CSE621

Parallel Algorithms
Lecture 4

Matrix Operation

September 20, 1999

Overview

- Review of the previous lecture
- Parallel Prefix Computations
- Parallel Matrix-Vector Product
- Parallel Matrix Multiplication
- Pointer Jumping
- Summary

Review of the previous lecture

- **Sorting on 2-D : n-step algorithm**
- **Sorting on 2-D : 0-1 sorting lemma**
  - **Proof of correctness and time complexity**
- **Sorting on 2-D : \root(n)(log n + 1)-step algorithm**
  - **Shear sort**
- **Sorting on 2-D : 3\root(n) + o(\root(n)) algorithm**
  - **Reducing dirty region**
- **Sorting : Matching lower bound**
  - **3\root(n) - o(\root(n))**
- **Sorting on 2-D : word-model vs. bit-model**

Parallel Prefix

- A primitive operation
- prefix computations: $x_1 \blacktriangle x_2 \blacktriangle \dots \blacktriangle x_i$, $i=1, \dots, n$ where $\blacktriangle$ is any associative operation on a set X.
- Used on applications such as carry-lookahead addition, polynomial evaluation, various circuit design, solving linear recurrences, scheduling problems, a variety of graph theoretic problems.
- For the purpose of discussion,
  - identity element exists
  - operator is an addition
  - S_{ij} denote the sum $x_i + x_{i+1} + \dots + x_j$, $I <= j$

Parallel Prefix : PRAM

- Based on parallel binary fan-in method (used by MinPRAM)
- Use a recursive doubling
- Assume that the elements x_1, x_2, …, x_n resides in the array X[0:n] where X[i]=x_i.
- Algorithm
  - In the first parallel step, P_i reads X[i-1] and X[i] and assigns the result to Prefix[i].
  - In the next parallel step, P_i reads Prefix[i-2] and Prefix[i], computes Prefix[i-2]+Prefix[i], and assigns the result to Prefix[i]
  - Repeat until m = log n steps.
- See Figure 11.1

procedure *PrefixPRAM*($X[1{:}n]$,*Prefix*$[1{:}n]$)

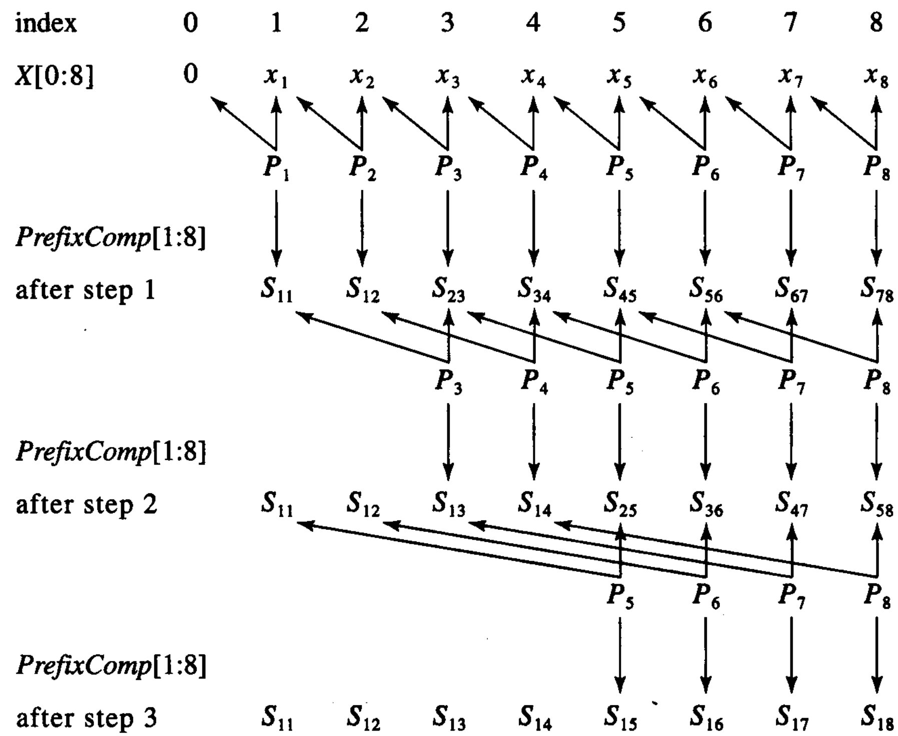
Model: EREW PRAM **with** $p = n$ **processors**

Input: $X[0{:}n]$ (an array of elements $x_1, x_2, \ldots, x_n$) $\{X[0] = 0, n = 2^m\}$

Output: *Prefix*$[1{:}n]$ (*Prefix*$[i] = x_1 \oplus \cdots \oplus x_i$, $i = 1, \ldots, n$)

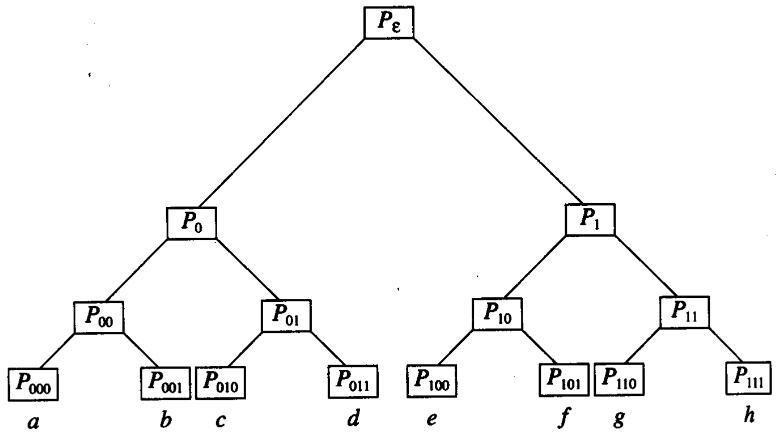
for $1 \le i \le n$ **do in parallel**
Prefix$[i] := X[i-1] \oplus X[i]$
end in parallel
$k := 2$
while $k < n$ **do**
for $k+1 \le i \le n$ **do in parallel**
Prefix$[i] :=$ *Prefix*$[i-k] \oplus$ *Prefix*$[i]$
end in parallel
$k := k + k$
endwhile

end *PrefixPRAM*

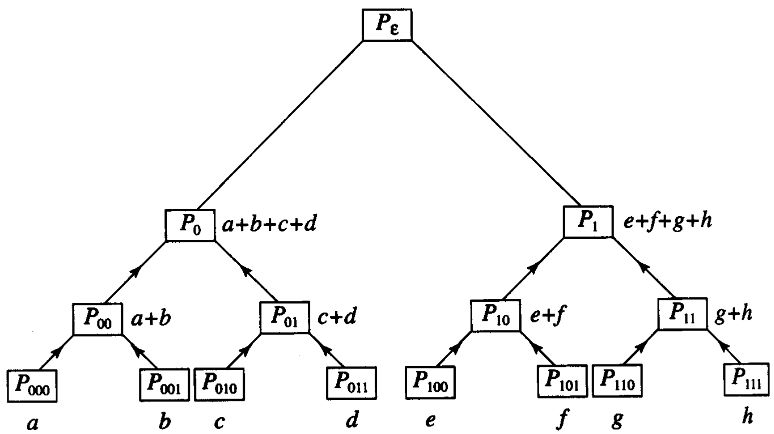


Parallel Prefix : On the complete binary tree

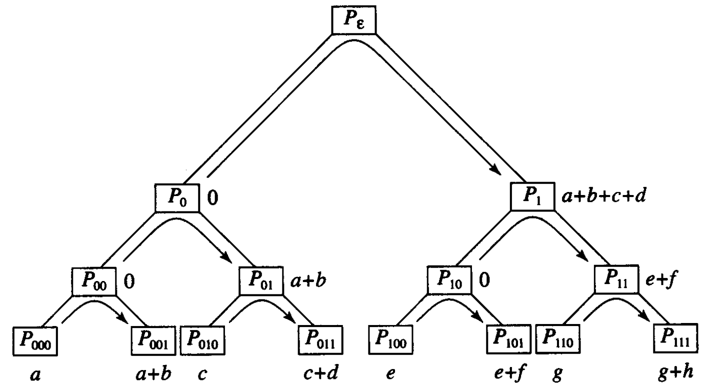
- **Assume that n operands are input to the leaves of the complete binary tree**
- **Algorithm**
  - **Phase1: binary fan-in computations are performed starting at the leaves and working up to the processors P_0 and P_1 at level one.**
  - **Phase2: for each pair of operands x_i, x_{i+1} in leaf nodes having the same parent, we replace the operand x_{i+1} in the right child by x_i+x_{i+1}.**
  - **Phase3: each right child that is not a leaf node replaces its binary fan-in computation with that of its sibling (left child), and the sibling replaces its binary fan-in computation with the identity element.**
  - **Phase 4: binary fan-in computations are performed as follows. Starting with the processors at level one and working our way down level by level to the leaves, a given processor communicates its element to both its children, and then each child adds the parent value to its value.**
- **See the figure**
- **Time: Phase1 : log n - 1, Phase2: 2 , Phase 3: 2, Phase 4: log n - 1**



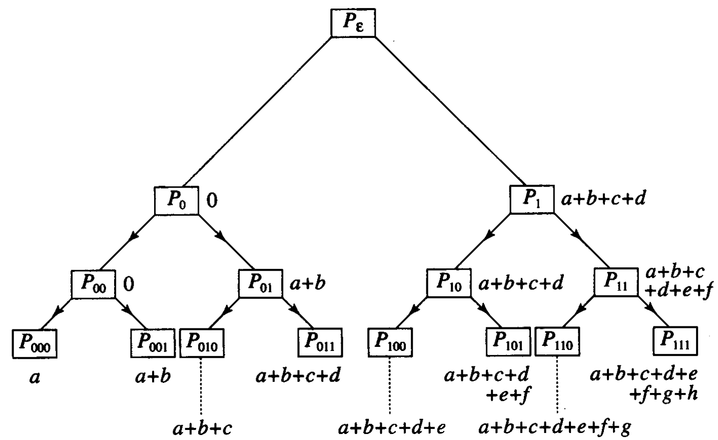
(a) Phase 1: Input the numbers in the leaves of PT_{2n-1}.



(b) Compute binary fan-in sums.



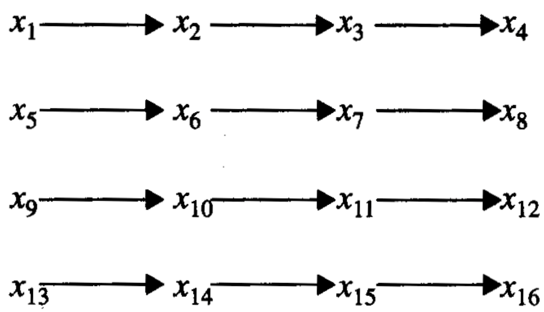
(c) Phase 2: For leaves, add sums in siblings and leave resulting sum in right child sibling. Phase 3: For non-root, non-leaf, left children, transfer binary fan-in sum to sibling then zero out own sum.



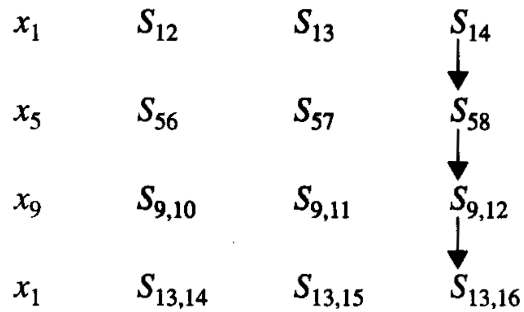
(d) Phase 4: Compute binary fan-out sums. Parallel prefix sums now reside in leaves.

Parallel Prefix : 2-D Mesh

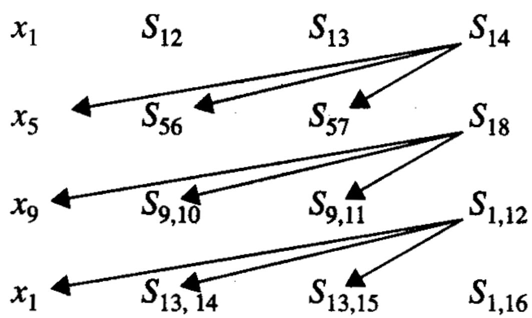
- **2-D Mesh $M_{q,q}$, $n = q*q$**
- **Elements are stored in row-major order in the distributed variable Prefix.**
- **Algorithm**
  - **Phase 1: consists of q-1 parallel steps where in the jth step column j of Prefix is added to column j+1.**
  - **Phase 2: consists of q-1 steps, where in the ith step $P_{i,q}$:prefix is communicated to processor $P_{i+1,q}$ and is then added to $P_{i+1,q}$:Prefix, $i=1,\ldots,q-1$**
  - **Phase 3: we add the value $P_{i-1,q}$:prefix to $P_{i,j}$ thereby obtaining the desired prefix sum $S_{1,(i-1)q+j}$ in $P_{i,j}$:Prefix, $i=2,\ldots,q$**
- **Time : 3*q steps**



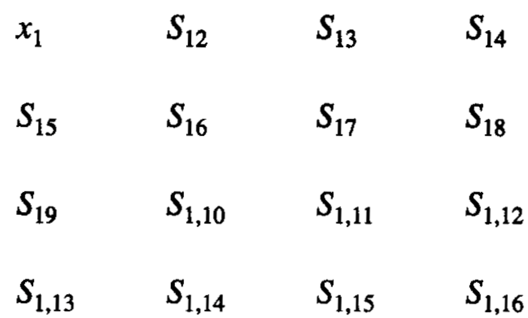
(a) Initial contents of *Prefix* and direction for performing the summing in phase 1.



(b) Contents of *Prefix* after phase 1 and direction for performing the summing in phase 2.



(c) Contents of *Prefix* after phase 2. Arrow from processor $P_{i-1,q}$ to P_{ij} indicates that $\mathbf{P}_{ij}$:*Prefix* should be replaced with $\mathbf{P}_{ij}$:*Prefix* $\oplus$ $\mathbf{P}_{i-1,q}$: *Prefix* in phase 3. This can be accomplished in q parallel steps.



(d) Contents of *Prefix* upon completion.

Parallel Prefix : Carry-Lookahead Addition

- When add two binary numbers, carry propagation is the delaying part.
- Three states
  - Stop Carry State {s}
  - Generate Carry State {r}
  - Propagate Carry State {p}
- Prefix operation determines the next carry
- Definition of prefix operation on {s, r, p}
- Carry-Lookahead algorithm
  - Find a carry state
  - Find a parallel prefix
  - Find a binary modular sum

Defining the binary operation $\oplus$ on $\{s,r,p\}$

$\oplus$	p	s	r
p	p	s	r
s	s	s	s
r	r	r	r

i	8	7	6	5	4	3	2	1	0
x_i		1	0	1	0	1	0	0	1
y_i		0	1	1	0	1	0	1	1
σ_i	p	p	r	s	r	s	p	r	s
S_{0i}	r	r	r	s	r	s	r	r	s
c_i	1	1	1	0	1	0	1	1	0
z_i	1	0	0	0	1	0	1	0	0

$z_i = (x_i + y_i + c_i) \bmod 2$

Figure 11.6 Using prefix computations and Proposition 11.1.2 to compute the binary sum of $x = 10101001$ and $y = 01101011$

Parallel Matrix-Vector Product

- Used often in scientific computations.
- Given an n x n matrix $A = (a_{ij})_{nxn}$ and the column vector $X=(x_1,x_2,\ldots,x_n)$, the matrix vector product AX is the column vector $B=(b_1,b_2,\ldots,b_n)$ defined by

$$b_i = \Sigma\, a_{ij}x_j, \quad i = 1,\ldots, n$$

- CREW PRAM Algorithm
  - Stored in the array A[1:n,1:n] and X[1:n]
  - Number of processors : n**2
  - Parallel call of DotProduct
  - Time : log n

Output: *Prod*[1:*n*] (matrix-vector product, where $Prod[i] = a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{in}x_n,\ i = 1, \ldots, n$)

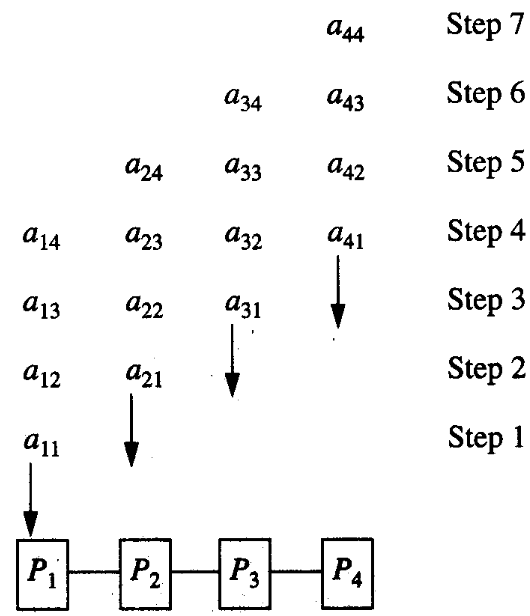
for $1 \le i \le n$ **do in parallel**
  $Prod[i] := DotProdPRAM(A[i,1:n], X[1:n])$
end in parallel

end *MatVecProdCREW*

Parallel Matrix-Vector Product : 1-D Mesh

- **Systolic Algorithm : Matrix and Vector are supplied as input**
- **Each processor holds one value of the matrix and vector in any processor's memory at each stage.**
- **The value received from the top and the value received from the left is multiplied and added to the value kept in the memory.**
- **The value received from the top is passed to the bottom and the value received from the left is passed to the right.**
- **The total time complexity is 2n-1**

Computation of AX when $n = 4$.



Phase 1	Step 1	$x_1 \to a_{11}x_1$	*	*	*
	Step 2	$x_2 \to a_{12}x_2$	$a_{21}x_1$	*	*
	Step 3	$x_3 \to a_{13}x_3$	$a_{22}x_2$	$a_{31}x_1$	*
	Step 4	$x_4 \to a_{14}x_4$	$a_{23}x_3$	$a_{32}x_2$	$a_{41}x_1$
Phase 2	Step 5	*	$a_{24}x_4$	$a_{33}x_3$	$a_{42}x_2$
	Step 6	*	*	$a_{34}x_4$	$a_{43}x_3$
	Step 7	*	*	*	$a_{44}x_4$

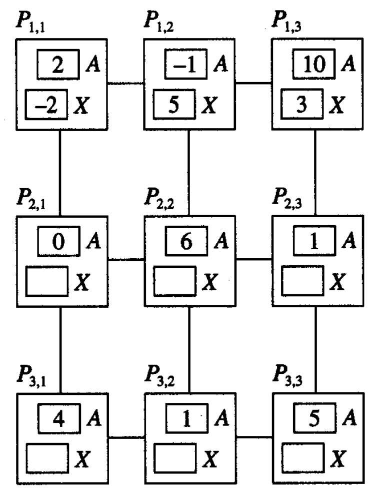
* indicates idle

Products shown are computed at the given step, and then added to *Prod*

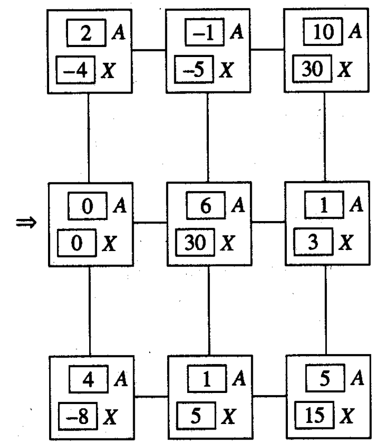
```
for 1 ≤ i ≤ n do in parallel {initialize Prod}
   P_i:Prod := 0
end in parallel
                                            {Phase 1}
for j := 1 to n do
   for P_i, 1 ≤ i ≤ j do in parallel
      if i < j then
         P_{i+1}:X ⇐ P_i:X                 {propagate X right}
      endif
      read(P_1:X)    {P_1:X = x_j}
      read(P_i:A)    {P_i:A = a_{i,j-i+1}}
      Prod := Prod + A*X
   end in parallel
endfor
                                            {Phase 2}
for j := 2 to n do
   for P_i, j - 1 ≤ i ≤ n - 1 do in parallel
      P_{i+1}:X ⇐ P_i:X                    {propagate X right}
      read(P_i:A)                          {input a_{i,n+j-i} to P_i:A}
      Prod := Prod + A*X
   end in parallel
endfor
end MatVecProd1DMesh
```

Parallel Matrix-Vector Product : 2-D Mesh and MOT

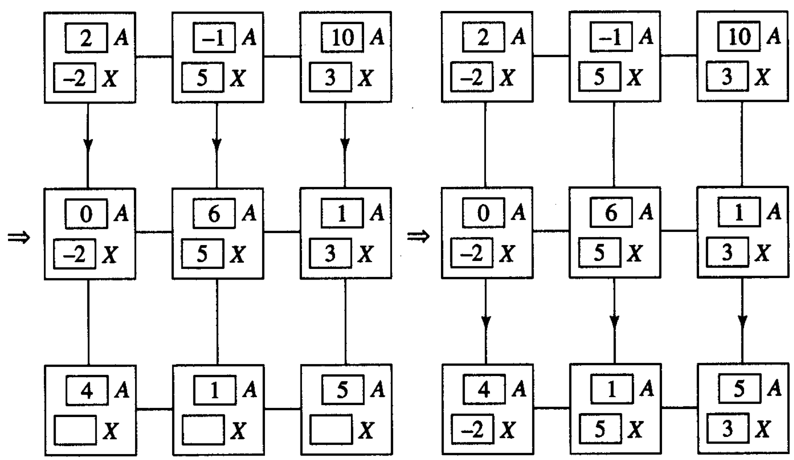
- **Matrix and Vector values are initially distributed.**
- **2-D Mesh Algorithm**
  - **Broadcast the dot vector to rows.**
  - **Each processor multiplies.**
  - **Sum at the leftmost processor by shifting the values to left.**
- **2-D Mesh of Trees Algorithm**
  - **See the architecture**
  - **Broadcast the dot vector to rows.**
  - **Each processor multiplies.**
  - **Sum at the tree by summing the children's values**



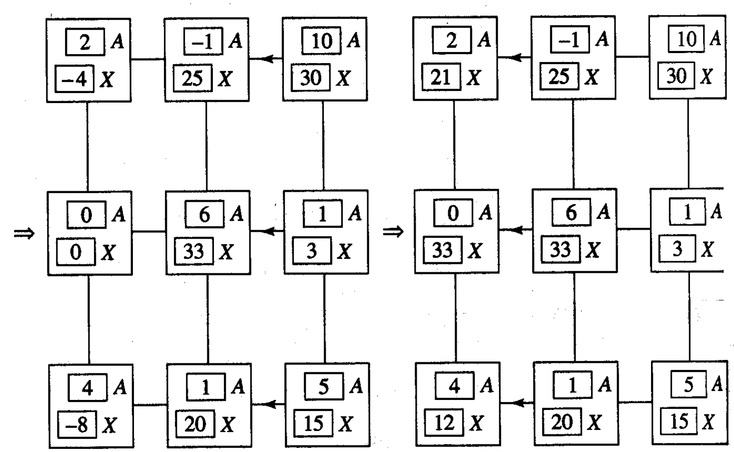
(a) Initial values of distributed variable A and X



(c) **for** $P_{i,j}$, $1 \le i,j \le q$ **do in parallel**
$X := A * X$
end in parallel



(b) Broadcast values of X in first row to rows 2 and 3.



(d) Add third column's X values to those in second column, then add second column's X values to first column. Dot product resides in first column's X.

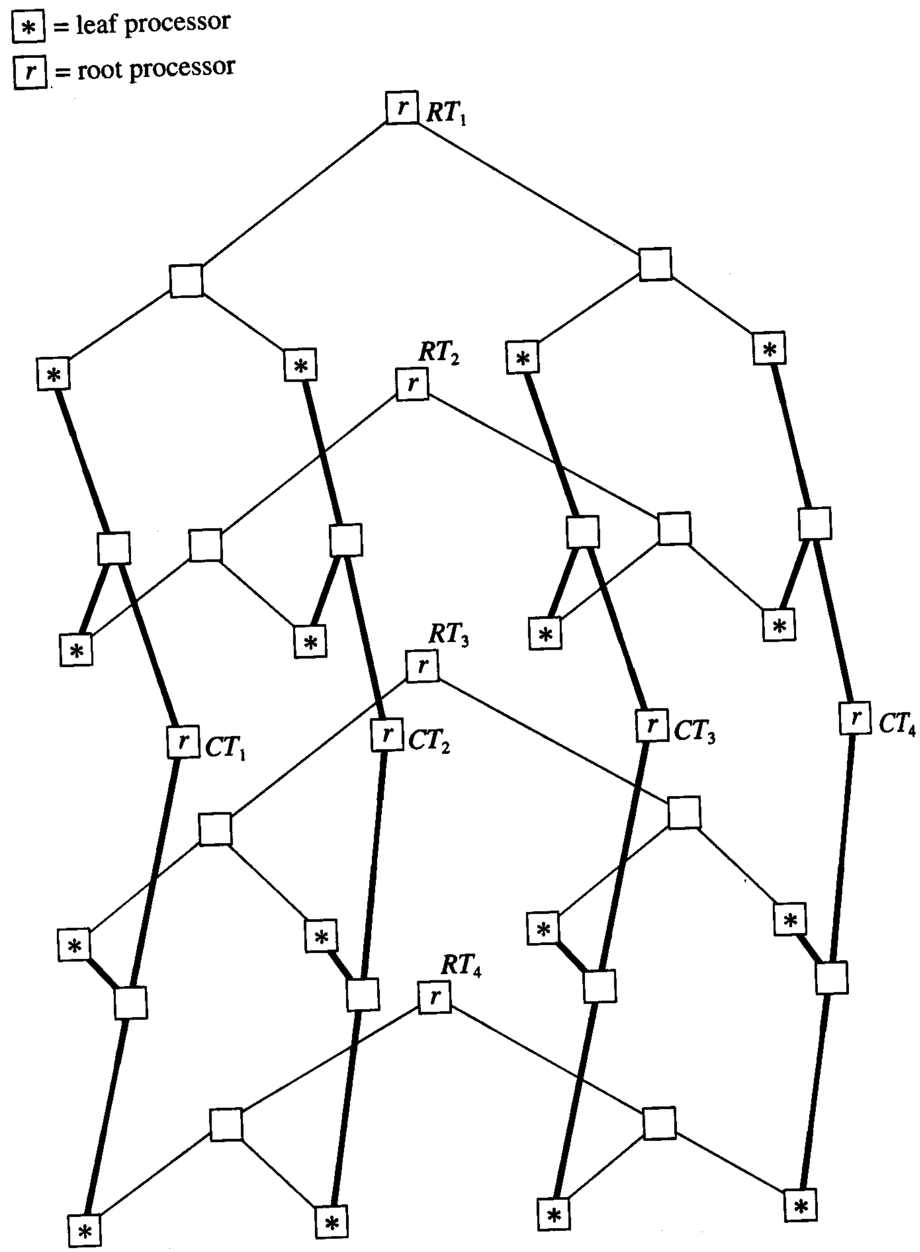
procedure *MatVecProd2DMesh*(A,n,X)

Model: two-dimensional mesh $\mathbf{M}_{n,n}$ **with** $p = n^2$ **processors**

Input: A ($\mathbf{P}_{i,j}$:A contains a_{ij}), **range:** $P_{i,j}$, $1 \le i,j \le n$
X ($\mathbf{P}_{1,j}$:X contains x_j), **range:** $P_{1,j}$, $1 \le j \le n$
Output: X ($\mathbf{P}_{i,1}$:X contains $a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{in}x_n$), **range:** $P_{i,1}$, $1 \le i \le n$

```
for i := 1 to n − 1 do
        {broadcast X from ith row to (i + 1)st row}
  for P_{i,j}, 1 ≤ j ≤ n do in parallel
    P_{i+1,j}:X ⇐ P_{i,j}:X    {propagate X down}
  end in parallel
endfor
        {compute a_{ij}x_j in parallel}
for P_{i,j}, 1 ≤ i,j ≤ n do in parallel
  X := A * X
end in parallel
        {sum across rows in parallel}
for j := n down to 2 do
  for P_{i,j}, 1 ≤ i ≤ n do in parallel
    P_{i,j−1}:Temp ⇐ P_{i,j}:X    {communicate left from X to Temp}
    X := X + Temp
  end in parallel
endfor
```

end *MatVecProd2DMesh*



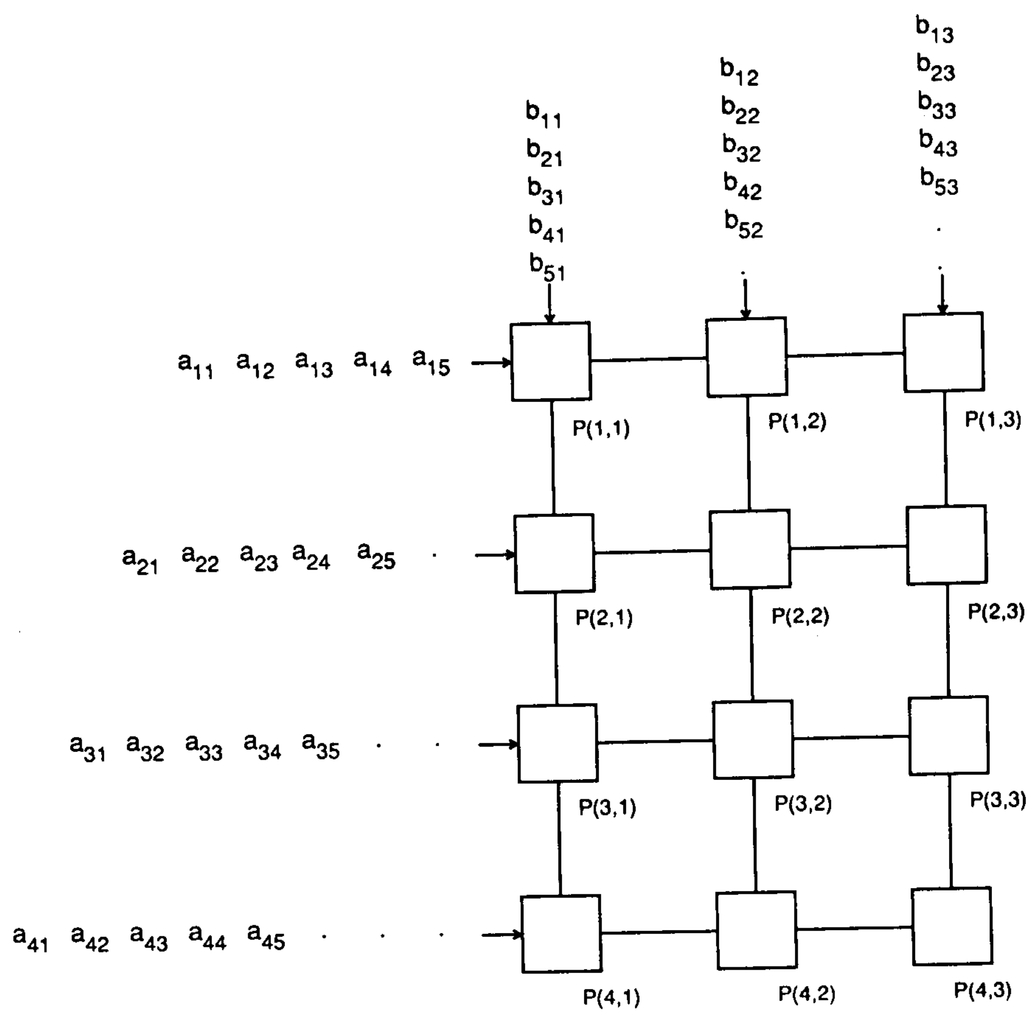
* = leaf processor
r = root processor
RT1
RT2
RT3
RT4
CT1
CT2
CT3
CT4

Parallel Matrix Multiplication

- Extension of Parallel Matrix Vector Product
- Assume square matrices A and B
- PRAM Algorithm
  - n**3 processors
  - Parallel extension of DotProduct
  - Time : log n

Parallel Matrix Multiplication : 2-D Mesh

- Systolic Algorithm : Matrices are supplied as input
- Inputing sequence is different
- Each processor holds one value of the matrices in any processor's memory at each stage.
- The value received from the top and the value received from the left is multiplied and added to the value kept in the memory.
- The value received from the top is passed to the bottom and the value received from the left is passed to the right.
- The total time complexity is 3n-1


b11 b21 b31 b41 b51
b12 b22 b32 b42 b52
b13 b23 b33 b43 b53
a11 a12 a13 a14 a15
a21 a22 a23 a24 a25
a31 a32 a33 a34 a35
a41 a42 a43 a44 a45
P(1,1)
P(1,2)
P(1,3)
P(2,1)
P(2,2)
P(2,3)
P(3,1)
P(3,2)
P(3,3)
P(4,1)
P(4,2)
P(4,3)

Parallel Matrix Multiplication : 3-D MOT

- **Extension of Parallel Matrix Vector Product on 2-D MOT**
- **Algorithm**
  - **Phase 1: Input a_{ij} and b_{ij} to the roots of T_{ij} and T_{ji}, respectively**
  - **Phase 2: Broadcast input values to the leaves, so that the leaves of T_{ij} all have the value a_{ij}, and the leaves of T_{ji} all have the vaue b_{ij}**
  - **Phase 3: After phase 2 is completed, the leaf processor L_{jik} has both the value a_{ik} and the value b_{kj}. In a single parallel step, compute the product $a_{ik}b_{kj}$**
  - **Phase 4: Sum the leaves of tree T_{ji} so that resultant sum is stored in the root of T_{ji}**
- **Time : log n steps**

Summary

- **Parallel Prefix Computations**
  - **PRAM, Tree, 1-D, 2-D algorithms**
  - **Carry-Lookahead Addition Application**
- **Parallel Matrix-Vector Product**
  - **PRAM, 1-D, 2-D MOT algorithms**
- **Parallel Matrix Multiplication**
  - **PRAM, 2-D, 3-D MOT algorithms**