

---

# **CS 267**

# **Parallel Matrix Multiplication**

Kathy Yelick

<http://www.cs.berkeley.edu/~yelick/cs267>

# Parallel Numerical Algorithms

---

- Lecture schedule:
  - **3/8: Dense Matrix Products**
    - BLAS 1: Vector operations
    - BLAS 2: Matrix-Vector operations
    - BLAS 3: Matrix-Matrix operations
    - Use of Performance models in algorithm design
  - **3/10: Dense Matrix Solvers**
  - **3/12: Matrix Multiply context results (HW1)**
    - 310 Soda at 1:30pm
  - **3/15: Sparse Matrix Products**
  - **3/17: Sparse Direct Solvers**

# Parallel Vector Operations

---

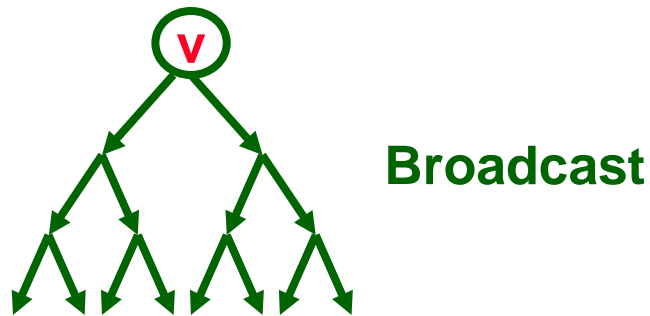
Some common vector operations for vectors  $x, y, z$ :

- Vector add:  $z = x + y$ 
  - Trivial to parallelize if vectors are aligned
- AXPY:  $z = a * x + y$  (where  $a$  is scalar)
  - Broadcast  $a$ , followed by independent  $*$  and  $+$
- Dot product:  $s = \sum_j x[j] * y[j]$ 
  - Independent  $*$  followed by  $+$  reduction

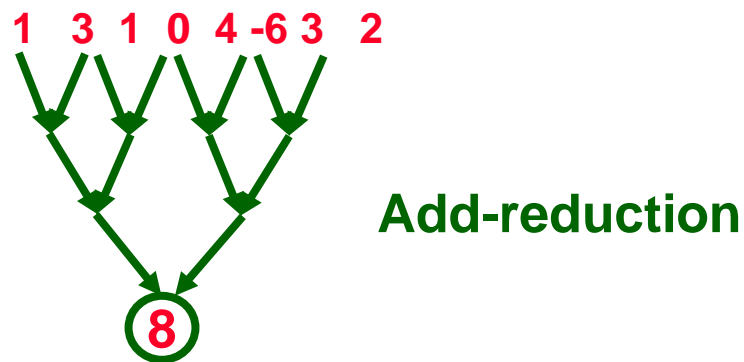
# Broadcast and reduction

---

- Broadcast of 1 value to  $p$  processors in  $\log p$  time



- Reduction of  $p$  values to 1 in  $\log p$  time
- Takes advantage of associativity in  $+$ ,  $*$ ,  $\min$ ,  $\max$ , etc.

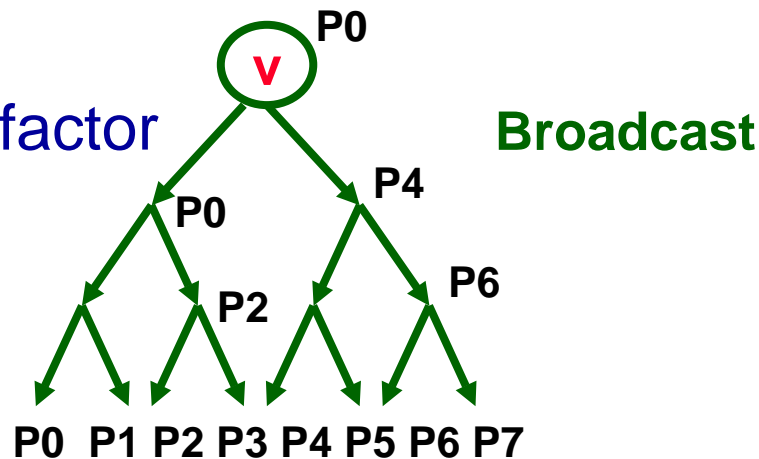


# Broadcast Algorithms

---

- Sequential or “centralized” algorithm
  - P0 sends value to P-1 other processors in sequence
  - $O(P)$  algorithm
  - Note: variations in UPC/Titanium model based on whether P0 writes to all others, or others read from P0

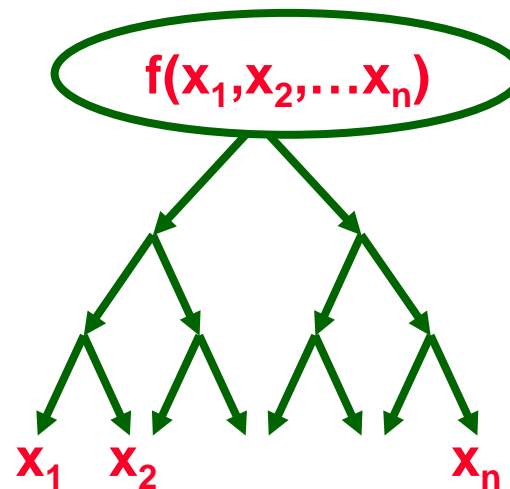
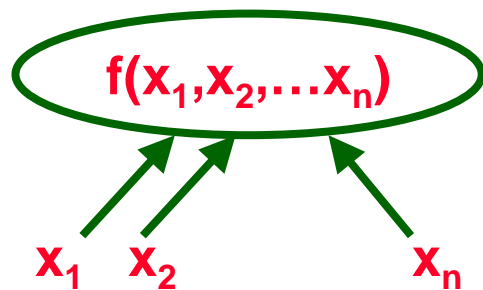
- Tree-based algorithm
  - May vary branching factor
  - $O(\log P)$  algorithm



- If broadcasting large data blocks, may break into pieces and pipeline

# Lower Bound on Parallel Performance

- To compute a function of  $n$  inputs  $x_1, \dots, x_n$
  - Given only binary operations on our machine.
    - In 1 time step, output depends on at most 2 inputs
    - In 2 time steps, output depends on at most 4 inputs
  - Adding a time step increases possible inputs by at most 2x
    - In  $k = \log n$  time steps, output depends on at most  $n$  inputs
- A function of  $n$  inputs requires at least  $\log n$  parallel steps.



## Scan (or Parallel prefix), A Digression

- What if you want to compute partial sums
- Definition: the **parallel prefix** operation take a **binary associative** operator  $\ominus$ , and an array of n elements

$$[a_0, a_1, a_2, \dots a_{n-1}]$$

and produces the array

$$[a_0, (a_0 \ominus a_1), \dots (a_0 \ominus a_1 \ominus \dots \ominus a_{n-1})]$$

- Example: add scan of

$$[1, 2, 0, 4, 2, 1, 1, 3] \text{ is } [1, 3, 3, 7, 9, 10, 11, 14]$$

- Can be implemented in  $O(n)$  time by a serial algorithm
  - **Obvious  $n-1$  applications of operator will work**

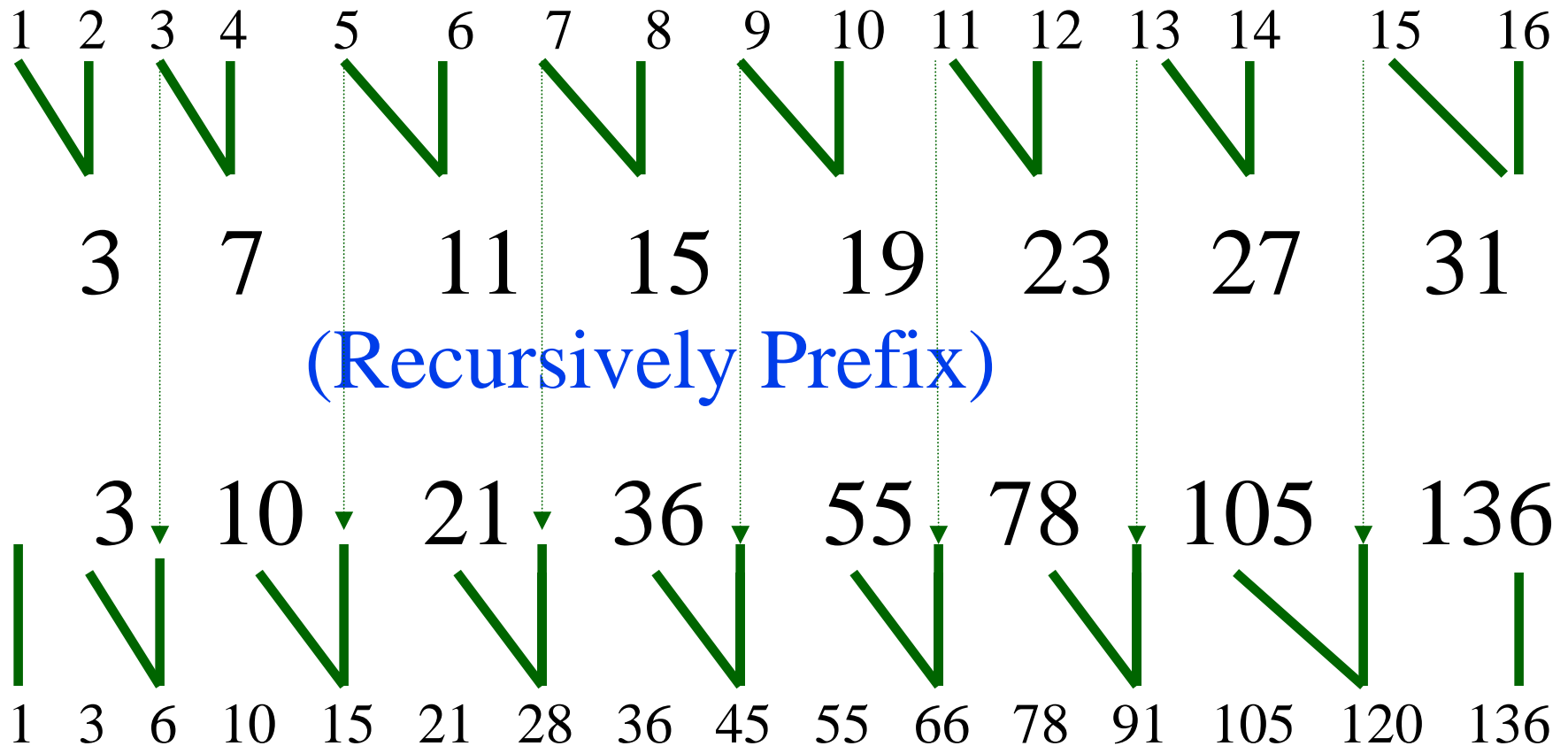
# Applications of scans

---

- There are several applications of scans, some more obvious than others
  - lexically compare string of characters
  - add multi-precision numbers (represented as array of numbers)
  - evaluate polynomials
  - implement bucket sort and radix sort
  - solve tridiagonal systems
  - to dynamically allocate processors
  - to search for regular expression (e.g., grep)

# Prefix Sum in parallel

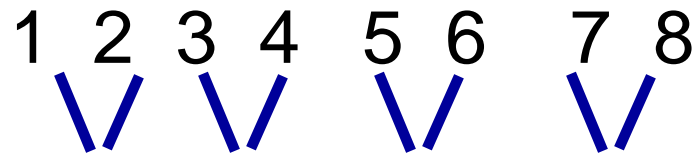
**Algorithm:** 1. Pairwise sum    2. Recursively Prefix    3. Pairwise Sum



# Parallel Prefix Cost

---

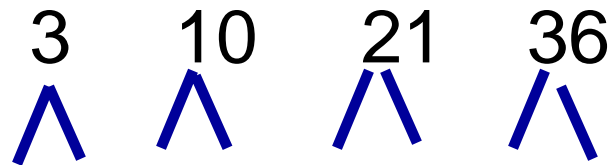
- Parallel prefix works on any **associative** operator



Pairwise sums



Recursive prefix



Update "odds"

1 3 6 10 15 21 28 36

- Names: +\ (APL), cumsum(Matlab), MPI\_SCAN
- Warning:  $2n$  operations used when only  $n-1$  needed

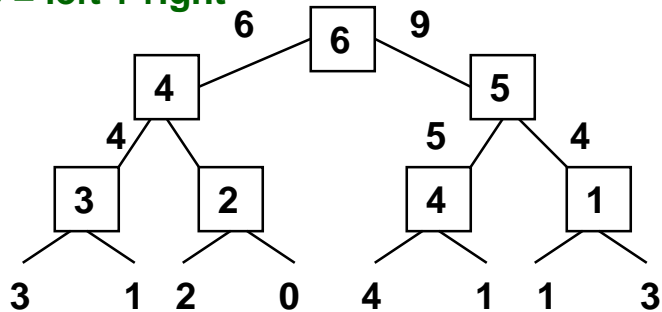
# Implementing Scans

- Tree summation 2 phases
  - up sweep
    - get values L and R from left and right child
    - save L in local variable Mine
    - compute  $Tmp = L + R$  and pass to parent
  - down sweep
    - get value Tmp from parent
    - send Tmp to left child
    - send  $Tmp + Mine$  to right child

Up sweep:

mine = left

tmp = left + right

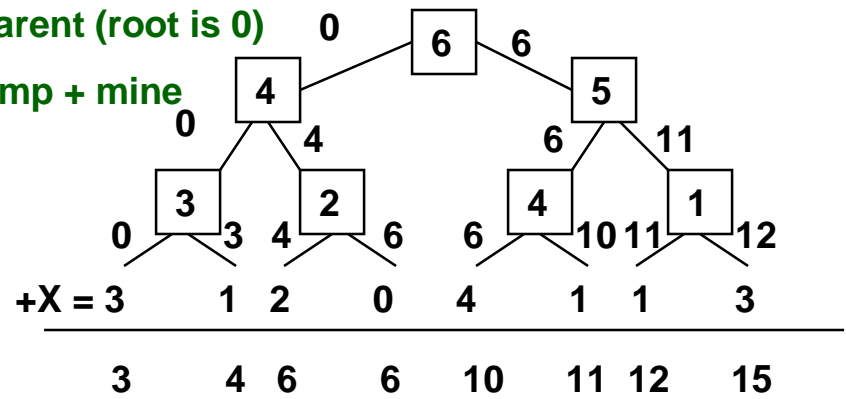


3/9/2004

Down sweep:

tmp = parent (root is 0)

right = tmp + mine



CS267 Lecure 13

11

## E.g., Using Scans for Array Compression

- Given an array of  $n$  elements

$[a_0, a_1, a_2, \dots, a_{n-1}]$

and an array of flags

$[1, 0, 1, 1, 0, 0, 1, \dots]$

compress the flagged elements

$[a_0, a_2, a_3, a_6, \dots]$

- Compute a “prescan” i.e., a scan that doesn’t include the element at position  $i$  in the sum  
 $[0, 1, 1, 2, 3, 3, 4, \dots]$
- Gives the index of the  $i^{\text{th}}$  element in the compressed array
  - If the flag for this element is 1, write it into the result array at the given position

## E.g., Fibonacci via Matrix Multiply Prefix

---

$$F_{n+1} = F_n + F_{n-1}$$

$$\begin{pmatrix} F_{n+1} \\ F_n \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix}$$

Can compute all  $F_n$  by `matmul_prefix` on

$$\left[ \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \right]$$

then select the upper left entry

# Segmented Operations

Inputs = Ordered Pairs  
(operand, boolean)

e.g. (x, T) or (x, F)

Change of  
segment indicated  
by switching T/F

$+_2$	(y, T)	(y, F)
(x, T)	(x+y, T)	(y, F)
(x, F)	(y, T)	(x $\oplus$ y, F)

e. g.	1	2	3	4	5	6	7	8
	T	T	F	F	F	T	F	T
Result	1	3	3	7	12	6	7	8

# The “Myth” of $\log n$

---

- The  $\log_2 n$  parallel steps is **not** the main reason for the usefulness of parallel prefix.

- **Say  $n = 1000p$**  (1000 summands per processor)

- Cost = (2000 adds) + ( $\log_2 P$  message passings)



fast & embarrassingly parallel

(2000 local adds are serial for each processor of course)

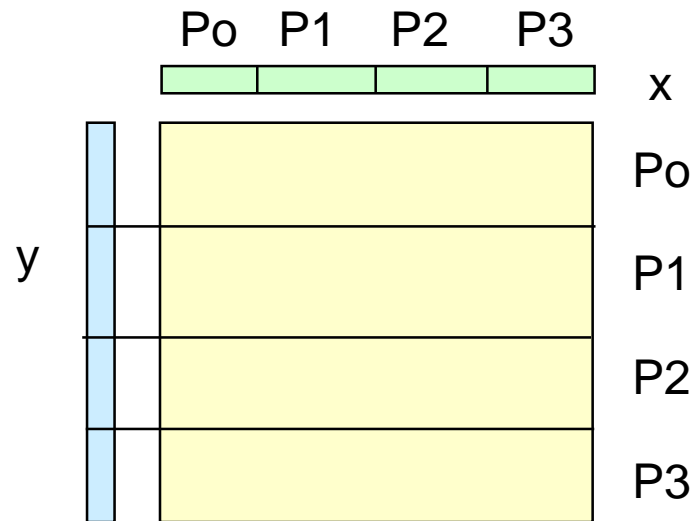
# End of Digression

---

- Summary of data parallel operations
  - Vector add, etc. is embarrassingly parallel
  - Broadcast used for axpy operations
  - Reduction used for dot product
  - Parallel prefix (scan) is a variation on reduction with partial results
    - Useful in parallelizing surprising algorithms
    - If something seems serial, try this
- Now back to our regular programming
  - We have covered the idea with most BLAS1 (vector) operations
  - Now onto vector/matrix (BLAS2) and matrix-matrix (BLAS3)

# Parallel Matrix-Vector Product

- Compute  $y = y + A*x$ , where  $A$  is a dense matrix



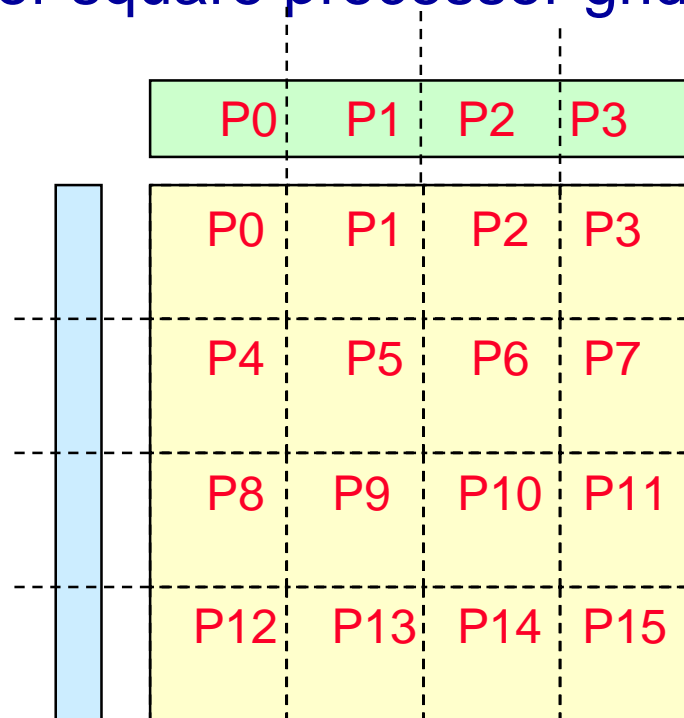
- Layout:
  - **1D by rows**
- Algorithm:
  - **Foreach processor  $i$**
  - **Broadcast  $x(i)$**
  - **Compute  $y(i) = A(i)*x$**
- $A(i)$  refers to the  $n$  by  $n/p$  block row that processor  $i$  owns,  $x(i)$  and  $y(i)$  similarly refer to segments of  $x, y$  owned by  $i$
- Algorithm uses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

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

# Matrix-Vector Product

---

- A **column layout** of the matrix eliminates the broadcast
  - But adds a reduction to update the destination
- A **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
  - Topology of machine
  - Scheduling communication
- Use of performance models for algorithm design
  - **Message Time = “latency” + #words \* time-per-word**  
 **$= \alpha + n*\beta$**

# Latency Bandwidth Model

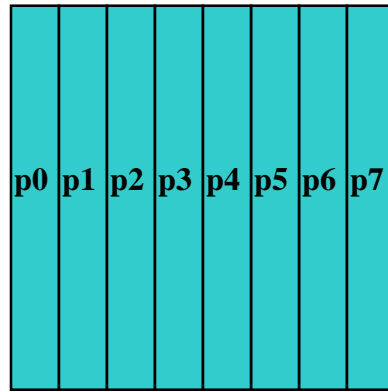
---

- Network of fixed number  $P$  of processors
  - fully connected
  - each with local memory
- Latency ( $\alpha$ )
  - accounts for varying performance with number of messages
  - gap ( $g$ ) in  $\log P$  model may be more accurate cost if messages are pipelined
- Inverse bandwidth ( $\beta$ )
  - accounts for performance varying with volume of data
- Efficiency (in any model):
  - serial time / ( $p$  \* parallel time)
  - perfect (linear) speedup  $\rightarrow$  efficiency = 1

# Matrix Multiply with 1D Column Layout

---

- Assume matrices are  $n \times n$  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 (similarly for  $B(i)$  and  $C(i)$ )
- $B(i,j)$  is the  $n/p$  by  $n/p$  subblock of  $B(i)$ 
  - in rows  $j*n/p$  through  $(j+1)*n/p$
- Algorithm uses the formula

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

## Matrix Multiply: 1D Layout on Bus or Ring

- Algorithm uses the formula

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

- 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:  $\alpha + \beta*n^2/p$  ... approximately

**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:**

**=  $(p*(p-1) + 1)*\text{computation} + p*(p-1)*\text{communication}$**

**$\sim 2*n^3 + p^2*\alpha + p*n^2*\beta$**

**this is worse than the serial time and grows with p**

# Matmul for 1D layout on a Processor Ring

- Pairs of processors can communicate simultaneously

Copy  $A(\text{myproc})$  into  $\text{Tmp}$

$C(\text{myproc}) = C(\text{myproc}) + \text{Tmp} * B(\text{myproc}, \text{myproc})$

for  $j = 1$  to  $p-1$

    Send  $\text{Tmp}$  to processor  $\text{myproc}+1 \bmod p$

    Receive  $\text{Tmp}$  from processor  $\text{myproc}-1 \bmod p$

$C(\text{myproc}) = C(\text{myproc}) + \text{Tmp} * B(\text{myproc}-j \bmod p, \text{myproc})$

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

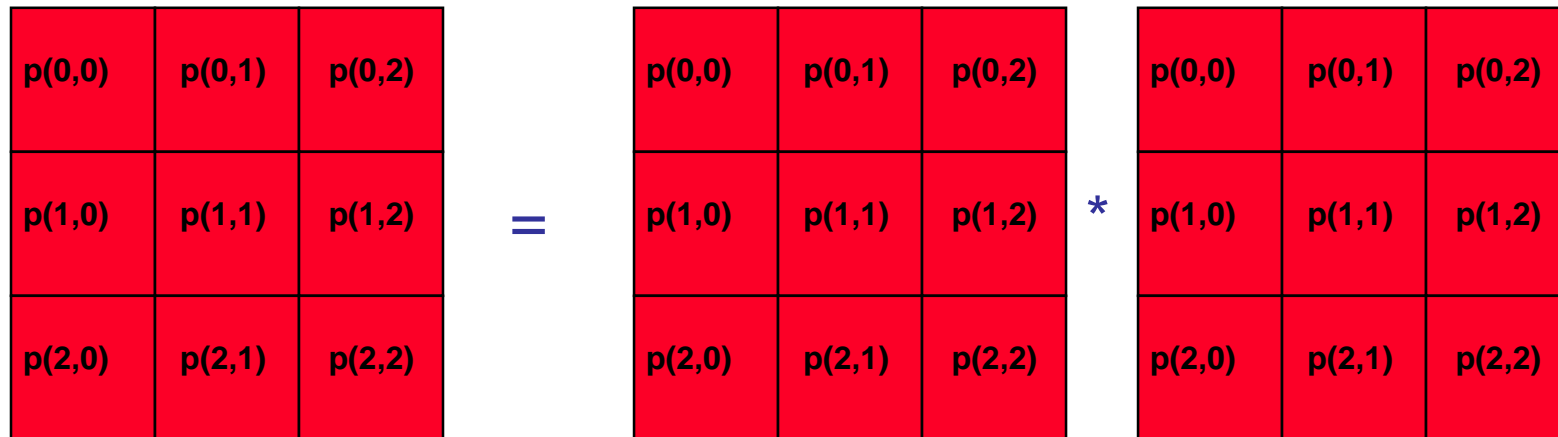
$(p-1)*\text{cost of sending } n*(n/p) \text{ words}$

- 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  $\alpha$  and  $\beta$  shrink)

# MatMul with 2D Layout

---

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



- Assume p is square s x s grid

# Cannon's Algorithm

---

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

... assume  $s = \text{sqrt}(p)$  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  $A(i,(j+i)\text{mod } s)$

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  $B((i+j)\text{mod } s), j)$

for  $k=0$  to  $s-1$  ... sequential

forall  $i=0$  to  $s-1$  and  $j=0$  to  $s-1$  ... all processors in parallel

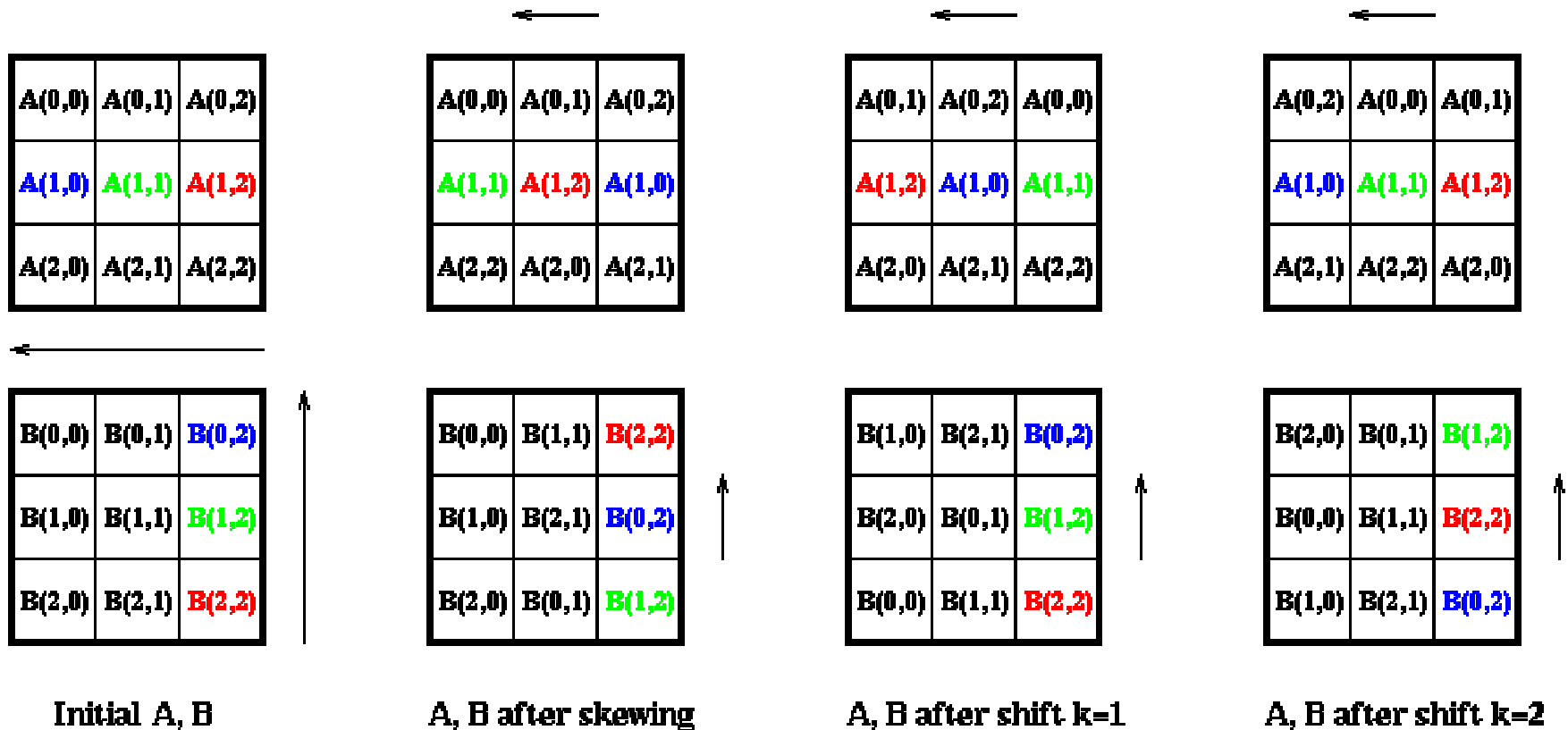
$C(i,j) = C(i,j) + A(i,j)*B(i,j)$

left-circular-shift each row of A by 1

up-circular-shift each row of B by 1

# Cannon's Matrix Multiplication

## Cannon's Matrix Multiplication Algorithm



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

# 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

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)

- Second

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

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

- Third

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

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

## 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 =  $s * (\alpha + \beta * n^2/p)$ 
forall i=0 to s-1
    up-circular-shift B column i of B by i ... cost =  $s * (\alpha + \beta * n^2/p)$ 
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 =  $2 * (n/s)^3 = 2 * n^3/p^{3/2}$ 
        left-circular-shift each row of A by 1 ... cost =  $\alpha + \beta * n^2/p$ 
        up-circular-shift each row of B by 1 ... cost =  $\alpha + \beta * n^2/p$ 

```

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

# Drawbacks to Cannon

---

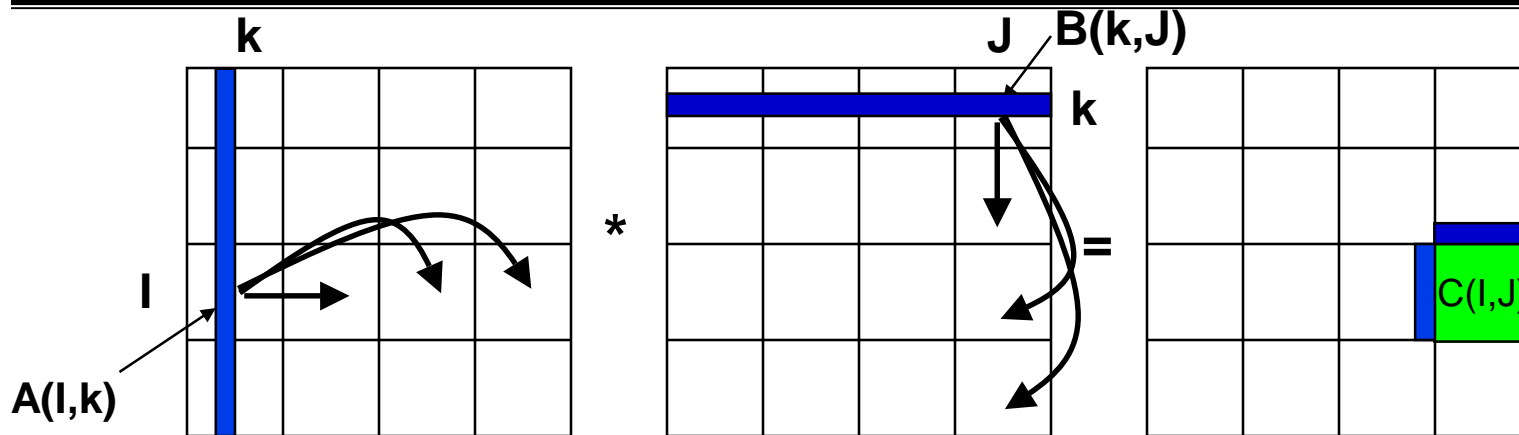
- Hard to generalize for
  - $p$  not a perfect square
  - $A$  and  $B$  not square
  - Dimensions of  $A$ ,  $B$  not perfectly divisible by  $s = \sqrt{p}$
  - $A$  and  $B$  not “aligned” in the way they are stored on processors
  - block-cyclic layouts
- Memory hog (extra copies of local matrices)

# SUMMA Algorithm

---

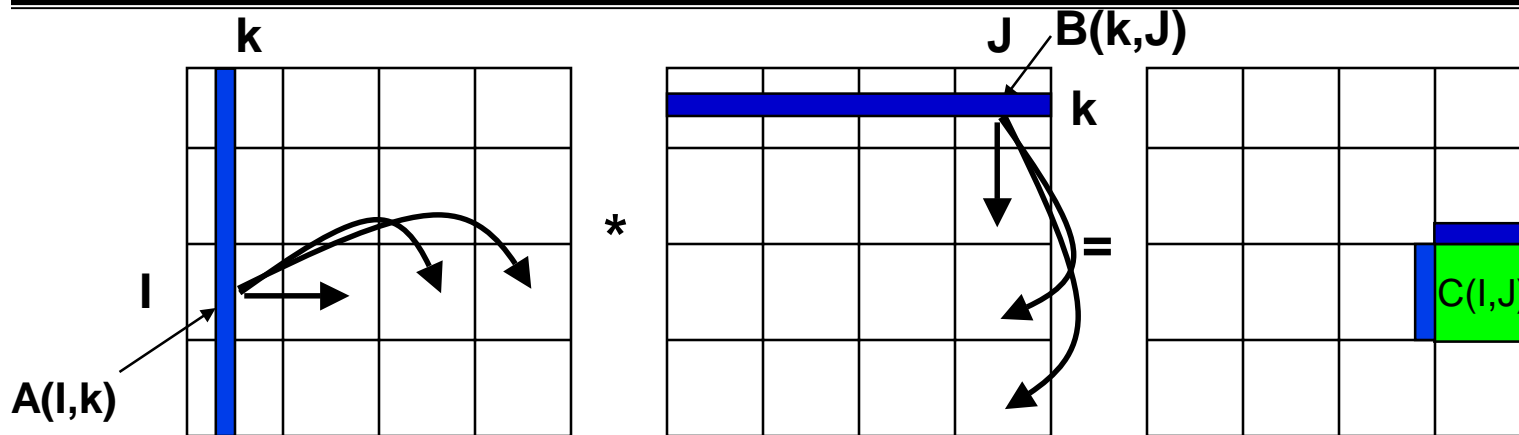
- SUMMA = Scalable Universal Matrix Multiply
- Slightly less efficient, but simpler and easier to generalize
- Presentation from van de Geijn and Watts
  - [www.netlib.org/lapack/lawns/lawn96.ps](http://www.netlib.org/lapack/lawns/lawn96.ps)
  - Similar ideas appeared many times
- Used in practice in PBLAS = Parallel BLAS
  - [www.netlib.org/lapack/lawns/lawn100.ps](http://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
- $C(I,J) = C(I,J) + \sum_k A(I,k)*B(k,J)$
- 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  $Brow$

$C(\text{myproc}, \text{myproc}) = C(\text{myproc}, \text{myproc}) + Acol * Brow$

# 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 =  $\log s * (\alpha + \beta * b * n/s)$ , using a tree

for all  $J = 1$  to  $s$

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

... time =  $\log s * (\alpha + \beta * b * n/s)$ , using a tree

Receive  $A(I,k)$  into  $Acol$

Receive  $B(k,J)$  into  $Brow$

$C(\text{myproc}, \text{myproc}) = C(\text{myproc}, \text{myproc}) + Acol * Brow$

... time =  $2 * (n/s)^2 * b$

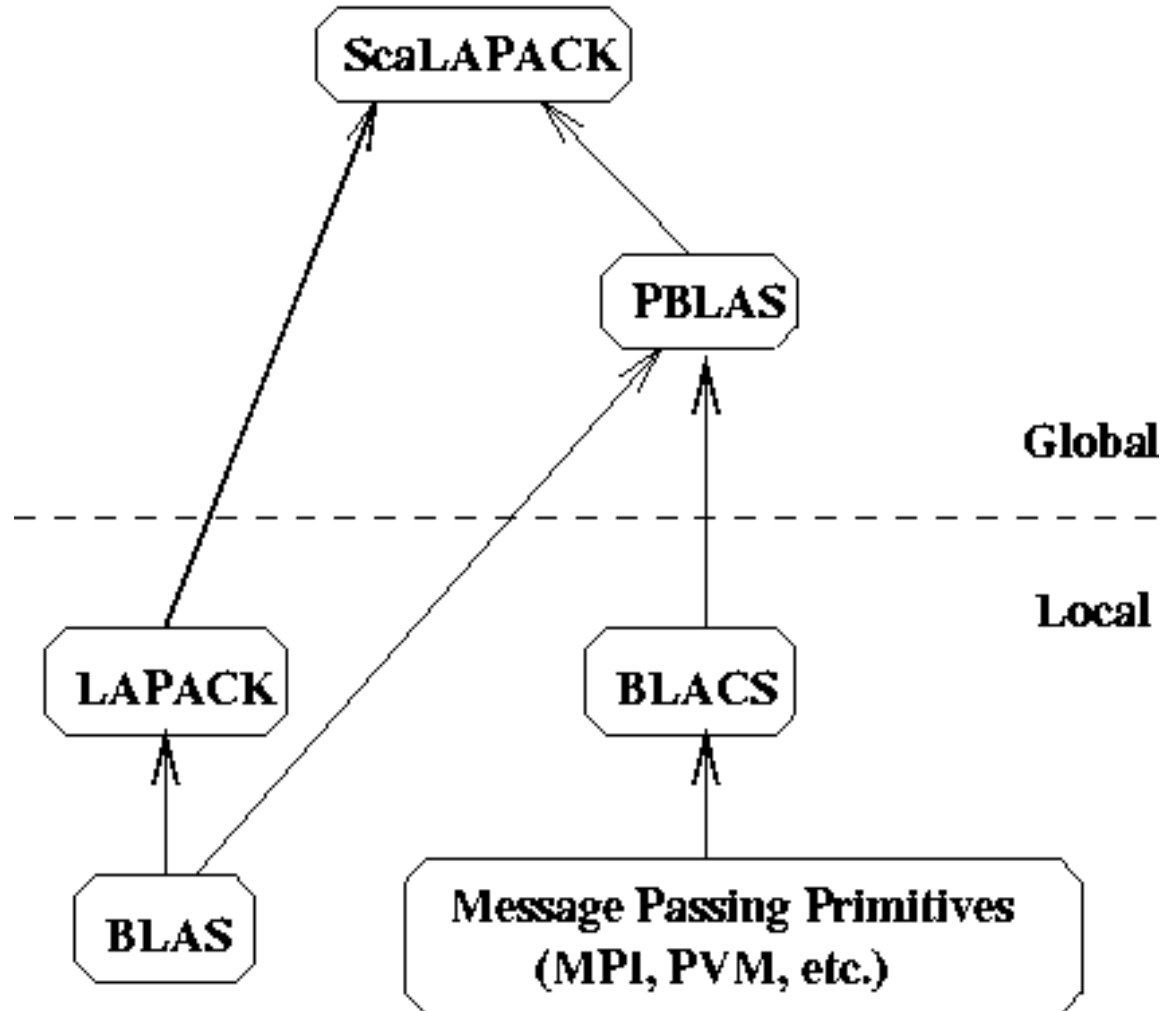
- Total time =  $2 * n^3/p + \alpha * \log p * n/b + \beta * \log p * n^2/s$

## SUMMA performance

- Total time =  $2 \cdot n^3 / p + \alpha \cdot \log p \cdot n / b + \beta \cdot \log p \cdot n^2 / s$
- Parallel Efficiency =  
$$1 / (1 + \alpha \cdot \log p \cdot p / (2 \cdot \beta \cdot n^2) + \beta \cdot \log p \cdot s / (2 \cdot n))$$
- ~Same  $\beta$  term as Cannon, except for  $\log p$  factor  
     $\log p$  grows slowly so this is ok
- Latency ( $\alpha$ ) term can be larger, depending on  $b$   
    When  $b=1$ , get  $\alpha \cdot \log p \cdot n$   
    As  $b$  grows to  $n/s$ , term shrinks to  
         $\alpha \cdot \log p \cdot s$  ( $\log p$  times Cannon)
- Temporary storage grows like  $2 \cdot b \cdot n / s$
- Can change  $b$  to tradeoff latency cost with memory

# ScaLAPACK Parallel Library

## ScaLAPACK SOFTWARE HIERARCHY



## Performance of PBLAS

Speed in Mflops of PDGEMM					
Machine	Procs	Block Size	N		
			2000	4000	10000
Cray T3E	4=2x2	32	1055	1070	0
	16=4x4		3630	4005	4292
	64=8x8		13456	14287	16755
IBM SP2	4	50	755	0	0
	16		2514	2850	0
	64		6205	8709	10774
Intel XP/S MP Paragon	4	32	330	0	0
	16		1233	1281	0
	64		4496	4864	5257
Berkeley NOW	4	32	463	470	0
	32=4x8		2490	2822	3450
	64		4130	5457	6647

**PDGEMM = PBLAS routine for matrix multiply**

**Observations:**

**For fixed N, as P increases Mflops increases, but less than 100% efficiency  
For fixed P, as N increases, Mflops (efficiency) rises**

**DGEMM = BLAS routine for matrix multiply**

**Maximum speed for PDGEMM = # Procs \* speed of DGEMM**

**Observations (same as above):  
Efficiency always at least 48%  
For fixed N, as P increases, efficiency drops  
For fixed P, as N increases, efficiency increases**

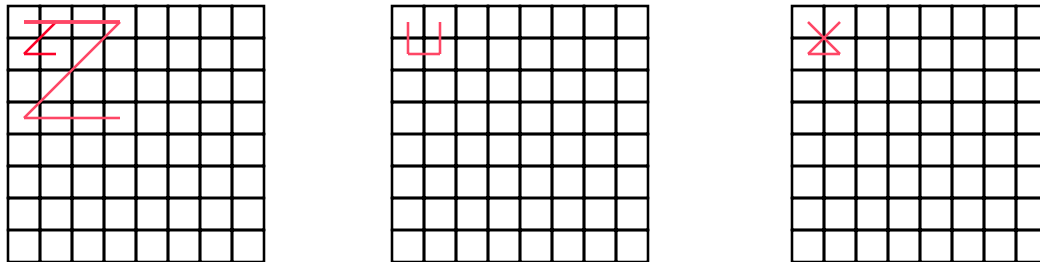
Efficiency = MFlops(PDGEMM)/(Procs*MFlops(DGEMM))						
Machine	Peak/proc	DGEMM Mflops	Procs	N		
				2000	4000	10000
Cray T3E	600	360	4	.73	.74	
			16	.63	.70	.75
			64	.58	.62	.73
IBM SP2	266	200	4	.94		
			16	.79	.89	
			64	.48	.68	.84
Intel XP/S MP Paragon	100	90	4	.92		
			16	.86	.89	
			64	.78	.84	.91
Berkeley NOW	334	129	4	.90	.91	
			32	.60	.68	.84
			64	.50	.66	.81

3/9/2004

# Recursive Layouts

---

- For both cache hierarchies and parallelism, recursive layouts may be useful
- Z-Morton, U-Morton, and X-Morton Layout



- Also Hilbert layout and others
- What about the user's view?
  - Fortunately, many problems can be solved on a permutation
  - Never need to actually change the user's layout

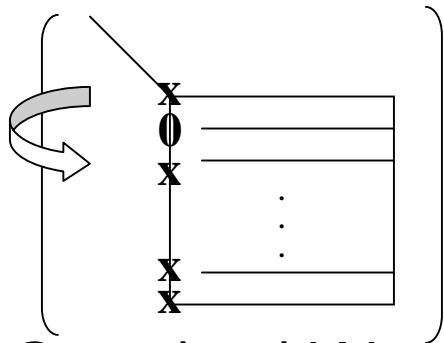
# Summary of Parallel Matrix Multiplication

- 1D Layout
  - Bus without broadcast - slower than serial
  - Nearest neighbor communication on a ring (or bus with broadcast): Efficiency =  $1/(1 + O(p/n))$
- 2D Layout
  - Cannon
    - Efficiency =  $1/(1+O(\sqrt{p})/n)$
    - Hard to generalize for general  $p$ ,  $n$ , block cyclic, alignment
  - SUMMA
    - Efficiency =  $1/(1 + O(\log p * p / (b*n^2) + \log p * \sqrt{p}) / n)$
    - Very General
    - $b$  small => less memory, lower efficiency
    - $b$  large => more memory, high efficiency
  - Recursive layouts

---

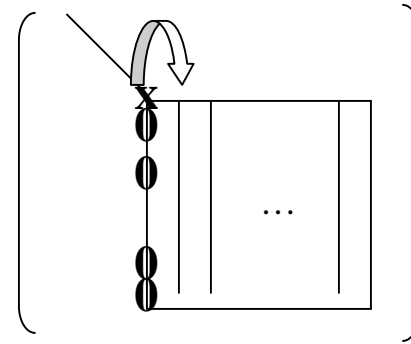
# Extra Slides

# Gaussian Elimination



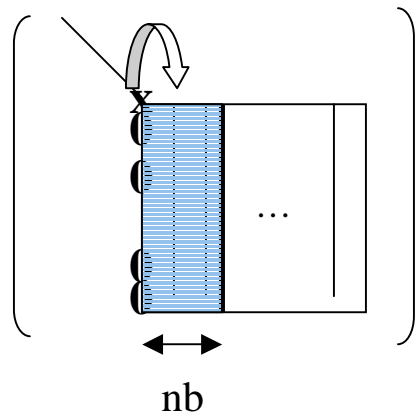
Standard Way

subtract a multiple of a row



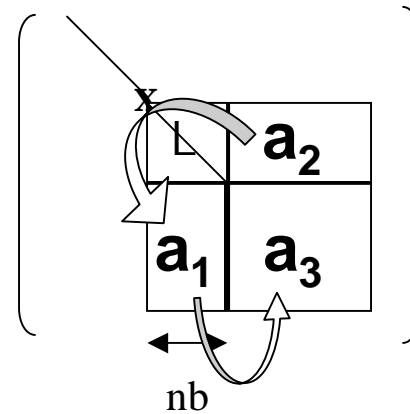
LINPACK

apply sequence to a column



apply sequence to nb

LAPACK



then apply nb to rest of matrix

$$\mathbf{a}_2 = \mathbf{L}^{-1} \mathbf{a}_2$$

$$\mathbf{a}_3 = \mathbf{a}_3 - \mathbf{a}_1^* \mathbf{a}_2$$

# Gaussian Elimination via a Recursive Algorithm

F. Gustavson and S. Toledo

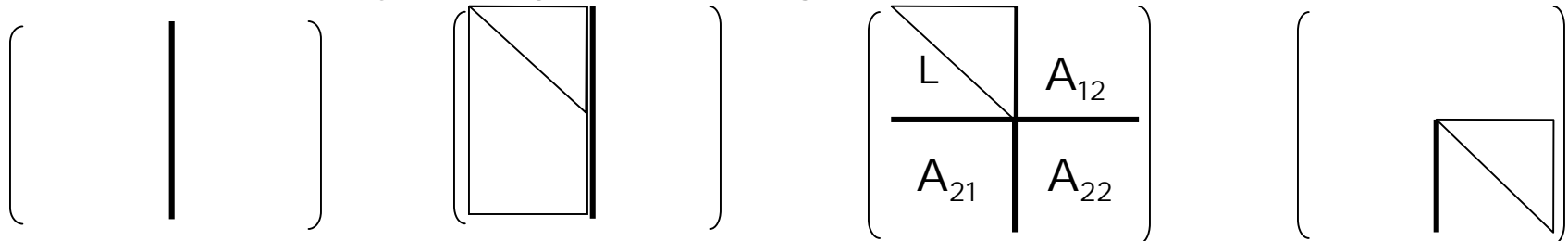
LU Algorithm:

1: Split matrix into two rectangles ( $m \times n/2$ )  
if only 1 column, scale by reciprocal of pivot & return

2: Apply LU Algorithm to the left part

3: Apply transformations to right part  
(triangular solve  $A_{12} = L^{-1}A_{12}$  and  
matrix multiplication  $A_{22} = A_{22} - A_{21} * A_{12}$ )

4: Apply LU Algorithm to right part



Most of the work in the matrix multiply  
Matrices of size  $n/2, n/4, n/8, \dots$

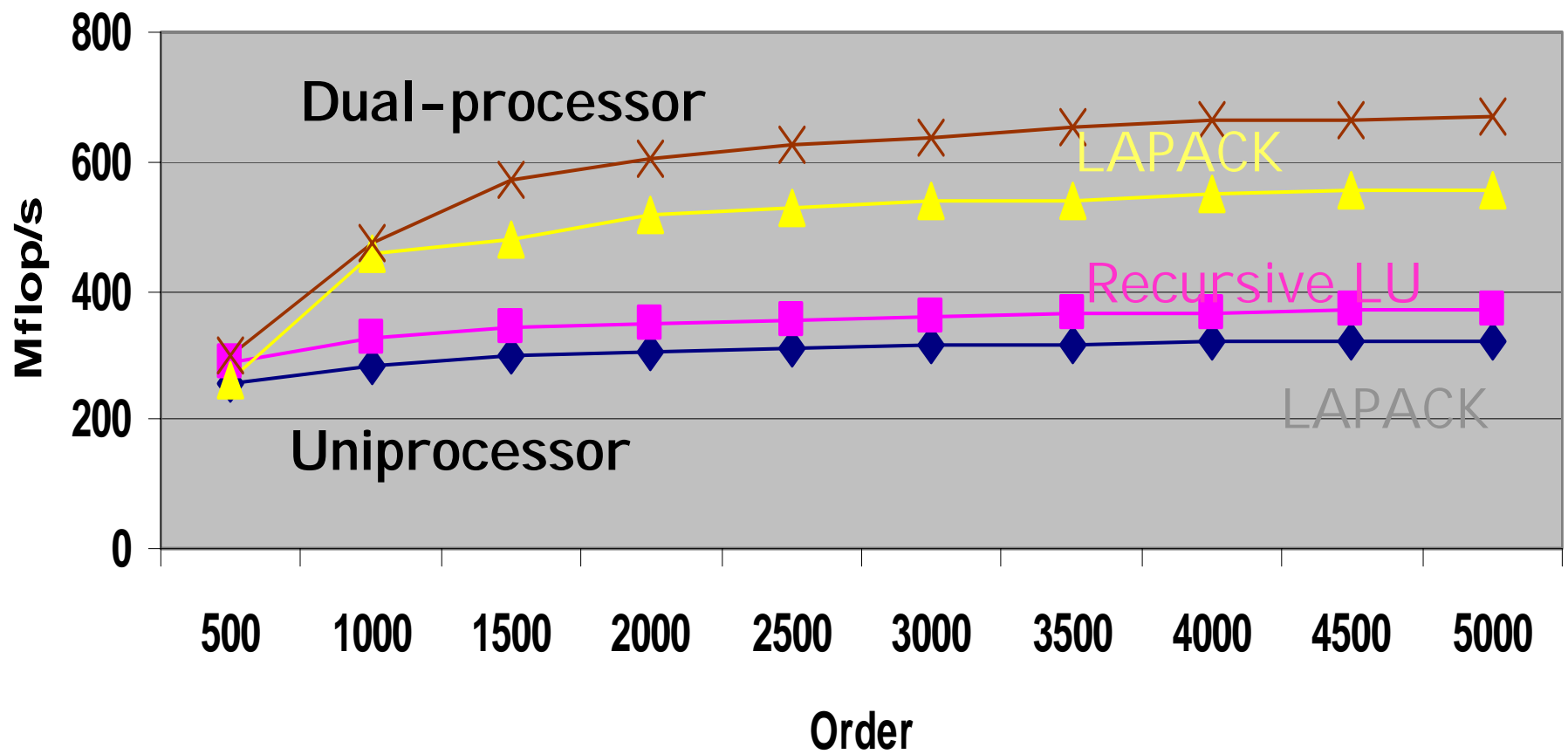
# Recursive Factorizations

---

- Just as accurate as conventional method
- Same number of operations
- Automatic variable blocking
  - **Level 1 and 3 BLAS only !**
- Extreme clarity and simplicity of expression
- Highly efficient
- The recursive formulation is just a rearrangement of the point-wise LINPACK algorithm
- The standard error analysis applies (assuming the matrix operations are computed the “conventional” way).

# Pentium III 550 MHz Dual Processor

## LU Factorization



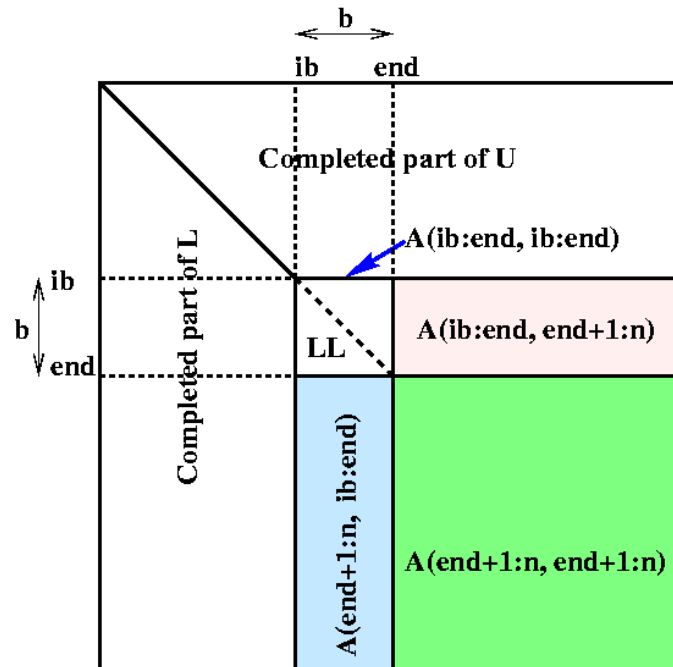
Slide source: Dongarra

# Review: BLAS 3 (Blocked) GEPP

```

for  ib = 1 to n-1 step b    ... Process matrix b columns at a time
    end = ib + b-1        ... Point to end of block of b columns
    apply BLAS2 version of GEPP to get  $A(ib:n, ib:end) = P' * L' * U'$ 
    ... let LL denote the strict lower triangular part of  $A(ib:end, ib:end) + I$ 
    {
    A(ib:end, end+1:n) =  $LL^{-1} * A(ib:end, end+1:n)$     ... update next b rows of U
    A(end+1:n, end+1:n) =  $A(end+1:n, end+1:n)$ 
      -  $A(end+1:n, ib:end) * A(ib:end, end+1:n)$ 
    ... apply delayed updates with single matrix-multiply
    ... with inner dimension b
    }
  
```

Gaussian Elimination using BLAS 3



# Review: Row and Column Block Cyclic Layout

---

bcol

brow

0	1	0	1	0	1	0	1
2	3	2	3	2	3	2	3
0	1	0	1	0	1	0	1
2	3	2	3	2	3	2	3
0	1	0	1	0	1	0	1
2	3	2	3	2	3	2	3
0	1	0	1	0	1	0	1
2	3	2	3	2	3	2	3

**processors and matrix blocks  
are distributed in a 2d array**

**pcol-fold parallelism  
in any column, and calls to the  
BLAS2 and BLAS3 on matrices of  
size brow-by-bcol**

4) Row and Column Block Cyclic Layout

**serial bottleneck is eased**

**need not be symmetric in rows and  
columns**

# Distributed GE with a 2D Block Cyclic Layout

---

---

**block size  $b$  in the algorithm and the block sizes  $b_{row}$  and  $b_{col}$  in the layout satisfy  $b=b_{row}=b_{col}$ .**

**shaded regions indicate busy processors or communication performed.**

**unnecessary to have a barrier between each step of the algorithm, e.g.. step 9, 10, and 11 can be pipelined**

## Distributed Gaussian Elimination with a 2D Block Cyclic Layout

for  $ib = 1$  to  $n-1$  step  $b$

$end = \min(ib+b-1, n)$

    for  $i = ib$  to  $end$

        (1) find pivot row  $k$ , column broadcast

        (2) swap rows  $k$  and  $i$  in block column, broadcast row  $k$

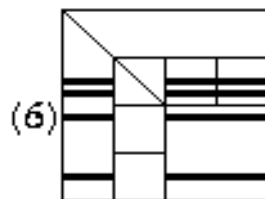
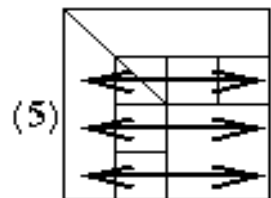
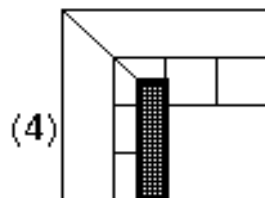
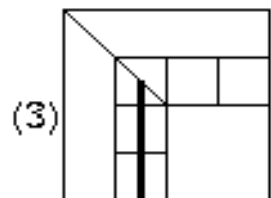
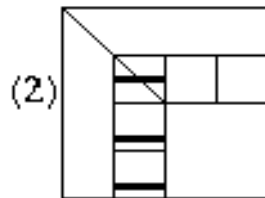
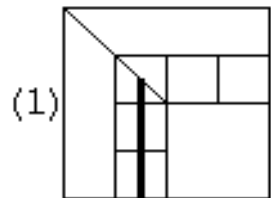
        (3)  $A(i+1:n, i) = A(i+1:n, i) / A(i, i)$

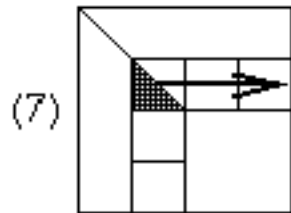
        (4)  $A(i+1:n, i+1:end) -= A(i+1:n, i) * A(i, i+1:end)$

    end for

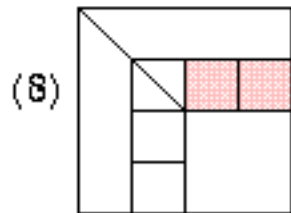
(5) broadcast all swap information right and left

(6) apply all rows swaps to other columns

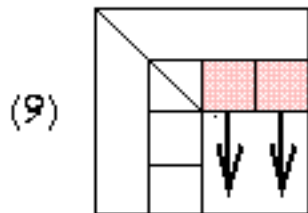




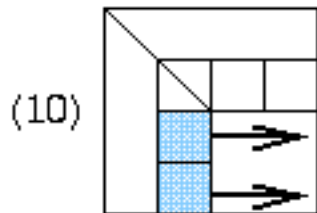
(7) Broadcast LL right



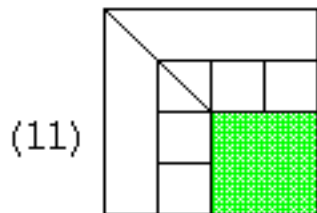
(8)  $A(ib:end, end+1:n) = LL \setminus A(ib:end, end+1:n)$



(9) Broadcast  $A(ib:end, end+1:n)$  down



(10) Broadcast  $A(end+1:n, ib:end)$  right



(11) Eliminate  $A(end+1:n, end+1:n)$

Matrix multiply of  
green = green - blue \* pink

## Performance of ScaLAPACK LU

**PDGESV = ScaLAPACK  
parallel LU routine**

Since it can run no faster than its  
inner loop (PDGEMM), we measure:  
**Efficiency =  
Speed(PDGESV)/Speed(PDGEMM)**

**Observations:**  
Efficiency well above 50% for large  
enough problems  
For fixed N, as P increases,  
efficiency decreases  
(just as for PDGEMM)  
For fixed P, as N increases  
efficiency increases  
(just as for PDGEMM)  
From bottom table, cost of solving  
 $Ax=b$  about half of matrix multiply  
for large enough matrices.  
From the flop counts we would  
expect it to be  $(2*n^3)/(2/3*n^3) = 3$   
times faster, but communication  
makes it a little slower.

Efficiency = MFlops(PDGESV)/MFlops(PDGEMM)					
Machine	Procs	Block Size	N		
			2000	4000	10000
Cray T3E	4	32	.67	.82	
	16		.44	.65	.84
	64		.18	.47	.75
IBM SP2	4	50	.56		
	16		.29	.52	
	64		.15	.32	.66
Intel XP/S MP Paragon	4	32	.64		
	16		.37	.66	
	64		.16	.42	.75
Berkeley NOW	4	32	.76		
	32		.38	.62	.71
	64		.28	.54	.69

Time(PDGESV)/Time(PDGEMM)					
Machine	Procs	Block Size	N		
			2000	4000	10000
Cray T3E	4	32	.50	.40	
	16		.75	.51	.40
	64		1.86	.72	.45
IBM SP2	4	50	.60		
	16		1.16	.64	
	64		2.24	1.03	.51
Intel XP/S GP Paragon	4	32	.52		
	16		.89	.50	
	64		2.08	.79	.44
Berkeley NOW	4	32	.44		
	32		.88	.54	.47
	64		1.18	.62	.49

---

## LAPACK and ScaLAPACK

	LAPACK	ScaLAPACK
<b>Machines</b>	<b>Workstations, Vector, SMP</b>	<b>Distributed Memory, DSM</b>
<b>Based on</b>	<b>BLAS</b>	<b>BLAS, BLACS</b>
<b>Functionality</b>	<b>Linear Systems Least Squares Eigenproblems</b>	<b>Linear Systems Least Squares Eigenproblems (less than LAPACK)</b>
<b>Matrix types</b>	<b>Dense, band</b>	<b>Dense, band, out-of-core</b>
<b>Error Bounds</b>	<b>Complete</b>	<b>A few</b>
<b>Languages</b>	<b>F77 or C</b>	<b>F77 and C</b>
<b>Interfaces to</b>	<b>C++, F90</b>	<b>HPF</b>
<b>Manual?</b>	<b>Yes</b>	<b>Yes</b>
<b>Where?</b>	<b><a href="http://www.netlib.org/lapack">www.netlib.org/ lapack</a></b>	<b><a href="http://www.netlib.org/scalapack">www.netlib.org/ scalapack</a></b>

## Performance of ScaLAPACK QR (Least squares)

**Scales well,  
nearly full machine speed**

Efficiency = MFlops(PDGELS)/MFlops(PDGEMM)					
Machine	Procs	Block Size	N		
			2000	4000	10000
Cray T3E	4	32	.54	.61	
	16		.46	.55	.60
	64		.26	.47	.54
IBM SP2	4	50	.51		
	16		.29	.51	
	64		.19	.36	.54
Intel XP/S GP Paragon	4	32	.61		
	16		.43	.63	
	64		.22	.48	.62
Berkeley NOW	4	32	.51	.77	
	32		.49	.66	.71
	64		.37	.60	.72

Time(PDGELS)/Time(PDGEMM)					
Machine	Procs	Block Size	N		
			2000	4000	10000
Cray T3E	4	32	1.2	1.1	
	16		1.5	1.2	1.1
	64		2.6	1.4	1.2
IBM SP2	4	50	1.3		
	16		2.3	1.3	
	64		3.6	1.8	1.2
Intel XP/S GP Paragon	4	32	1.1		
	16		1.6	1.1	
	64		3.0	1.4	1.1
Berkeley NOW	4	32	1.3	.9	
	32		1.4	1.0	.9
	64		1.8	1.1	.9

## Performance of Symmetric Eigensolvers

Old version,  
pre 1998 Gordon Bell Prize

Still have ideas to accelerate  
Project Available!

Time(PDSYEVX)/Time(PDGEMM) (bisection + inverse iteration)				
Machine	Procs	Block Size	N	
			2000	4000
Cray T3E	4	32	10	
	16		13	10
	64		29	14
IBB SP2	16	50	24	
	64		40	29
Intel XP/S GP Paragon	16	32	22	
	64		34	20
Berkeley NOW	16	32	20	
	32		24	52

Old Algorithm,  
plan to abandon

Time(PDSYEV)/Time(PDGEMM) (QR iteration)				
Machine	Procs	Block Size	N	
			2000	4000
Cray T3E	4	32	35	
	16		37	35
	64		57	41
IBM SP2	16	50	38	
	64		58	47
Intel XP/S GP Paragon	16	32	99	
	64		193	
Berkeley NOW	16	32	31	
	32		35	55

## Performance of SVD (Singular Value Decomposition)

**Have good ideas to speedup  
Project available!**

Time(PDGESVD)/Time(PDGEMM)				
Machine	Procs	Block Size	N	
			2000	4000
Cray T3E	4	32	67	
	16		66	64
	64		93	70
IBM SP2	4	50	97	
	16		60	
	64		81	
Berkeley NOW	4	32	72	
	16		38	16
	32		59	26

## Performance of Nonsymmetric Eigensolver (QR iteration)

**Hardest of all to parallelize  
Have alternative, and  
would like to compare  
Project available!**

Time(PDLAHQR)/Time(PDGEMM)				
Machine	Procs	Block Size	N	
			1000	1500
Intel XP/S MP Paragon	16	50	123	97

## Out-of-Core Performance Results for Least Squares

- **Prototype code for Out-of-Core extension**
- **Linear solvers based on “Left-looking” variants of LU, QR, and Cholesky factorization**
- **Portable I/O interface for reading/writing ScaLAPACK matrices**

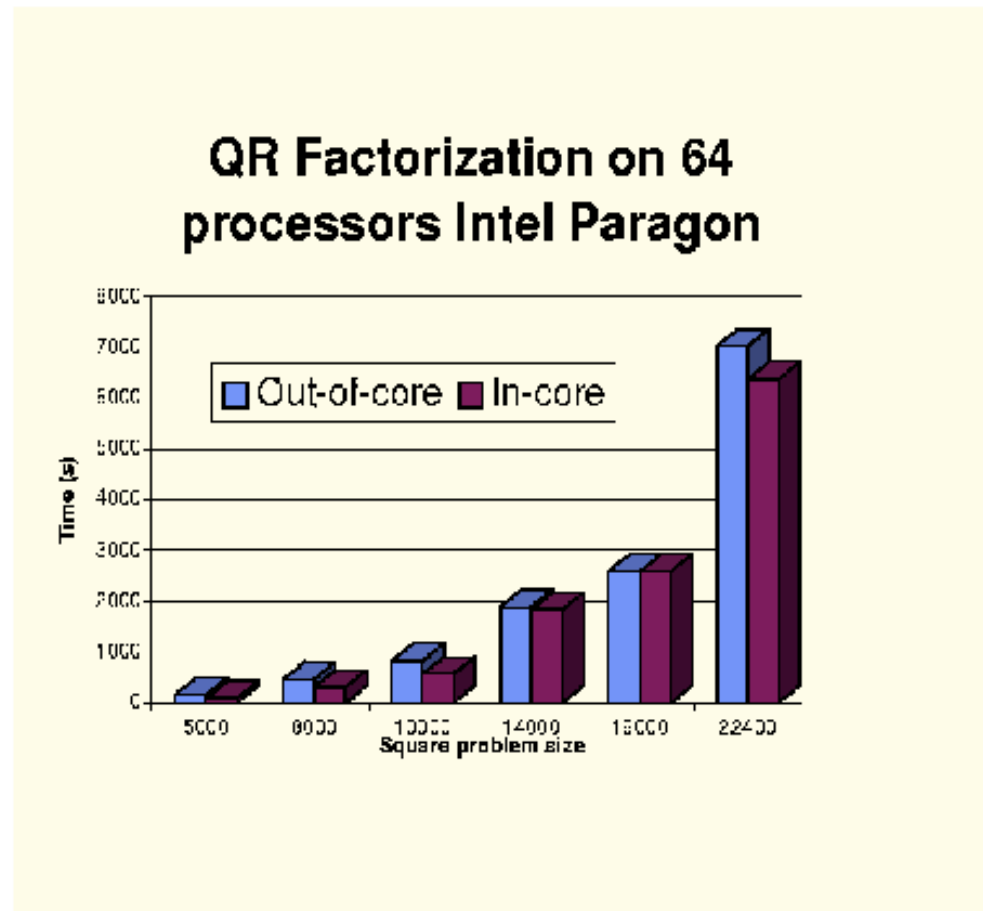
**Out-of-core means  
matrix lives on disk;  
too big for main mem**

**Much harder to hide  
latency of disk**

**QR much easier than LU  
because no pivoting  
needed for QR**

**Moral: use QR to solve  $Ax=b$**

**Projects available  
(perhaps very hard...)**



# A small software project ...

---

## Participants

Krste Asanovic (UC Berkeley)	Zhaojun Bai (U Kentucky)
Richard Barrett (U. Tenn)	Michael Berry (U Tenn)
Jeff Bilmes (UC Berkeley)	Chris Bischof (ANL)
Susan Blackford (ORNL)	Soumen Chakrabarti (UC Berkeley)
Tony Chan (UCLA)	Chee-Whye Chin (UC Berkeley)
Jaeyoung Choi (LBNL)	Andy Cleary (LLNL)
Ed D'Azeveda (ORNL)	Jim Demmel (UC Berkeley)
Inderjit Dhillon (UC Berkeley)	June Donato (ORNL)
Jack Dongarra (U Tenn, ORNL)	Zlatka Drmaš (U Hagen)
Jeremy Du Croz (NAG)	Victor Eijkhout (UCLA)
Stan Eisenstat (Yale)	Vince Fernando (NAG)
Jahn Gilbert (Xerox PARC)	Ming Gu (UC Berkeley, LBL)
Sven Hammarling (NAG)	Mike Heath (U Illinois)
Greg Henry (Intel)	Dominic Lam (UC Berkeley)
Steve Huss-Lederman (SRC)	Bo Kågström (U Umeå)
W. Kahan (UC Berkeley)	Youngbae Kim (U Tenn)
Rencang Li (UC Berkeley)	Xiaoye Li (UC Berkeley)
Joseph Liu (York)	Beresford Parlett (UC Berkeley)
Antoine Petitot (U Tenn)	Peter Pormaa (U Umeå)
Roldan Pozo (U Tenn)	Padma Raghavan (U Illinois)
Huan Ren (UC Berkeley)	Howard Robinson (UC Berkeley)
Charles Romine (ORNL)	Jeff Rutter (UC Berkeley)
Ivan Slapničar (U Split)	Dan Sorensen (Rice U)
Ken Stanley (UC Berkeley)	Xiaobai Sun (ANL)
Bernard Tourancheau (U Tenn)	Anna Tsao (SRC)
Robert van de Geijn (U Texas)	Henk van der Vorst (Utrecht U)
Paul Van Dooren (U Illinois)	Krešimir Veselić (U Hagen)
David Walker (ORNL)	Clint Whaley (U Tenn)
Kathy Yelick (UC Berkeley)	

With the cooperation of  
Cray, IBM, Convex, DEC, Fujitsu, NEC, NAG, IMSL

# Work-Depth Model of Parallelism

- The work depth model:
  - The simplest model is used
  - For algorithm design, independent of a machine
- The **work**,  $W$ , is the total number of operations
- The **depth**,  $D$ , is the longest chain of dependencies
- The parallelism,  $P$ , is defined as  $W/D$
  
- Specific examples include:
  - circuit model, each input defines a graph with ops at nodes
  - vector model, each step is an operation on a vector of elements
  - language model, where set of operations defined by language