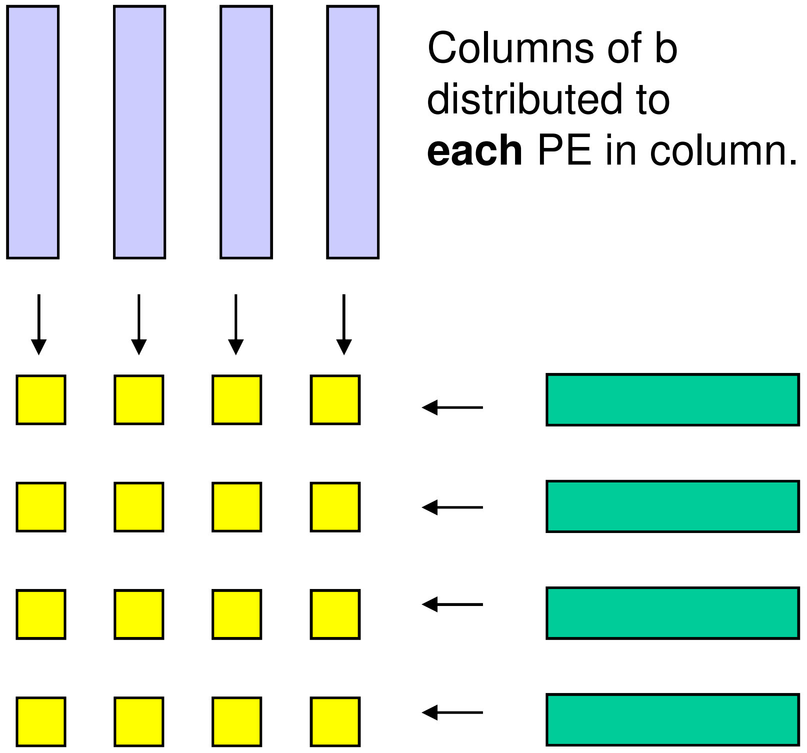
# Examples of Two-Dimensional Systolic Arrays

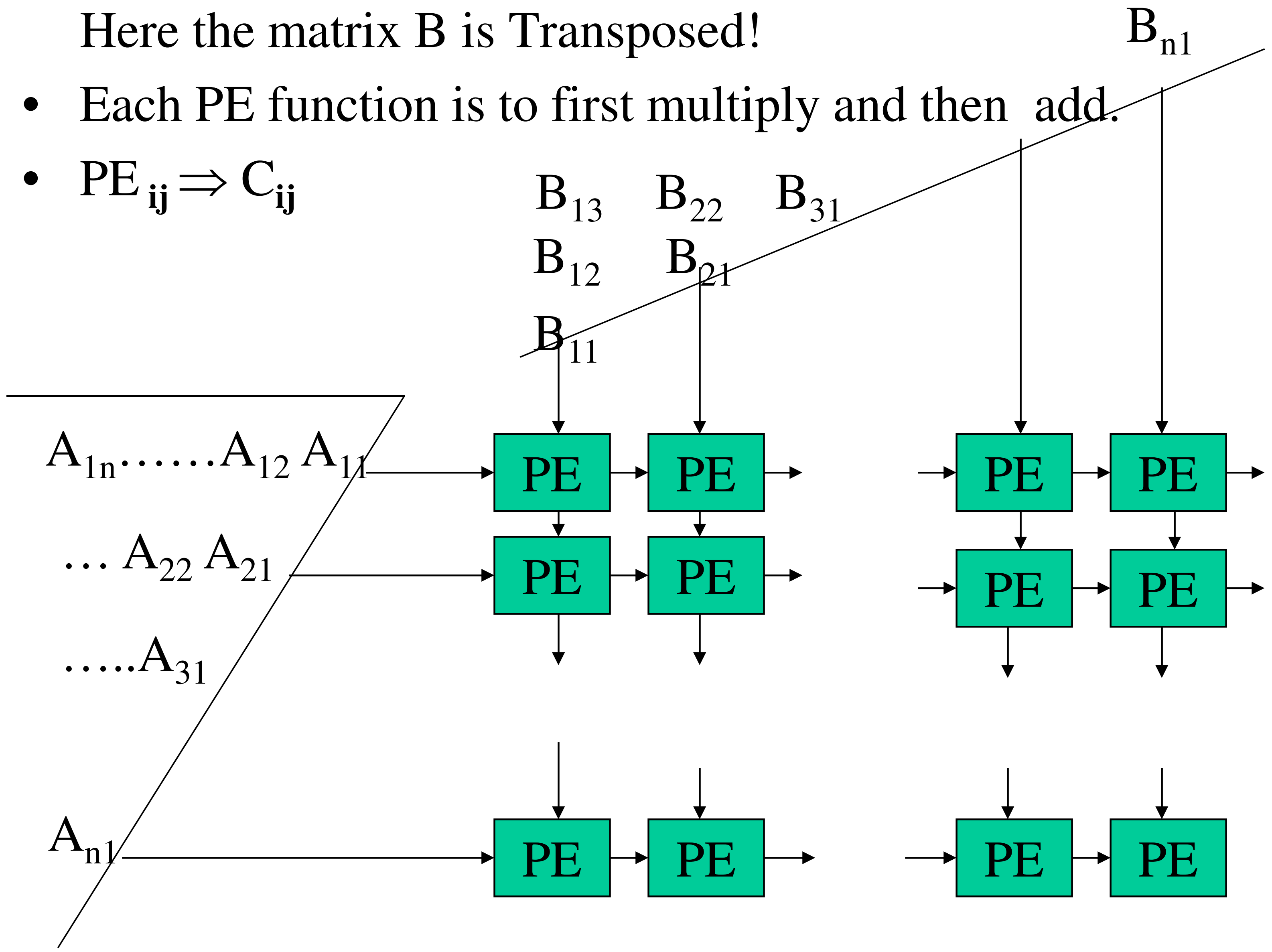
# Obvious Matrix Multiply


Columns of b distributed to **each** PE in column.

Rows of a distributed to **each** PE in row.

Row x Column on respective PEs.

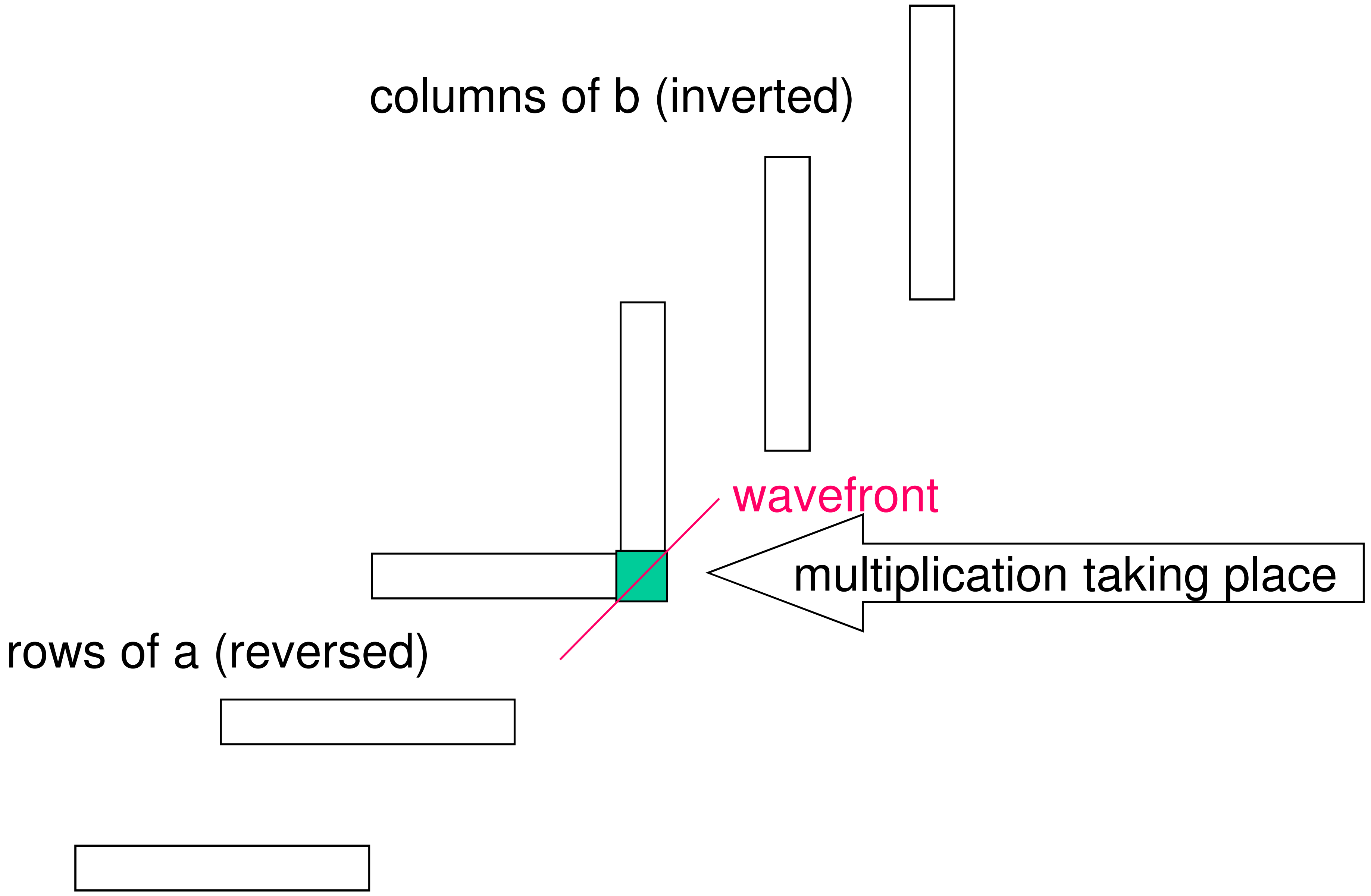
- Multiplication
  Here the matrix B is Transposed!
- Each PE function is to first multiply and then add.
- $PE_{ij} \Rightarrow C_{ij}$



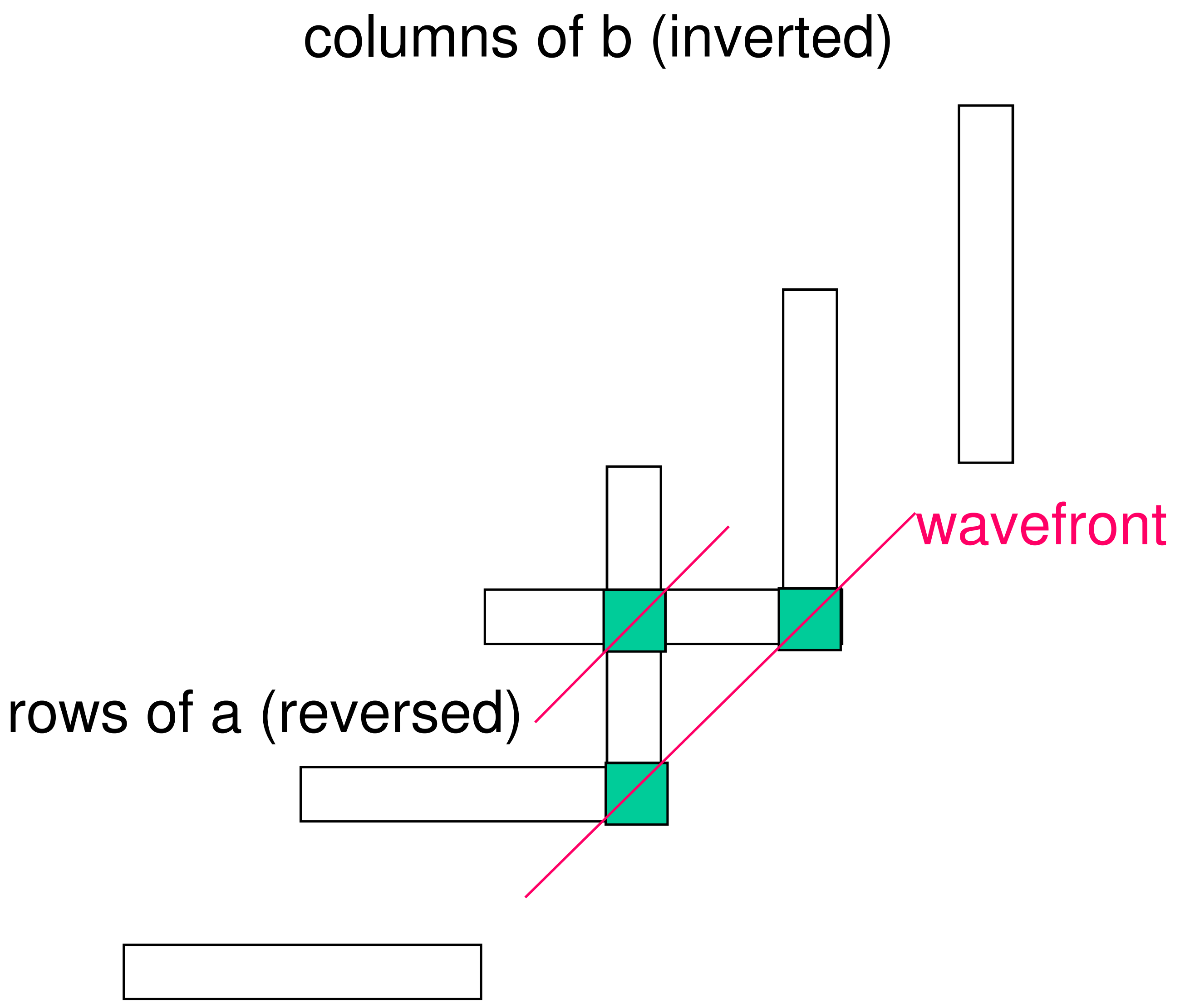
## Example 4: A Related Algorithm: Cannon’s Method

- Let’s take another view of systolic multiplication:
  - Consider the rows and columns of the matrices to be multiplied as **strips** that are slide past each other.
- The strips are staggered so that the correct elements are multiplied at each time step.

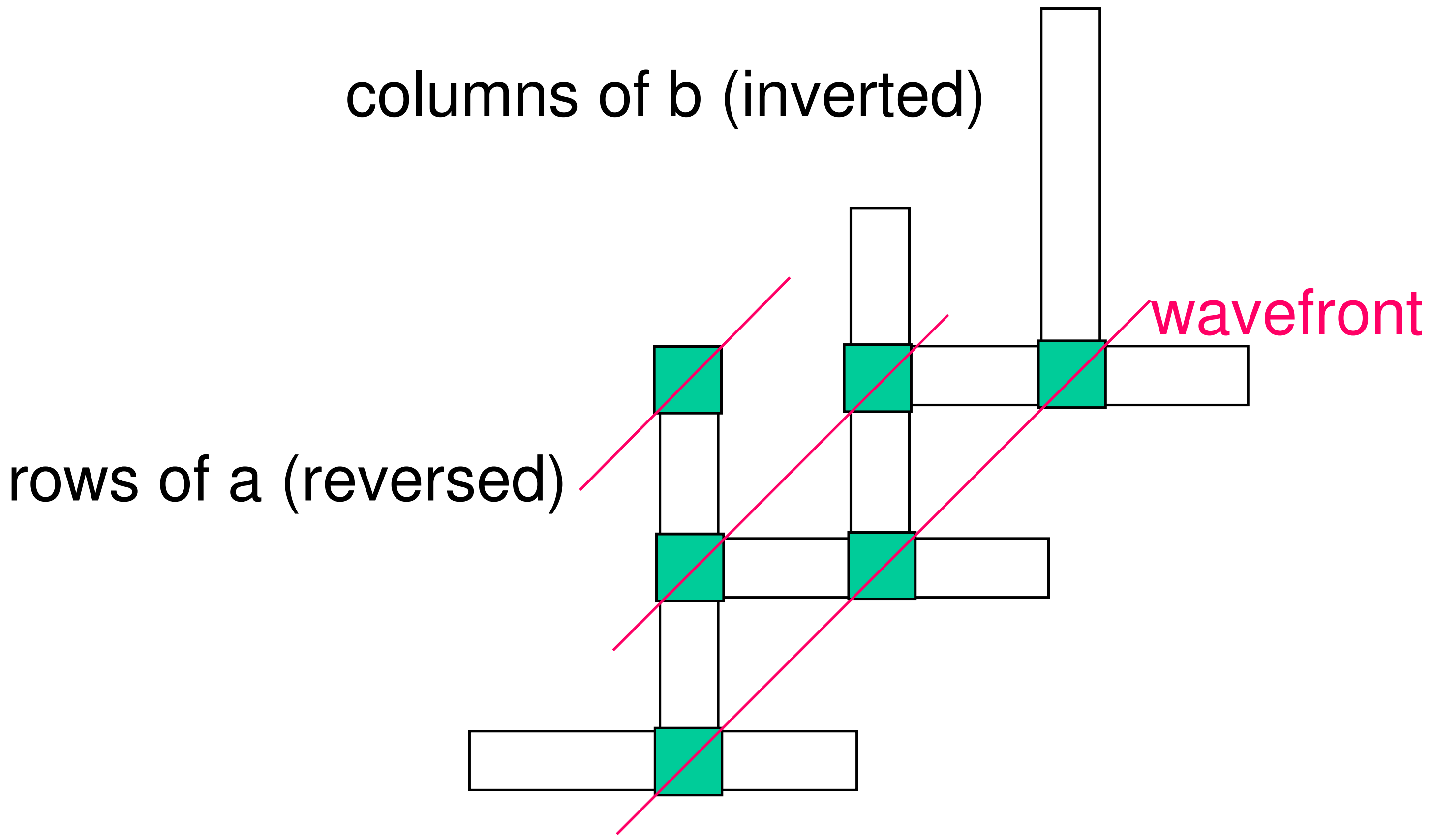
## First step


columns of b (inverted)
wavefront
multiplication taking place
rows of a (reversed)

## Second step



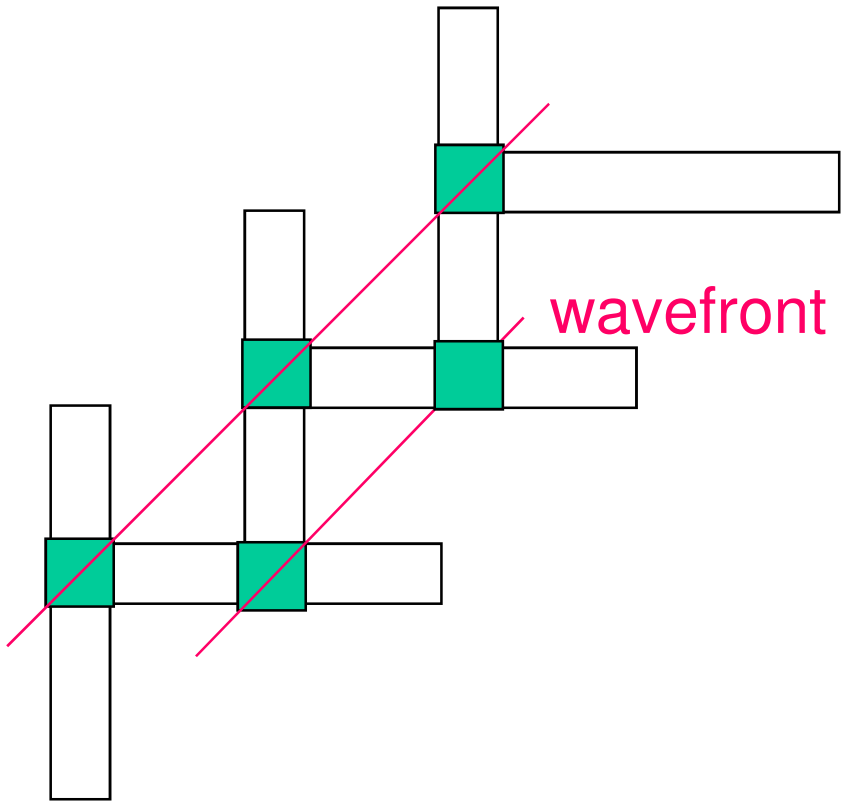
## Third step



Fourth step

columns of b (inverted)

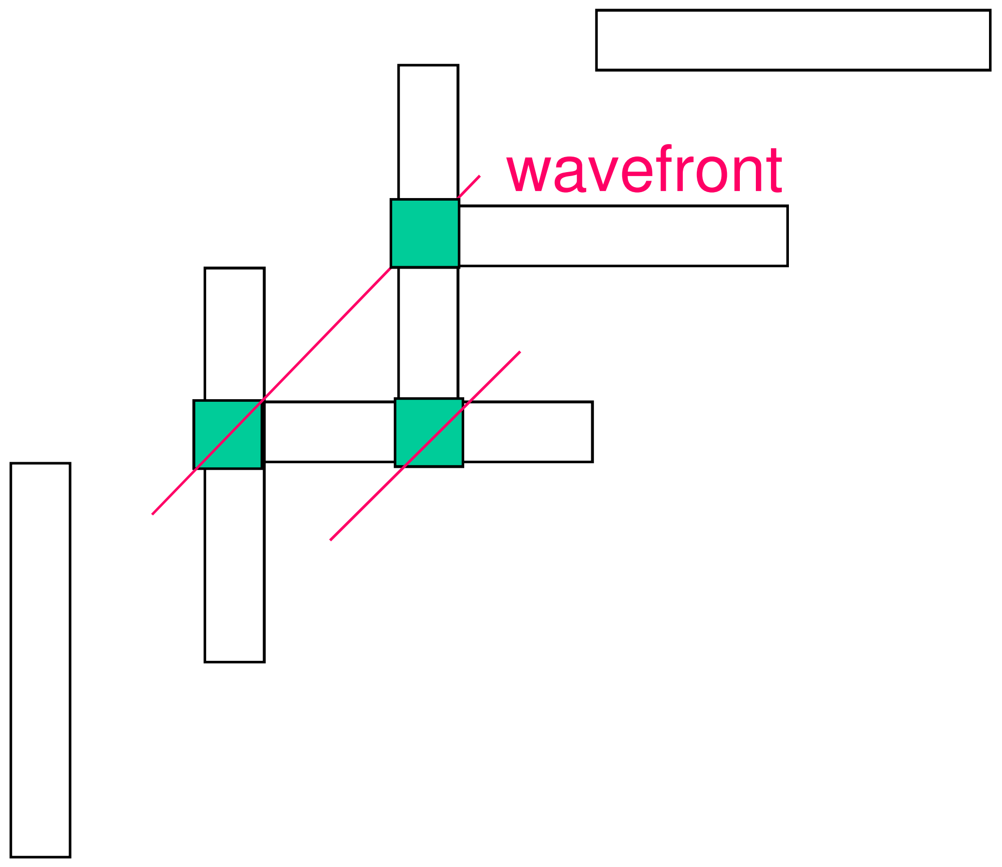
rows of a (reversed)



## Fifth step

columns of b (inverted)

rows of a (reversed)

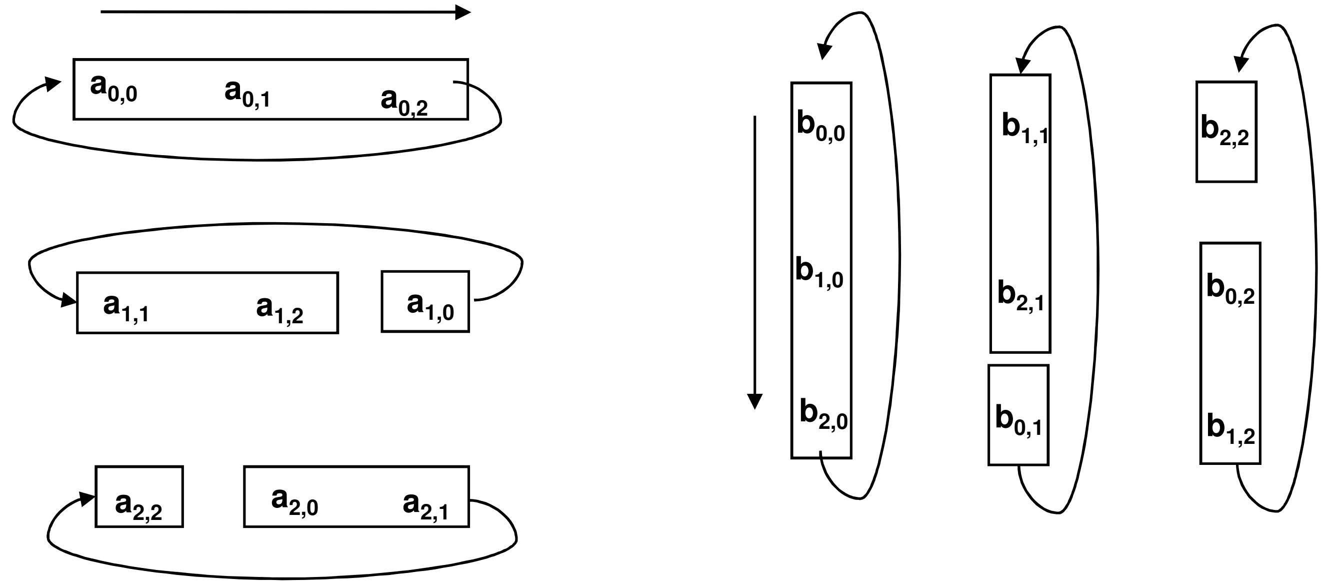


# Cannon’s Method

- Rather than have some processors idle,
  - wrap the array rows and columns so that **every processor is doing something on each step**.
- In other words, rather than feeding in the elements, they are rotated around,
  - starting in an initially staggered position as in the systolic model.
- We also change the order of products slightly, to make it correspond to more natural storage by rows and columns.

# Cannon Variation

Note that the a diagonal is in the left column and the b diagonal is in the top row.



Example sum: $c_{0,2} = a_{0,2}*b_{2,2} + a_{1,2}*b_{1,2} + a_{0,0}*b_{0,2}$

Products computed at each step:

Step 1

| | | |
|---|---|---|
| a00*b00 | a01*b11 | a02*b22 |
| a11*b10 | a12*b21 | a10*b02 |
| a22*b20 | a20*b01 | a21*b12 |

Step 2

| | | |
|---|---|---|
| a02*b20 | a00*b01 | a01*b12 |
| a10*b00 | a11*b11 | a12*b22 |
| a21*b10 | a22*b21 | a20*b02 |

Step 3

| | | |
|---|---|---|
| a01*b10 | a02*b21 | a00*b02 |
| a12*b20 | a10*b01 | a11*b12 |
| a20*b00 | a21*b11 | a22*b22 |

## Application of Cannon's Technique

- Consider matrix multiplication of 2 n x n matrices on a distributed memory machine, on say, $n^2$ processing elements.
- An obvious way to compute is to think of the PE's as a matrix, with each computing one element of the product.
- We would send each row of the matrix to n processors and each column to n processors.
- **In effect, in the obvious way, each matrix is stored a total of n times.**

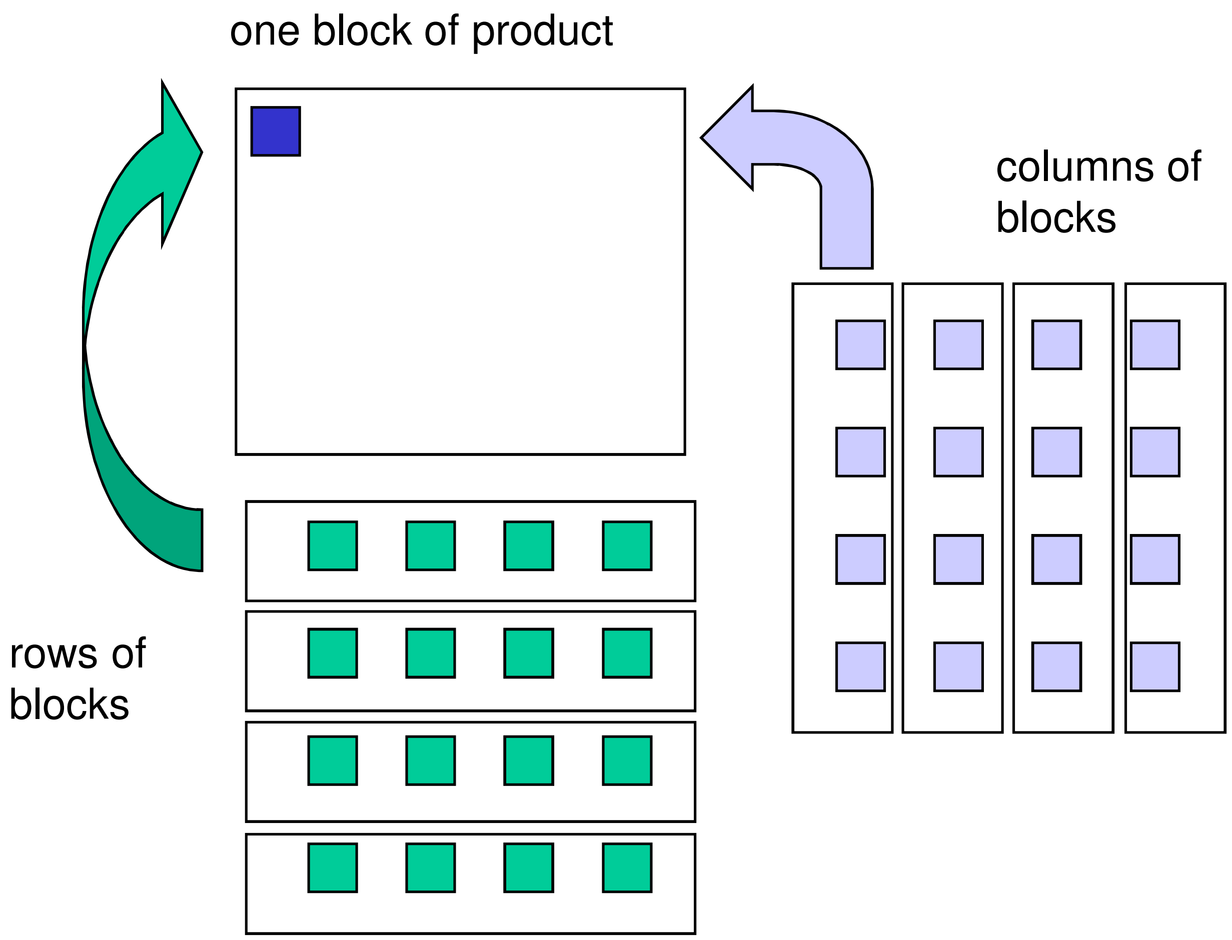
## Cannon’s Method

- Cannon’s method avoids storing each matrix n times, instead **cycling** (“piping”) the elements through the PE array.
- (It is sometimes called the “pipe-roll” method.)
- The problem is that this cycling is typically too fine-grain to be useful for element-by-element multiply.

## Partitioned Multiplication

- Partitioned multiplication divides the matrices into **blocks**.
- It can be shown that multiplying the individual blocks as if elements of matrices themselves gives the matrix product.

# Block Multiplication



## Cannon’s Method is Fine for Block Multiplication

- The blocks are aligned initially as the elements were in our description.
- At each step, entire blocks are transmitted down and to the left of neighboring PE’s.
- Memory space is conserved.

## Exercise

- Analyze the running time for the block version of Cannon's method for two n x n matrices on p processors, using $t_{comp}$ as the unit operation time and $t_{comm}$ as the unit communication time and $t_{start}$ as the per-message latency .
- Assume that any pair of processors can communicate in parallel.
- Each block is (n/sqrt(p)) x (n/sqrt(p)).

## Example 6: Fox’s Algorithm

- This algorithm is also for block matrix multiplication; it has a resemblance to Cannon’s algorithm.

- The difference is that on each cycle:
  - A row block is **broadcast** to every other processor in the row.
  - The column blocks are rolled cyclically.

# Fox's Algorithm

Step 1

| | | |
|---|---|---|
| a00*b00 | a00*b01 | a00*b02 |
| a11*b10 | a11*b11 | a11*b12 |
| a22*b20 | a22*b21 | a22*b22 |

A different row block of a is broadcast in each step.

Step 2

| | | |
|---|---|---|
| a01*b10 | a01*b11 | a01*b12 |
| a12*b20 | a12*b21 | a12*b22 |
| a20*b00 | a20*b01 | a20*b02 |

b columns are rolled

Step 3

| | | |
|---|---|---|
| a02*b20 | a02*b21 | a02*b22 |
| a10*b00 | a10*b01 | a10*b02 |
| a21*b10 | a21*b11 | a21*b12 |

# Synchronous Computations

## Barriers

- Mentioned earlier
- Synchronize all of a group of processes
- Used in both distributed and shared-memory
- Issue: Implementation & cost

# Counter Method for Barriers

- One-phase version
  - Use for distributed-memory
  - Each processor sends a message to the others when barrier reached.
  - When each processor has received a message from all others, the processors pass the barrier

## Counter Method for Barriers

- Two-phase version
  - Use for shared-memory
  - Each processor sends a message to the master process.
  - When the master has received a message from all others, it sends messages to each indicating they can pass the barrier.
  - Easily implemented with blocking receives, or semaphores (one per processor).

# Tree Barrier

- Processors are organized as a tree, with each sending to its parent.
- **Fan-in phase:** When the root of the tree receives messages from both children, the barrier is complete.
- **Fan-out phase:** Messages are then sent down the tree in the reverse direction, and processes pass the barrier upon receipt.

# Butterfly Barrier

- Essentially a fan-in tree for **each** processor, with some sharing toward the leaves.
- Advantage is that no separate fan-out phase is required.

# Butterfly Barrier

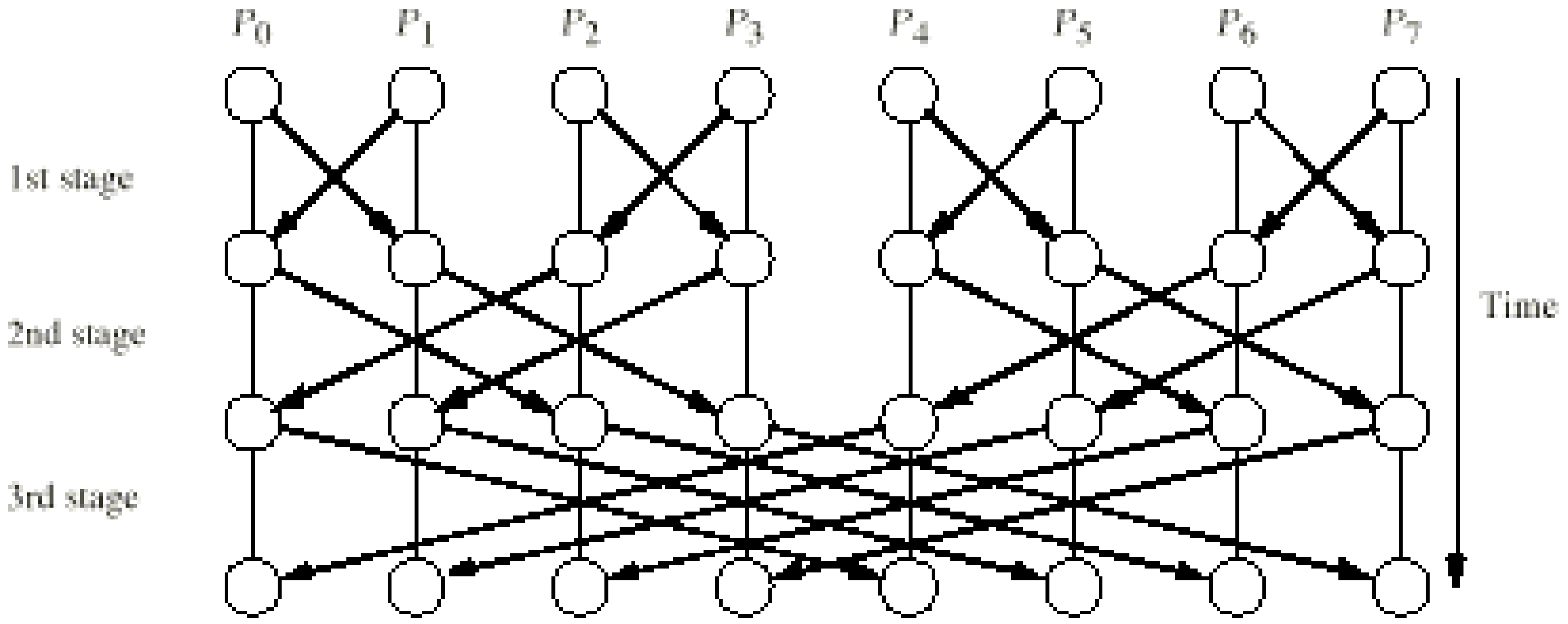


**Figure 6.6** Butterfly construction.

## Barrier Bonuses

- To implement a barrier, it is only necessary to increment a count (shared memory) or send a couple of messages per process.
- These are communications with null content.
- By adding content to messages, barriers can have added utility.

## Barrier Bonuses

- These can be accomplished along with a barrier:
  - Reduce according to binary operator (esp. good for tree or butterfly barrier)
  - All-to-all broadcast

## Data Parallel Computations

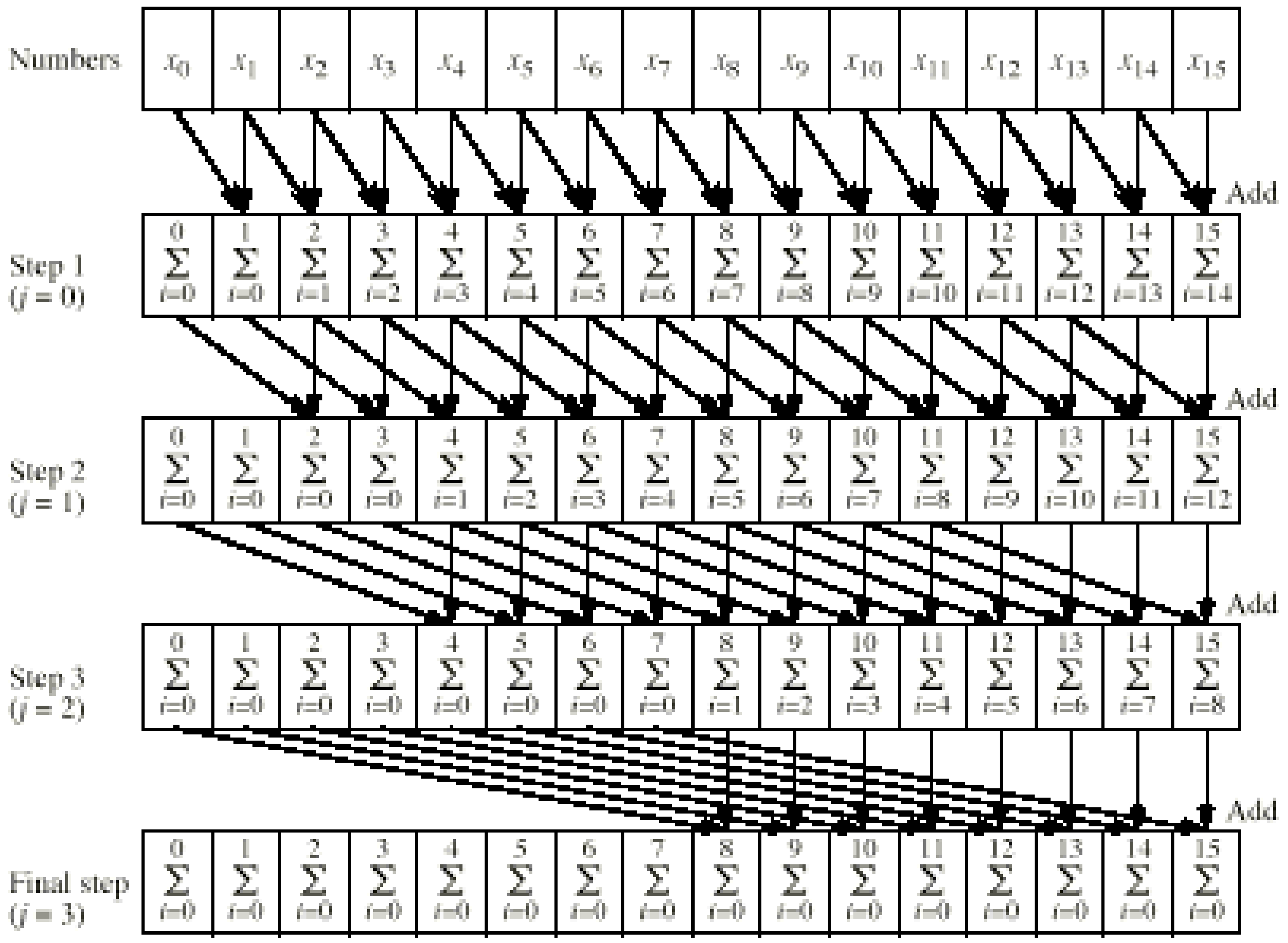
- **forall** statement:

```
forall( j = 0; j < n; j++ )
        {
        … body done in parallel for all j ...
        }
```

## forall **synchronization assumptions**

- There are different interpretations of **forall**, so you need to “read the fine print”.
- Possible assumptions from weakest to strongest:
  - No implied synchronization
  - Implied barrier at the end of each loop body
  - Implied barrier before each assignment
  - Each machine instruction synchronized, SIMD-fashion

# Example: Prefix-Sum



Numbers
$x_0$ $x_1$ $x_2$ $x_3$ $x_4$ $x_5$ $x_6$ $x_7$ $x_8$ $x_9$ $x_{10}$ $x_{11}$ $x_{12}$ $x_{13}$ $x_{14}$ $x_{15}$
Add
Step 1 ($j = 0$)
Add
Step 2 ($j = 1$)
Add
Step 3 ($j = 2$)
Add
Final step ($j = 3$)

**Figure 6.8** Data parallel prefix sum operation.

# Example: Prefix-Sum

- Assume that n is a power of 2.
- Assume shared memory.
- Assume barrier before assignments
- for( j = 0; j < log(n); j++ )

```
    forall( i = $2^j$; i < n; i++)

        $\underbrace{x[i]}_{\text{effectively \textbf{buffered} new value}} \mathrel{+}= \underbrace{x[i - 2^j]}_{\text{old value}};$
```

## **Implementing** forall **using SPMD:**
### Assuming PP: "Synchronous Iteration" (barrier at end of body)

```
for( j = 0; j < log(n); j++ )
        forall( i = 0; i < n; i++)
                Body(i);
```

implementable in SPMD as:

```
for( j = 0; j < log(n); j++ )
        {
        i = my_process_rank();
        Body(i);
        barrier();
        }
```

Outer **forall** processes **implicit**

# Example: Iterative Linear Equation Solver

```
for( iter = 0; iter < numIterations; iter++ )

        forall( i = 0; i < n; i++)
                {
                double sum = 0;
                for( j = 0; j < n; j++ )
                        sum += a[i][j]*x[j];
                x[i] = sum;
                }
```

Note:
*Local* memory for each i.

barrier desired here

# Iterative Linear Equation Solver: Translation to SPMD

Outer
forall processes
implicit

```
for( iter = 0; iter < numIterations; iter++ )
    {
    i = my_process_rank();
    double sum = 0;
    for( j = 0; j < n; j++ )
            sum += a[i][j]*x[j];
    new_x[i] = sum;
    all gather new_x to x (implied barrier)
    }
```

## Nested forall's

-
  ```
  for( iter = 0; iter < numIterations; iter++ )
      forall( i = 0; i < m; i++)
          forall( j = 0; j < n; i++)
              Body(i, j)
  ```

## Example of nested forall's: Laplace Heat equation

-
  ```
  for( iter = 0; iter < numIterations; iter++ )
        forall( i = 0; i < m; i++)
              forall( j = 0; j < n; i++)
                 x[i][j] = (x[i-1][j] + x[i][j-1] + x[i+1][j] + x[i][j+1])/4.0;
  ```

## Exercise

- How would you translate nested forall's to SPMD?

# Synchronous Computations

- Synchronous computations have the form

  (Barrier)  
  Computation  
  Barrier  
  Computation  
  …
- Frequency of the barrier and homogeneity of the intervening computations on the processors may vary
- We've seen some synchronous computations already (Jacobi2D, Systolic MM)

## Synchronous Computations

- Synchronous computations can be simulated using asynchronous programming models
  - Iterations can be tagged so that the appropriate data is combined
- Performance of such computations depends on the granularity of the platform, how expensive synchronizations are, and how much time is spent idle waiting for the right data to arrive

## Barrier Synchronizations

- Barrier synchronizations can be implemented in many ways:
  - As part of the algorithm
  - As a part of the communication library
    - PVM and MPI have barrier operations
  - In hardware
- Implementations vary

## Review

- What is time balancing? How do we use time-balancing to decompose Jacobi2D for a cluster?
- Describe the general flow of data and computation in a pipelined algorithm. What are possible bottlenecks?
- What are the three stages of a pipelined program? How long will each take with P processors and N data items?
- Would pipelined programs be well supported by SIMD machines? Why or why not?
- What is a systolic program? Would a systolic program be efficiently supported on a general-purpose MPP? Why or why not?

## Common Parallel Programming Paradigms

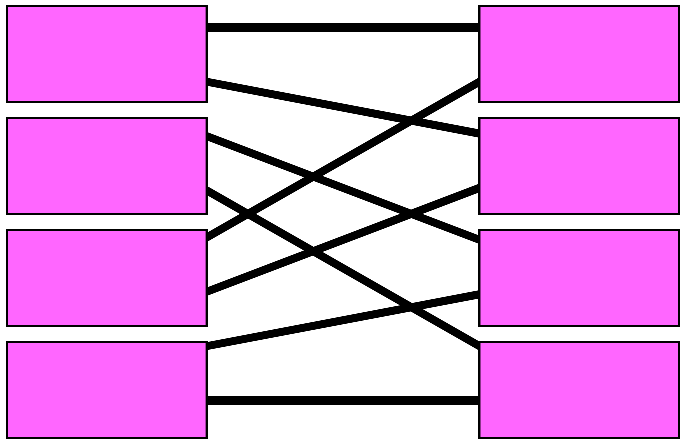
- Embarrassingly parallel programs
- Workqueue
- Master/Slave programs
- Monte Carlo methods
- Regular, Iterative (Stencil) Computations
- Pipelined Computations
- Synchronous Computations

# Synchronous Computations

- Synchronous computations are programs structured as a group of separate computations which must at times wait for each other before proceeding
- Fully synchronous computations = programs in which all processes synchronized at regular points
- Computation between synchronizations often called stages

# Synchronous Computation Example: Bitonic Sort

- Bitonic Sort an interesting example of a synchronous algorithm
- Computation proceeds in stages where each stage is a (smaller or larger) shuffle-exchange network
- Barrier synchronization at each stage



# Bitonic Sort

- A bitonic sequence is a list of keys

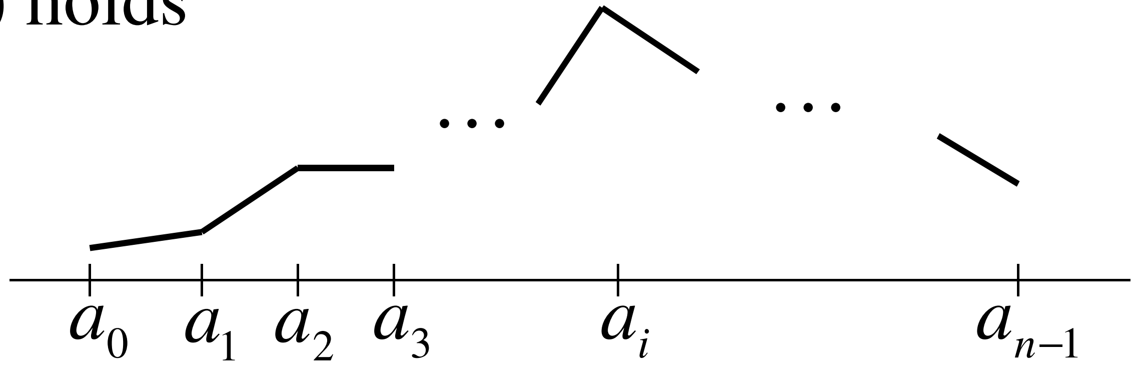
  $a_0, a_1, \cdots, a_{n-1}$ such that

  1 For some i, the keys have the ordering

  $$a_0 \le a_1 \le \cdots \le a_i \ge \cdots \ge a_{n-1}$$

  or

  2 The sequence can be shifted cyclically so that 1) holds



## Bitonic Sort Algorithm

- The bitonic sort algorithm recursively calls two procedures:
  - BSORT(i,j,X) takes bitonic sequence $a_i, a_{i+1}, \cdots, a_j$ and produces a non-decreasing (X=+) or a non-increasing sorted sequence (X=-)
  - BITONIC(i,j) takes an unsorted sequence $a_i, a_{i+1}, \cdots, a_j$ and produces a bitonic sequence
- The main algorithm is then
  - BITONIC(0,n-1)
  - BSORT(0,n-1,+)

# How does it do this?

- We'll show how BSORT and BITONIC work but first consider an interesting property of bitonic sequences:

Assume that $a_0, a_1, \cdots, a_{n-1}$ is bitonic and that n is even. Let

$$B_1 = \min\{a_0, a_{\frac{n}{2}}\}, \min\{a_1, a_{\frac{n}{2}+1}\}, \cdots, \min\{a_{\frac{n}{2}-1}, a_{n-1}\}$$

$$B_2 = \max\{a_0, a_{\frac{n}{2}}\}, \max\{a_1, a_{\frac{n}{2}+1}\}, \cdots, \max\{a_{\frac{n}{2}-1}, a_{n-1}\}$$

Then $B_1$ and $B_2$ are bitonic sequences and for all

$x \in B_1, y \in B_2 \quad x \leq y$

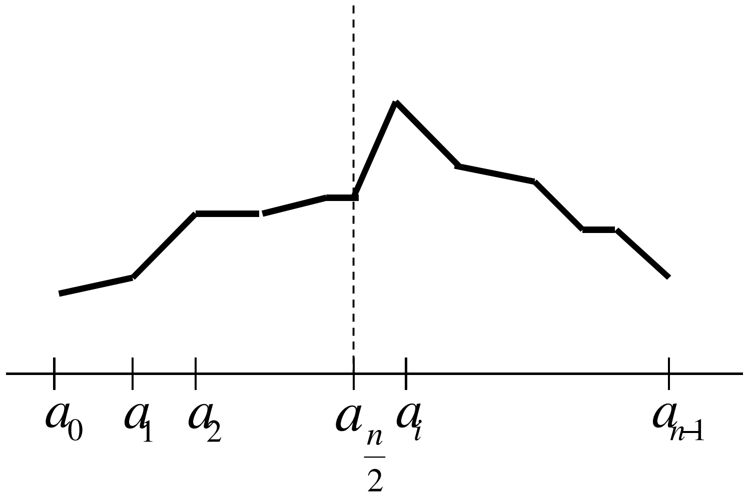
# Picture "Proof" of Interesting Property

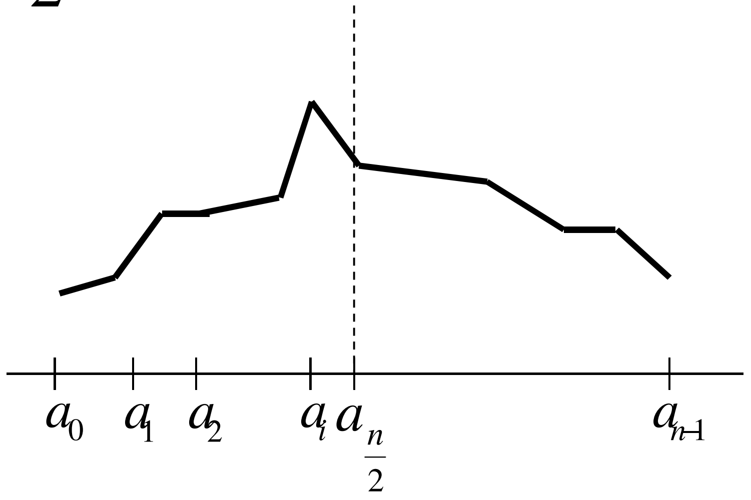
- Consider

$$B_1 = \min\{a_0, a_{\frac{n}{2}}\}, \min\{a_1, a_{\frac{n}{2}+1}\}, \cdots, \min\{a_{\frac{n}{2}-1}, a_{n-1}\}$$

$$B_2 = \max\{a_0, a_{\frac{n}{2}}\}, \max\{a_1, a_{\frac{n}{2}+1}\}, \cdots, \max\{a_{\frac{n}{2}-1}, a_{n-1}\}$$

- Two cases: $i \geq \frac{n}{2}$ and $i < \frac{n}{2}$





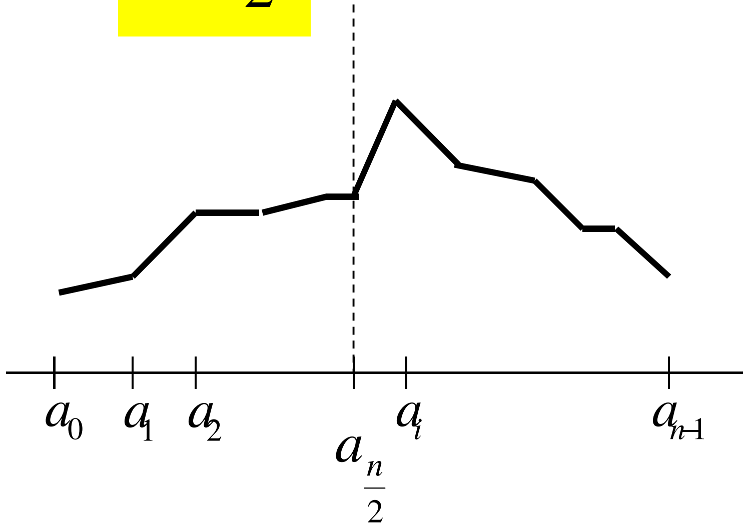
# Picture “Proof” of Interesting Property

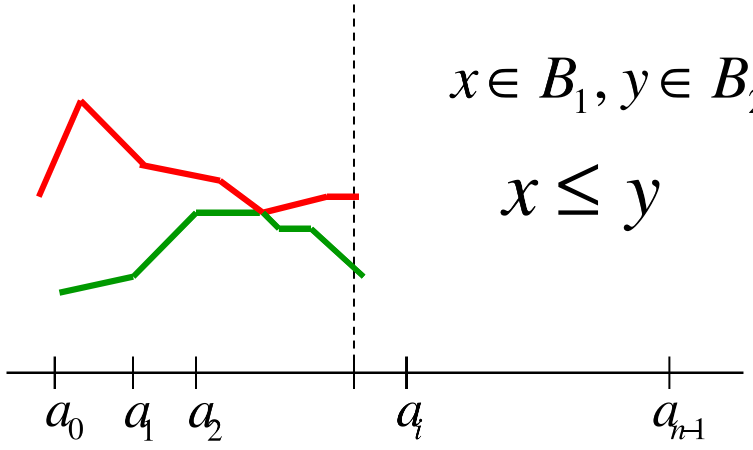
- Consider

$$\boxed{B_1 = \min\{a_0, a_{\frac{n}{2}}\}, \min\{a_1, a_{\frac{n}{2}+1}\}, \cdots, \min\{a_{\frac{n}{2}-1}, a_{n-1}\}}$$

$$\boxed{B_2 = \max\{a_0, a_{\frac{n}{2}}\}, \max\{a_1, a_{\frac{n}{2}+1}\}, \cdots, \max\{a_{\frac{n}{2}-1}, a_{n-1}\}}$$

$$i \geq \frac{n}{2}$$





$x \in B_1, y \in B_2$

$$x \leq y$$

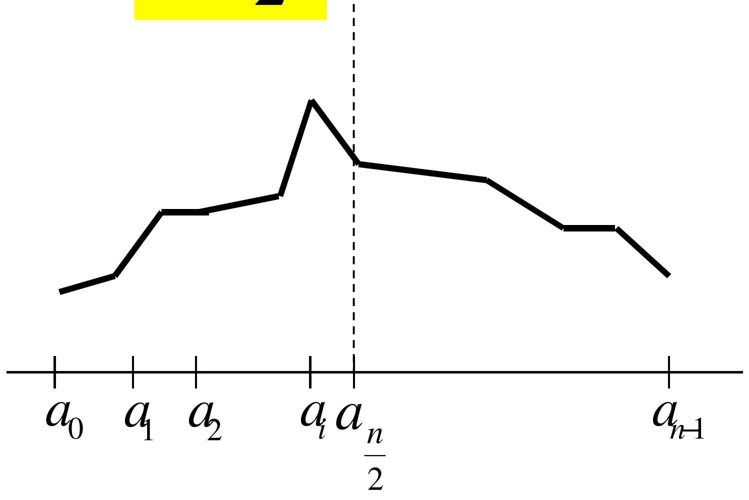
# Picture "Proof" of Interesting Property

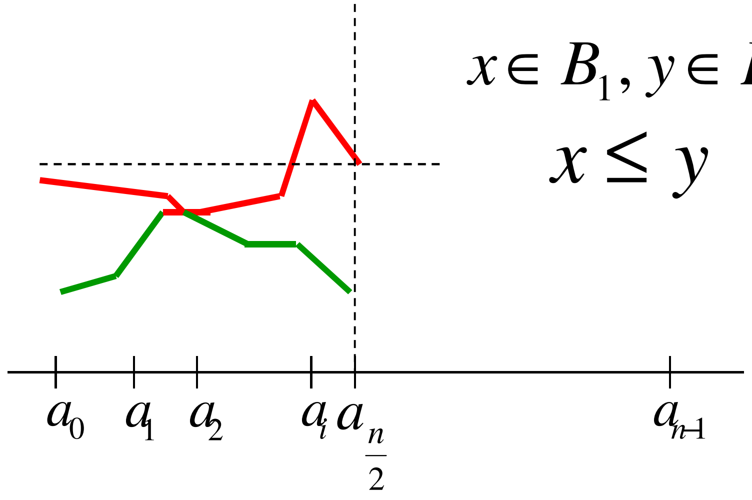
- Consider

$$\boxed{B_1 = \min\{a_0, a_{\frac{n}{2}}\}, \min\{a_1, a_{\frac{n}{2}+1}\}, \cdots, \min\{a_{\frac{n}{2}-1}, a_{n-1}\}}$$

$$\boxed{B_2 = \max\{a_0, a_{\frac{n}{2}}\}, \max\{a_1, a_{\frac{n}{2}+1}\}, \cdots, \max\{a_{\frac{n}{2}-1}, a_{n-1}\}}$$

$i < \frac{n}{2}$



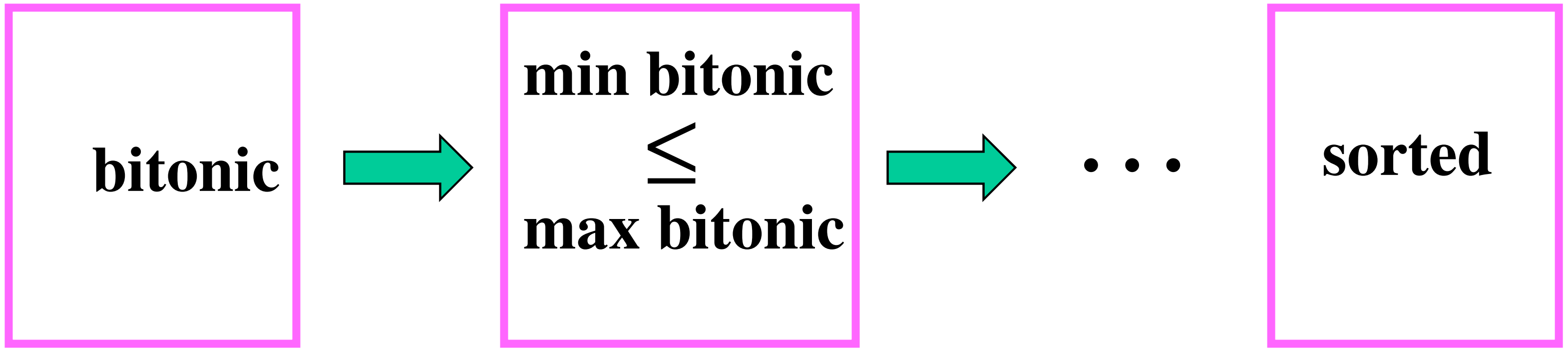


$x \in B_1, y \in B_2$

$x \leq y$

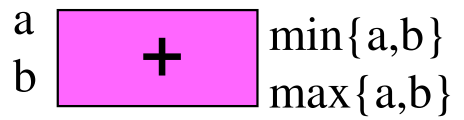
# Back to Bitonic Sort

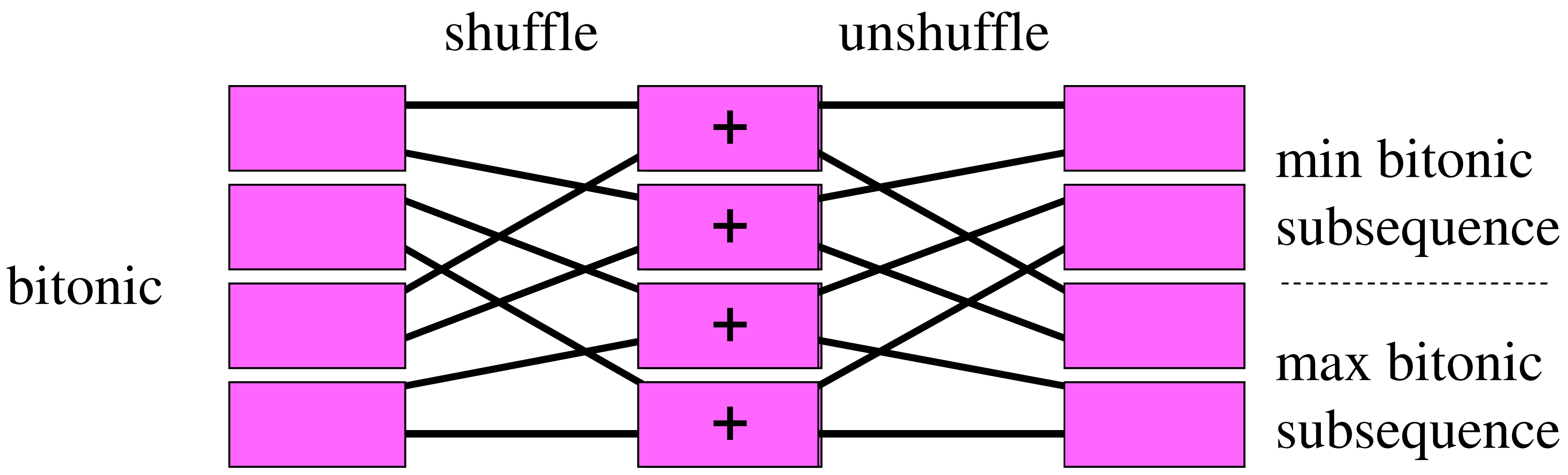
- Remember
  - BSORT(i,j,X) takes bitonic sequence $a_i, a_{i+1}, \cdots, a_j$ and produces a non-decreasing (X=+) or a non-increasing sorted sequence (X=-)
  - BITONIC(i,j) takes an unsorted sequence $a_i, a_{i+1}, \cdots, a_j$ and produces a bitonic sequence
- Let's look at BSORT first …



# Here's where the shuffle-exchange comes in …

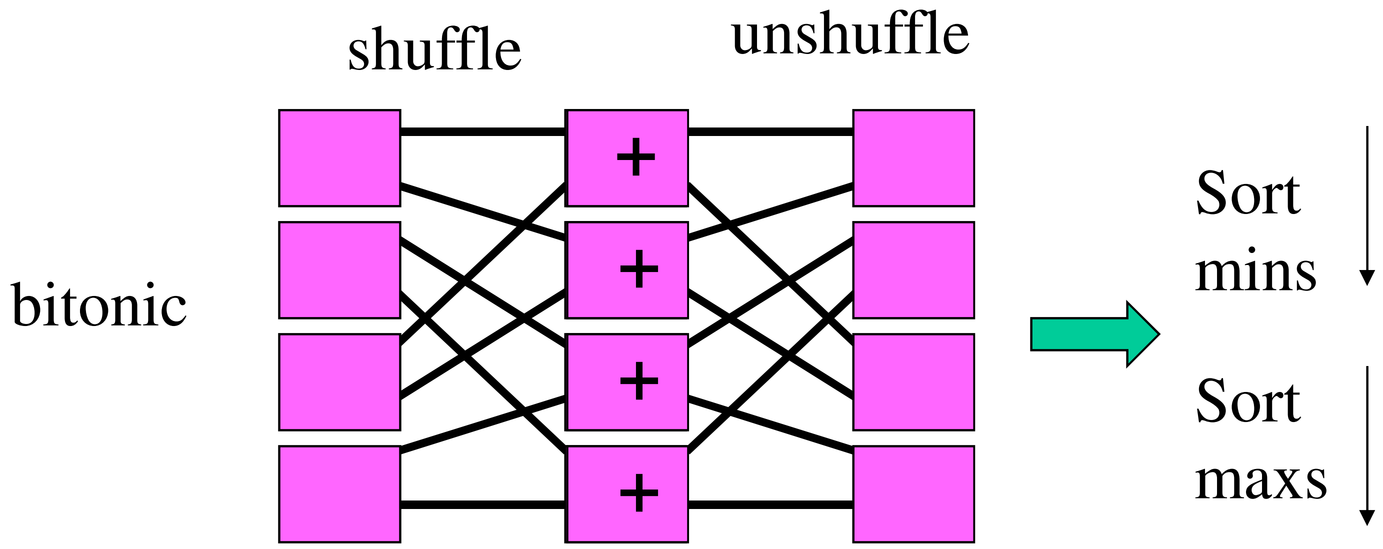
- Shuffle-exchange network routes the data correctly for comparison
- At each shuffle stage, can use + switch to separate B1 and B2





## Sort bitonic subsequences to get a sorted sequence

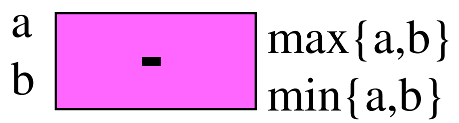
- **BSORT(i,j,X)**
  - If |j-i|<2 then return [min(i,i+1), max(i,i+1)]
  - Else
    - Shuffle(i,j,X)
    - Unshuffle(i,j)
    - Pardo
      - BSORT (i,i+(j-i+1)/2 - 1,X)
      - BSORT (i+(j-i+1)/2 +1,j,X)

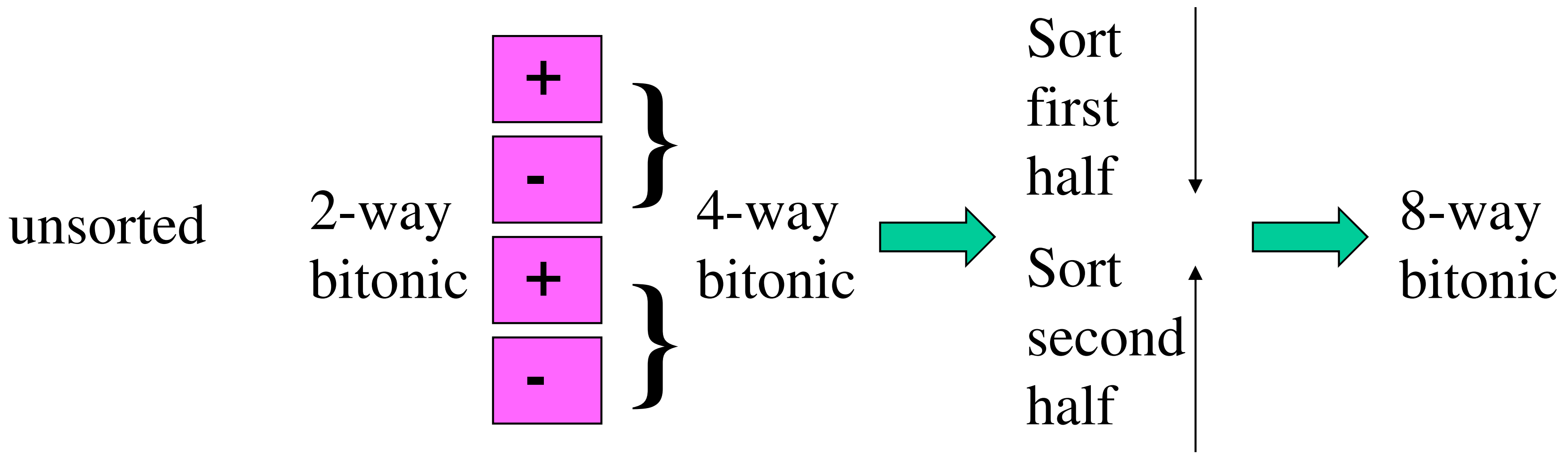


## BITONIC takes an unsorted sequence as input and returns a bitonic sequence

- **BITONIC(i,j)**
  - If |j-i|<2 then return [i,i+1]
  - Else
    - Pardo
      - BITONIC(i,i+(j-i+1)/2 – 1); BSORT (i,i+(j-i+1)/2 - 1,+)
      - BITONIC(i+(j-i+1)/2 +1,j); BSORT (i+(j-i+1)/2 +1,j,-)

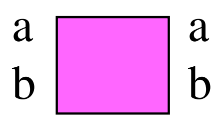
(note that any 2 keys are already a bitonic sequence)

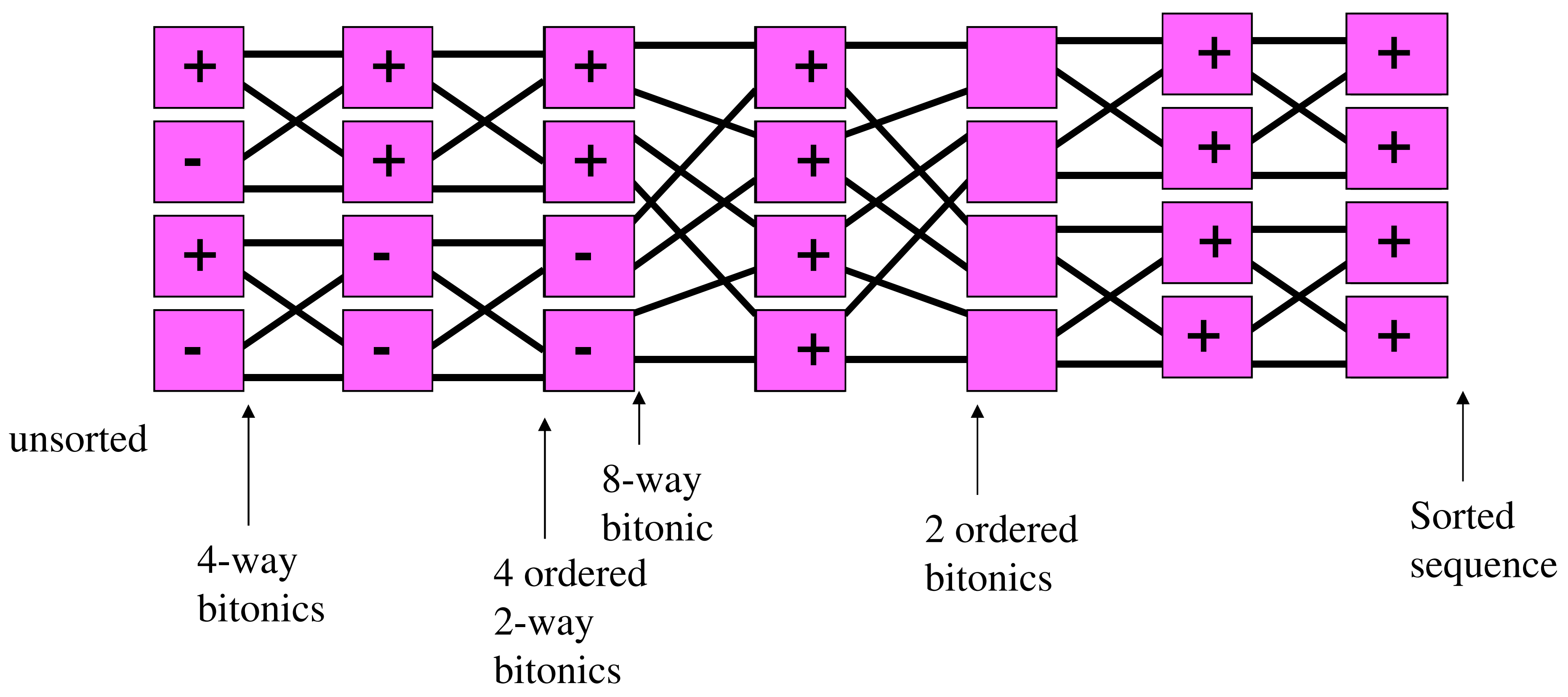




# Putting it all together

- Bitonic sort for 8 keys:





## Complexity of Bitonic Sort

$$\begin{aligned} T_{BSORT}(n = 2^j) &= 2 + T(2^{j-1}) \\ &= \cdots = 2(J-1) + T(2) = O(j) \\ &= O(\log n) \end{aligned}$$

$$\begin{aligned} T_{BITONIC}(n = 2^j) &= T(2^{j-1}) + 2(j-1) + 1 \\ &= \sum_{i=2}^{j-1} 2(i-1) + 1 \\ &= O(j^2) \\ &= O(\log^2 n) \end{aligned}$$

## Programming Issues

- Flow of data is assumed to transfer from stage to stage synchronously; usual issues with performance if algorithm is executed asynchronously
- Note that logical interconnect for each problem size is different
  - Bitonic sort must be mapped efficiently to target platform
- Unless granularity of platform very fine, multiple comparators will be mapped to each processor

## Review

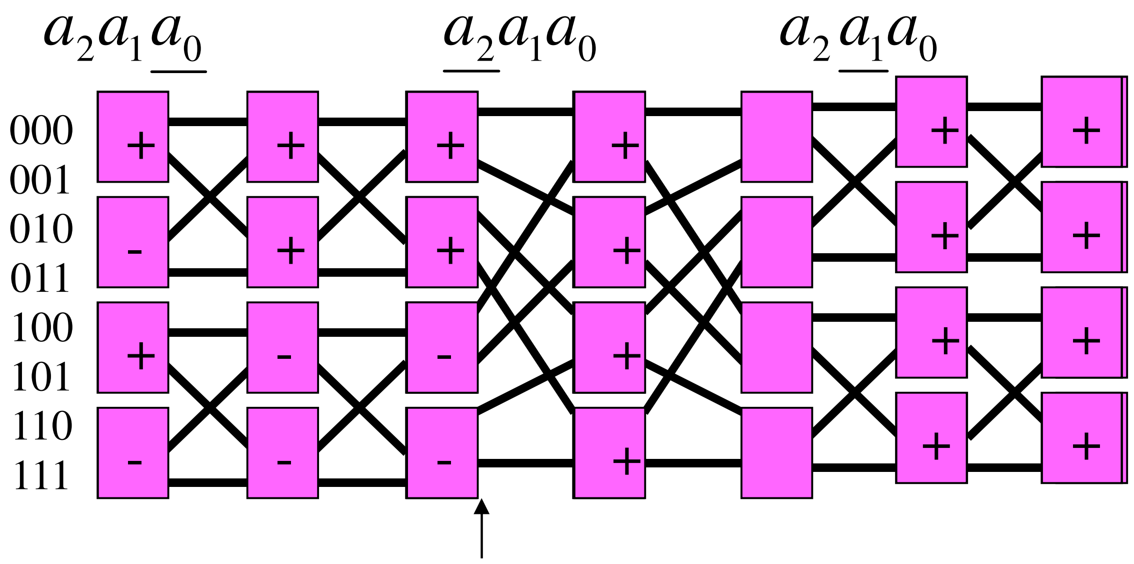
- What is a a synchronous computation? What is a fully synchronous computation?
- What is a bitonic sequence?
- What do the procedures BSORT and BITONIC do?
- How would you implement Bitonic Sort in a performance-efficient way?

# Mapping Bitonic Sort

For every stage the 2X2 switches compare keys which differ in a single bit

$$a_k \cdots a_{i+1} 1 a_{i-1} \cdots a_0$$

$$a_k \cdots a_{i+1} 0 a_{i-1} \cdots a_0$$

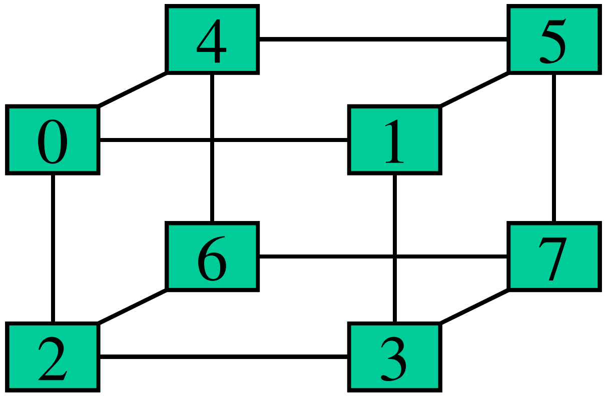


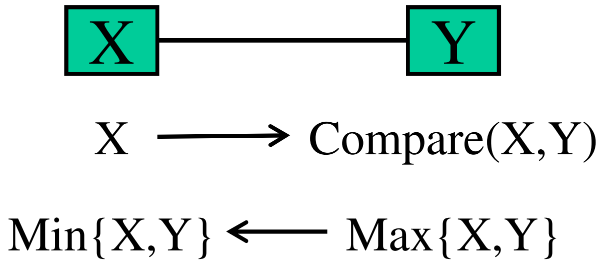
$a_2 a_1 \underline{a_0}$
$\underline{a_2} a_1 a_0$
$a_2 \underline{a_1} a_0$
000
001
010
011
100
101
110
111
+
-

# Supporting Bitonic Sort on a hypercube

- Switch comparisons can  performed in constant time on hypercube interconnect.

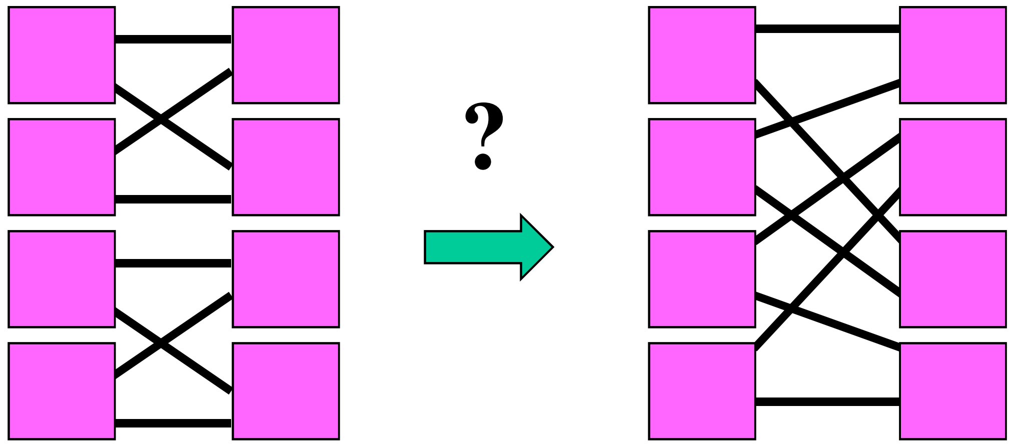
$$a_k \cdots a_{i+1} 1 a_{i-1} \cdots a_0$$
$$a_k \cdots a_{i+1} 0 a_{i-1} \cdots a_0$$





# Mappings of Bitonic Sort

- Bitonic sort on a multistage full shuffle
  - Small shuffles do not map 1-1 to larger shuffles!
  - Stone used a clever approach to map logical stages into full-sized shuffle stages while preserving O(log^2 n) complexity



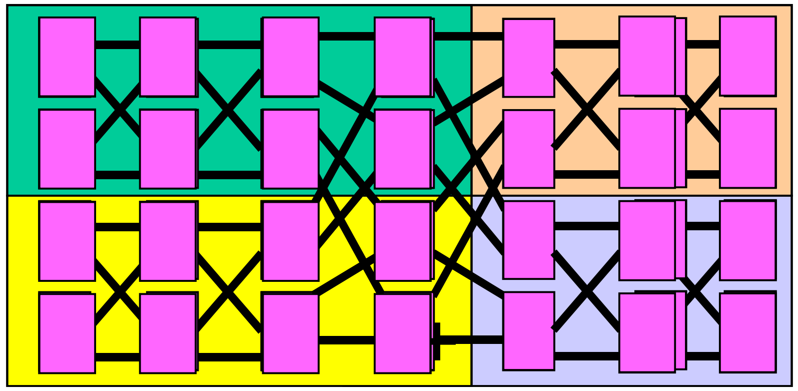
# Outline of Stone's Method

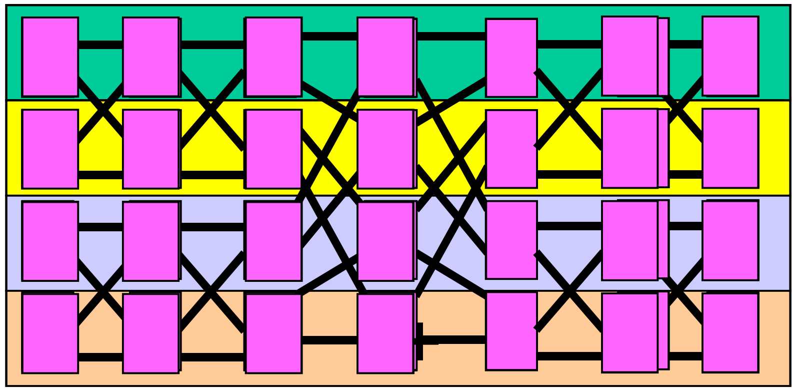
- Pivot bit = index being shuffled
- Stone noticed that for successive stages, the pivot bits are $\underline{a_0}, \underline{a_1, a_0}, \underline{a_2, a_1, a_0}, \cdots, \underline{a_i, a_{i-1}, \cdots, a_1, a_0}, \cdots$
- If the pivot bit is in place, each subsequent stage can be done using a full-sized shuffle (a_0 done with a single comparator)
- For pivot bit j, need k-j full shuffles to position bit j for comparison $a_k \cdots a_{j+1} a_j \cdots a_0 \rightarrow a_j \cdots a_0 a_k \cdots a_{j+1}$
- Complexity of Stone's method:

$$k + [(k-1)+1] + [(k-2)+1+1] + \cdots + [(k-k)+1+\cdots+1]$$
$$= k(k+1) = O(k^2) = O(\log^2 n)$$

# Many-one Mappings of Bitonic Sort

- For platforms where granularity is coarser, it will be more cost-efficient to map multiple comparators to one processor
- Several possible conventional mappings

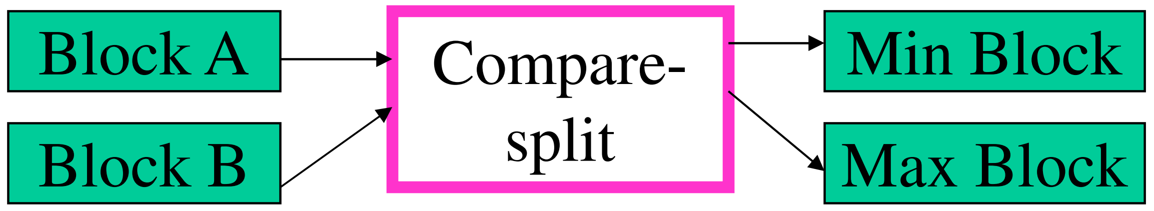




- Compare-split provides another approach …

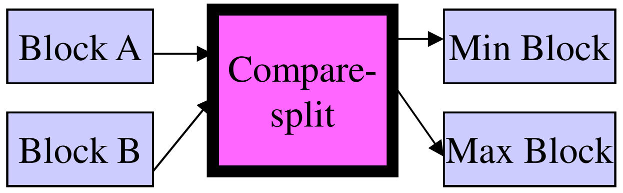
# Compare-Split

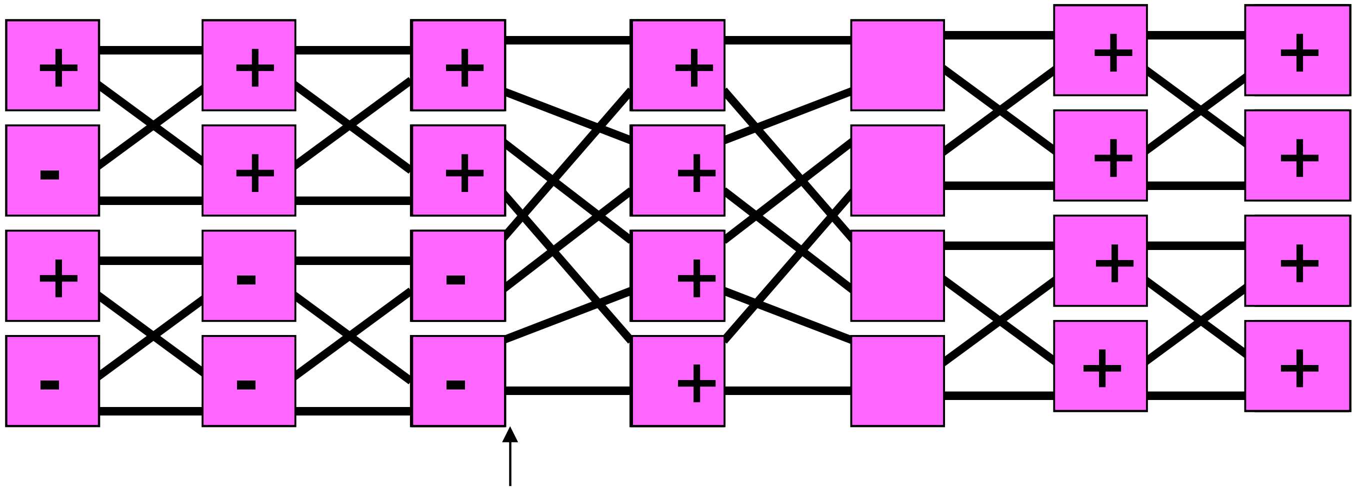
- For a block of keys, may want to use a compare-split operation (rather than compare-exchange) to accommodate multiple keys at a processor
- Idea is to assume that each processor is assigned a block of keys, rather than 2 keys
  - Blocks are already sorted with a sequential sort
  - To perform compare-split, a processor compares blocks and returns the smaller half of the aggregate keys as the min block and the larger half of the aggregate keys as the max block



# Compare-Split

- Each Block represents more than one datum
  - Blocks are already sorted with a sequential sort
  - To perform compare-split, a processor compares blocks and returns the smaller half of the aggregate keys as the min block and the larger half of the aggregate keys as the max block





## Performance Issues

- What is the complexity of compare-split?
- How do we optimize compare-split?
  - How many datum per block?
  - How to allocate blocks per processors?
  - How to synchronize intra-block sorting with inter-block communication?

## Conclusion on Systolic Arrays

- Advantages of systolic arrays are:
  - 1. Regularity and modular design(Perfect for VLSI implementation).
  - 2. Local interconnections(Implements algorithms locality).
  - 3. High degree of pipelining.
  - 4. Highly synchronized multiprocessing.
  - 5. Simple I/O subsystem.
  - 6. Very efficient implementation for great variety of algorithms.
  - 7. High speed and Low cost.
  - 8. Elimination of global broadcasting and modular expansibility.

## Disadvantages of systolic arrays

- The main disadvantages of systolic arrays are:
  - 1. Global synchronization limits due to signal delays.
  - 2. High bandwidth requirements both for periphery(RAM) and between PEs.
  - 3. Poor run-time fault tolerance due to lack of interconnection protocol.

# Parallel overhead.

Food for thought

- Running time for a program running on several processors including an allowance for parallel overhead compared with the ideal running time.
- There is often a point beyond which adding further processors doesn't result in further efficiency.
- There may also be a point beyond which adding further processors results in slower execution.

# Sources

1. Seth Copen Goldstein, CMU
2. David E. Culler, UC. Berkeley,
3. Keller@cs.hmc.edu
4. Syeda Mohsina Afroze
and other students of Advanced Logic Synthesis,
ECE 572, 1999 and 2000.
5. Berman