CSE 613: Parallel Programming

Lecture 10
(Cache Performance of Divide-and-Conquer Algorithms)

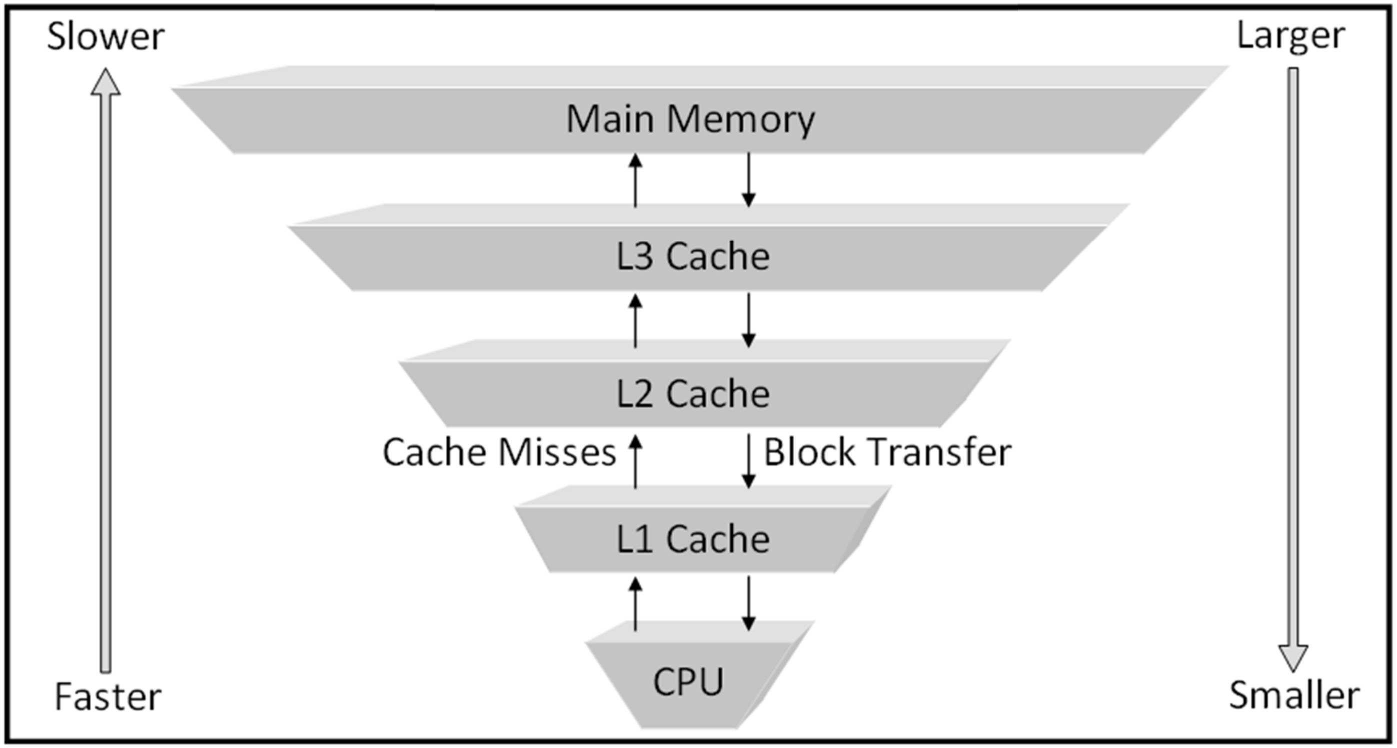
Rezaul A. Chowdhury

Department of Computer Science
SUNY Stony Brook

Spring 2012

Modern Single Core Machines

Memory Hierarchy

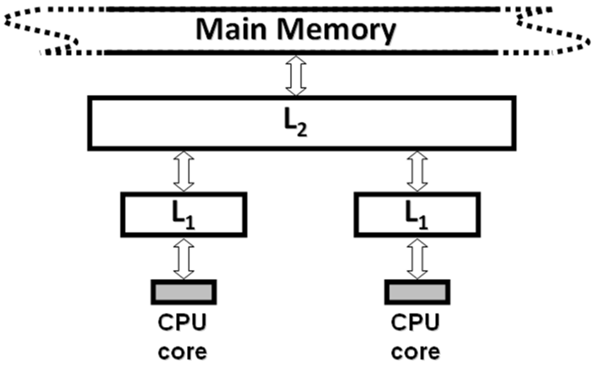


— Cost of memory access depends on whether it’s a (cache) hit or miss

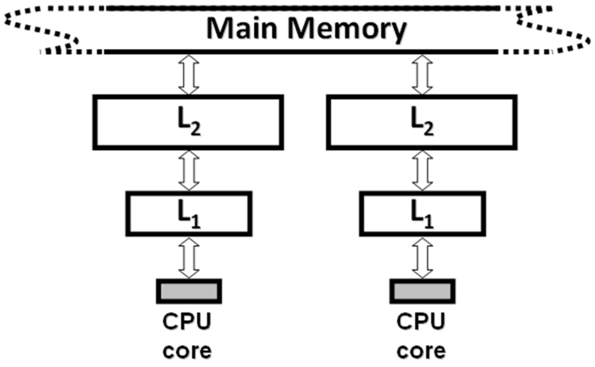
— Data in cache may get evicted to make space for new data items

— Good performance requires high locality in memory accesses

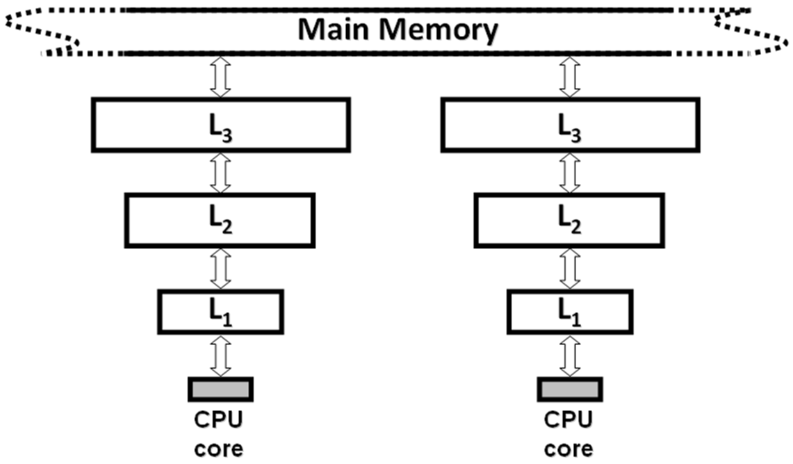
Multicores



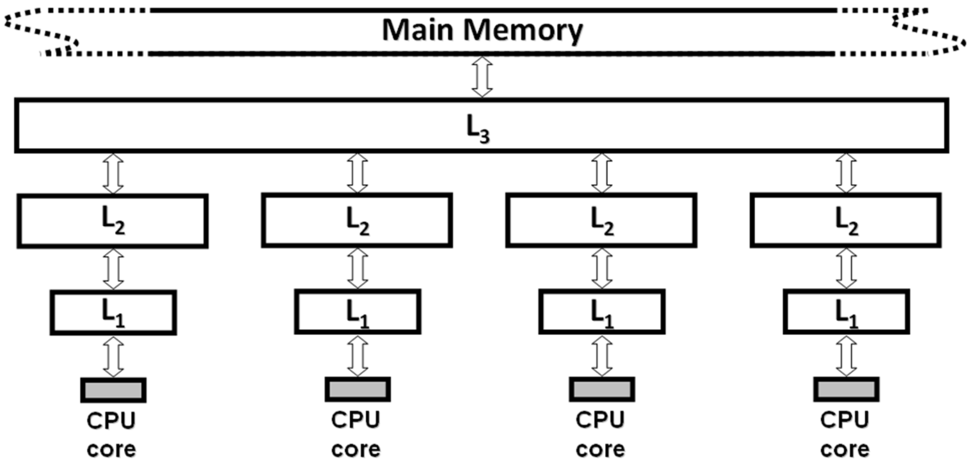
Intel Core Duo, Intel Xeon



AMD Opteron, AMD Athlon



Intel Itanium 2



Intel Nehalem, AMD Barcelona

Assumptions

Two Level Memory Hierarchy

A single level of cache (caches) of size *C* (each) connected to a main memory of unbounded size and block size B

LRU (Least Recently Used) Cache Replacement Policy

When a new block must be brought into the cache, but the cache is full, then the cache block that was accessed least recently is evicted to make space for the new block

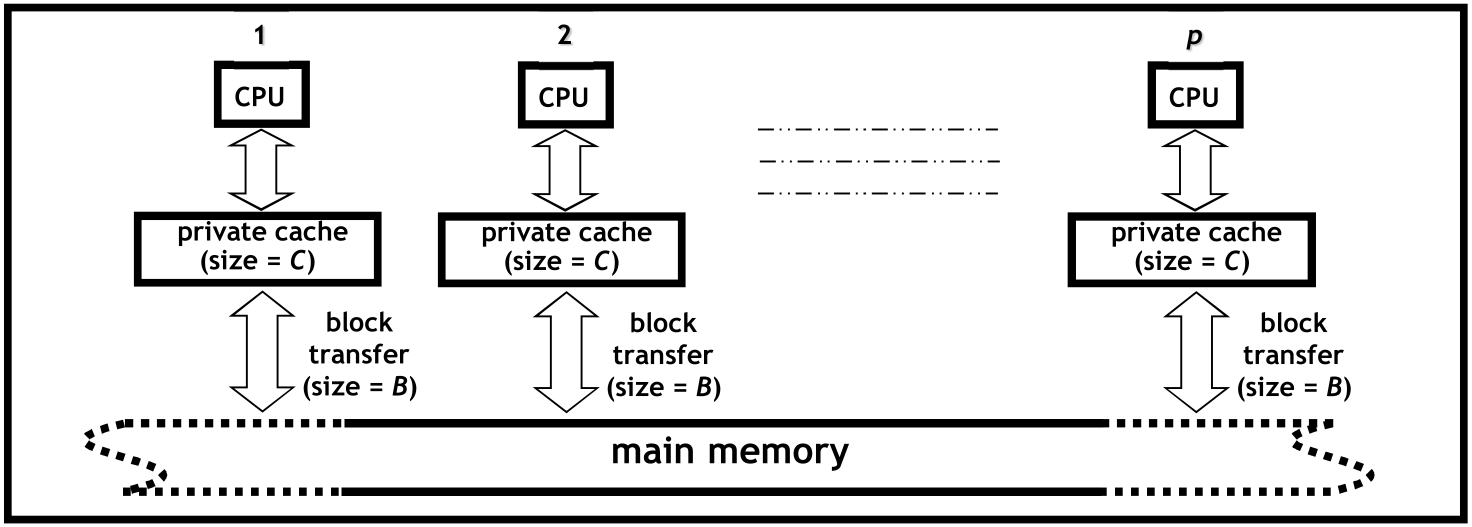
Automatic Cache Replacement

Done automatically by the OS or the hardware

Fully Associative Caches

A block brought into the cache from memory can reside anywhere in the cache

Parallel Caching Model: Distributed Caches



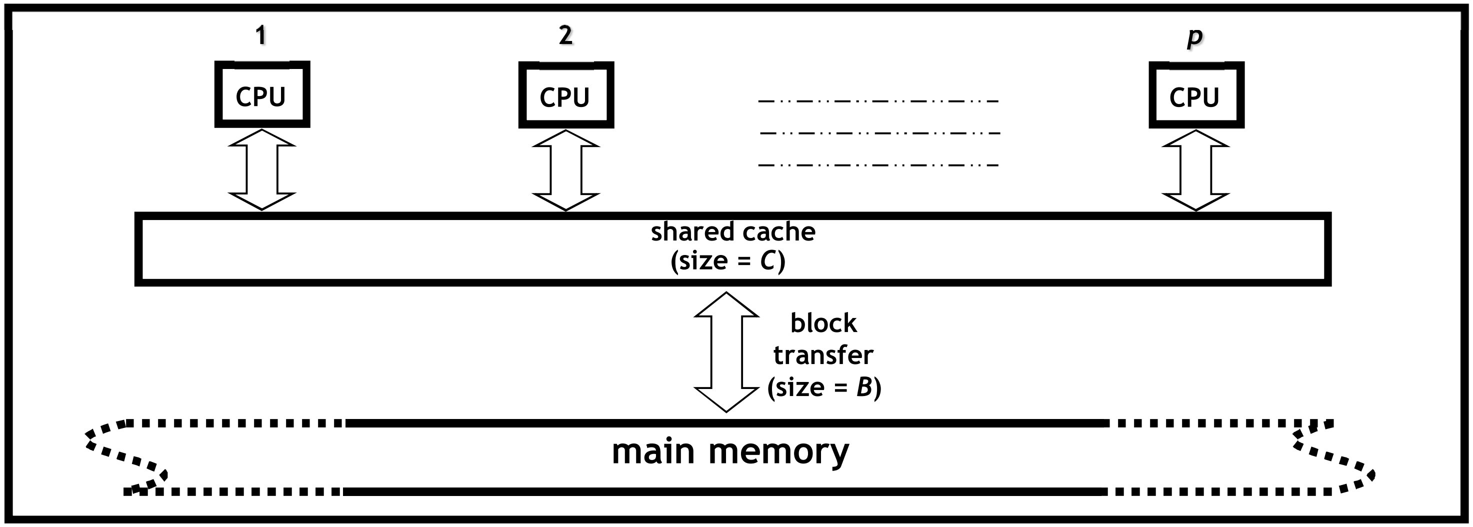
Configuration:

- *p* processing elements
- a *private cache* of size *C* for each processing element
- an arbitrarily large *global shared memory*
- block transfer size *B* (between caches and memory)

Cache Performance Measure:

- number of *block transfers* across all caches

Parallel Caching Model: Shared Cache



Configuration:

- p processing elements
- a *shared cache* of size C
- an arbitrarily large *global shared memory*
- $C \geq p \cdot B$, where C is block transfer size

Cache Performance Measure:

- number of *block transfers* between the cache and the memory

Locality of Reference

Spatial Locality

If a particular memory location is accessed at a particular time, then it is likely that nearby memory locations will also be accessed in the near future.

Take advantage of the block size *B* to load all memory locations in the same block into the cache when a particular memory location in that block is accessed.

Temporal Locality

If a particular memory location is accessed at a particular time, then it is likely that the same memory location will be accessed again in the near future.

Take advantage of the cache size *C* to retain memory locations already loaded into the cache for future references.

Iterative Matrix Multiplication

$$\boxed{z_{ij} = \sum_{k=1}^{n} x_{ik} y_{kj}}$$

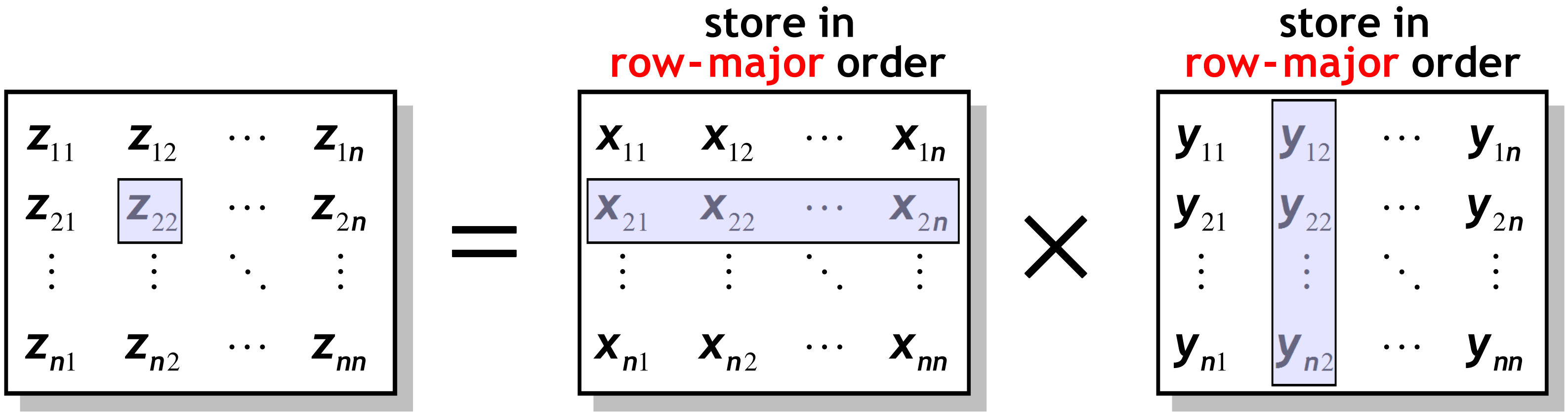
$$\begin{bmatrix} z_{11} & z_{12} & \cdots & z_{1n} \\ z_{21} & z_{22} & \cdots & z_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ z_{n1} & z_{n2} & \cdots & z_{nn} \end{bmatrix} = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1n} \\ x_{21} & x_{22} & \cdots & x_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \cdots & x_{nn} \end{bmatrix} \times \begin{bmatrix} y_{11} & y_{12} & \cdots & y_{1n} \\ y_{21} & y_{22} & \cdots & y_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ y_{n1} & y_{n2} & \cdots & y_{nn} \end{bmatrix}$$

```
Iter-MM ( Z, X, Y )          { X, Y, Z are n × n matrices,
                               where n is a positive integer }
for i ← 1 to n do
    for j ← 1 to n do
        Z[ i ][ j ] ← 0
        for k ← 1 to n do
            Z[ i ][ j ] ← Z[ i ][ j ] + X[ i ][ k ] · Y[ k ][ j ]
```

Iterative Matrix Multiplication

```
Iter-MM ( Z, X, Y )          { X, Y, Z are n × n matrices,
                               where n is a positive integer }

1.  for i ← 1 to n do
2.      for j ← 1 to n do
3.          Z[ i ][ j ] ← 0
4.          for k ← 1 to n do
5.              Z[ i ][ j ] ← Z[ i ][ j ] + X[ i ][ k ] · Y[ k ][ j ]
```



store in **row-major** order

store in **row-major** order

$$\begin{pmatrix} z_{11} & z_{12} & \cdots & z_{1n} \\ z_{21} & \boxed{z_{22}} & \cdots & z_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ z_{n1} & z_{n2} & \cdots & z_{nn} \end{pmatrix} = \begin{pmatrix} x_{11} & x_{12} & \cdots & x_{1n} \\ x_{21} & x_{22} & \cdots & x_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \cdots & x_{nn} \end{pmatrix} \times \begin{pmatrix} y_{11} & y_{12} & \cdots & y_{1n} \\ y_{21} & y_{22} & \cdots & y_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ y_{n1} & y_{n2} & \cdots & y_{nn} \end{pmatrix}$$

Each iteration of the *for* loop in line 3 incurs $\mathrm{O}(n)$ cache misses.

Cache-complexity of *Iter-MM*, $\mathrm{Q}(n) = \mathrm{O}(n^3)$.

Iterative Matrix Multiplication

```
Iter-MM ( Z, X, Y )            { X, Y, Z are n × n matrices,
                                 where n is a positive integer }
1. for i ← 1 to n do
2.     for j ← 1 to n do
3.         Z[ i ][ j ] ← 0
4.         for k ← 1 to n do
5.             Z[ i ][ j ] ← Z[ i ][ j ] + X[ i ][ k ] · Y[ k ][ j ]
```

store in **row-major** order

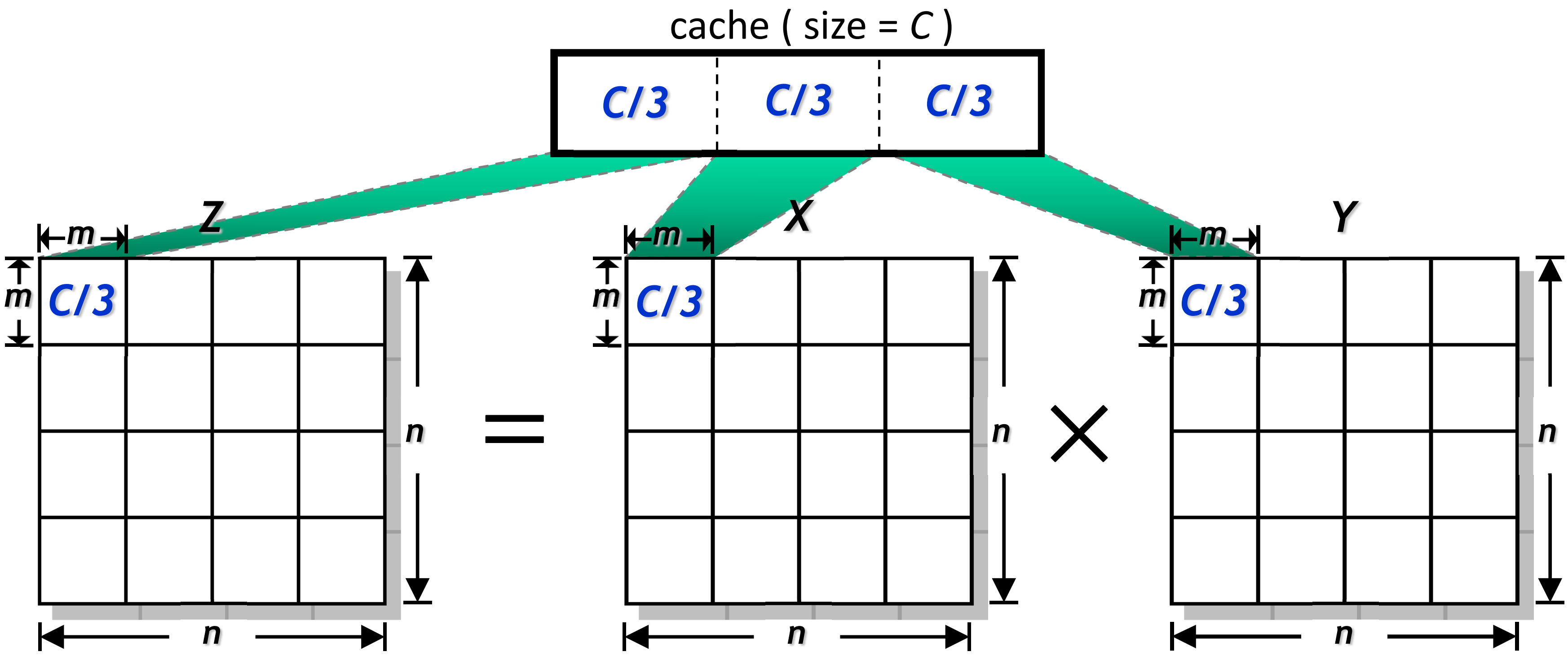
store in **column-major** order

$$\begin{pmatrix} z_{11} & z_{12} & \cdots & z_{1n} \\ z_{21} & \boxed{z_{22}} & \cdots & z_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ z_{n1} & z_{n2} & \cdots & z_{nn} \end{pmatrix} = \begin{pmatrix} x_{11} & x_{12} & \cdots & x_{1n} \\ x_{21} & x_{22} & \cdots & x_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \cdots & x_{nn} \end{pmatrix} \times \begin{pmatrix} y_{11} & y_{12} & \cdots & y_{1n} \\ y_{21} & y_{22} & \cdots & y_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ y_{n1} & y_{n2} & \cdots & y_{nn} \end{pmatrix}$$

Each iteration of the *for* loop in line 3 incurs $\mathrm{O}\left(1 + \frac{n}{B}\right)$ cache misses.

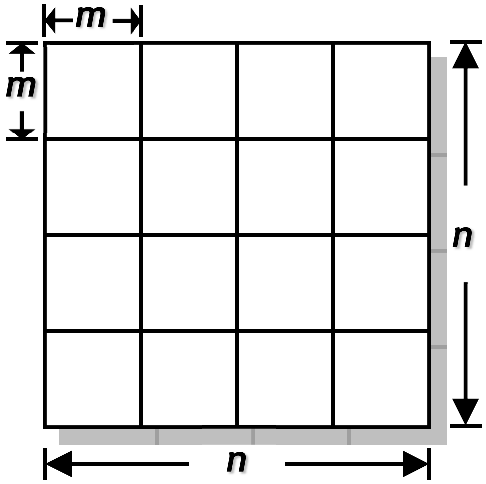
Cache-complexity of *Iter-MM*, $\mathrm{Q}(n) = \mathrm{O}\left(n^2\left(1 + \frac{n}{B}\right)\right) = \mathrm{O}\left(\frac{n^3}{B} + n^2\right)$.

Block Matrix Multiplication



```
Block-MM( X, Y, Z )
    for $i \leftarrow 1$ to $n / m$ do
        for $j \leftarrow 1$ to $n / m$ do
            for $k \leftarrow 1$ to $n / m$ do
                Iter-MM( $X_{ik}$, $Y_{kj}$, $Z_{ij}$ )
```

Block Matrix Multiplication



```
Block-MM( X, Y, Z )
1. for i ← 1 to n / m do
2.     for j ← 1 to n / m do
3.         for k ← 1 to n / m do
4.             Iter-MM( $X_{ik}$, $Y_{kj}$, $Z_{ij}$ )
```

Choose $m = \sqrt{C/3}$, so that X_{ik}, Y_{kj} and Z_{ij} just fit into the cache.

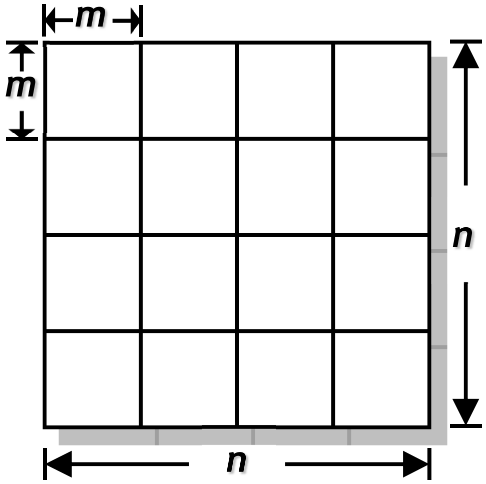
Then line 4 incurs $\Theta\left(m\left(1+\frac{m}{B}\right)\right)$ cache misses.

Cache-complexity of *Block-MM* [assuming a *tall cache*, i.e., $C =$ ⍰(B^2)]

$$= \Theta\left(\left(\frac{n}{m}\right)^3\left(m+\frac{m^2}{B}\right)\right) = \Theta\left(\frac{n^3}{m^2}+\frac{n^3}{Bm}\right) = \Theta\left(\frac{n^3}{C}+\frac{n^3}{B\sqrt{C}}\right) = \Theta\left(\frac{n^3}{B\sqrt{C}}\right)$$

(Optimal: Hong & Kung, STOC’81)

Block Matrix Multiplication



```
Block-MM( X, Y, Z )
for $i \leftarrow 1$ to $n / m$ do
    for $j \leftarrow 1$ to $n / m$ do
        for $k \leftarrow 1$ to $n / m$ do
            Iter-MM( $X_{ik}$, $Y_{kj}$, $Z_{ij}$ )
```

Choose $m = \sqrt{C/2}$ so that X, Y and Z just fit into the cache.

Optimal for any algorithm that performs the operations given by the following definition of matrix multiplication:

$$z_{ij} = \sum_{k=1}^{n} x_{ik} y_{kj}$$

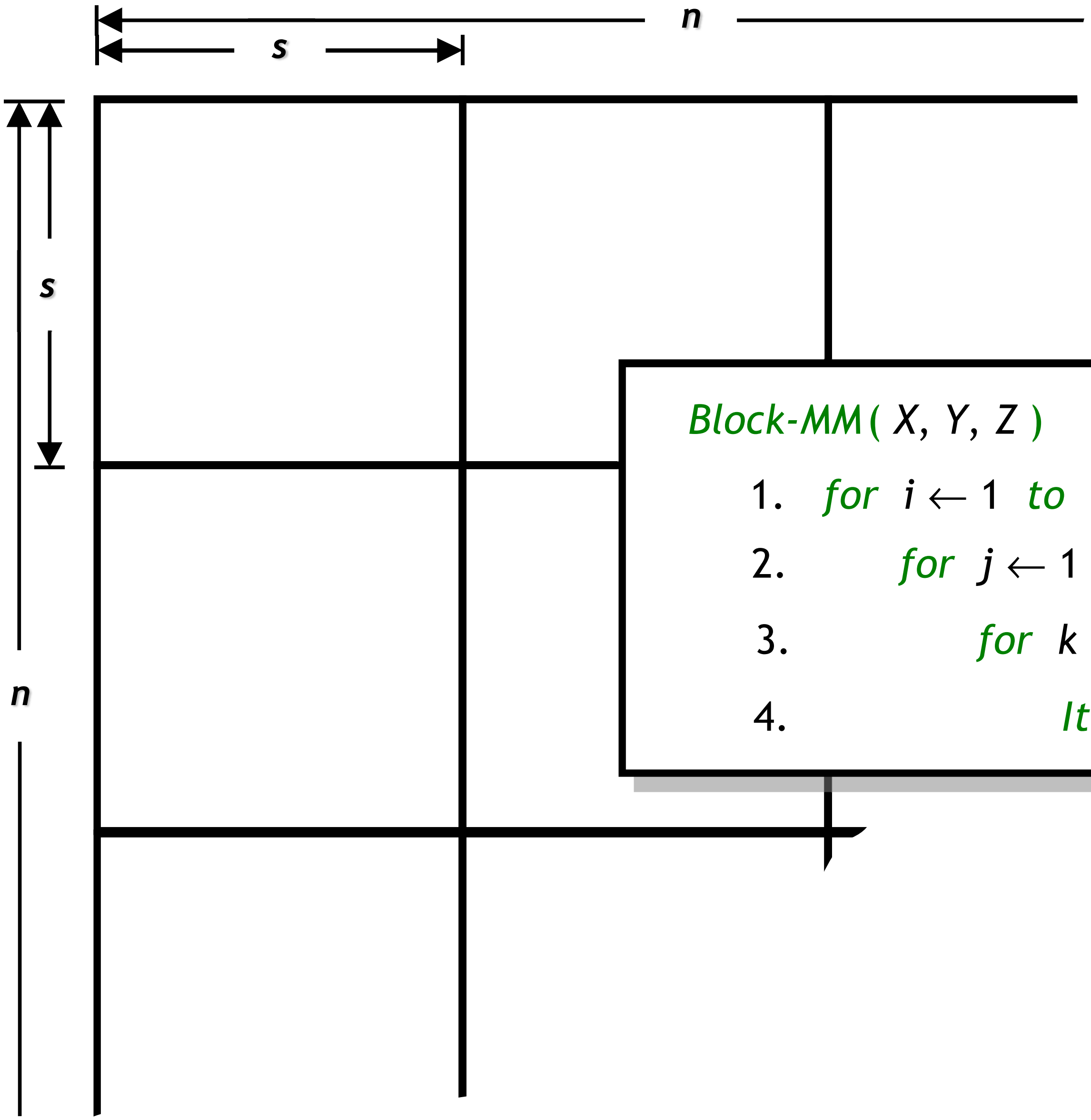
The ses.

Cac *all cache*, i.e., $C = \square(B^2)$]

$$= \Theta\left(\left(\frac{n}{m}\right)^3\left(m + \frac{m^2}{B}\right)\right) = \Theta\left(\frac{n}{m^2} + \quad \frac{}{Bm}\right) \quad \Theta\left(\frac{n^3}{C} + \frac{n^3}{B\sqrt{C}}\right) = \Theta\left(\frac{n^3}{B\sqrt{C}}\right)$$

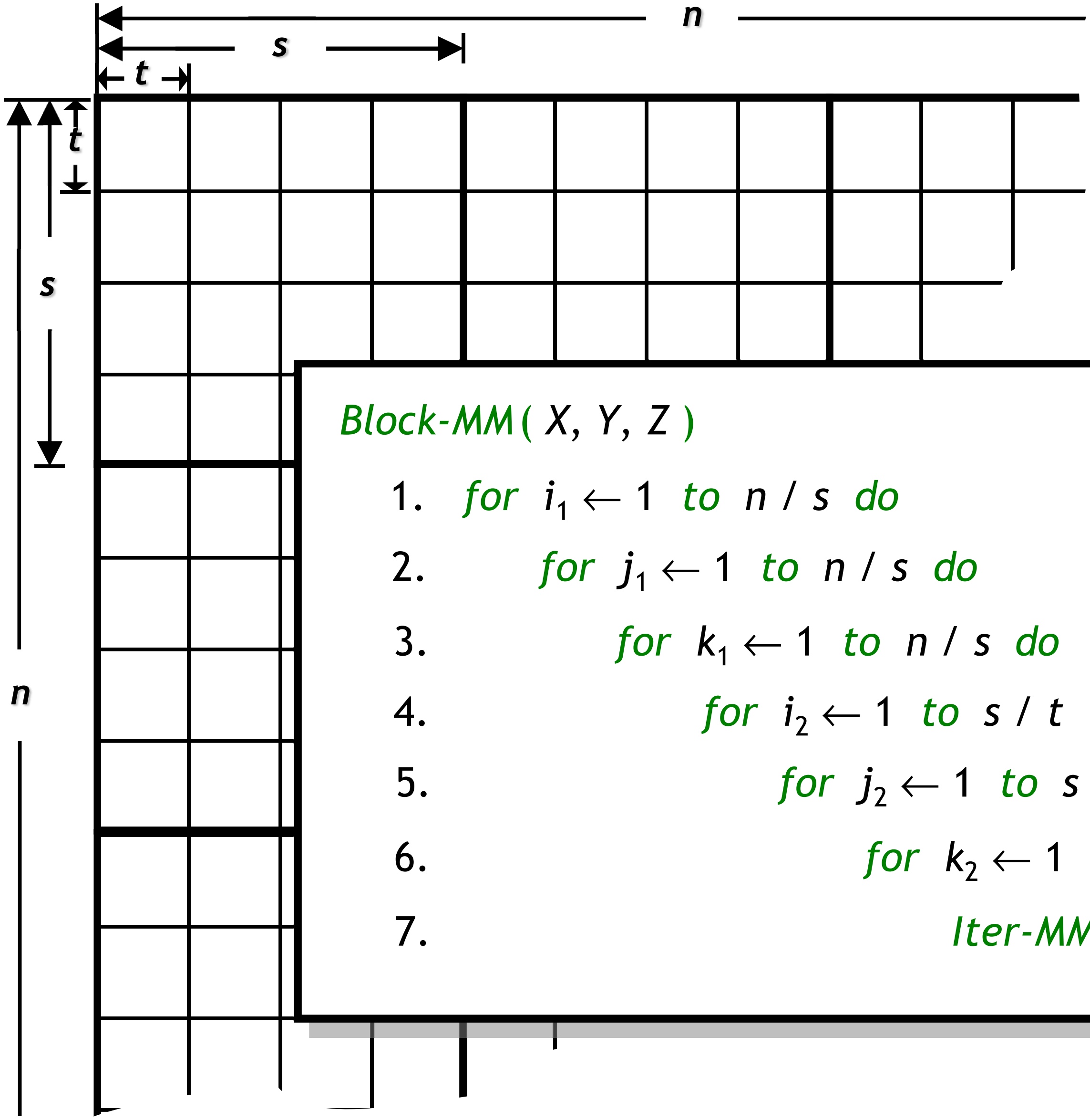
(Optimal: Hong & Kung, STOC’81)

Multiple Levels of Cache



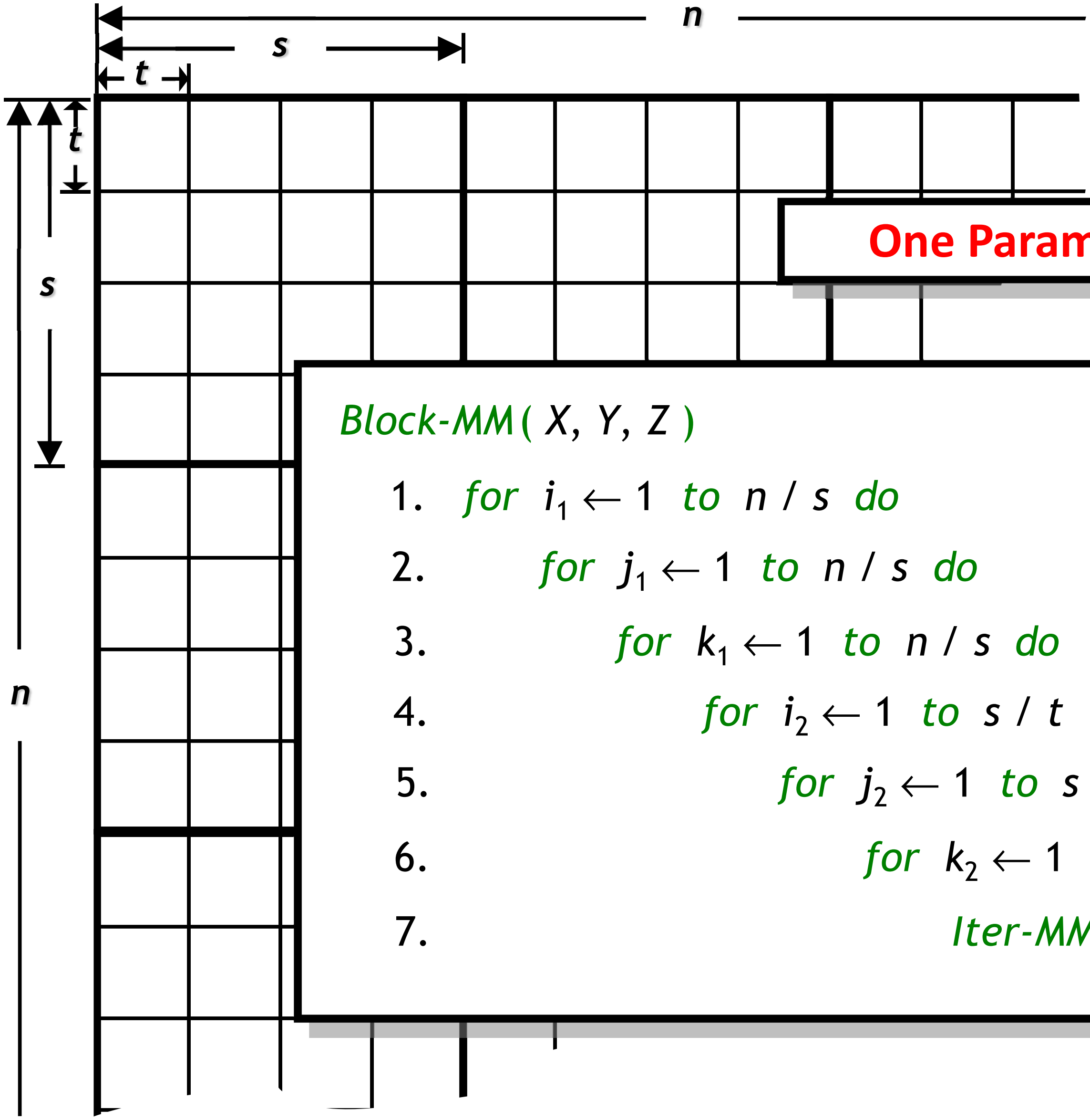
```
Block-MM( X, Y, Z )
    for $i \leftarrow 1$ to $n / s$ do
        for $j \leftarrow 1$ to $n / s$ do
            for $k \leftarrow 1$ to $n / s$ do
                Iter-MM( $X_{ik}$, $Y_{kj}$, $Z_{ij}$ )
```

Multiple Levels of Cache



```
Block-MM( X, Y, Z )
    for $i_1 \leftarrow 1$ to $n / s$ do
        for $j_1 \leftarrow 1$ to $n / s$ do
            for $k_1 \leftarrow 1$ to $n / s$ do
                for $i_2 \leftarrow 1$ to $s / t$ do
                    for $j_2 \leftarrow 1$ to $s / t$ do
                        for $k_2 \leftarrow 1$ to $s / t$ do
                            Iter-MM( $(X_{i_1k_1})_{i_2k_2}$, $(Y_{k_1j_1})_{k_2j_2}$, $(Z_{i_1j_1})_{i_2j_2}$ )
```

Multiple Levels of Cache



One Parameter Per Caching Level!

```
Block-MM( X, Y, Z )
    for $i_1 \leftarrow 1$ to $n / s$ do
        for $j_1 \leftarrow 1$ to $n / s$ do
            for $k_1 \leftarrow 1$ to $n / s$ do
                for $i_2 \leftarrow 1$ to $s / t$ do
                    for $j_2 \leftarrow 1$ to $s / t$ do
                        for $k_2 \leftarrow 1$ to $s / t$ do
                            Iter-MM( $(X_{i_1k_1})_{i_2k_2}$, $(Y_{k_1j_1})_{k_2j_2}$, $(Z_{i_1j_1})_{i_2j_2}$ )
```

Parallel Recursive MM

```
Par-Rec-MM ( Z, X, Y )    { X, Y, Z are $n \times n$ matrices,
                            where $n = 2^k$ for integer $k \geq 0$ }
if $n = 1$ then
    $Z \leftarrow Z + X \cdot Y$
else
    spawn Par-Rec-MM ( $Z_{11}$, $X_{11}$, $Y_{11}$ )
    spawn Par-Rec-MM ( $Z_{12}$, $X_{11}$, $Y_{12}$ )
    spawn Par-Rec-MM ( $Z_{21}$, $X_{21}$, $Y_{11}$ )
          Par-Rec-MM ( $Z_{21}$, $X_{21}$, $Y_{11}$ )
    sync
    spawn Par-Rec-MM ( $Z_{11}$, $X_{12}$, $Y_{21}$ )
    spawn Par-Rec-MM ( $Z_{12}$, $X_{12}$, $Y_{22}$ )
    spawn Par-Rec-MM ( $Z_{21}$, $X_{22}$, $Y_{21}$ )
          Par-Rec-MM ( $Z_{22}$, $X_{22}$, $Y_{22}$ )
    sync
endif
```

Work:

$$T_1(n) = \begin{cases} \Theta(1), & if\ n = 1, \\ 8T_1\left(\frac{n}{2}\right) + \Theta(1), & otherwise. \end{cases}$$

$$= \Theta(n^3) \qquad \text{[MT Case 1]}$$

Span:

$$T_\infty(n) = \begin{cases} \Theta(1), & if\ n = 1, \\ 2T_\infty\left(\frac{n}{2}\right) + \Theta(1), & otherwise. \end{cases}$$

$$= \Theta(n) \qquad \text{[MT Case 1]}$$

Parallel Running Time:

$$T_p(n) = \mathrm{O}\left(\frac{T_1(n)}{p} + T_\infty(n)\right) = \mathrm{O}\left(\frac{n^3}{p} + n\right)$$

Parallelism: $\frac{T_1(n)}{T_\infty(n)} = \Theta(n^2)$

Additional Space:

$$s_\infty(n) = \Theta(1)$$

Parallel Recursive MM on a Single Core

```
Par-Rec-MM ( $Z$, $X$, $Y$ )    { $X$, $Y$, $Z$ are $n \times n$ matrices,
                                  where $n = 2^k$ for integer $k \geq 0$ }
if $n = 1$ then
    $Z \leftarrow Z + X \cdot Y$
else
    spawn Par-Rec-MM ( $Z_{11}$, $X_{11}$, $Y_{11}$ )
    spawn Par-Rec-MM ( $Z_{12}$, $X_{11}$, $Y_{12}$ )
    spawn Par-Rec-MM ( $Z_{21}$, $X_{21}$, $Y_{11}$ )
          Par-Rec-MM ( $Z_{21}$, $X_{21}$, $Y_{11}$ )
    sync
    spawn Par-Rec-MM ( $Z_{11}$, $X_{12}$, $Y_{21}$ )
    spawn Par-Rec-MM ( $Z_{12}$, $X_{12}$, $Y_{22}$ )
    spawn Par-Rec-MM ( $Z_{21}$, $X_{22}$, $Y_{21}$ )
          Par-Rec-MM ( $Z_{22}$, $X_{22}$, $Y_{22}$ )
    sync
endif
```

Cache Complexity:

$$Q_1(n) = \begin{cases} O\left(n + \frac{n^2}{B}\right), & if\ n^2 \leq \alpha C, \\ 8Q_1\left(\frac{n}{2}\right), & otherwise. \end{cases}$$

$$= \Theta\left(\frac{n^3}{B\sqrt{C}} + \frac{n^2}{B} + 1\right),$$

when $C = \Omega(B^2)$

Parallel Recursive MM on Distributed Caches

```
Par-Rec-MM ( Z, X, Y )    { X, Y, Z are n × n matrices,
                            where n = 2^k for integer k ≥ 0 }
if n = 1 then
    $Z \leftarrow Z + X \cdot Y$
else
    spawn Par-Rec-MM ( $Z_{11}$, $X_{11}$, $Y_{11}$ )
    spawn Par-Rec-MM ( $Z_{12}$, $X_{11}$, $Y_{12}$ )
    spawn Par-Rec-MM ( $Z_{21}$, $X_{21}$, $Y_{11}$ )
          Par-Rec-MM ( $Z_{21}$, $X_{21}$, $Y_{11}$ )
    sync
    spawn Par-Rec-MM ( $Z_{11}$, $X_{12}$, $Y_{21}$ )
    spawn Par-Rec-MM ( $Z_{12}$, $X_{12}$, $Y_{22}$ )
    spawn Par-Rec-MM ( $Z_{21}$, $X_{22}$, $Y_{21}$ )
          Par-Rec-MM ( $Z_{22}$, $X_{22}$, $Y_{22}$ )
    sync
endif
```

$$\boxed{p = 4^q < \frac{n^2}{\alpha C}}$$

Assumption: if $p > 1$, then p is evenly distributed among the simultaneously spawned functions.

Cache Complexity:

$$Q_p(n) = \begin{cases} Q_1(n), & if\ p = 1, \\ 8Q_{\frac{p}{4}}\left(\frac{n}{2}\right), & otherwise. \end{cases}$$

$$= \Theta\left(\frac{n^3}{B\sqrt{C}} + \sqrt{p} \cdot \frac{n^2}{B} + p\sqrt{p}\right),$$

when $C = \Omega(B^2)$

The Longest Common Subsequence (LCS) Problem

A *subsequence* of a sequence *X* is obtained by deleting zero or more symbols from *X*.

Example:

X = *abcba*

Z = *bca* ← obtained by deleting the 1st '*a*' and the 2nd '*b*' from *X*

A *Longest Common Subsequence* (*LCS*) of two sequence *X* and *Y* is a sequence *Z* that is a subsequence of both *X* and *Y*, and is the longest among all such subsequences.

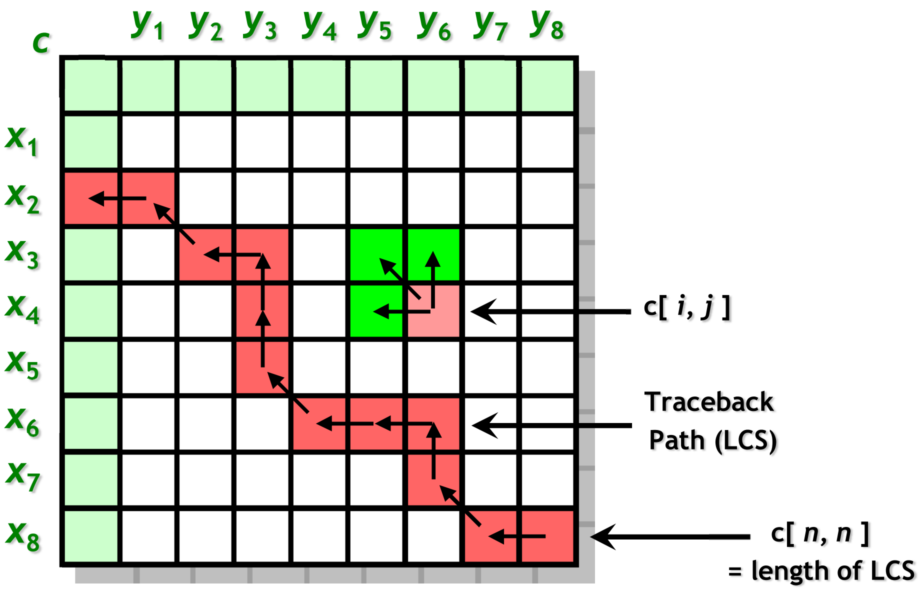
Given *X* and *Y*, the *LCS problem* asks for such a *Z*.

The Longest Common Subsequence (LCS) Problem

Given: $X = x_1 x_2 \ldots x_n$ and $Y = y_1 y_2 \ldots y_n$

Fills up an array $c[0 \ldots n, 0 \ldots n]$ using the following recurrence.

$$c[i,j] = \begin{cases} 0 & \textit{if } i = 0 \vee j = 0, \\ c[i-1, j-1] + 1 & \textit{if } i, j > 0 \wedge x_i = y_j, \\ \max\{c[i, j-1], c[i-1, j]\} & \textit{otherwise}. \end{cases}$$


c
y_1 y_2 y_3 y_4 y_5 y_6 y_7 y_8
x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8
c[i, j]
Traceback Path (LCS)
c[n, n]
= length of LCS

Local Dependency:

value of each cell depends only on values of adjacent cells.

The Longest Common Subsequence (LCS) Problem

The classic (iterative) serial LCS DP runs in $\Theta(n^2)$ time,
uses $\Theta(n^2)$ space, and
incurs $\Theta\left(\frac{n^2}{B}\right)$ cache misses.

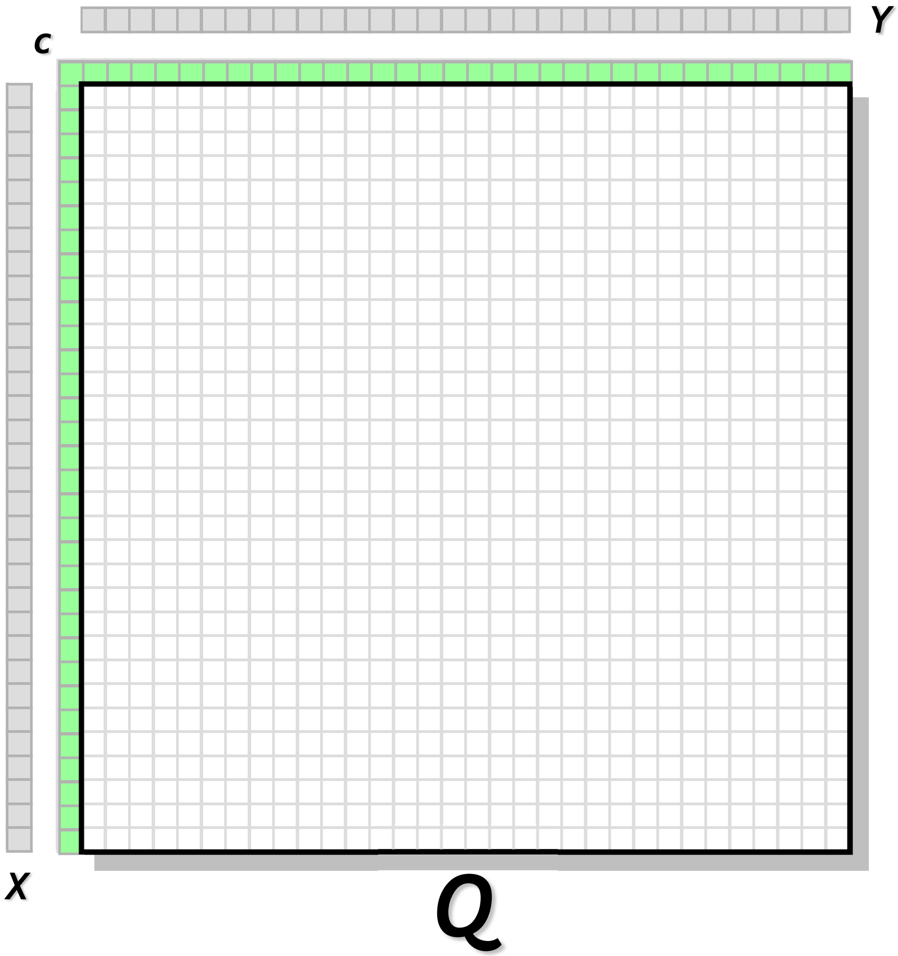
Any algorithm using $\Theta(s)$ space must incur $\Omega\left(\frac{s}{B}\right)$ cache misses.

Hence in order to reduce the cache complexity of the LCS algorithm from $\Theta\left(\frac{n^2}{B}\right)$ we must first reduce its space usage below $\Theta(n^2)$.

Sequential Cache-efficient LCS Algorithm

$Q \equiv c[1 \ldots n, 1 \ldots n]$

$n = 2^q$



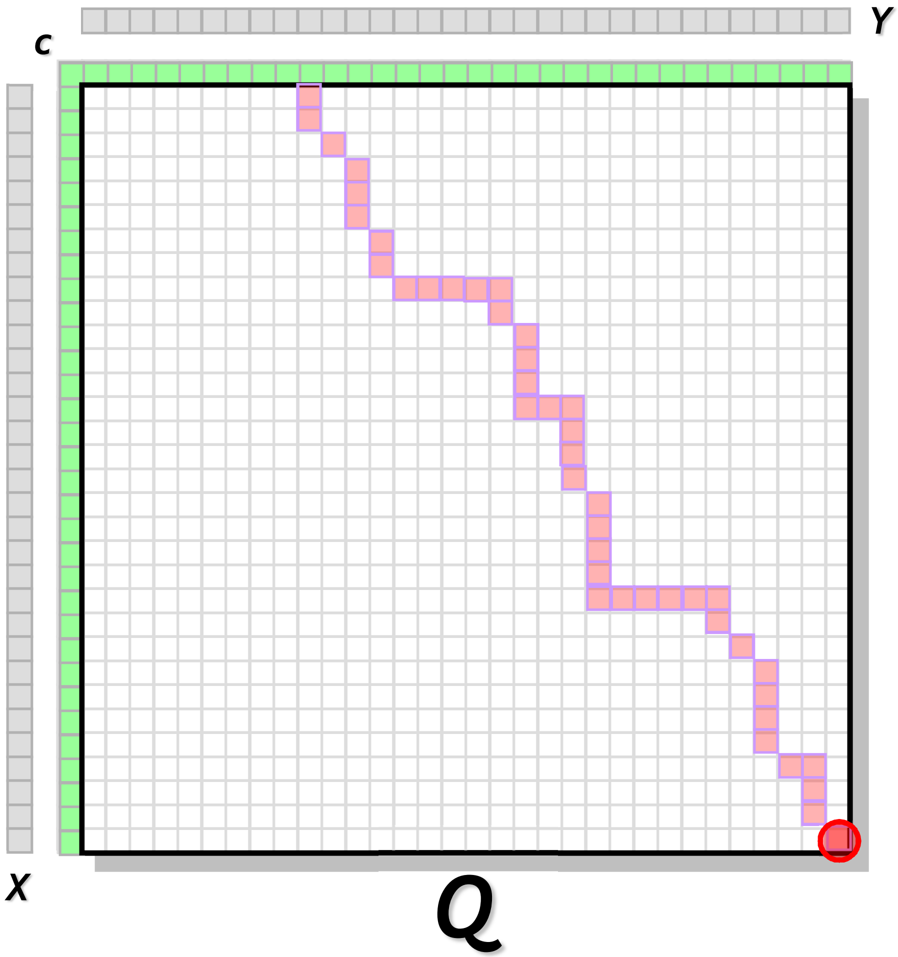
🟩 stored values

🟥 traceback path

Sequential Cache-efficient LCS Algorithm

$Q \equiv c[1 \ldots n, 1 \ldots n]$

$n = 2^q$



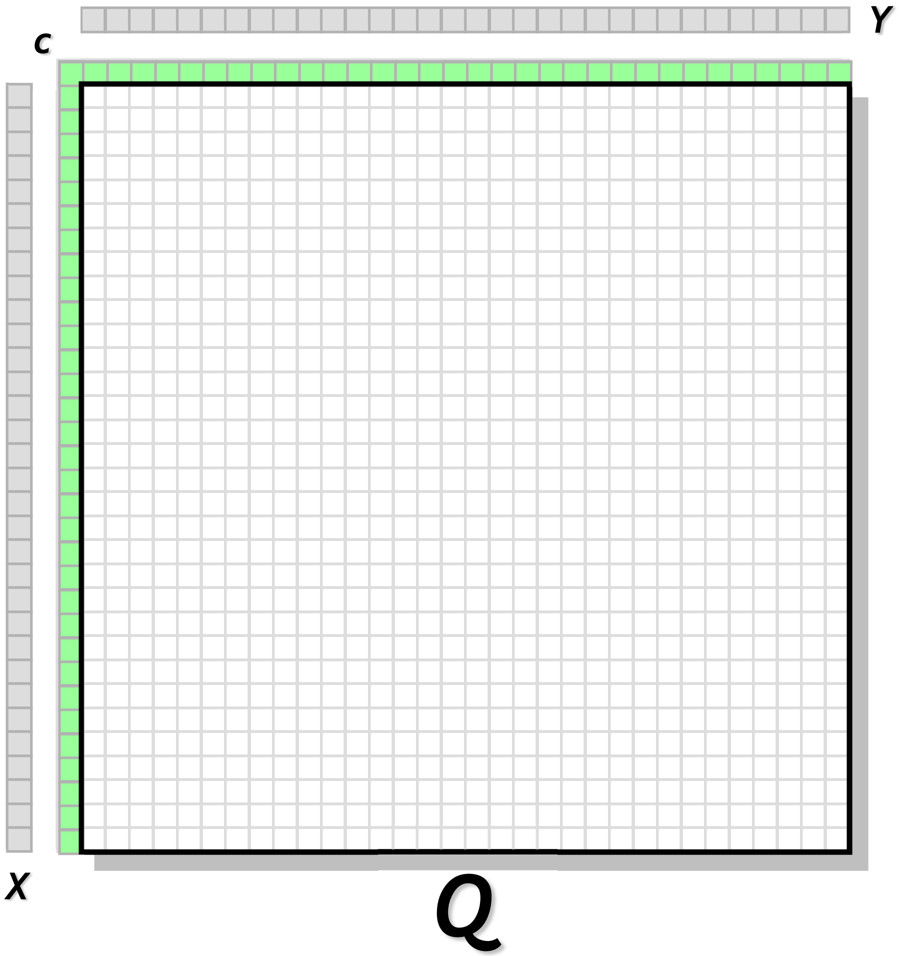
🟩 stored values

🟥 traceback path

Sequential Cache-efficient LCS Algorithm

$Q \equiv c[1 \ldots n, 1 \ldots n]$

$n = 2^q$



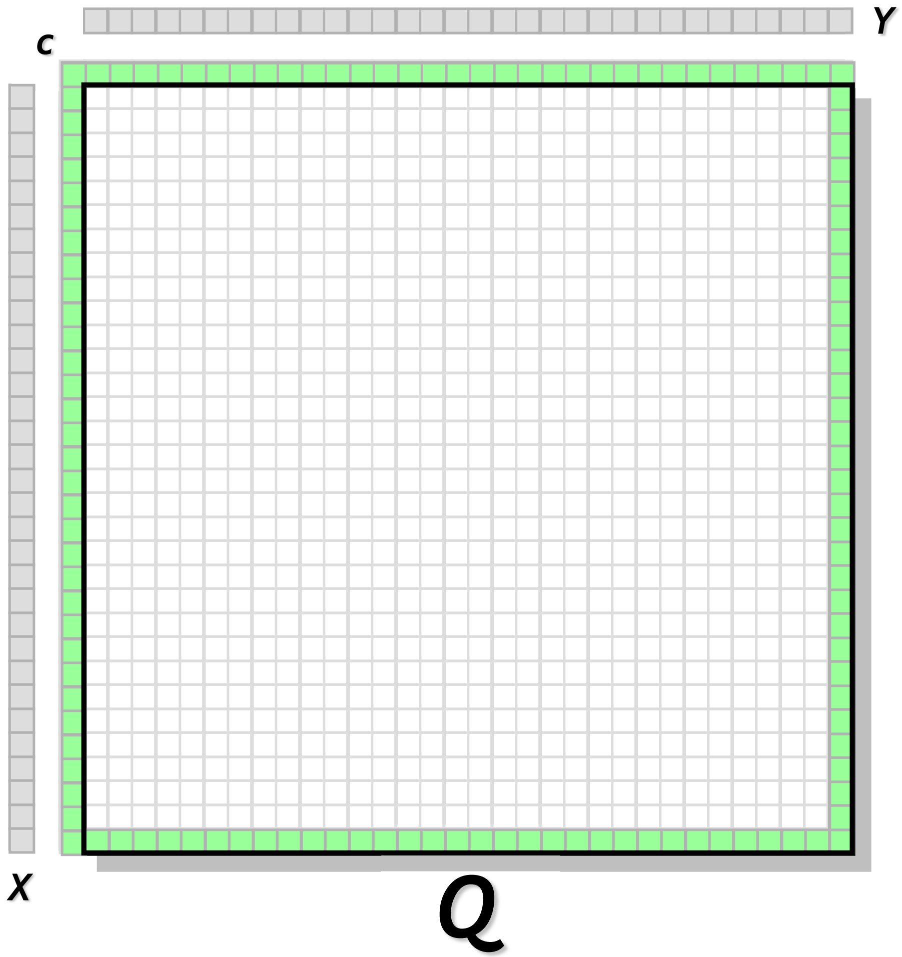
🟩 stored values

🟥 traceback path

Sequential Cache-efficient LCS Algorithm

$Q \equiv c[1 \ldots n,\ 1 \ldots n]$

$n = 2^q$



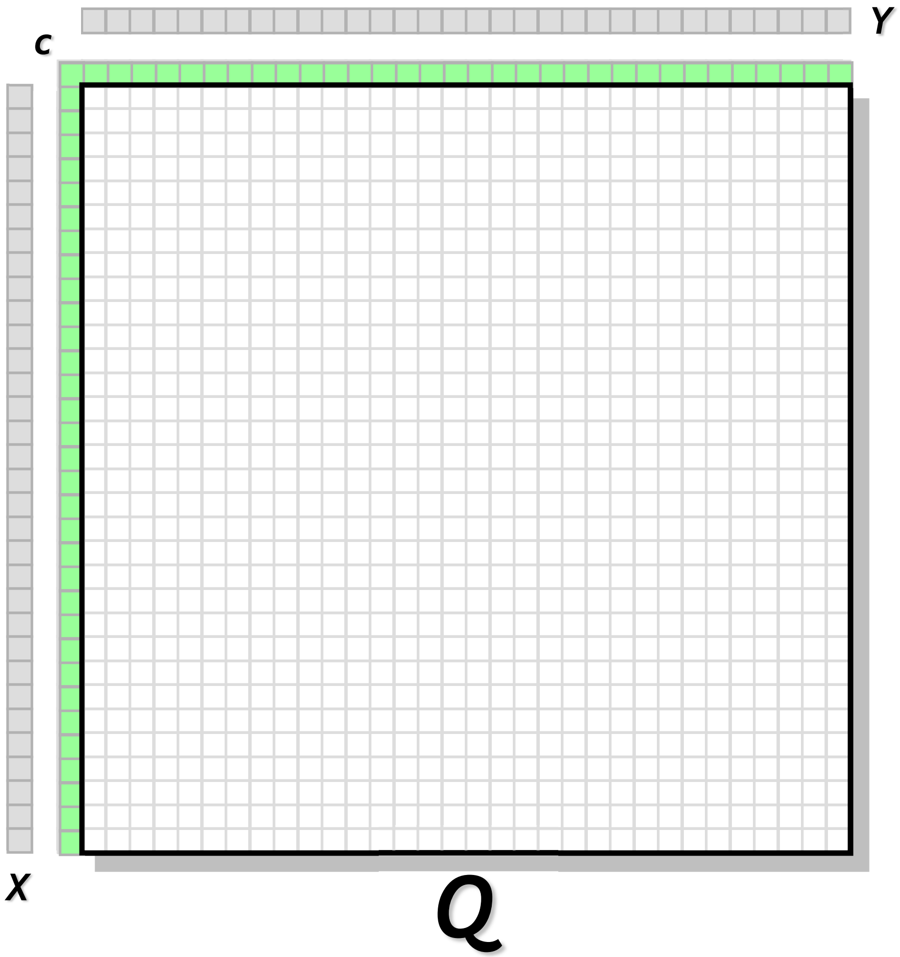
🟩 stored values

🟥 traceback path

Sequential Cache-efficient LCS Algorithm

$Q \equiv c[1 \ldots n, 1 \ldots n]$

$n = 2^q$



🟩 stored values

🟥 traceback path

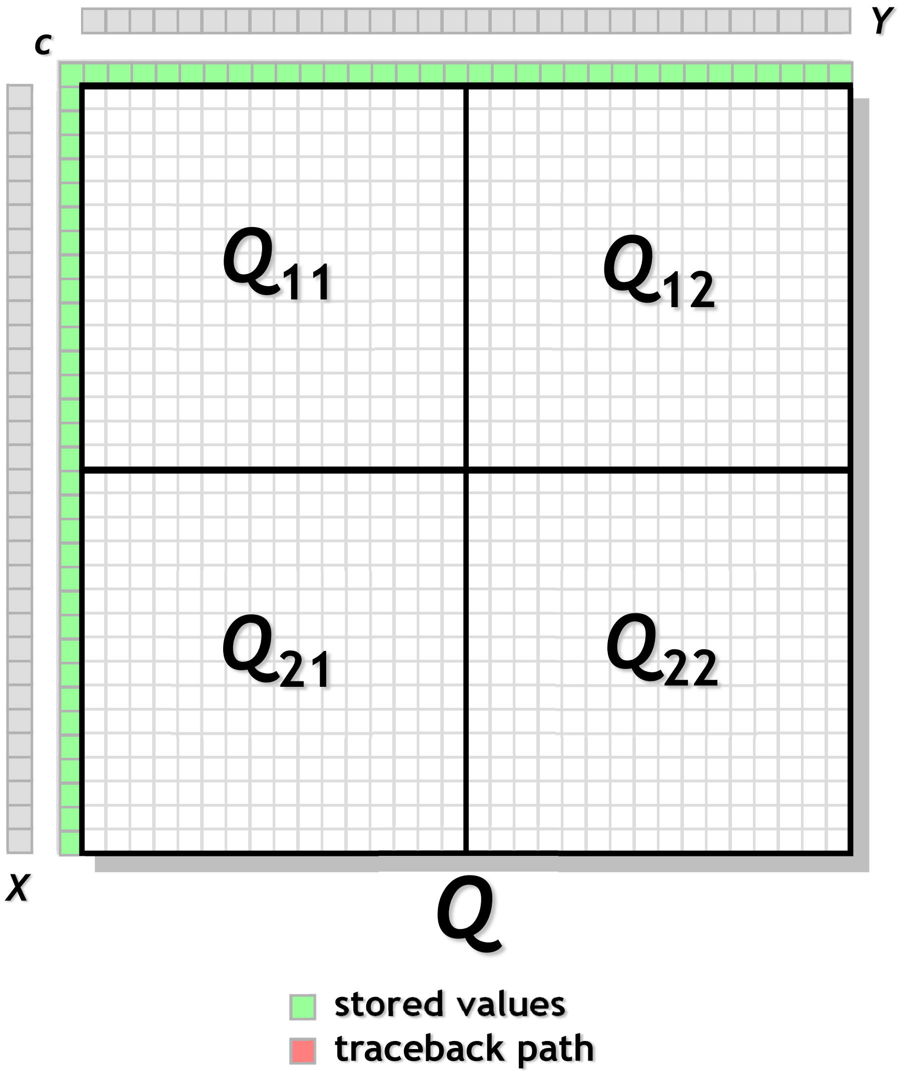
Sequential Cache-efficient LCS Algorithm

$Q \equiv c[1 \ldots n, 1 \ldots n]$

$n = 2^q$

1. **Decompose Q:**

   Split Q into four quadrants.



Sequential Cache-efficient LCS Algorithm

$Q \equiv c[1 \ldots n, 1 \ldots n]$

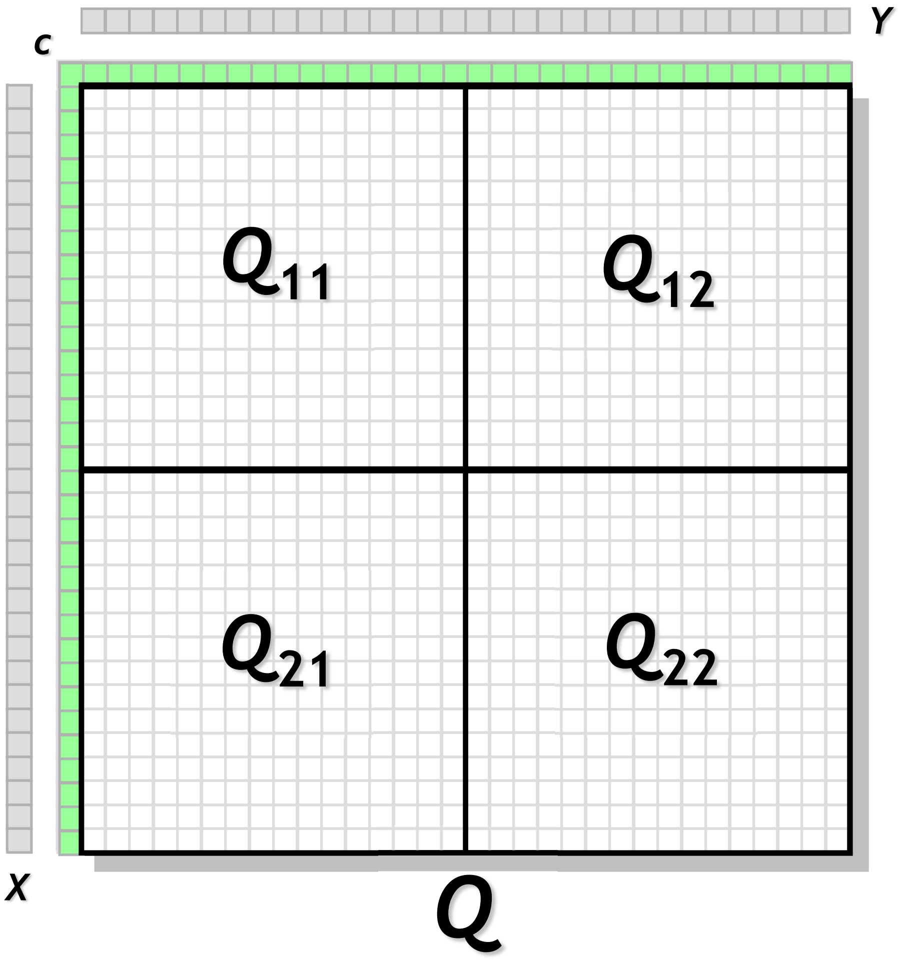
$n = 2^q$

1. **Decompose Q:**

   Split *Q* into four quadrants.

2. **Forward Pass (Generate Boundaries):**

   Generate the right and the bottom boundaries of the quadrants recursively. (of at most 3 quadrants)



🟩 stored values
🟥 traceback path

Sequential Cache-efficient LCS Algorithm

1. **Decompose Q:**

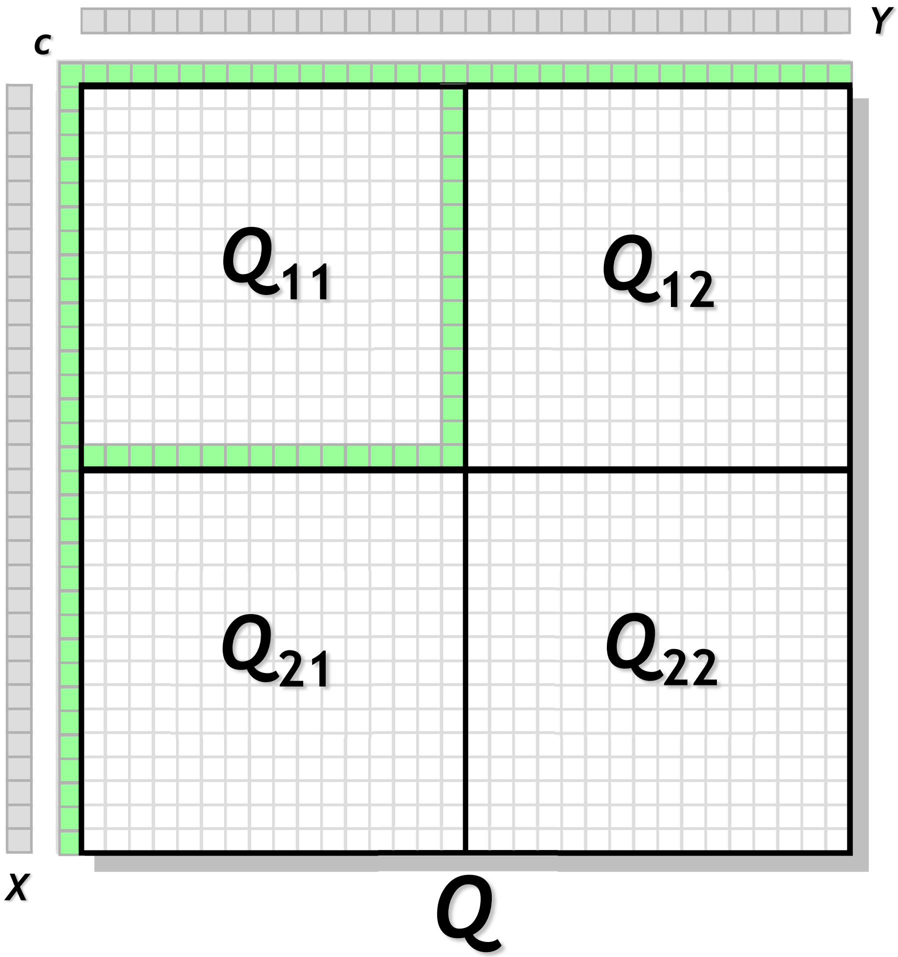
   Split Q into four quadrants.

2. **Forward Pass (Generate Boundaries):**

   Generate the right and the bottom boundaries of the quadrants recursively. (of at most 3 quadrants)

$Q \equiv c[1 \ldots n, 1 \ldots n]$

$n = 2^q$



🟩 stored values

🟥 traceback path

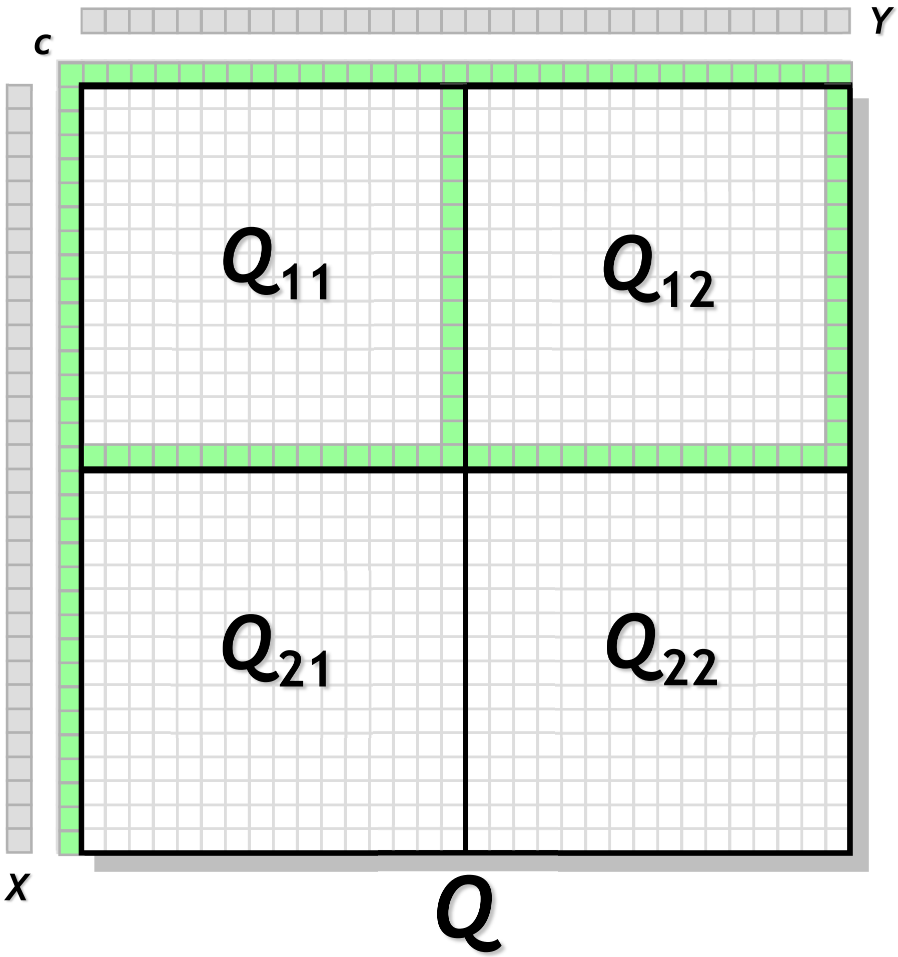
Sequential Cache-efficient LCS Algorithm

$Q \equiv c[1 \dots n, 1 \dots n]$

$n = 2^q$

1. **Decompose Q:**
   Split Q into four quadrants.

2. **Forward Pass (Generate Boundaries):**
   Generate the right and the bottom boundaries of the quadrants recursively.
   (of at most 3 quadrants)



🟩 stored values

🟥 traceback path

Sequential Cache-efficient LCS Algorithm

$Q \equiv c[1 \dots n, 1 \dots n]$

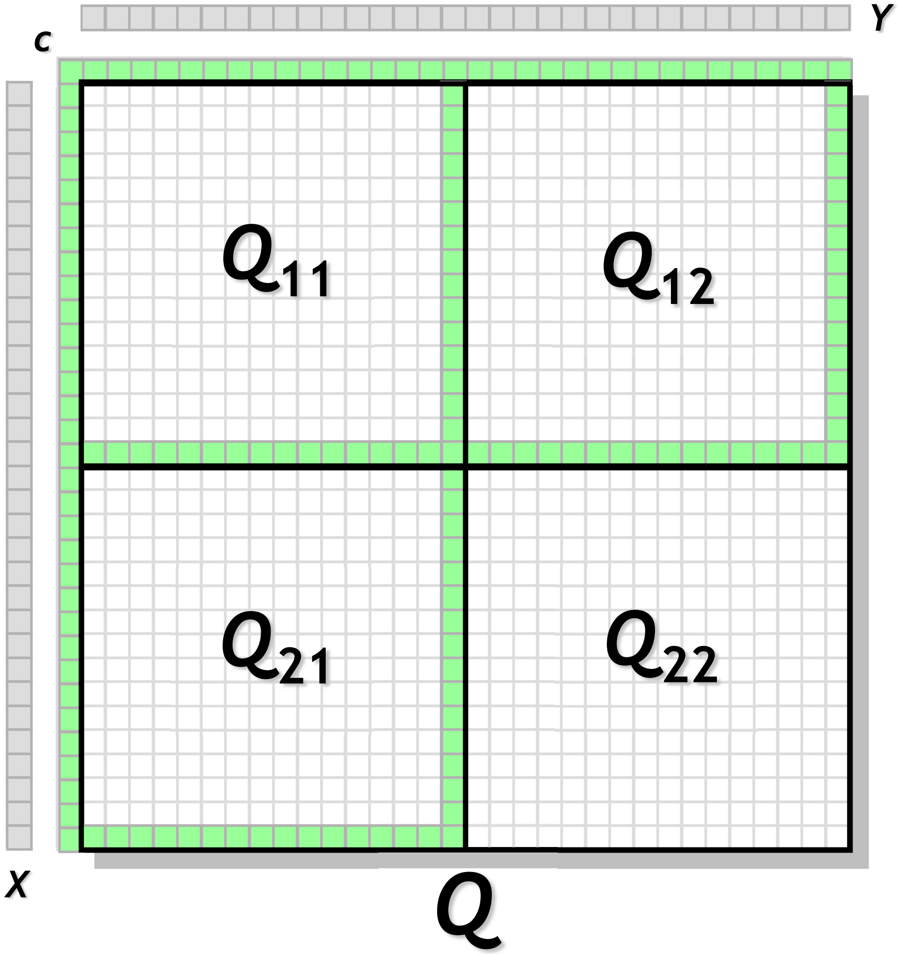
$n = 2^q$

1. **Decompose Q:**

   Split Q into four quadrants.

2. **Forward Pass (Generate Boundaries):**

   Generate the right and the bottom boundaries of the quadrants recursively. (of at most 3 quadrants)



🟩 stored values

🟥 traceback path

Sequential Cache-efficient LCS Algorithm

$Q \equiv c[1 \ldots n, 1 \ldots n]$

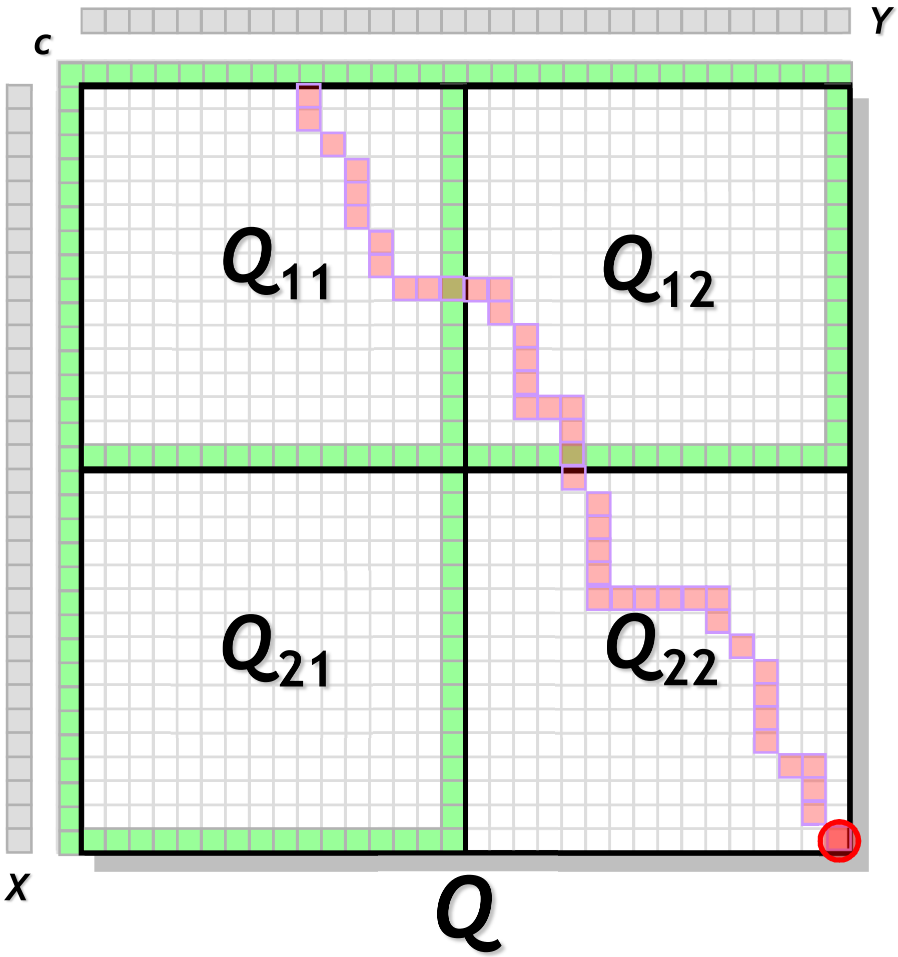
$n = 2^q$

1. **Decompose Q:**

   Split Q into four quadrants.

2. **Forward Pass (Generate Boundaries):**

   Generate the right and the bottom boundaries of the quadrants recursively. (of at most 3 quadrants)



🟩 stored values

🟥 traceback path

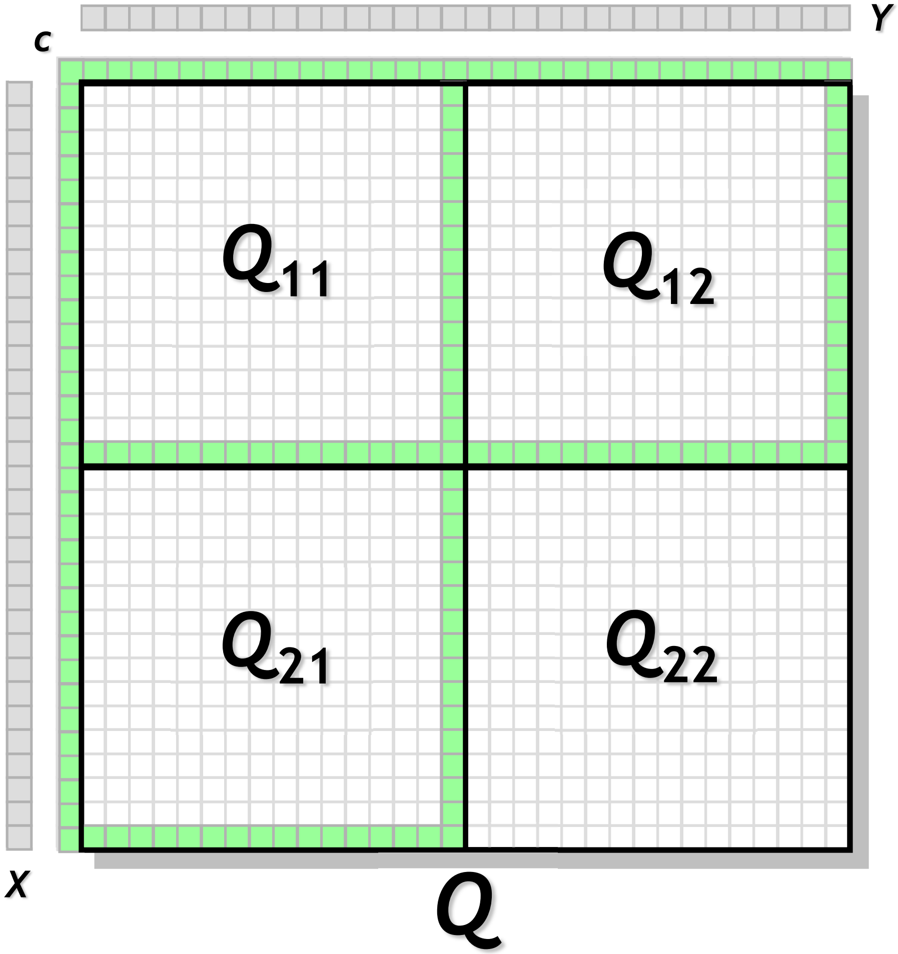
Sequential Cache-efficient LCS Algorithm

$Q \equiv c[1 \dots n, 1 \dots n]$

$n = 2^q$

1. **Decompose Q:**
   Split Q into four quadrants.

2. **Forward Pass (Generate Boundaries):**
   Generate the right and the bottom boundaries of the quadrants recursively.
   (of at most 3 quadrants)



🟩 stored values
🟥 traceback path

Sequential Cache-efficient LCS Algorithm

$Q \equiv c[1 \ldots n, 1 \ldots n]$

$n = 2^q$

1. **Decompose Q:**

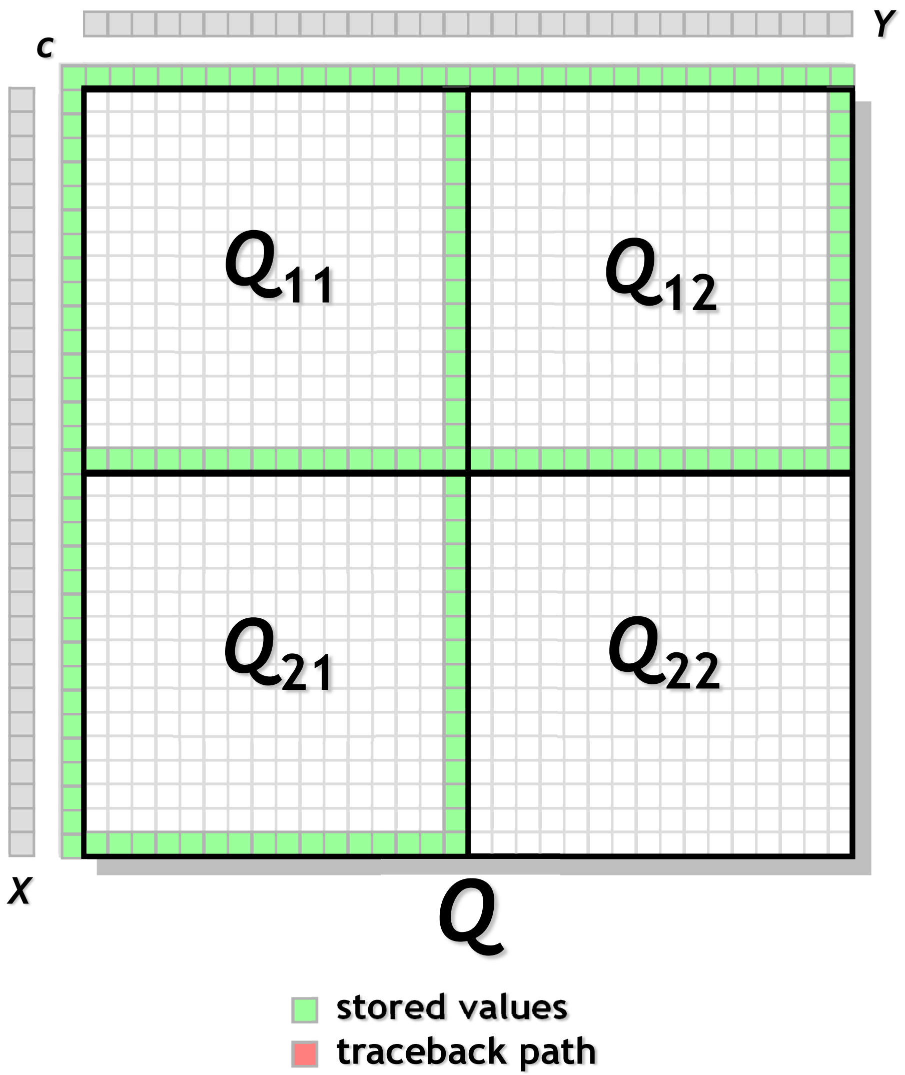
   Split Q into four quadrants.

2. **Forward Pass (Generate Boundaries):**

   Generate the right and the bottom boundaries of the quadrants recursively. (of at most 3 quadrants)

3. **Backward Pass (Extract Traceback Path Fragments):**

   Extract fragments of the traceback path from the quadrants recursively. (from at most 3 quadrants)



Sequential Cache-efficient LCS Algorithm

1. **Decompose Q:**

   Split Q into four quadrants.

2. **Forward Pass (Generate Boundaries):**

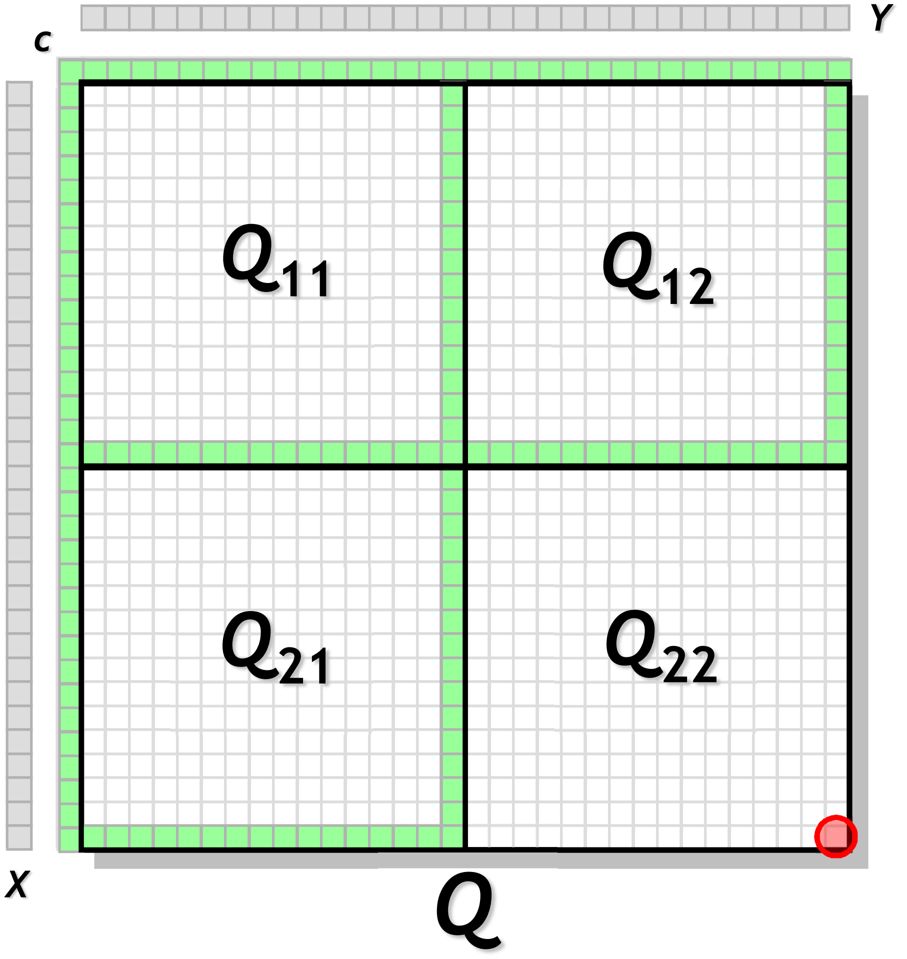
   Generate the right and the bottom boundaries of the quadrants recursively. (of at most 3 quadrants)

3. **Backward Pass (Extract Traceback Path Fragments):**

   Extract fragments of the traceback path from the quadrants recursively. (from at most 3 quadrants)

$Q \equiv c[1 \ldots n, 1 \ldots n]$

$n = 2^q$



🟩 stored values

🟥 traceback path

Sequential Cache-efficient LCS Algorithm

$Q \equiv c[1 \ldots n, 1 \ldots n]$

$n = 2^q$

1. Decompose Q:

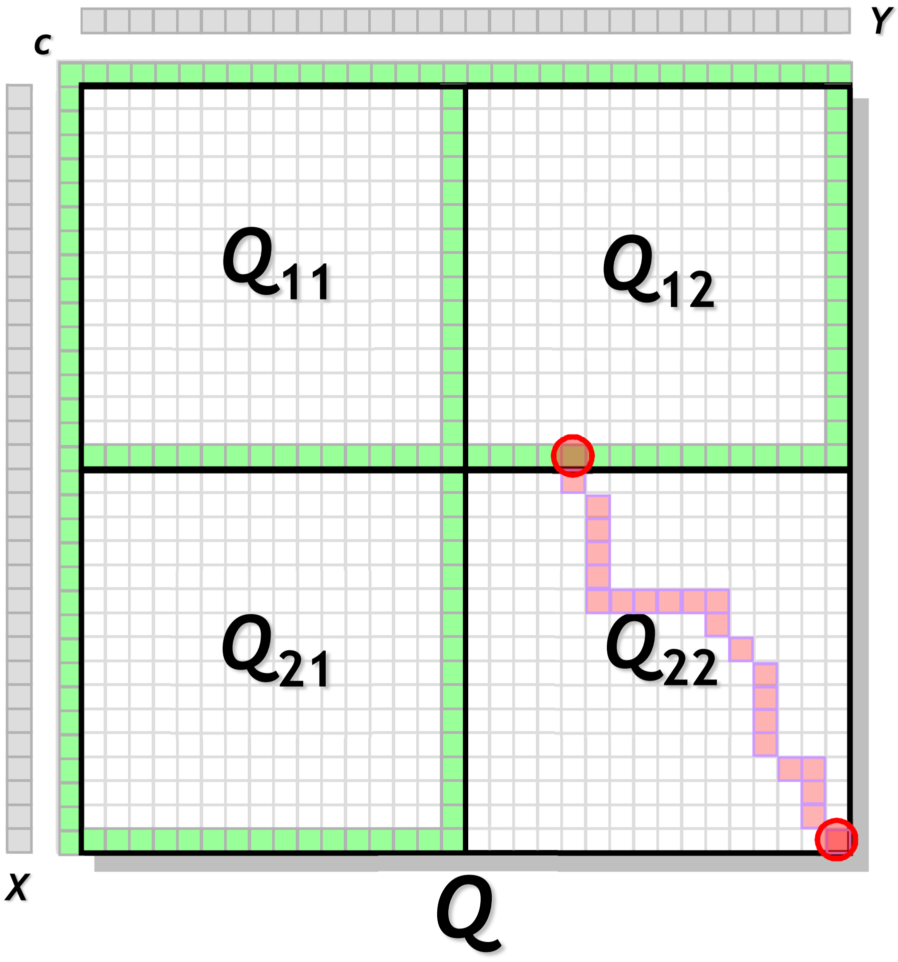
   Split *Q* into four quadrants.

2. Forward Pass (Generate Boundaries):

   Generate the right and the bottom boundaries of the quadrants recursively. (of at most 3 quadrants)

3. Backward Pass (Extract Traceback Path Fragments):

   Extract fragments of the traceback path from the quadrants recursively. (from at most 3 quadrants)



🟩 stored values
🟥 traceback path

Sequential Cache-efficient LCS Algorithm

$Q \equiv c[1 \ldots n, 1 \ldots n]$

$n = 2^q$

1. **Decompose Q:**

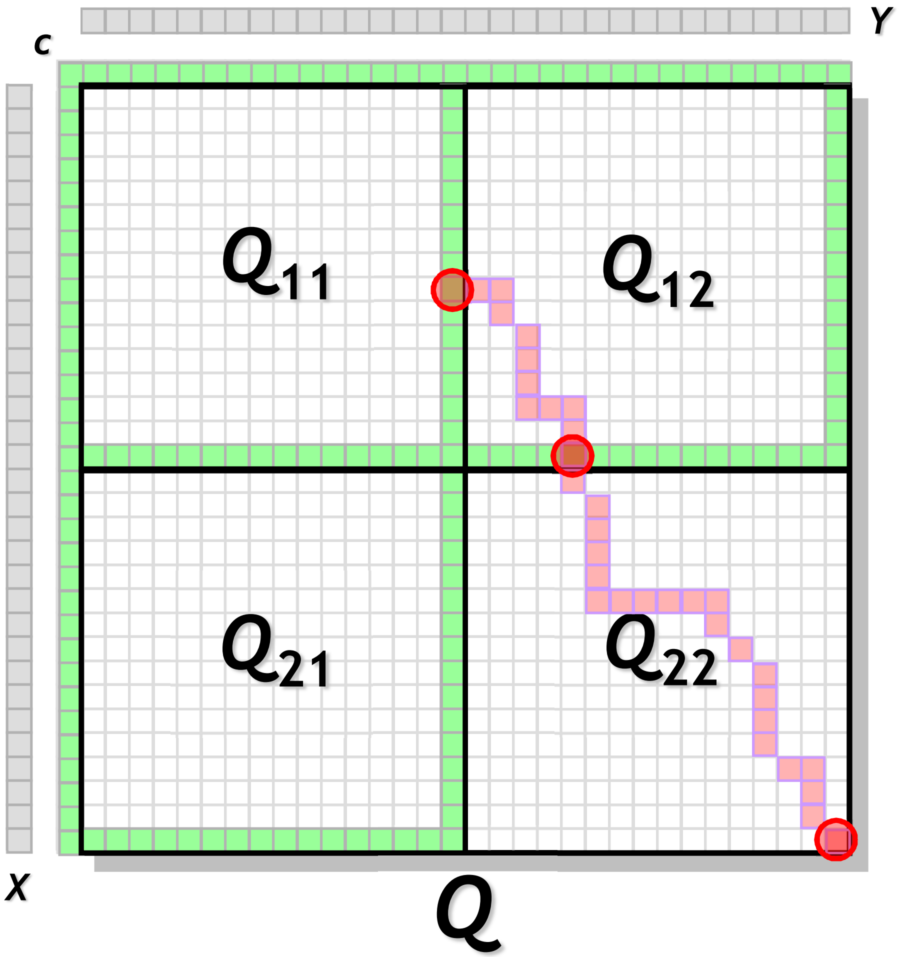
   Split Q into four quadrants.

2. **Forward Pass (Generate Boundaries):**

   Generate the right and the bottom boundaries of the quadrants recursively. (of at most 3 quadrants)

3. **Backward Pass (Extract Traceback Path Fragments):**

   Extract fragments of the traceback path from the quadrants recursively. (from at most 3 quadrants)



🟩 stored values

🟥 traceback path

Sequential Cache-efficient LCS Algorithm

1. **Decompose Q:**

   Split Q into four quadrants.

2. **Forward Pass (Generate Boundaries):**

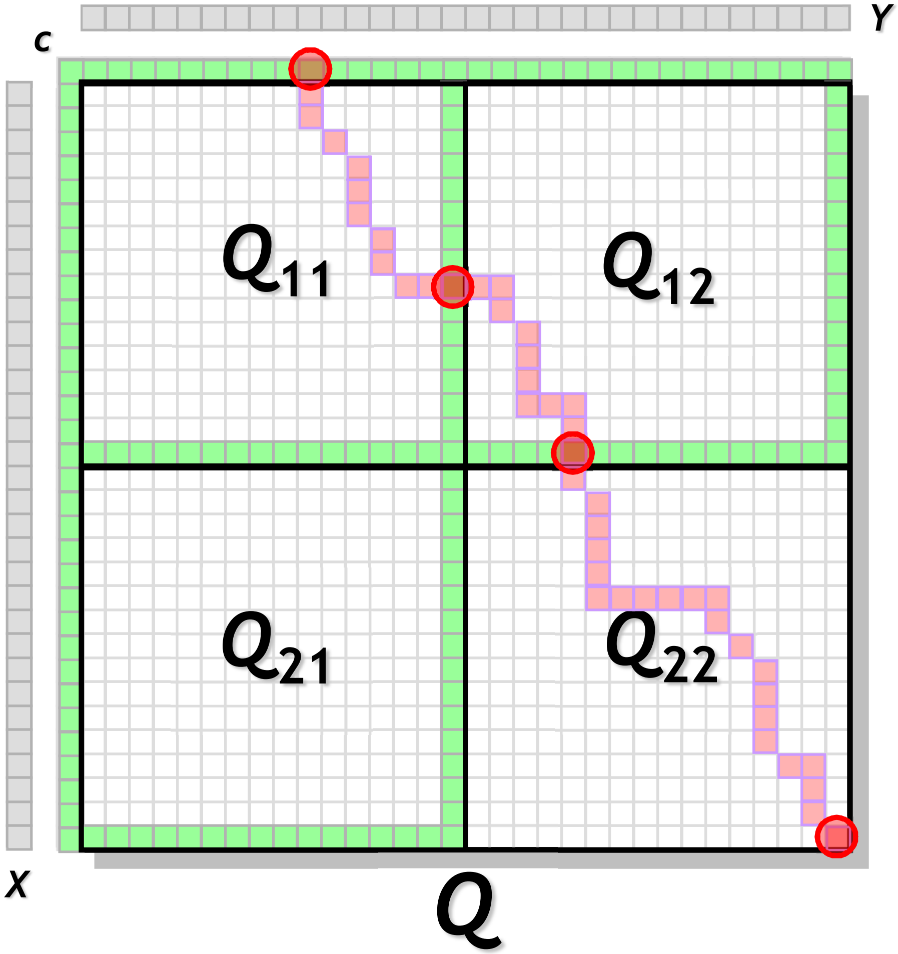
   Generate the right and the bottom boundaries of the quadrants recursively. (of at most 3 quadrants)

3. **Backward Pass (Extract Traceback Path Fragments):**

   Extract fragments of the traceback path from the quadrants recursively. (from at most 3 quadrants)

$Q \equiv c[1 \ldots n, 1 \ldots n]$

$n = 2^q$



🟩 stored values

🟥 traceback path

Sequential Cache-efficient LCS Algorithm

$Q \equiv c[1 \ldots n, 1 \ldots n]$

$n = 2^q$

1. **Decompose Q:**

   Split Q into four quadrants.

2. **Forward Pass (Generate Boundaries):**

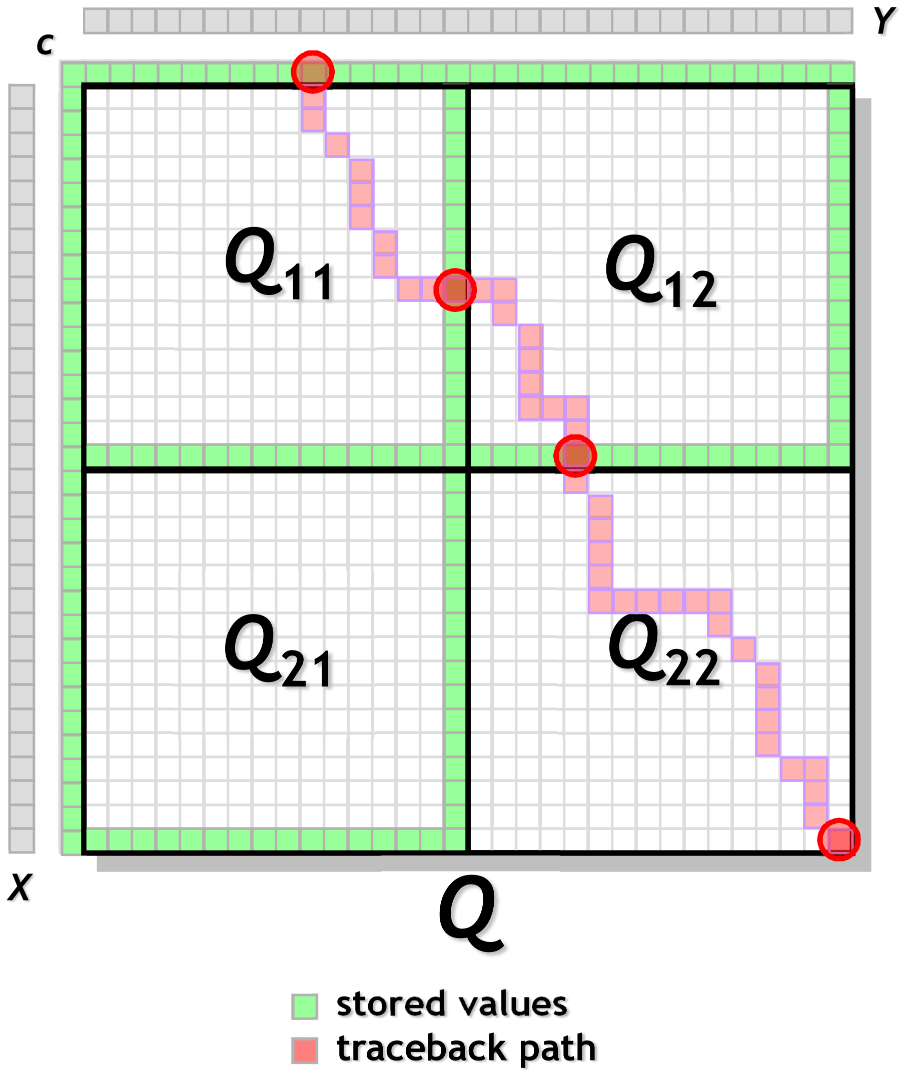
   Generate the right and the bottom boundaries of the quadrants recursively. (of at most 3 quadrants)

3. **Backward Pass (Extract Traceback Path Fragments):**

   Extract fragments of the traceback path from the quadrants recursively. (from at most 3 quadrants)

4. **Compose Traceback Path:**

   Combine the path fragments.



Cache Performance

Cache complexity:

$$Q_1(n) \leq \begin{cases} \mathrm{O}\left(1 + \frac{n}{B}\right), & if\ n \leq \alpha C, \\ 3Q'_1\left(\frac{n}{2}\right) + 3Q_1\left(\frac{n}{2}\right) + \mathrm{O}\left(1 + \frac{n}{B}\right), & otherwise. \end{cases}$$

where $Q'_1(n)$ is the cache complexity of *recursive boundary generation* (in the forward pass):

$$Q'_1(n) \leq \begin{cases} \mathrm{O}\left(1 + \frac{n}{B}\right), & if\ n \leq \alpha C, \\ 4Q'_1\left(\frac{n}{2}\right) + \mathrm{O}(1), & otherwise. \end{cases}$$

$$= \mathrm{O}\left(\frac{n^2}{BC} + \frac{n}{B} + 1\right)$$

Substituting, $\boxed{Q_1(n) = \mathrm{O}\left(\frac{n^2}{BC} + \frac{n}{B} + 1\right)}$

Cache Performance

Cache complexity:

$$Q_1(n) \le \begin{cases} \mathrm{O}\left(1 + \frac{n}{B}\right), & if\ n \le \alpha C, \\ 3Q'_1\left(\frac{n}{2}\right) + 3Q_1\left(\frac{n}{2}\right) + \mathrm{O}\left(1 + \frac{n}{B}\right), & otherwise. \end{cases}$$

where $Q'_1(n)$ is the cache complexity of *recursive boundary generation* (in the forward pass):

$$Q'_1(n) \le \begin{cases} \mathrm{O}\left(1 + \frac{n}{B}\right), & if\ n \le \alpha C, \\ 4Q'_1\left(\frac{n}{2}\right) + \mathrm{O}(1), & otherwise. \end{cases}$$

$$= \mathrm{O}\left(\frac{n^2}{BC} + \frac{n}{B} + 1\right)$$

Substituting, $\boxed{Q_1(n) = \mathrm{O}\left(\frac{n^2}{BC} + \frac{n}{B} + 1\right)}$ ← ***optimal***

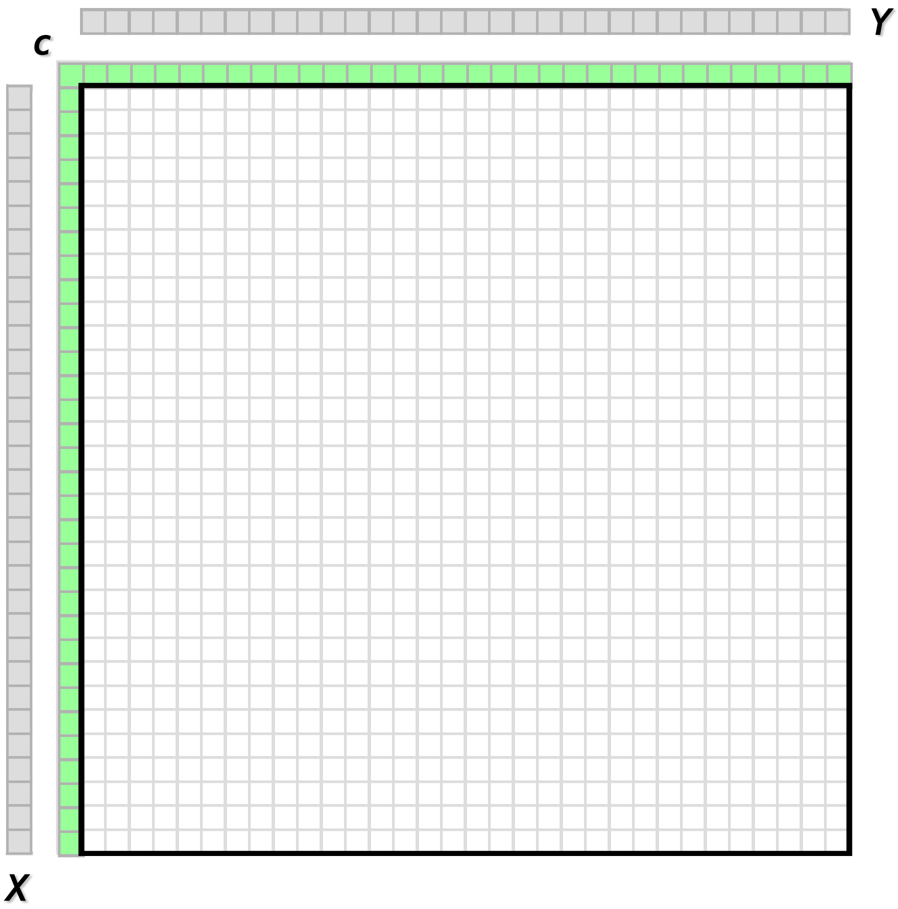
Parallel
Cache-efficient LCS Algorithm
for Distributed Caches

Parallel Cache-efficient Boundary Computation

(Par-Boundary)

$Q \equiv c[1 \ldots n, 1 \ldots n]$

$\boxed{n > pC}$

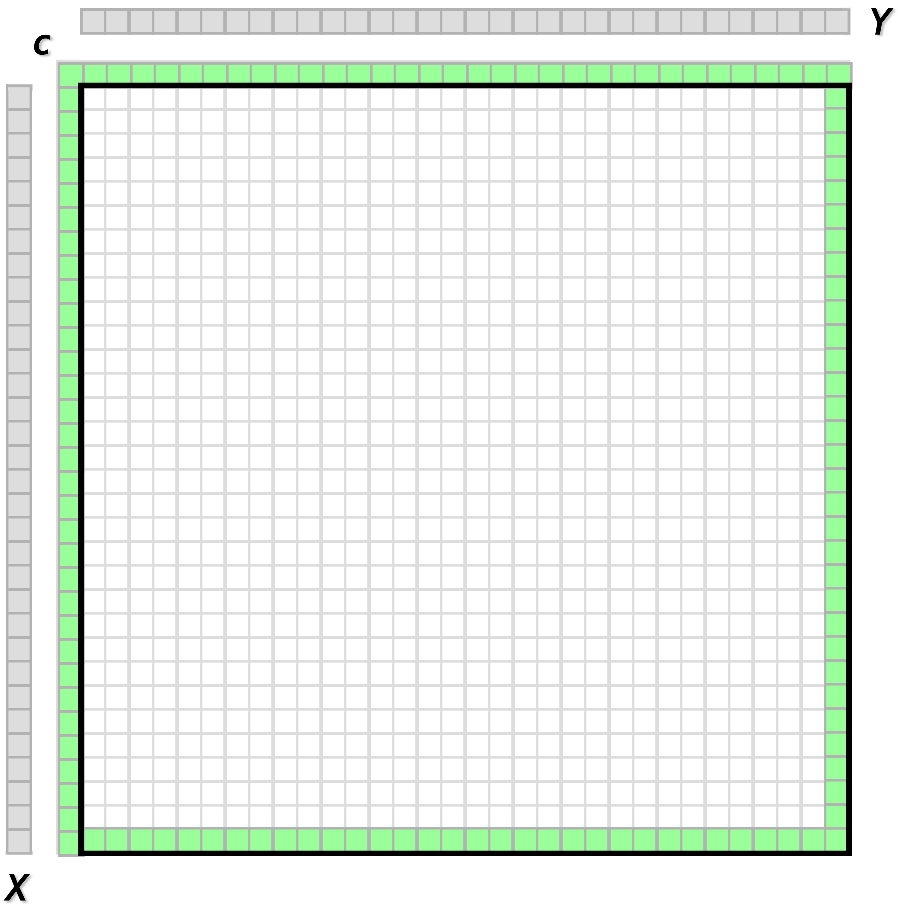


🟩 stored values

Parallel Cache-efficient Boundary Computation
(Par-Boundary)

$Q \equiv c[1 \ldots n, 1 \ldots n]$

$\boxed{n > pC}$



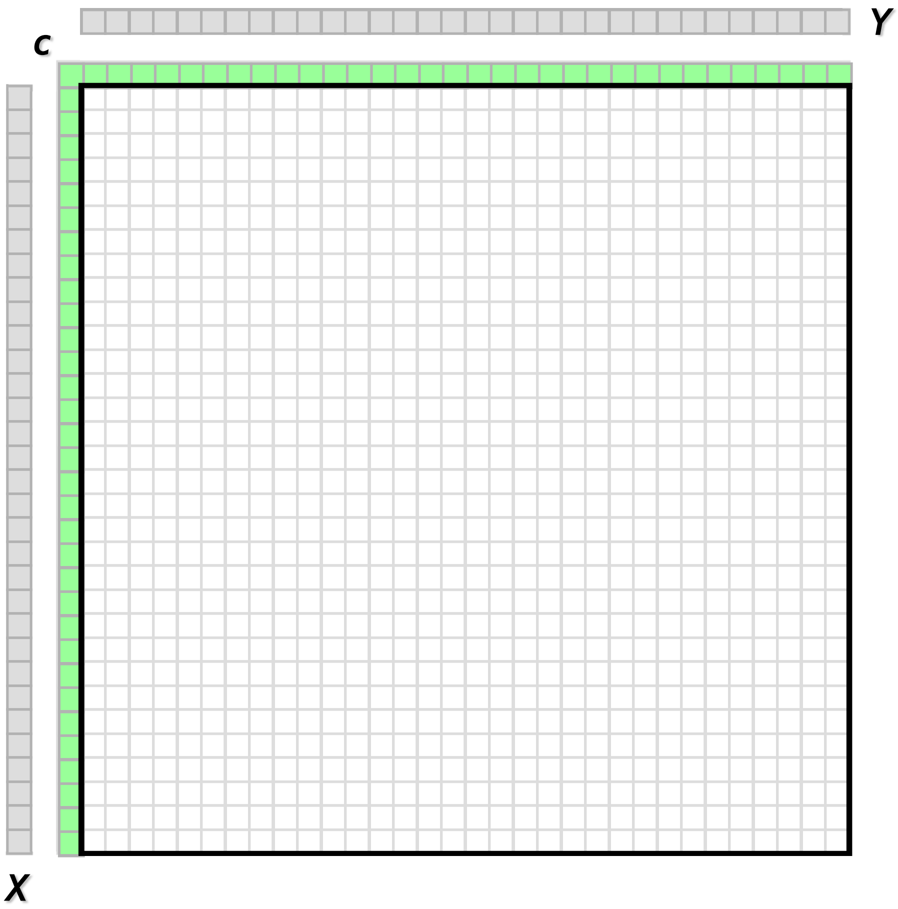
🟩 stored values

Parallel Cache-efficient Boundary Computation

(PAR-BOUNDARY)

$Q \equiv c[1 \ldots n, 1 \ldots n]$

$\boxed{n > pC}$



🟩 stored values

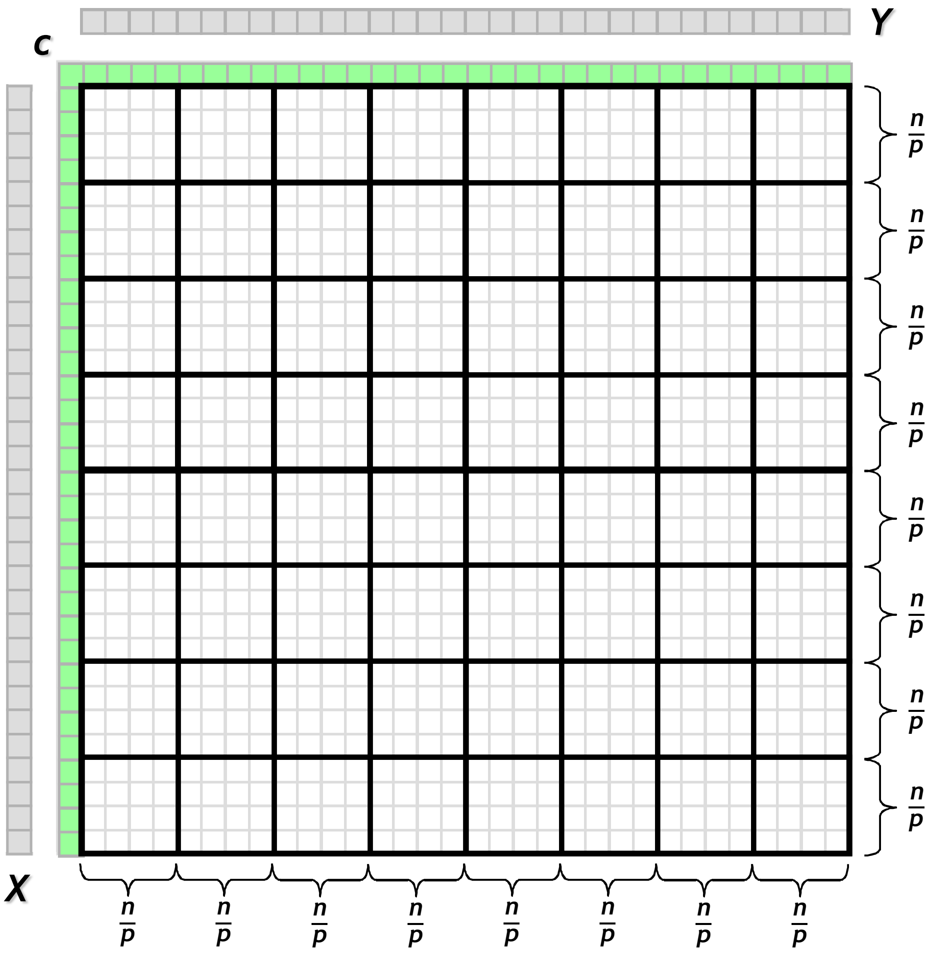
Parallel Cache-efficient Boundary Computation
(PAR-BOUNDARY)

$Q \equiv c[1 \ldots n, 1 \ldots n]$

$\boxed{n > pC}$

1. **Decompose Q:**

   Split Q into p^2 submatrices of size $(n / p) \times (n / p)$ each.



🟩 stored values

Parallel Cache-efficient Boundary Computation
(PAR-BOUNDARY)

$Q \equiv c[1 \dots n, 1 \dots n]$

$\boxed{n > pC}$

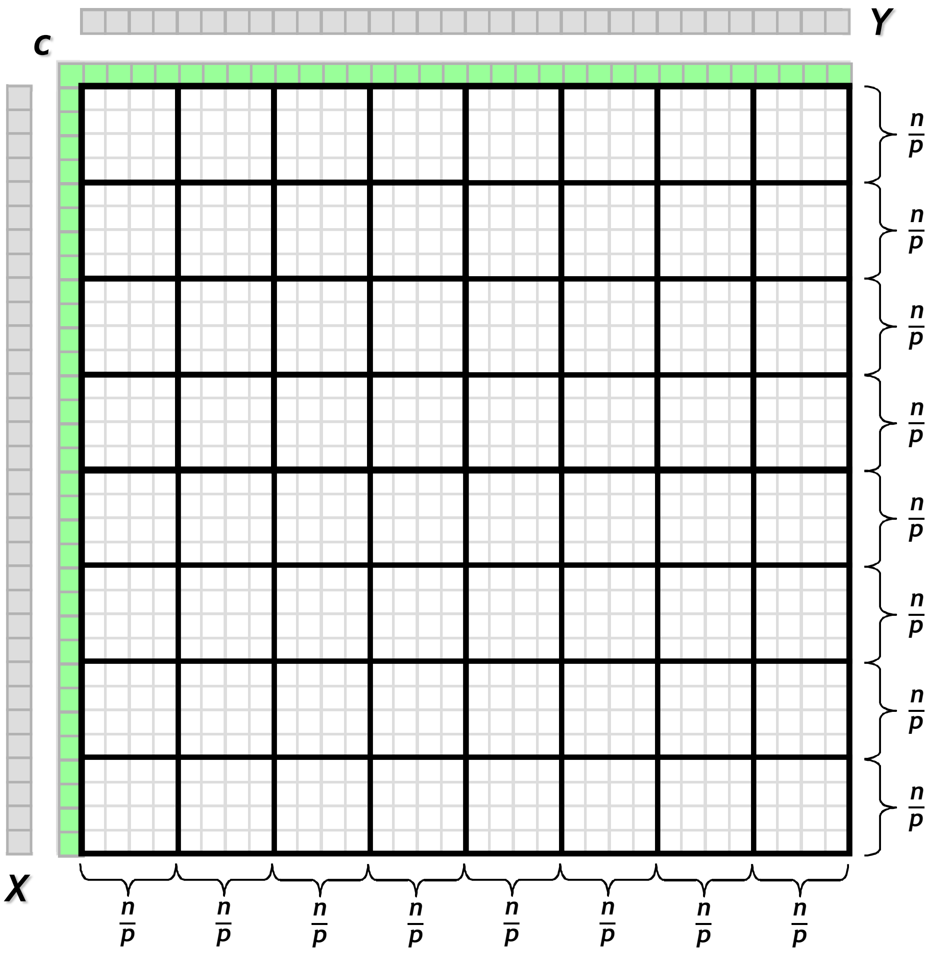
1. **Decompose Q:**

   Split Q into p^2 submatrices of size $(n / p) \times (n / p)$ each.

2. **Generate Boundaries:**

   In iteration $i \in [1, 2p- 1]$, solve all submatrices on the i-th forward diagonal in parallel using the sequential cache-oblivious algorithm.

   For each cell also compute:
   the cell on the input boundary where the traceback path through the given cell intersects.



🟩 stored values

Parallel Cache-efficient Boundary Computation
(PAR-BOUNDARY)

$Q \equiv c[1 \ldots n, 1 \ldots n]$

$n > pC$

1. Decompose Q:

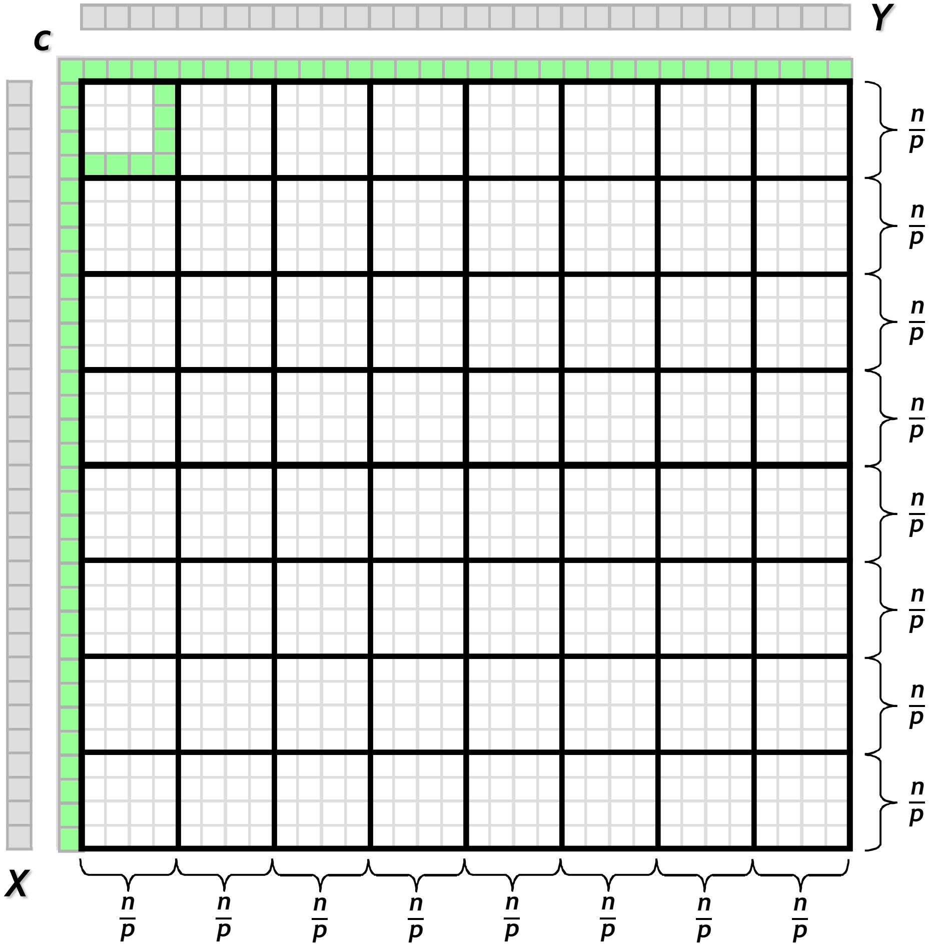
   Split Q into p^2 submatrices of size $(n / p) \times (n / p)$ each.

2. Generate Boundaries:

   In iteration $i \in [1, 2p - 1]$, solve all submatrices on the i-th forward diagonal in parallel using the sequential cache-oblivious algorithm.

For each cell also compute:

the cell on the input boundary where the traceback path through the given cell intersects.



🟩 stored values

Parallel Cache-efficient Boundary Computation
(Par-Boundary)

$Q \equiv c[1 \ldots n, 1 \ldots n]$

$\boxed{n > pC}$

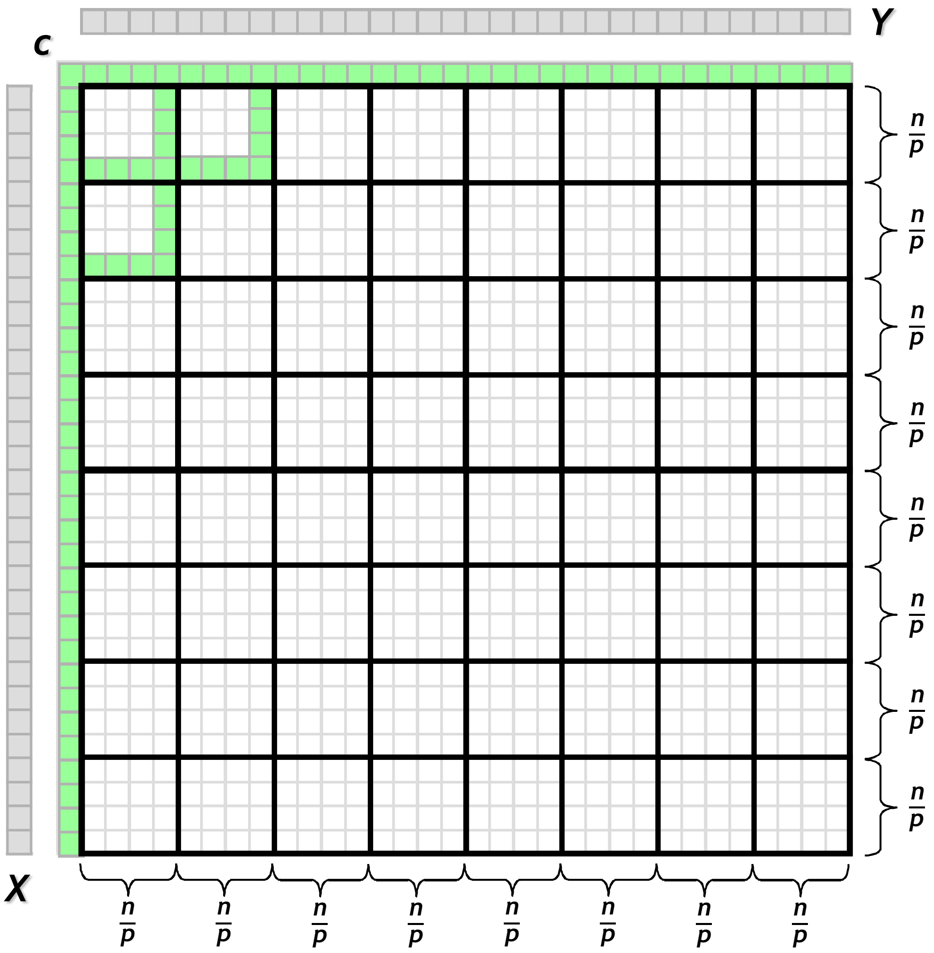
1. **Decompose Q:**

   Split Q into p^2 submatrices of size $(n / p) \times (n / p)$ each.

2. **Generate Boundaries:**

   In iteration $i \in [1, 2p- 1]$, solve all submatrices on the i-th forward diagonal in parallel using the sequential cache-oblivious algorithm.

   For each cell also compute:
   the cell on the input boundary where the traceback path through the given cell intersects.



🟩 stored values

Parallel Cache-efficient Boundary Computation
(PAR-BOUNDARY)

$Q \equiv c[1 \dots n,\ 1 \dots n]$

$\boxed{n > pC}$

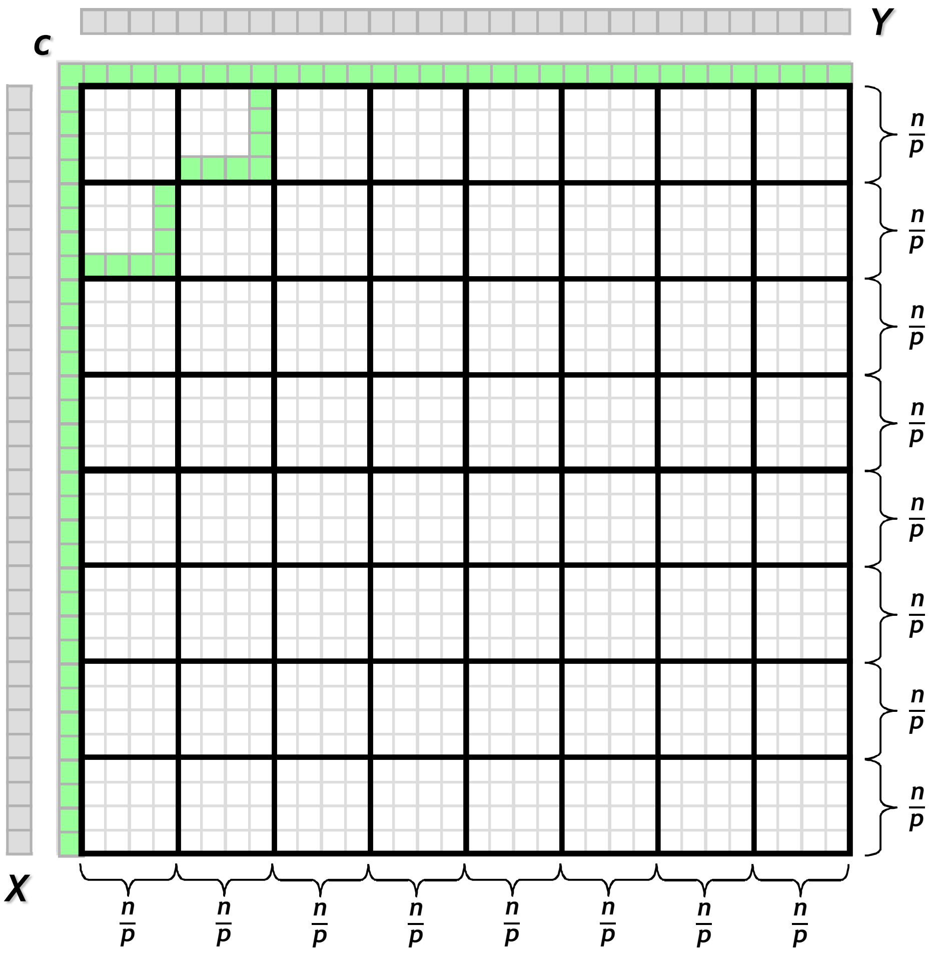
1. **Decompose Q:**

   Split Q into p^2 submatrices of size $(\ n\ /\ p\ ) \times (\ n\ /\ p\ )$ each.

2. **Generate Boundaries:**

   In iteration $i \in [1,\ 2p-1]$, solve all submatrices on the i-th forward diagonal in parallel using the sequential cache-oblivious algorithm.

   For each cell also compute:
   the cell on the input boundary where the traceback path through the given cell intersects.



🟩 stored values

Parallel Cache-efficient Boundary Computation
(Par-Boundary)

$Q \equiv c[1 \ldots n,\ 1 \ldots n]$

$\boxed{n > pC}$

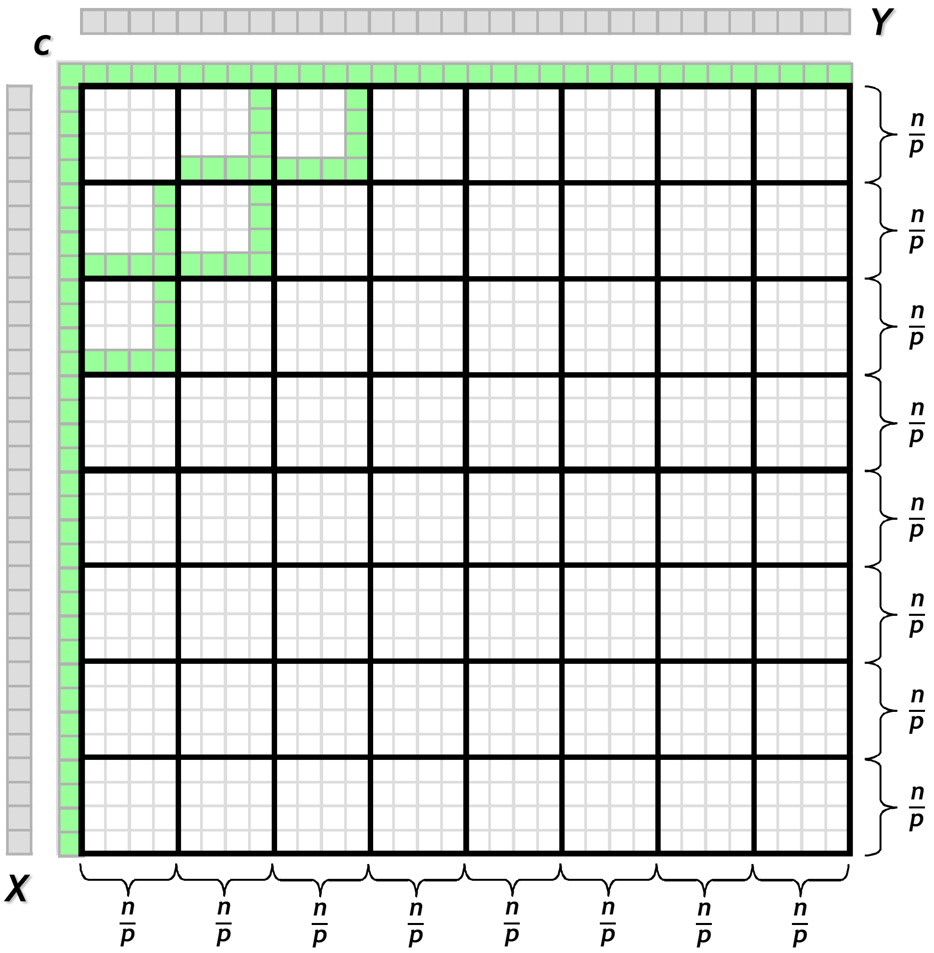
1. Decompose Q:

   Split Q into p^2 submatrices of size $(n / p) \times (n / p)$ each.

2. Generate Boundaries:

   In iteration $i \in [1, 2p - 1]$, solve all submatrices on the i-th forward diagonal in parallel using the sequential cache-oblivious algorithm.

   For each cell also compute:
   the cell on the input boundary where the traceback path through the given cell intersects.



🟩 stored values

Parallel Cache-efficient Boundary Computation
(Par-Boundary)

$Q \equiv c[1 \dots n,\ 1 \dots n]$

$\boxed{n > pC}$

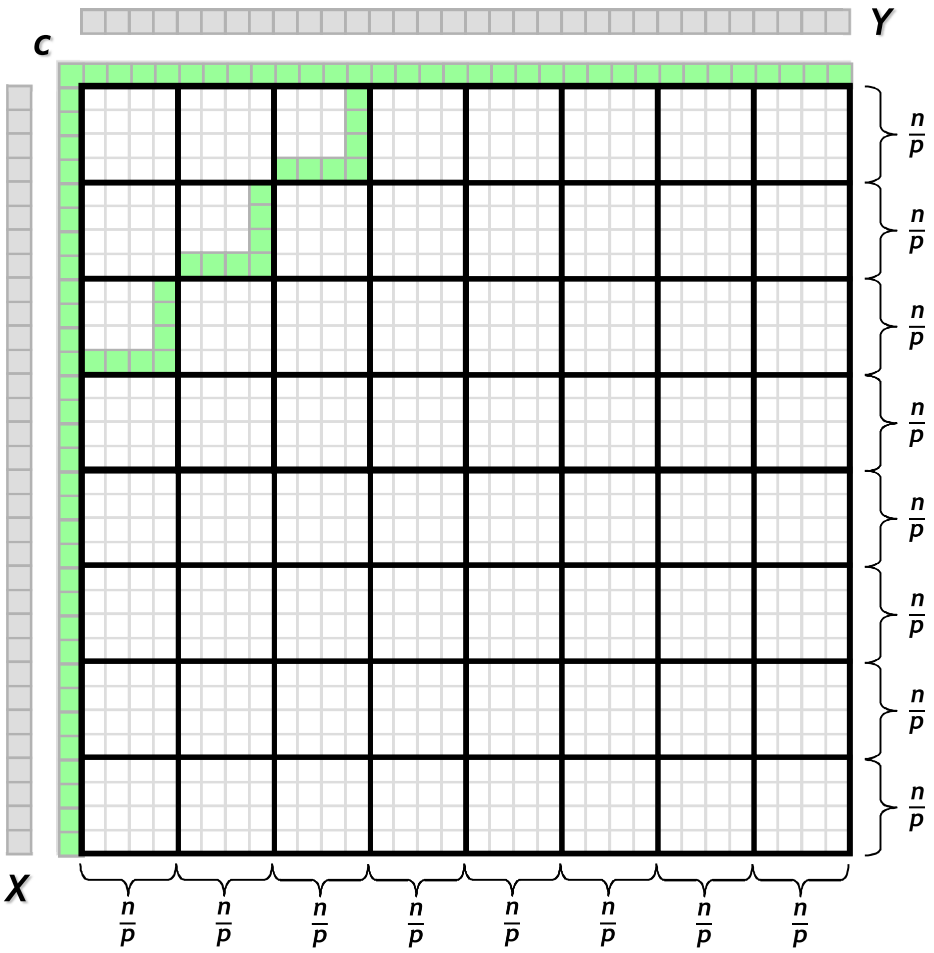
1. **Decompose Q:**

   Split Q into p^2 submatrices of size $(n / p) \times (n / p)$ each.

2. **Generate Boundaries:**

   In iteration $i \in [1,\ 2p-1]$, solve all submatrices on the i-th forward diagonal in parallel using the sequential cache-oblivious algorithm.

   For each cell also compute:
   the cell on the input boundary where the traceback path through the given cell intersects.



🟩 stored values

Parallel Cache-efficient Boundary Computation

(Par-Boundary)

$Q \equiv c[1 \ldots n,\ 1 \ldots n]$

$\boxed{n > pC}$

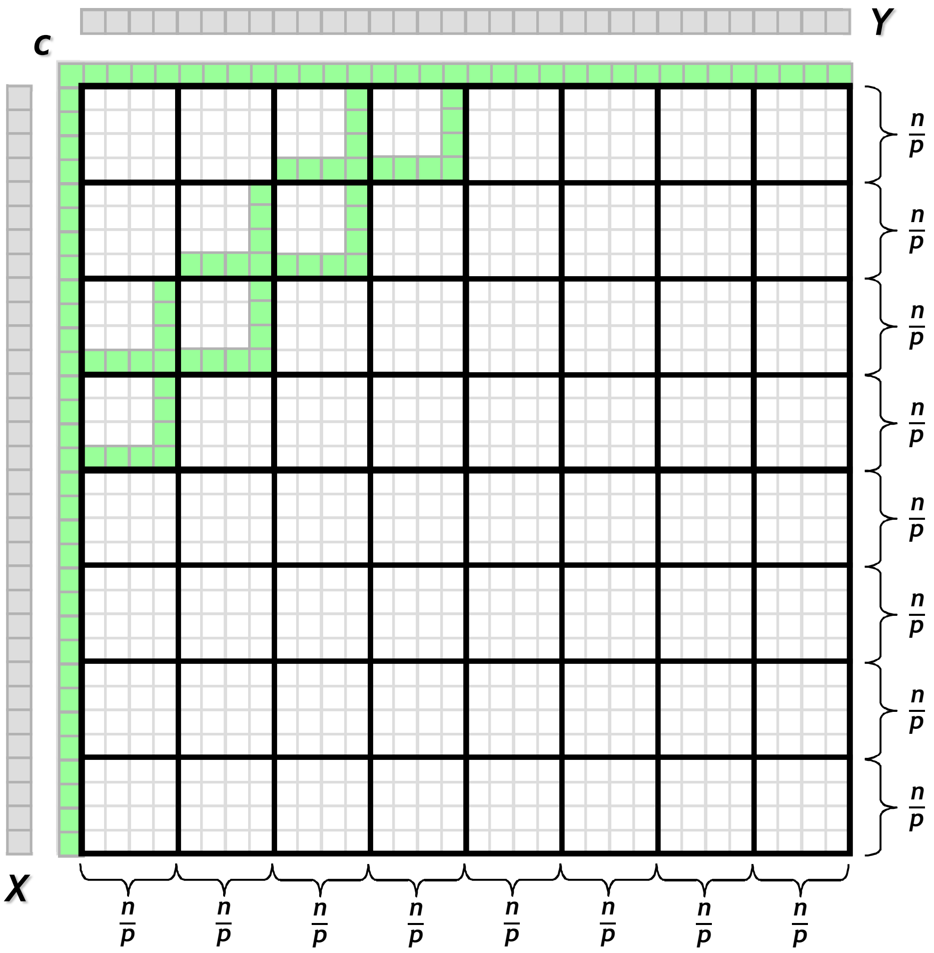
1. **Decompose Q:**

   Split Q into p^2 submatrices of size $(n / p) \times (n / p)$ each.

2. **Generate Boundaries:**

   In iteration $i \in [1,\ 2p-1]$, solve all submatrices on the i-th forward diagonal in parallel using the sequential cache-oblivious algorithm.

   For each cell also compute:
   the cell on the input boundary where the traceback path through the given cell intersects.



🟩 stored values

Parallel Cache-efficient Boundary Computation
(Par-Boundary)

$Q \equiv c[1 \ldots n,\ 1 \ldots n]$

$\boxed{n > pC}$

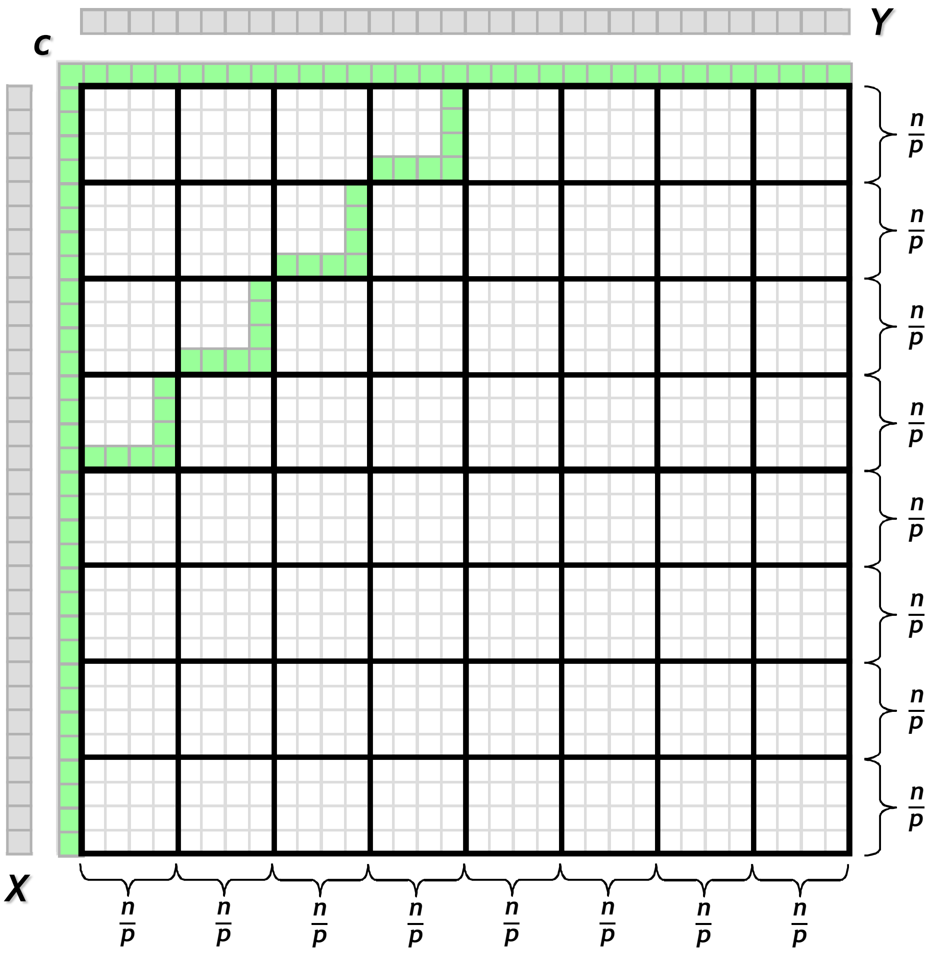
1. **Decompose Q:**

   Split Q into p^2 submatrices of size $(n / p) \times (n / p)$ each.

2. **Generate Boundaries:**

   In iteration $i \in [1, 2p - 1]$, solve all submatrices on the i-th forward diagonal in parallel using the sequential cache-oblivious algorithm.

   For each cell also compute:
   the cell on the input boundary where the traceback path through the given cell intersects.



🟩 stored values

Parallel Cache-efficient Boundary Computation
(Par-Boundary)

$Q \equiv c[1 \ldots n,\ 1 \ldots n]$

$\boxed{n > pC}$

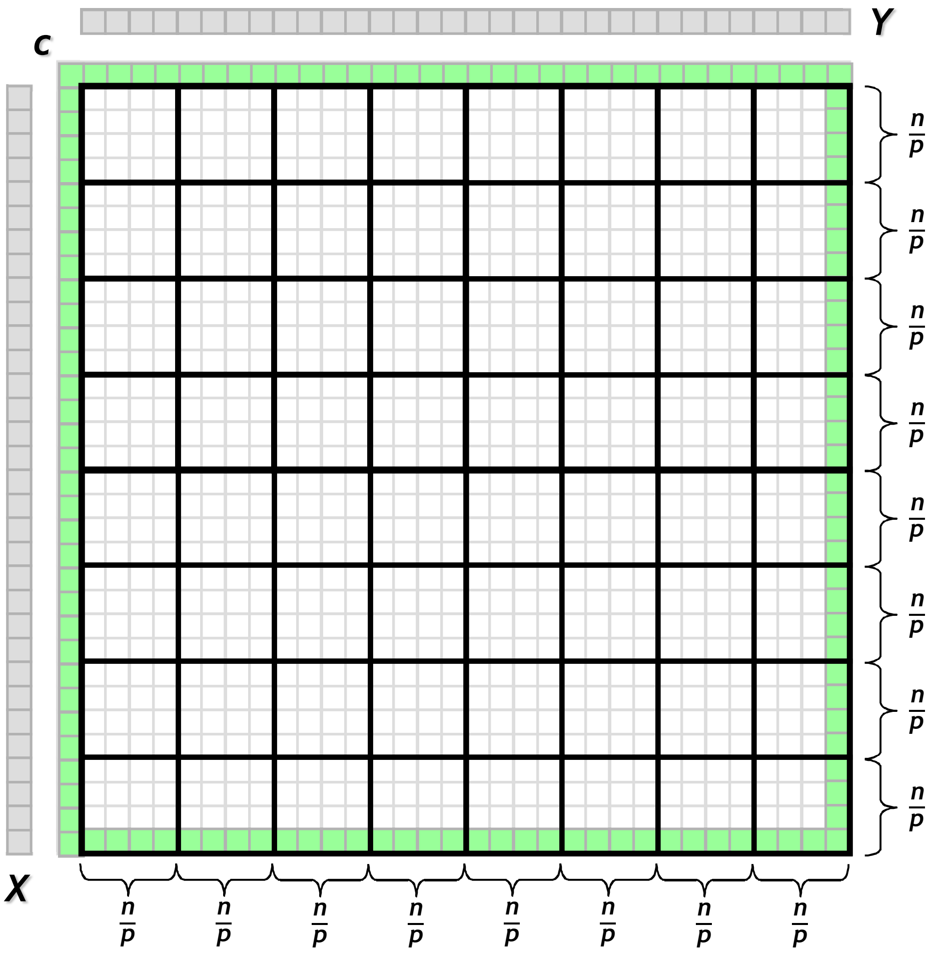
1. **Decompose Q:**

   Split Q into p^2 submatrices of size $(\ n\ /\ p\ ) \times (\ n\ /\ p\ )$ each.

2. **Generate Boundaries:**

   In iteration $i \in [1,\ 2p-1]$, solve all submatrices on the i-th forward diagonal in parallel using the sequential cache-oblivious algorithm.

   For each cell also compute:
   the cell on the input boundary where the traceback path through the given cell intersects.



🟩 stored values

Parallel Cache-efficient Boundary Computation
(Par-Boundary)

$Q \equiv c[1 \ldots n,\ 1 \ldots n]$

$\boxed{n > pC}$

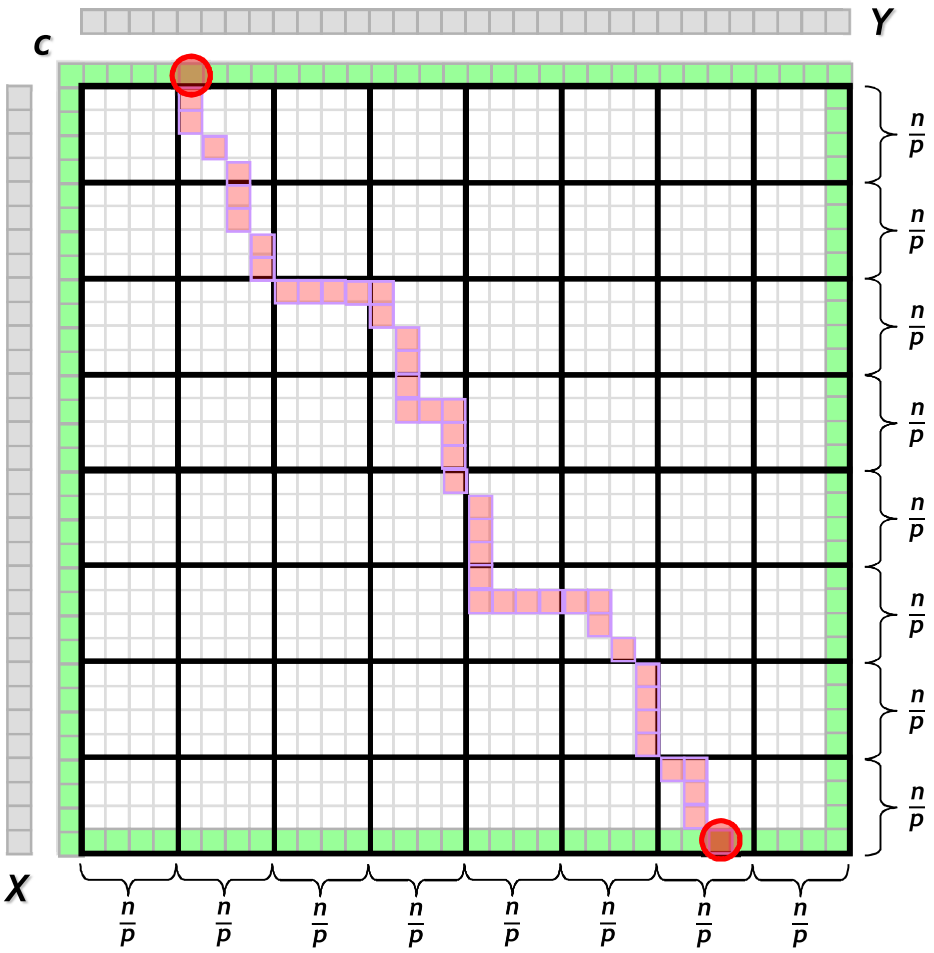
1. **Decompose Q:**

   Split Q into p^2 submatrices of size $(n / p) \times (n / p)$ each.

2. **Generate Boundaries:**

   In iteration $i \in [1, 2p-1]$, solve all submatrices on the i-th forward diagonal in parallel using the sequential cache-oblivious algorithm.

   For each cell also compute:
   the cell on the input boundary where the traceback path through the given cell intersects.



🟩 stored values

Parallel Cache-efficient Boundary Computation
(Par-Boundary)

$Q \equiv c[1 \ldots n,\ 1 \ldots n]$

$\boxed{n > pC}$

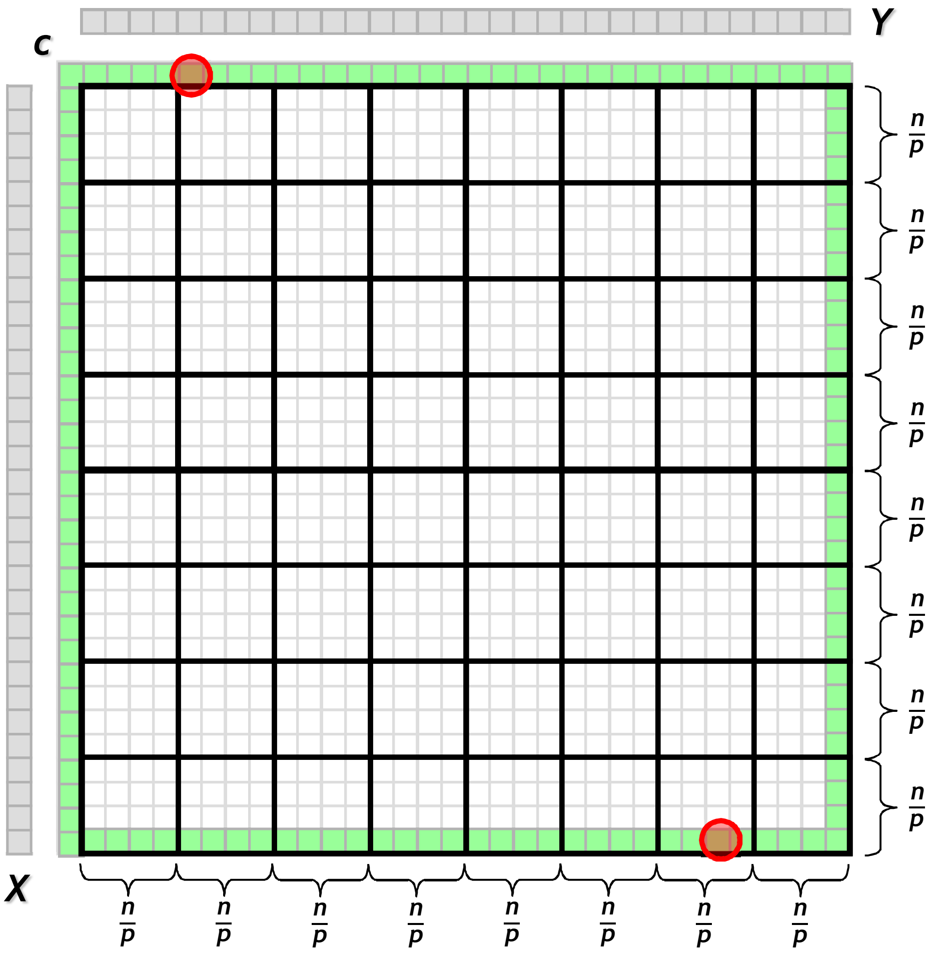
1. Decompose Q:

   Split Q into p^2 submatrices of size $(n / p) \times (n / p)$ each.

2. Generate Boundaries:

   In iteration $i \in [1, 2p - 1]$, solve all submatrices on the i-th forward diagonal in parallel using the sequential cache-oblivious algorithm.

   For each cell also compute:
   the cell on the input boundary where the traceback path through the given cell intersects.



🟩 stored values

Performance Bounds
(PAR-BOUNDARY)

Q ≡ c[1 … n, 1 … n]

n > pC

Parallel Time Complexity:

$$T'_p(n) = \mathrm{O}\left(p \times \left(\frac{n}{p}\right)^2\right) = \mathrm{O}\left(\frac{n^2}{p}\right)$$

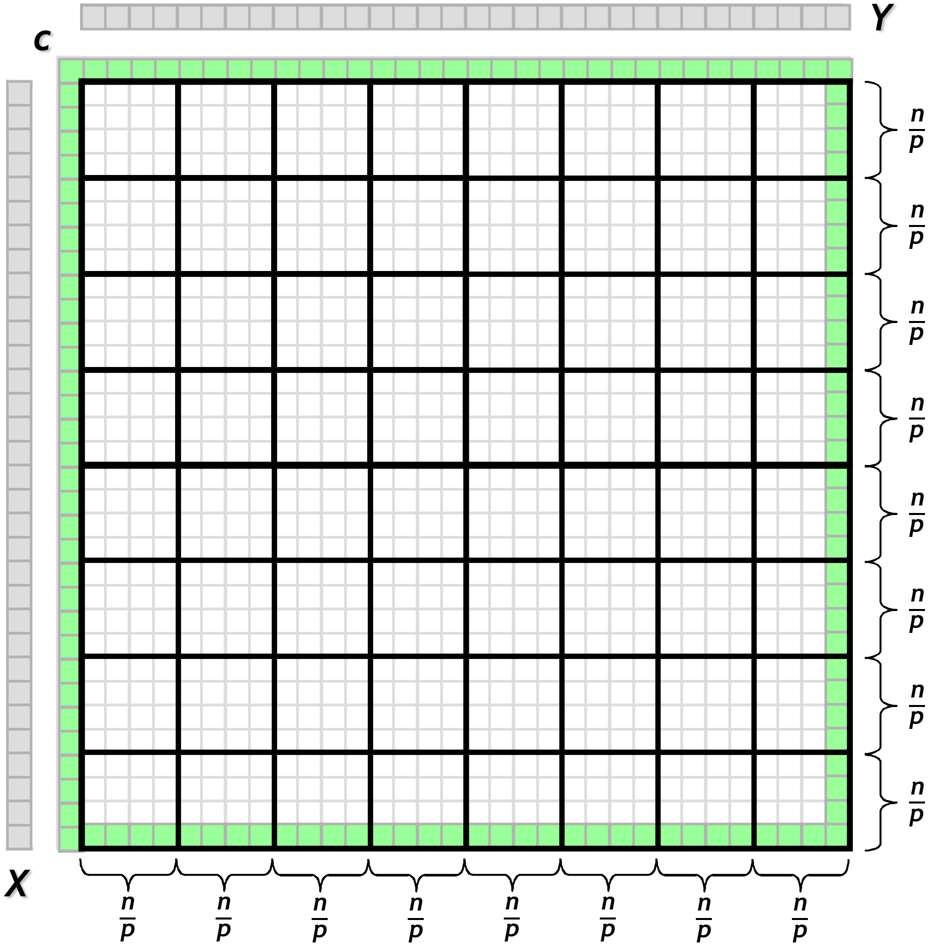
Cache Complexity:

$$Q'_p(n) = \mathrm{O}\left(p^2 \times Q'_1\left(\frac{n}{p}\right)\right)$$

$$= \mathrm{O}\left(p^2 \times \left(\frac{\left(\frac{n}{p}\right)^2}{BC} + \frac{\frac{n}{p}}{B} + 1\right)\right)$$

$$= \mathrm{O}\left(\frac{n^2}{BC} + p \cdot \frac{n}{B} + p^2\right)$$

$$= \mathrm{O}\left(\frac{n^2}{BC}\right) \qquad [since\ n \geq pC]$$



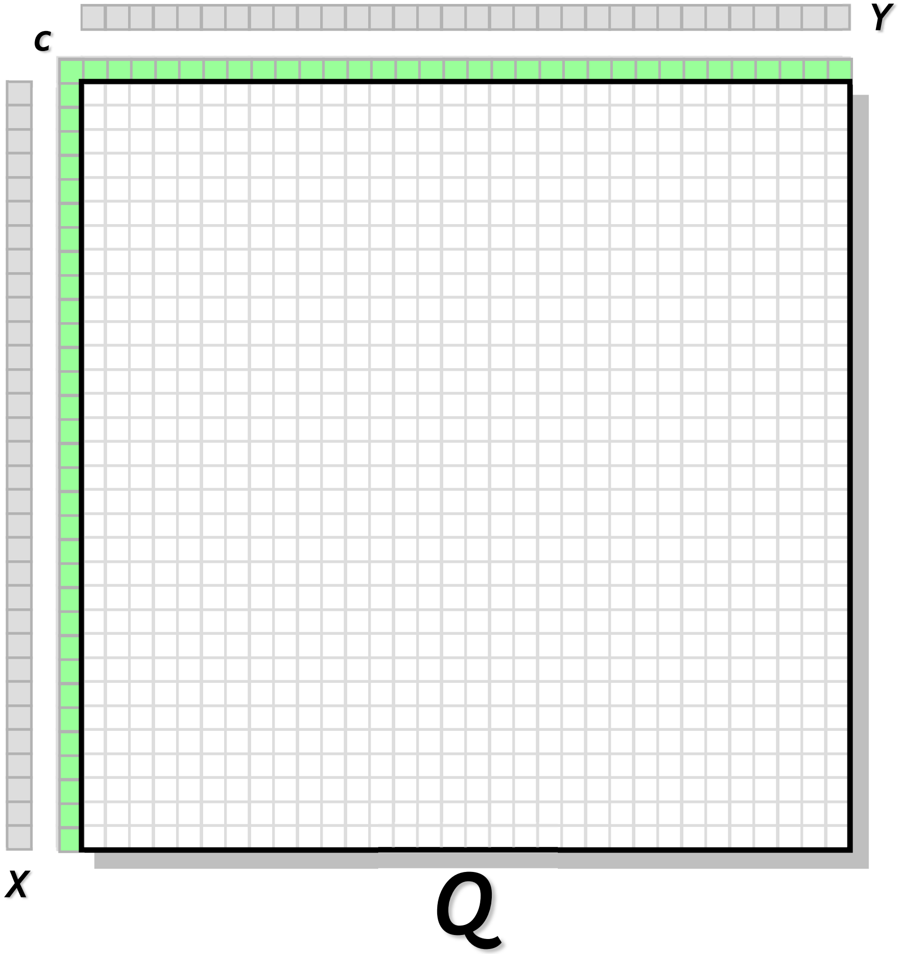
🟩 stored values

Parallel Cache-efficient Traceback Path

(PAR-TRACEBACK)

$Q \equiv c[1 \ldots n, 1 \ldots n]$

$n > pC$



🟩 stored values

🟥 traceback path

Parallel Cache-efficient Traceback Path

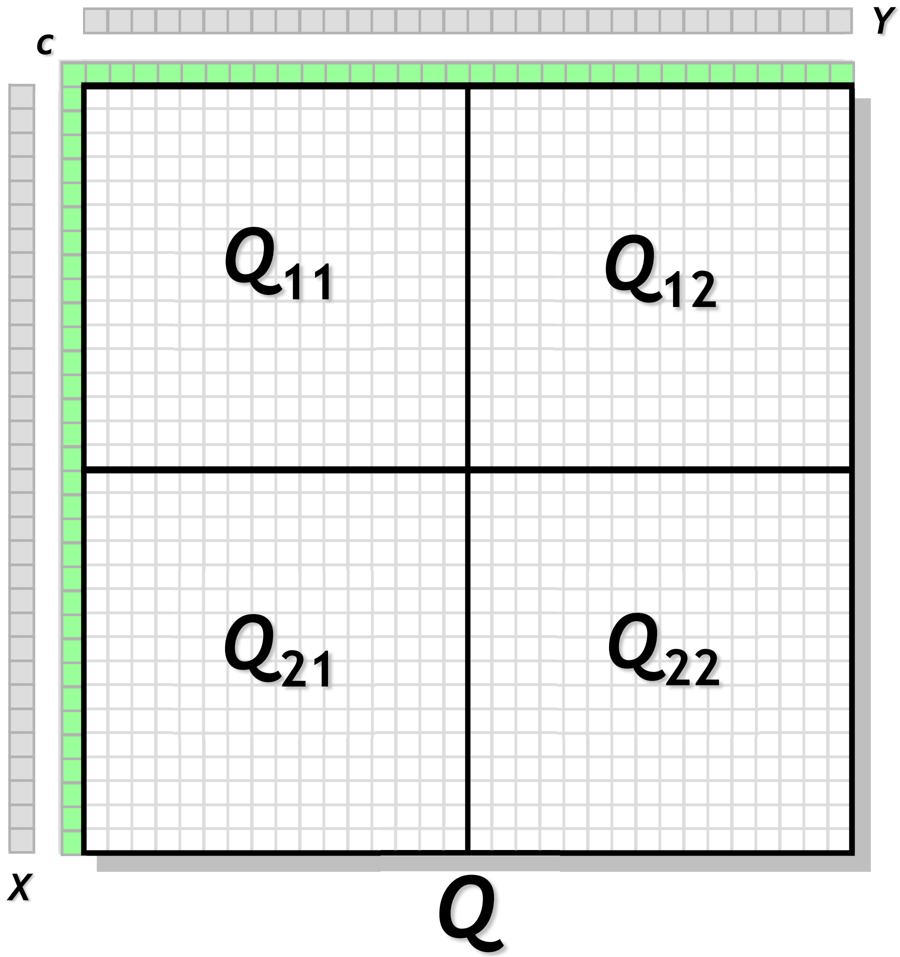
(Par-Traceback)

$Q \equiv c[1 \ldots n, 1 \ldots n]$

$\boxed{n > pC}$

1. Decompose Q:

   Split Q into four quadrants.



🟩 stored values

🟥 traceback path

Parallel Cache-efficient Traceback Path
(Par-Traceback)

$Q \equiv c[1 \ldots n, 1 \ldots n]$

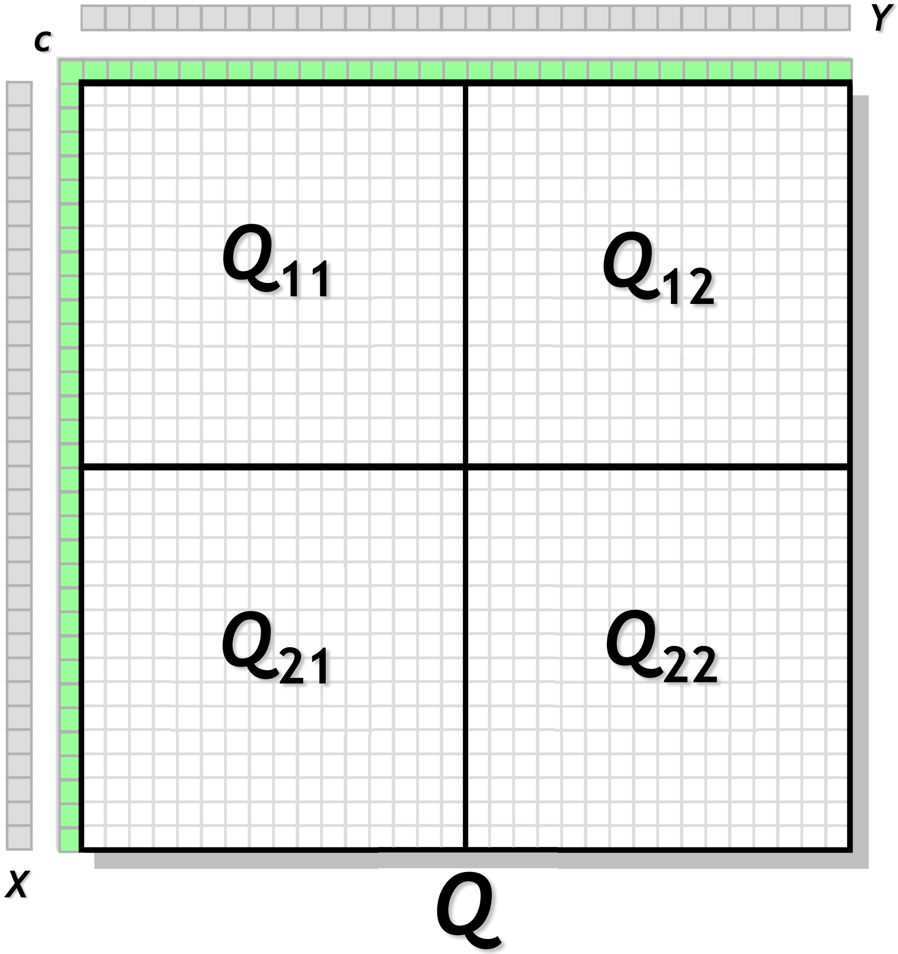
$\boxed{n > pC}$

1. **Decompose Q:**

   Split Q into four quadrants.

2. **Forward Pass (Generate Boundaries):**

   Generate the right and the bottom boundaries of all quadrants by calling Par-Boundary (using all p processors).



🟩 stored values

🟥 traceback path

Parallel Cache-efficient Traceback Path

(Par-Traceback)

$Q \equiv c[1 \ldots n, 1 \ldots n]$

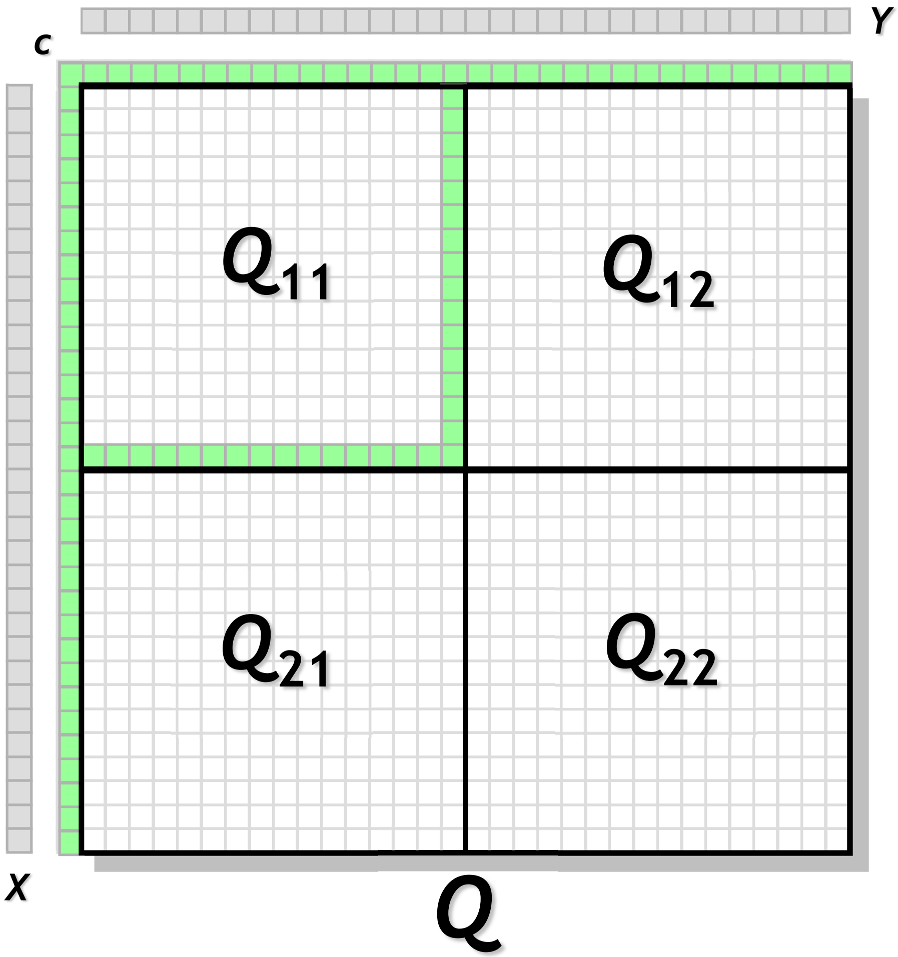
$\boxed{n > pC}$

1. **Decompose Q:**

   Split Q into four quadrants.

2. **Forward Pass (Generate Boundaries):**

   Generate the right and the bottom boundaries of all quadrants by calling Par-Boundary (using all p processors).



🟩 stored values
🟥 traceback path

Parallel Cache-efficient Traceback Path
(PAR-TRACEBACK)

$Q \equiv c[1 \ldots n, 1 \ldots n]$

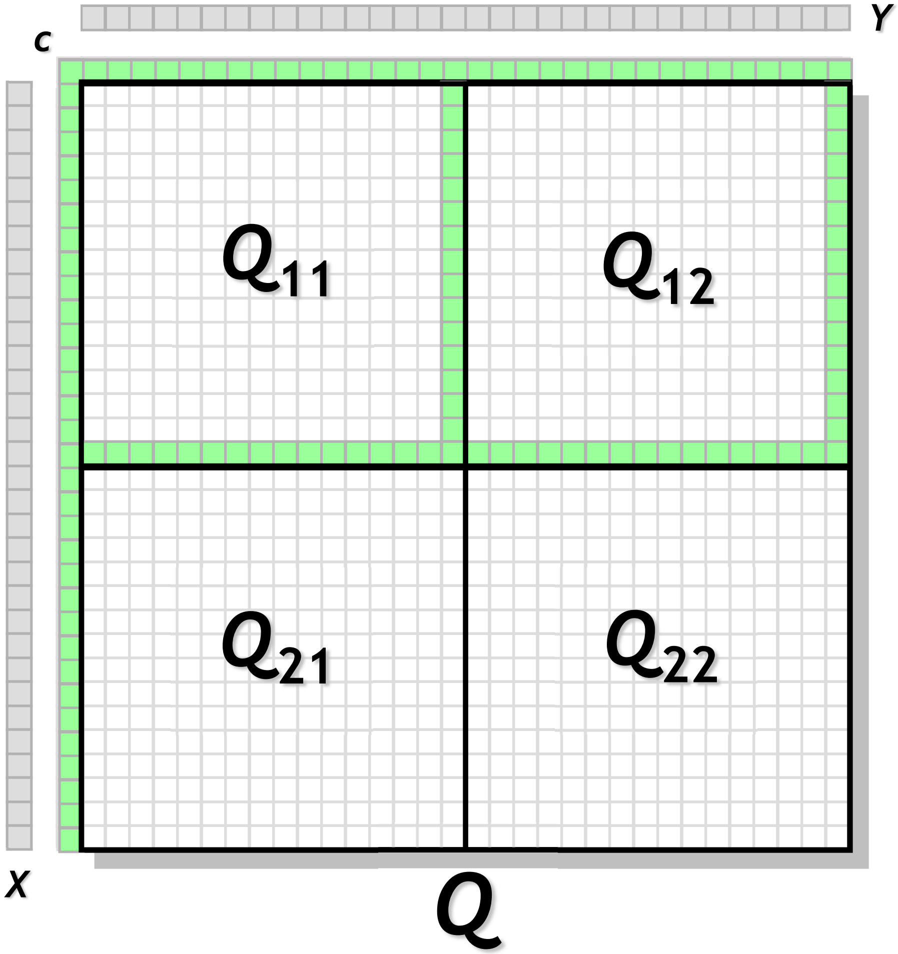
$n > pC$

1. Decompose Q:

   Split Q into four quadrants.

2. Forward Pass (Generate Boundaries):

   Generate the right and the bottom boundaries of all quadrants by calling PAR-BOUNDARY (using all p processors).



🟩 stored values

🟥 traceback path

Parallel Cache-efficient Traceback Path
(Par-Traceback)

$Q \equiv c[1 \ldots n, 1 \ldots n]$

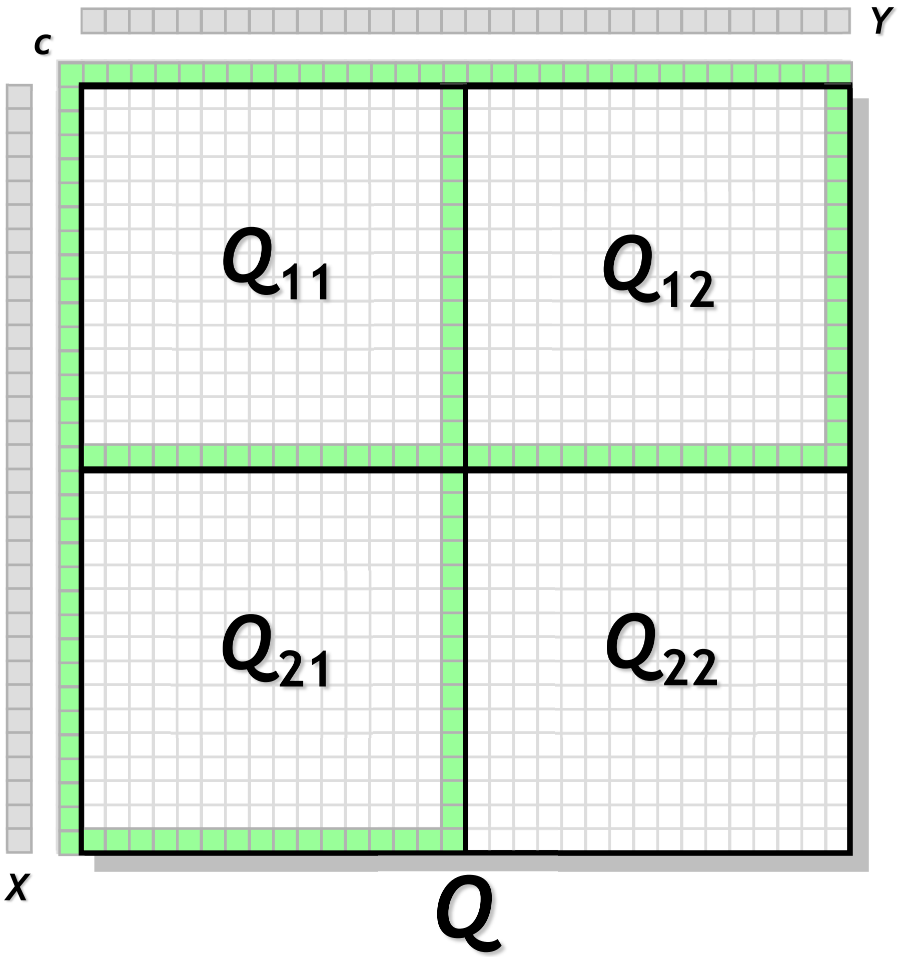
$n > pC$

1. **Decompose Q:**

   Split Q into four quadrants.

2. **Forward Pass (Generate Boundaries):**

   Generate the right and the bottom boundaries of all quadrants by calling Par-Boundary (using all p processors).



🟩 stored values

🟥 traceback path

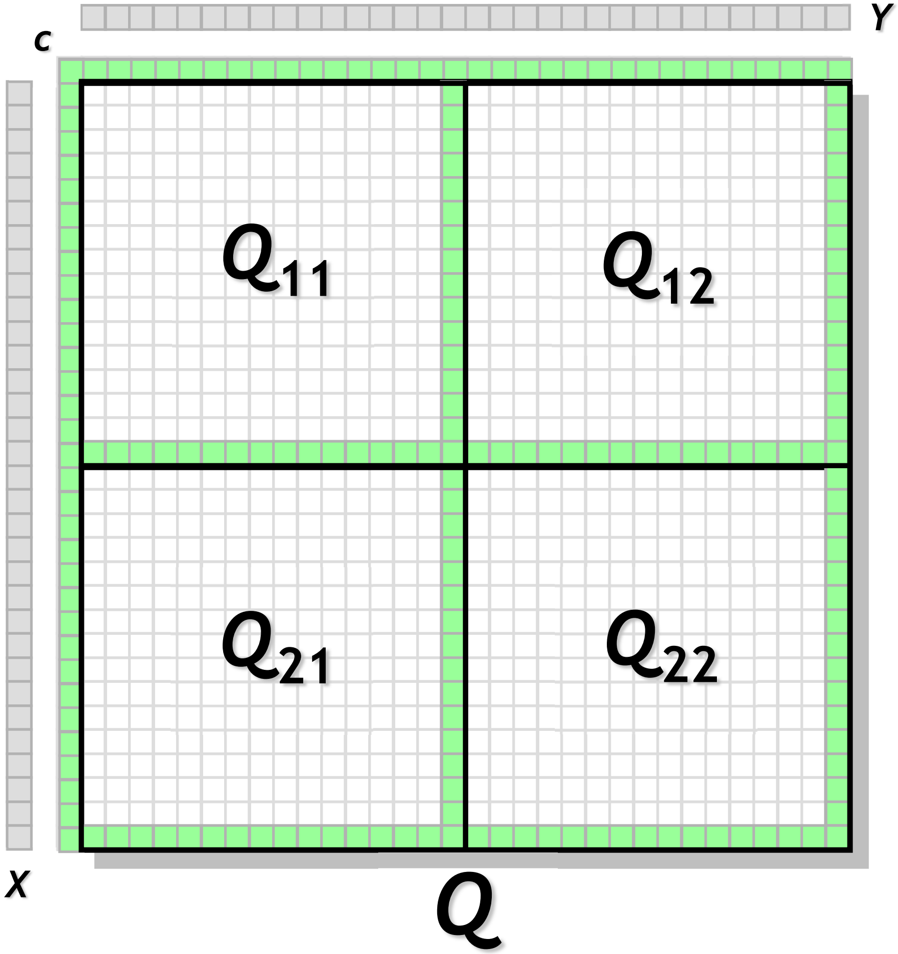
Parallel Cache-efficient Traceback Path
(Par-Traceback)

$Q \equiv c[1 \ldots n, 1 \ldots n]$

$n > pC$

1. Decompose Q:
   Split Q into four quadrants.

2. Forward Pass (Generate Boundaries):
   Generate the right and the bottom boundaries of all quadrants by calling Par-Boundary (using all p processors).



🟩 stored values
🟥 traceback path

Parallel Cache-efficient Traceback Path

(PAR-TRACEBACK)

$Q \equiv c[1 \ldots n, 1 \ldots n]$

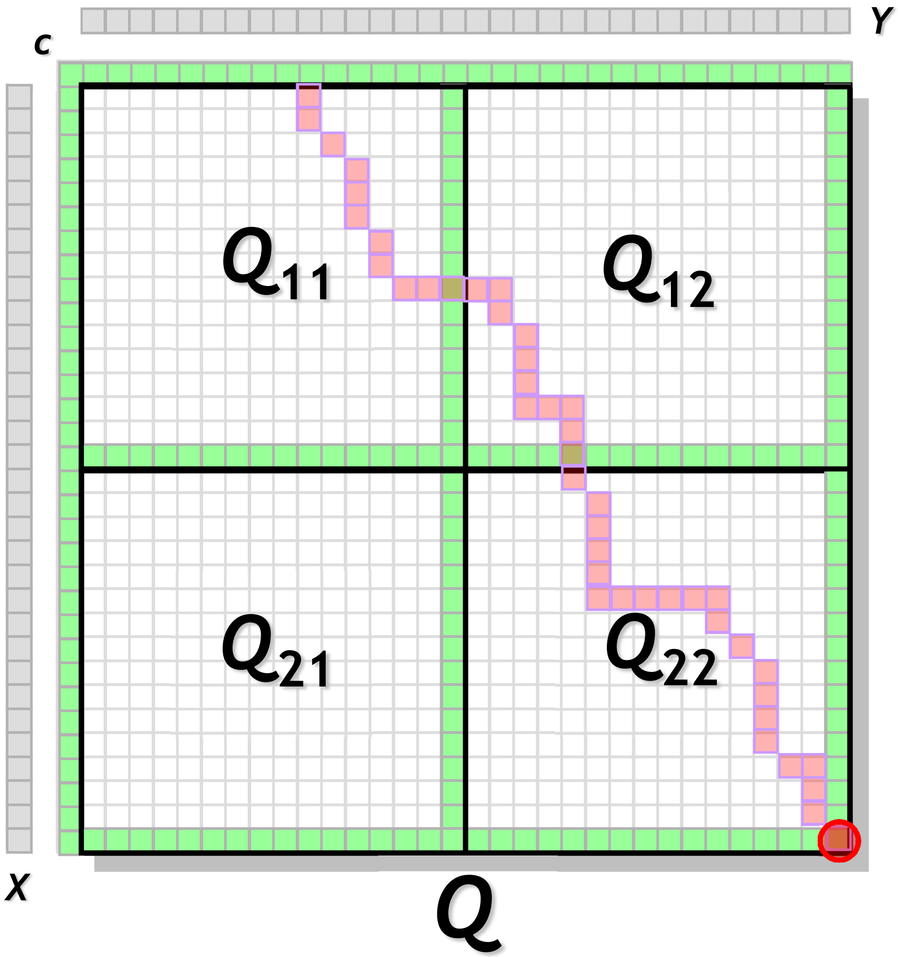
$n > pC$

1. **Decompose Q:**

   Split Q into four quadrants.

2. **Forward Pass (Generate Boundaries):**

   Generate the right and the bottom boundaries of all quadrants by calling PAR-BOUNDARY (using all p processors).



🟩 stored values
🟥 traceback path

Parallel Cache-efficient Traceback Path

(Par-Traceback)

$Q \equiv c[1 \ldots n, 1 \ldots n]$

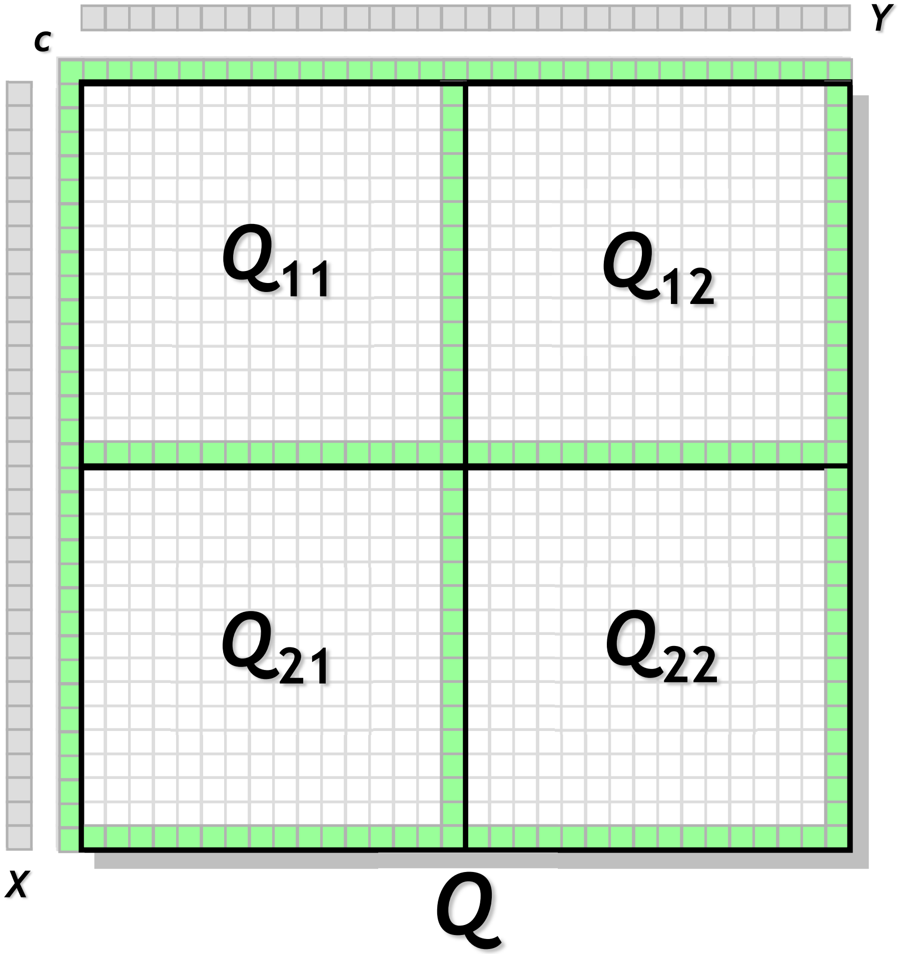
$n > pC$

1. Decompose Q:

   Split Q into four quadrants.

2. Forward Pass (Generate Boundaries):

   Generate the right and the bottom boundaries of all quadrants by calling Par-Boundary (using all p processors).



🟩 stored values

🟥 traceback path

Parallel Cache-efficient Traceback Path

(Par-Traceback)

1. Decompose Q:

   Split Q into four quadrants.

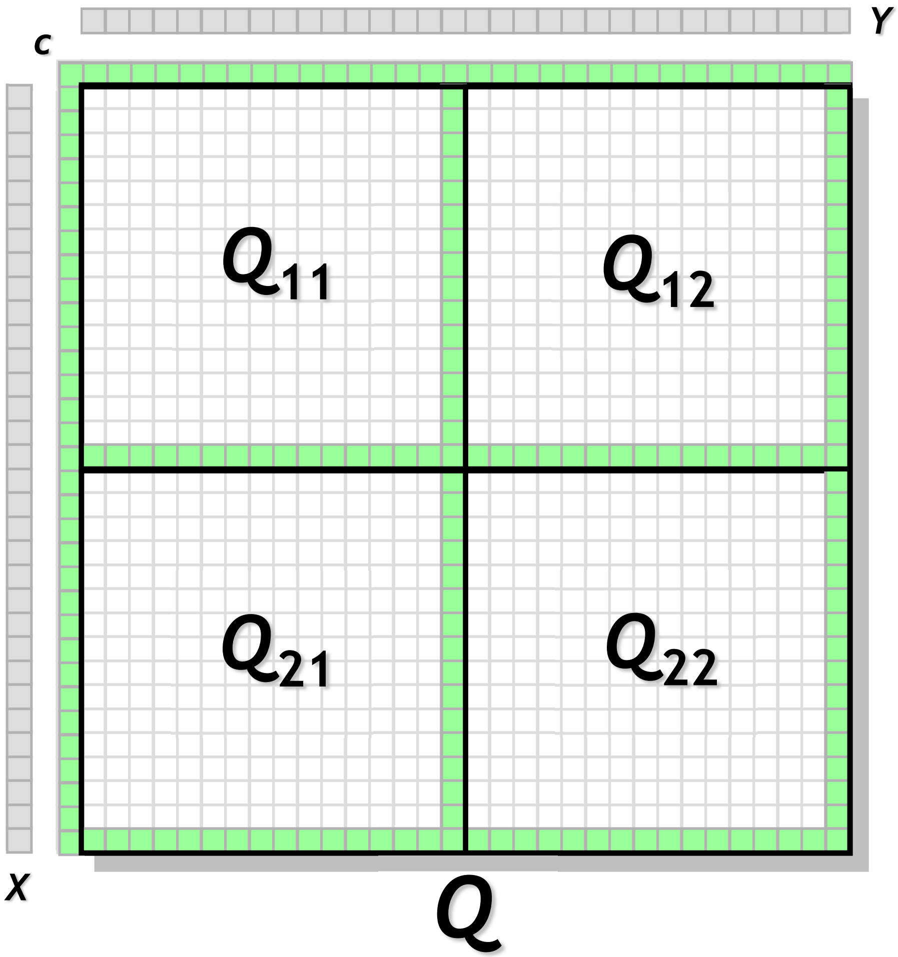
2. Forward Pass (Generate Boundaries):

   Generate the right and the bottom boundaries of all quadrants by calling Par-Boundary (using all p processors).

3. Backward Pass (Extract Traceback Path Fragments):

$Q \equiv c[1 \dots n, 1 \dots n]$

$n > pC$



🟩 stored values

🟥 traceback path

Parallel Cache-efficient Traceback Path

(Par-Traceback)

$Q \equiv c[1 \ldots n, 1 \ldots n]$

$n > pC$

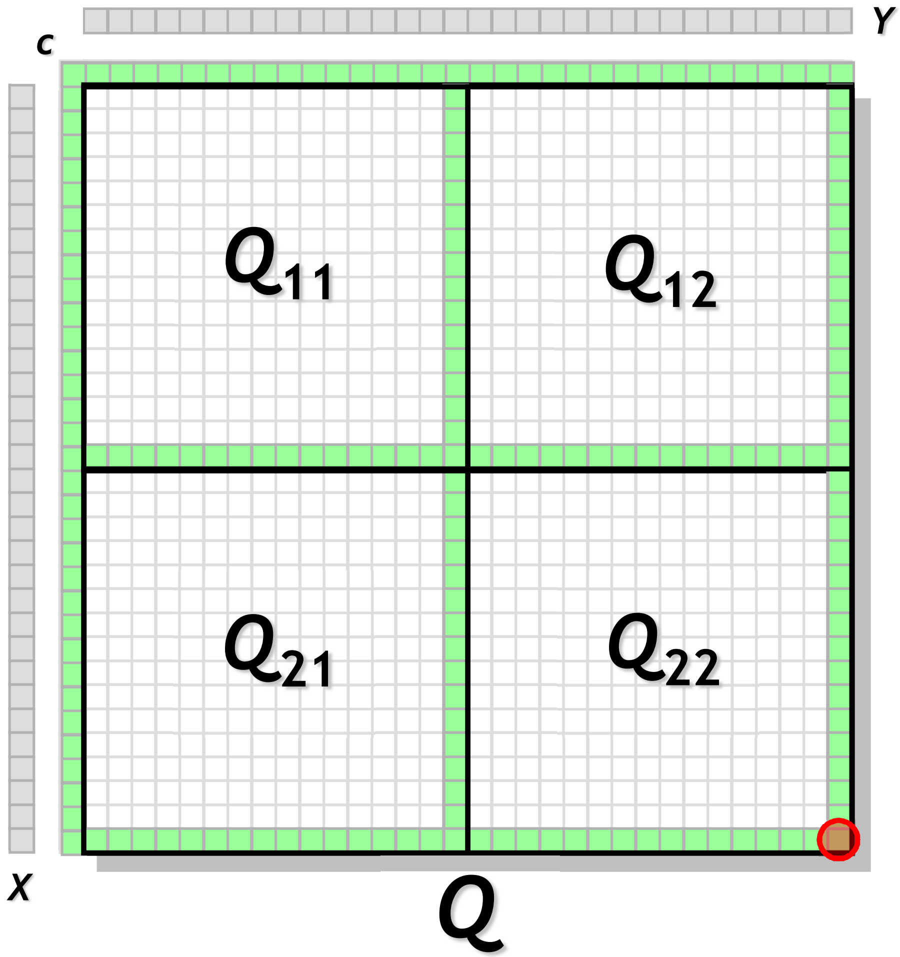
1. Decompose Q:

   Split Q into four quadrants.

2. Forward Pass (Generate Boundaries):

   Generate the right and the bottom boundaries of all quadrants by calling Par-Boundary (using all p processors).

3. Backward Pass (Extract Traceback Path Fragments):



🟩 stored values

🟥 traceback path

Parallel Cache-efficient Traceback Path

(Par-Traceback)

$Q \equiv c[1 \ldots n, 1 \ldots n]$

$n > pC$

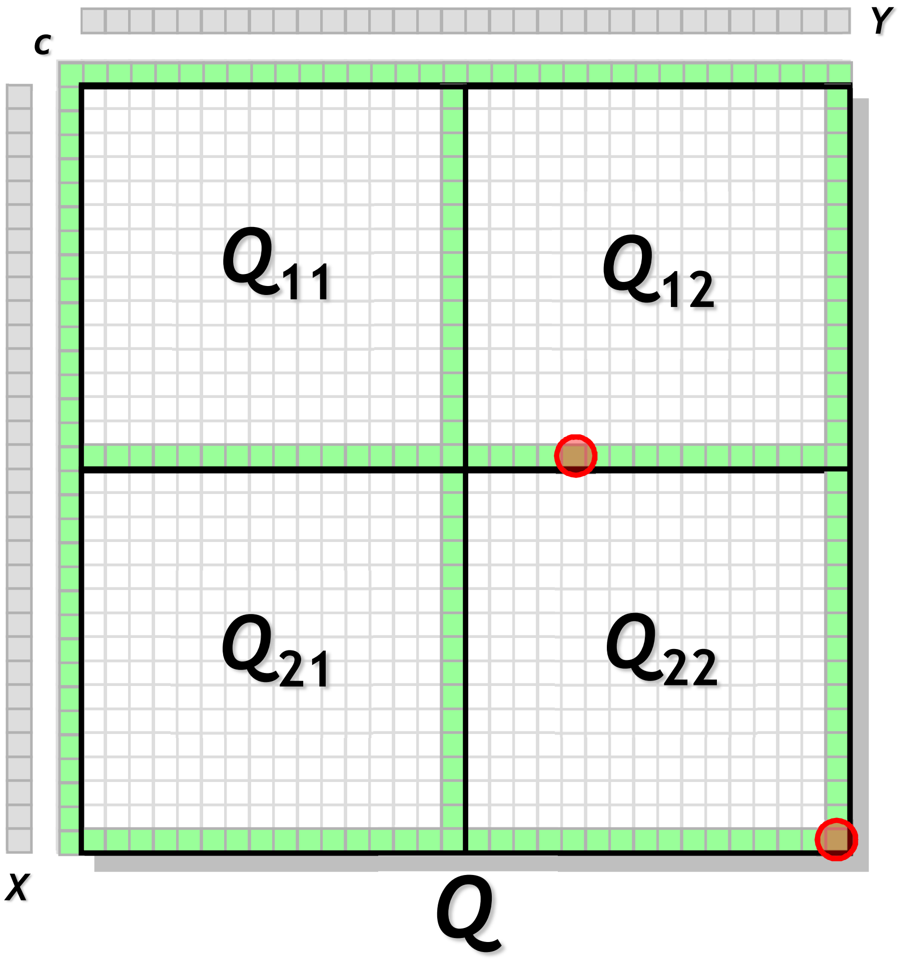
1. **Decompose Q:**

   Split Q into four quadrants.

2. **Forward Pass (Generate Boundaries):**

   Generate the right and the bottom boundaries of all quadrants by calling Par-Boundary (using all p processors).

3. **Backward Pass (Extract Traceback Path Fragments):**



🟩 stored values

🟥 traceback path

Parallel Cache-efficient Traceback Path

(Par-Traceback)

$Q \equiv c[1 \ldots n, 1 \ldots n]$

$\boxed{n > pC}$

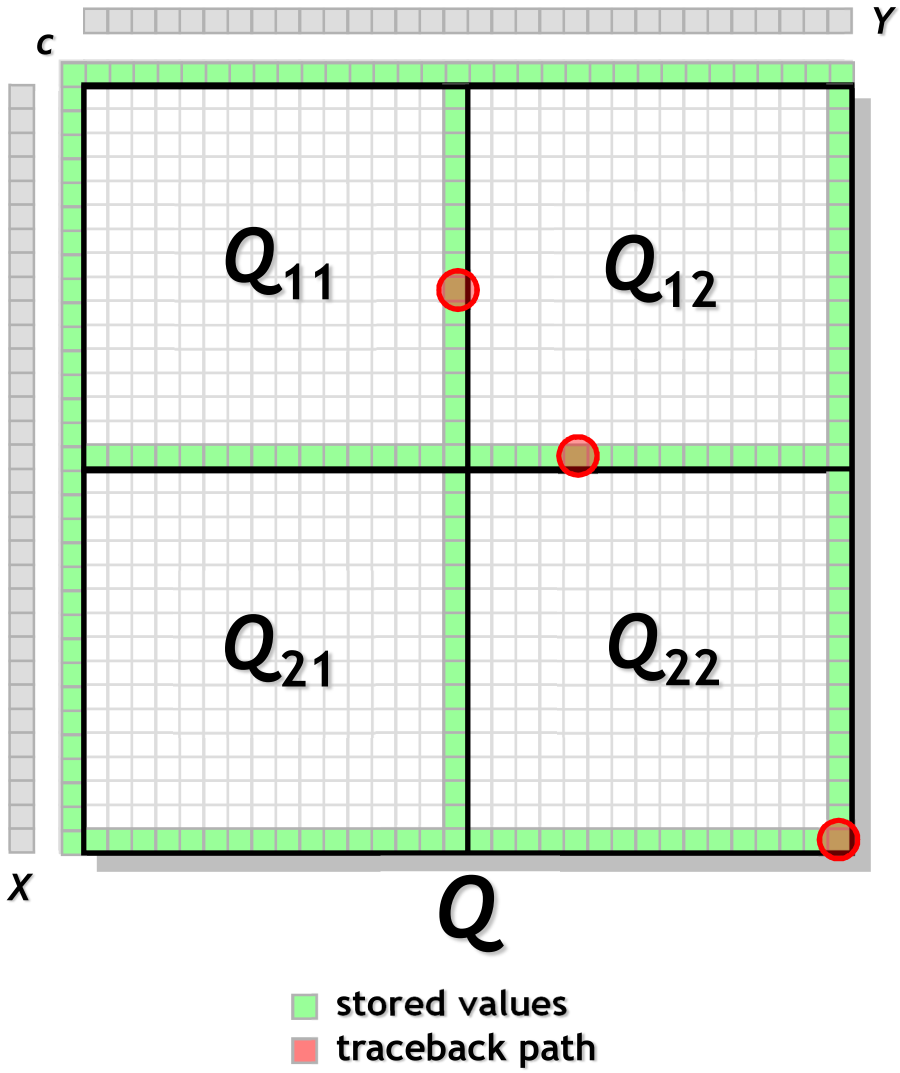
1. **Decompose Q:**

   Split Q into four quadrants.

2. **Forward Pass (Generate Boundaries):**

   Generate the right and the bottom boundaries of all quadrants by calling Par-Boundary (using all p processors).

3. **Backward Pass (Extract Traceback Path Fragments):**



Parallel Cache-efficient Traceback Path
(Par-Traceback)

$Q \equiv c[1 \ldots n, 1 \ldots n]$

$n > pC$

1. **Decompose Q:**

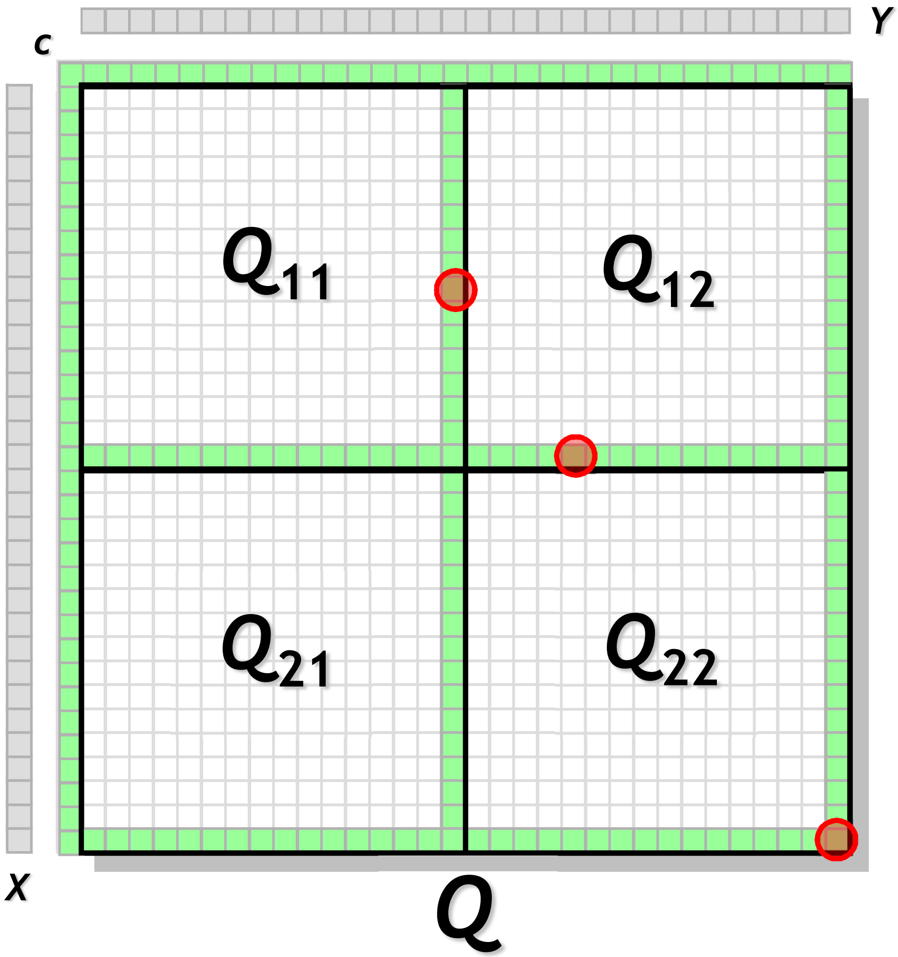
   Split Q into four quadrants.

2. **Forward Pass (Generate Boundaries):**

   Generate the right and the bottom boundaries of all quadrants by calling Par-Boundary (using all p processors).

3. **Backward Pass (Extract Traceback Path Fragments):**

   Extract path fragments from Q_{22}, Q_{12} and Q_{11} in parallel by calling Par-Traceback with p / 3 processors each.



🟩 stored values
🟥 traceback path

Parallel Cache-efficient Traceback Path
(Par-Traceback)

$Q \equiv c[1 \dots n, 1 \dots n]$

$\boxed{n > pC}$

1. **Decompose Q:**

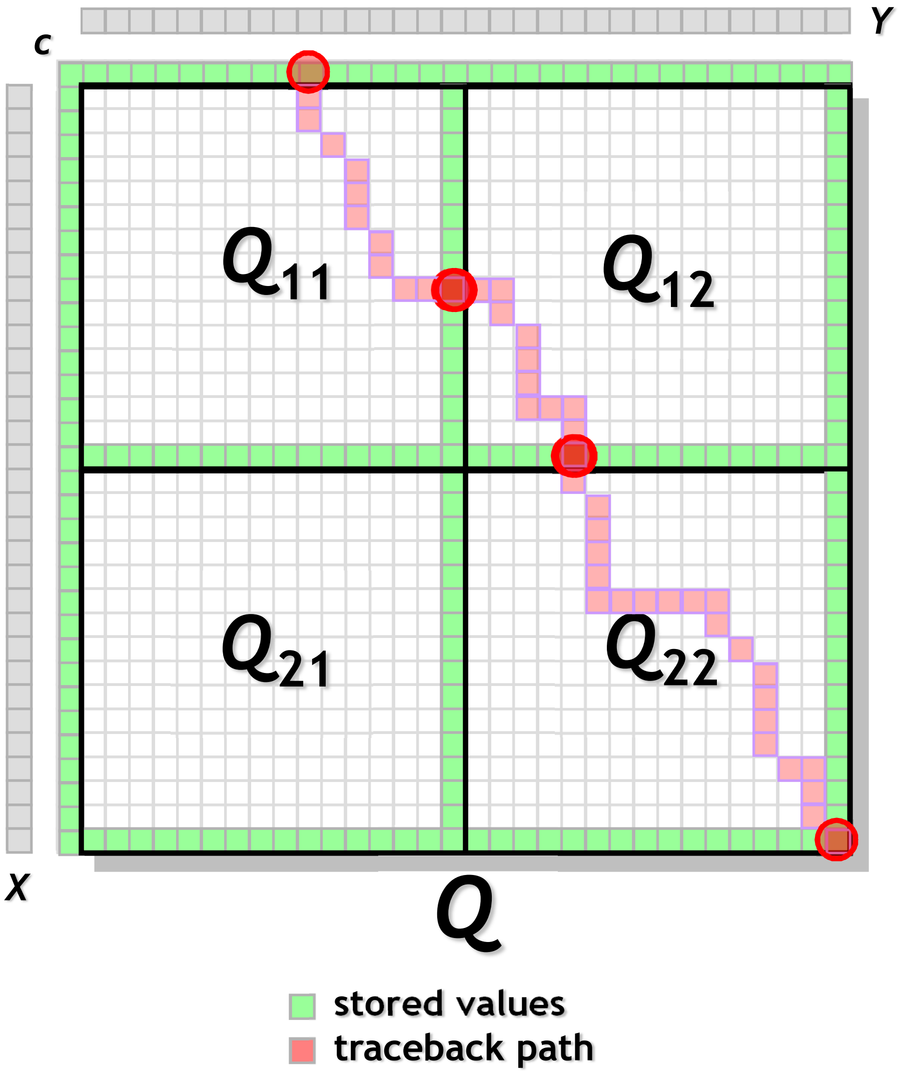
   Split Q into four quadrants.

2. **Forward Pass (Generate Boundaries):**

   Generate the right and the bottom boundaries of all quadrants by calling Par-Boundary (using all p processors).

3. **Backward Pass (Extract Traceback Path Fragments):**

   Extract path fragments from Q_{22}, Q_{12} and Q_{11} in parallel by calling Par-Traceback with p / 3 processors each.



Parallel Cache-efficient Traceback Path
(Par-Traceback)

$Q \equiv c[1 \ldots n, 1 \ldots n]$

$n > pC$

1. **Decompose Q:**

   Split Q into four quadrants.

2. **Forward Pass (Generate Boundaries):**

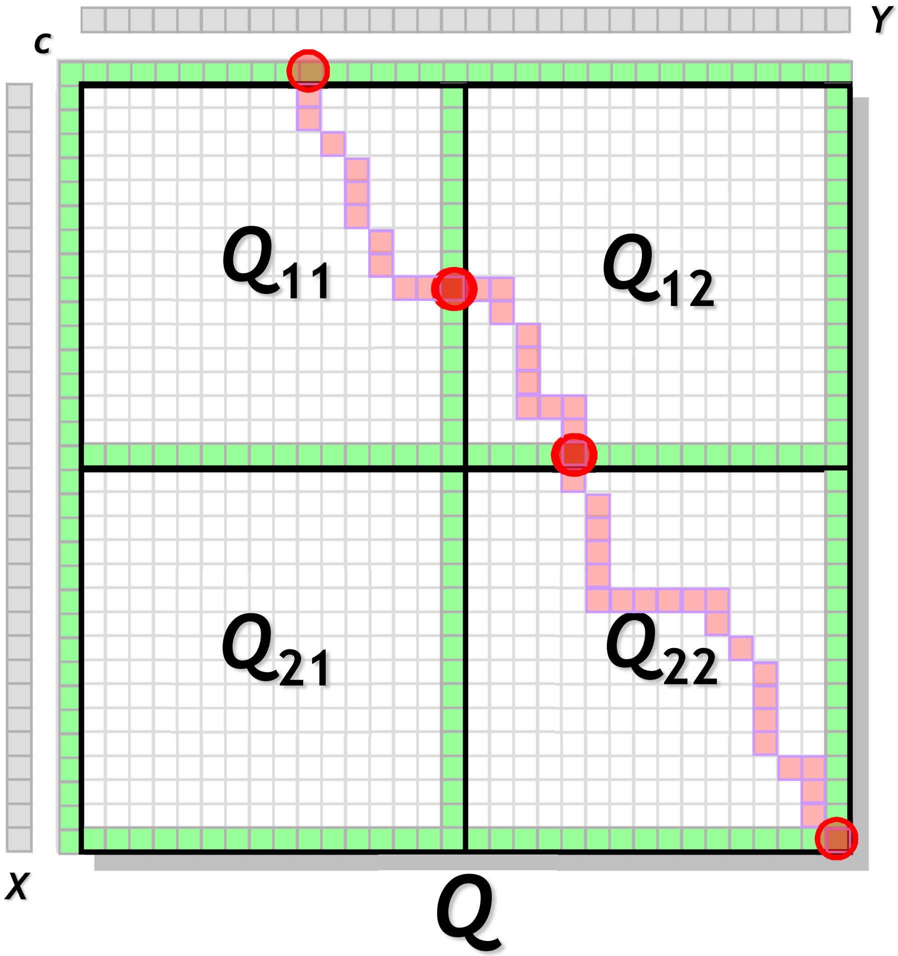
   Generate the right and the bottom boundaries of all quadrants by calling Par-Boundary (using all p processors).

3. **Backward Pass (Extract Traceback Path Fragments):**

   Extract path fragments from Q_{22}, Q_{12} and Q_{11} in parallel by calling Par-Traceback with p / 3 processors each.

4. **Compose Traceback Path:**

   Combine the path fragments.



🟩 stored values
🟥 traceback path

Performance Bounds
(PAR-TRACEBACK)

$Q \equiv c[1 \ldots n,\ 1 \ldots n]$

$n > pC$

Parallel Time Complexity:

$$T_p(n) = 4T'_p\left(\frac{n}{2}\right) + T_{\frac{p}{3}}\left(\frac{n}{2}\right) + \mathrm{O}\left(\frac{n}{p}\right)$$

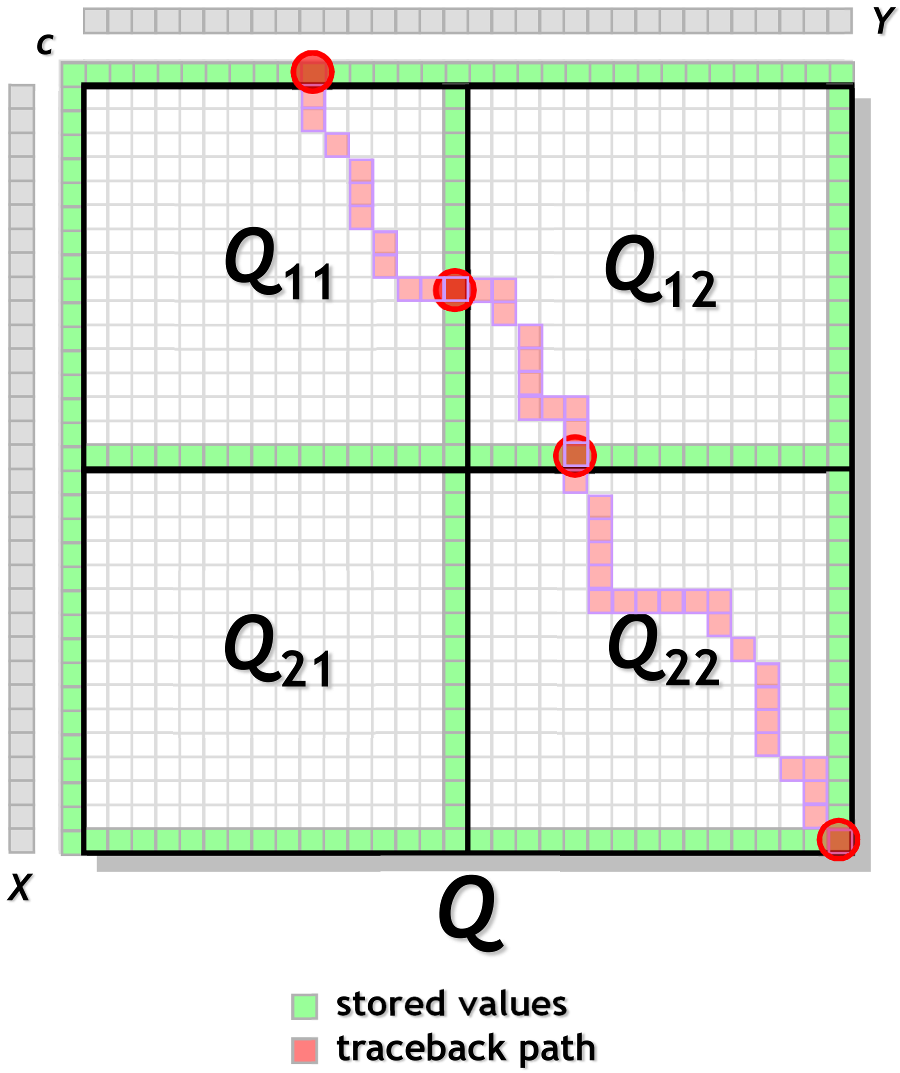
$$= \mathrm{O}\left(\frac{n^2}{p} + n\right)$$

Cache Complexity:

$$Q_p(n) = 4Q'_p\left(\frac{n}{2}\right) + 3Q_{\frac{p}{3}}\left(\frac{n}{2}\right) + \mathrm{O}\left(1 + \frac{n}{B}\right)$$

$$= \mathrm{O}\left(\frac{n^2}{BC} + p \cdot \frac{n}{B} + p^2\right)$$

$$= \mathrm{O}\left(\frac{n^2}{BC}\right) \qquad [since\ n \geq pC]$$



DP with Local Dependencies
Generalization of the LCS Result

Problem	Time	Space	Cache-complexity		Parallel Time
Longest Common Subsequence	$\Theta\left(n^2\right)$	$\Theta\left(n\right)$	$O\left(\frac{n^2}{BC}\right)$	Classic $O\left(\frac{n^2}{B}\right)$	$O\left(\frac{n^2}{p}+n\right)$
Pairwise Sequence Alignment (affine gap costs)				Gotoh, 1982 $O\left(\frac{n^2}{B}\right)$	
Median of three Sequences (affine gap costs)	$\Theta\left(n^3\right)$	$\Theta\left(n^2\right)$	$O\left(\frac{n^3}{B\sqrt{C}}\right)$	Knudsen, 2003 $O\left(\frac{n^3}{B}\right)$	$O\left(\frac{n^3}{p}+n\right)$
RNA Secondary Structure Prediction with Simple Pseudoknots	$\Theta\left(n^4\right)$		$O\left(\frac{n^4}{B\sqrt{C}}\right)$	Akutsu, 2000 $O\left(\frac{n^4}{B}\right)$	$O\left(\frac{n^4}{p}+n\log^2 n\right)$

n = sequence length, C = cache size, B = block transfer size, p = #processors

Performance of Cache-efficient Serial LCS

Algorithms compared:

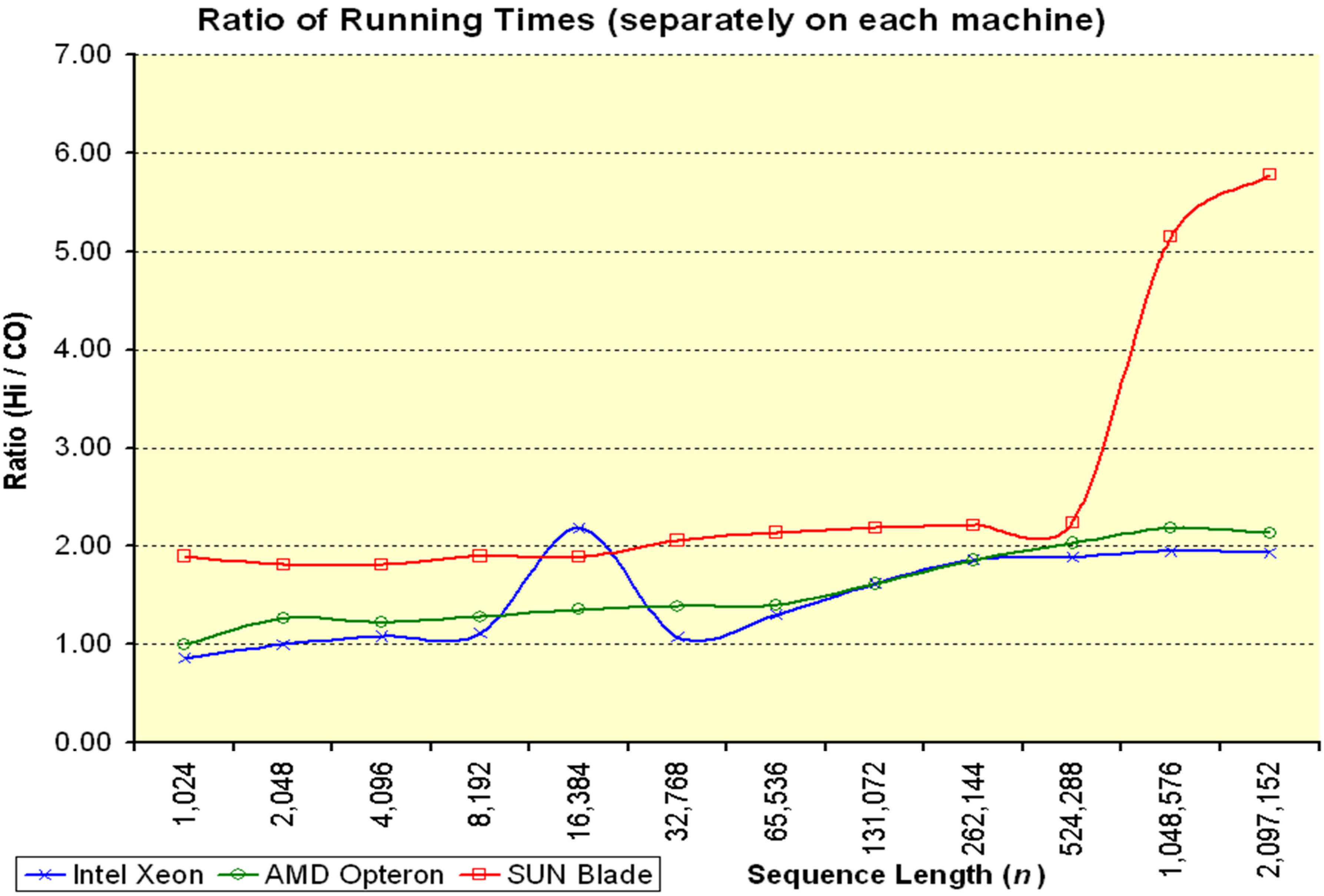
☑ The cache-efficient LCS algorithm (*CO*)

☑ Hirschberg’s linear-space LCS algorithm (*Hi*)

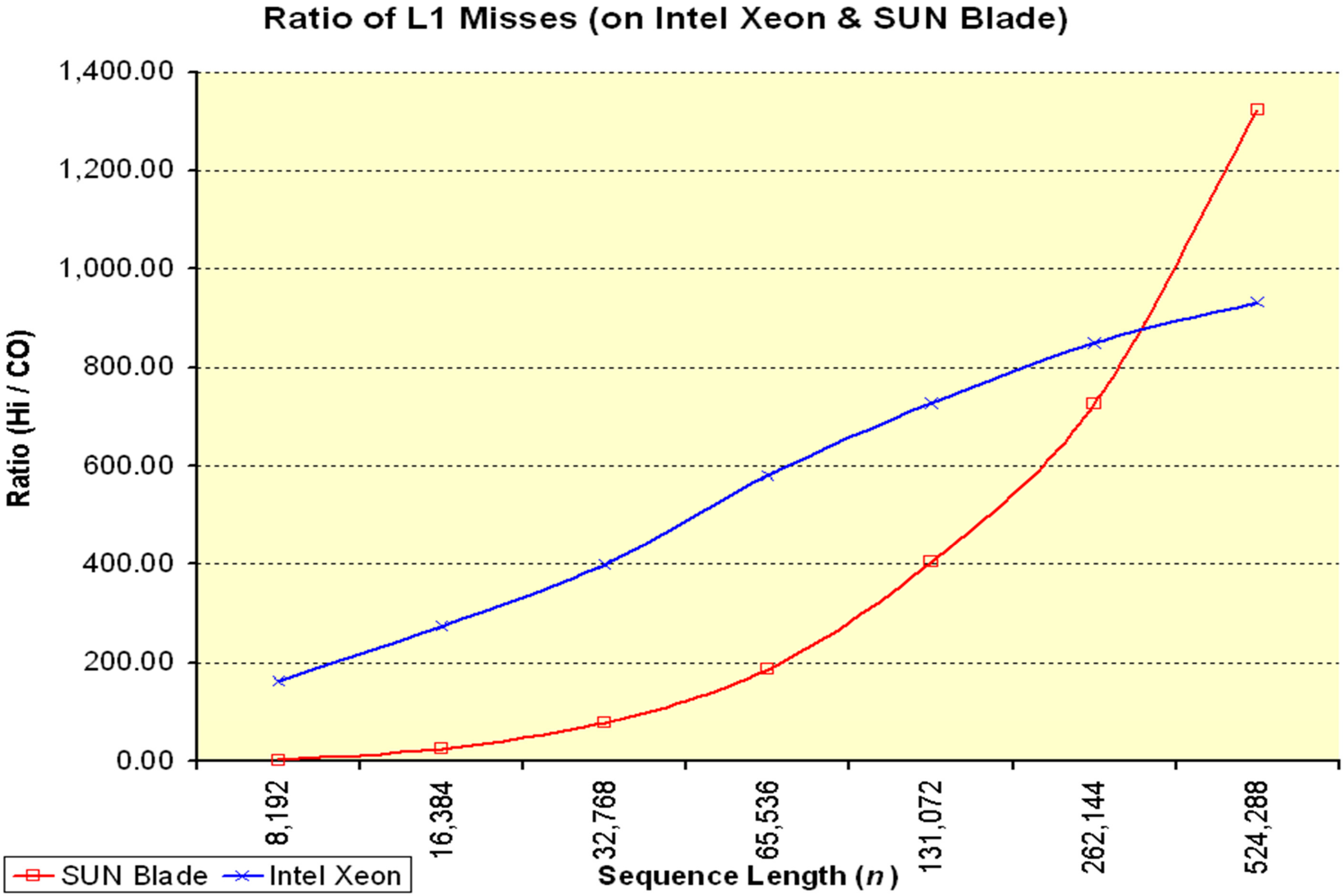
Computing Environment:

Architecture	Processor Speed	L1 Cache (*B*)	L2 Cache (*B*)	RAM
Intel Xeon	3 GHz	8 KB (64 B)	512 KB (64 B)	4 GB
AMD Opteron	2.4 GHz	64 KB (64 B)	1 MB (64 B)	4 GB
SUN Blade	1 GHz	64 KB (32 B)	8 MB (512 B)	1 GB

Ratio of Running Times on Random Sequences (Hirschberg vs the Cache-efficient Algorithm)

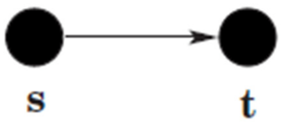


Ratio of L1 Misses on Random Sequences (Hirschberg vs the Cache-efficient Algorithm)

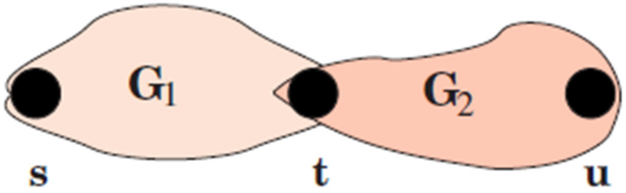


Cache Performance of Divide-and-Conquer Algorithms under the Work-Stealing Scheduler

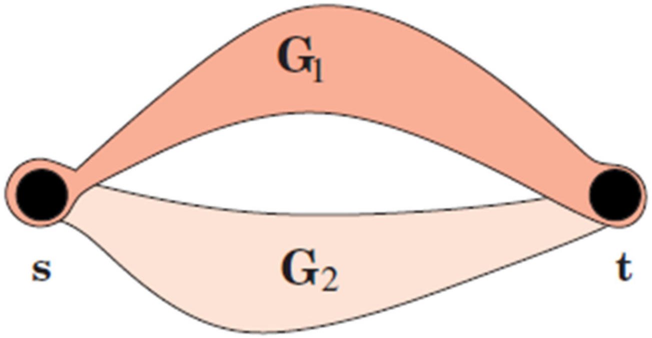
Series-Parallel DAG



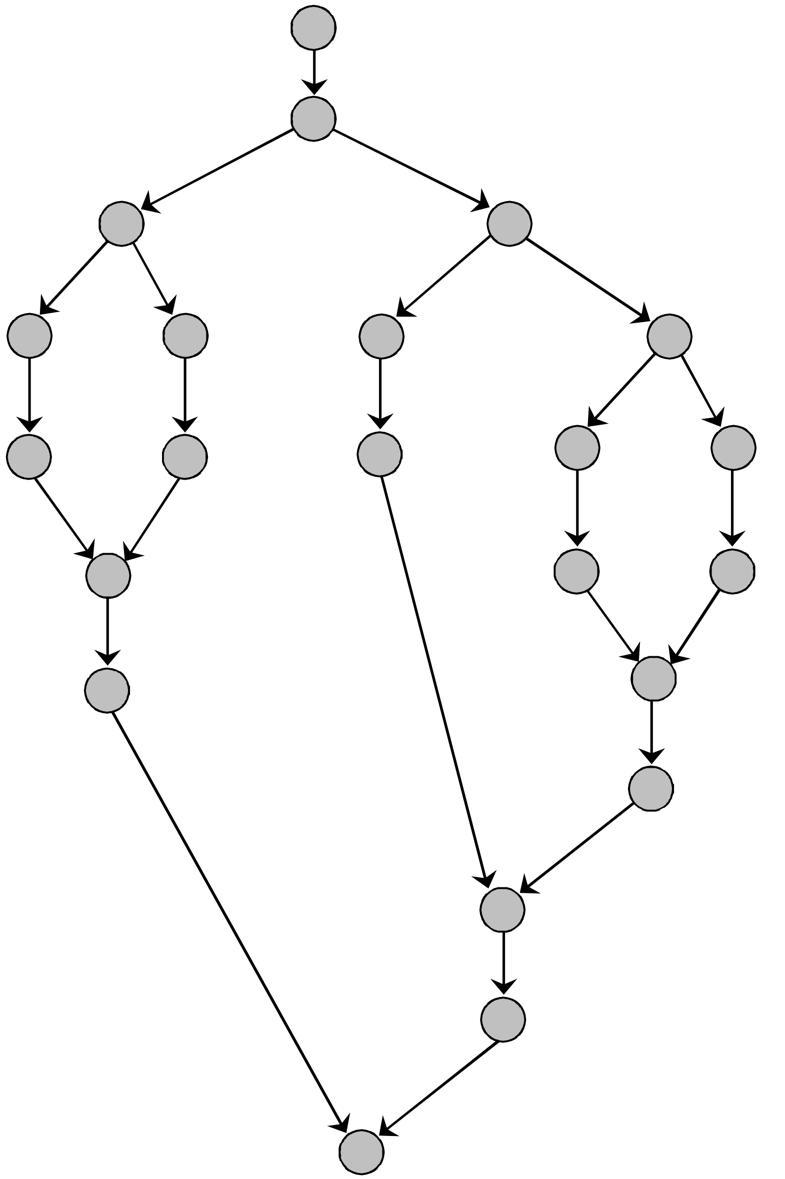
base case



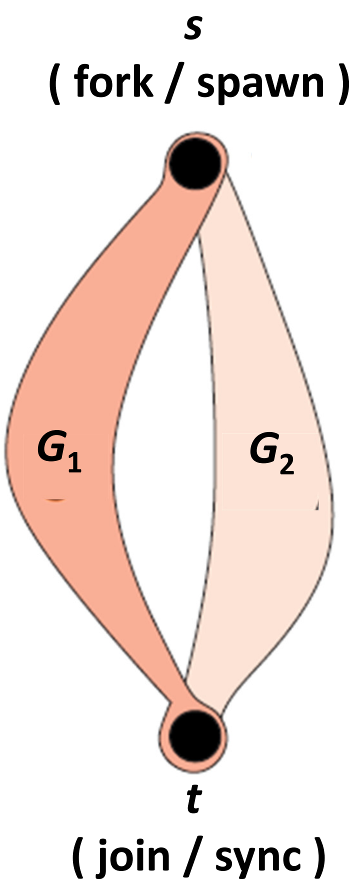
serial composition



parallel composition



Assumptions



Two-way Division (Spawn):

Each division generates only two subtasks.

Serial Execution:

The left (first) subtask generated by a fork node is always executed first.

Parallel Execution:

Only the right (second) subtask generated by a fork node can be stolen.

Drifted Nodes:

In a parallel execution we say that a node is *drifted* when it is executed on a different processing element than its predecessor in the serial execution.

Observations

Observation 1:

Consider two executions of a sequence of instructions *X*. Each execution takes place completely on a single processing element connected to a cache of size *C* and block size *B*. Then the number of cache misses incurred by the two executions can differ by at most *C / B*.

(As under LRU cache replacement policy only the first access to each of the *C* / *B* blocks can cause a cache miss in one execution that is not a miss in the other)

Observation 2:

Each steal can cause at most two nodes to drift: the stolen node and possibly the join node with its sibling.

Implications

If there are S successful steals during parallel execution then there will be at most $2S \times \frac{C}{B}$ additional cache misses compared to the sequential execution.

Now suppose a divide-and-conquer algorithm incurs $Q_1(n)$ cache misses on a serial machine.

Then on a parallel machine with p parallel processing elements each connected to a cache of size C and block size B, the total number of cache misses incurred:

$$Q_p(n) \leq Q_1(n) + \mathrm{O}\left(S \cdot \frac{C}{B}\right)$$

$$= Q_1(n) + \mathrm{O}\left(pT_\infty(n) \cdot \frac{C}{B}\right) \quad [\,w.h.p.\,]$$