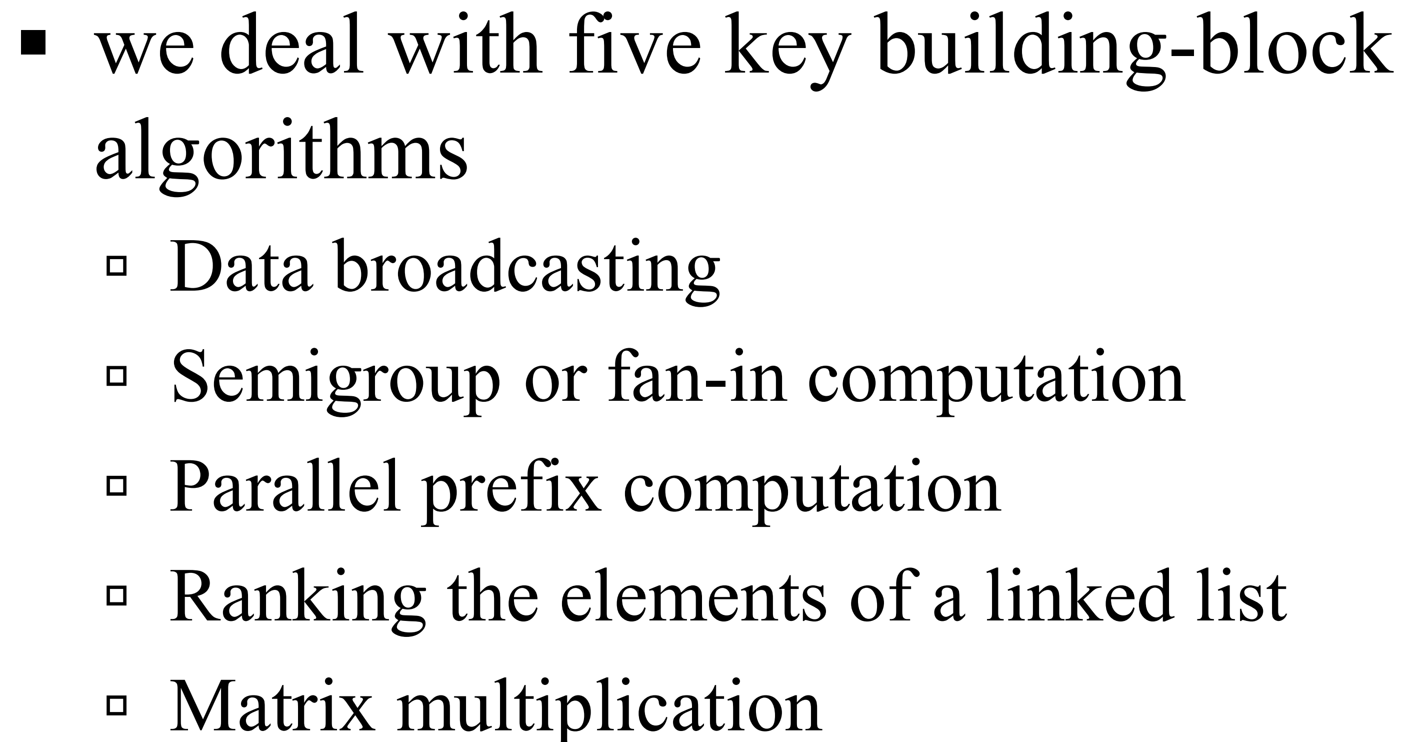
PARALLEL PROCESSING SYSTEMS

Chapter 5: PRAM and Basic Algorithms

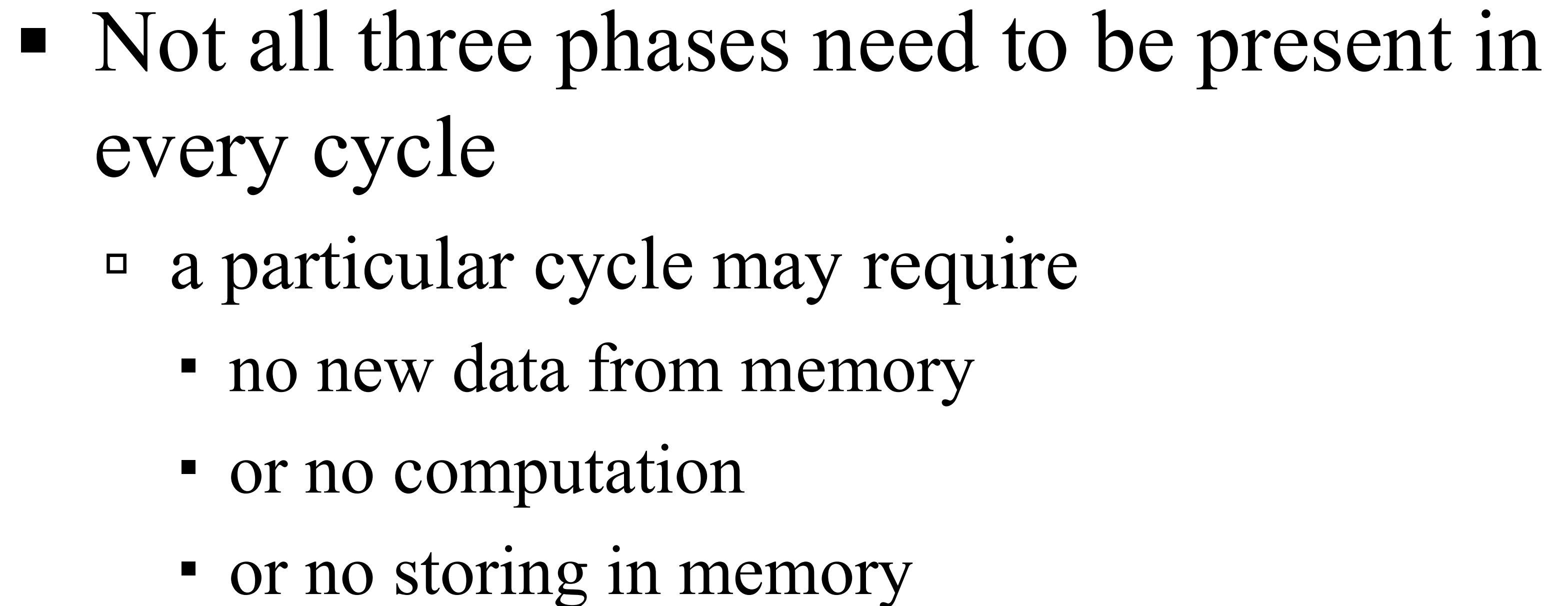
Introduction


we deal with five key building-block algorithms
Data broadcasting
Semigroup or fan-in computation
Parallel prefix computation
Ranking the elements of a linked list
Matrix multiplication

PRAM submodels and assumptions

- concurrent operation of p processors on m-word
  - accessible to all of them
  - in three phases of one cycle, processor i can do :
    1. Fetch an operand from the source address s_i in the shared memory
    2. Perform some computations on the data held in local registers
    3. Store a value into the destination address d_i in the shared memory

PRAM submodels and assumptions


Not all three phases need to be present in every cycle
a particular cycle may require
no new data from memory
or no computation
or no storing in memory

PRAM submodels and assumptions

- several processors may want to read/write the same memory location
  - four submodels of the PRAM

Writes to Same Location \ Reads from Same Location	Exclusive	Concurrent
Exclusive	EREW Least "Powerful", Most "Realistic"	CREW Default
Concurrent	ERCW Not Useful	CRCW Most "Powerful", Further Subdivided

PRAM submodels and assumptions

- CRCW sub-models
  - Undefined: In case of multiple writes, the value written is undefined (CRCW-U)
  - Detecting: A special code representing “detected collision” is written (CRCW-D).
  - Common: Multiple writes allowed only if all store the same value (CRCW-C)
    - sometimes called the consistent-write submodel.
  - Random: The value written is randomly chosen from among those offered (CRCWR)
    - sometimes called the arbitrary-write submodel.
  - Priority: The processor with the lowest index succeeds in writing its value (CRCW-P)
  - Max/Min: The largest/smallest of the multiple values is written (CRCW-M)
  - Reduction: The arithmetic sum (CRCW-S), logical AND (CRCW-A), logical XOR (CRCW-X), or some other combination of the multiple values is written.

PRAM submodels and assumptions

- Using computational power to order the submodels:
  - Two PRAM submodels are equally powerful if each can emulate the other with a constant-factor slowdown
  - A PRAM submodel is (strictly) less powerful than another submodel (denoted by the “<” symbol)
    - if there exist problems for which the former requires significantly more computational.
    - E.g., the CRCW-D PRAM submodel is less powerful than the one that writes the maximum value
      - the latter can find the largest number in a vector A of size p in a single step
        - Processor i reads A[i] and writes it to an agreed-upon location x
      - the former needs at least $\Omega(\log n)$ steps.

EREW < CREW < CRCW-D < CRCW-C < CRCW-R < CRCW-P

Data broadcasting

- Simple, or one-to-all, broadcasting
  - one processor needs to send a data value to all other processors.
  - Trivial in the CREW or CRCW submodels
    - sending processor can write the data value into a memory location
    - all processors reading that data value in the next cycle.
    - done in $\Theta(1)$ steps.
- Multicasting within groups
  - equally simple if each processor knows its group membership(s)
    - only members of each group read the multicast data for that group.
- All-to-all broadcasting
  - each of the p processors needs to send a data value to all other processors
  - can be done through p separate broadcast operations
    - $\Theta(p)$ steps, which is optimal.

Data broadcasting

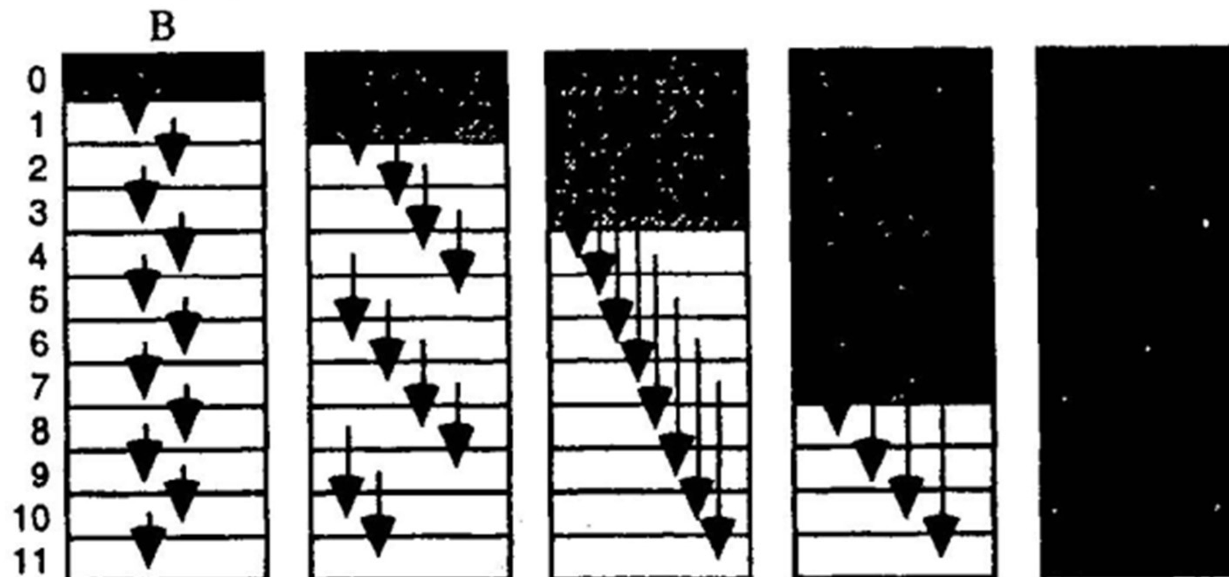
- Previous scheme is inapplicable to the EREW
- The simplest scheme for EREW
  - make p copies of the data value
    - say in a broadcast vector B of length p
    - initially, Processor i writes its data value into B[0].
    - Use recursive doubling to copy B[0] into all elements of B
      - in $\lceil \log 2 \, p \rceil$ steps
  - let each processor read its own copy by accessing B[j].

Data broadcasting

- The simplest scheme for EREW
  - Recursive doubling

Making p copies of $B[0]$ by recursive doubling

for $k = 0$ to $\lceil \log_2 p \rceil - 1$ Processor j, $0 \le j < p$, do
  Copy $B[j]$ into $B[j + 2^k]$
endfor



Data broadcasting

- The simplest scheme for EREW
  - allow us to use the idle processors for other tasks
    - to speed up algorithm execution
    - or to reduce the memory access traffic
      - when the algorithm is ported to a physical shared-memory machine
  - in Step k of doubling process, only the first 2^k processors need to be active

Data broadcasting

- The simplest scheme for EREW

<u>EREW PRAM algorithm for broadcasting by Processor i</u>

Processor i write the data value into $B[0]$
$s := 1$
while $s < p$ Processor j, $0 \le j < \min(s, p - s)$, do
    Copy $B[j]$ into $B[j + s]$
    $s := 2s$
endwhile
Processor j, $0 \le j < p$, read the data value in $B[j]$

Data broadcasting

- The simplest scheme for EREW
  - It is optimal for EREW PRAM
    - initially a single copy of the data value exists
      - at most one other processor can get the value
        - through a memory access in the first step
    - In the second step
      - two more processors can become aware of the data value
    - Continuing in this manner
      - at least $\lceil \log 2\ p \rceil$ read–write cycles are necessary

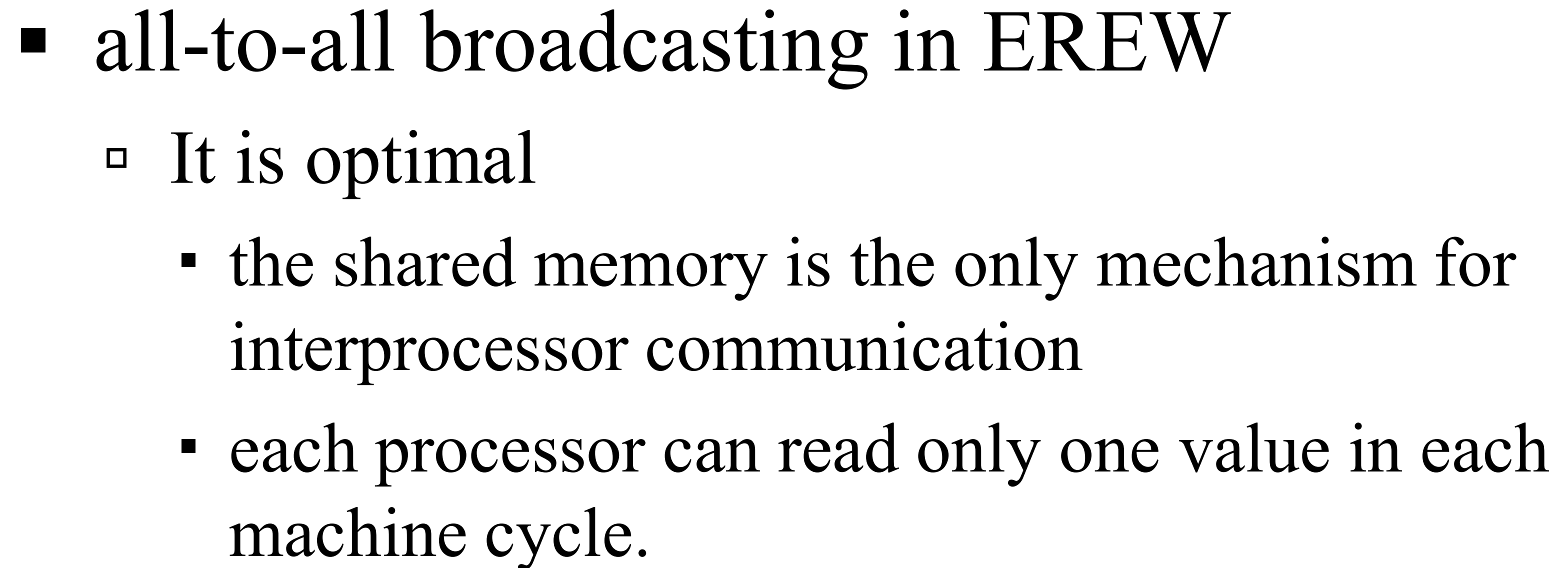
Data broadcasting

- all-to-all broadcasting in EREW
  - let Processor j write its value into B[j]
    - rather than into B[0].
  - Each processor then reads the other p – 1 values in p – 1 memory accesses
    - To ensure that all reads are exclusive
      - Processor j
        - begins reading the values starting with B[j + 1]
        - wrapping around to B[0] after reading B[p – 1].

<u>EREW PRAM algorithm for all-to-all broadcasting</u>

Processor j, $0 \le j < p$, write own data value into $B[j]$
for $k = 1$ to $p - 1$ Processor j, $0 \le j < p$, do
  Read the data value in $B[(j + k) \bmod p]$
endfor

Data broadcasting



- all-to-all broadcasting in EREW
  - It is optimal
    - the shared memory is the only mechanism for interprocessor communication
    - each processor can read only one value in each machine cycle.

Data broadcasting

- a naive sorting algorithm
  - let Processor j compute the rank R[j] of the data element S[j]
    - examining all other data elements
    - counting the number of elements S[l] that are smaller than S[j].
  - then store S[j] into S[R[j]].

Naive EREW PRAM sorting algorithm using all-to-all broadcasting

Processor j, $0 \le j < p$, write 0 into $R[j]$
for $k = 1$ to $p - 1$ Processor j, $0 \le j < p$, do
  $l := (j + k) \bmod p$
  if $S[l] < S[j]$ or $S[l] = S[j]$ and $l < j$
  then $R[j] := R[j] + 1$
  endif
endfor
Processor j, $0 \le j < p$, write $S[j]$ into $S[R[j]]$

Semigroup or fan-in computation

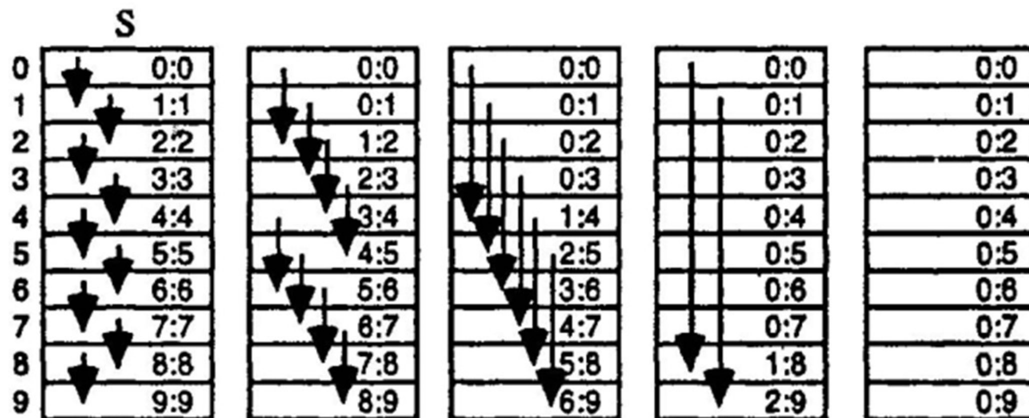
- trivial for a CRCW of the "reduction"
  - if the reduction operator happens to be $\otimes$
  - E.g., sum of p values, one per processor
    - can be done in a single cycle
    - each processor writing its corresponding value into a common

Semigroup or fan-in computation

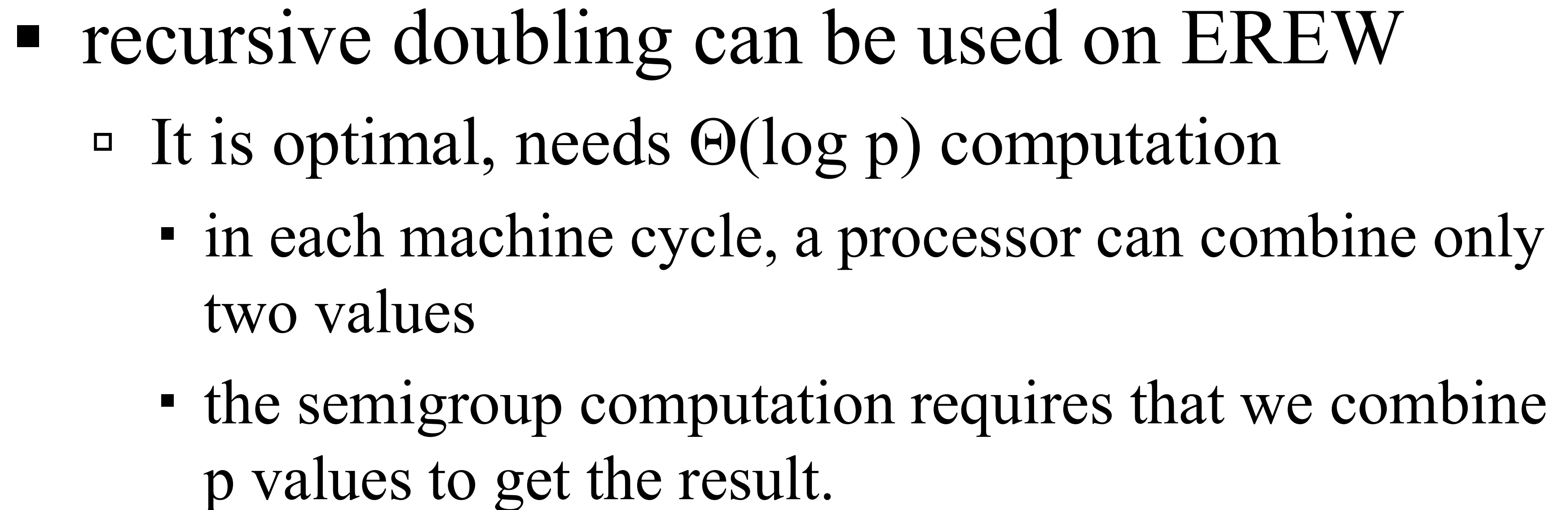
- recursive doubling can be used on EREW
  - virtually identical EREW broadcasting
    - with the copying operation replaced by a (⊗) operation.

EREW PRAM semigroup computation algorithm

Processor j, $0 \le j < p$, copy $X[j]$ into $S[j]$
$s := 1$
while $s < p$ Processor j, $0 \le j < p - s$, do
  $S[j + s] := S[j] \otimes S[j + s]$
  $s := 2s$
endwhile
Broadcast $S[p - 1]$ to all processors



Semigroup or fan-in computation



- recursive doubling can be used on EREW
  - It is optimal, needs $\Theta(\log p)$ computation
    - in each machine cycle, a processor can combine only two values
    - the semigroup computation requires that we combine p values to get the result.

Semigroup or fan-in computation

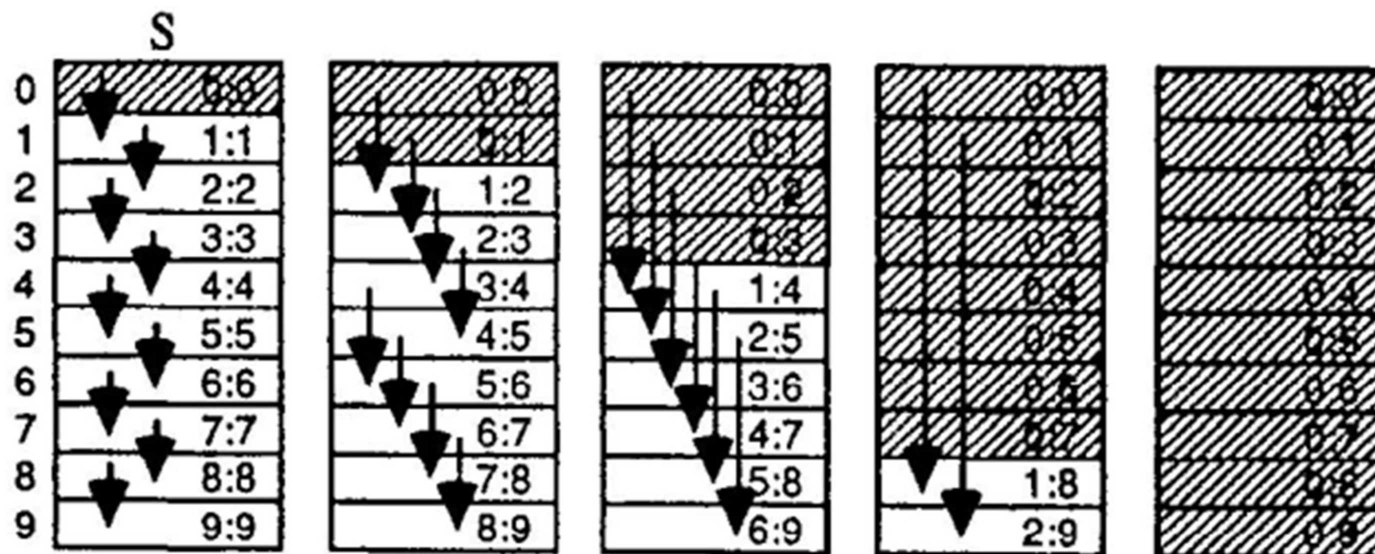
- When each processors have n/p elements
  - each processor
    - first combines its n/p elements
      - n/p steps to get a single value.
    - Then, the algorithm just discussed is used
      - the first step replaced by copying the result of the above into S[j].

Semigroup or fan-in computation

- Speed-up and efficiency
  - final broadcasting takes log 2 p steps
  - algorithm needs $n/p + 2 \log_2 p$ EREW steps in all
  - $n/(n/p + 2 \log_2 p)$ speed-up over the sequential version
  - If $p = \Theta(n)$
    - a sublinear speed-up of $\Theta(n/\log n)$ is obtained.
    - efficiency is $\Theta(n/\log n)/\Theta(n) = \Theta(1/\log n)$.

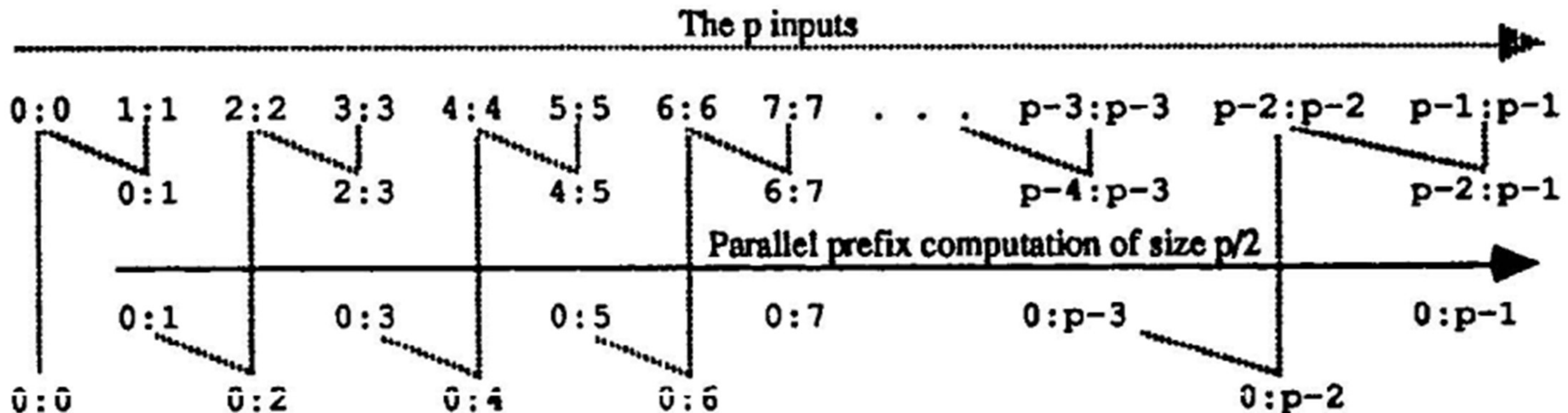
Parallel prefix computation

- Can be done like the first phase of the semigroup computation
- as we find the semigroup result in S[p – 1]
  - all partial prefixes are also obtained in the previous elements of S



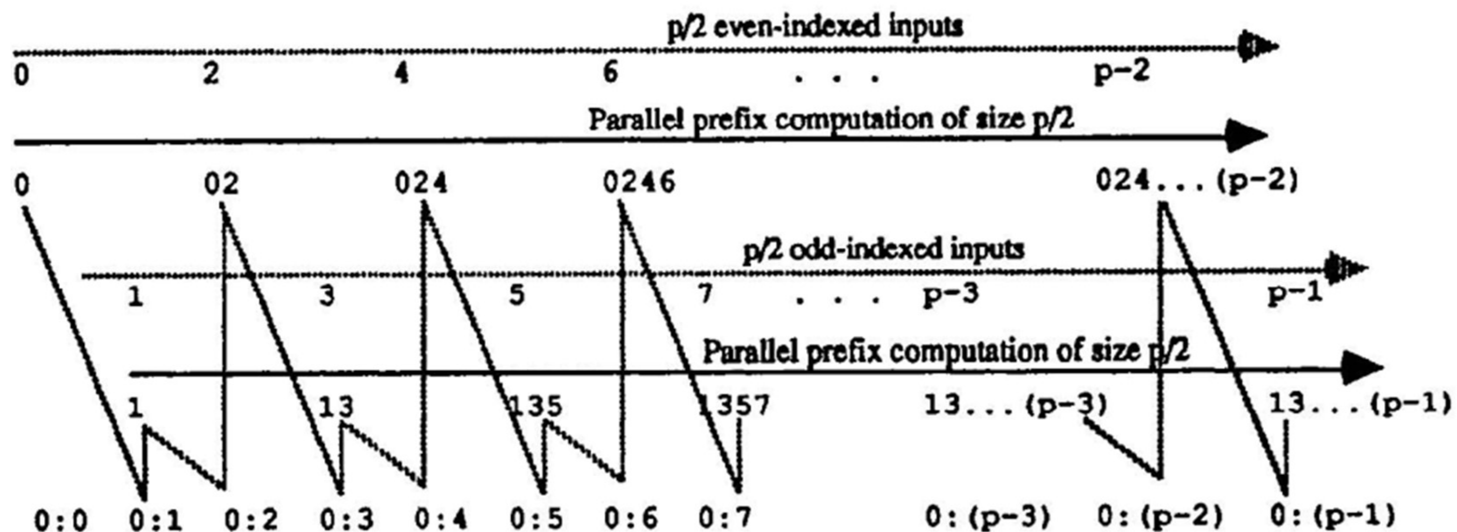
Parallel prefix computation

- a solution using divide and conquer
  - Sub-problem 1: computing odd-indexed results s1 , s3, s5, . . .
    - Combine pairs of consecutive elements to obtain a list of half the size
    - Perform parallel prefix computation on this list
  - Sub-problem 2: computing the even-indexed results s0, s2, s4 , . . .
    - combine even-indexed inputs with their next odd-indexed result
      - By a single PRAM step
  - $T(p) = T(p/2) + 2 \Rightarrow T(p) = 2 \log_2 p$



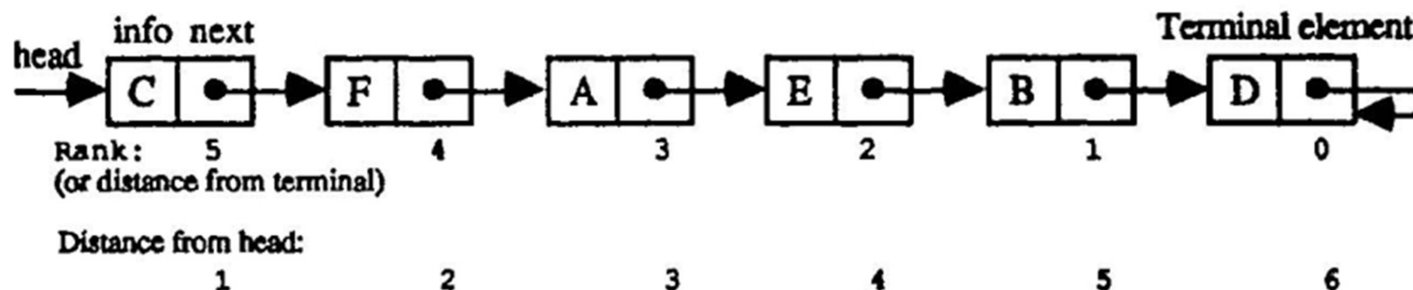
Parallel prefix computation

- Another solution using divide and conquer
  - view the input list as composed of two odd/even sublists
  - Perform parallel prefix separately on each sublist
  - Obtain final results by pairwise combination of adjacent partial results
    - a single PRAM step
  - $T(p) = T(p/2) + 1 \Rightarrow T(p) = \log_2 p$
  - applicable only if the operator $\otimes$ is commutative



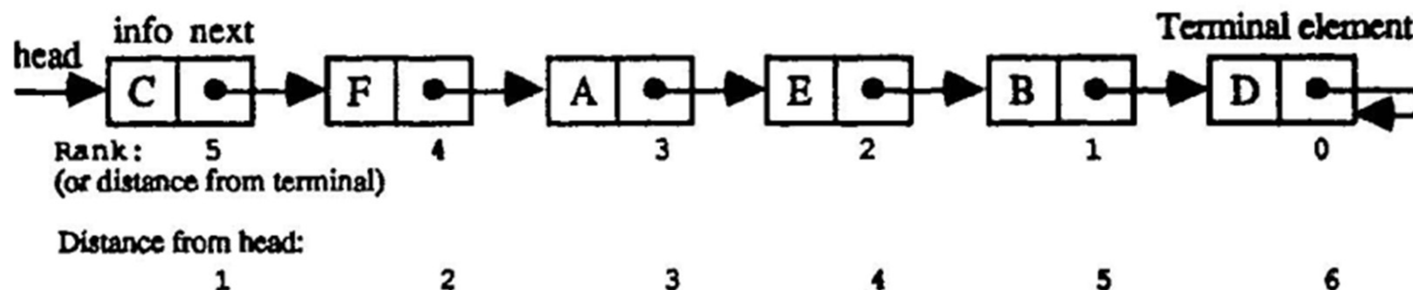
Ranking the elements of a linked list

- rank the list elements in terms of the distance of each to the terminal element
  - It is important
    - it is a very useful building block in many
    - it demonstrates how a problem that seems hopelessly sequential can be efficiently parallelized



Ranking the elements of a linked list

- A sequential algorithm
  - requires $\Theta(p)$ time.
    - list must be traversed once to determine the distance of each element from the head
    - second pass, through the list to compute all the ranks

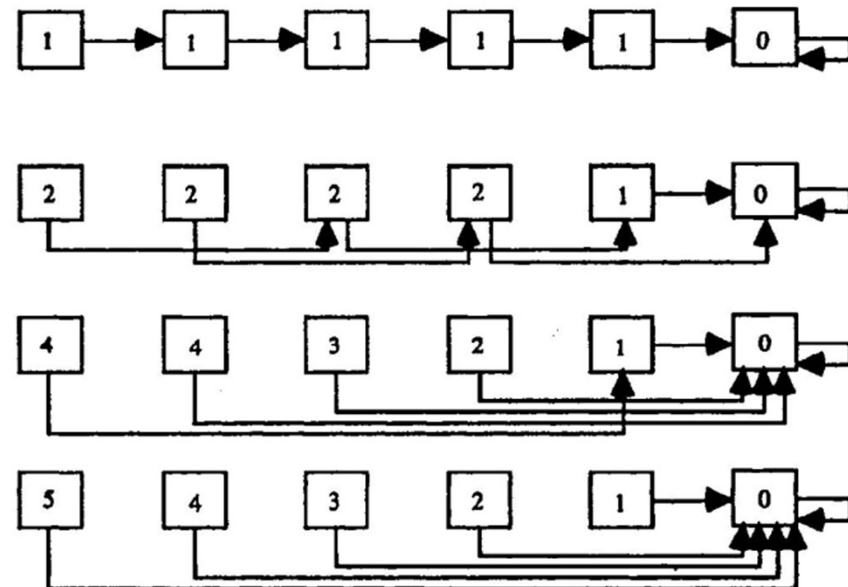


Ranking the elements of a linked list

- parallel solution (pointer jumping)
  - Processor j, $0 \leq j < p$, is responsible for computing rank [j]
  - number of elements that are skipped doubles with each iteration
    - the number of iterations is logarithmic in p

PRAM list ranking algorithm (via pointer jumping)

```
Processor j, 0 ≤ j < p, do {initialize the partial ranks}
   if next[j] = j
   then rank[j] := 0
   else rank[j] := 1
   endif
while rank[next[head]] ≠ 0 Processor j, 0 ≤ j < p, do
   rank[j] := rank[j] + rank[next[j]]
   next[j] := next[next[j]]
endwhile
```



Matrix multiplication

- The product C of m×m matrices A and B:

$$c_{ij} = \sum_{k=0}^{m-1} a_{ik} b_{kj}$$

- O(m³)-step sequential algorithm

Sequential matrix multiplication algorithm

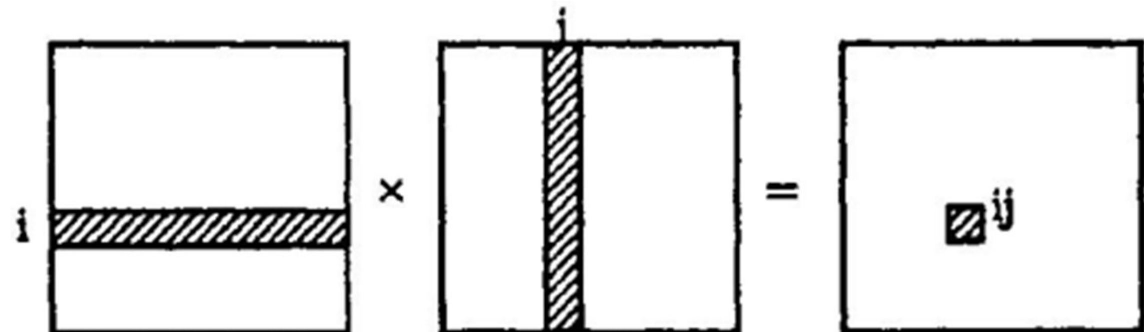
```
for i = 0 to m – 1 do
    for j = 0 to m – 1 do
        t := 0
        for k = 0 to m – 1 do
            t := t + a_ik b_kj
        endfor
        c_ij := t
    endfor
endfor
```

- If the PRAM has $p = m^3$ processors
  - can be done in $\Theta(\log m)$ time
    - using one processor to compute each product $a_{ik}b_{kj}$
    - then allowing groups of m processors to perform m-input summations (semigroup computation)
      - in $\Theta(\log m)$ time.
  - is not a practical solution.
    - we are usually not interested in parallel processing unless m is large

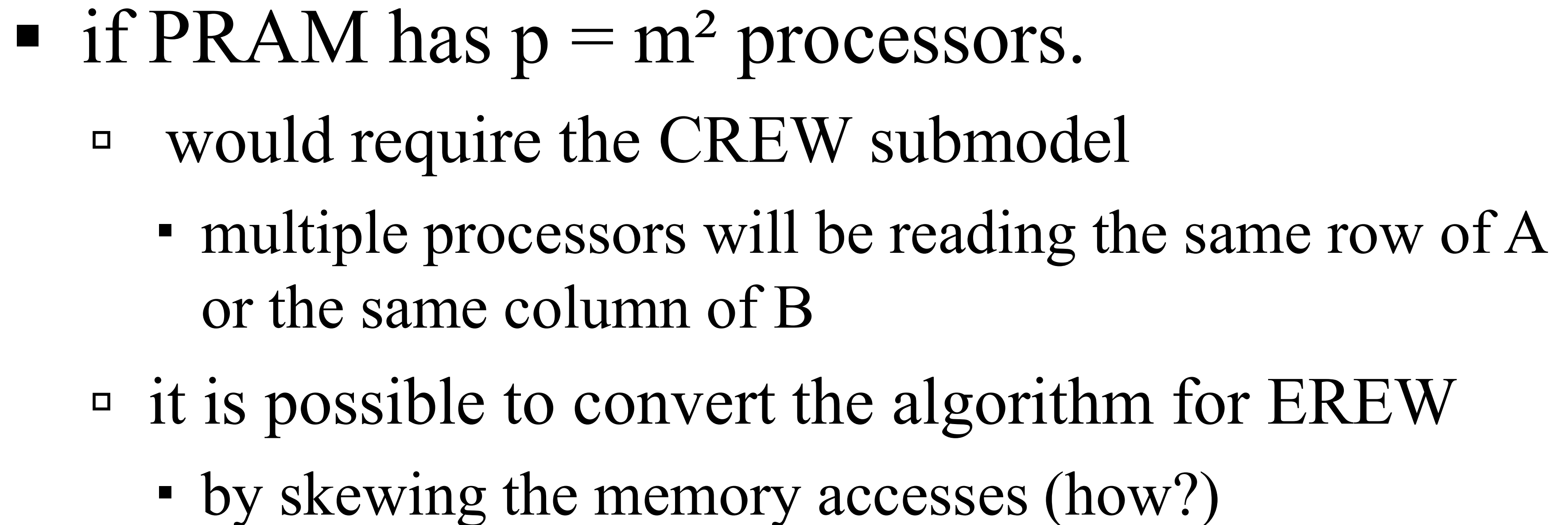
Matrix multiplication

- if PRAM has $p = m^2$ processors.
  - can be done in $\Theta(m)$ by parallelizing the i and j loops
    - using one processor to compute each c_{ij}
      - The processor
        - reads the elements of Row i in A and the elements of Column j in B
        - multiplies their corresponding kth elements
        - adds each of the obtained products to a running total t.

```
Processor (i, j), 0 ≤ i, j < m, do
begin
   t := 0
   for k = 0 to m – 1 do
      t := t + a_ik b_kj
   endfor
   c_ij := t
end
```



Matrix multiplication


if PRAM has p = m² processors.
would require the CREW submodel
multiple processors will be reading the same row of A or the same column of B
it is possible to convert the algorithm for EREW
by skewing the memory accesses (how?)

Matrix multiplication

- if PRAM has p = m processors
  - can be done in $\Theta(m^2)$ by parallelizing the i loop
    - Processor i compute the m elements in Row i of the product matrix C
      - Processor i
        - read the elements of Row i in A and the elements of all columns in B
        - multiply their corresponding kth elements
        - add each of the obtained products to a running total t.

```
for j = 0 to m – 1 Processor i, 0 ≤ i < m, do
    t := 0
    for k = 0 to m – 1 do
        t := t + a_ik b_kj
    endfor
    c_ij := t
endfor
```

Matrix multiplication

- if PRAM has p = m processors
  - each processor reads a different row of the matrix A
    - no concurrent reads from A are ever attempted.
  - however, all m processors access the same element b_{kj} at the same time.
  - one can skew the memory accesses for B for the EREW

for $j = 0$ to $m - 1$ Processor i, $0 \le i < m$, do
  $t := 0$
  for $k = 0$ to $m - 1$ do
    $t := t + a_{ik} b_{kj}$
  endfor
  $c_{ij} := t$
endfor

Matrix multiplication

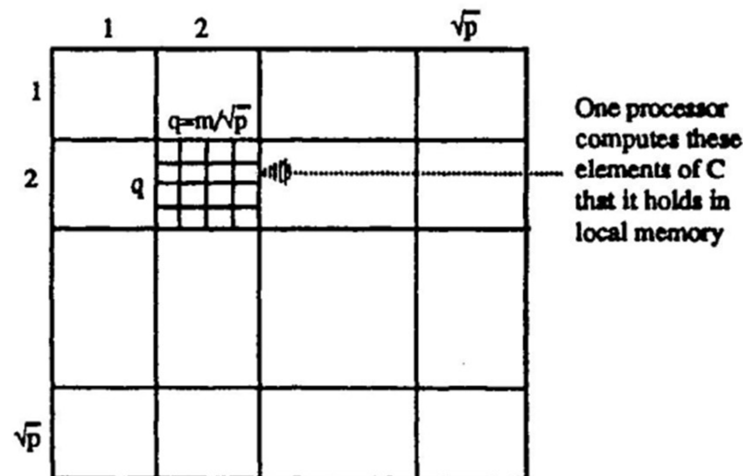
- If the number of processors is even less than m
  - Occurs in many practical situations
  - parallelizing the k loop is not good
    - has data dependencies
  - parallelizing the j loop is not good
    - imply m synchronizations of the processors once at the end of each i iteration
      - assuming the SPMD model

- If the number of processors is even less than m
  - parallelizing the i loop
    - Processor i compute a set of m/p rows in the result matrix C
      - Rows i, i + p, i + 2p, . . . , i + (m/p – 1)p
    - almost linear speedup for UMA
      - speed-up of about 22 for 24 processors multiplying two 256 × 256 matrices

- If the number of processors is even less than m
  - parallelizing the i loop
    - Has drawback for NUMA
      - Low computation to memory access ratio
        - each element of B is fetched m/p times
        - with only two arithmetic operations for each element
          - one multiplication and one addition

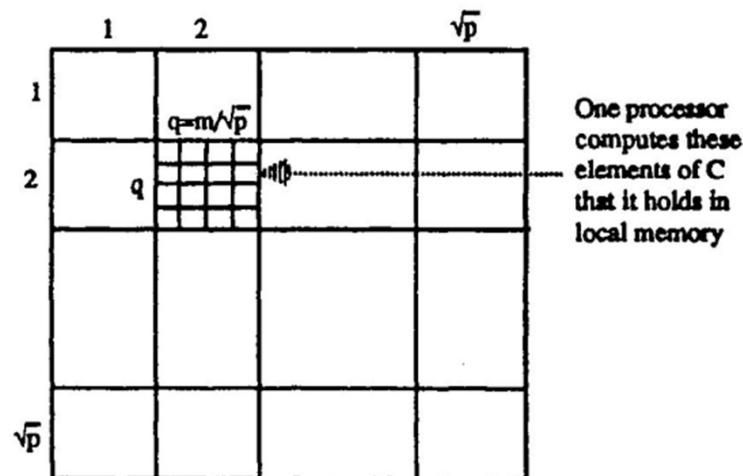
Matrix multiplication

- If the number of processors is even less than m
  - Block matrix multiplication
    - divide the m × m matrices A, B, and C into p blocks of size q × q, where $q = m/\sqrt{p}$
    - then multiply the m×m matrices using $\sqrt{p} \times \sqrt{p}$ matrix multiplication with $\sqrt{p}^2 = p$ processors where
      - the terms in the algorithm statement t := t $+a_{ik}b_{kj}$ are now q × q matrices
      - Processor (i, j) computes Block (i, j) of the result matrix C.



Matrix multiplication

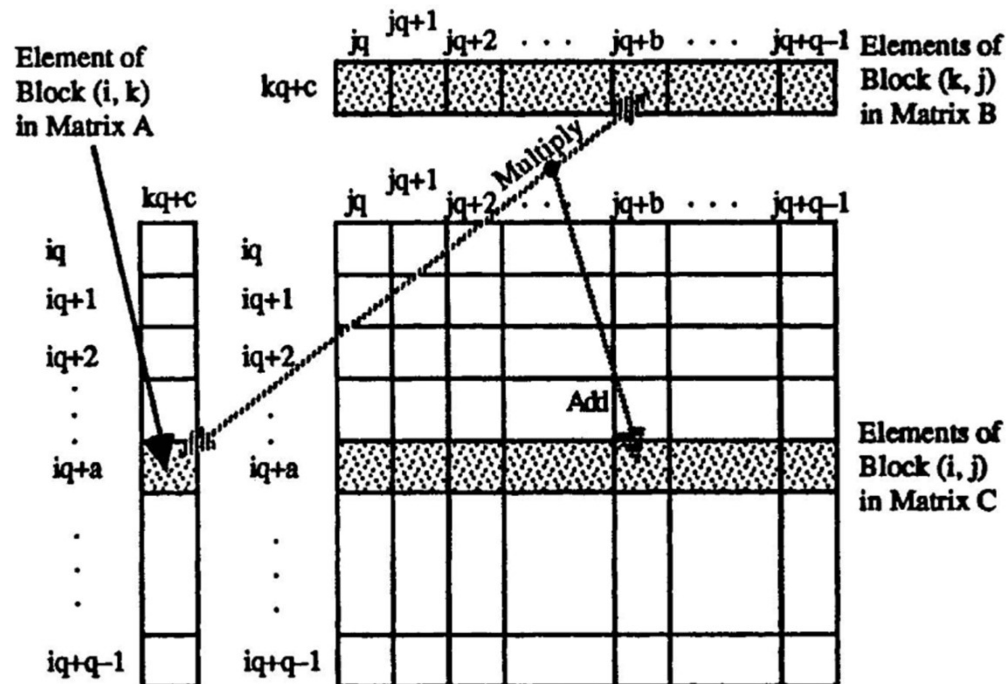
- If the number of processors is even less than m
  - Block matrix multiplication
    - the algorithm is like our second algorithm above
      - the statement $t := t + a_{ik}b_{kj}$ replaced by a sequential $q \times q$ matrix multiplication algorithm



- If the number of processors is even less than m
  - Block matrix multiplication
    - The assumption here is
      - Processor (i, j) has sufficient local memory to hold
        - Block (i, j) of the result matrix C (q^2 elements)
        - one block-row of the matrix B
          - the q elements in Row kq + c of Block (k, j) of B.

Matrix multiplication

- If the number of processors is even less than m
  - Block matrix multiplication
    - Elements of A can be brought in one at a time.
      - as element in Row iq+a of Column kq+c in Block (i, k) of A is brought in
        - it is multiplied in turn by the locally stored q elements of B
        - and the results added to the appropriate q elements of C



Matrix multiplication

- If the number of processors is even less than m
  - Block matrix multiplication
    - Each multiply–add computation on $q \times q$ blocks needs
      - $2q^2 = 2m^2/p$ memory accesses to read the blocks
      - $2q^3$ arithmetic operations.
    - So, q arithmetic operations are performed for each memory access
      - better performance will be achieved as a result of improved locality