CS 240A : Matrix multiplication

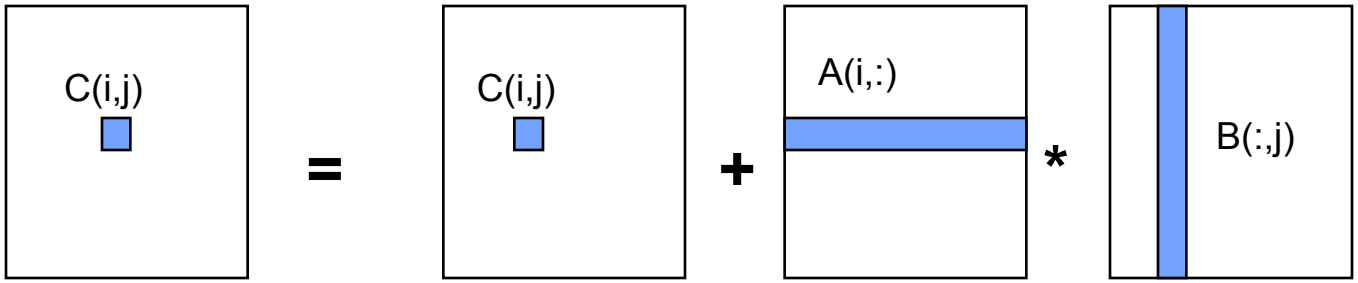
- Matrix multiplication I : parallel issues
- Matrix multiplication II: cache issues

Thanks to Jim Demmel and Kathy Yelick (UCB) for some of these slides

Matrix-Matrix Multiplication ("DGEMM")

```
{ implements C = C + A*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)
```

Algorithm has $2*n^3 = O(n^3)$ Flops and
operates on $3*n^2$ words of memory



Parallel matrix multiply

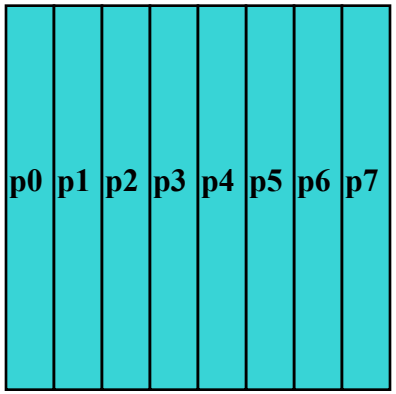
- Compute $C = C + A*B$
- Basic sequential algorithm:
  - C(i,j) += A(i,1)*B(1,j) + A(i,2)*B(1,j) +…+ A(i,n)*B(n,j)
  - work = $t_1 = 2n^3$ floating point operations (“flops”)
- Highly parallel: $t_p = 2n^3 / p$ is easy for p up to at least n^2
- The issue is *communication cost*, as affected by:
  - Data layout
  - Schedule of communication
  - Structure of communication

Communication volume model

- Network of p processors
  - Each with local memory
  - Message-passing
- Communication volume (v)
  - Total size (words) of all messages passed during computation
  - Broadcasting one word costs volume p (actually, p-1)
- No explicit accounting for communication time
  - Thus, can't really model parallel efficiency or speedup; for that, we'd use the latency-bandwidth model

Parallel Matrix Multiply with 1D Column Layout

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



(A reasonable assumption for analysis, not for code)

- Let A(k) be the n-by-n/p block column that processor k owns
  - similarly B(k) and C(k))

C(k) += A * B(k)

- Now let B(i,k) be a subblock of B(k) with n/p rows

C(k) += A(0)*B(0,k) + A(1)*B(1,k) +…+ A(p-1)*B(p-1,k)

Matmul for 1D layout on a Processor Ring

- Proc k communicates only with procs k-1 and k+1
- Different pairs of processors can communicate simultaneously
- Round-Robin "Merry-Go-Round" algorithm

```
Copy A(myproc) into MGR                          (MGR = "Merry-Go-Round")
C(myproc) = C(myproc) + MGR*B(myproc , myproc)
for j = 1 to p-1
    send MGR to processor myproc+1 mod p         (but see deadlock below)
    receive MGR from processor myproc-1 mod p    (but see below)
    C(myproc) = C(myproc) + MGR * B( myproc-j mod p , myproc)
```

- Avoiding deadlock:
  - even procs send then recv, odd procs recv then send
  - or, use nonblocking sends

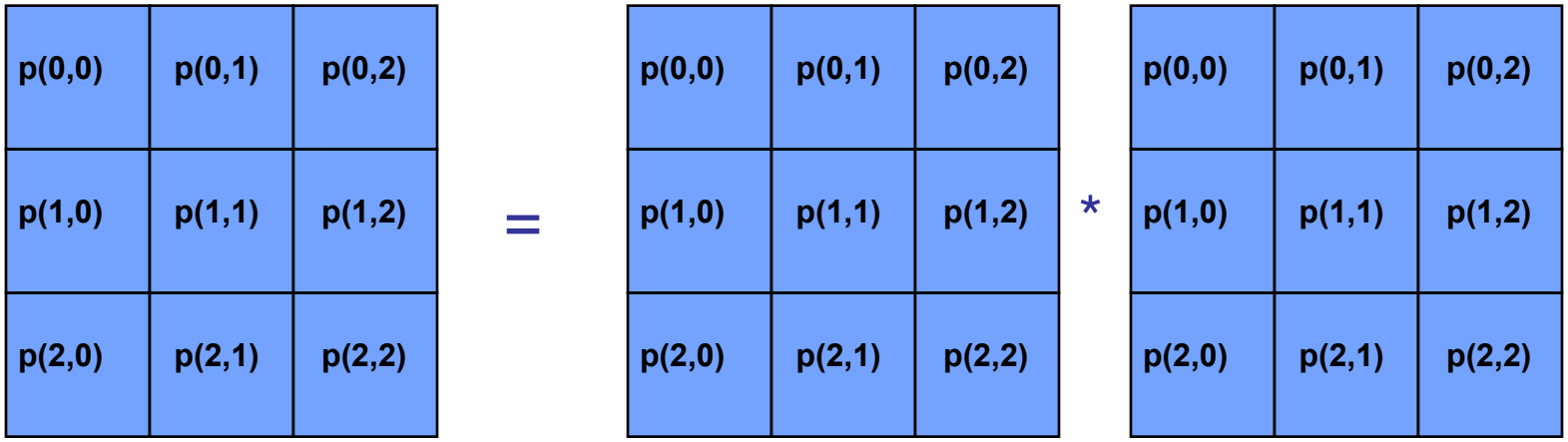
- Comm volume of one inner loop iteration = n^2

Matmul for 1D layout on a Processor Ring

- **One iteration:** $v = n^2$
- **All p-1 iterations:** $v = (p-1) * n^2 \sim pn^2$
- **Optimal for 1D data layout:**
  - **Perfect speedup for arithmetic**
  - **A(myproc) must meet each C(myproc)**
- **"Nice" communication pattern – can probably overlap independent communications in the ring.**
- **In latency/bandwidth model (see extra slides), parallel efficiency** $e = 1 - O(p/n)$

MatMul with 2D Layout

- Consider processors in 2D grid (physical or logical)
- Processors can communicate with 4 nearest neighbors
  - Alternative pattern: broadcast along rows and columns



- Assume p is square s x s grid

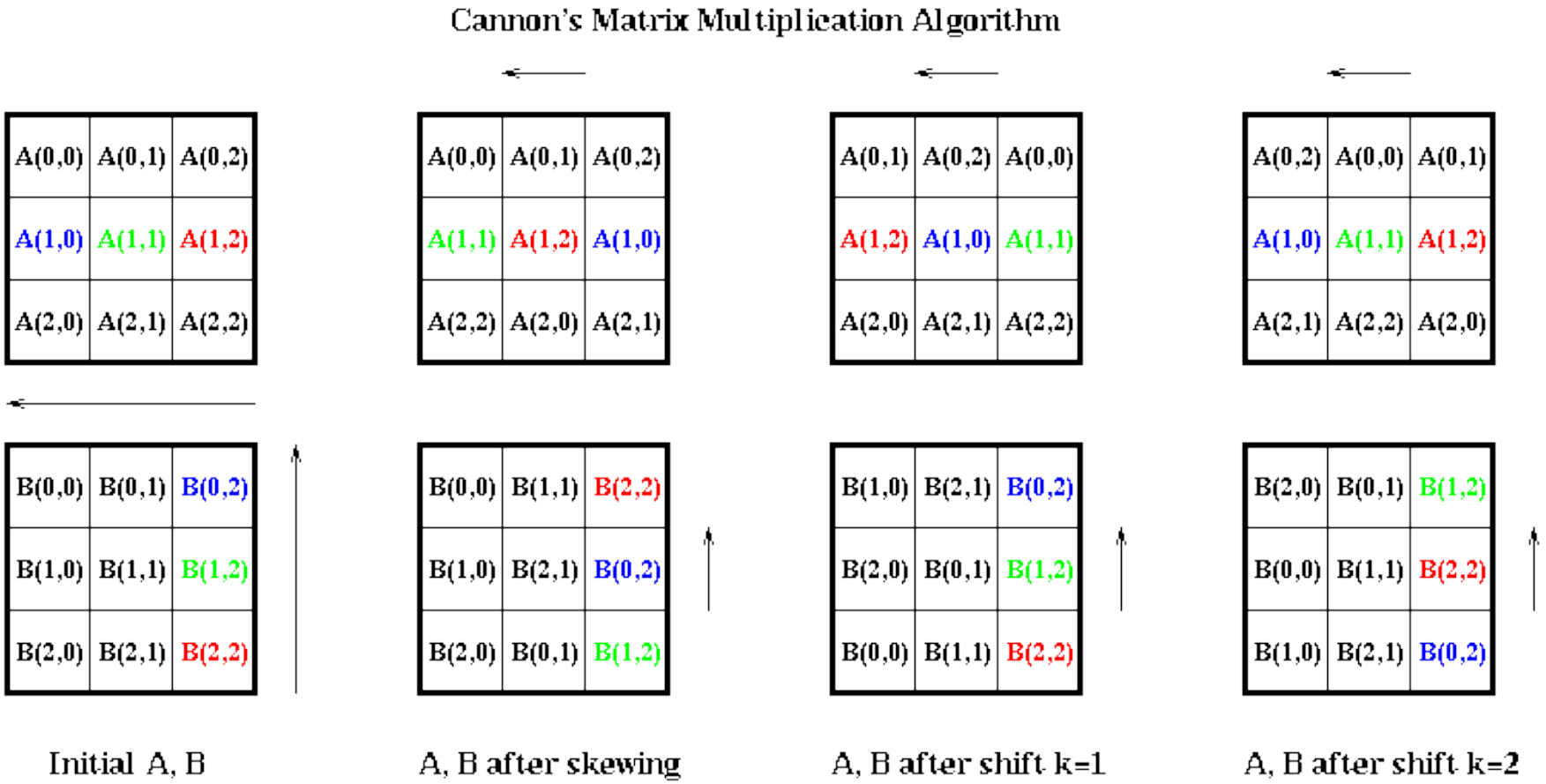
Cannon's Algorithm: 2-D merry-go-round

… C(i,j) = C(i,j) + $\sum_k$ A(i,k)*B(k,j)

… assume s = sqrt(p) is an integer

```
forall  i=0 to s-1                … "skew" A
      left-circular-shift row i of A by i
      … so that A(i,j) overwritten by A(i,(j+i)mod s)
forall  i=0 to s-1                … "skew" B
      up-circular-shift B column i of B by i
      … so that B(i,j) overwritten by B((i+j)mod s), j)
for k=0 to s-1          … sequential
      forall i=0 to s-1 and j=0 to s-1    … all processors in parallel
          C(i,j) = C(i,j) + A(i,j)*B(i,j)
      left-circular-shift each row of A by 1
      up-circular-shift each row of B by 1
```

Cannon's Matrix Multiplication



C(1,2) = A(1,0) * B(0,2) + A(1,1) * B(1,2) + A(1,2) * B(2,2)

Initial Step to Skew Matrices in Cannon

- Initial blocked input

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)

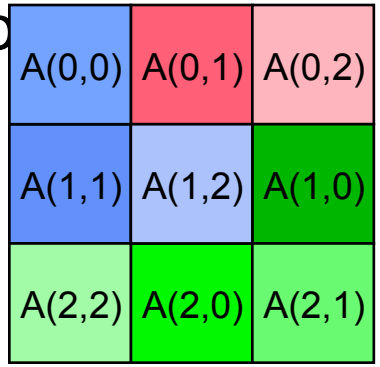
- After skewing before initial block multiplies

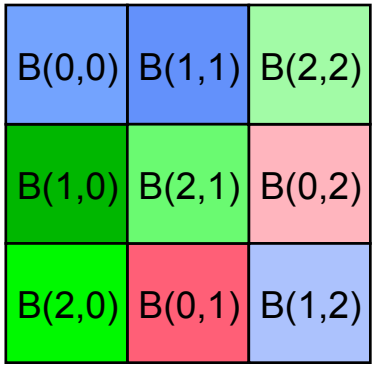
A(0,0)	A(0,1)	A(0,2)
A(1,1)	A(1,2)	A(1,0)
A(2,2)	A(2,0)	A(2,1)

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

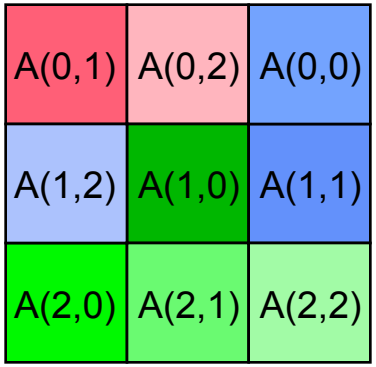
Skewing Steps in Cannon

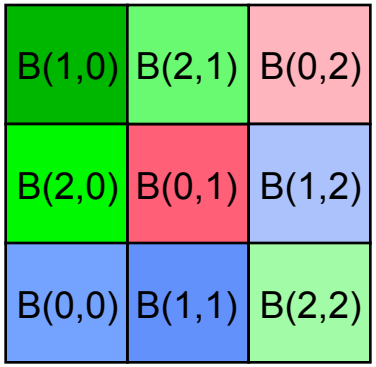
- First step



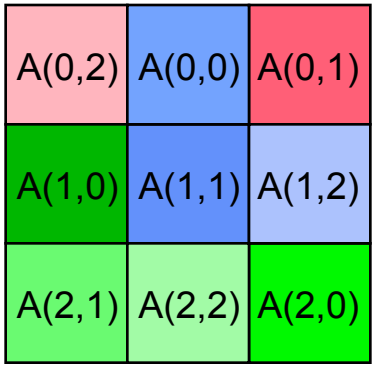


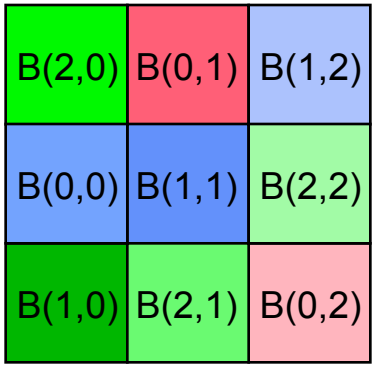
- Second





- Third





Communication Volume of Cannon's Algorithm

```
forall  i=0 to s-1                                  … recall s = sqrt(p)
      left-circular-shift row i of A by i           … v = n2 / s   for each i
 forall  i=0 to s-1
      up-circular-shift B column i of B by i        … v = n2 / s   for each i
 for k=0 to s-1
       forall  i=0 to s-1 and j=0 to s-1
            C(i,j) = C(i,j) + A(i,j)*B(i,j)
       left-circular-shift each row of A by 1       … v = n2   for each k
       up-circular-shift each row of B by 1         … v = n2   for each k
```

- **Total comm** $v = 2*n^2 + 2*s*n^2 \sim 2*\sqrt{p}*n^2$
- **Again, "nice" communication pattern**
- **In latency/bandwidth model (see extra slides), parallel efficiency** $e = 1 - O(\sqrt{p}/n)$

Drawbacks to Cannon

- Hard to generalize for
  - p not a perfect square
  - A and B not square
  - dimensions of A, B not perfectly divisible by s = sqrt(p)
  - A and B not "aligned" in the way they are stored on processors
  - block-cyclic layouts

- Memory hog (extra copies of local matrices)

- Algorithm used instead in practice is SUMMA
  - uses row and column broadcasts, not merry-go-round
  - see extra slides below for details

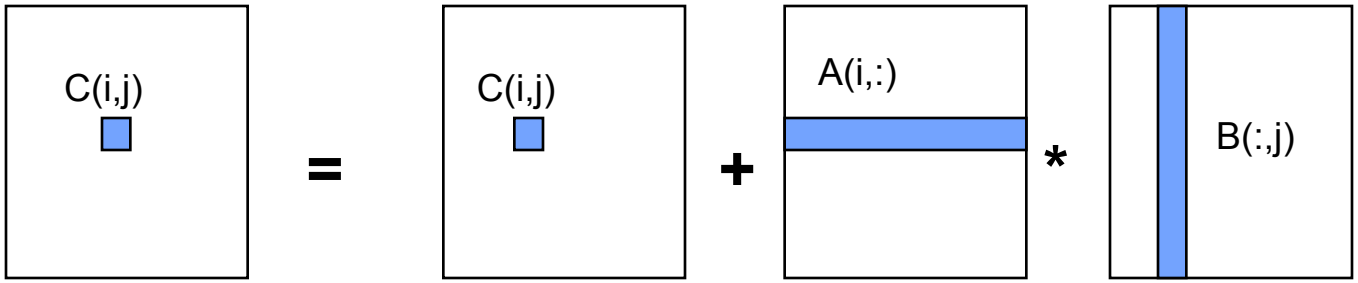
Sequential Matrix Multiplication

Simple mathematics, but getting good performance is complicated by memory hierarchy --- cache issues.

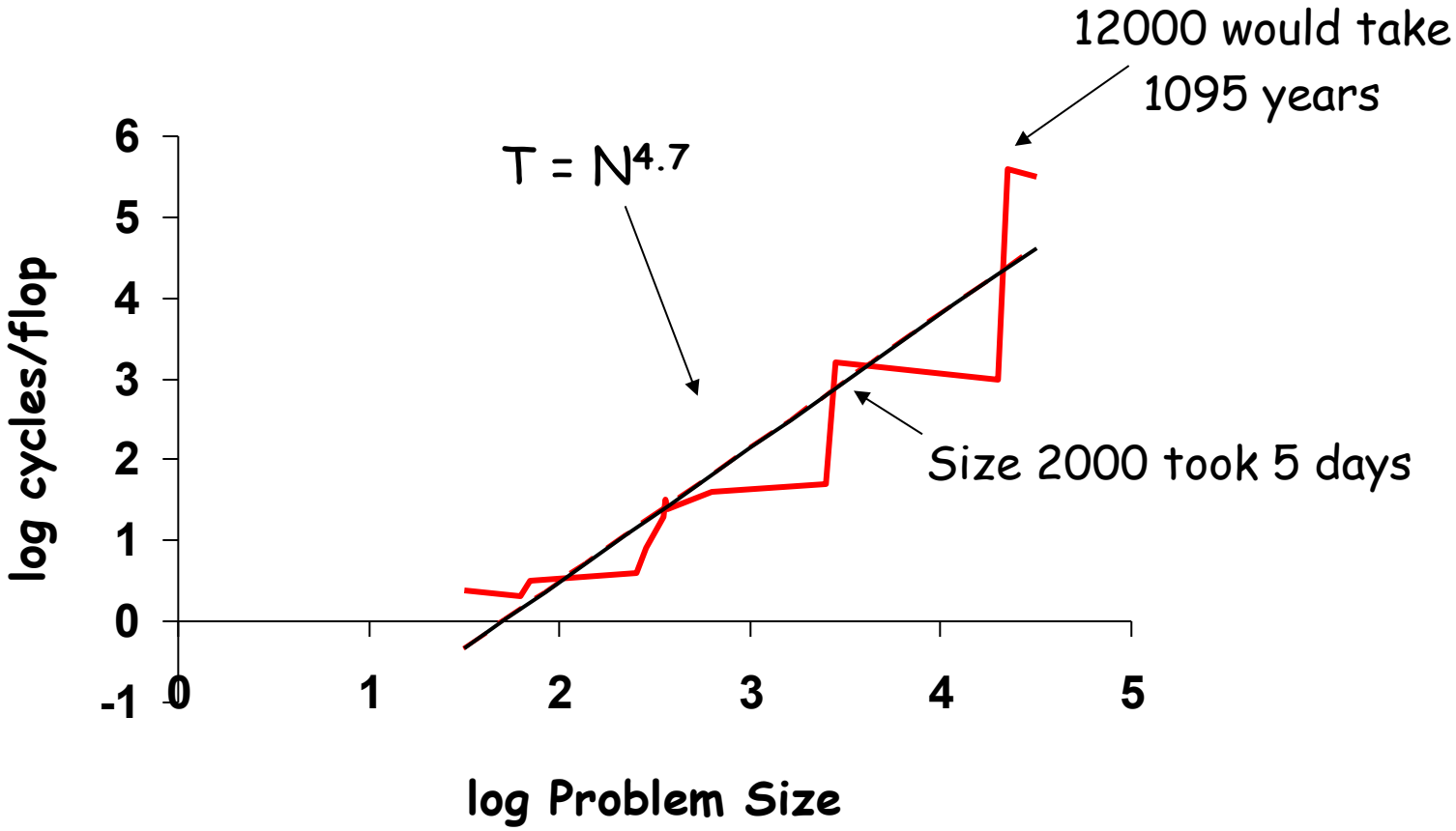
"Naïve" 3-Loop Matrix Multiply

```
{implements C = C + A*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)
```

Algorithm has $2*n^3 = O(n^3)$ Flops and
operates on $3*n^2$ words of memory



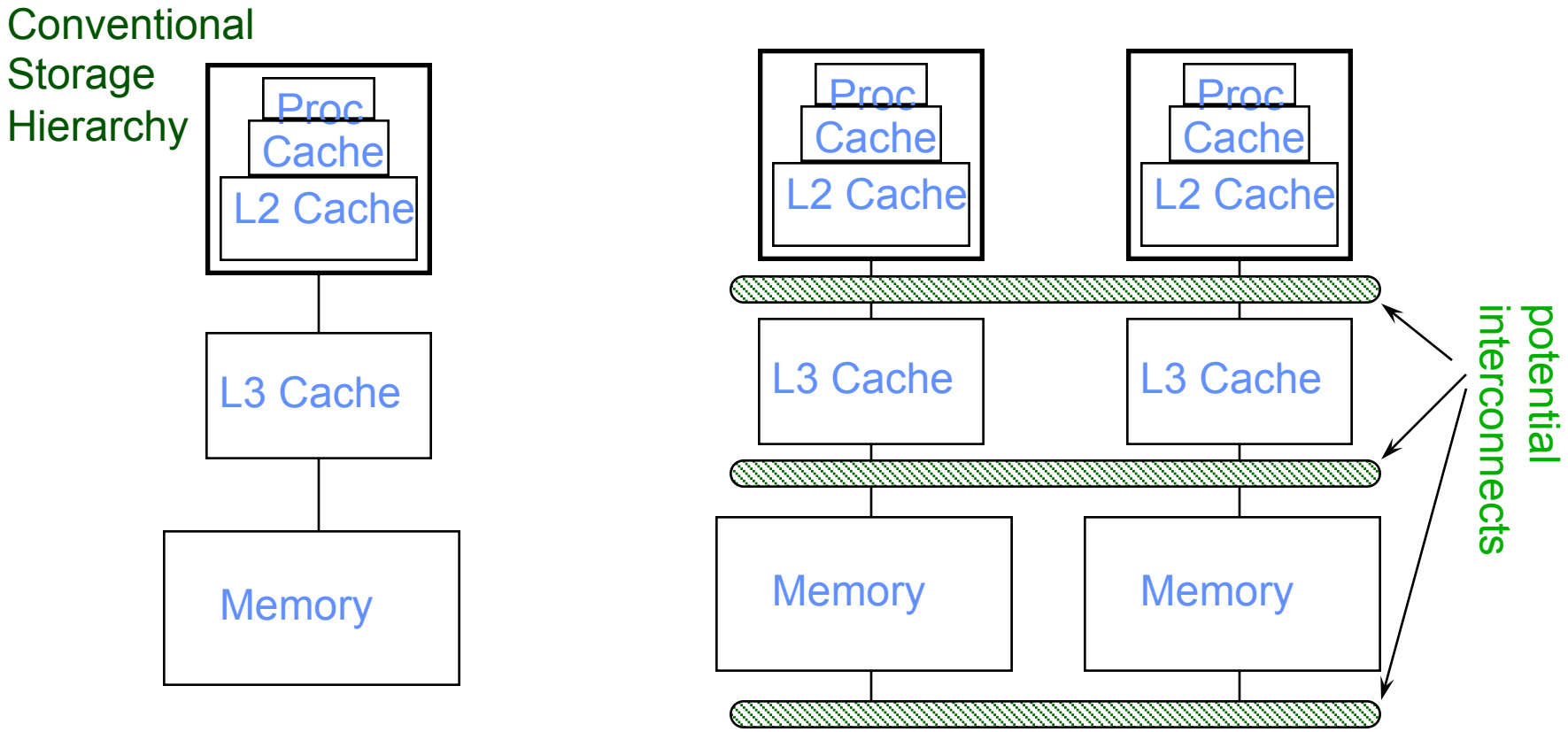
3-Loop Matrix Multiply *[Alpern et al., 1992]*



$O(N^3)$ performance would have constant cycles/flop
Performance looks much closer to $O(N^5)$

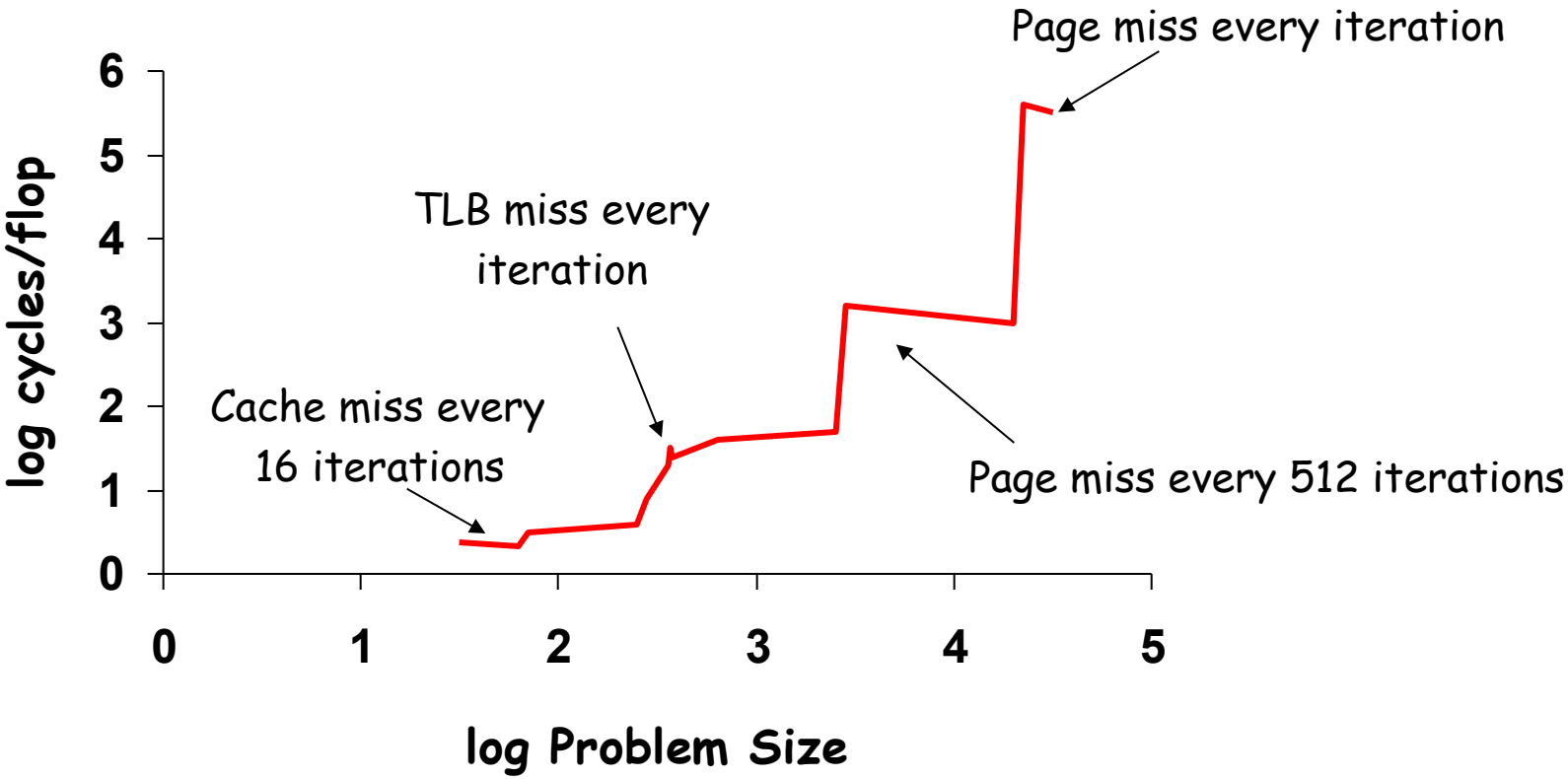
Slide source: Larry Carter, UCSD

Avoiding data movement: Reuse and locality



- Large memories are slow, fast memories are small
- Parallel processors, collectively, have large, fast cache
  - the slow accesses to “remote” data we call “communication”
- Algorithm should do most work on local data

3-Loop Matrix Multiply *[Alpern et al., 1992]*



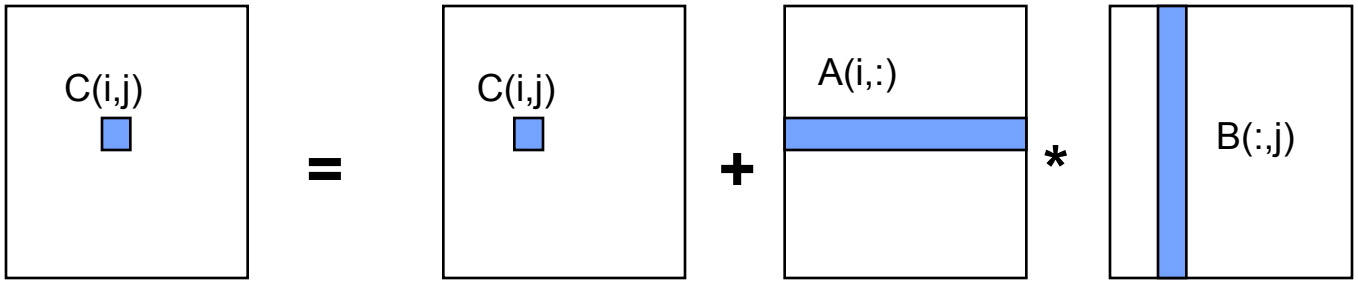
Slide source: Larry Carter, UCSD

Simplified model of hierarchical memory

- Assume just 2 levels in the hierarchy, fast and slow
  - v = number of words moved between fast and slow memory
  - op_m = time per slow memory operation
  - t_1 = number of arithmetic operations
  - op_f = time per arithmetic operation << op_m
- Computational Intensity: $q = t_1 / v$
  - average number of flops per slow element access
- Minimum possible time = $t_1 * op_f$ when all data in fast memory
- Actual time
  - $t_1 * op_f + v * op_m = t_1 * op_f * (1 + op_m/op_f * 1/q)$
- Larger q means time closer to minimum $t_1 * op_f$

"Naïve" Matrix Multiply

```
{implements C = C + A*B}
for i = 1 to n
  {read row i of A into fast memory}
   for j = 1 to n
       {read C(i,j) into fast memory}
       {read column j 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}
```

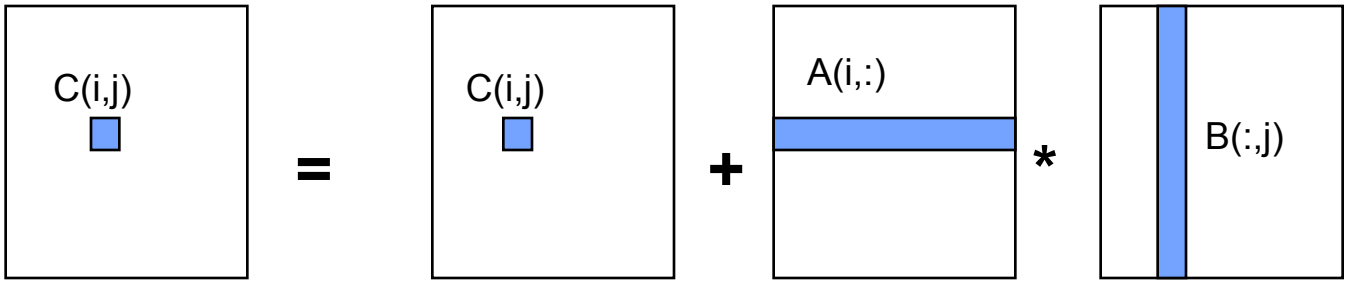


"Naïve" Matrix Multiply

How many references to slow memory?

$v = n^3$ read each column of B n times
+ n^2 read each row of A once
+ $2n^2$ read and write each element of C once
$= n^3 + 3n^2$

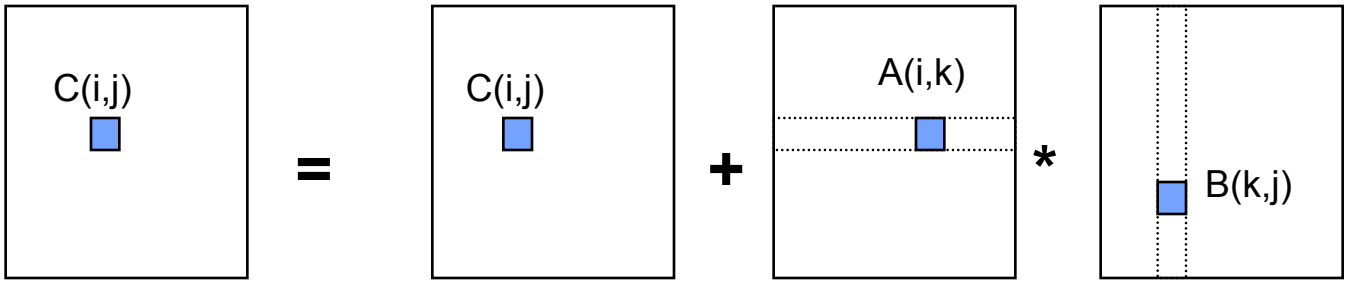
So $q = t / v = 2n^3 / (n^3 + 3n^2)$
$\sim= 2$ for large n, no improvement over matrix-vector multiply



Blocked Matrix Multiply

Consider A,B,C to be N by N matrices of b by b subblocks where b=n / N is called the block size

```
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 Multiply

v is amount memory traffic between slow and fast memory
matrix has nxn elements, and NxN blocks each of size bxb
t is number of floating point operations, $2n^3$ for this problem
q = t / v measures data reuse, or computational intensity

v = $N*n^2$ read every block of B N times
+ $N*n^2$ read every block of A N times
+ $2n^2$ read and write every block of C once
= $(2N + 2) * n^2$

Computational intensity $q = t / v = 2n^3 / ((2N + 2) * n^2)$
$\sim= n / N = b$ for large n

We can improve performance by increasing the blocksize b
(but only until $3b^2$ gets as big as the fast memory size)

Can be much faster than matrix-vector multiply (q = 2)

Multi-Level Blocked Matrix Multiply

- More levels of memory hierarchy => more levels of blocking!
- Version 1: One level of blocking for each level of memory
  (L1 cache, L2 cache, L3 cache, DRAM, disk, …)
  - Version 2: Recursive blocking, O(log n) levels deep

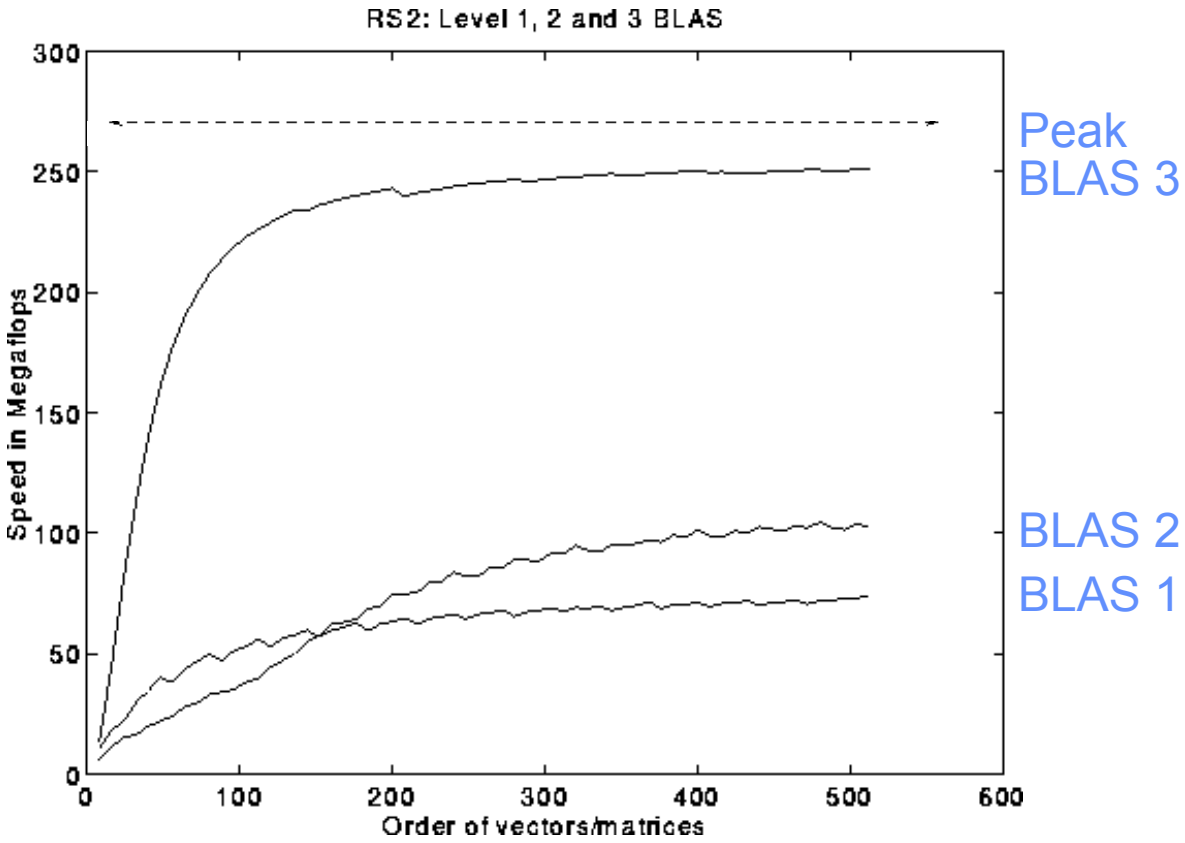
In the "Uniform Memory Hierarchy" cost model,
the 3-loop algorithm is $O(N^5)$ time,
but the blocked algorithms are $O(N^3)$

BLAS: Basic Linear Algebra Subroutines

- Industry standard interface
- Vendors, others supply optimized implementations
- History
  - BLAS1 (1970s):
    - vector operations: dot product, saxpy ($y=\alpha*x+y$), etc
    - m=2*n, t=2*n, q ~1 or less
  - BLAS2 (mid 1980s)
    - matrix-vector operations: matrix vector multiply, etc
    - v=n^2, t=2*n^2, q~2, less overhead
    - somewhat faster than BLAS1
  - BLAS3 (late 1980s)
    - matrix-matrix operations: matrix matrix multiply, etc
    - v >= n^2, t=O(n^3), so q can possibly be as large as n
    - BLAS3 is potentially much faster than BLAS2
- Good algorithms use BLAS3 when possible (LAPACK)
  - See `www.netlib.org/blas, www.netlib.org/lapack`

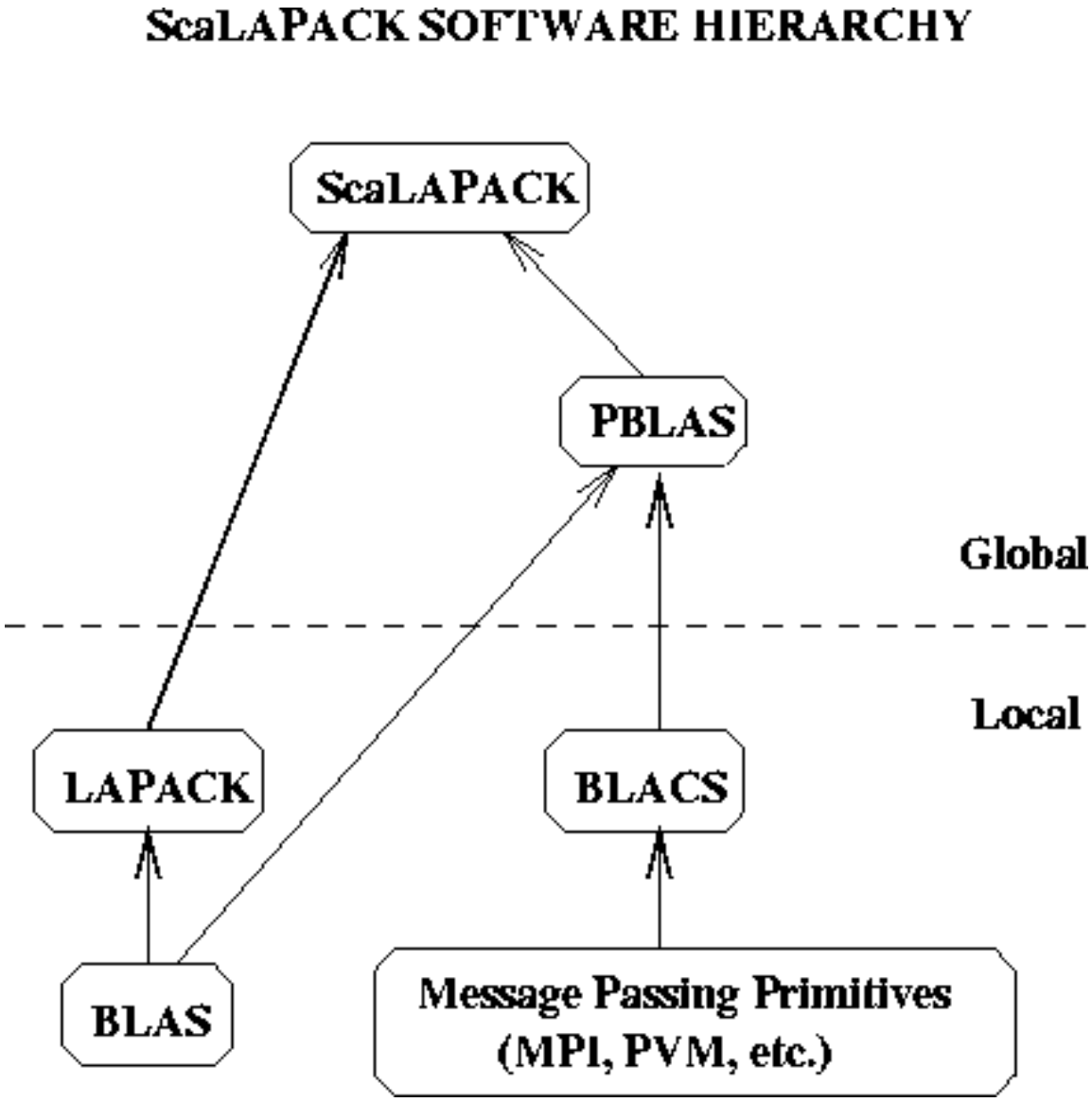
BLAS speeds on an IBM RS6000/590

Peak speed = 266 Mflops



BLAS 3 (n-by-n matrix matrix multiply) vs
BLAS 2 (n-by-n matrix vector multiply) vs
BLAS 1 (saxpy of n vectors)

ScaLAPACK Parallel Library



Extra Slides:

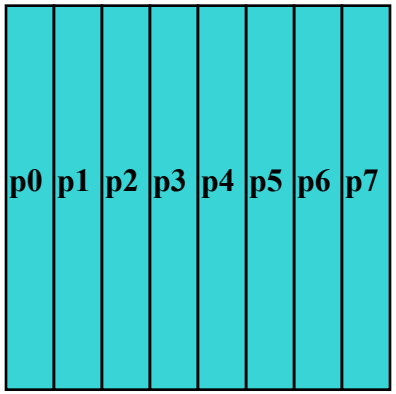
Parallel matrix multiplication in the latency-bandwidth cost model

Latency Bandwidth Model

- Network of p processors, each with local memory
  - Message-passing
- Latency (α)
  - Cost of communication per message
- Inverse bandwidth (β)
  - Cost of communication per unit of data
- Parallel time (t_p)
  - Computation time plus communication time
- Parallel efficiency:
  - $e(p) = t_1 / (p * t_p)$
  - perfect speedup → $e(p) = 1$

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) is the n-by-n/p block column that processor i owns (similarly 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
- Formula: C(i) = C(i) + A*B(i) = C(i) + $\sum_{j=0:p}$ A(j) * B(j,i)

Matmul for 1D layout on a Processor Ring

- Proc k communicates only with procs k-1 and k+1
- Different pairs of processors can communicate simultaneously
- Round-Robin "Merry-Go-Round" algorithm

```
Copy A(myproc) into MGR                          (MGR = "Merry-Go-Round")
C(myproc) = C(myproc) + MGR*B(myproc , myproc)
for j = 1 to p-1
    send MGR to processor myproc+1 mod p         (but see deadlock below)
    receive MGR from processor myproc-1 mod p    (but see below)
    C(myproc) = C(myproc) + MGR * B( myproc-j mod p , myproc)
```

- Avoiding deadlock:
  - even procs send then recv, odd procs recv then send
  - or, use nonblocking sends

- 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 = $2*n*(n/p)^2 + (p-1) * \text{Time of inner loop}$
- $\sim 2*n^3/p + 2*p*\alpha + 2*\beta*n^2$

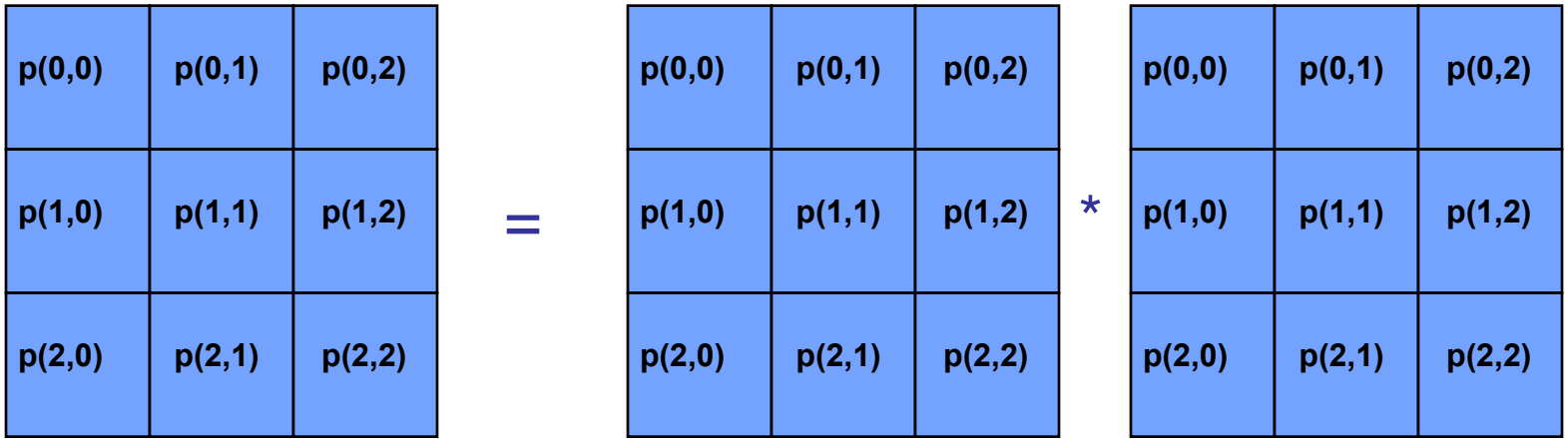
- Optimal for 1D layout on Ring or Bus, even with broadcast:
  - Perfect speedup for arithmetic
  - A(myproc) must move to each other processor, costs at least
    (p-1)*cost of sending n*(n/p) words

- Parallel Efficiency = $2*n^3 / (p * \text{Total Time})$
  $= 1/(1 + \alpha * p^2/(2*n^3) + \beta * p/(2*n))$
  $= 1/(1 + O(p/n))$
  $= 1 - O(p/n)$

- Grows to 1 as n/p increases (or α and β shrink)

MatMul with 2D Layout

- Consider processors in 2D grid (physical or logical)
- Processors can communicate with 4 nearest neighbors
  - Alternative pattern: broadcast along rows and columns



- Assume p is square s x s grid

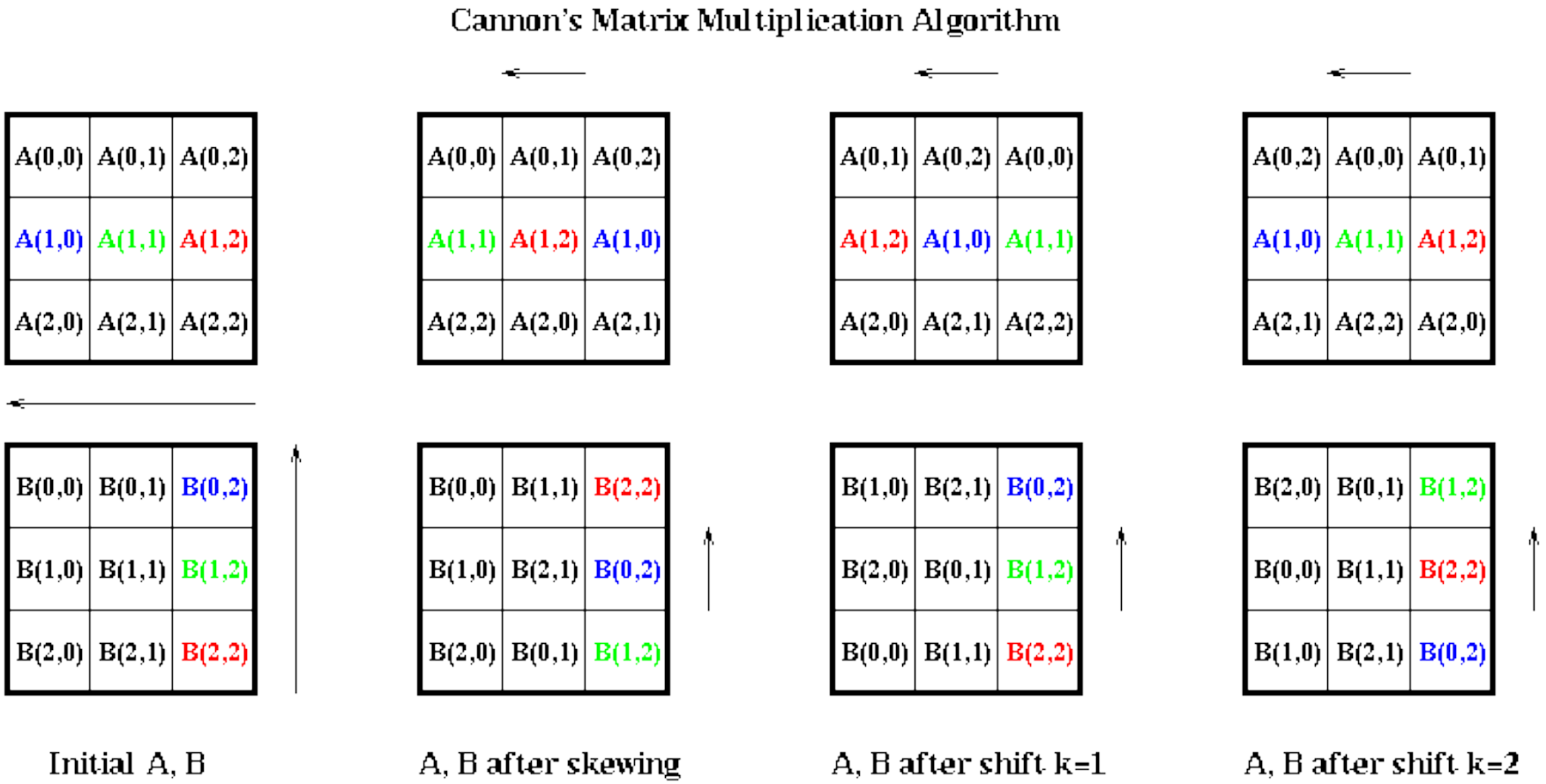
Cannon's Algorithm: 2-D merry-go-round

… C(i,j) = C(i,j) + $\sum_k$ A(i,k)*B(k,j)

… assume s = sqrt(p) is an integer

```
forall  i=0 to s-1                … "skew" A
     left-circular-shift row i of A by i
     … so that A(i,j) overwritten by A(i,(j+i)mod s)
forall  i=0 to s-1                … "skew" B
     up-circular-shift B column i of B by i
     … so that B(i,j) overwritten by B((i+j)mod s), j)
for k=0 to s-1          … sequential
     forall i=0 to s-1 and j=0 to s-1    … all processors in parallel
          C(i,j) = C(i,j) + A(i,j)*B(i,j)
          left-circular-shift each row of A by 1
          up-circular-shift each row of B by 1
```

Cannon's Matrix Multiplication



C(1,2) = A(1,0) * B(0,2) + A(1,1) * B(1,2) + A(1,2) * B(2,2)

Initial Step to Skew Matrices in Cannon

- Initial blocked input

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)

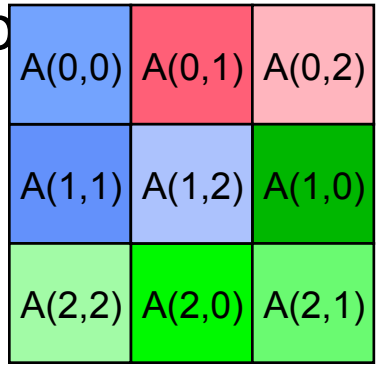
- After skewing before initial block multiplies

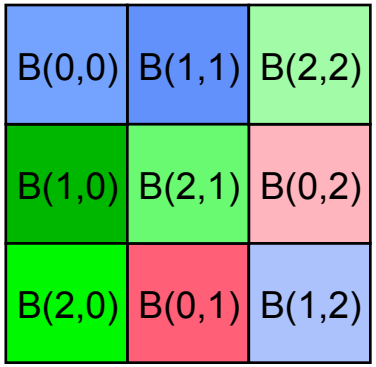
A(0,0)	A(0,1)	A(0,2)
A(1,1)	A(1,2)	A(1,0)
A(2,2)	A(2,0)	A(2,1)

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

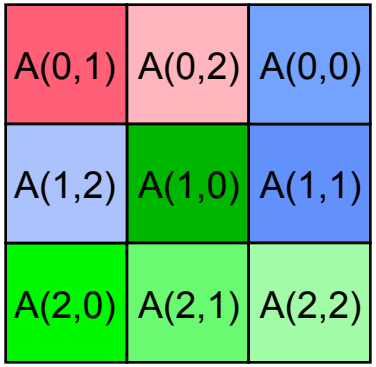
Skewing Steps in Cannon

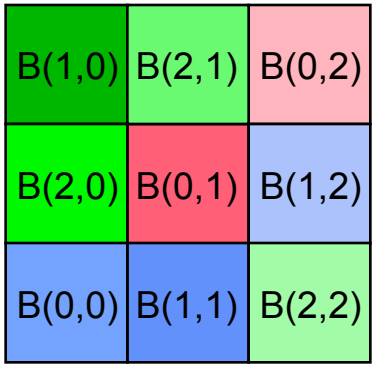
- First step



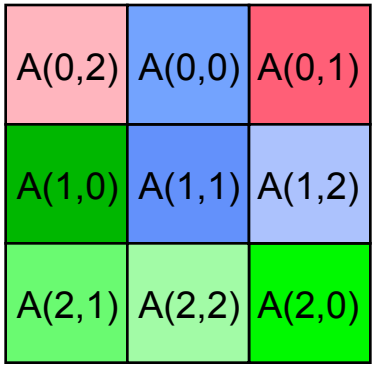


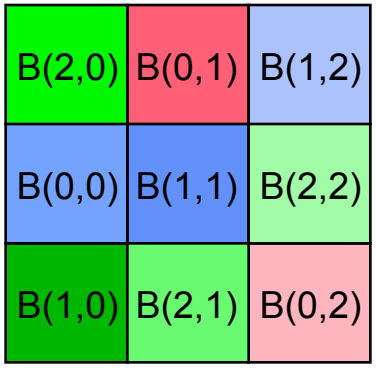
- Second





- Third





Cost of Cannon's Algorithm

```
forall  i=0 to s-1                    …  recall s = sqrt(p)
      left-circular-shift row i of A by i     … cost = s*(α + β*n2/p)
 forall  i=0 to s-1
      up-circular-shift B column i of B by i … cost = s*(α + β*n2/p)
 for k=0 to s-1
       forall  i=0 to s-1 and j=0 to s-1
           C(i,j) = C(i,j) + A(i,j)*B(i,j)   … cost = 2*(n/s)3 = 2*n3/p3/2
           left-circular-shift each row of A by 1   … cost = α + β*n2/p
           up-circular-shift each row of B by 1     … cost = α + β*n2/p
```

- **Total Time = $2*n^3/p + 4* s*\alpha + 4*\beta*n^2/s$**
- **Parallel Efficiency = $2*n^3$ / (p * Total Time)**
  = $1/(1 + \alpha * 2*(s/n)^3 + \beta * 2*(s/n))$
  = 1 - O(sqrt(p)/n)
- **Grows to 1 as n/s = n/sqrt(p) = sqrt(data per processor) grows**
- **Better than 1D layout, which had Efficiency = 1 - O(p/n)**

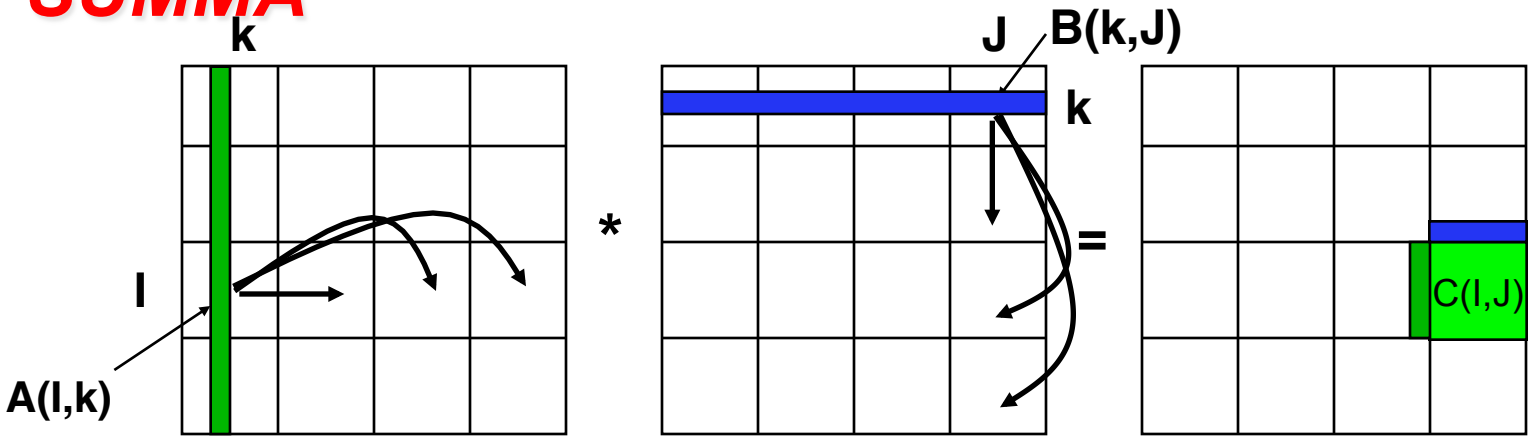
Extra Slides:

SUMMA parallel matrix multiplication algorithm

SUMMA Algorithm

- SUMMA = Scalable Universal Matrix Multiply
- Slightly less efficient than Cannon
  … but simpler and easier to generalize
- Presentation from van de Geijn and Watts
  - www.netlib.org/lapack/lawns/lawn96.ps
  - Similar ideas appeared many times
- Used in practice in PBLAS = Parallel BLAS
  - www.netlib.org/lapack/lawns/lawn100.ps

SUMMA

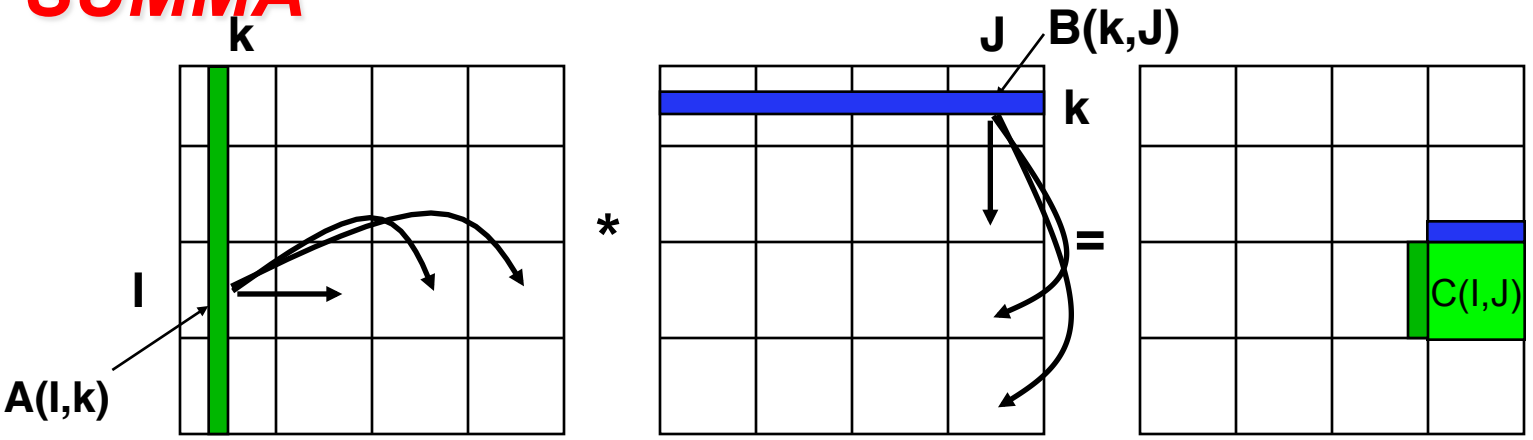


- I, J represent all rows, columns owned by a processor
- k is a single row or column
  - or a block of b rows or columns

- $C(I,J) = C(I,J) + \Sigma_k A(I,k)*B(k,J)$

- Assume a p_r by p_c processor grid ($p_r = p_c = 4$ above)
  - Need not be square

SUMMA



For k=0 to n-1 … or n/b-1 where b is the block size

… = # cols in A(I,k) and # rows in B(k,J)

for all I = 1 to p_r … in parallel

owner of A(I,k) broadcasts it to whole processor row

for all J = 1 to p_c … in parallel

owner of B(k,J) broadcasts it to whole processor column

Receive A(I,k) into Acol

Receive B(k,J) into Brow

C(myproc , myproc) = C(myproc , myproc) + Acol * Brow

SUMMA performance

- To simplify analysis only, assume s = sqrt(p)

```
For k=0 to n/b-1
    for all I = 1 to s   …  s = sqrt(p)
        owner of A(I,k) broadcasts it to whole processor row
            … time = log s *( α + β * b*n/s), using a tree
    for all J = 1 to  s
        owner of B(k,J) broadcasts it to whole processor column
            … time = log s *( α + β * b*n/s), using a tree
    Receive A(I,k) into Acol
    Receive B(k,J) into Brow
    C( myproc , myproc ) = C( myproc , myproc) + Acol * Brow
            … time = 2*(n/s)^2*b
```

- Total time = $2*n^3/p + \alpha * \log p * n/b + \beta * \log p * n^2/s$

SUMMA performance

- Total time = $2*n^3/p + \alpha * \log p * n/b + \beta * \log p * n^2/s$
- Parallel Efficiency =

  $1/(1 + \alpha * \log p * p / (2*\beta*n^2) + \beta * \log p * s/(2*n))$
- ~Same β term as Cannon, except for log p factor

  log p grows slowly so this is ok
- Latency (α) term can be larger, depending on b

  When b=1, get $\alpha * \log p * n$

  As b grows to n/s, term shrinks to

  $\alpha * \log p * s$ (log p times Cannon)
- Temporary storage grows like 2*b*n/s
- Can change b to tradeoff latency cost with memory