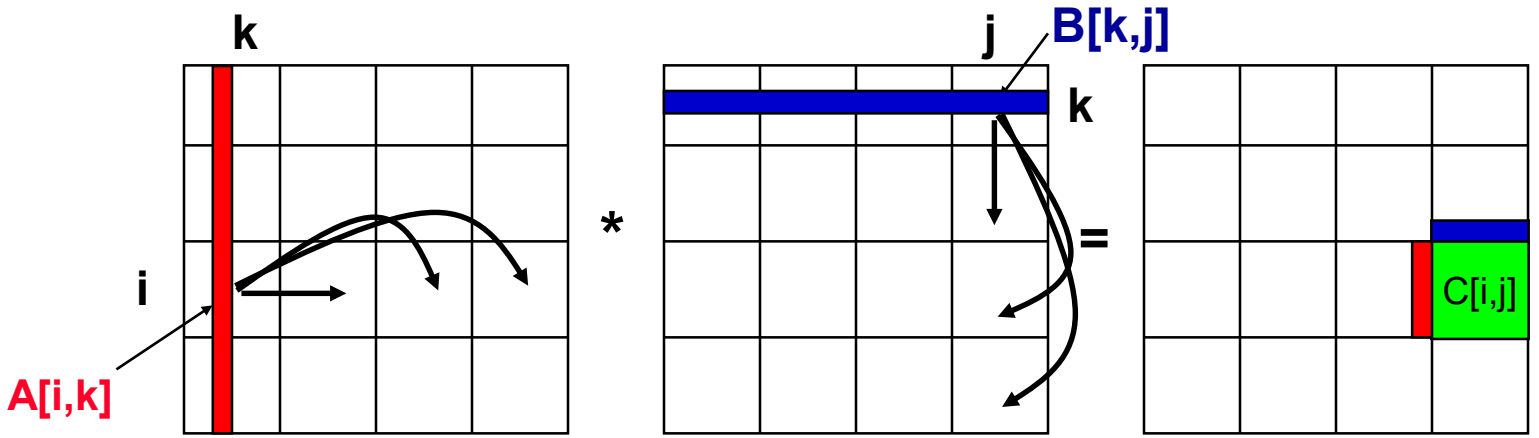
  C = A*B

  = A(:,1:4)*B(1:4,:) + A(:,5:8)*B(5:8,:) + …

  $= \Sigma_k$ (k-th block of 4 cols of A)*

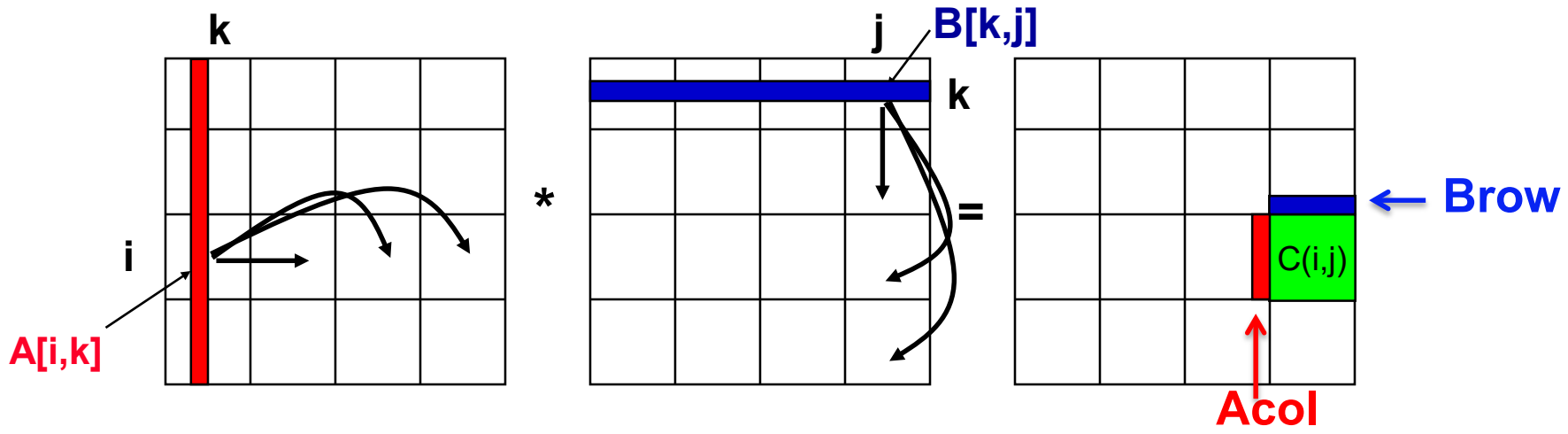
  (k-th block of 4 rows of B)

SUMMA – n x n matmul on $P^{1/2}$ x $P^{1/2}$ grid



- C[i, j] is $n/P^{1/2}$ x $n/P^{1/2}$ submatrix of C on processor P_{ij}
- A[i,k] is $n/P^{1/2}$ x b submatrix of A
- B[k,j] is b x $n/P^{1/2}$ submatrix of B
- C[i,j] = C[i,j] + $\Sigma_{\mathbf{k}}$ A[i,k]*B[k,j]
  - summation over submatrices
- Need not be square processor grid

SUMMA– n x n matmul on $P^{1/2}$ x $P^{1/2}$ grid



For k=0 to n/b-1

for all i = 1 to $P^{1/2}$

owner of A[i,k] broadcasts it to whole processor row (using binary tree)

for all j = 1 to $P^{1/2}$

owner of B[k,j] broadcasts it to whole processor column (using bin. tree)

Receive A[i,k] into Acol

Receive B[k,j] into Brow

C_myproc = C_myproc + Acol * Brow

SUMMA Costs

```
For k=0 to n/b-1
    for all i = 1 to P^1/2
        owner of A[i,k] broadcasts it to whole processor row (using binary tree)
          … #words  = log P^1/2 *b*n/P^1/2 ,     #messages = log P^1/2
    for all j = 1 to  P^1/2
        owner of B[k,j] broadcasts it to whole processor column (using bin. tree)
          …  same #words and #messages
    Receive A[i,k] into Acol
    Receive B[k,j] into Brow
    C_myproc = C_myproc + Acol * Brow     … #flops = 2n^2*b/P
```

- Total #words = $\log P * n^2 / P^{1/2}$ versus 2*n/P on page 17
  - Within factor of log P of lower bound
  - (more complicated implementation removes log P factor)
- Total #messages = log P * n/b
  - Choose b close to maximum, $n/P^{1/2}$, to approach lower bound $P^{1/2}$
- Total #flops = $2n^3/P$

Some communication lower bounds of matrix multiply

- Lower bound assumed 1 copy of data: $M = O(n^2/P)$ per proc.
- What if matrix small enough to fit c>1 copies, so $M = cn^2/P$?
  - #words_moved = $\Omega(\ \#flops / M^{1/2}\ ) = \Omega(\ n^2 / (\ c^{1/2} P^{1/2}\ ))$
  - #messages = $\Omega(\ \#flops / M^{3/2}\ ) = \Omega(\ P^{1/2} / c^{3/2})$
- Can we attain new lower bound?
  - Special case: "3D Matmul": $c = P^{1/3}$
    - Bernsten 89, Agarwal, Chandra, Snir 90, Aggarwal 95
    - Processors arranged in $P^{1/3} \times P^{1/3} \times P^{1/3}$ grid
    - Processor (i,j,k) performs C(i,j) = C(i,j) + A(i,k)*B(k,j), where each submatrix is $n/P^{1/3} \times n/P^{1/3}$
  - Not always that much memory available…

No further discussions

Question ?