Embedded Intelligent System and Novel Computer Architecture

Lecture 04 – Memory Hierarchy and Programming

(Leverage the Spatial and Temporal Locality)

Pengju Ren
Institute of Artificial Intelligence and Robotics
Xi'an Jiaotong University

http://gr.xjtu.edu.cn/web/pengjuren

Review: Accessing Memory

- **Memory latency**
  - The amount of time for a memory request (e.g., load, store) from a processor to be serviced by the memory system
  - Example: 100 cycles, 100 nsec
- **Memory bandwidth**
  - The rate at which the memory system can provide data to a processor
  - Example: 20 GB/s

Memory Stalls

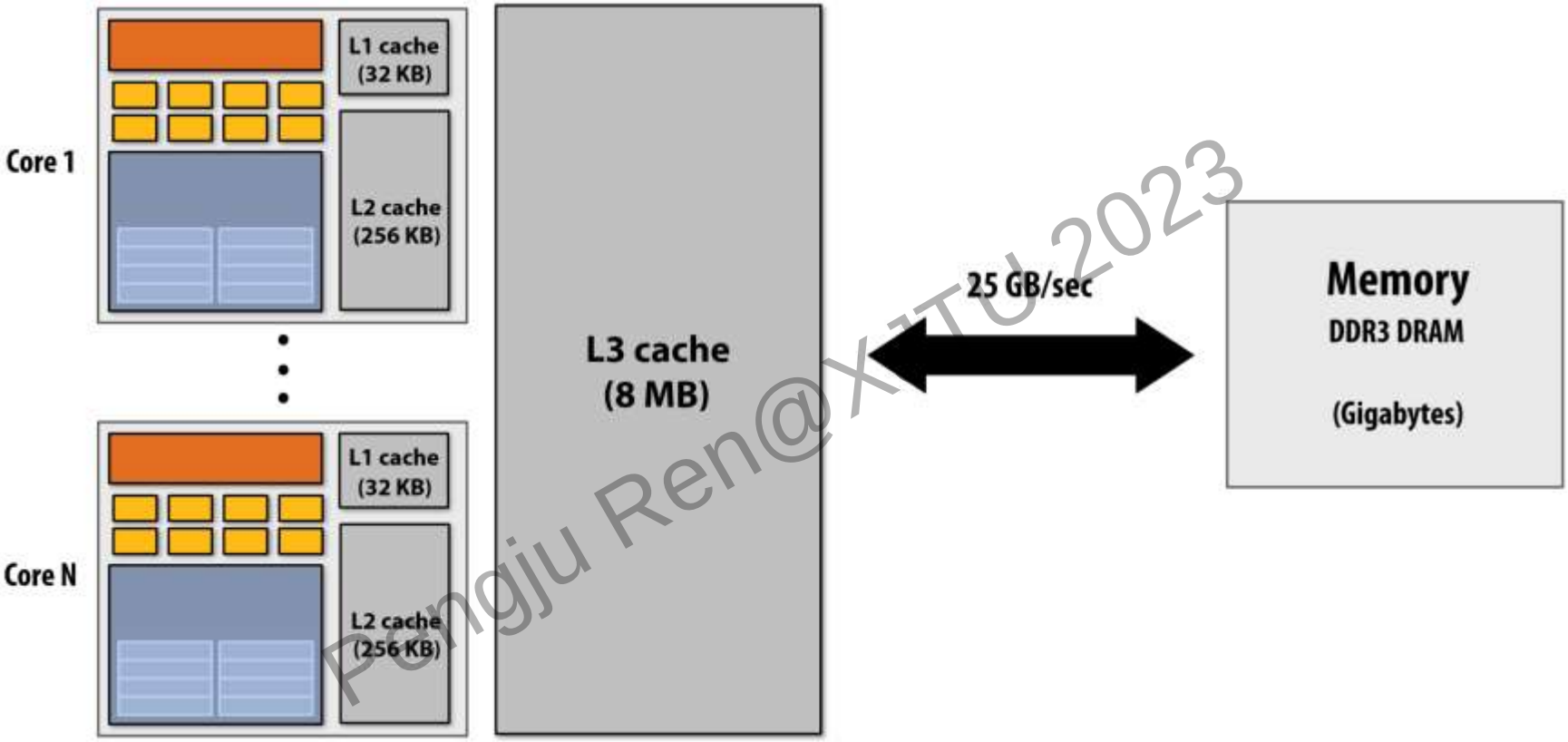
- **A processor "stalls" when it cannot run the next instruction in an instruction stream because of a dependency on a previous instruction.**
- **Accessing memory is a major source of stalls**

```
ld r0 mem[r2]
ld r1 mem[r3]
add r0, r0, r1
```

Dependency: cannot execute 'add' instruction until data at mem[r2] and mem[r3] have been loaded from memory

- **Memory access times ~ 100's of cycles**
  - **Memory "access time" is a measure of latency**

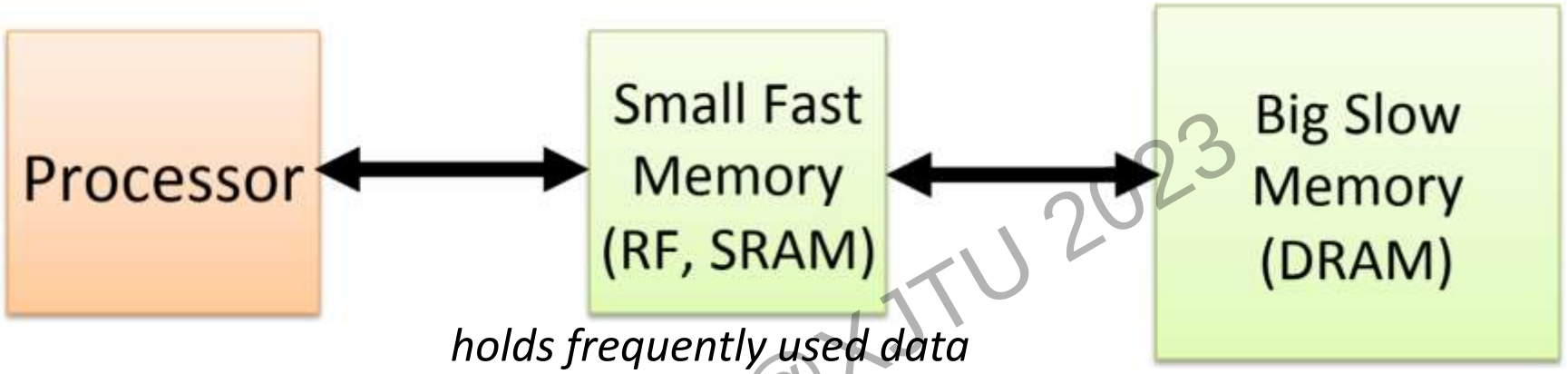
Caches reduce lengths of stalls



Processors run efficiently when data is resident in caches

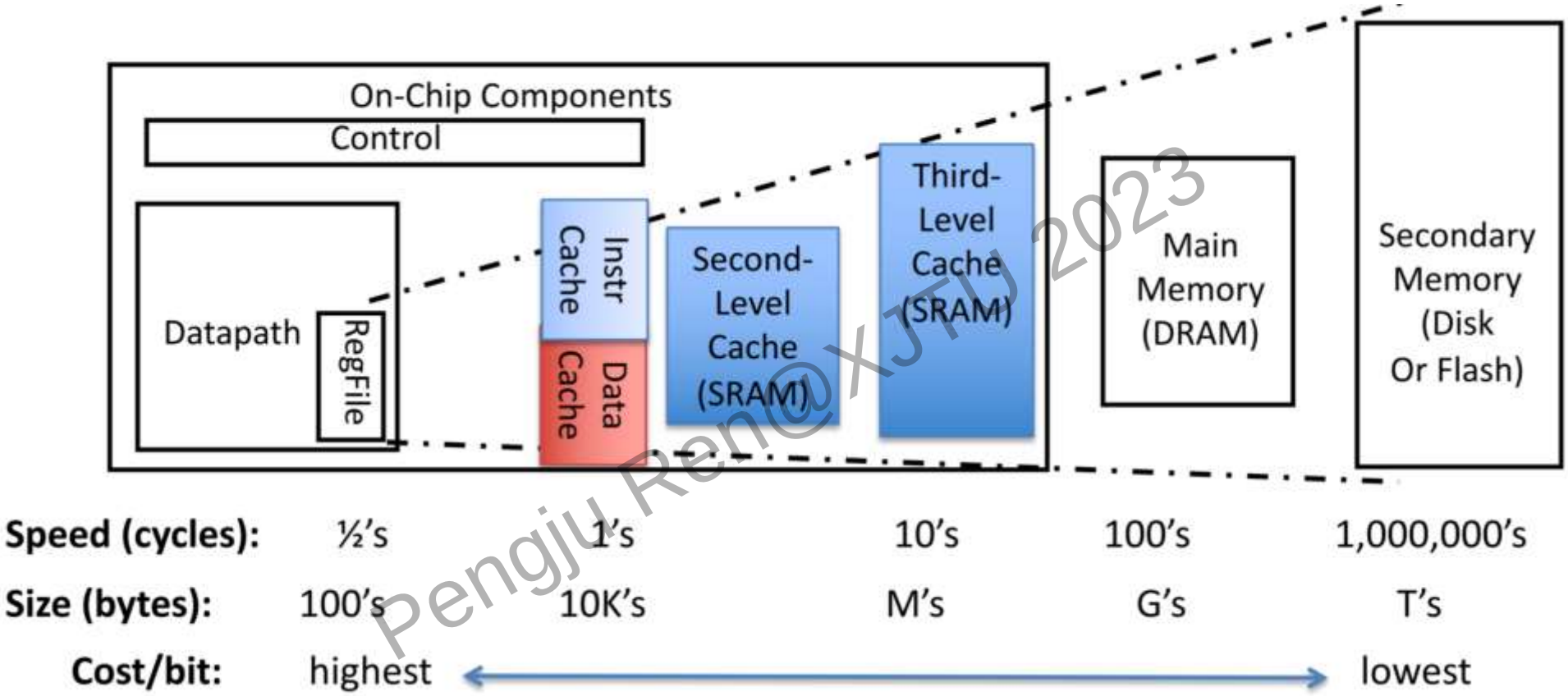
- **Caches reduce memory access latency**
- **Caches also provide high bandwidth data transfer to CPU**

Memory Hierarchy



- **Capacity: Register << SRAM << DRAM**
- **Latency: Register << SRAM << DRAM**
- **Bandwidth: on-chip >> off-chip**
- **On a data access:**
  - *if* data $\in$ fast memory $\Rightarrow$ low latency access *(SRAM)*
  - *if* data $\notin$ fast memory $\Rightarrow$ high latency access *(DRAM)*
- **Memory hierarchies only work if the small, fast memory actually stores data that is reused by the processor**

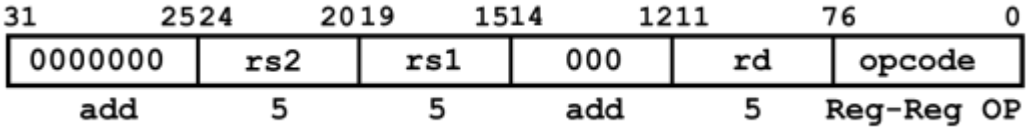
Typical Memory Hierarchy



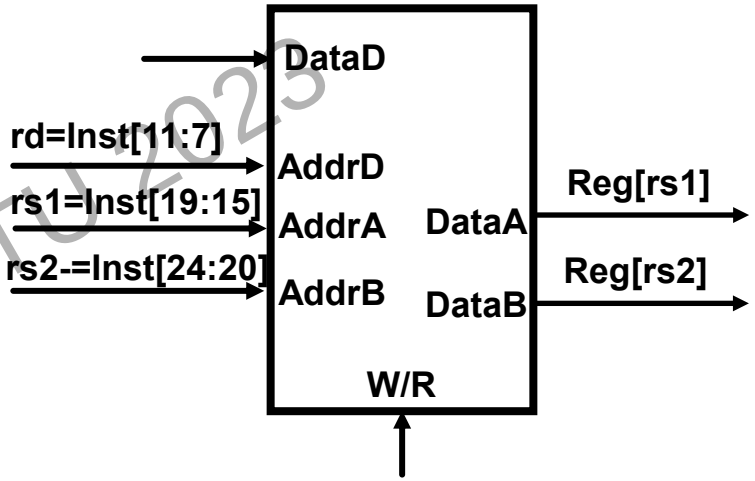
Principle of locality + memory hierarchy presents programmer with
≈ as much memory as is available in the cheapest technology at the
≈ speed offered by the fastest technology

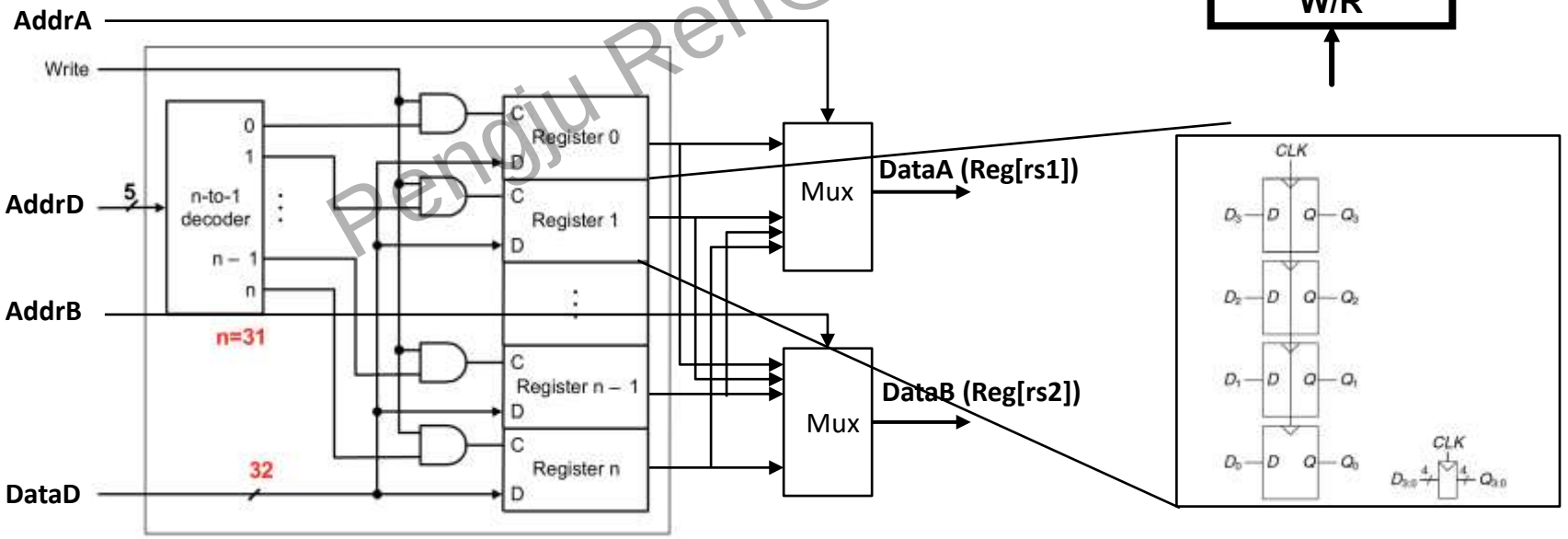
Register File

Taking 32-bits RISC-V add instruction for an example:



With two read ports and one write port

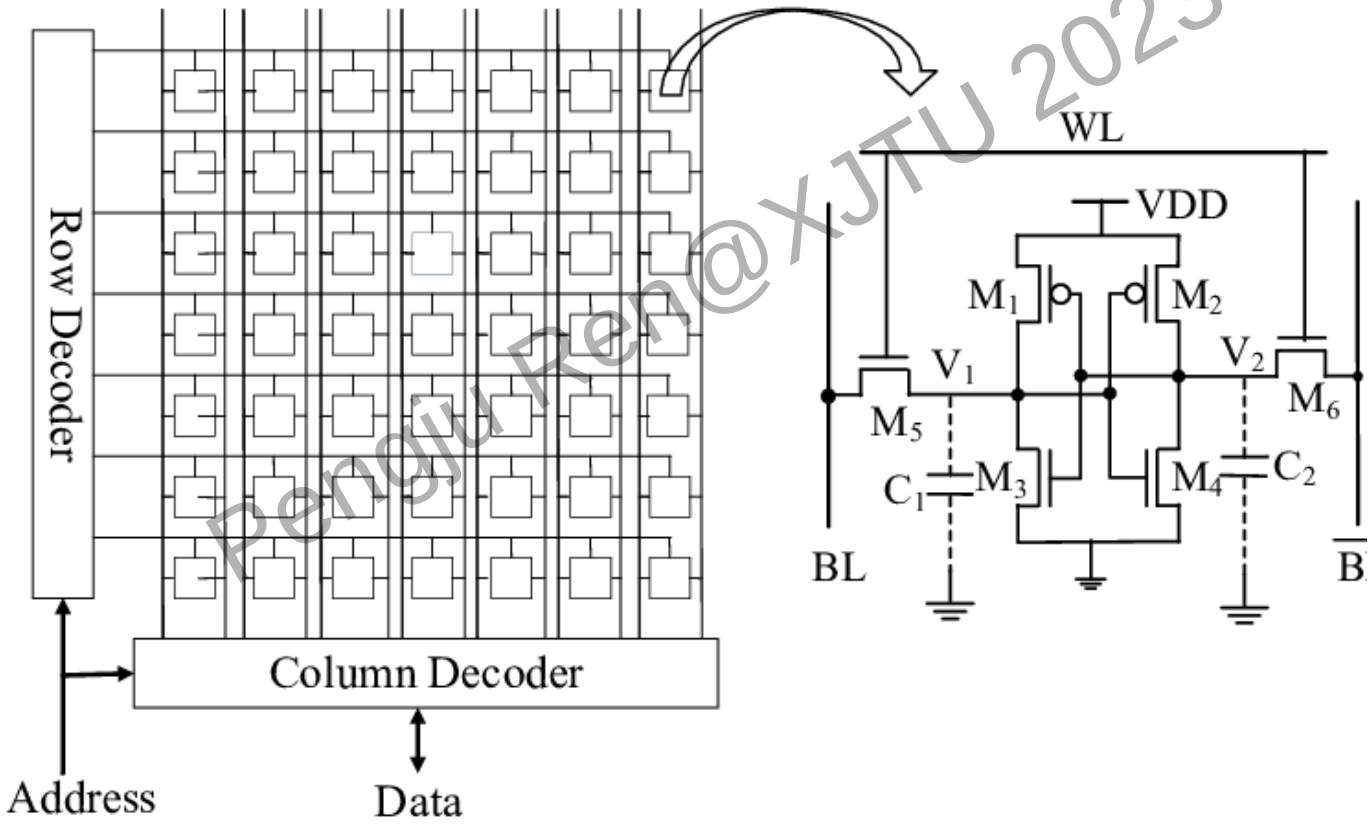




RF design at the level of registers and multiplexers

SRAM (Cache)

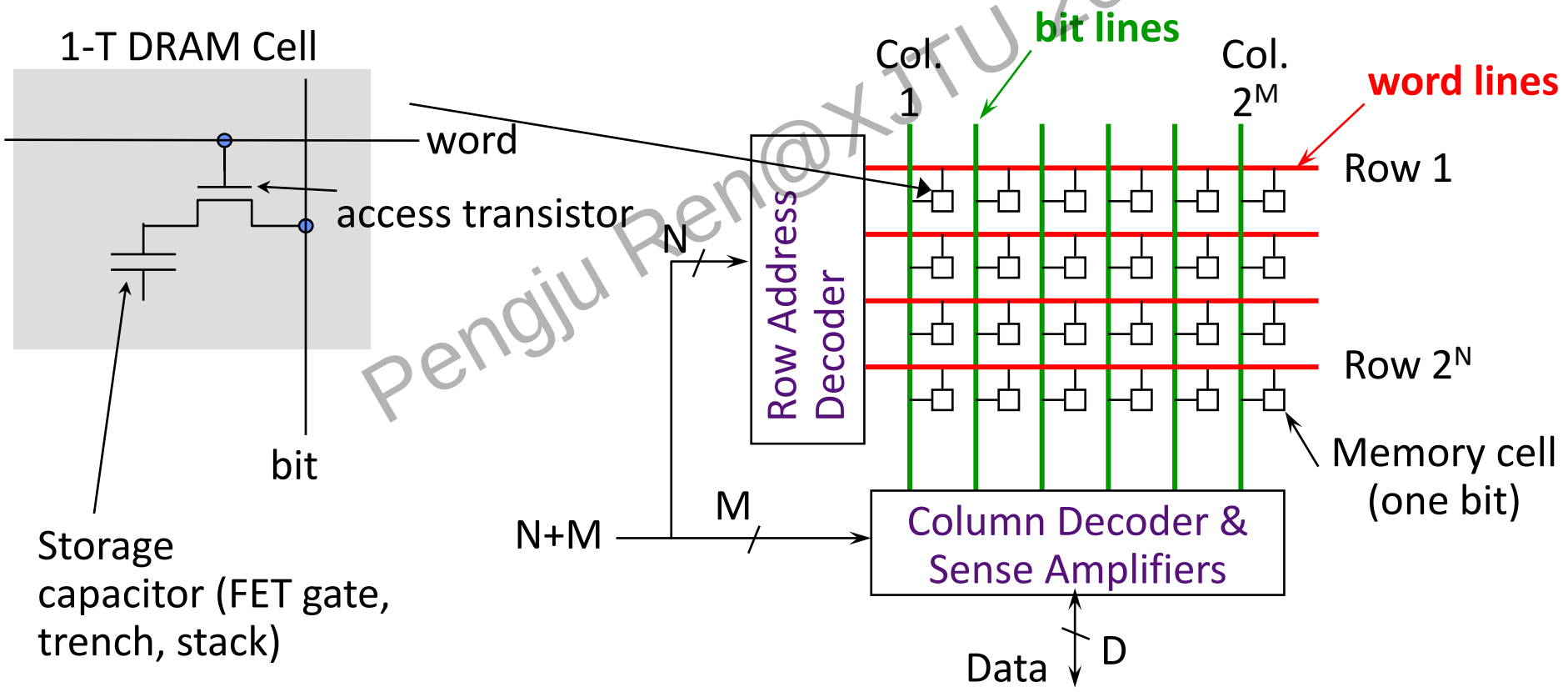
Static Random Access Memory (SRAM): Data is stored in transistors and requires a constant power flow. Because of the continuous power, SRAM *doesn't need to be refreshed* to remember the data being stored.



(a) A typical SRAM array. (b) A six-transistor SRAM cel

DRAM (Main Memory)

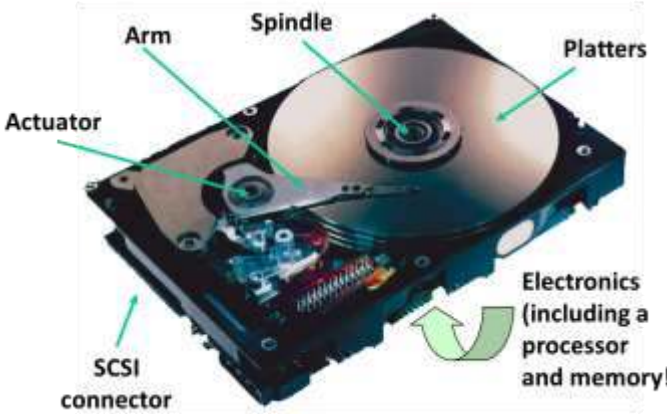
Dynamic Random Access Memory (DRAM): Data is stored in capacitors. Capacitors that store data in DRAM gradually discharge energy. DRAM is called dynamic as *refreshing is needed* to keep the data intact.

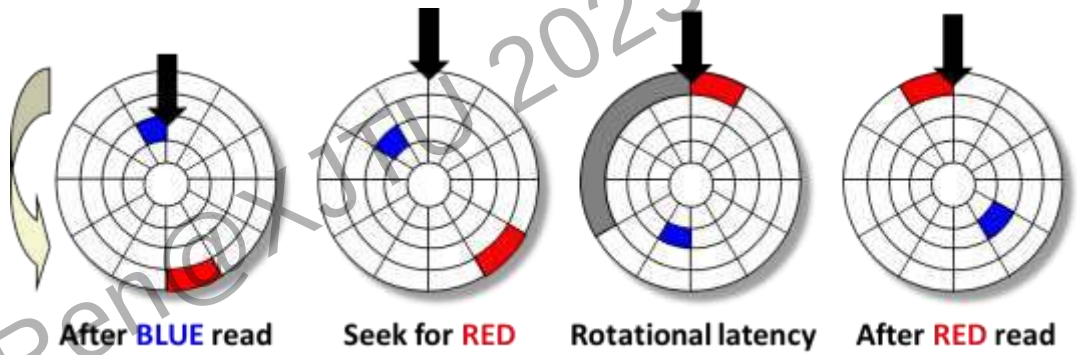


Magnetic Disk

Average time to access some target sector approximated by:

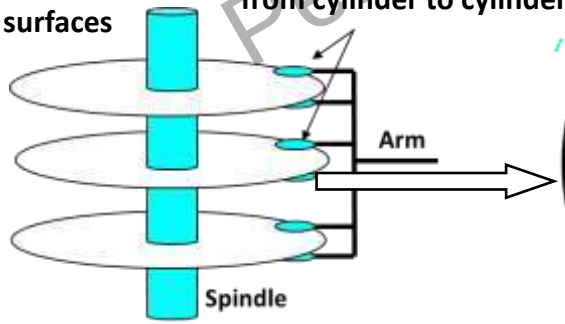
$$T_{access} = T_{avg\ seek} + T_{avg\ rotation} + T_{avg\ transfer}$$



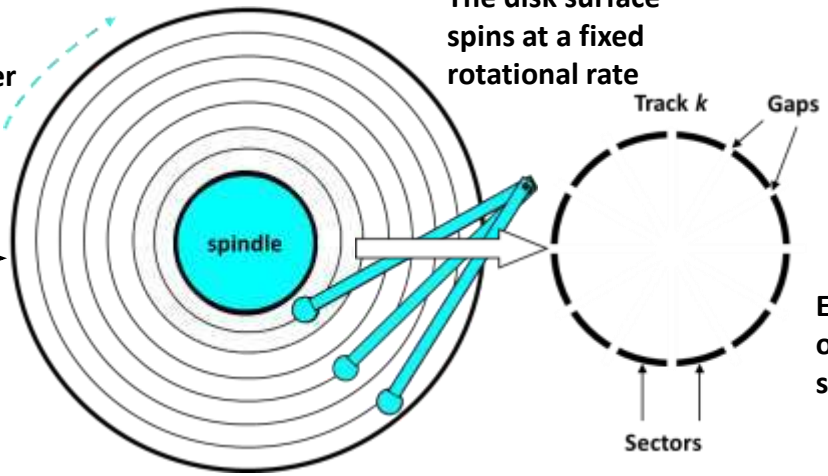


Disks consist of platters, each with two surfaces

Read/write heads move in unison from cylinder to cylinder



The disk surface spins at a fixed rotational rate



Each surface consists of concentric rings called tracks

Each track consists of sectors separated by gaps

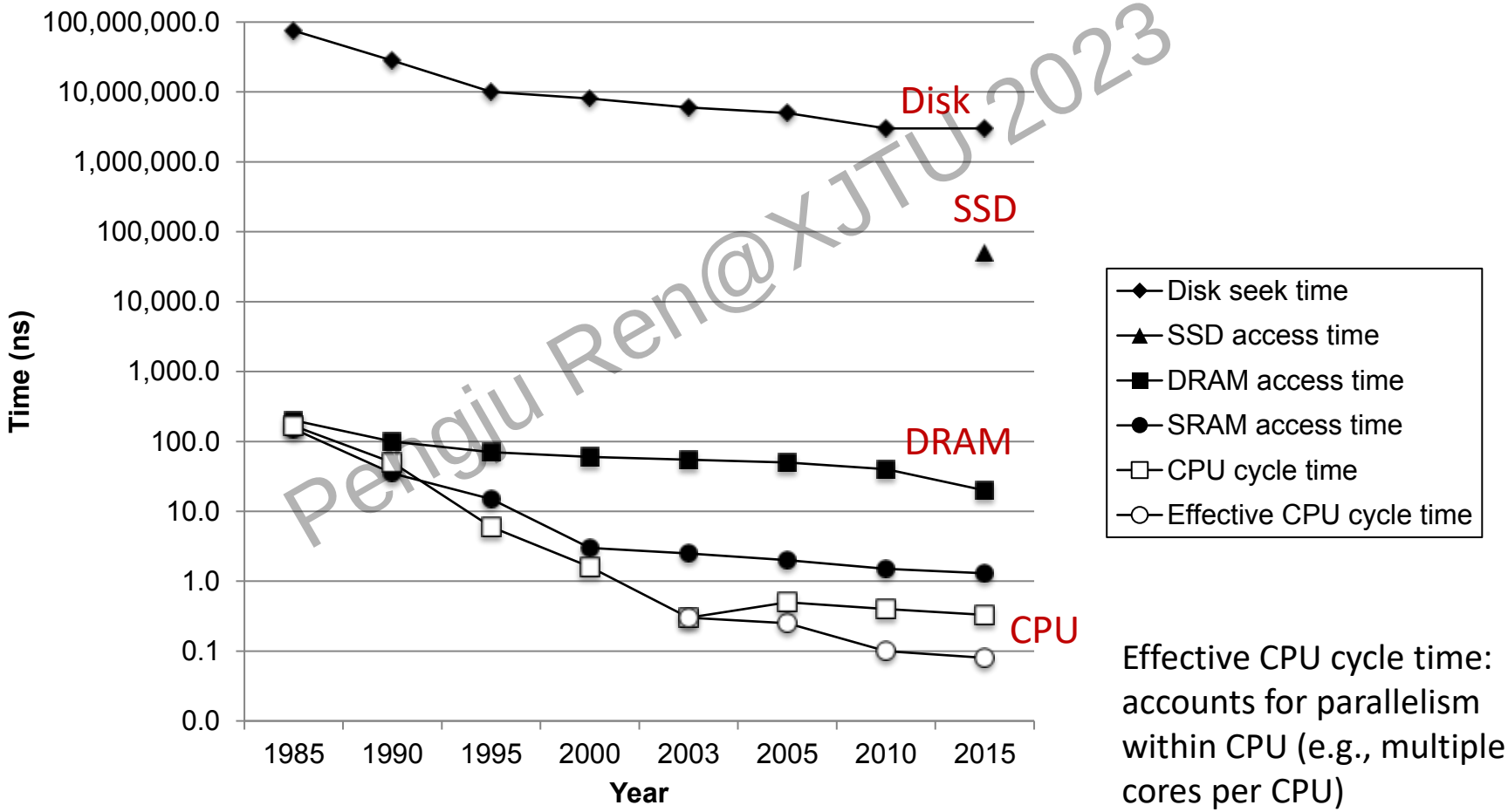
Disk Access Time

- **Average time to access some target sector approximated by:**
  - $T_{access} = T_{avg\ seek} + T_{avg\ rotation} + T_{avg\ transfer}$
- **Seek time ($T_{avg\ seek}$)**
  - Time to position heads over cylinder containing target sector.
  - Typical $T_{avg\ seek}$ is 3—9 ms
- **Rotational latency ($T_{avg\ rotation}$)**
  - Time waiting for first bit of target sector to pass under r/w head.
  - $T_{avg\ rotation}$ = 1/2 x 1/RPMs x 60 sec/1 min
  - Typical rotational rate = 7,200 RPMs
- **Transfer time ($T_{avg\ transfer}$)**
  - Time to read the bits in the target sector.
  - $T_{avg\ transfer}$ = 1/RPM x 1/(avg # sectors/track) x 60 secs/1 min

time for one rotation (in minutes) fraction of a rotation to be read

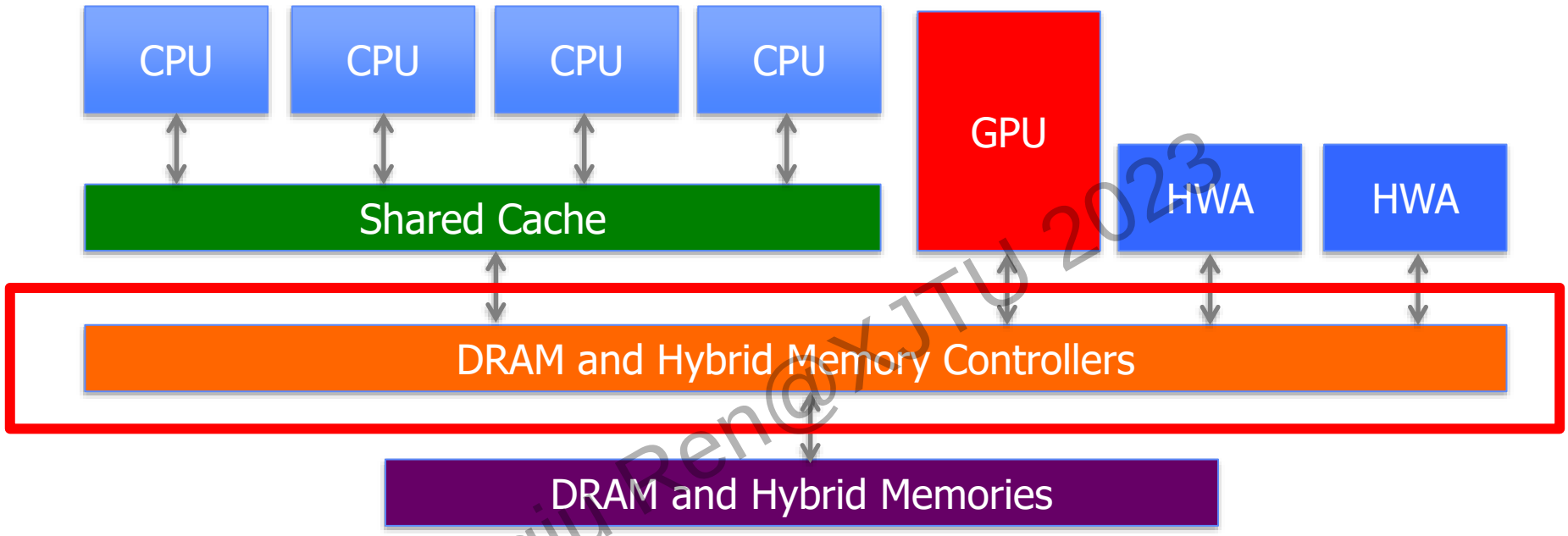
The CPU-Memory Gap

The gap *widens* between DRAM, disk, and CPU speeds.



Effective CPU cycle time: accounts for parallelism within CPU (e.g., multiple cores per CPU)

Memory Control is Getting More Complex



- **Heterogeneous agents: CPUs, GPUs, and HWAs**
- **Main memory interference between CPUs, GPUs, HWAs**

Many goals, many constraints, many metrics …

Management of Memory Hierarchy

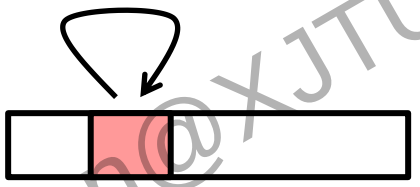
- **Small/fast storage, e.g., registers**
  - **Address usually specified in instruction**
  - **Generally implemented directly as a register file**
    - » ***but hardware might do things behind software's back, e.g., stack management, register renaming***

- **Larger/slower storage, e.g., main memory**
  - **Address usually computed from values in register**
  - **Generally implemented as a hardware-managed cache hierarchy (hardware decides what is kept in fast memory)**
    - » ***but software may provide "hints", e.g., don't cache or prefetch***

Two predictable properties of memory references

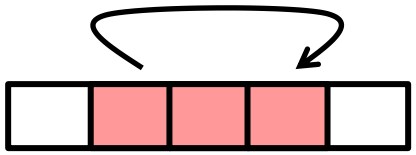
- **Temporal Locality:** If a location is referenced it is likely to be referenced again in the near future.

  -- Exploit *temporal locality* by remembering the contents of recently accessed locations.



- **Spatial Locality:** If a location is referenced it is likely that locations near it will be referenced in the near future.

  -- Exploit *spatial locality* by fetching blocks of data around recently accessed locations.

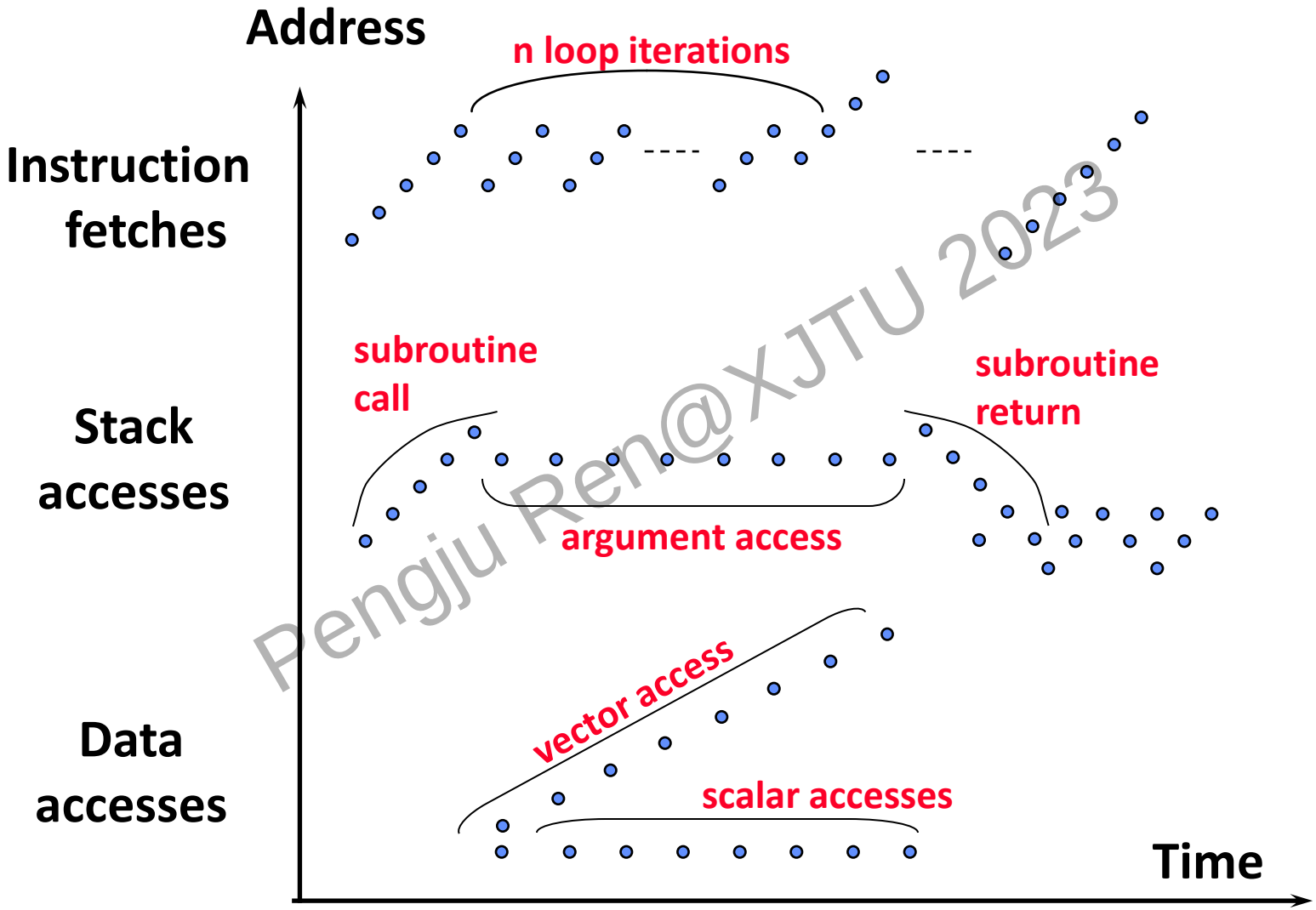


Approaches to Handling Memory Latency

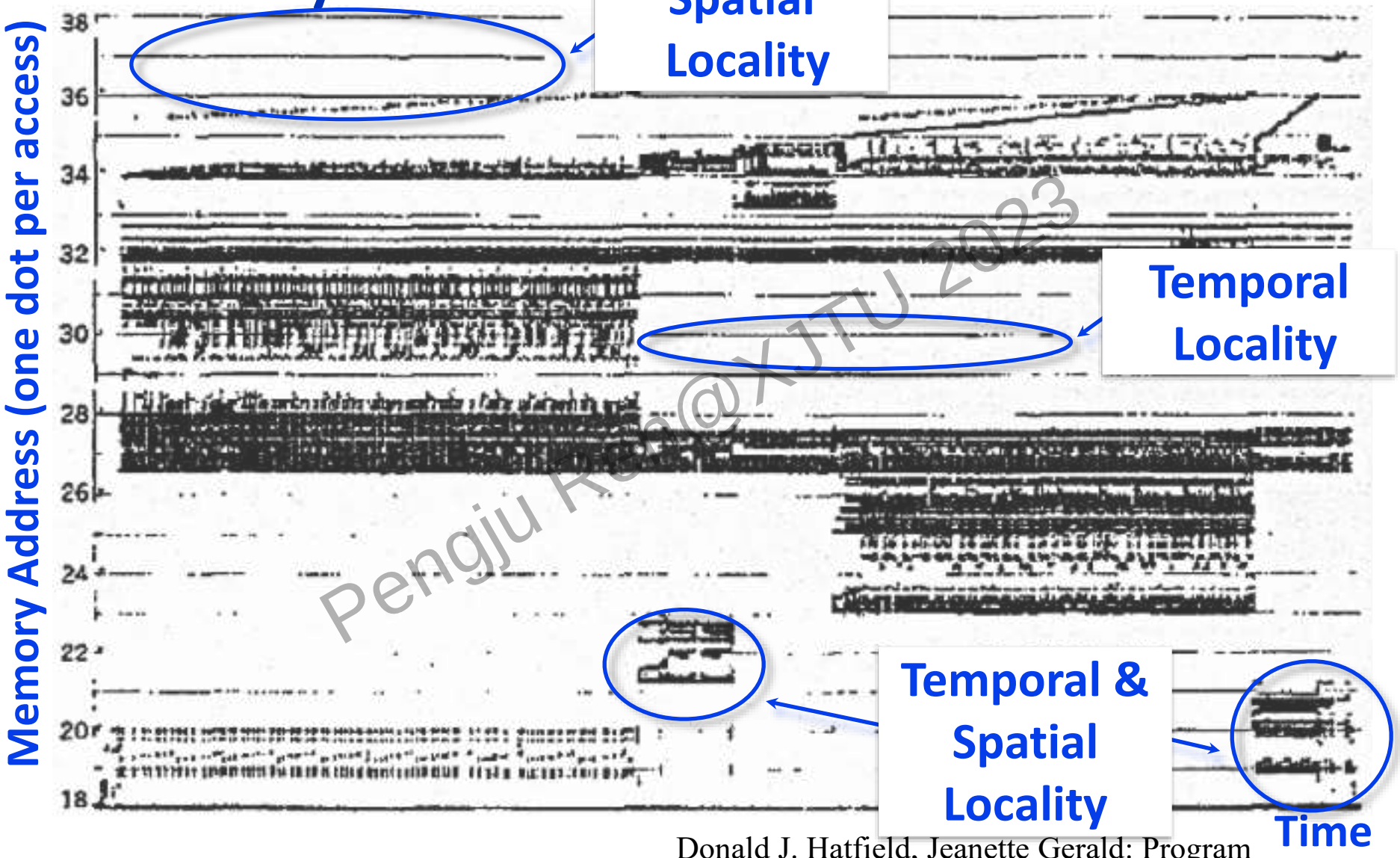
- Reuse values in fast memory (bandwidth filtering)
  - need temporal locality in program
- Move larger chunks (achieve higher bandwidth)
  - need spatial locality in program
- Issue multiple reads/writes in single instruction (higher bw)
  - vector operations require access set of locations (typically neighboring)
- Issue multiple reads/writes in parallel (hide latency)
  - prefetching issues read hint
  - delayed writes (write buffering) stages writes for later operation
  - both require that nothing dependent is happening (parallelism)

concurrency

Typical Memory Reference Patterns



Memory Reference Pattern



Donald J. Hatfield, Jeanette Gerald: Program Restructuring for Virtual Memory. IBM Systems Journal 10(3): 168-192 (1971)

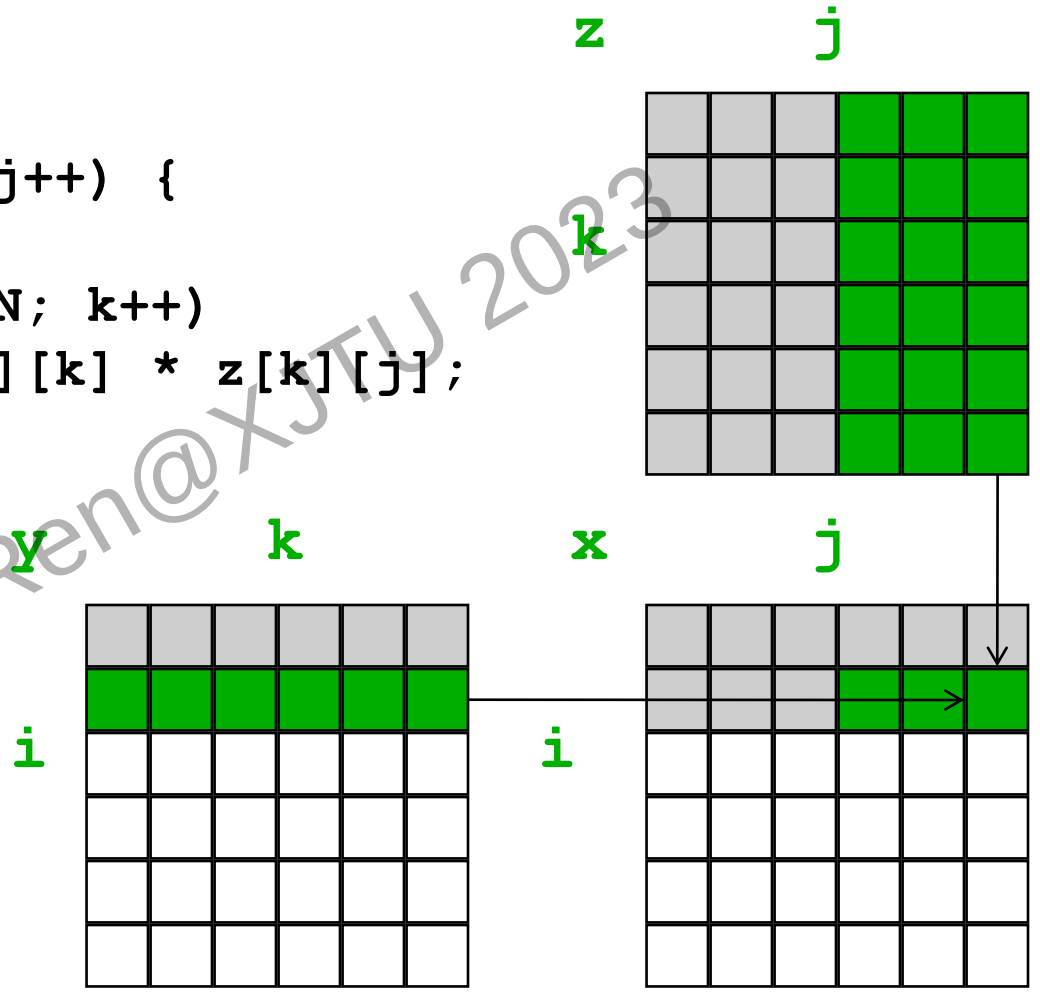
Understanding the Memory Hierarchy is critical to Programming

Parallelism and Locality

- Parallelism and data locality both critical to performance
  - Recall that moving data is the most expensive operation
- Real world problems have parallelism and locality:
  - Many objects operate independently of others.
  - Objects often depend much more on nearby than distant objects.
  - Dependence on distant objects can often be simplified.
    - » Example of all three: particles moving under gravity（n-body）
- Scientific models may introduce more parallelism:
  - When a continuous problem is discretized, time dependencies are generally limited to adjacent time steps.
    - » Helps limit dependence to nearby objects (e.g, collisions)
  - Far-field effects may be ignored or approximated in many cases.
- Many problems exhibit parallelism at multiple levels

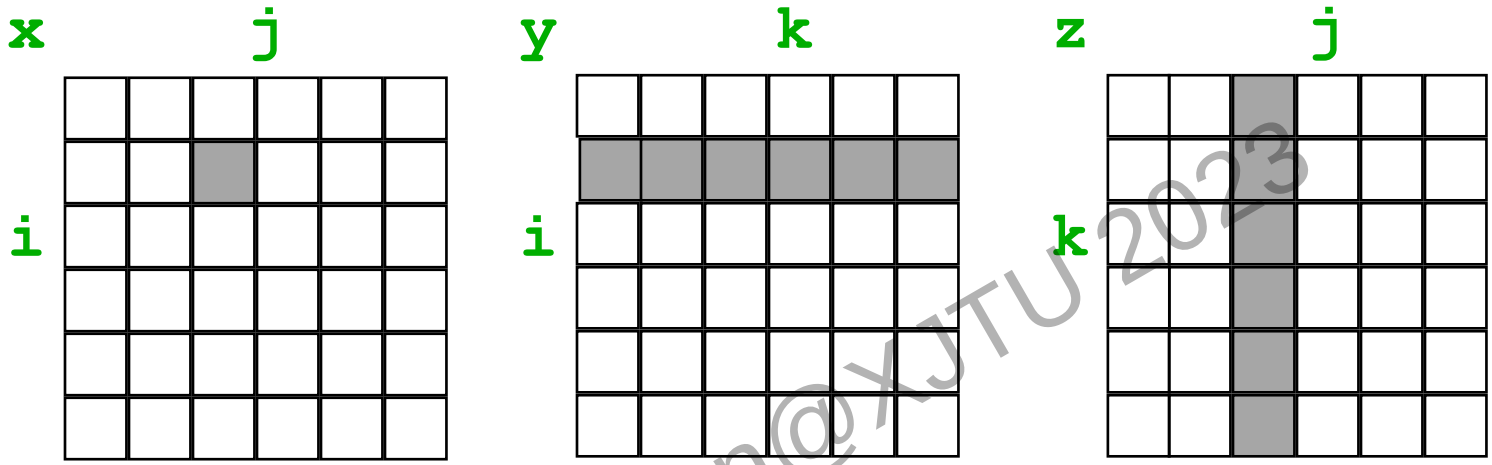
Matrix Multiply, Naïve Code

```
for(i=0; i < N; i++)
     for(j=0; j < N; j++) {
        r = 0;
        for(k=0; k < N; k++)
          r = r + y[i][k] * z[k][j];
        x[i][j] = r;
     }
```



Not touched *Old access* *New access*

Matrix Multiply, Naïve Code (If there is no Cache)



For each element of X, read one row of Y and one column of Z

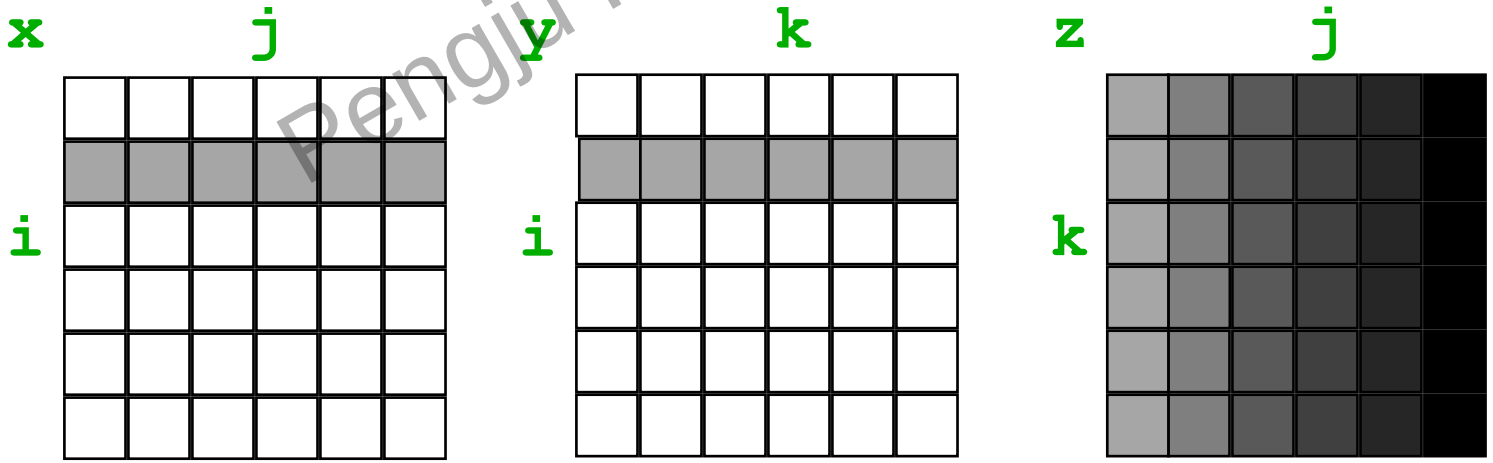
Total Mem access = N^2 x **(2 +** N **+** N**) =** $2\mathrm{N}^2 + 2\mathrm{N}^3$

Computational intensity : $2\mathrm{N}^3/(2\mathrm{N}^2 + 2\mathrm{N}^3) \approx 1$

(including N^3**multiplies and** N^3**addition)**

Matrix Multiply, Naïve Code (If Cache size is 3N)

```
for(i=0; i < N; i++)
        【read row i of y into fast Mem】
       for(j=0; j < N; j++) {
          【read x[i][j] into fast Mem】
          【read column j of z into fast Mem】
          r = 0;
          for(k=0; k < N; k++)
            r = r + y[i][k] * z[k][j];
          x[i][j] = r;
          【write x[i][j] back to fast Mem】
       }
```



For each row of X, read one row of Y and every column of Z

Matrix Multiply, Naïve Code (If Cache size is 3N)

```
for(i=0; i < N; i++)
        【read row i of y into fast Mem】
       for(j=0; j < N; j++) {
          【read x[i][j] into fast Mem】
          【read column j of z into fast Mem】
          r = 0;
          for(k=0; k < N; k++)
            r = r + y[i][k] * z[k][j];
          x[i][j] = r;
          【write x[i][j] back to fast Mem】
       }
```


Total Mem access = N^3 **to read each column of z** N^2 **times** $(N * N^2)$

+ N^2 **to read each row of y once** $(N * N)$

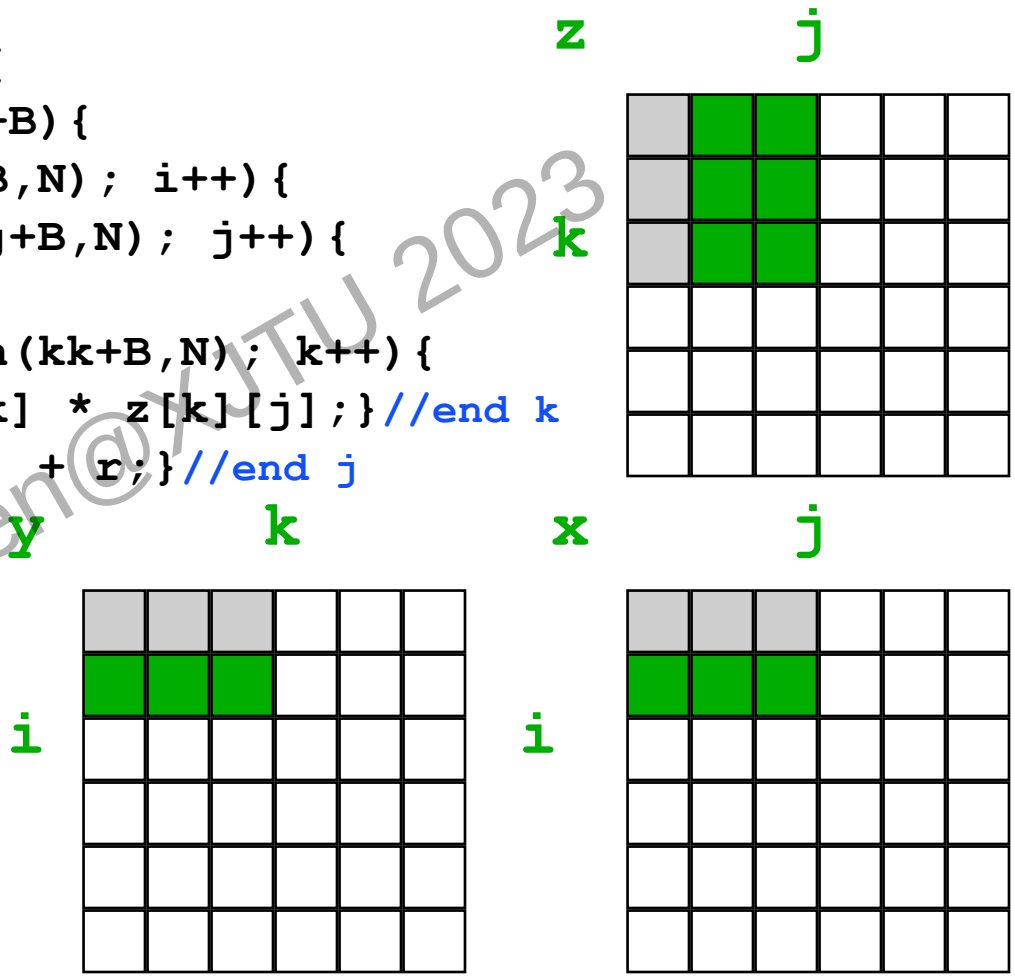
+ $2N^2$ **to read and write each element of x** $(N^2 + N^2)$

= $N^3 + 3N^2$

Computational intensity : $2N^3/(N^3 + 3N^2) \approx 2$

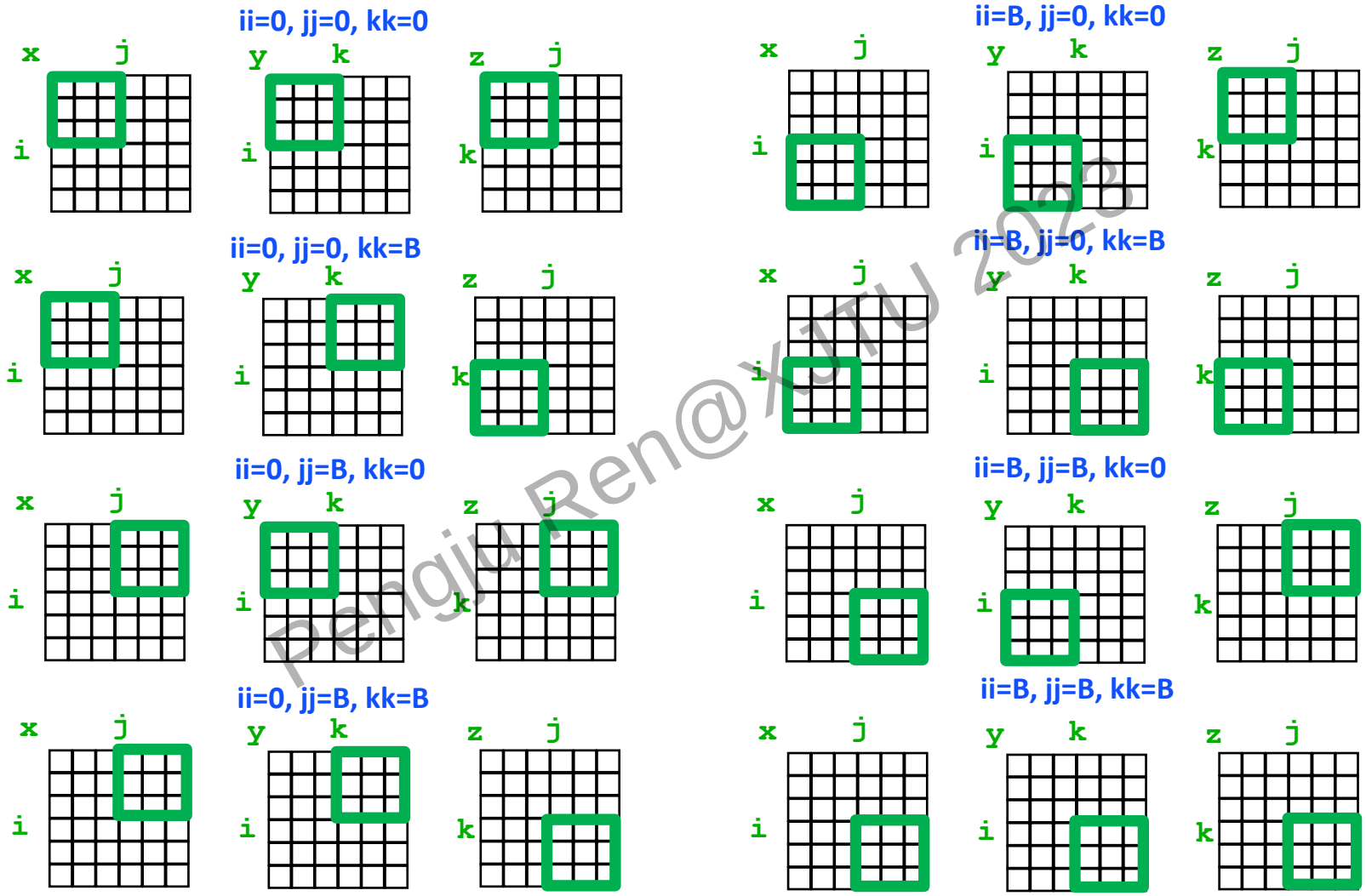
Matrix Multiply with Cache Tiling (If Cache size is bigger than $3B^2$)

```
for(ii=0; ii < N; ii=ii+B){
  for(jj=0; jj < N; jj=jj+B){
    for(kk=0; kk < N; kk=kk+B){
      for(i=ii; i < min(ii+B,N); i++){
        for(j=jj; j < min(jj+B,N); j++){
          r = 0;
          for(k=kk; k < min(kk+B,N); k++){
            r = r + y[i][k] * z[k][j];}//end k
          x[i][j] = x[i][j] + r;}//end j
      }//end i
    }//end kk
  }//end jj
}//end ii
```



What type of locality does this improve?

Matrix Multiply with Cache Tiling (If Cache size is bigger than $3B^2$)



Matrix Multiply with Cache Tiling (If Cache size is bigger than $3B^2$)

```
for(ii=0; ii < N; ii=ii+B){
   for(jj=0; jj < N; jj=jj+B){
     【read B*B block of x into fast Mem】
     for(kk=0; kk < N; kk=kk+B){
        【read B*B block of y into fast Mem】
        【read B*B block of z into fast Mem】
       for(i=ii; i < min (ii+B,N); i++)
          for(j=jj; j < min(jj+B,N); j++) {
             r = 0;
             for(k=kk; k < min(kk+B,N); k++)
                r = r + y[i][k] * z[k][j];
             x[i][j] = x[i][j] + r;}
           }
       }
     }
```

The larger the block size, the more efficient our algorithm will be, however all three blocks from x,y,z must fit in Cache

$3b^2 \le M_{fast}$, so $b \le \sqrt{M_{fast}/3}$

Total Mem access = N^3 /B to read each block of z $(\frac{N}{B})^3$ times ($(\frac{N}{B})^3 * B^2 = N^3$ /B)

+ N^3 /B to read each block of y $(\frac{N}{B})^3$ times

+ $2N^2$ read and write each block of x once ($2(\frac{N}{B})^2 * B^2 = 2N^2$)

= $2N^3$ /B + $2N^2$

Computational intensity : $2N^3/(2N^3$ /B + $2N^2) \approx B$ when N is big

Is there a more elegant approach ?

Recursive Matrix Multiplication

$$C = \begin{pmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{pmatrix} = A \cdot B = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} \cdot \begin{pmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{pmatrix} = \begin{pmatrix} A_{00}\cdot B_{00} + A_{01}\cdot B_{10} & A_{00}\cdot B_{01} + A_{01}\cdot B_{11} \\ A_{10}\cdot B_{00} + A_{11}\cdot B_{10} & A_{10}\cdot B_{01} + A_{11}\cdot B_{11} \end{pmatrix}$$

C_{00}	C_{01}
C_{10}	C_{11}

A_{00}	A_{01}
A_{10}	A_{11}

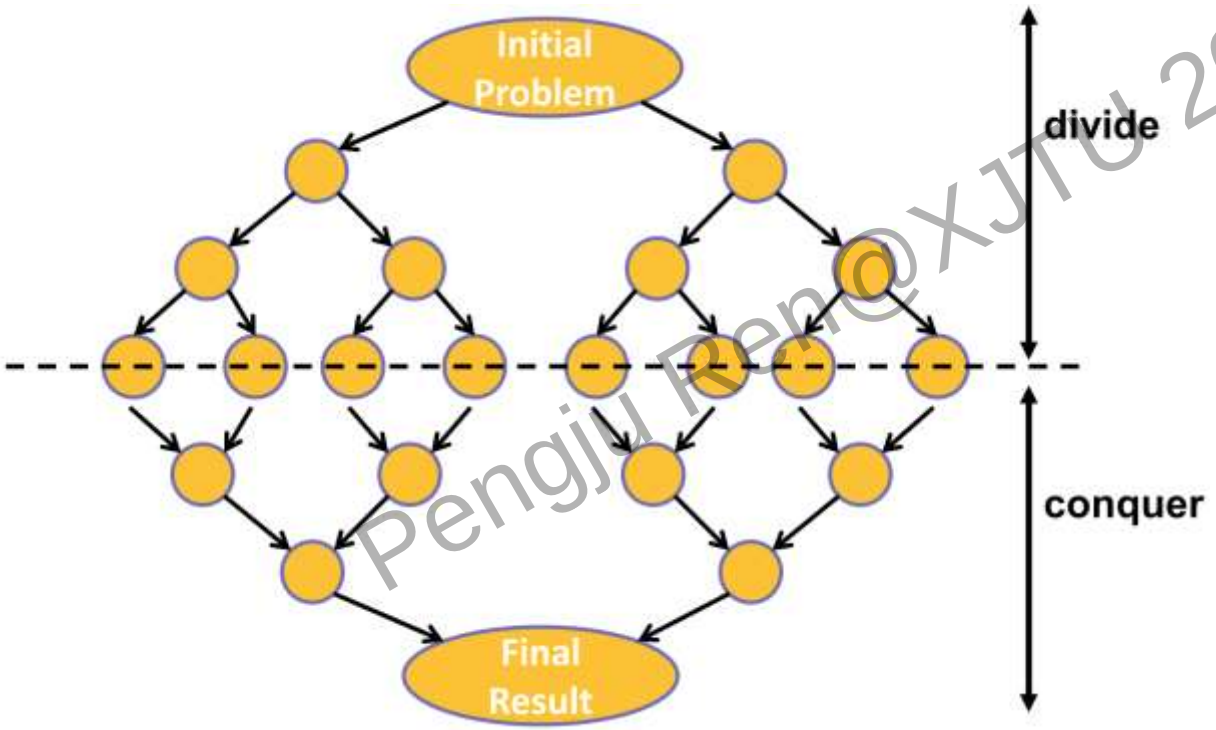
B_{00}	B_{10}
B_{10}	B_{11}

$A_{00}*B_{00} + A_{01}*B_{10}$	$A_{00}*B_{01} + A_{01}*B_{11}$
$A_{10}*B_{00} + A_{11}*B_{10}$	$A_{10}*B_{01} + A_{11}*B_{11}$

- True when each bock is a 1x1 or n/2 x n/2
- For simplicity: square matrices with $n = 2^m$
  - Extends to general rectangular case

Divide and conquer

- Split the problem into smaller sub-problems
- continue until the sub-problems can be solve directly



3 Options:

- Do work as you split into sub-problems
- Do work only at the leaves
- Do work as you recombine

Recursive Matrix Multiplication

```
Define C = RMM (A, B, n)
    if (n==1) { C00 = A00 * B00 ; } else
       {  C00 = RMM (A00 , B00 , n/2) + RMM (A01 , B10 , n/2)
          C01 = RMM (A00 , B01 , n/2) + RMM (A01 , B11 , n/2)
          C10 = RMM (A10 , B00 , n/2) + RMM (A11 , B10 , n/2)
          C11 = RMM (A11 , B01 , n/2) + RMM (A11 , B11 , n/2)}
```

For RMM, we DO NOT need to know M_{fast} in stead of Blocking and Tiling

Arith(n) = # arithmetic operations in `RMM(.,.,n)`

$= 8 \cdot \text{Arith}(n/2) + 4(n/2)^2$ if $n > 1$, else 1

$= 2n^3$

W(n) = # words moved between fast, slow memory by `RMM(.,.,n)`

$= 8 \cdot W(n/2) + 4 \cdot 3(n/2)^2$ if $3n^2 > M_{fast}$, else $3n^2$

$= O(n^3 / \sqrt{M_{fast}})$ … same as blocked matmul

Strassen's Matrix Multiply

- The traditional algorithm (with or without tiling) has $O(n^3)$ flops
- Strassen discovered an algorithm with asymptotically lower flops $O(n^{2.81})$
- Consider a 2x2 matrix multiply, normally takes 8 multiplies, 4 adds
  - Strassen does it with 7 multiplies and 18 adds

$$\text{Let } M = \begin{pmatrix} m11 & m12 \\ m21 & m22 \end{pmatrix} = \begin{pmatrix} a11 & a12 \\ a21 & a22 \end{pmatrix} \begin{pmatrix} b11 & b12 \\ b21 & b22 \end{pmatrix}$$

```
Let   p1 = (a12 - a22) * (b21 + b22)     p5 = a11 * (b12 - b22)
      p2 = (a11 + a22) * (b11 + b22)     p6 = a22 * (b21 - b11)
      p3 = (a11 - a21) * (b11 + b12)     p7 = (a21 + a22) * b11
      p4 = (a11 + a12) * b22

Then  m11 = p1 + p2 - p4 + p6
      m12 = p4 + p5
      m21 = p6 + p7
      m22 = p2 - p3 + p5 - p7
```

Extends to nxn by divide & conquer

Strassen (continued)

T(n) = Cost of multiplying nxn matrices

$$= 7T\left(\frac{n}{2}\right) + 18\left(\frac{n}{2}\right)^2$$

$$= O(n^{\log_2 7})$$

$$= O(n^{2.81})$$

- Asymptotically faster
  - Several times faster for large n in practice
  - Cross-over depends on machine
  - "Tuning Strassen's Matrix Multiplication for Memory Efficiency", M. S. Thottethodi, S. Chatterjee, and A. Lebeck, in Proceedings of Supercomputing '98

Master Theorem

The *Master Theorem* for solving recurrences applies to recurrences of the form

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) \qquad \text{Let } f(n) = n^c$$

where $a \geq 1$, $b > 1$, and f is asymptotically positive.

IDEA: Compare $n^{\log_b a}$ with n^c .

- **CASE 1:** $\log_b a \gg c$

  $f(n) = O(n^{\log_b a - \varepsilon})$, constant $\varepsilon > 0$ $\Rightarrow T(n) = \Theta(n^{\log_b a})$.
- **CASE 2:** $\log_b a \approx c$

  $f(n) = \Theta(n^{\log_b a} \lg^k n)$, constant $k \geq 0$ $\Rightarrow T(n) = \Theta(n^{\log_b a} \log n)$.
- **CASE 3:** $\log_b a \ll c$

  $f(n) = \Omega(n^{\log_b a + \varepsilon})$, constant $\varepsilon > 0$ $\Rightarrow T(n) = \Theta(f(n))$.

Master Theorem

- **Recursive Matrix Multiplication**

$$T(n) = 8T\left(\frac{n}{2}\right) + n^2 \quad \Rightarrow \quad O(n^{\log_2 8}) = O(n^3)$$

- **Strassen Matrix Multiplication**

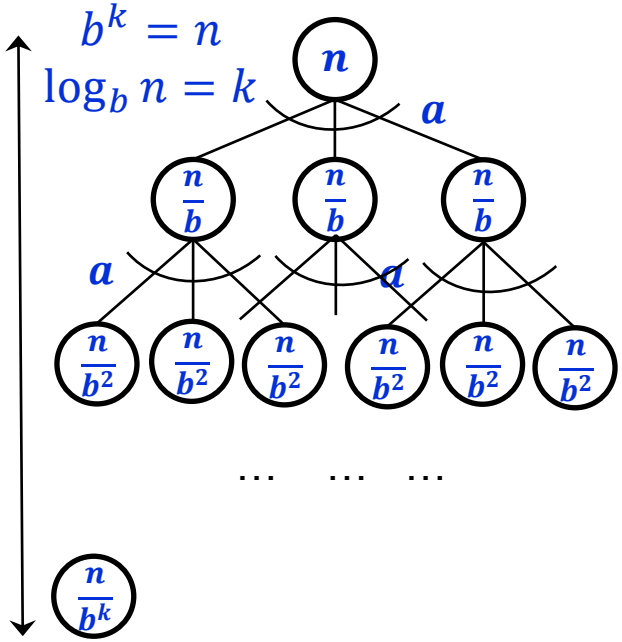
$$T(n) = 7T\left(\frac{n}{2}\right) + 18\left(\frac{n}{2}\right)^2 \quad \Rightarrow \quad O(n^{\log_2 7}) = O(n^{2.81})$$

- **Binary Search:** $T(n) = T\left(\frac{n}{2}\right) + O(1) \quad \Rightarrow \quad O(\log_2 n)$

- **Merge Sort:** $T(n) = 2T\left(\frac{n}{2}\right) + O(n) \quad \Rightarrow \quad O(n \log_2 n)$

- **Binary Tree traversal:** $T(n) = 2T\left(\frac{n}{2}\right) + O(1) \quad \Rightarrow \quad O(n)$

Master Theorem



$\rightarrow f(n)$

$\rightarrow af(\frac{n}{b})$

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

Let $f(n) = n^c$

$$T(n) = f(n) + af\left(\frac{n}{b}\right) + a^2 f\left(\frac{n}{b^2}\right) + \cdots a^k f(1)$$

$$= \sum_{i=0}^{k} a^i f\left(\frac{n}{b^i}\right) = \sum_{i=0}^{k} a^i \left(\frac{n}{b^i}\right)^c = n^c \sum_{i=0}^{k} \left(\frac{a}{b^c}\right)^i$$

Let $\frac{a}{b^c} = q$

$$T(n) = n^c \frac{1 - q^{k+1}}{1 - q}$$

1) $q < 1 \Rightarrow \frac{a}{b^c} < 1 \Rightarrow a < b^c \Rightarrow c > \log_b a$ $\quad T(n) = n^c \frac{1}{1-q} = \alpha\, n^c \Rightarrow T(n) = O(n^c)$

2) $q = 1 \Rightarrow \frac{a}{b^c} = 1 \Rightarrow a = b^c \Rightarrow c = \log_b a$ $\quad T(n) = n^c k = n^c \log_b n = n^c \frac{\log_2 n}{\log_2 b} \Rightarrow \beta n^c \log_2 n$

$\Rightarrow T(n) = O(n^c \log_2 n) = O(n^{\log_b a} \log_2 n)$

3) $q > 1 \Rightarrow \frac{a}{b^c} > 1 \Rightarrow a > b^c \Rightarrow c < \log_b a$ $\quad T(n) = n^c q^k = n^c \left(\frac{a}{b^c}\right)^{\log_b n} = n^c \frac{a^{\log_b n}}{b^{c \log_b n}}$

$$= n^c \frac{a^{\log_b n}}{(b^{\log_b n})^c} = n^c \frac{a^{\log_b n}}{n^c} = a^{\frac{\log_a n}{\log_a b}} = a^{\log_a n \log_b n} = n^{\log_b a} \quad \Rightarrow T(n) = O(n^{\log_b a})$$

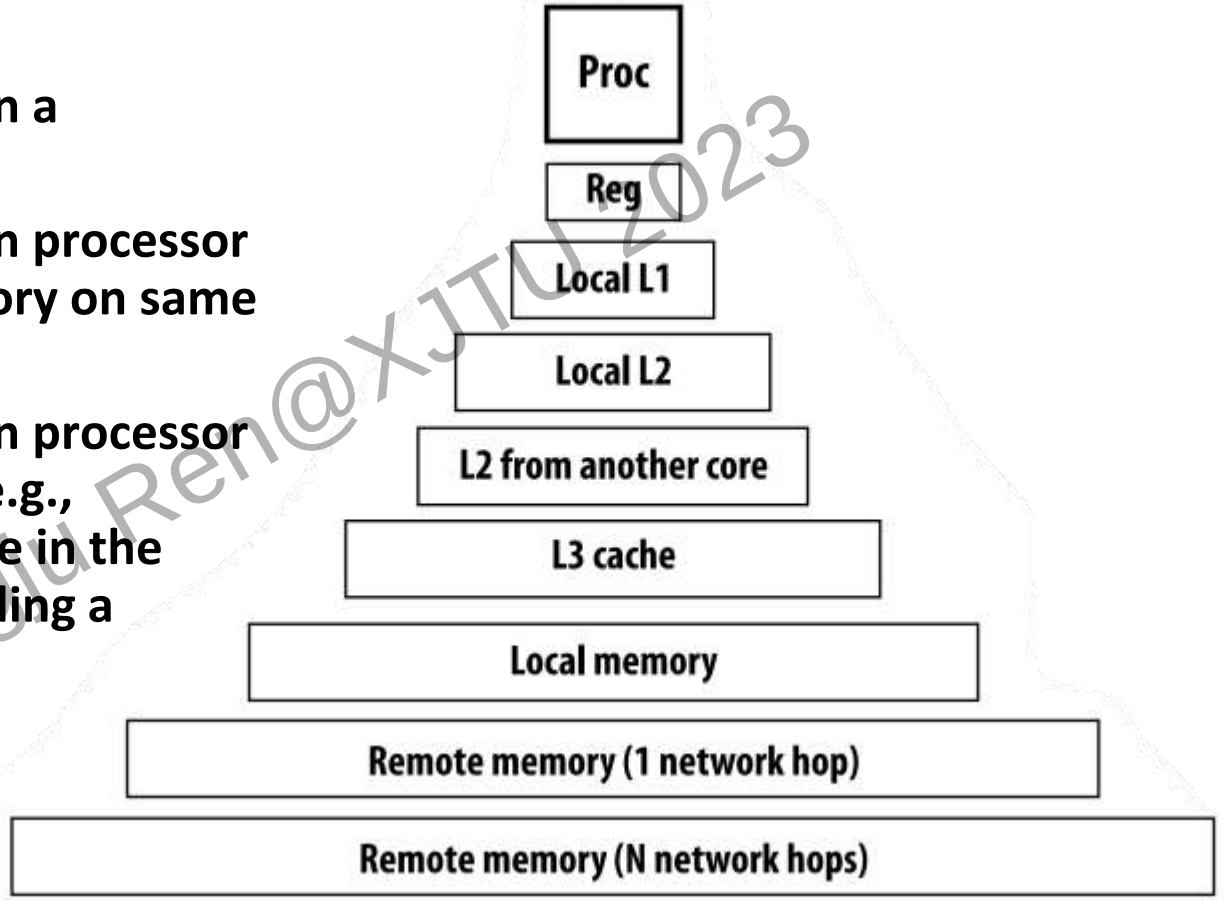
Matrix-matrix multiplication take-aways

- **Matrix matrix multiplication**
  - Computational intensity $O(2n^3)$ flops on $O(3n^2)$ data
- **Tiling matrix multiplication (cache aware)**
  - Can increase to B if BxB blocks fit in fast memory
  - B = $\sqrt{M_{fast}/3}$, the fast memory size M_{fast}
  - Tiling (a.k.a blocking) "*cache-aware*"
  - *Cache-oblivious (Recursive Matrix Multiplication)*

Roofline Model
(How fast can an algorithm go in practice?)

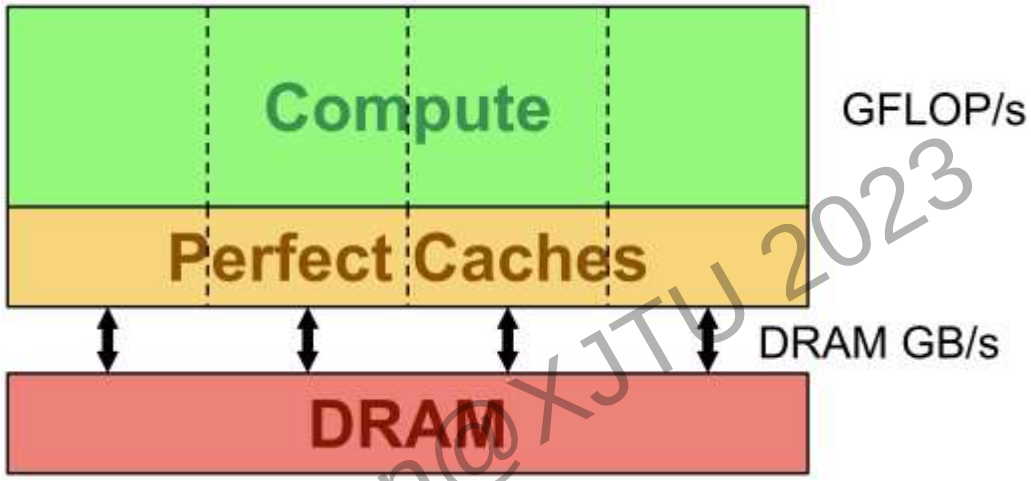
Think of “Data movement” very generally

- **Data movement** between a processor and its cache
- **Data movement** between processor and memory (e.g., memory on same machine)
- **Data movement** between processor and a remote memory (e.g., memory on another node in the cluster, accessed by sending a network message)



Accesses not satisfied in “local memory” cause communication with “next level”

Data Movement or Compute



Which takes longer? Data movement or Compute?

Time = max (#FP ops/Peak GFLOP/s, #Bytes/Peak GB/s)

#FP ops/Time = min (Peak GFLOP/s, (#FP ops/#Bytes)*Peak GB/s)

AI (Arithmetic or Computational Intensity)

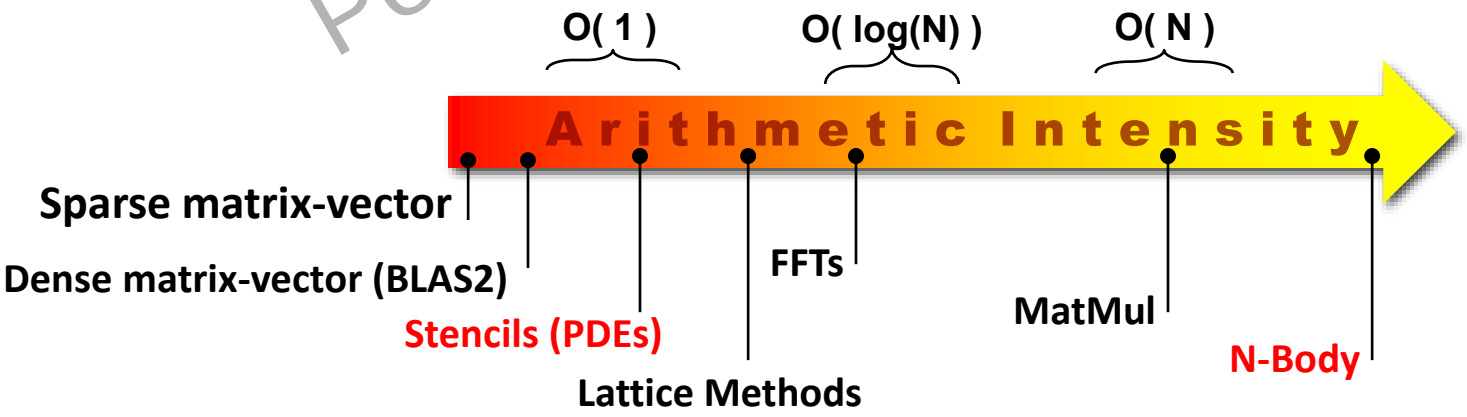
Arithmetic or Computational Intensity

- Can look at *Arithmetic intensity* as a *spectrum*
- Constants (at least leading constants) will matter

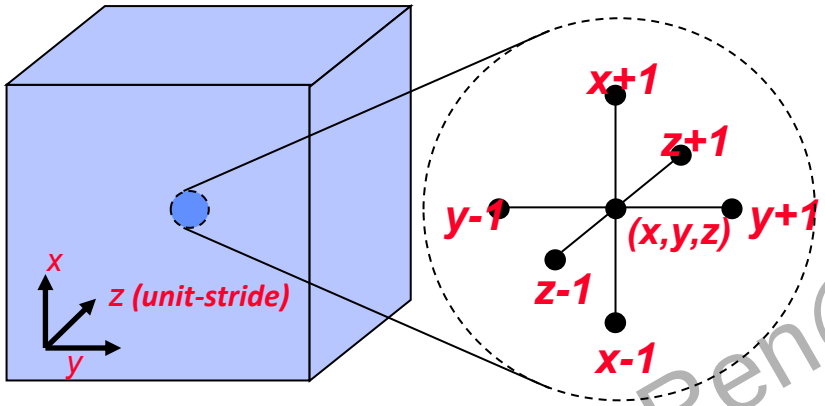
Operation	FLOPs	Data
Dot Prod	O(n)	O(n)
Mat Vec	$O(n^2)$	$O(n^2)$
MatMul	$O(n^3)$	$O(n^2)$
N-Body	$O(n^2)$	O(n)
FFT	O(n log n)	O(n)

Ideal (infinite cache)

Ideal cache – to be refined



Cases : 3D 7-point Stencil




```
for x,y,z in 0 to n-1
 Next[x,y,z]=
    C0 * Current[x,y,z]+
    C1 *(Current[x-1,y,z]+
         Current[x+1,y,z]+
         Current[x,y-1,z]+
         Current[x,y+1,z]+
         Current[x,y,z-1]+
         Current[x,y,z+1]);
```

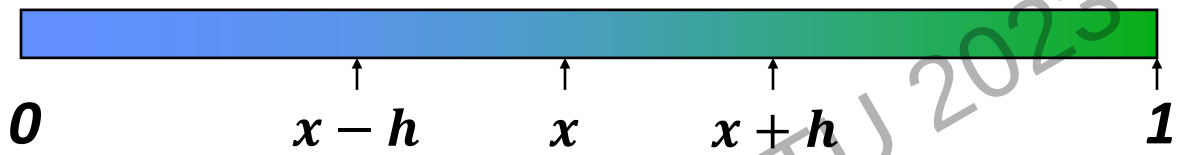

3D: *"7-point stencil"*

8 flops, 8 memory references (7 reads, 1 store) per point

The memory layout of data structure and Cache matters

PDEs (Partial Differential Equations)

Continuous variables depending on continuous parameters
E.g. Heat, Elasticity, Electrostatics, Finance, Circuits …



Consider Deriving the heat equation:
A bar of uniform material, insulated except at ends
Let $u(x,t)$ be the temperature at position x at time t
Heat travels from $x-h$ to $x+h$ at rate proportional to:

$$\frac{du(x,t)}{dt} = c\frac{\frac{u(x-h,t)-u(x,t)}{h} - \frac{u(x,t)-u(x+h,t)}{h}}{h}$$

As $h \to 0$, we get the heat equation:

$$\frac{du(x,t)}{dt} = c\frac{d^2u(x,t)}{dx^2}$$

PDEs (Partial Differential Equations)

$$\frac{du(x,t)}{dt} = c\frac{d^2u(x,t)}{dx^2}$$

Discretize time and space using explicit approach (forward Euler) to approximate time derivative:

$$\frac{u(x,t+\delta)-u(x,t)}{\delta} = C\frac{\frac{u(x-h,t)-u(x,t)}{h} - \frac{u(x,t)-u(x+h,t)}{h}}{h}$$

$$u(x,t+\delta) = u(x,t) + C\frac{\delta}{h^2}\big(u(x-h,t) - 2u(x,t) + u(x+h,t)\big)$$

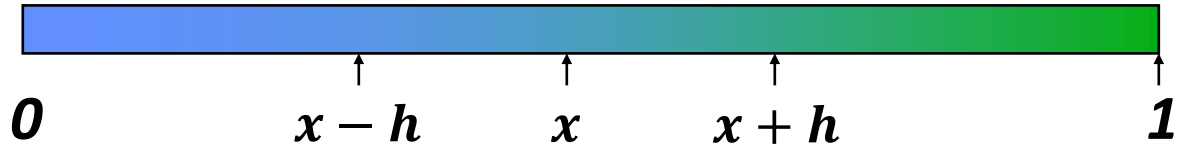
Let $z = C\frac{\delta}{h^2}$, then

$$u(x,t+\delta) = z * u(x-h,t) + (1-2z) * u(x,t) + z * u(x+h,t)$$

and change variable $x = \mathrm{i} * \mathrm{h}$, $t = \mathrm{k} * \delta$ and $u(x,t)$ to $u(i,k)$, we get:

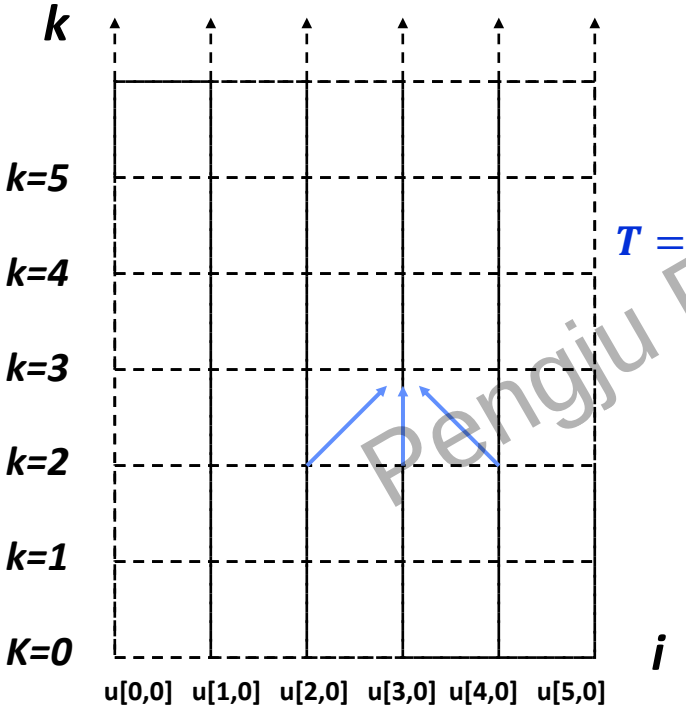
$$u(i,k+1) = z * u(i-1,k) + (1-2z) * u(i,k) + z * u(i+1,k))$$

PDEs (Partial Differential Equations)



$$u(i,k+1) = z * u(i-1,k) + (1-2z) * u(i,k) + z * u(i+1,k))$$

$$u(:,k+1) = T * u(:,k)$$



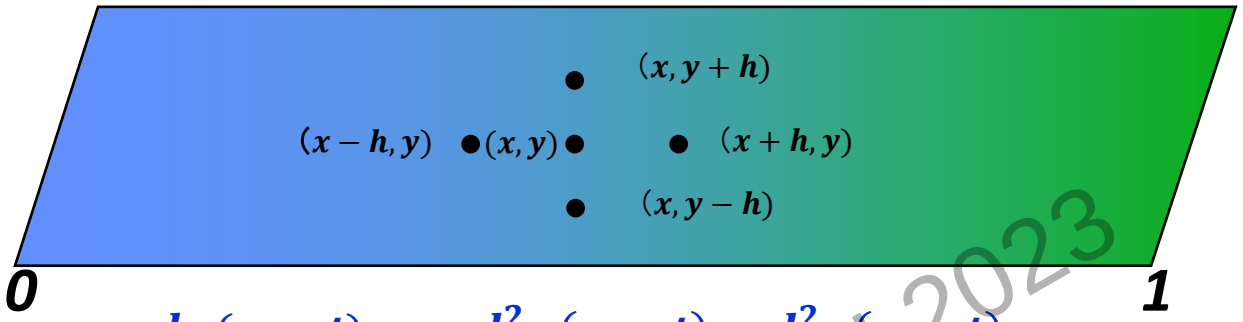
$$T = \begin{bmatrix} 1-2z & z & & & & \\ z & 1-2z & z & & & \\ & z & 1-2z & z & & \\ & & \dots & \dots & \dots & \\ & & & z & 1-2z & z \\ & & & & z & 1-2z \end{bmatrix} = I + z * L$$

$$L = \begin{bmatrix} -2 & 1 & & & & \\ 1 & -2 & 1 & & & \\ & 1 & -2 & 1 & & \\ & & & \dots & \dots & \dots \\ & & & 1 & -2 & 1 \\ & & & & 1 & -2 \end{bmatrix}$$

initial conditions on $u(i,0)$
boundary conditions on $u(0,k)$ and $u(N,k)$

1-D : "*3-point Stencil*"

PDEs (Partial Differential Equations)



$$\frac{du(x,y,t)}{dt} = c\left(\frac{d^2u(x,y,t)}{dx^2} + \frac{d^2u(x,y,t)}{dy^2}\right)$$

$$\frac{u(x,y,t+\delta) - u(x,y,t)}{\delta}$$

$$= C\left(\frac{\frac{u(x-h,y,t)-u(x,y,t)}{h} - \frac{u(x,y,t)-u(x+h,y,t)}{h}}{h} + \frac{\frac{u(x,y-h,t)-u(x,y,t)}{h} - \frac{u(x,y,t)-u(x,y+h,t)}{h}}{h}\right)$$

$$u(x,y,t+\delta)$$

$$= u(x,y,t) + C\frac{\delta}{h^2}\left((x-h,y,t) + u(x+h,y,t) + u(x,y-h,t) + u(x,y+h,t) - 4u(x,y,t)\right)$$

Let $z = C\frac{\delta}{h^2}$, then

$$u(x,y,t+\delta)$$
$$= z * u(x-h,y,t) + z * u(x+h,y,t) + z * u(x,y-h,t) + z * u(x,y+h,t) - (1-4z) * u(x,y,t))$$

PDEs (Partial Differential Equations)

$$u(x,y,t+\delta)$$
$$= z*u(x-h,y,t) + z*u(x+h,y,t) + z*u(x,y-h,t) + z*u(x,y+h,t) - (1-4z)*u(x,y,t))$$

change variable $x = \mathrm{i}*\mathrm{h}$, $\mathrm{y} = \mathrm{j}*\mathrm{h}$, $\mathrm{t} = \mathrm{k}*\delta$ and $u(x,y,t)$ to $u(i,j,k)$, we get:

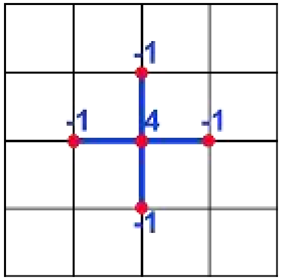
$$u(i,j,k+1)$$
$$= z*u(i-1,j,k) + z*u(i+1,j,k) + z*u(i,j-1,k) + z*u(i,j+1,k) - (1-4z)*u(i,j,k))$$

$$T = \left[\begin{array}{ccc:ccc:ccc} 1-4z & z & & z & & & & & \\ & & & & & & & & \\ & & & & & & & & \\ & & & & & & & & \\ & & z & & z & 1-4z & & & z \\ \hdashline & & & z & & & 1-4z & z & \\ & & & & z & & & 1-4z & z \\ & & & & & z & & z & 1-4z \end{array}\right] - z*L$$

How to speedup stencil kernel ?

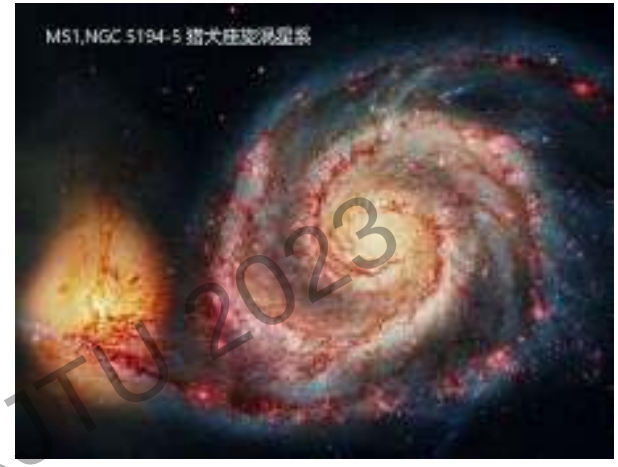
Can it benefits from Cache?

$$L = \left[\begin{array}{ccc:ccc:ccc} 4 & -1 & & -1 & & & & & \\ -1 & 4 & -1 & & -1 & & & & \\ & -1 & 4 & & & -1 & & & \\ \hdashline -1 & & & 4 & -1 & & -1 & & \\ & -1 & & -1 & 4 & -1 & & -1 & \\ & & -1 & & -1 & 4 & & & -1 \\ \hdashline & & & -1 & & & 4 & -1 & \\ & & & & -1 & & & 4 & -1 \\ & & & & & -1 & & -1 & 4 \end{array}\right]$$



2-D : "*5-point Stencil*"

Cases : Galaxy evolution (N-body)



- Newtonian laws of physics
  - The gravitational force between two bodies of masses m_a & m_b :

$$F = \frac{Gm_a m_b}{r^2}$$

  - Subject to the force, acceleration occurs

$$F = m \times a$$

- Let the time interval be Δt & current velocity v^t, position x^t
  - New velocity v^{t+1} :

$$F = m\frac{v^{t+1}-v^t}{\Delta t} \Rightarrow v^{t+1} = v^t + \frac{F\Delta t}{m}$$

  - New position x^{t+1} :

$$x^{t+1} = x^t + v^{t+1}\,\Delta t$$

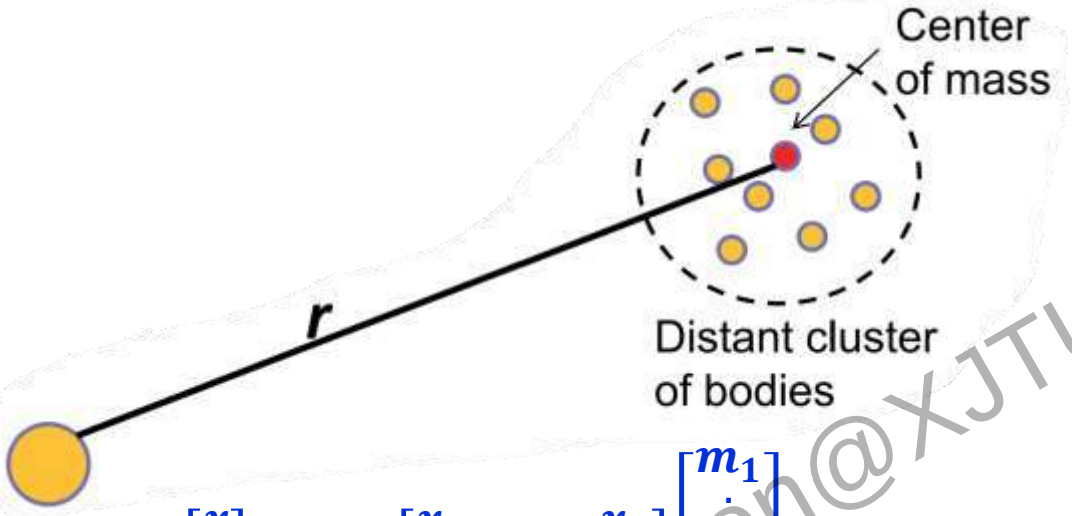
Cases : Galaxy evolution (N-body)

Represent galaxy as a collection of N particles (stars), assume mass is mi :

```
for(t=0; t<T; t++) {
    for(i=0; i<N; i++) {
        F = Compute_Force(i); // compute force in O(N^2)
        v_new[i]=v[i]+F*dt/mi; // compute new velocity
        x_new[i]=x[i]+v_new[i]*dt; // compute new position
    }
    for(i=0; i<N; i++) {
        x[i] = x_new[i]; // update position
        v[i] = v_new[i]; // update velocity
    }
}
```


Non-feasible as N increases due to O(N^2) complexity

Cases : Galaxy evolution (N-body)

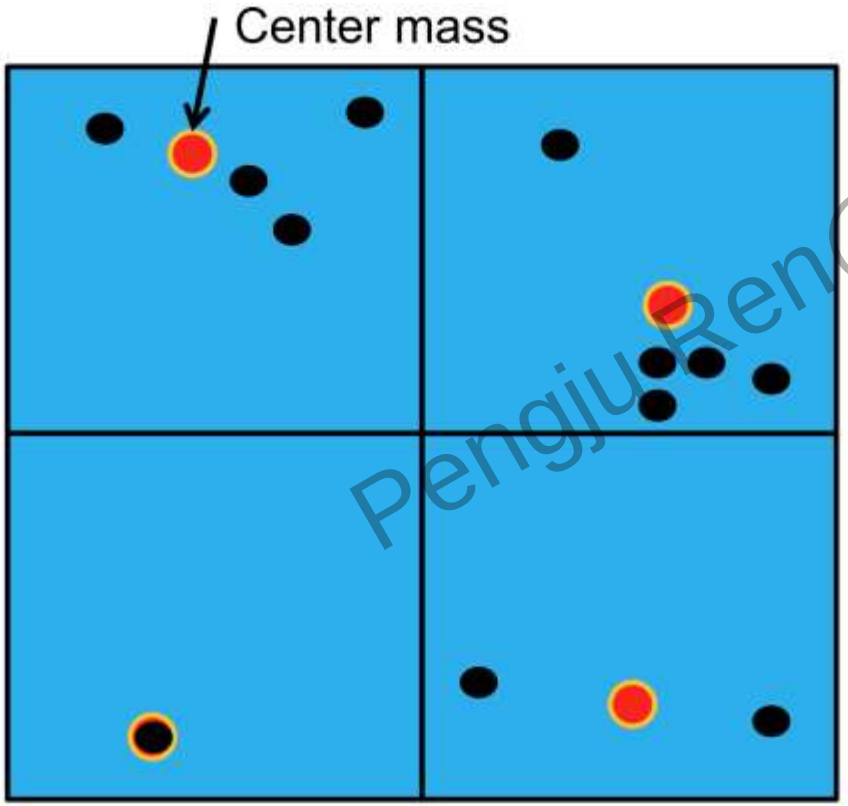


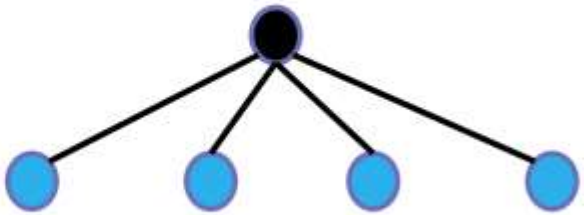
$$\begin{bmatrix} x \\ y \end{bmatrix} = \frac{1}{\sum m_i} \begin{bmatrix} x_1 & \dots & x_n \\ y_1 & \dots & y_n \end{bmatrix} \begin{bmatrix} m_1 \\ \vdots \\ m_n \end{bmatrix},$$ there are *n* particles in the cluster

- Naive algorithm is $O(N^2)$ — all particles interact with all others
- Magnitude of gravitational force falls off with distance, so reduce time complexicity by approximating a cluster of bodies as a single distant body.
- Result is an $O(N \log N)$ algorithm for computing gravitational forces between all stars

Cases : N-body (Barnes-hut algorithm)

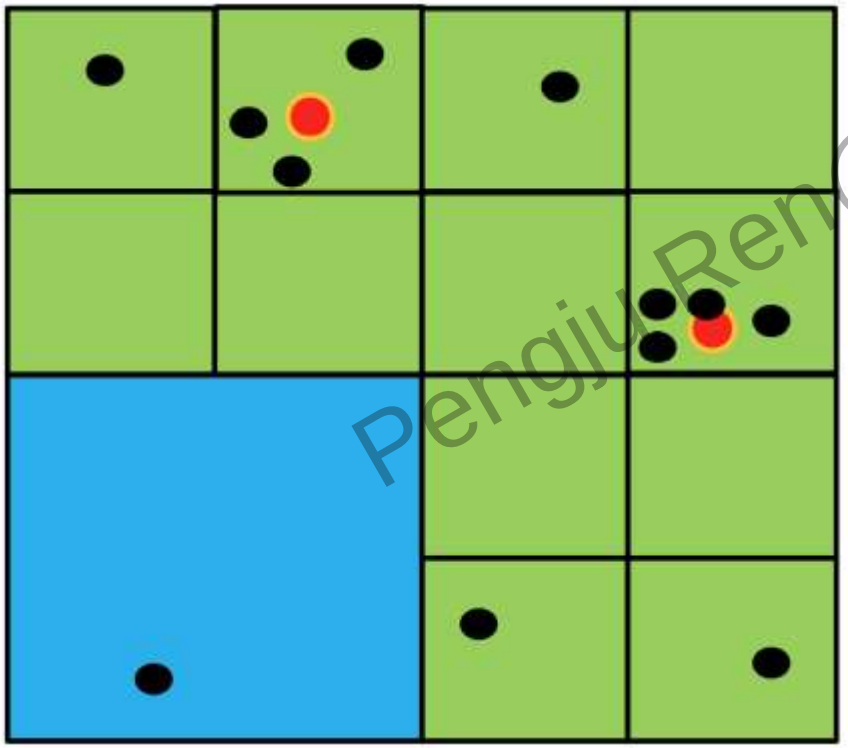
Step1: Recursively divide space by two in each dimensions
Record the center mass and position of each internal node

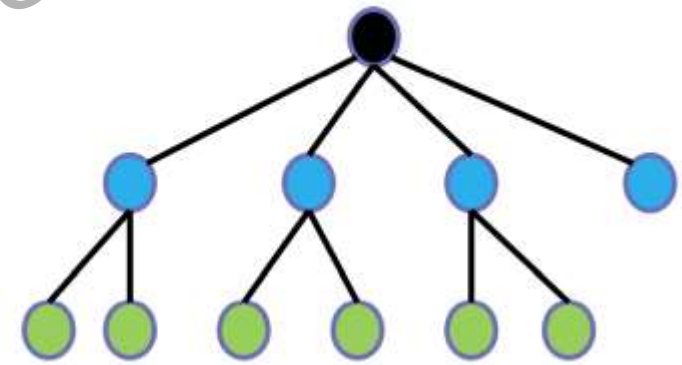




Cases : N-body (Barnes-hut algorithm)

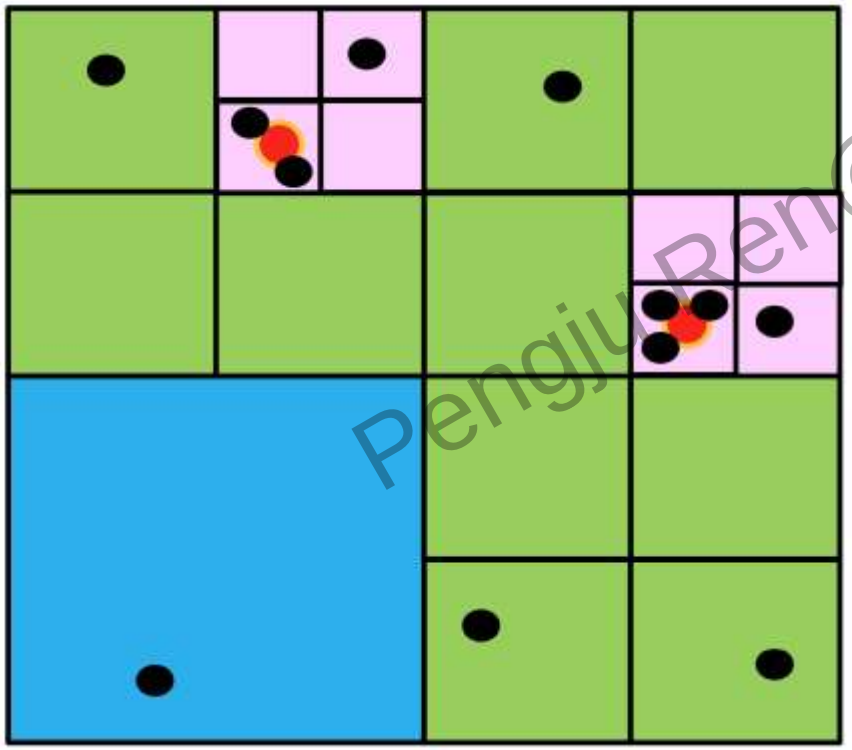
Step1: Recursively divide space by two in each dimensions
Record the center mass and position of each internal node

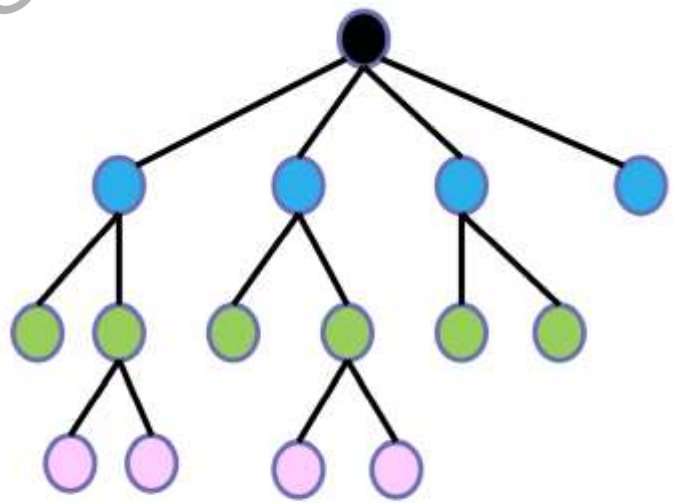




Cases : N-body (Barnes-hut algorithm)

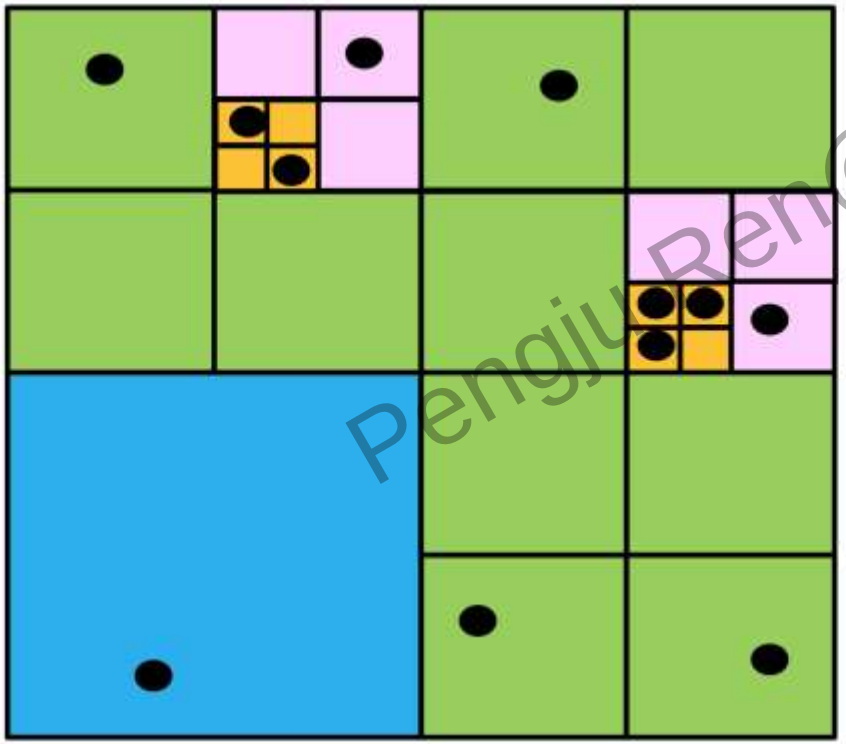
Step1: Recursively divide space by two in each dimensions
Record the center mass and position of each internal node

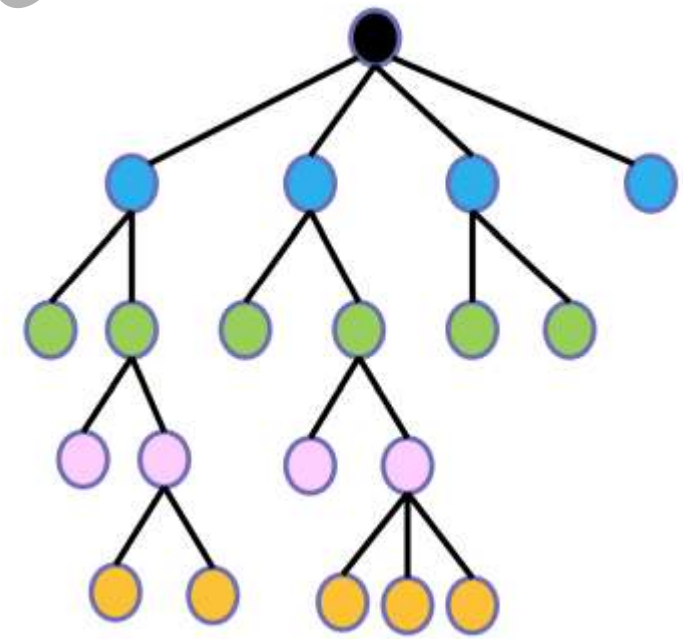




Cases : N-body (Barnes-hut algorithm)

Step1: Recursively divide space by two in each dimensions
Record the center mass and position of each internal node





Cases : N-body (Barnes-hut algorithm)

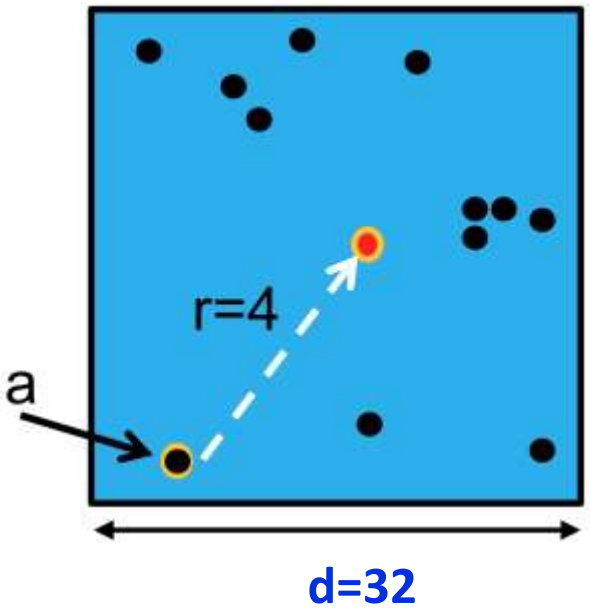
Step2: Compute ***approximate forces*** on each object

1. traverse the nodes of the tree, starting from the root.
2. If the center-of-mass of an internal node is *sufficiently far* from the body, approximate the internal node as a single body

Far means: **d/r< θ** (e.t. **0< θ<1)**

r: the distance between the body and the node's center-of-mass

d: the width of the region



Cases : N-body (Barnes-hut algorithm)

Step2: Compute ***approximate forces*** on each object

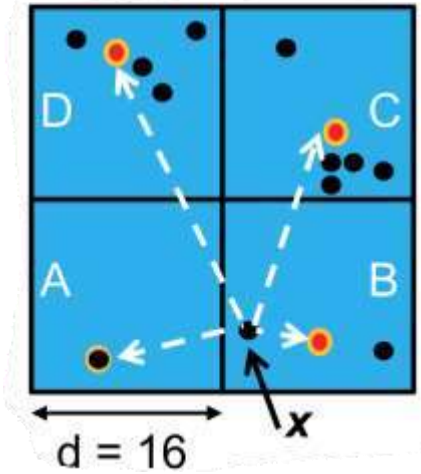
1. traverse the nodes of the tree, starting from the root.
2. If the center-of-mass of an internal node is *sufficiently far* from the body, approximate the internal node as a single body

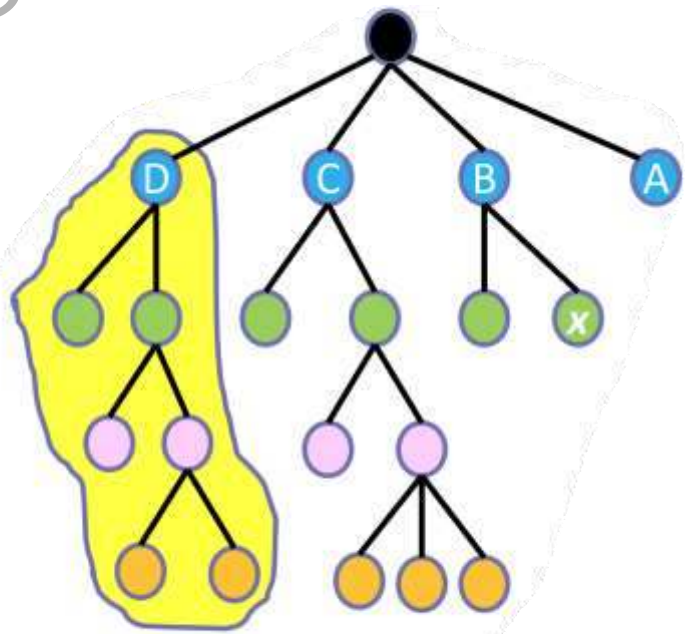
Far means: **d/r< θ** (e.t. **0< θ<1)**

r: the distance between the body and the node's center-of-mass

d: the width of the region

d/r_A=16/10 > θ
d/r_B=16/2 > θ
d/r_C=16/15 > θ
d/r_D=16/20 < θ

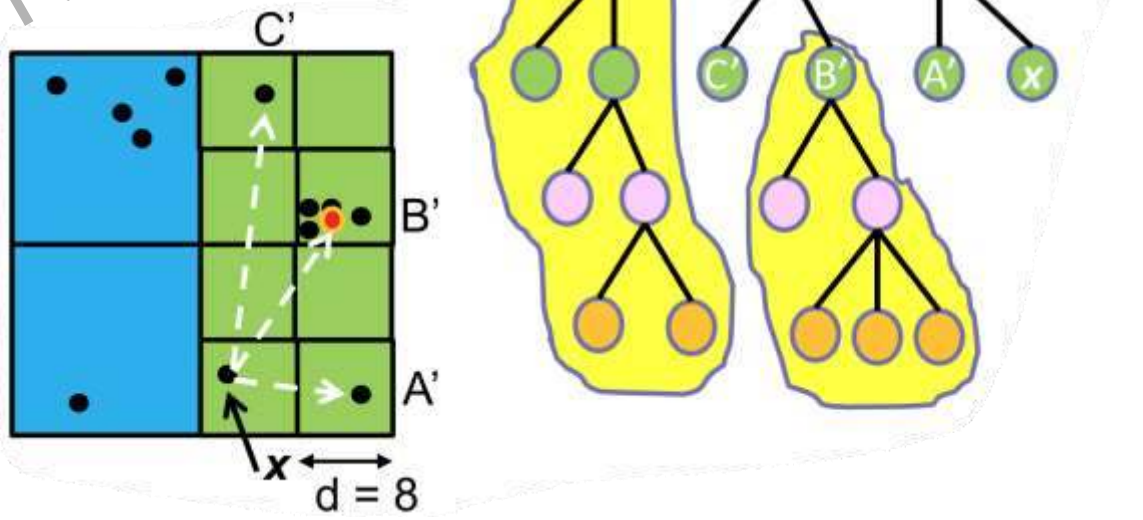




Cases : N-body (Barnes-hut algorithm)

Step2: Compute ***approximate forces*** on each object

3. If it is a leaf node, calculate the force and add to the object.
4. Otherwise, recursively compute the force from children of the internal node



$d/r_{A'}$= 8/7 > θ

$d/r_{B'}$= 8/15 < θ

$d/r_{C'}$= 8/20 < θ

A' and C' are leaf nodes, B' treated like a single node

Cases : N-body (Barnes-hut algorithm)

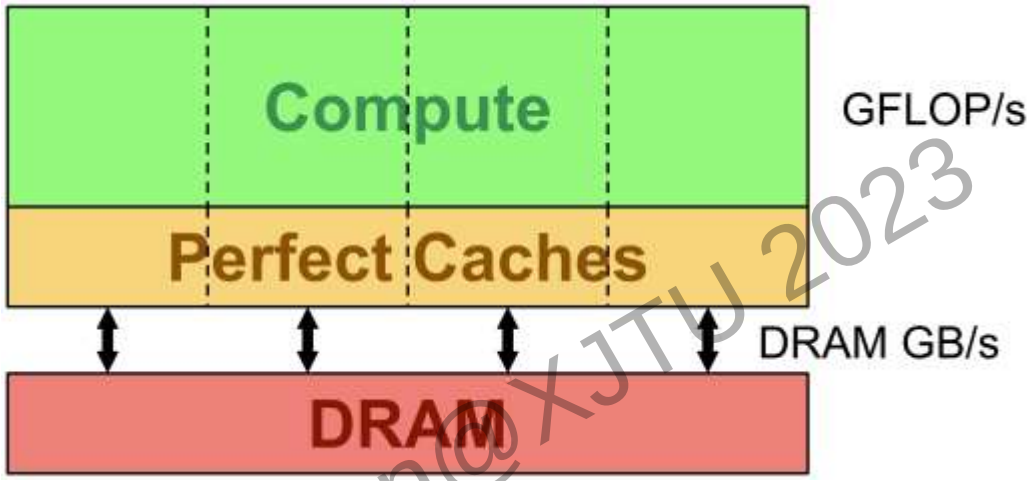
- θ controls the accuracy and approximation error of the algorithm
  - θ = 0 -> d/r ALWAYS larger than θ -> same as brute force
  - θ = 1 -> most likely only need to consider the object within the same cluster/region
- If
  -
- The tree must be re-built for each time interval

How to speedup N-body kernel ?
(Load balancing and Data Locality)


```
for each time step in simulation:
   build tree structure compute(aggregate mass, center-of-
mass) for interior nodes
   for each particle:
       traverse tree to accumulate gravitational forces
       update particle position based on gravitational forces
```

https://www.cs.utexas.edu/~pingali/CS378/2011sp/papers/nbody.pdf

Recap: Data Movement or Compute



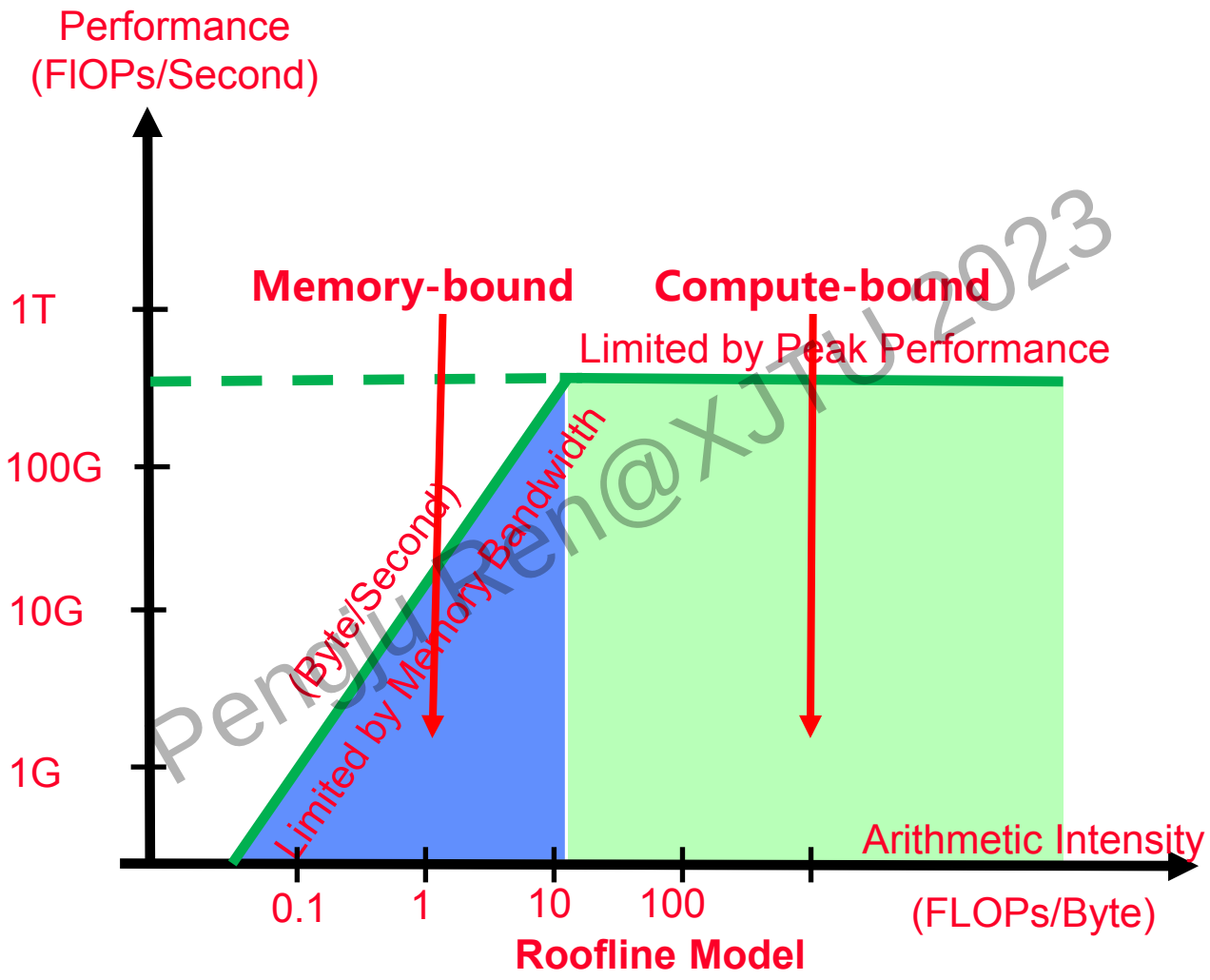
Which takes longer? Data movement or Compute?

Time = max (#FP ops/Peak GFLOP/s, #Bytes/Peak GB/s)

#FP ops/Time = min (Peak GFLOP/s, (#FP ops/#Bytes)*Peak GB/s)

AI (Arithmetic or Computational Intensity)

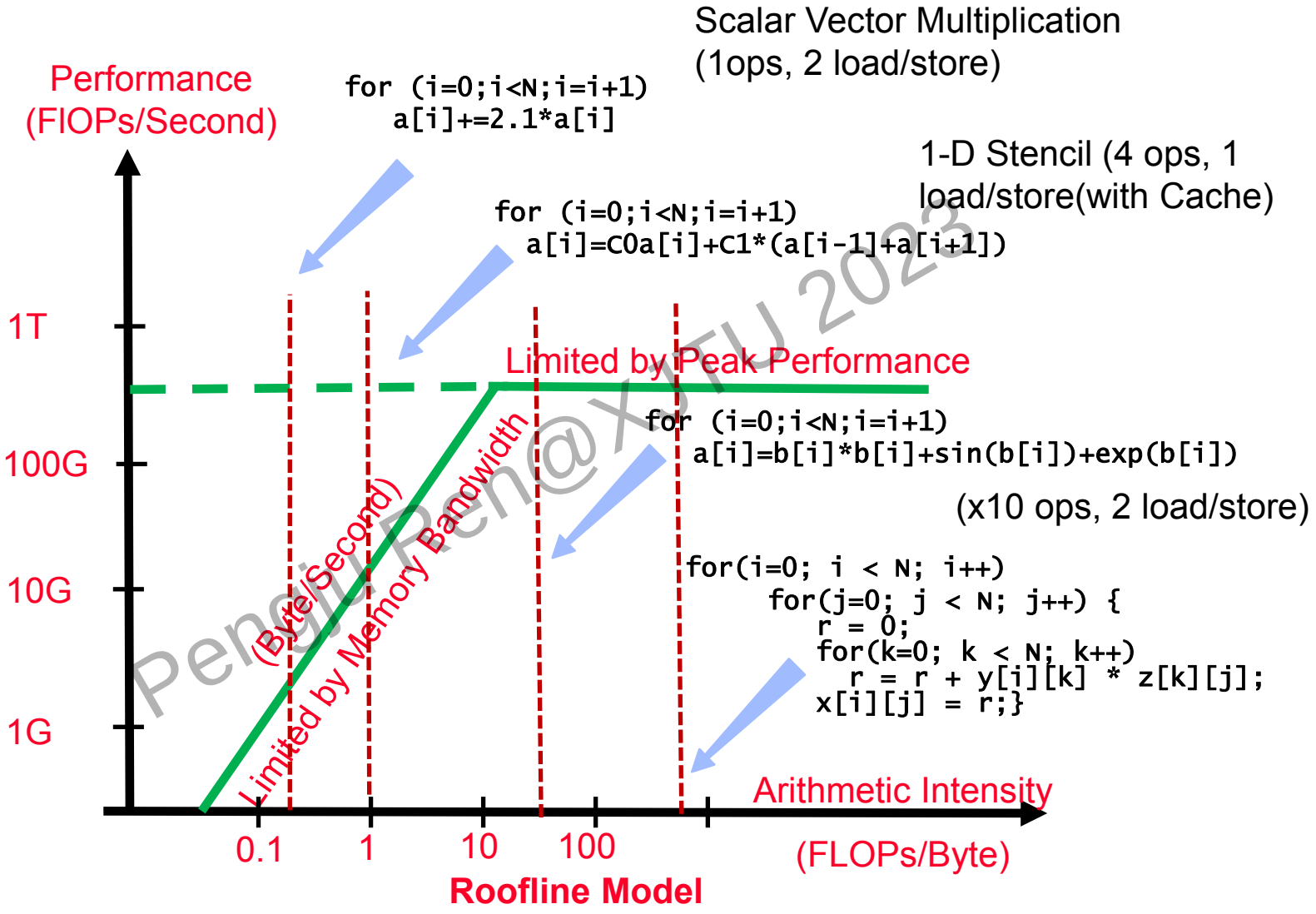
Roofline Model



#FP ops/Time = min (Peak GFLOP/s, AI*Peak GB/s)

Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: an insightful visual performance model for multicore architectures. Commun. ACM 52, 4

Roofline Model



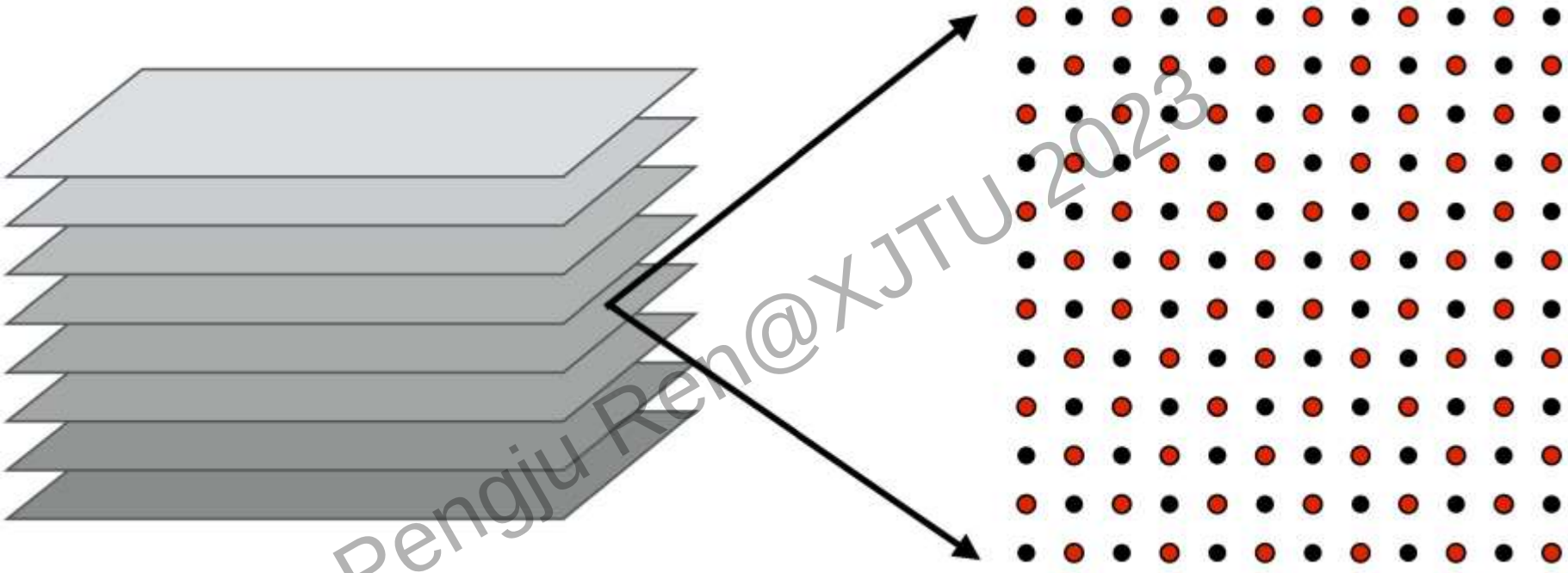
Summary of Lecture 1-4

- Details of machine are important for performance
  - Processor and memory system (not just parallelism)
  - What to expect? Use understanding of hardware limits
- There is parallelism hidden within processors
  - Pipelining, OoO, SIMD, SMT, VLIW, etc
- Machines have memory hierarchies
  - 100s of cycles to read from DRAM (main memory)
  - Caches are fast (small) memory that optimize average case
- Locality is at least as important as computation
  - Temporal: re-use of data recently used
  - Spatial: using data nearby to recently used data
- Can rearrange or prefetch code/data to improve locality
  - Goal: minimize communication = data movement

Next Lecture : Systolic Array

Appendix: Data Movement and Programming (remote memory access)

Simulating of ocean currents (2D stencil)



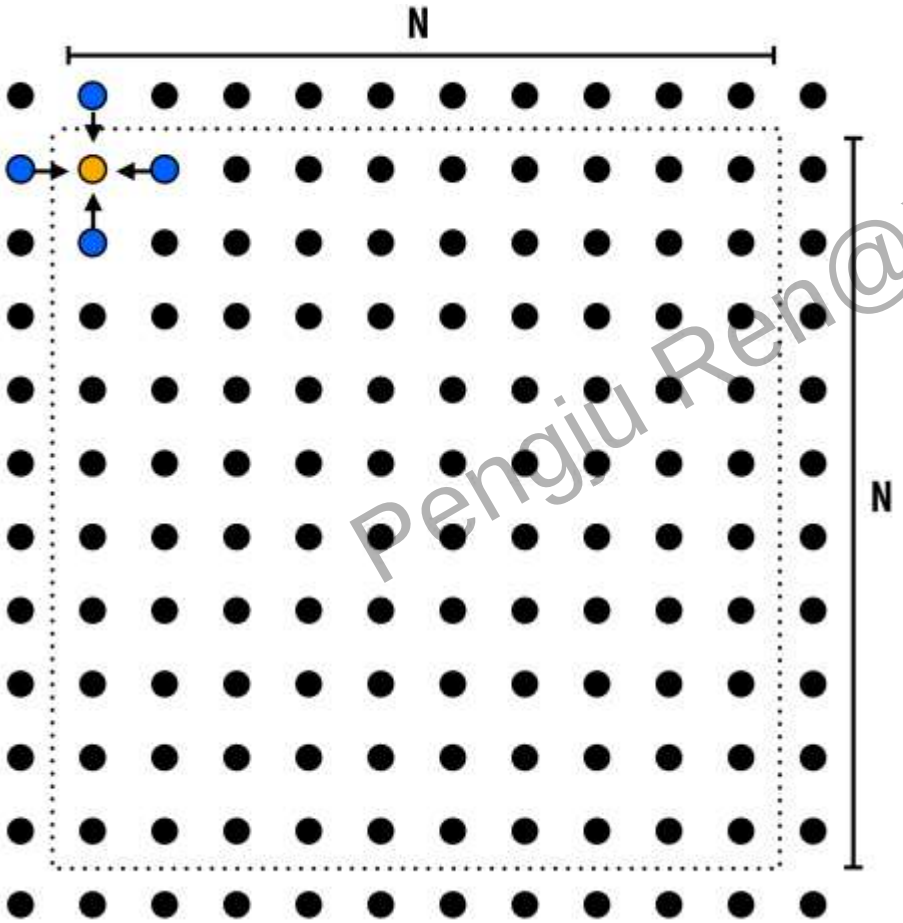
Discretize 3D ocean volume into slices represented as 2D grids

Discretize time evolution of ocean: Δt

High accuracy simulation requires *small Δt* and *high resolution grids*

A 2D-grid based solver

Solve partial differential equation (PDE) on (N+2) x (N+2) grid
Iterative solution
- Perform Gauss-Seidel sweeps over grid until convergence

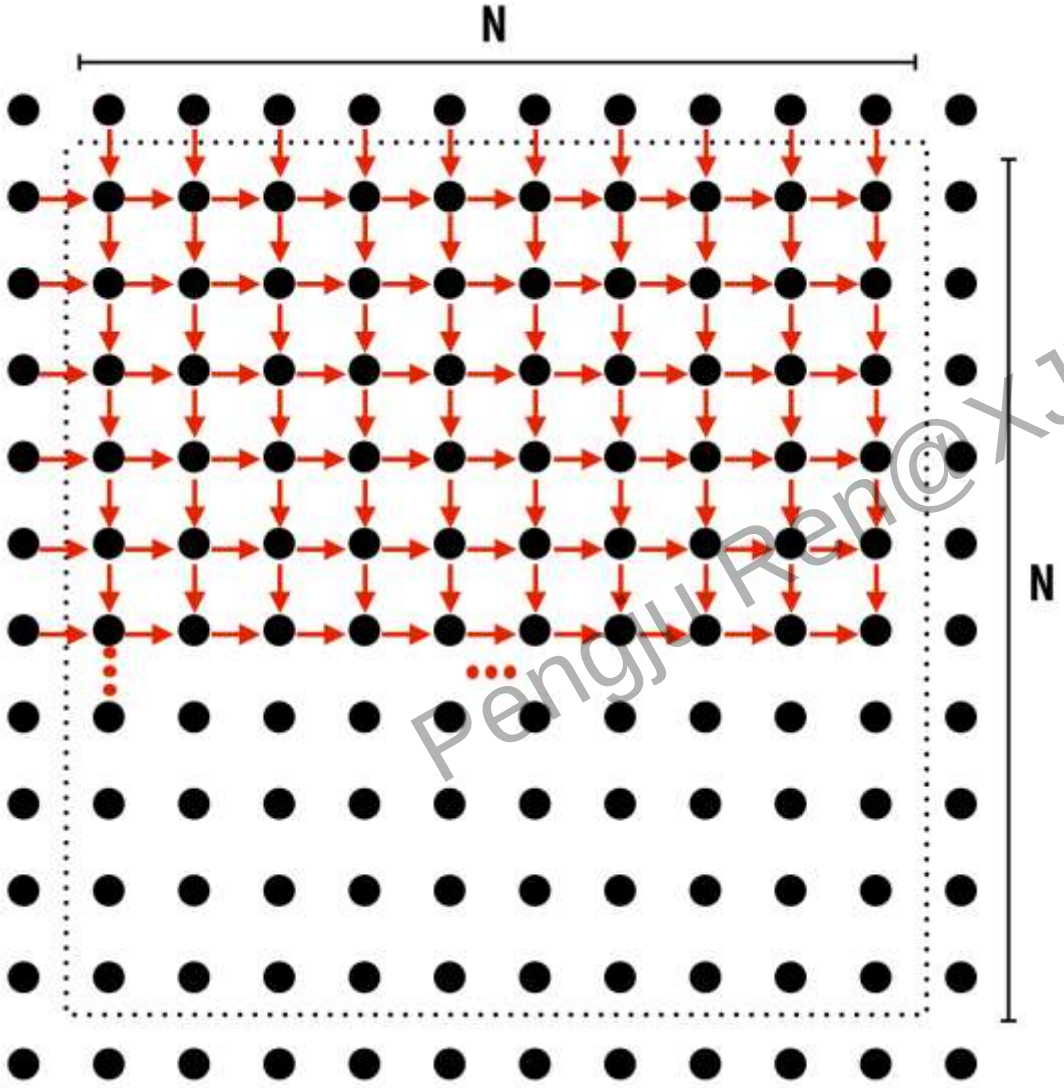


$$A[i,j] = 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j] + A[i, j+1] + A[i+1, j+1])$$

Grid solver algorithm

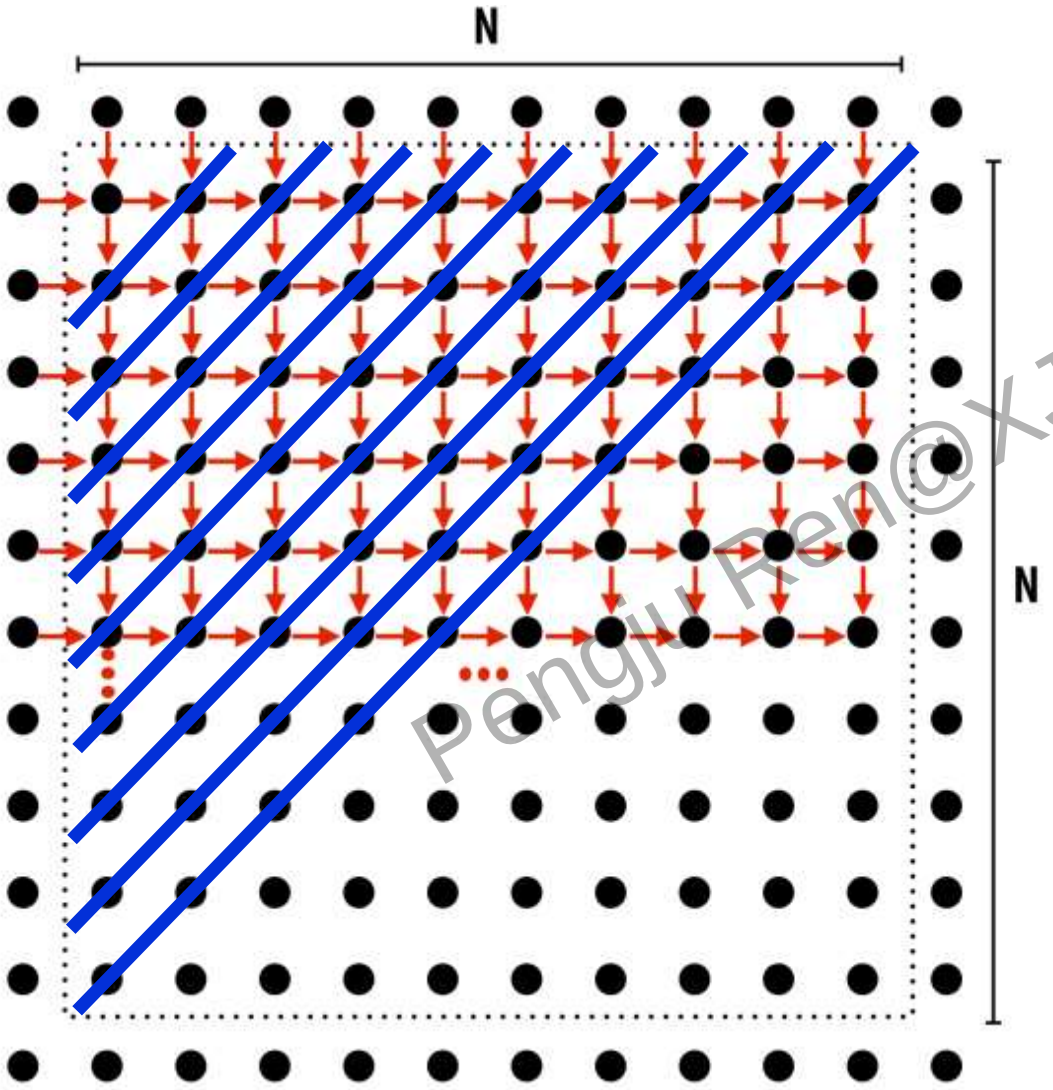
```
const int n;
float* A;                       // assume allocated to grid of N+2 x N+2 elements
void solve(float* A) {
      float diff, prev;
      bool done = false;

      while (!done) {           // outermost loop: iterations
      diff = 0.f;
      for (int i=1; i<n; i++) {  // iterate over non-border points of grid
            for (int j=1; j<n; j++) {
                prev = A[i,j];
                A[i,j] = 0.2* (A[i,j] + A[i-1,j] + A[i,j-1] + A[i+1, j] + A[i, j+1]);
                diff += fabs (A[I,j] – prev);  // compute amount of change
             }
         }
       if (diff/(n*n) < TOLERANCE)     // quit if converged
          done = true;
      }
}
```

Step1: identify dependencies(problem decomposition)



- Each row element depends on element to left.
- Each column depends on previous column.

Step1: identify dependencies(problem decomposition)



There is independent work along the diagonals!

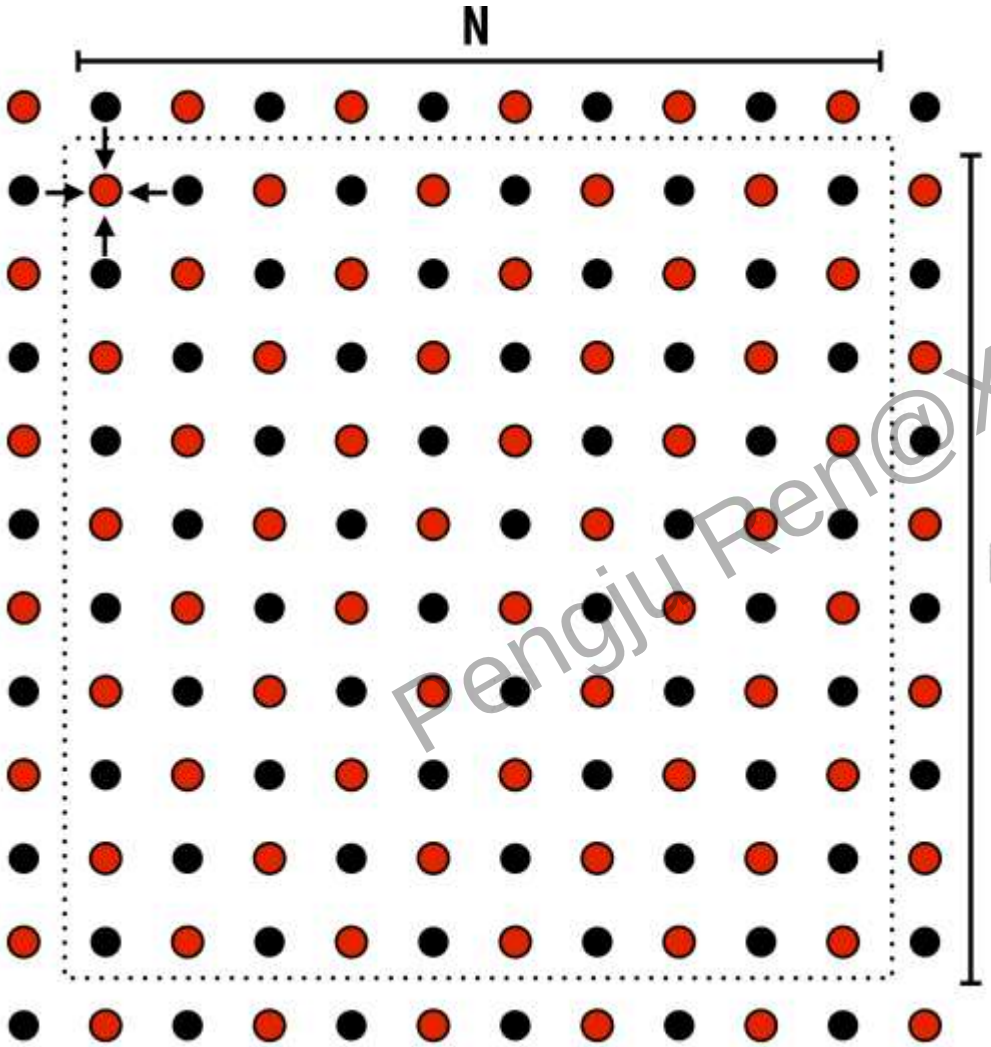
Good: parallelism exists!

Possible implementation strategy:
1. Partition grid cells on a diagonal into tasks
2. Update values in parallel
3. When complete, move to next diagonal

Bad: independent work is hard to exploit

- Not much parallelism at beginning and end of computation.
- Frequent synchronization (after completing each diagonal)

New approach: reorder grid cell update via "red-black" coloring

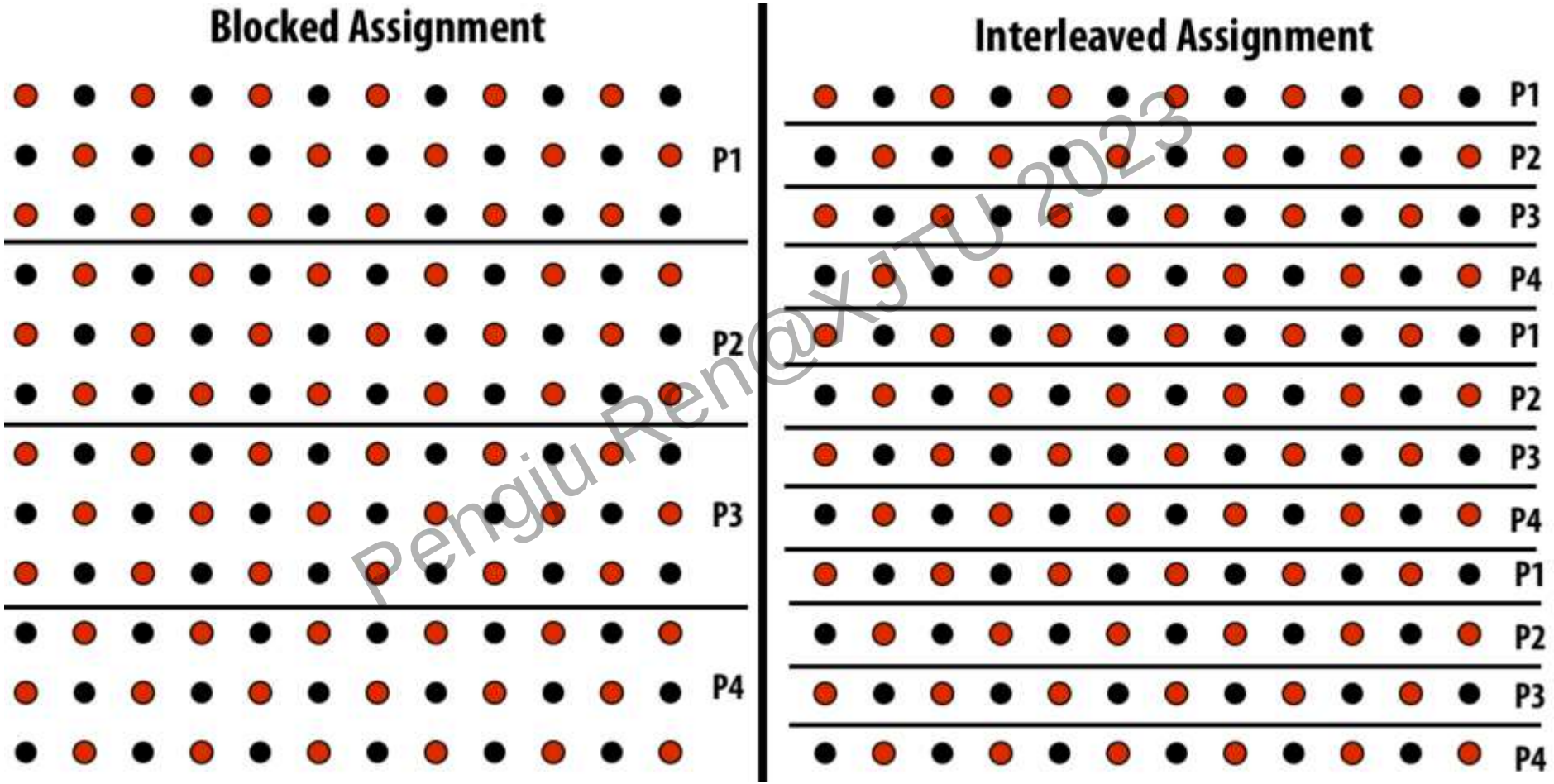


Update all red cells in parallel

When done updating red cells , update all black cells in parallel (respect dependency on red cells)

Repeat until convergence

Possible assignments of work to processors



Which is better ? Does it matter ?

Consider dependencies (data flow)

1.Perform *red* update in parallel

2.Wait until all processors done with update

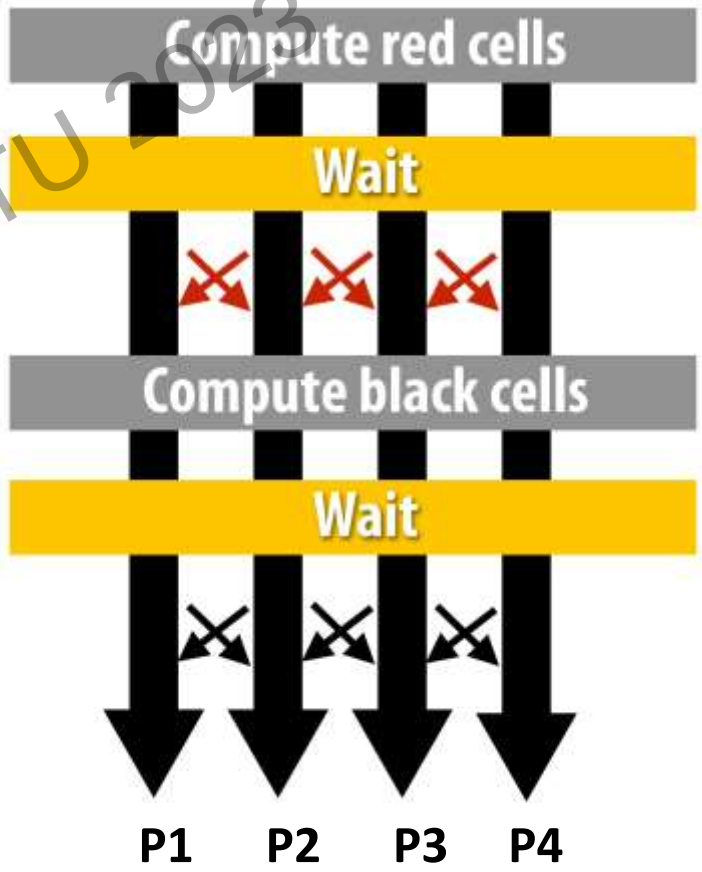
3.Communicate updated *red* cells to other processors

4.Perform *black* update in parallel

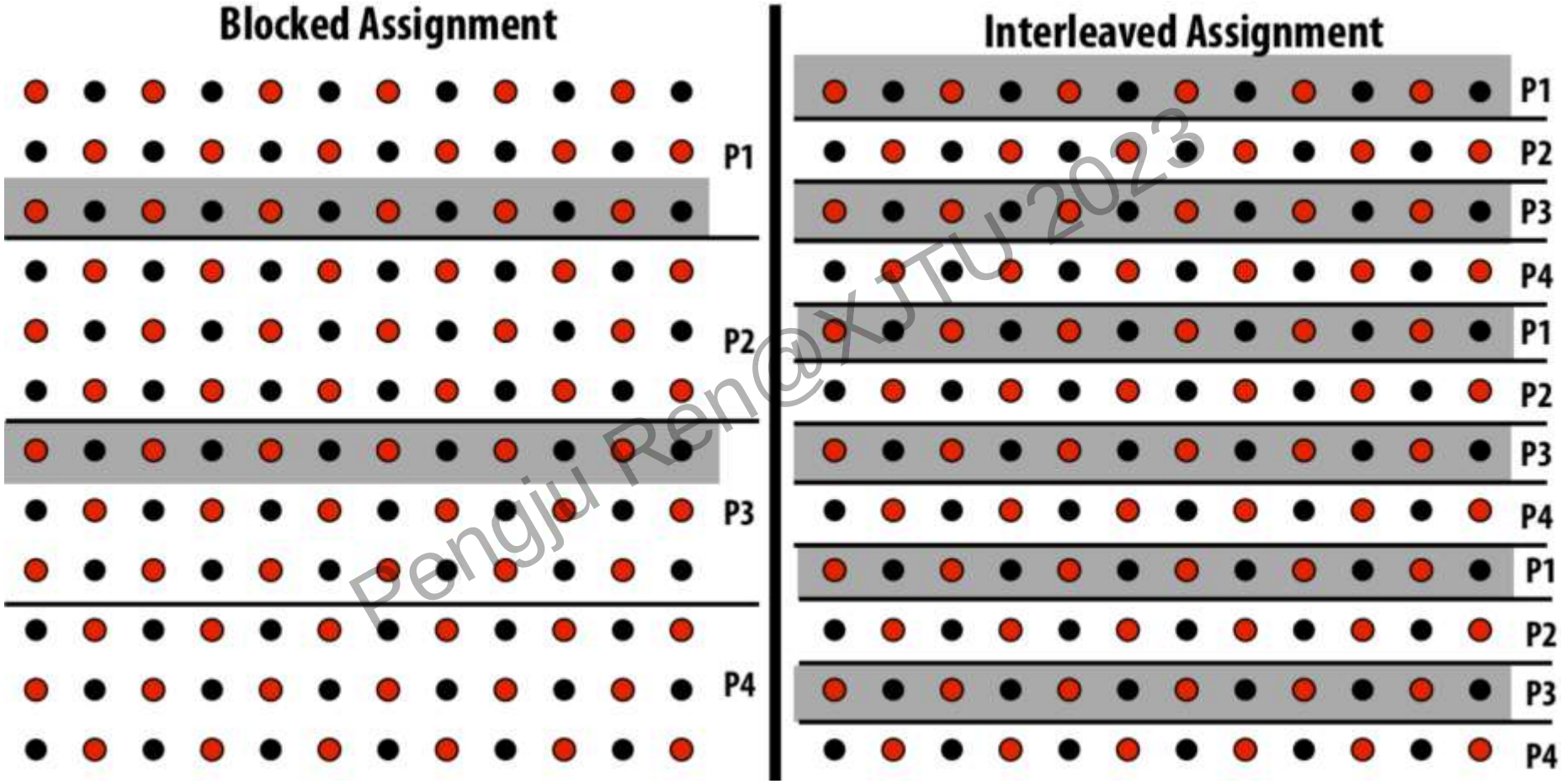
5.Wait until all processors done with update

6.Communicate updated *black* cells to other processors

7.Repeat



Communication resulting from assignment



data that must be sent to others each iteration

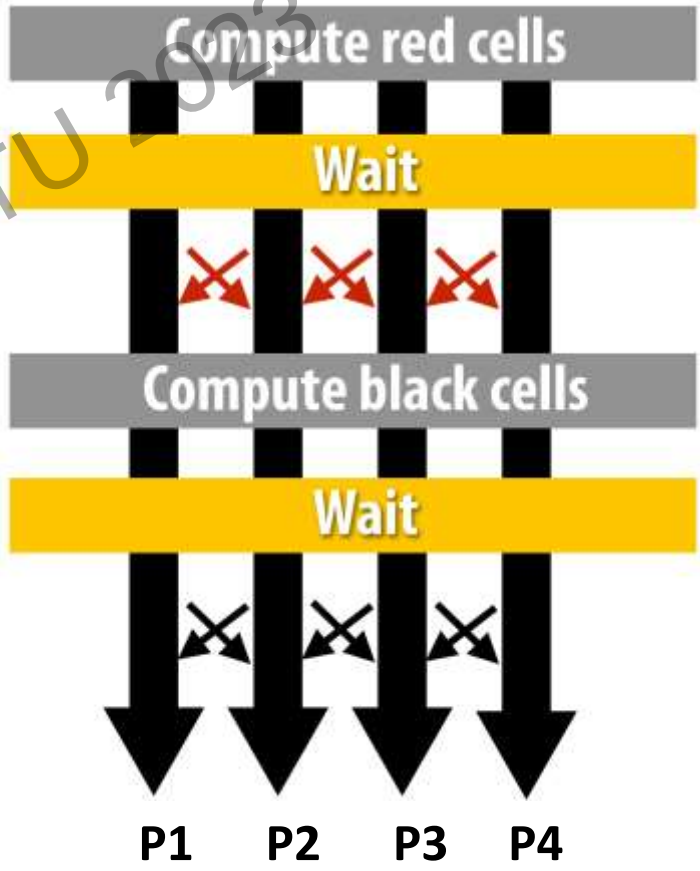
Blocked assignment requires less data to be communicated between processors

Shared address space expression of solver

Programmer is responsible for *synchronization*

Common synchronization primitives:

- **Locks (provide mutual exclusion): only one thread in the critical region at a time**
- **Barriers: wait for threads to reach this point**

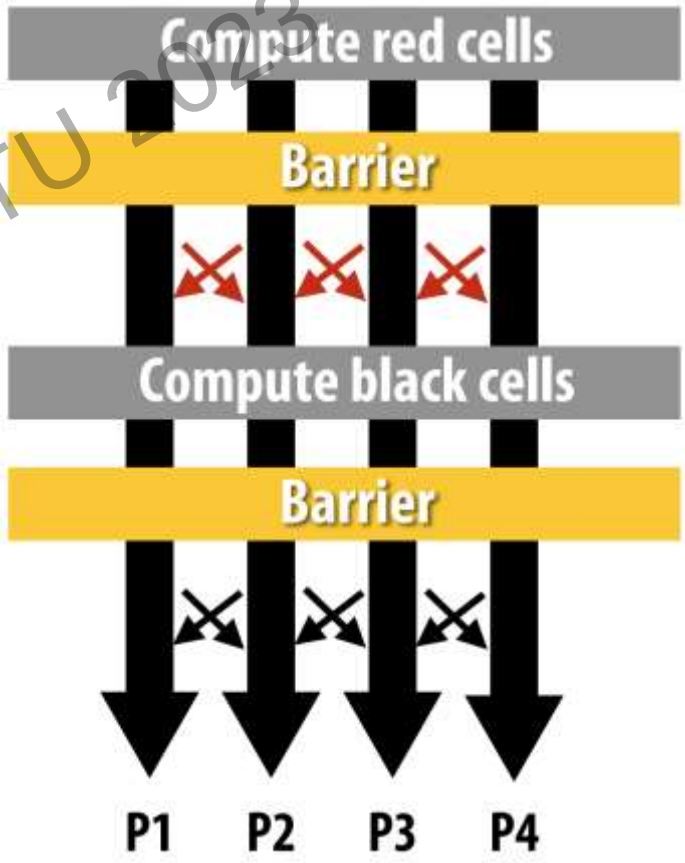


Barrier Synchronization Primitive

barrier(num_threads)

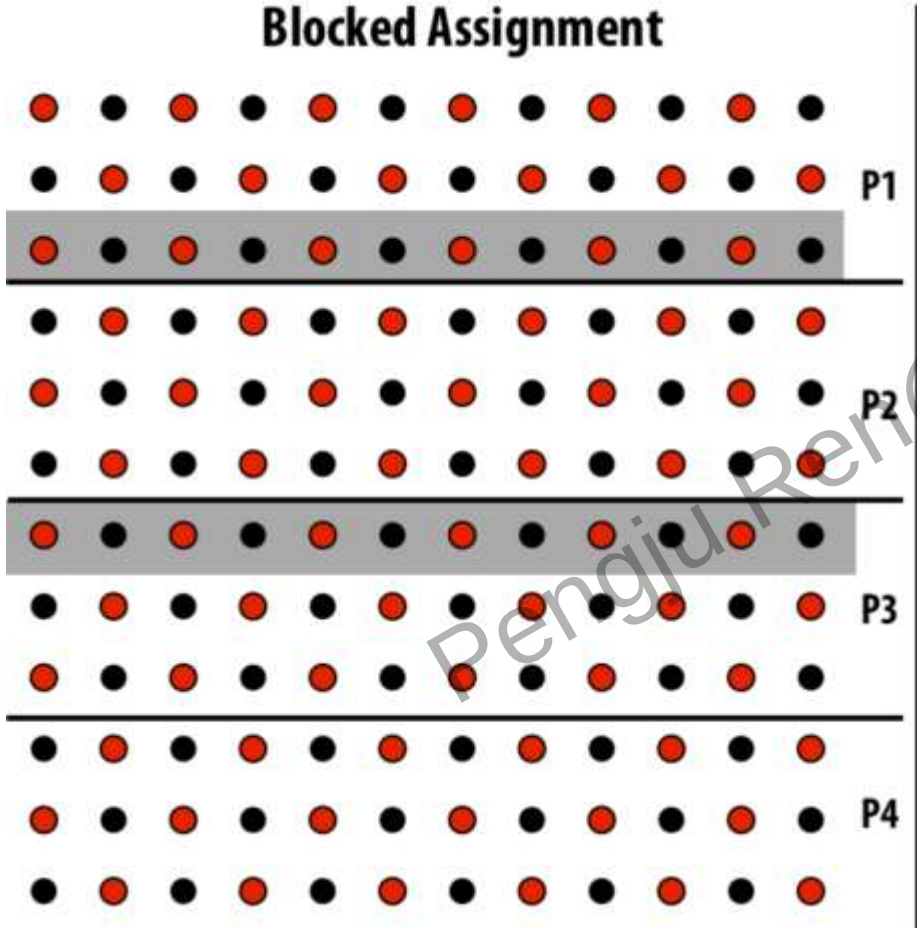
Barriers are a conservative way to express dependencies

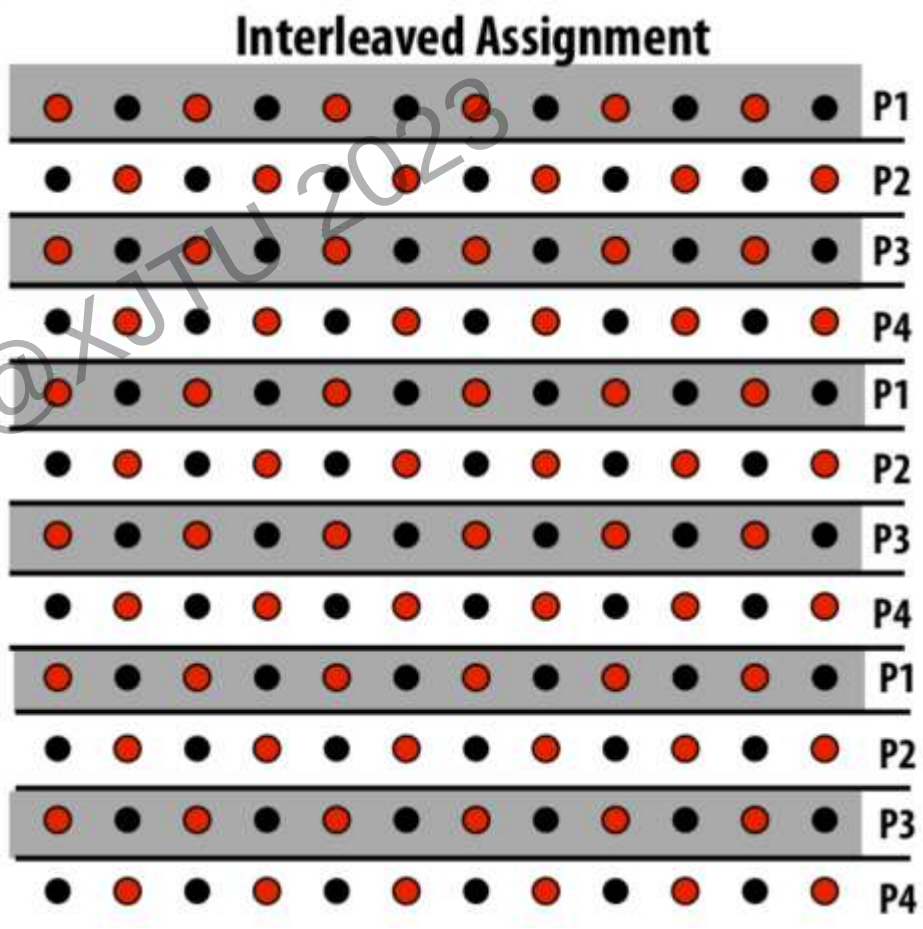
- **Barriers divide computation into phases**
- **All computations by all threads before the barrier complete before any computation in any thread after the barrier begins**



Inherent Communication

For NxN grid and P processor, the Arithmetic intensity are:

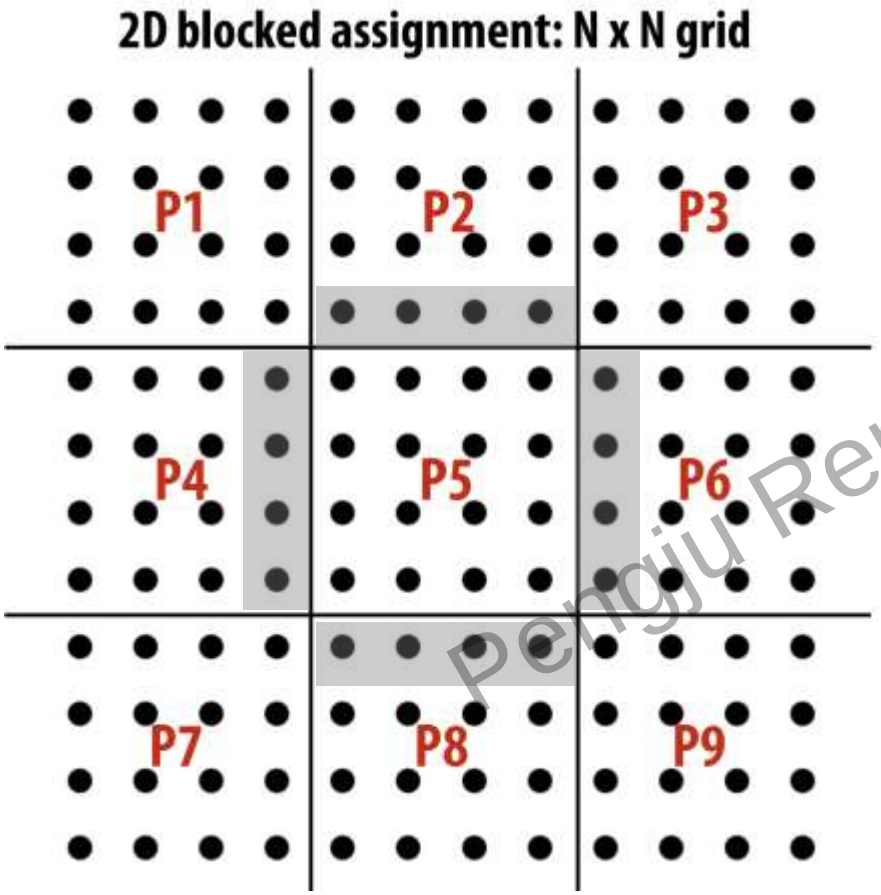




$$\frac{elements\ computed\ (per\ processor) \approx \frac{N^2}{P}}{elements\ communicated\ (per\ processor) \approx 2N} \propto \frac{N}{P}$$

$$\frac{elements\ computed}{elements\ communicated} = \frac{1}{2}$$

Reducing inherent communication



Elements: N^2

Processors: P

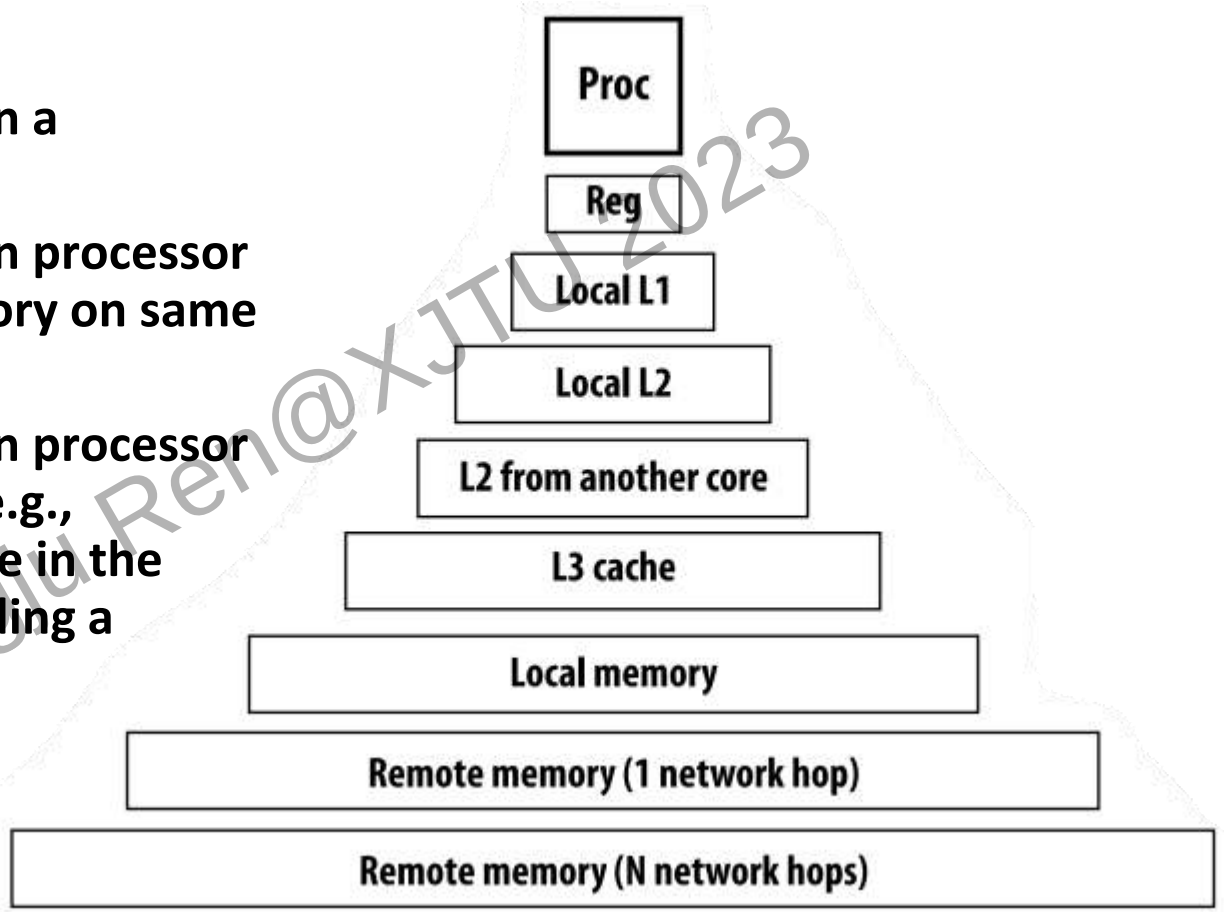
Elements computed (per processor): $\frac{N^2}{P}$

Elements communicated(per processor): $\frac{N}{\sqrt{P}}$

Arithmetic intensity: $\frac{N}{\sqrt{P}}$

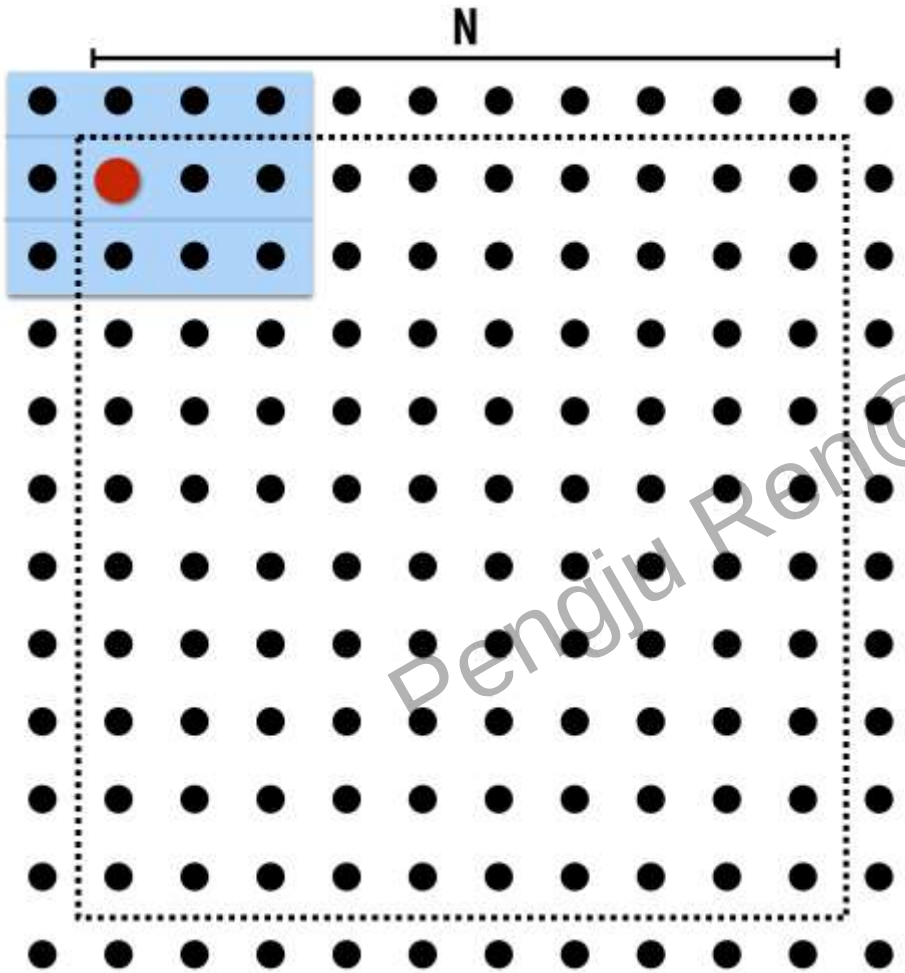
Recap: Think of "Data movement" very generally

- **Data movement** between a processor and its cache
- **Data movement** between processor and memory (e.g., memory on same machine)
- **Data movement** between processor and a remote memory (e.g., memory on another node in the cluster, accessed by sending a network message)



Accesses not satisfied in "local memory" cause communication with "next level"

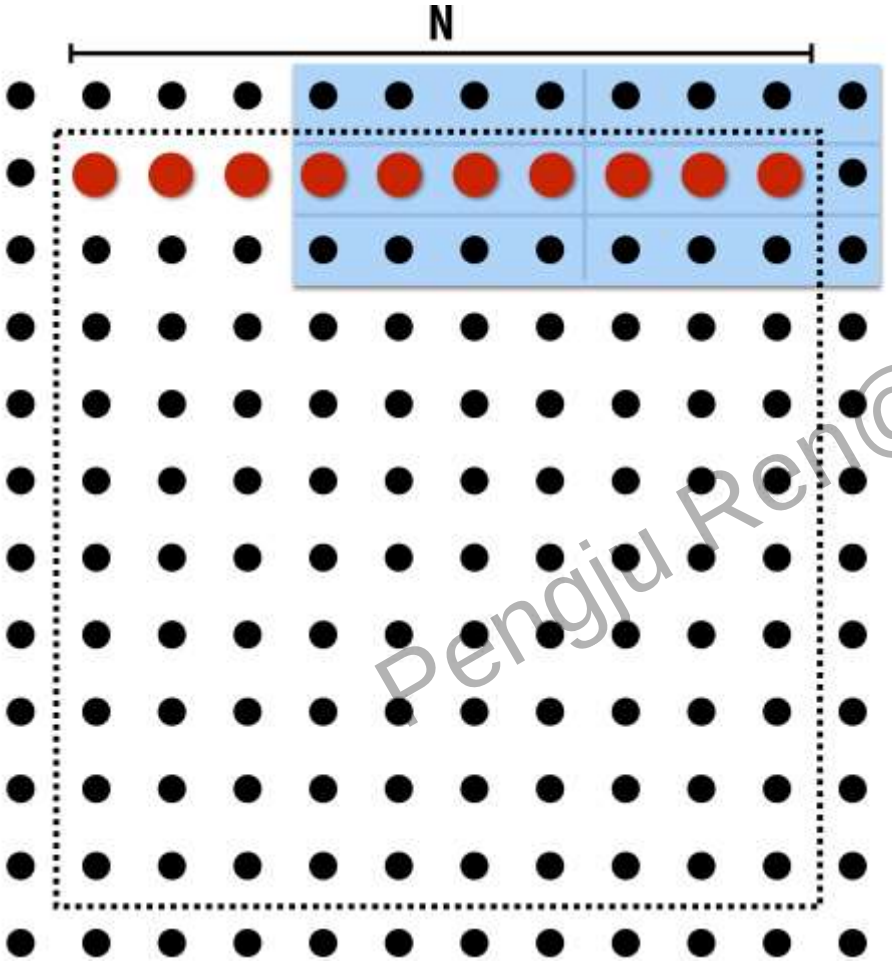
Data access in grid solver: row-major traversal



Assume row-major layout.
Assume cache line is 4 elements.
Cache capacity is 24 elements

Blue elements show data in cache after update to red element.

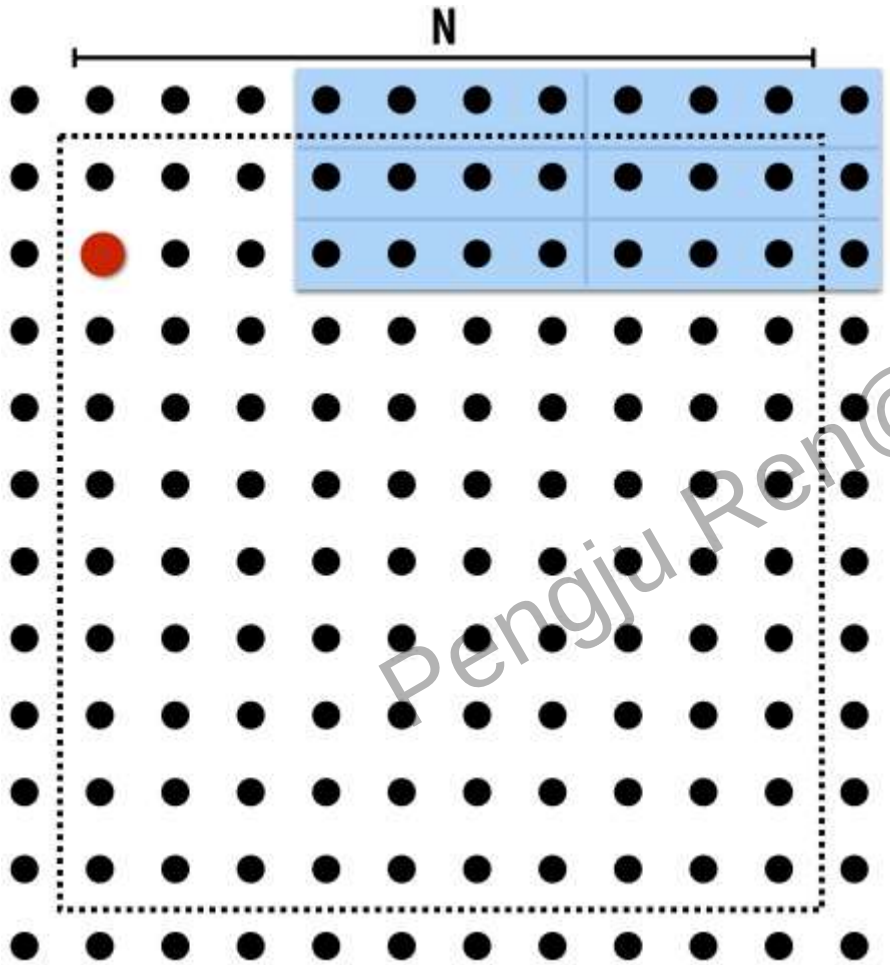
Data access in grid solver: row-major traversal



Assume row-major layout.
Assume cache line is 4 elements.
Cache capacity is 24 elements

Blue elements show data in cache after update to red element.

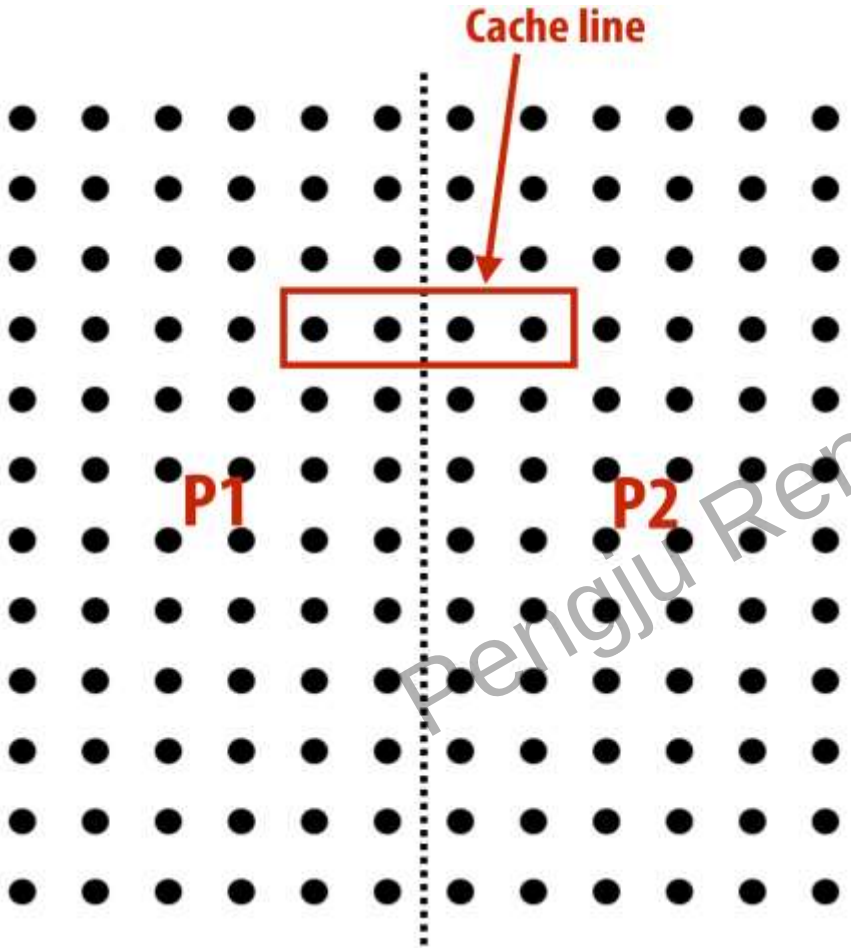
Data access in grid solver: row-major traversal



Assume row-major layout.
Assume cache line is 4 elements.
Cache capacity is 24 elements

Although elements (0,2) and (1,1) had been accessed previously, they are no longer present in cache at start of processing row 2

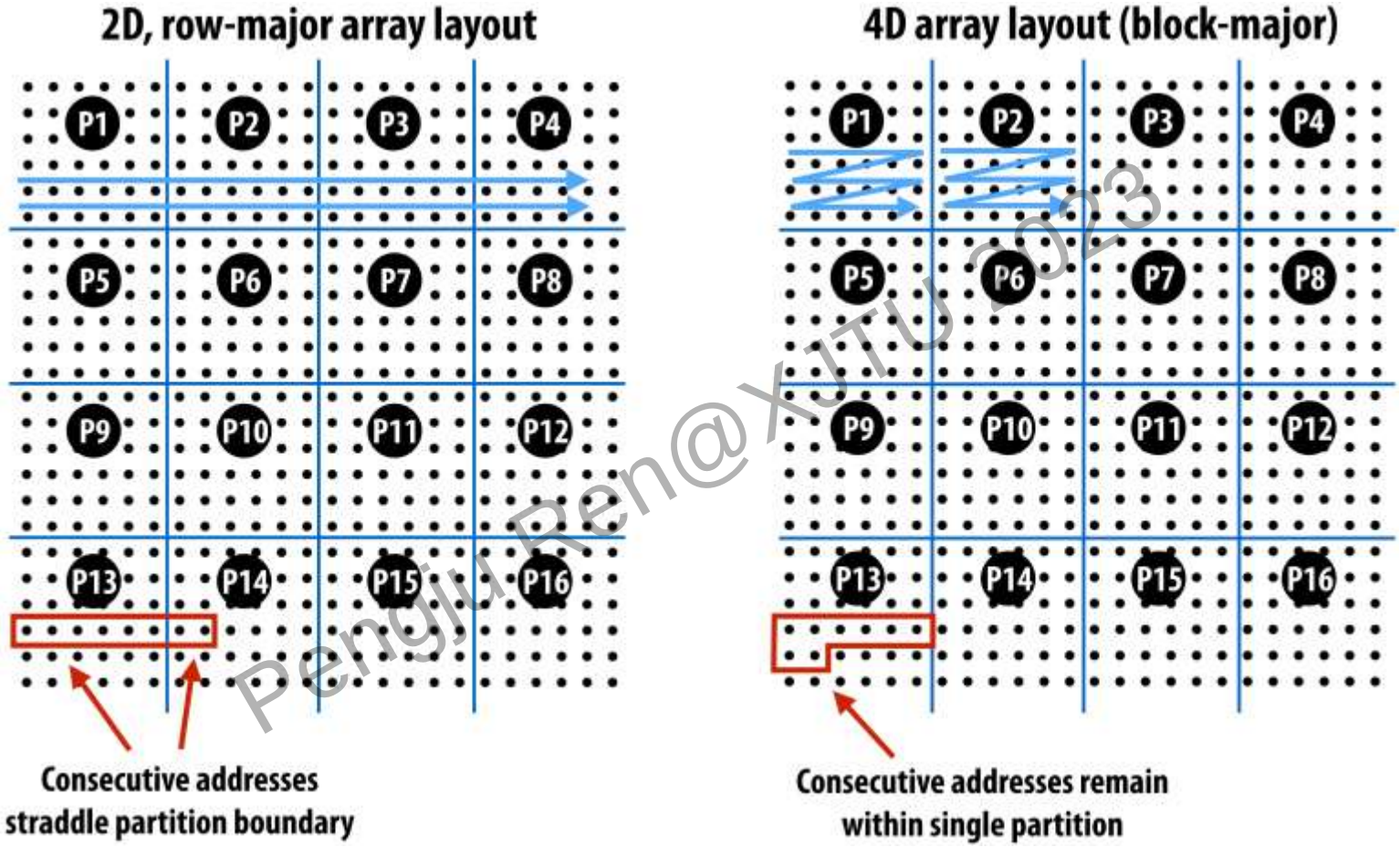
Artifactual communication due to cache line communication granularity



Threads access their assigned elements (no inherent communication exists)

But data access on real machine triggers (artifactual) communication due to the cache line being written to by both processors *

Reducing artifactual comm: blocked data layout



- Blue lines indicate consecutive memory addresses)
- don't confuse blocked assignment of work to threads with blocked data layout in the address space