CSE-40533

Introduction to Parallel Processing

Chapter 5

PRAM and Basic Algorithms

- **Define PRAM and its various submodels**
- **Show PRAM to be a natural extension of the sequential computer (RAM)**
- **Develop five important parallel algorithms that can serve as building blocks**

5.1 PRAM Submodels and Assumptions

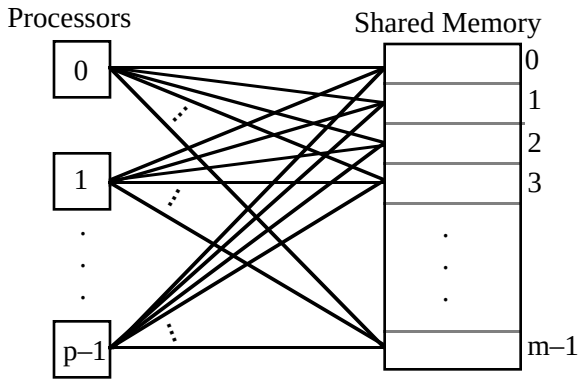


Fig. 4.6. Conceptual view of a parallel random-access machine (PRAM).

Processor i can do the following in 3 phases of one cycle:
1. Fetch an operand from address s_i in shared memory
2. Perform computations on data held in local registers
3. Store a value into address d_i in shared memory

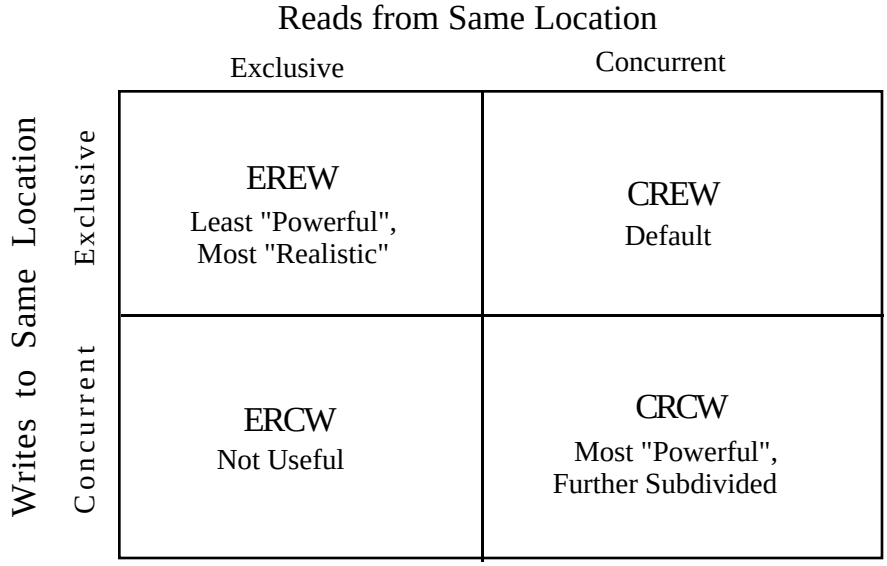


Fig. 5.1 Submodels of the PRAM model.

CRCW PRAM is classified according to how concurrent writes are handled. These submodels are all different from each other and from EREW and CREW.

Undefined: In case of multiple writes, the value written is undefined (CRCW-U)

Detecting: A code representing "detected collision" is written (CRCW-D)

Common: Multiple writes allowed only if all store the same value (CRCW-C); this is sometimes called the consistent-write submodel

Random: The value written is randomly chosen from those offered (CRCW-R)

Priority: The processor with the lowest index succeeds in writing (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 another combination of the multiple values is written.

One way to order these submodels is by their computational power:

$$\text{EREW} < \text{CREW} < \text{CRCW-D} < \text{CRCW-C} < \text{CRCW-R} < \text{CRCW-P}$$

Theorem 5.1: A p-processor CRCW-P (priority) PRAM can be simulated (emulated) by a p-processor EREW PRAM with a slowdown factor of $\Theta(\log p)$.

5.2 Data Broadcasting

Broadcasting is built-in for the CREW and CRCW models

EREW broadcasting: make p copies of the data in a broadcast vector B

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

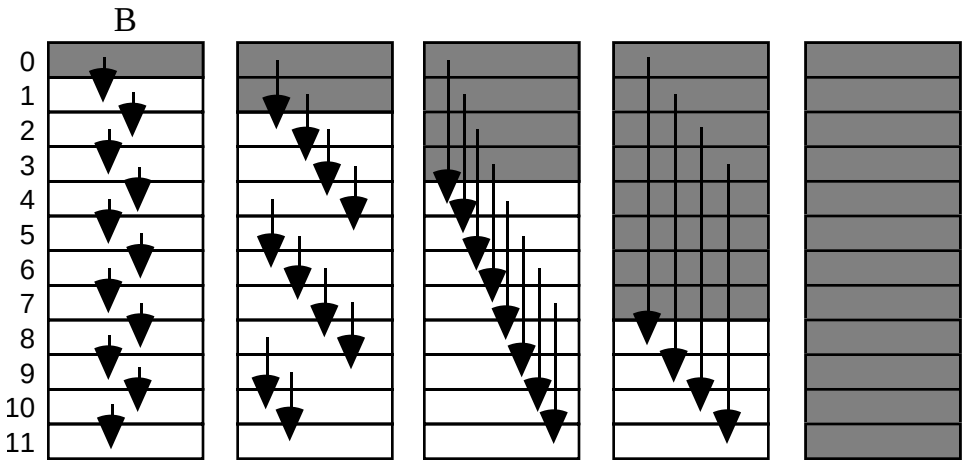


Fig. 5.2. Data broadcasting in EREW PRAM via recursive doubling.

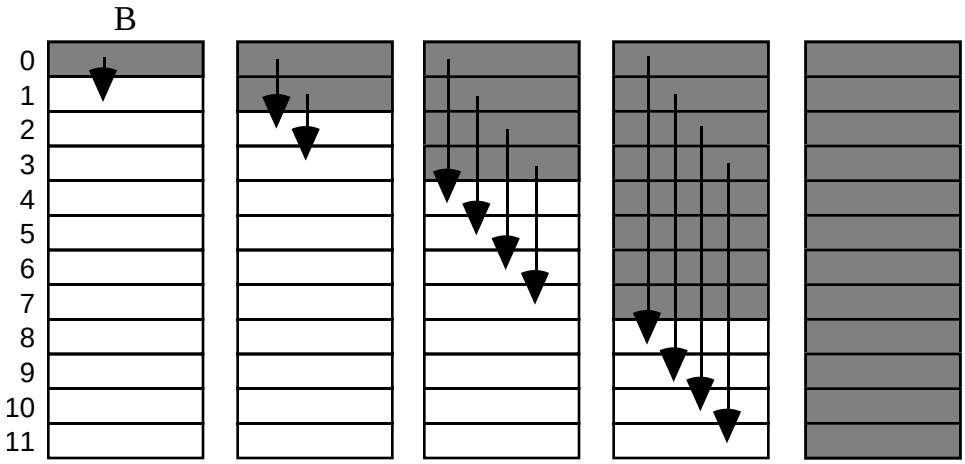


Fig. 5.3. EREW PRAM data broadcasting without redundant copying.

EREW PRAM algorithm for broadcasting by Processor i

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

EREW PRAM algorithm for all-to-all broadcasting

```
Processor j, 0 ≤ j < p, write own data value into B[j]
for k = 1 to p – 1 Processor j, 0 ≤ j < p, do
      Read the data value in B[(j + k) mod p]
endfor
```

Both of the preceding algorithms are time-optimal (shared memory is the only communication mechanism and each processor can read but one value per cycle)

In the following naive sorting algorithm, processor j determines the rank R[j] of its data element S[j] by examining all the other data elements; it then writes S[j] in element R[j] of the output (sorted) vector

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

```
Processor j, 0 ≤ j < p, write 0 into R[j]
for k = 1 to p – 1 Processor j, 0 ≤ j < p, do
    l := (j + k) mod 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 ≤ j < p, write S[j] into S[R[j]]
```

This O(p)-time algorithms is far from being optimal

5.3 Semigroup or Fan-in Computation

This computation is trivial for a CRCW PRAM of the reduction variety if the reduction operator happens to be ⊗

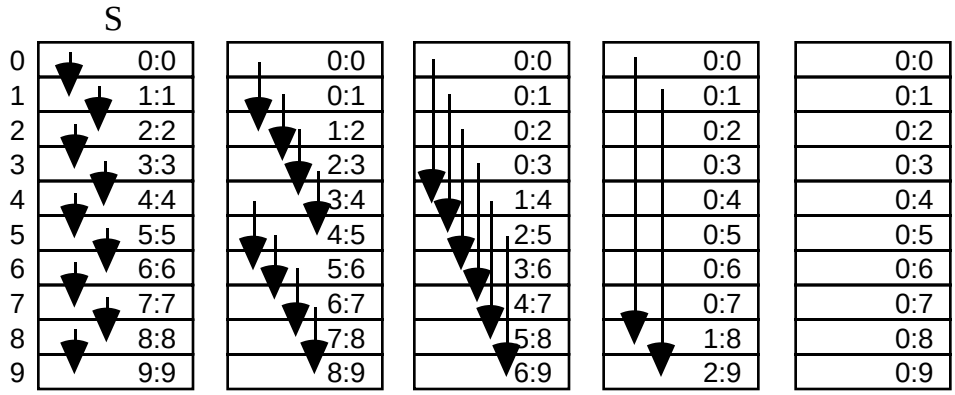


Fig. 5.4. Semigroup computation in EREW PRAM.

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] ⊗ S[j + s]
    s := 2s
endwhile
Broadcast S[p – 1] to all processors

The preceding algorithm is time-optimal (CRCW can do better: problem 5.16)

Speed-up = $p/\log_2 p$

Efficiency = Speed-up/p = $1/\log_2 p$

$$\text{Utilization} = \frac{W(p)}{pT(p)} \approx \frac{(p-1)+(p-2)+(p-4)+ \ \ldots \ +(p-p/2)}{p \ \log_2 p} \approx 1 - 1/\log_2 p$$

Semigroup computation with each processor holding n/p data elements:

Each processor combine its sublist n/p steps

Do semigroup computation on results $\log_2 p$ steps

$$\text{Speedup}(n, p) = \frac{n}{n/p + 2\log_2 p} = \frac{p}{1 + (2p\log_2 p)/n}$$

$$\text{Efficiency}(n, p) = \text{Speedup}/p = \frac{1}{1 + (2p\log_2 p)/n}$$

For $p = \Theta(n)$, a sublinear speedup of $\Theta(n/\log n)$ is obtained

The efficiency in this case is $\Theta(n/\log n)/\Theta(n) = \Theta(1/\log n)$

Limiting the number of processors to $p = O(n/\log n)$, yields:

Speedup(n, p) = $n/O(\log n) = \Omega(n/\log n) = \Omega(p)$

Efficiency(n, p) = $\Theta(1)$

Using fewer processors than tasks = parallel slack

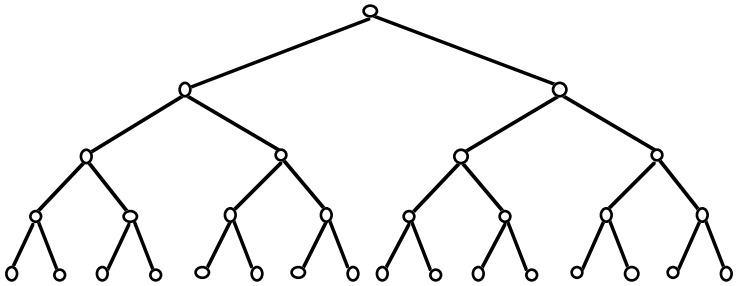


Fig. 5.5. Intuitive justification of why parallel slack helps improve the efficiency.

5.4 Parallel Prefix Computation

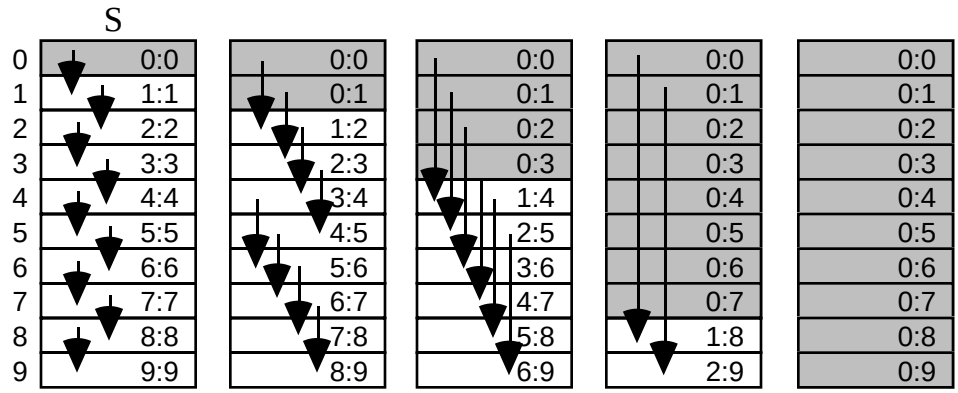


Fig. 5.6. Parallel prefix computation in EREW PRAM via recursive doubling.

Two other solutions, based on divide and conquer

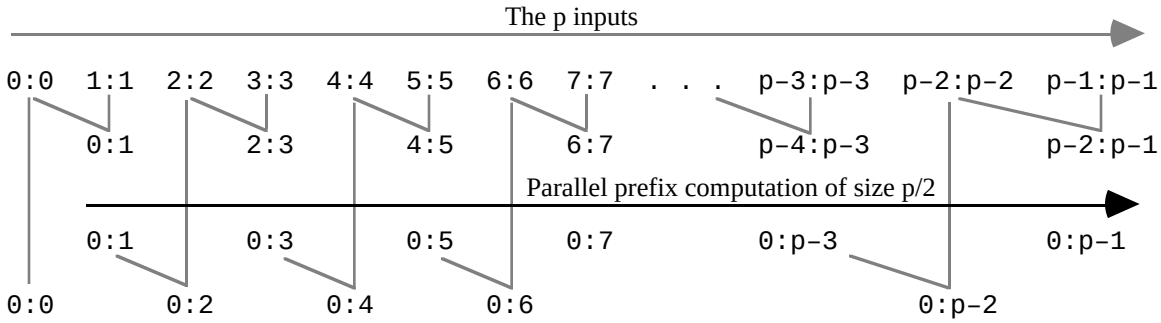


Fig. 5.7 Parallel prefix computation using a divide-and-conquer scheme.

$$T(p) = T(p/2) + 2 = 2 \log_2 p$$

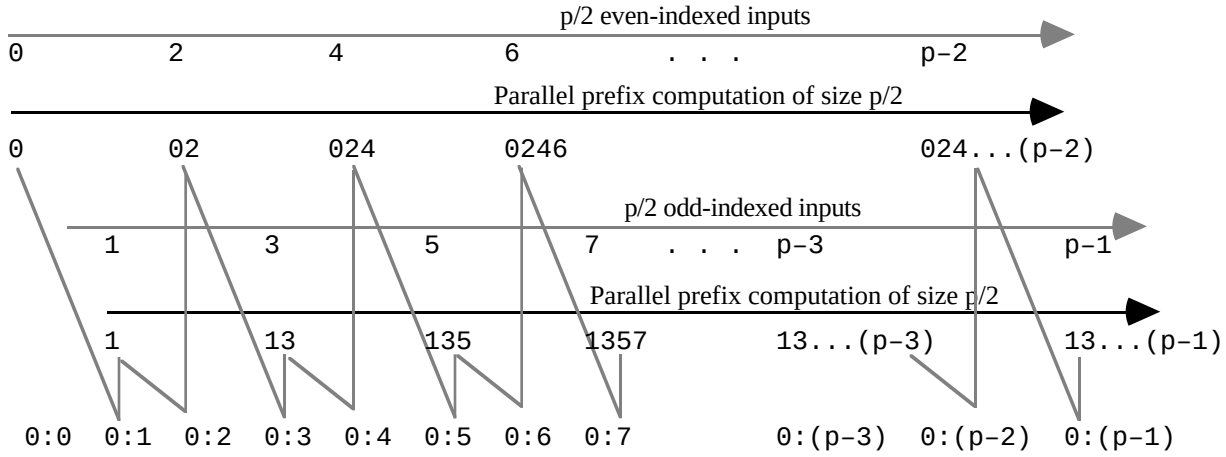


Fig. 5.8. Another divide-and-conquer scheme for parallel prefix computation.

$$T(p) = T(p/2) + 1 = \log_2 p$$ Requires commutativity

5.5 Ranking the Elements of a Linked List

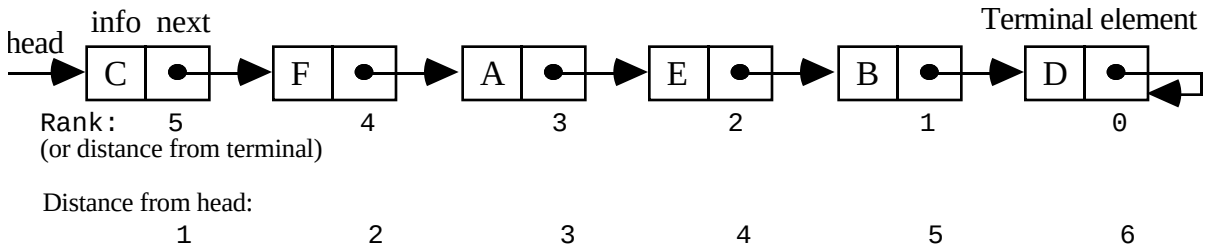


Fig. 5.9. Example linked list and the ranks of its elements.

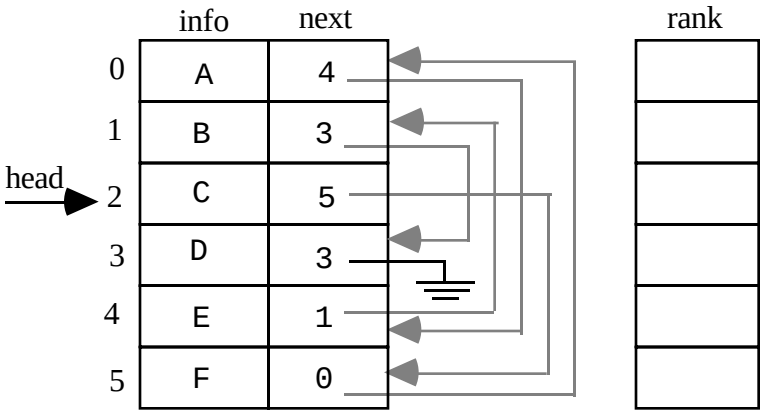


Fig. 5.10. PRAM data structures representing a linked list and the ranking results.

List-ranking appears to be hopelessly sequential

However, we can in fact use a recursive doubling scheme to determine the rank of each element in optimal time

There exist other problems that appear to be unparallizable

This is why intuition can be misleading when it comes to determining which computations are or are not efficiently parallelizable (i.e., whether a computation is or is not in NC)

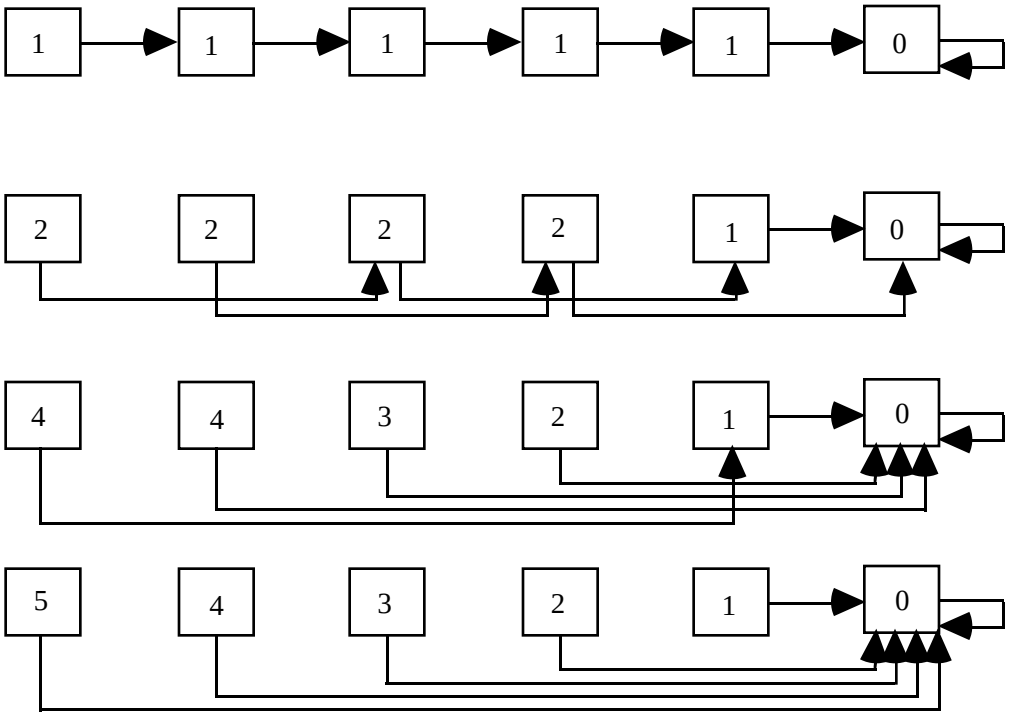


Fig. 5.11. Element ranks initially and after each of the three iterations.

PRAM list ranking algorithm (via pointer jumping)
Processor j, $0 \le j < p$, do {initialize the partial ranks}
    if next[j] = j
    then rank[j] := 0
    else rank[j] := 1
    endif
while rank[next[head]] $\neq$ 0 Processor j, $0 \le j < p$, do
    rank[j] := rank[j] + rank[next[j]]
    next[j] := next[next[j]]
endwhile

Which PRAM submodel is implicit in the preceding algorithm?

5.6 Matrix Multiplication

For $m \times m$ matrices, $C = A \times B$ means: $c_{ij} = \sum_{k=0}^{m-1} a_{ik} \; b_{kj}$

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 + aik bkj
        endfor
        cij := t
    endfor
endfor
```

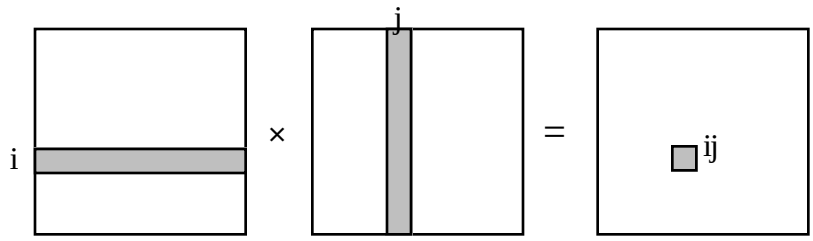
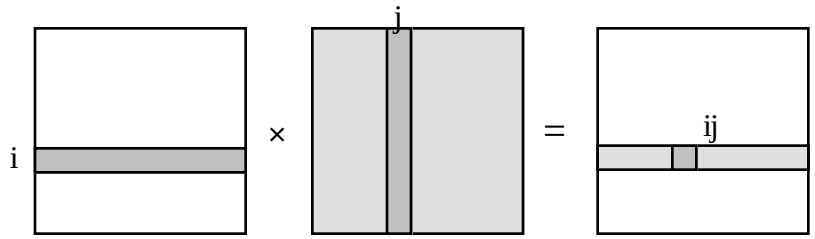


Fig. 5.12. PRAM matrix multiplication by using $p = m^2$ processors.

PRAM matrix multiplication algorithm using m^2 processors

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

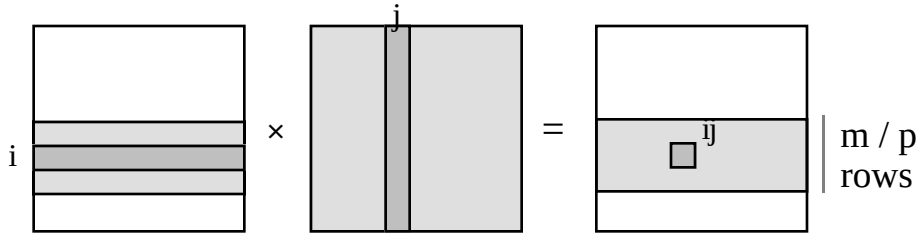



PRAM matrix multiplication algorithm using m processors

```
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
```

Both of the preceding algorithms are efficient and provide linear speedup

Using fewer than m processors: each processor computes m/p rows of C



The preceding solution is not efficient for NUMA parallel architectures

Each element of B is fetched m/p times

For each such data access, only two arithmetic operations are performed

Block matrix multiplication

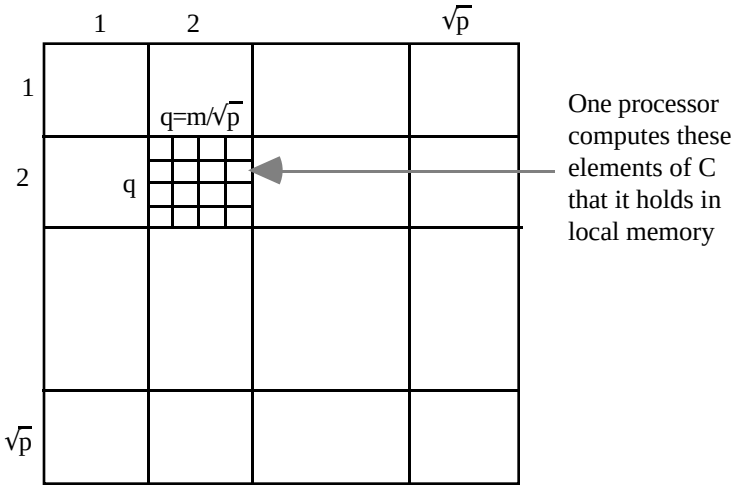
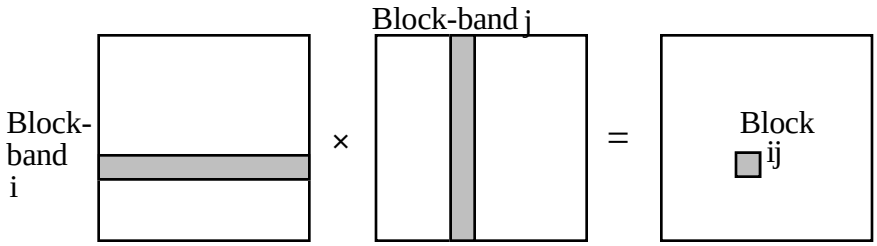


Fig. 5.13. Partitioning the matrices for block matrix multiplication.



Each multiply-add computation on q × q blocks needs

$2q^2 = 2m^2/p$ memory accesses to read the blocks

$2q^3$ arithmetic operations

So, q arithmetic operations are done per memory access

We assume that processor (i, j) has local memory to hold

Block (i, j) of the result matrix C (q^2 elements)

One block-row of B; say Row kq + c of Block (k, j) of B

(Elements of A can be brought in one at a time)

For example, 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

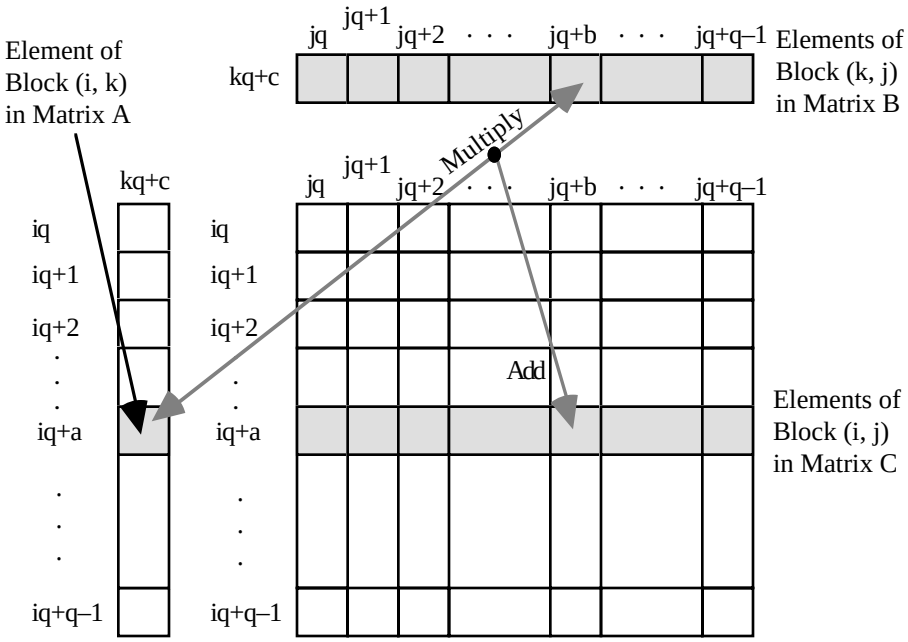


Fig. 5.14. How Processor (i, j) operates on an element of A and one block-row of B to update one block-row of C.

On the Cm* NUMA-type shared-memory multiprocessor, this block algorithm exhibited good, but sublinear, speedup

p = 16, speed-up = 5 in multiplying 24 × 24 matrices;

improved to 9 (11) for larger 36 × 36 (48 × 48) matrices

The improved locality of block matrix multiplication can also improve the running time on a uniprocessor, or distributed shared-memory multiprocessor with caches

Reason: higher cache hit rates.

