大数据计算基础

第二章 大数据算法

哈尔滨工业大学
刘显敏
liuxianmin@hit.edu.cn

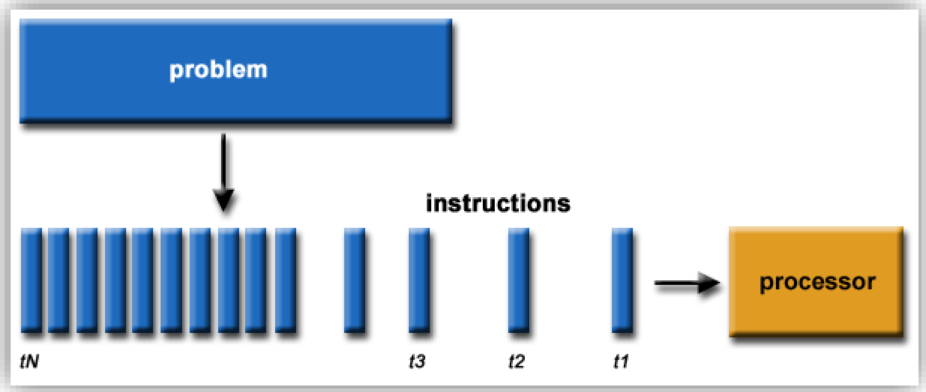
大数据计算基础

第二章 大数据算法——并行算法

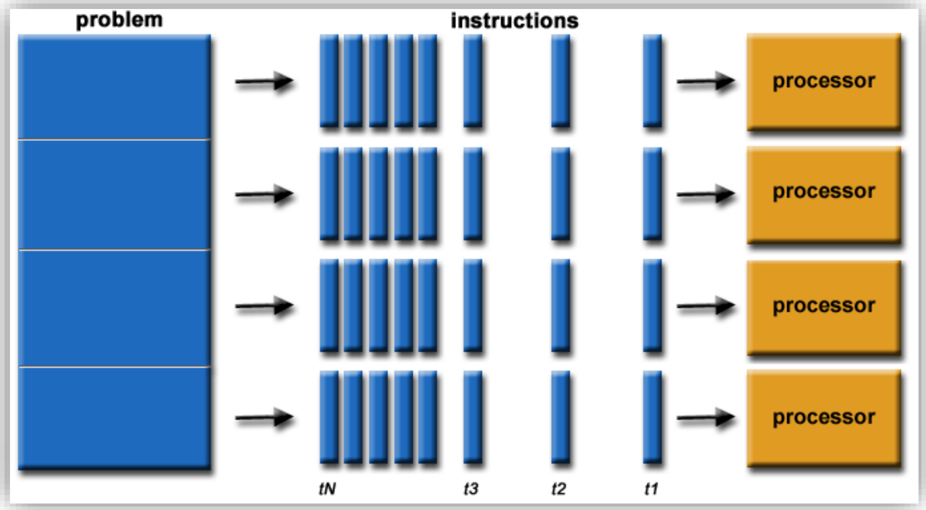
哈尔滨工业大学
刘显敏
liuxianmin@hit.edu.cn

What is Parallel Computing?

- Serial Computing



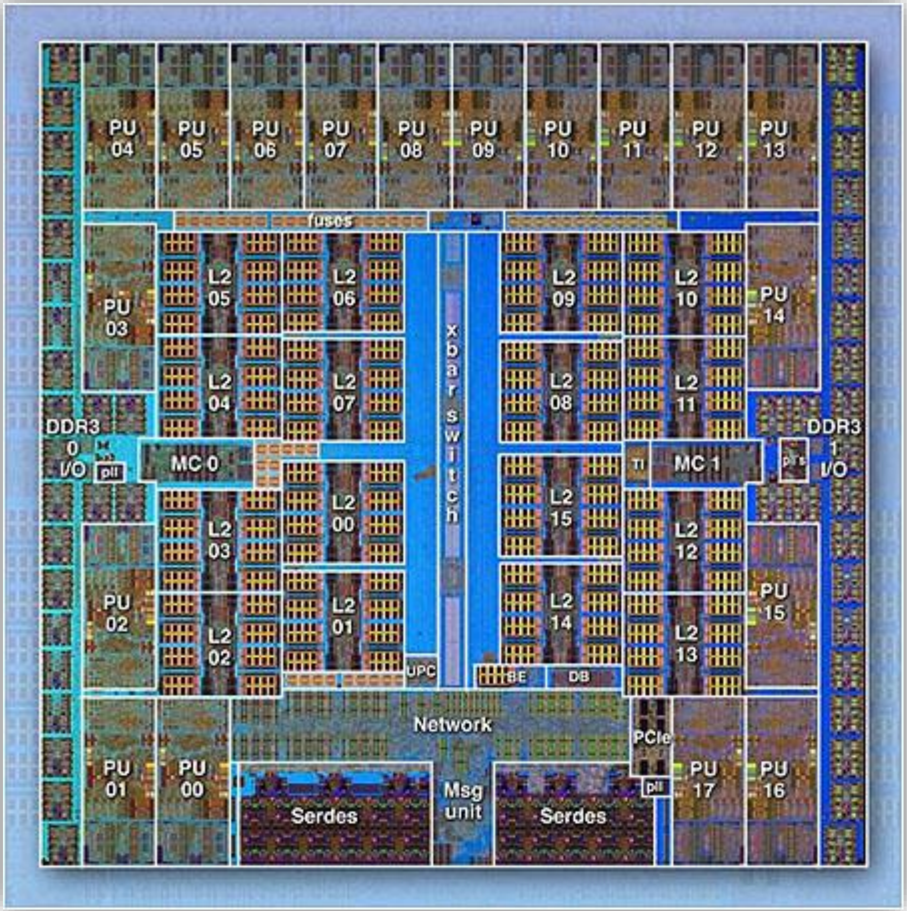
- Parallel Computing



- **parallel computing** is the simultaneous use of multiple compute resources to solve a computational problem:
  - A problem is broken into discrete parts that can be solved concurrently
  - Each part is further broken down to a series of instructions
  - Instructions from each part execute simultaneously on different processors
  - An overall control/coordination mechanism is employed

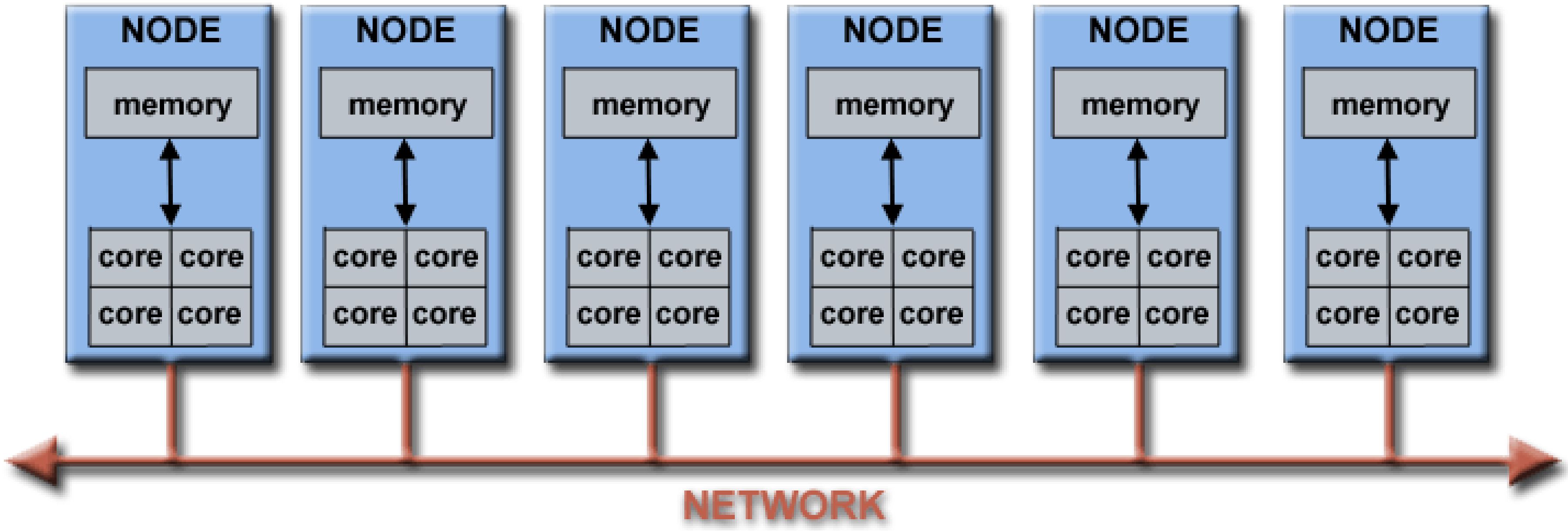
Parallel Computers

- Virtually all standalone computers today are parallel from a hardware perspective:
- IBM BG/Q Compute Chip with 18 cores (PU) and 16 L2 Cache units (L2)



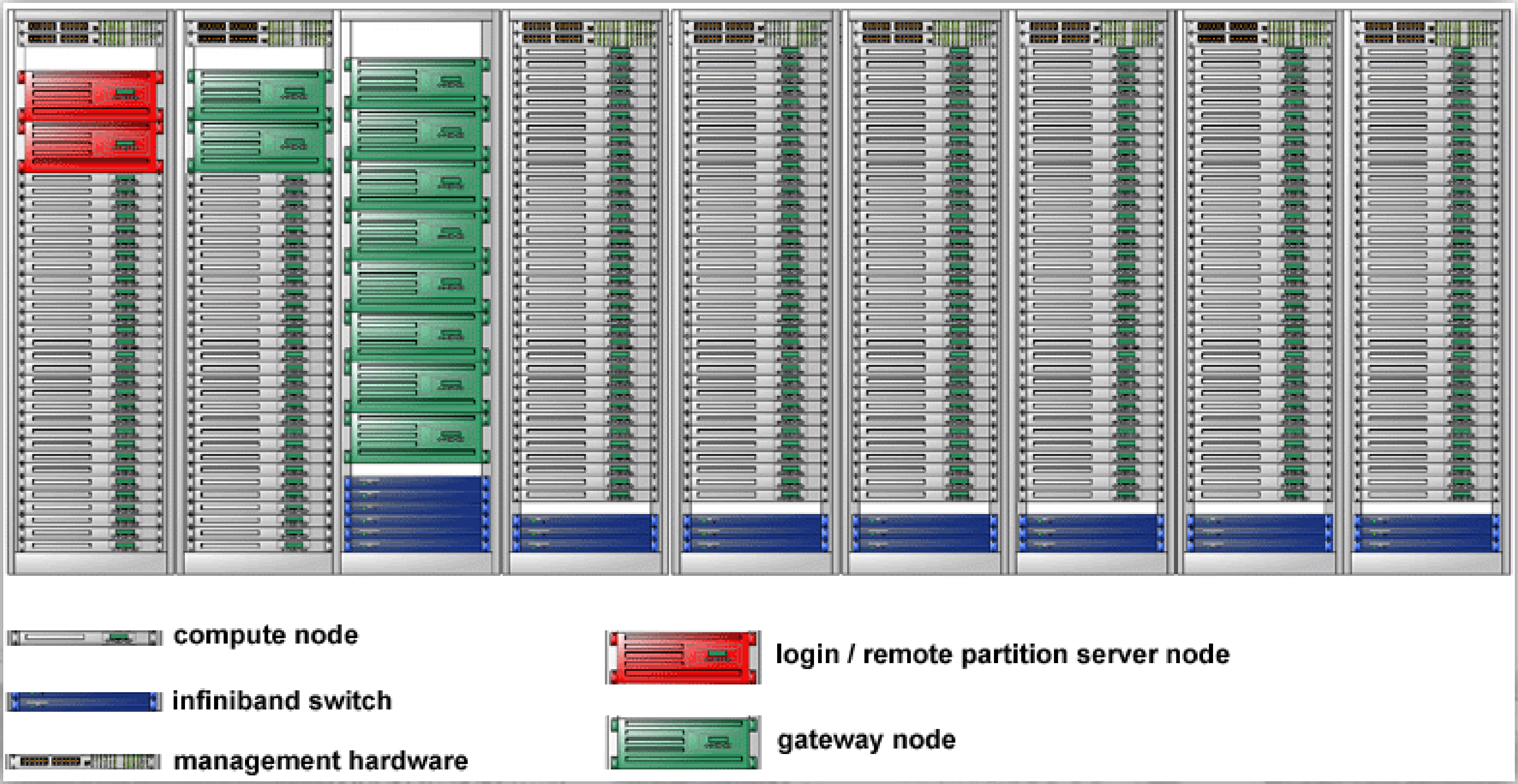
Parallel Computers

- Networks connect multiple stand-alone computers (nodes) to make larger parallel computer clusters.



Parallel Computers

- Networks connect multiple stand-alone computers (nodes) to make larger parallel computer clusters.



Why Use Parallel Computing?

Galaxy Formation

Planetary Movments

Climate Change

Rush Hour Traffic

Plate Tectonics

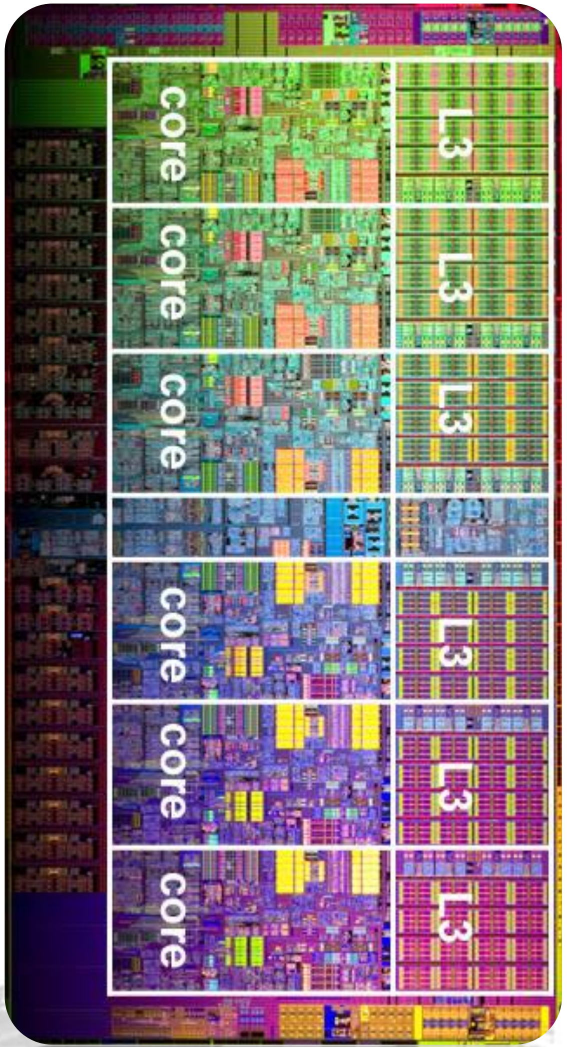
Weather

Auto Assembly

Jet Construction

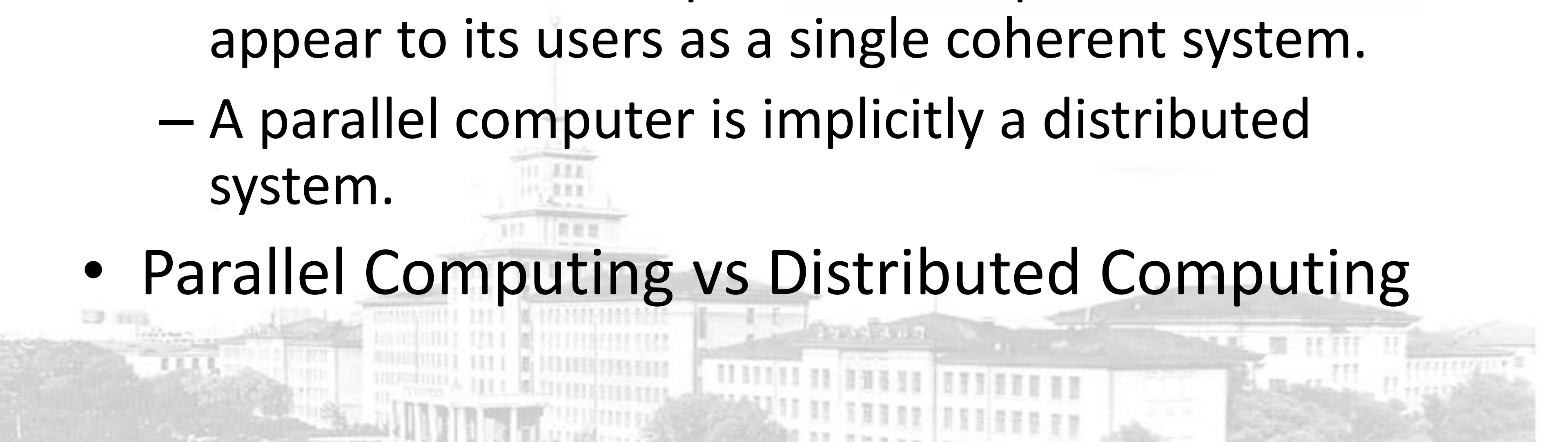
Drive-thru Lunch

Why Use Parallel Computing?



Parallel vs distributed computing

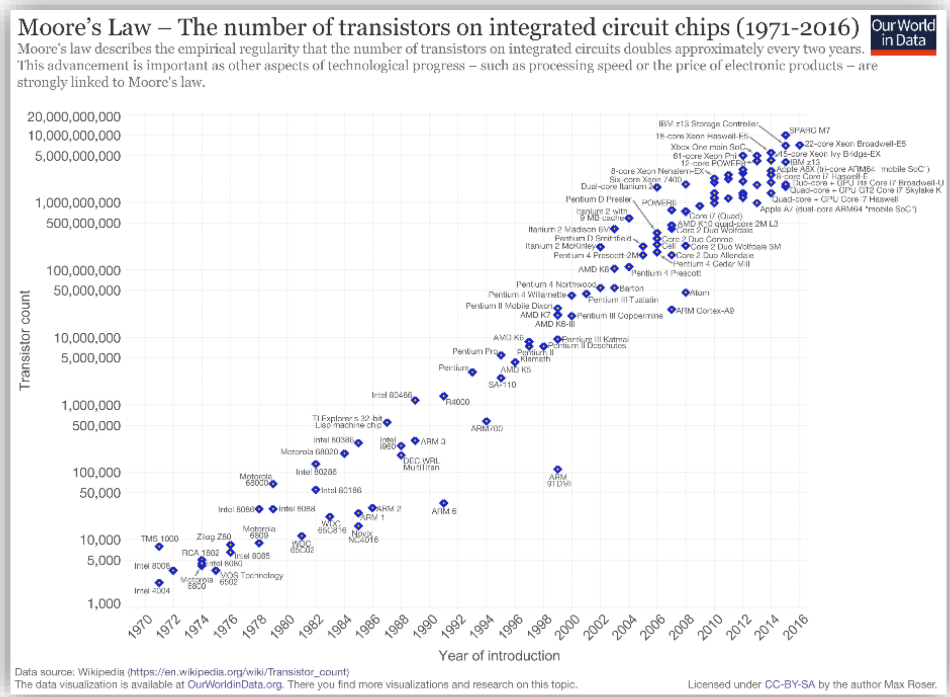
- What is a parallel computer?
  - A collection of processing elements that communicate and coorperate to solve large problems fast.
- What is a distributed system?
  - A collection of independent computers that appear to its users as a single coherent system.
  - A parallel computer is implicitly a distributed system.
- Parallel Computing vs Distributed Computing



Why parallel computing?

- Moore's law: the number of transistors double every 18 months.

- How to make good use of the increasing number of transistors on a chip?
  - Increase the computation width (70's and 80's)
    - 4bit->8bit->16bit->32bit->64bit->....
  - Instruction level parallelism (90's)
    - Pipeline, LIW, etc
  - ILP to the extreme (early 00's)
    - Out of order execution, 6-way issues, etc
  - Sequential program performance keeps increasing.
    - The clock rate keeps increasing, clock cycles get smaller and smaller.


Moore's Law – The number of transistors on integrated circuit chips (1971-2016)

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important as other aspects of technological progress – such as processing speed or the price of electronic products – are strongly linked to Moore's law.

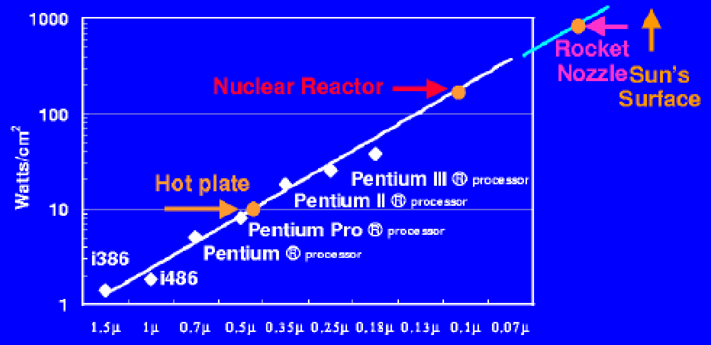
Data source: Wikipedia (https://en.wikipedia.org/wiki/Transistor_count)
The data visualization is available at OurWorldinData.org. There you find more visualizations and research on this topic.

Licensed under CC-BY-SA by the author Max Roser.

Why parallel computing?

- The fundament limit of the speed of a sequential processor.
  - Power wall (high frequency results in heat)
  - Latency wall (speed of light does not change)

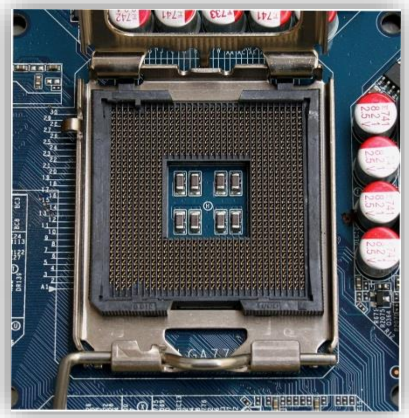
Power density continues to get worse



Surpassed hot-plate power density in 0.5μ

Not too long to reach nuclear reactor

Micro32
Fred Pollack

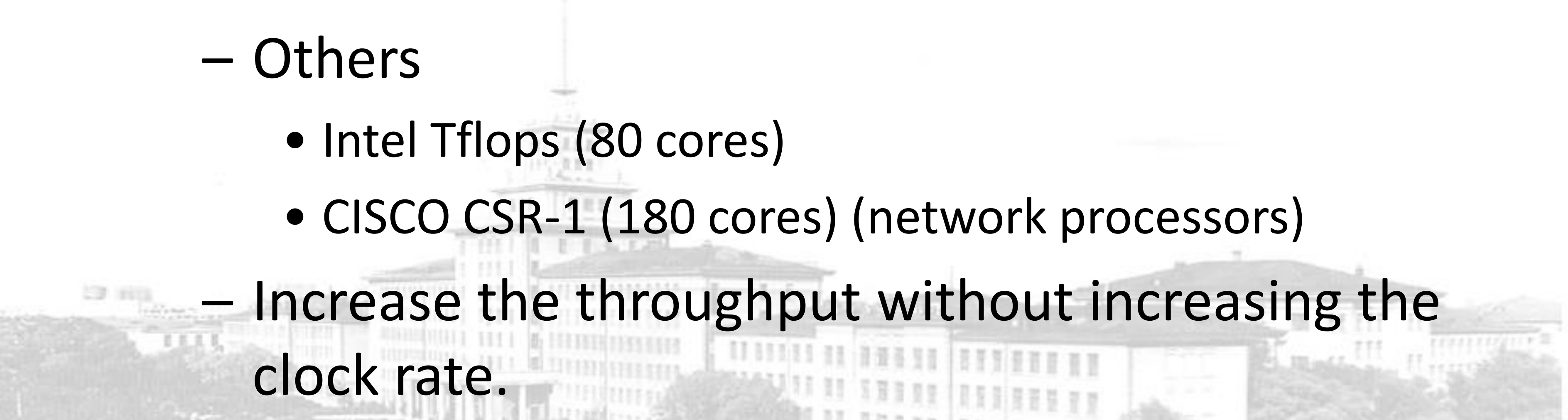


Intel chip dimension
= 1.47 in x 1.47 in
= 3.73cm x 3.73cm

- Speed of light = 300000000m/s
- One cycle at 4Ghz = 0.00000000025s
- The distance that the light can move at one cycle:
  - 0.00000000025 * 300000000 = 7.5cm

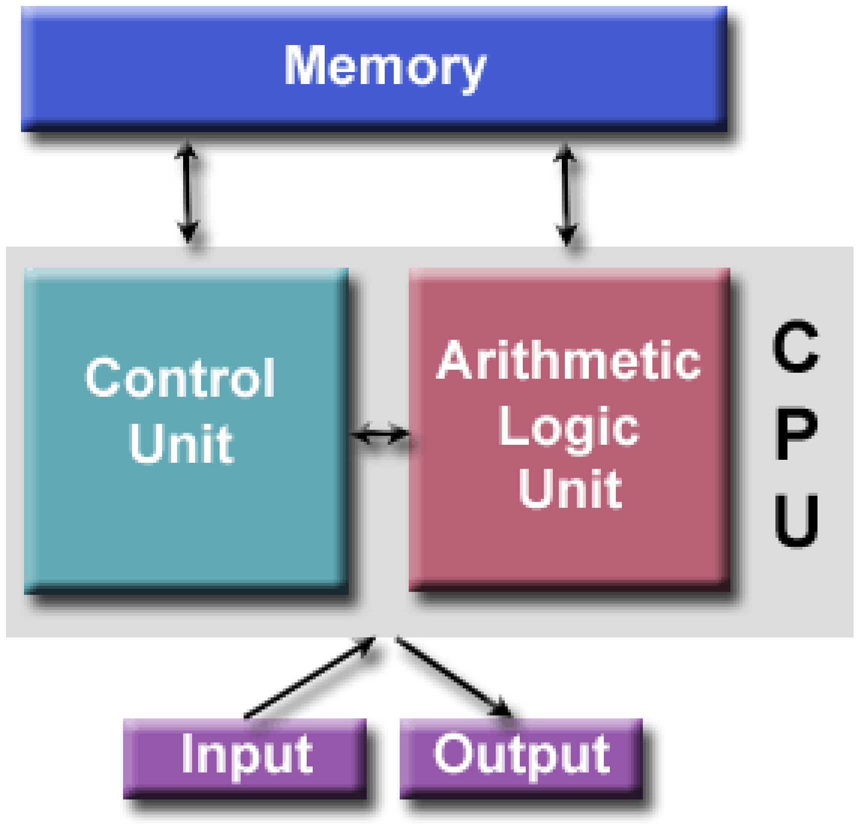
Why parallel computing?

- The marching of multi-core (2004-now)
  - Mainstream processors:
    - INTEL Quad-core Xeon (4 cores),
    - AMD Quad-core Opteron (4 cores),
    - SUN Niagara (8 cores),
    - IBM Power6 (2 cores)
  - Others
    - Intel Tflops (80 cores)
    - CISCO CSR-1 (180 cores) (network processors)
  - Increase the throughput without increasing the clock rate.



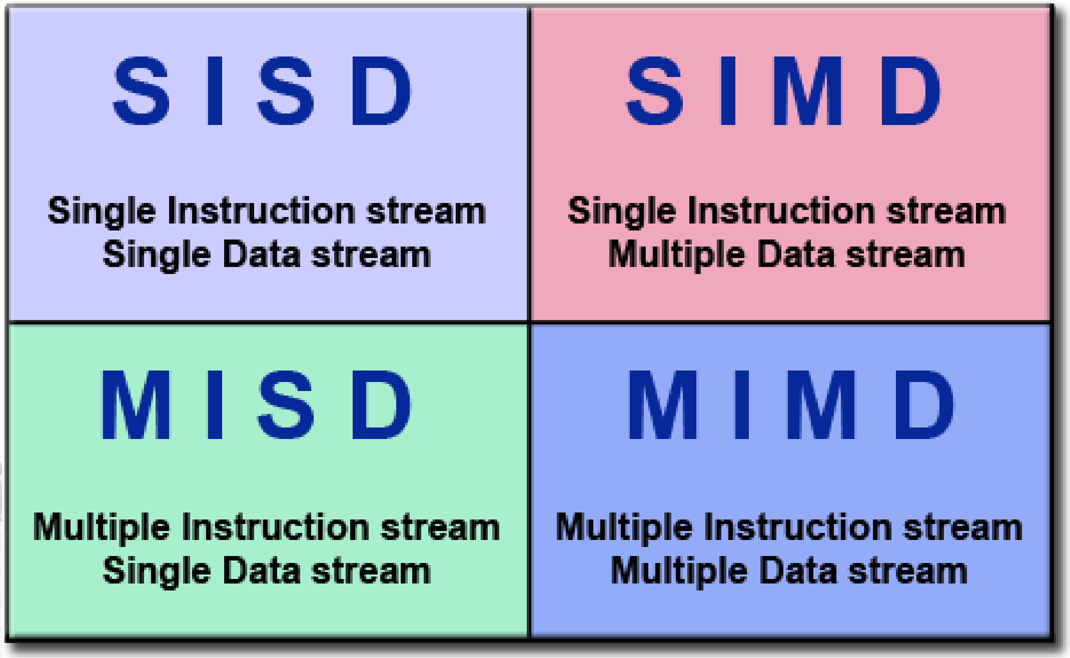
Parallel Computing Classifications

- von Neumann Architecture



Parallel Computing Classifications

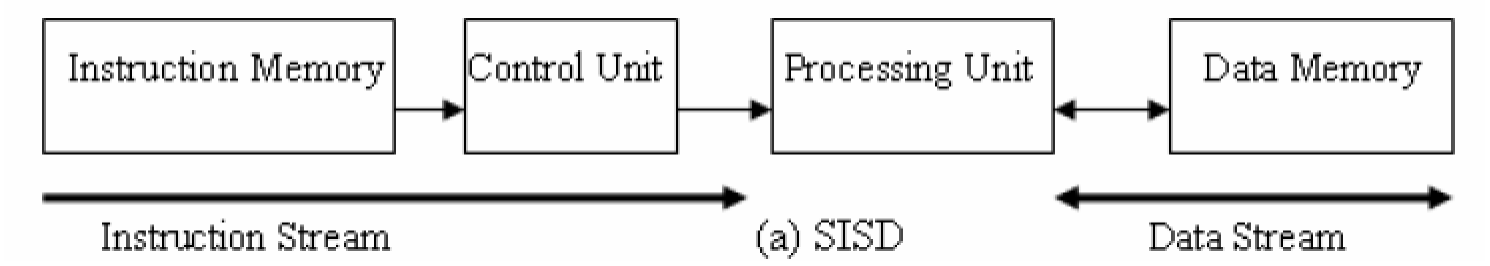
- Flynn's Classical Taxonomy
  - Flynn’s taxonomy (Michael Flynn, 1967) classifies computer architectures based on the number of instructions that can be executed and how they operate on data.



SISD Single Instruction stream Single Data stream	**SIMD** Single Instruction stream Multiple Data stream
MISD Multiple Instruction stream Single Data stream	**MIMD** Multiple Instruction stream Multiple Data stream

SISD

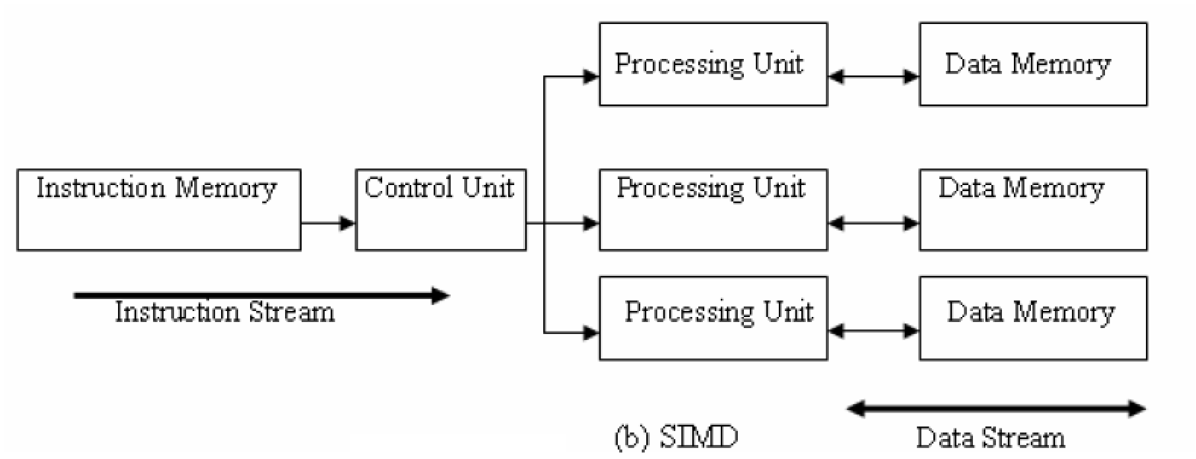
- At one time, one instruction operates on one data
- Traditional sequential architecture



(a) SISD

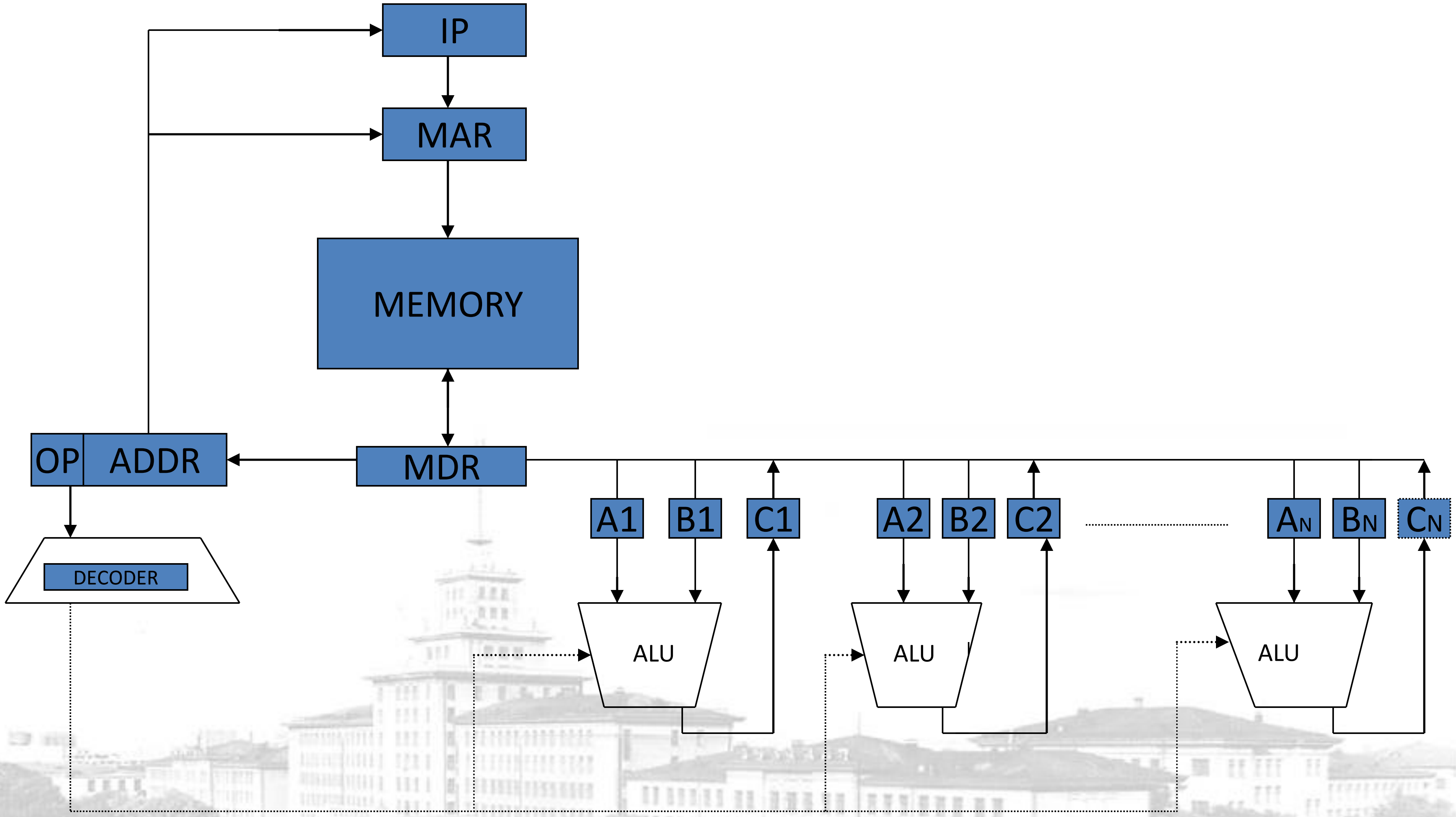
SIMD

- At one time, one instruction operates on many data
  - Data parallel architecture
  - Vector architecture has similar characteristics, but achieve the parallelism with pipelining.
- Array processors



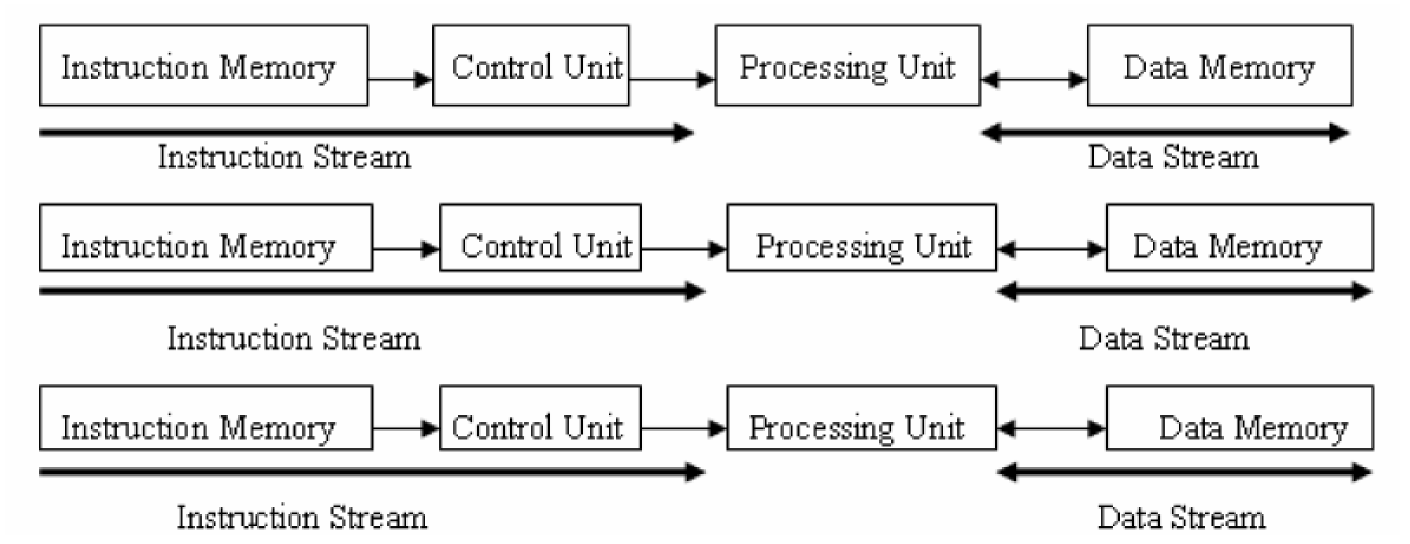
(b) SIMD

Array processor (SIMD)



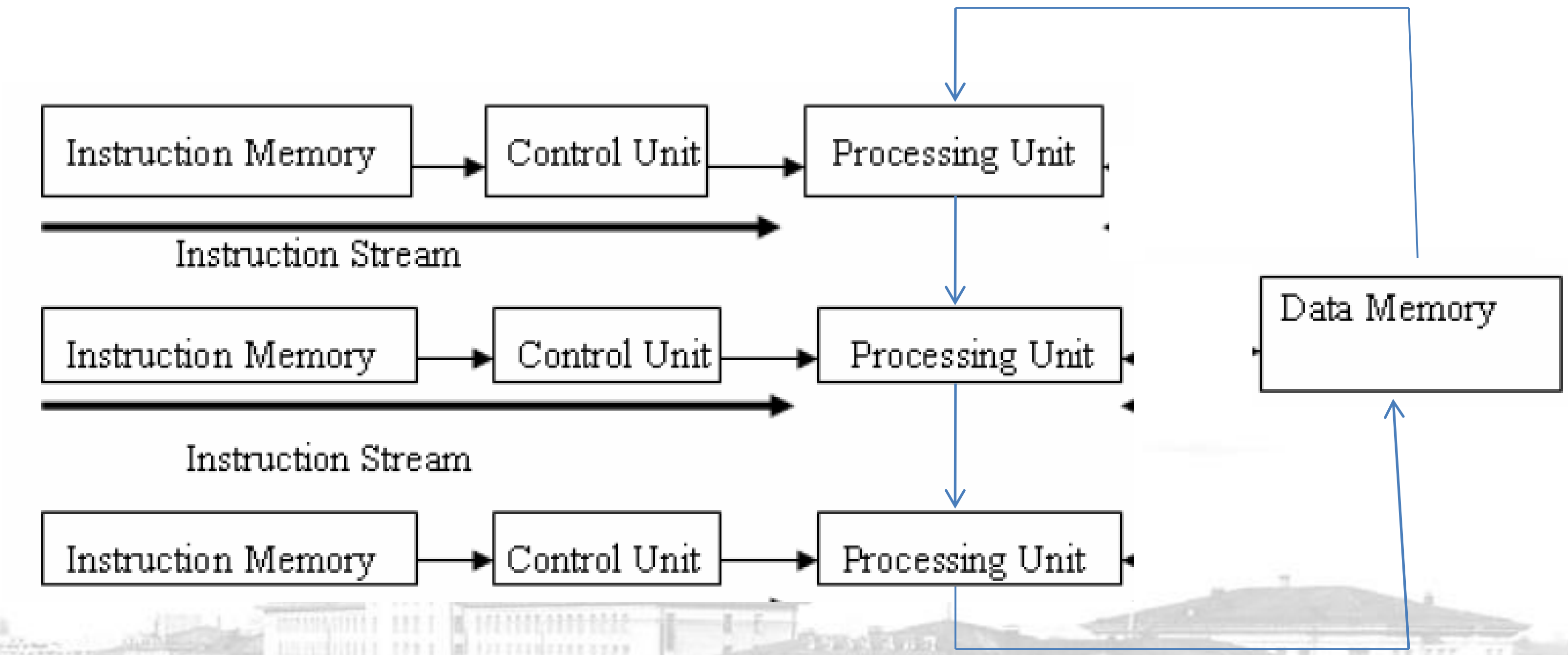
MIMD

- Multiple instruction streams operating on multiple data streams
  - Classical distributed memory or SMP architectures



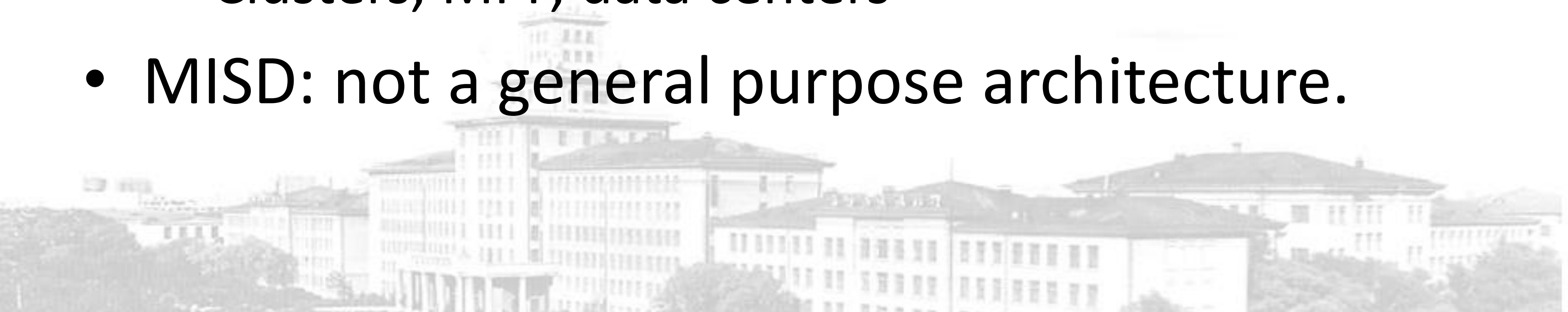
MISD machine

- Not commonly seen.
- Systolic array is one example of an MISD architecture.



Flynn’s taxonomy summary

- SISD: traditional sequential architecture
- SIMD: processor arrays, vector processor
  - Parallel computing on a budget – reduced control unit cost
  - Many early supercomputers
- MIMD: most general purpose parallel computer today
  - Clusters, MPP, data centers
- MISD: not a general purpose architecture.

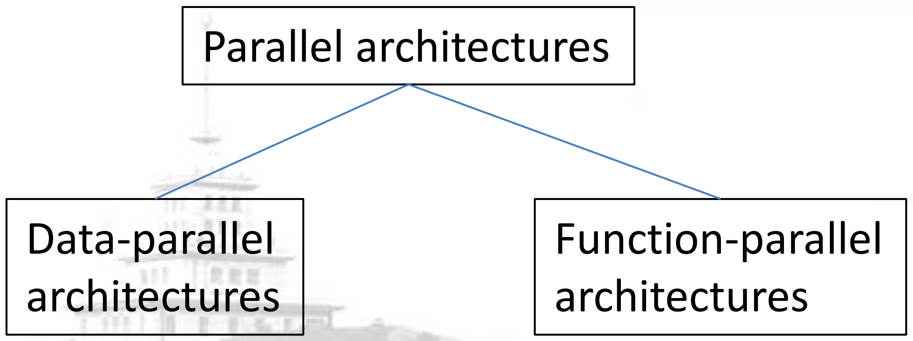


Flynn's classification on today's architectures

- Multicore processors
- Superscalar: Pipelined + multiple issues.
- SSE (Intel and AMD's support for performing operation on 2 doubles or 4 floats simultaneously).
- GPU: Cuda architecture
- IBM BlueGene

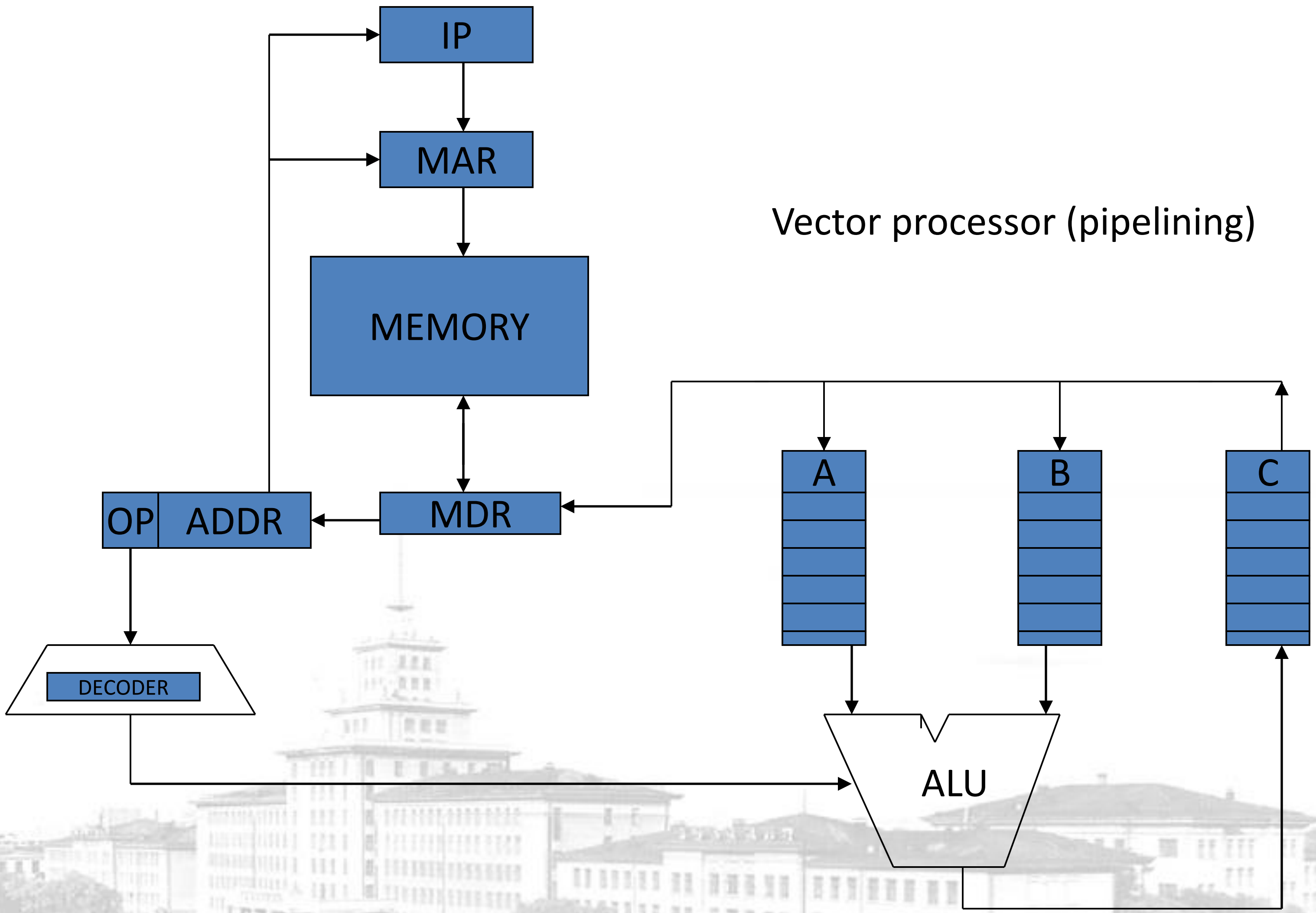
Modern classification (Sima, Fountain, Kacsuk)

- Classify based on how parallelism is achieved
  - by operating on multiple data: data parallelism
  - by performing many functions in parallel: function parallelism
    - Control parallelism, task parallelism depending on the level of the functional parallelism.



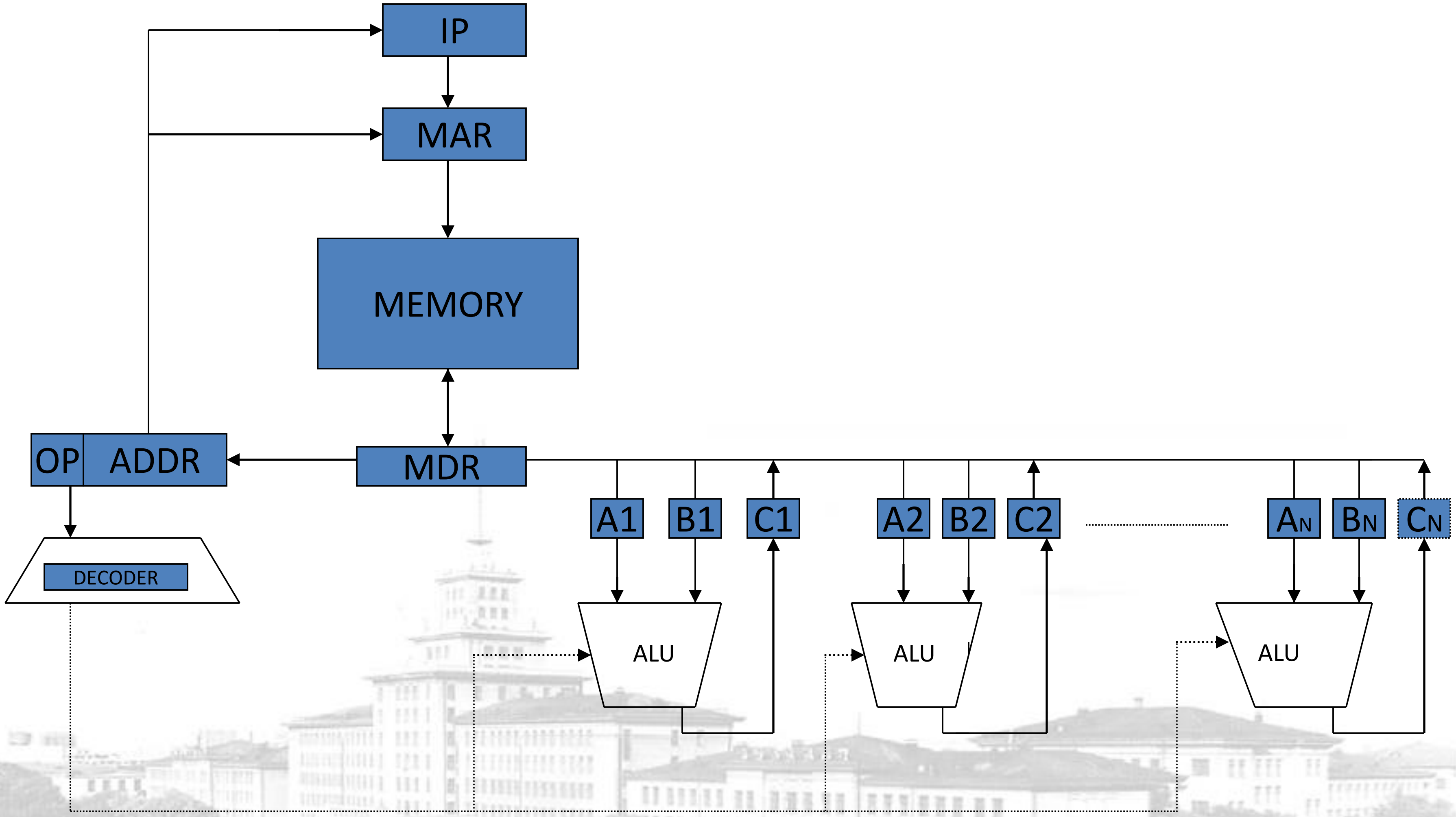
Data parallel architectures

- Vector processors, SIMD (array processors), systolic arrays.

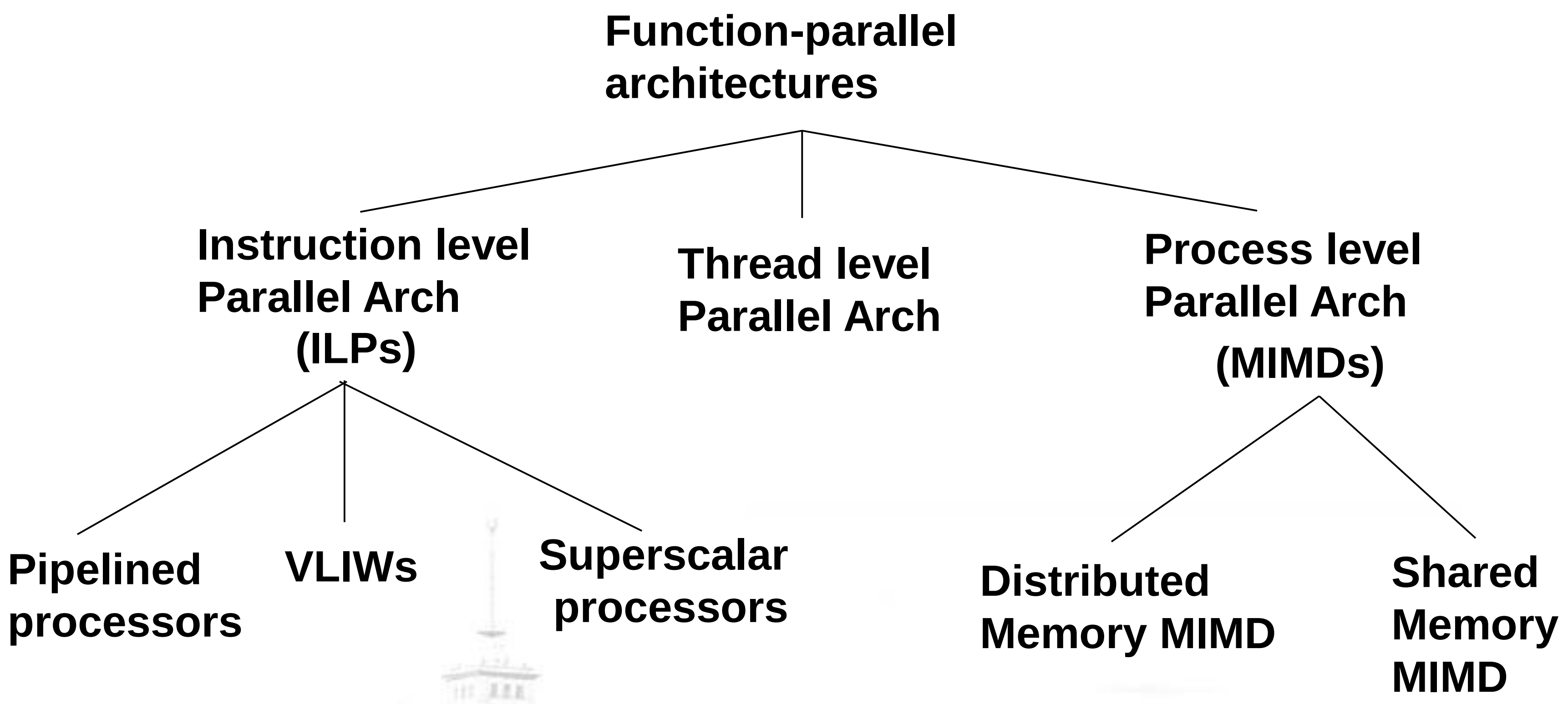


Vector processor (pipelining)

Data parallel architecture: Array processor



Control parallel architectures

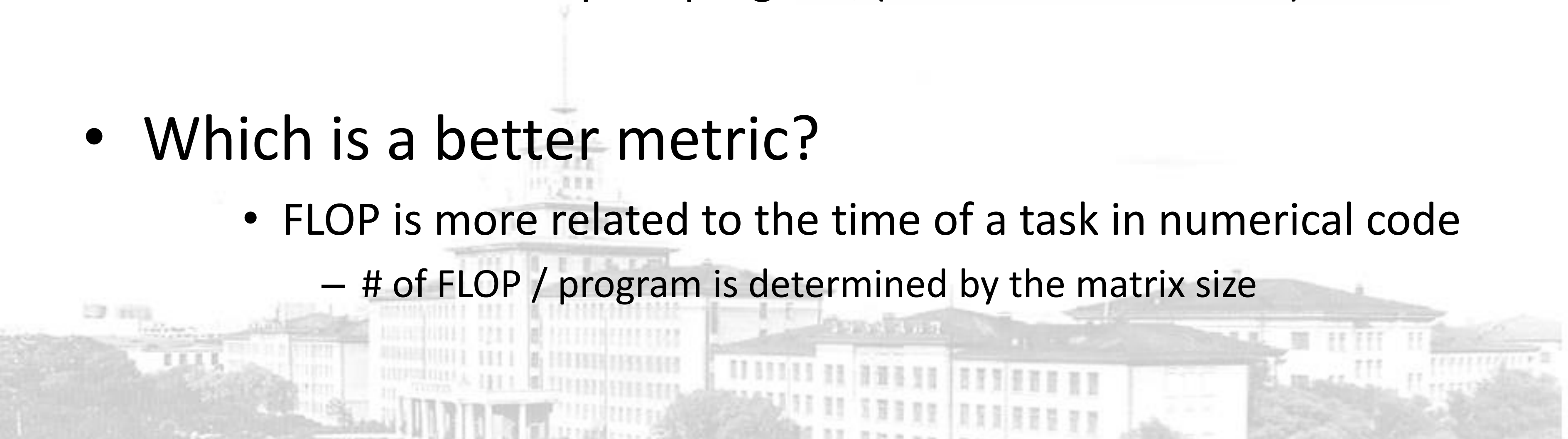

Function-parallel architectures
Instruction level Parallel Arch (ILPs)
Thread level Parallel Arch
Process level Parallel Arch (MIMDs)
Pipelined processors
VLIWs
Superscalar processors
Distributed Memory MIMD
Shared Memory MIMD

Classifying today's architectures

- Multicore processors?
- Superscalar?
- SSE?
- GPU: Cuda architecture?
- IBM BlueGene?

Performance of parallel architectures

- Common metrics
  - MIPS: million instructions per second
    - MIPS = instruction count/(execution time x 10^6)
  - MFLOPS: million floating point operations per second.
    - MFLOPS = FP ops in program/(execution time x 10^6)
- Which is a better metric?
    - FLOP is more related to the time of a task in numerical code
      - # of FLOP / program is determined by the matrix size



Performance of parallel architectures

- FlOPS units
  - kiloFLOPS (KFLOPS) 10^3
  - megaFLOPS (MFLOPS) 10^6
  - gigaFLOPS (GFLOPS) 10^9 ← single CPU performance
  - teraFLOPS (TFLOPS) 10^12

  - petaFLOPS (PFLOPS) 10^15 ← we are here right now
    - »10 petaFLOPS supercomputers

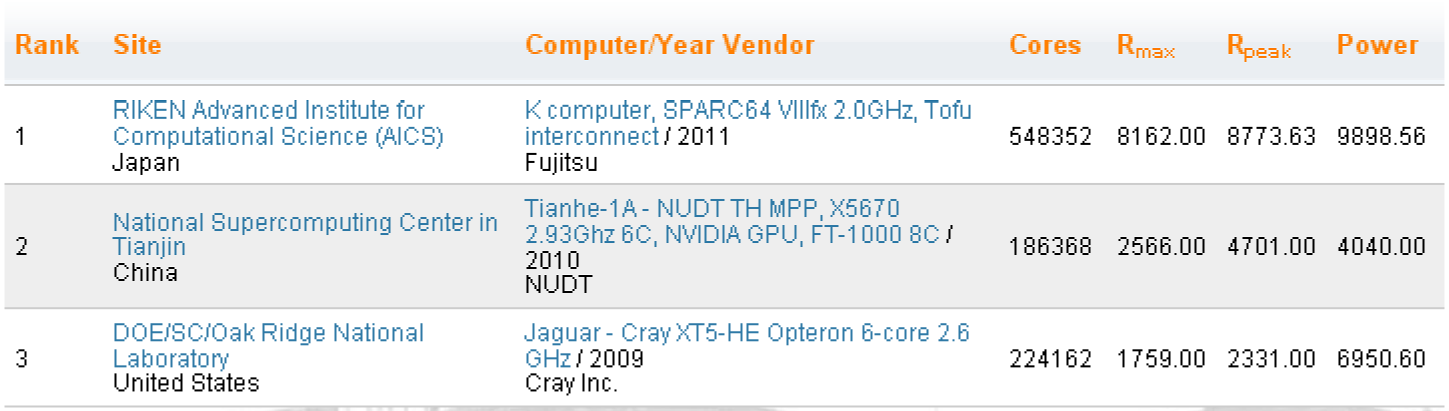
  - exaFLOPS (EFLOPS) 10^18 ← the next milestone

Peak and sustained performance

- Peak performance
  - Measured in MFLOPS
  - Highest possible MFLOPS when the system does nothing but numerical computation
  - Rough hardware measure
  - Little indication on how the system will perform in practice.

Peak and sustained performance

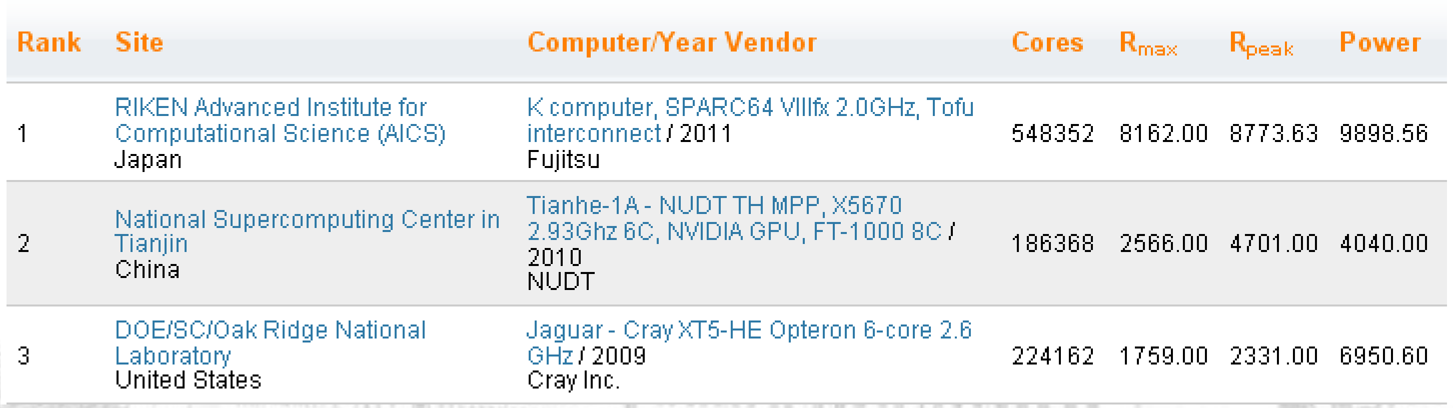
- Sustained performance
  - The MFLOPS rate that a program achieves over the entire run.
- Measuring sustained performance
  - Using benchmarks
- Peak MFLOPS is usually much larger than sustained MFLOPS
  - Efficiency rate = sustained MFLOPS / peak MFLOPS



Rank	Site	Computer/Year Vendor	Cores	R_{max}	R_{peak}	Power
1	RIKEN Advanced Institute for Computational Science (AICS) Japan	K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect / 2011 Fujitsu	548352	8162.00	8773.63	9898.56
2	National Supercomputing Center in Tianjin China	Tianhe-1A - NUDT TH MPP, X5670 2.93Ghz 6C, NVIDIA GPU, FT-1000 8C / 2010 NUDT	186368	2566.00	4701.00	4040.00
3	DOE/SC/Oak Ridge National Laboratory United States	Jaguar - Cray XT5-HE Opteron 6-core 2.6 GHz / 2009 Cray Inc.	224162	1759.00	2331.00	6950.60

Measuring the performance of parallel computers

- Benchmarks: programs that are used to measure the performance.
  - LINPACK benchmark: a measure of a system's floating point computing power
    - Solving a dense N by N system of linear equations Ax=b
    - Use to rank supercomputers in the top500 list.

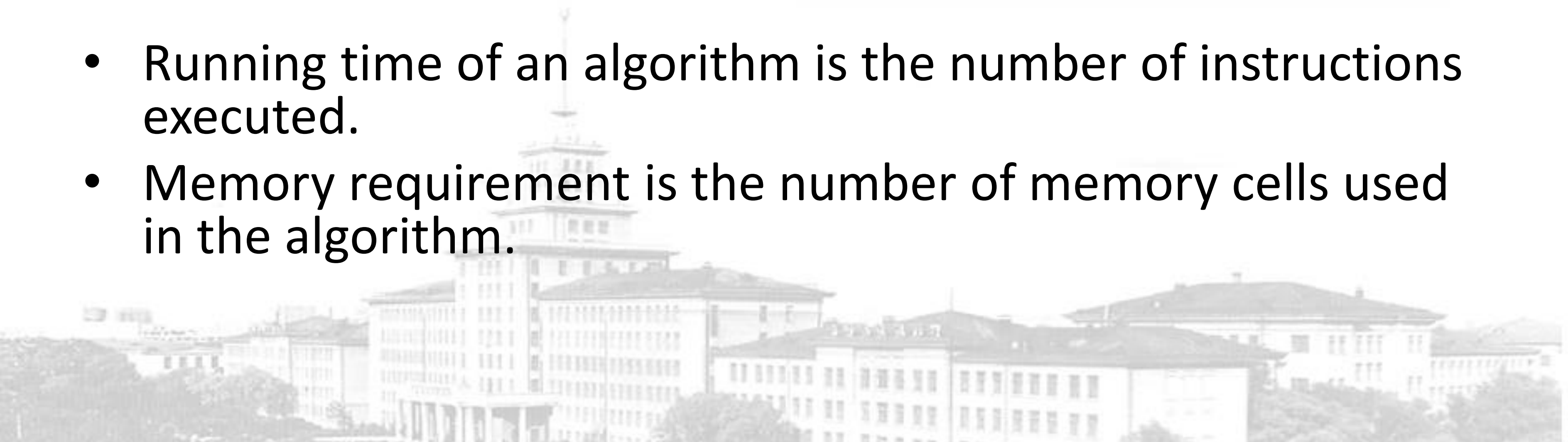


Rank	Site	Computer/Year Vendor	Cores	R_{max}	R_{peak}	Power
1	RIKEN Advanced Institute for Computational Science (AICS) Japan	K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect / 2011 Fujitsu	548352	8162.00	8773.63	9898.56
2	National Supercomputing Center in Tianjin China	Tianhe-1A - NUDT TH MPP, X5670 2.93Ghz 6C, NVIDIA GPU, FT-1000 8C / 2010 NUDT	186368	2566.00	4701.00	4040.00
3	DOE/SC/Oak Ridge National Laboratory United States	Jaguar - Cray XT5-HE Opteron 6-core 2.6 GHz / 2009 Cray Inc.	224162	1759.00	2331.00	6950.60

RAM (random access machine) model

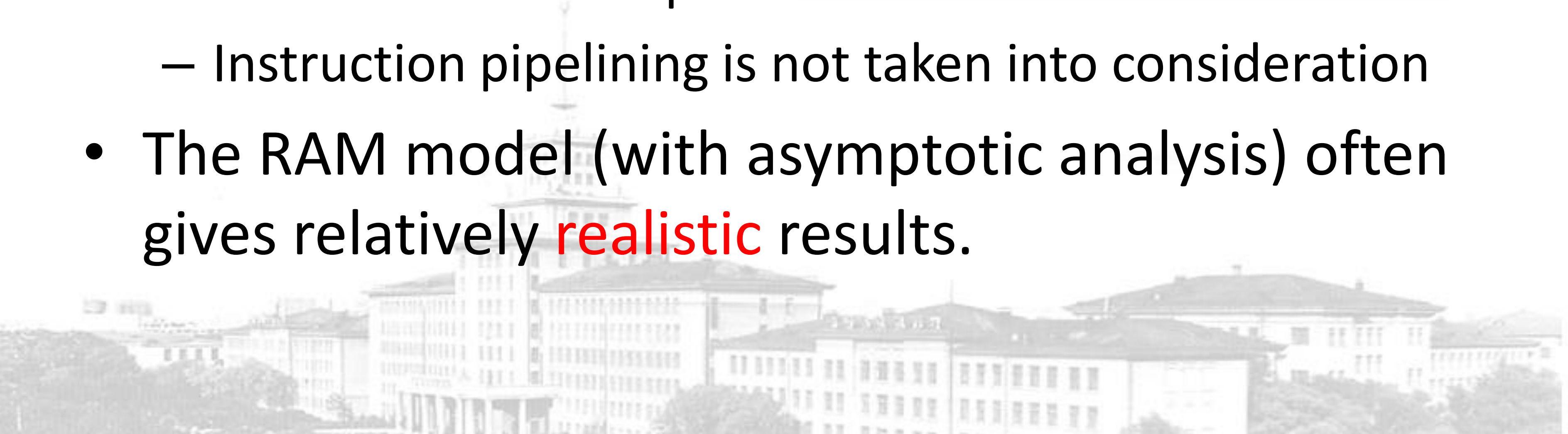
- Memory consists of infinite array (memory cells).
- Each memory cell holds an infinitely large number.
- Instructions execute sequentially one at a time.
- All instructions take unit time
  - Load/store
  - Arithmetic
  - Logic

- Running time of an algorithm is the number of instructions executed.
- Memory requirement is the number of memory cells used in the algorithm.



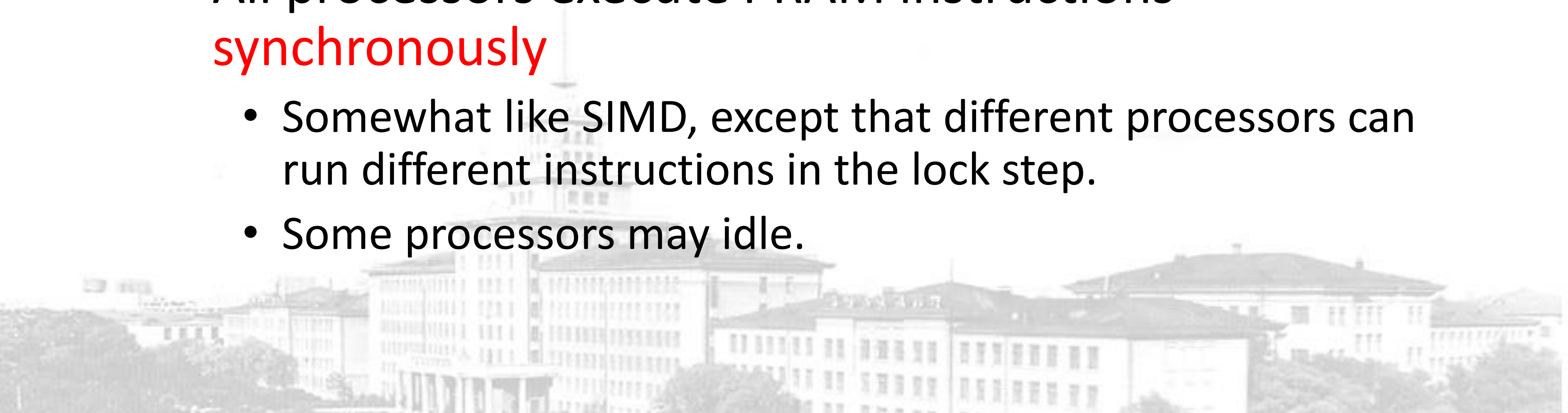
RAM (random access machine) model

- The RAM model is the base of algorithm analysis for sequential algorithms although it is not perfect.
  - Memory not infinite
  - Not all memory access take the same time
  - Not all arithmetic operations take the same time
  - Instruction pipelining is not taken into consideration
- The RAM model (with asymptotic analysis) often gives relatively realistic results.



PRAM (Parallel RAM)

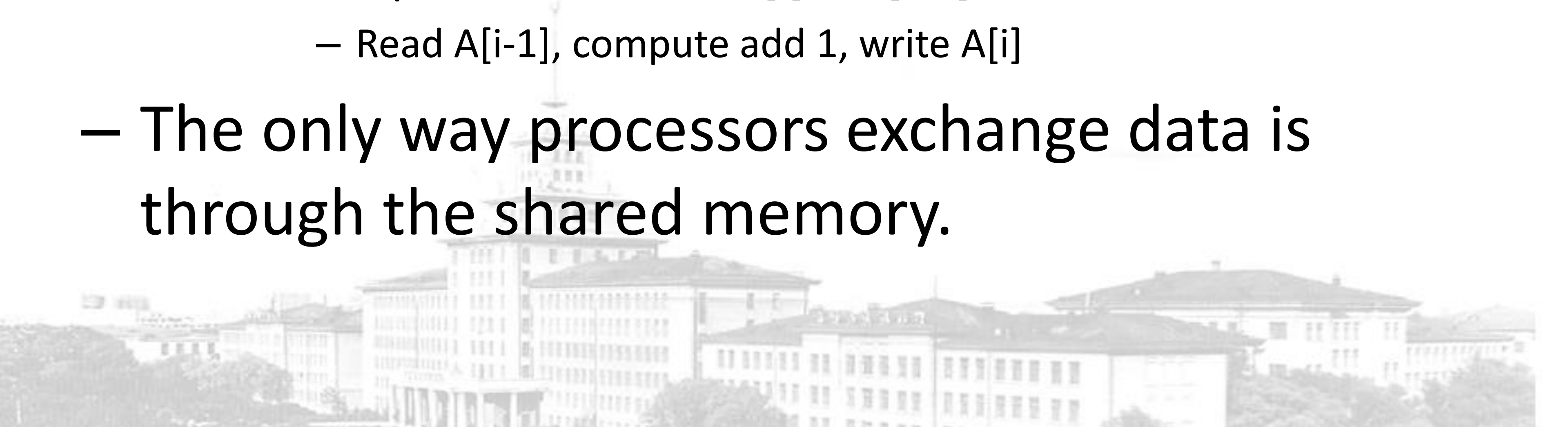
- A model developed for parallel machines
  - An unbounded collection of processors
  - Each processor has an infinite number of registers
  - An unbounded collection of shared memory cells.
  - All processors can access all memory cells in unit time (when there is no memory conflict).
  - All processors execute PRAM instructions synchronously
    - Somewhat like SIMD, except that different processors can run different instructions in the lock step.
    - Some processors may idle.



PRAM (Parallel RAM)

- A model developed for parallel machines
  - Each PRAM instruction executes in 3-phase cycles
    - – Read from a share memory cell (if needed)
    - – Computation
    - – Write to a share memory cell (if needed)
    - – Example: for all I, do A[i] = A[i-1]+1;
      - – Read A[i-1], compute add 1, write A[i]

– The only way processors exchange data is through the shared memory.



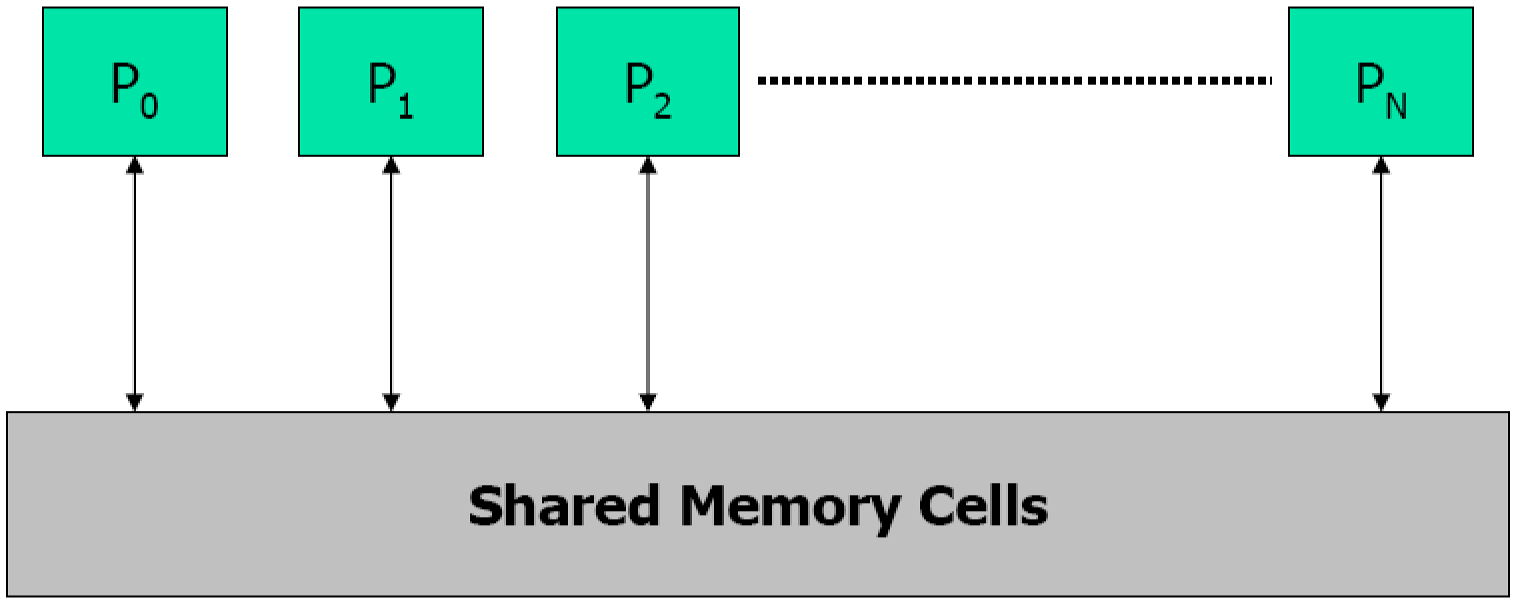
PRAM (Parallel RAM)

Parallel time complexity: the number of synchronous steps in the algorithm

Space complexity: the number of share memory

Parallelism: the number of processors used

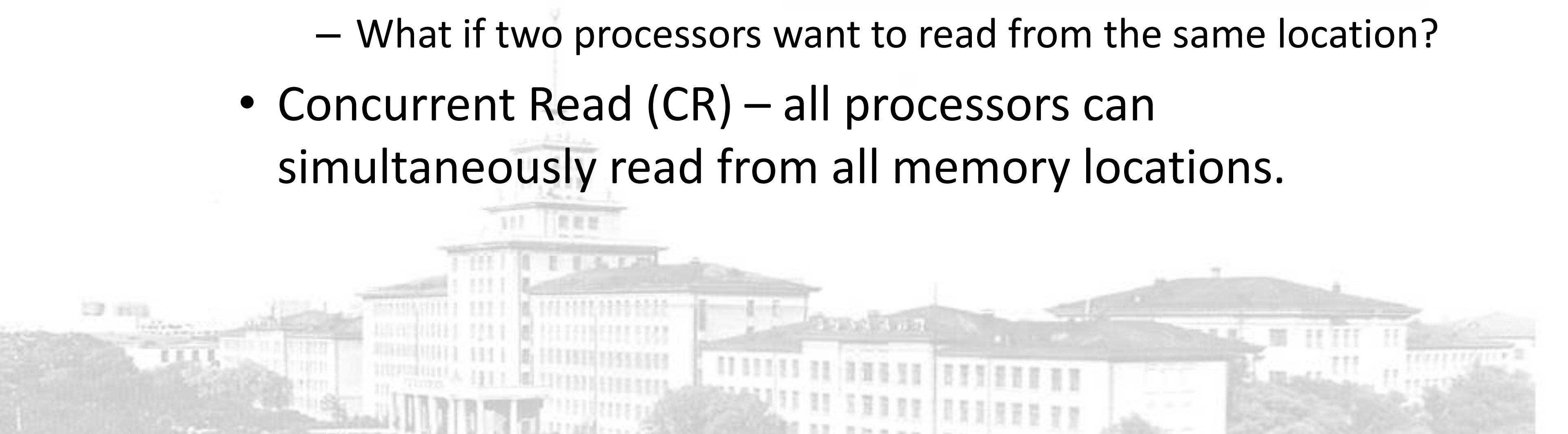
PRAM



All processors can do things in a synchronous manner (with infinite shared Memory and infinite local memory), how many steps do it take to complete the task?

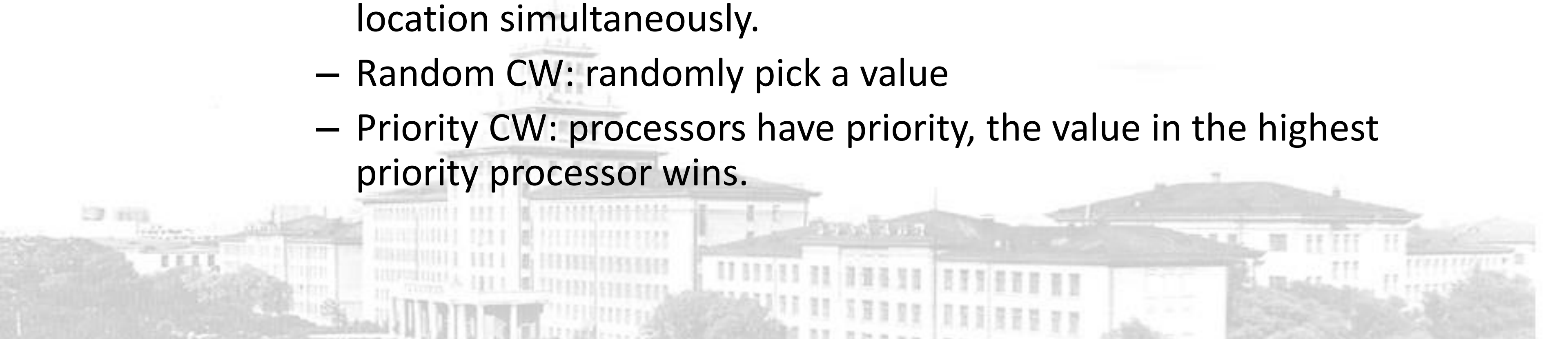
PRAM – further refinement

- PRAMs are further classifed based on how the memory conflicts are resolved.
  - Read
    - Exclusive Read (ER) – all processors can only simultaneously read from distinct memory location (but not the same location).
      - What if two processors want to read from the same location?
    - Concurrent Read (CR) – all processors can simultaneously read from all memory locations.



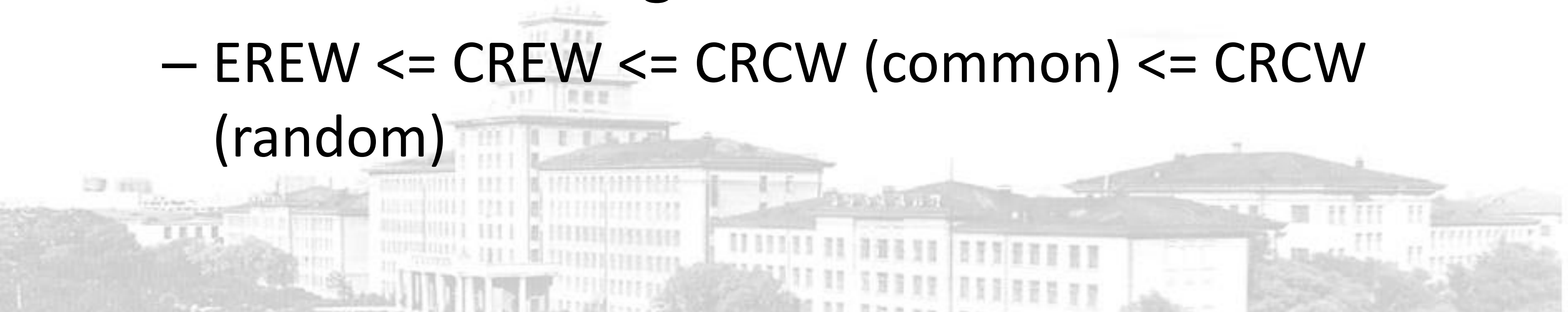
PRAM – further refinement

- PRAMs are further classified based on how the memory conflicts are resolved.
  - Write
    - Exclusive Write (EW) – all processors can only simultaneously write to distinct memory location (but not the same location).
    - Concurrent Write (CW) – all processors can simultaneously write to all memory locations.
      - Common CW: only allow same value to be written to the same location simultaneously.
      - Random CW: randomly pick a value
      - Priority CW: processors have priority, the value in the highest priority processor wins.



PRAM model variations

- EREW, CREW, CRCW (common), CRCW (random), CRCW (Priority)
  - Which model is closer to the practical SMP or multicore machines?
- Model A is computationally stronger than model B if and only if any algorithm written in B will run unchange in A.
  - EREW <= CREW <= CRCW (common) <= CRCW (random)



PRAM Algorithm 1—求和

- SUM: Add N numbers in memory M[0, 1, …, N-1]
- Sequential SUM algorithm (O(N) complexity)

```
for (i=0; i<N; i++) sum = sum + M[i];
```

- PRAM SUM algorithm?

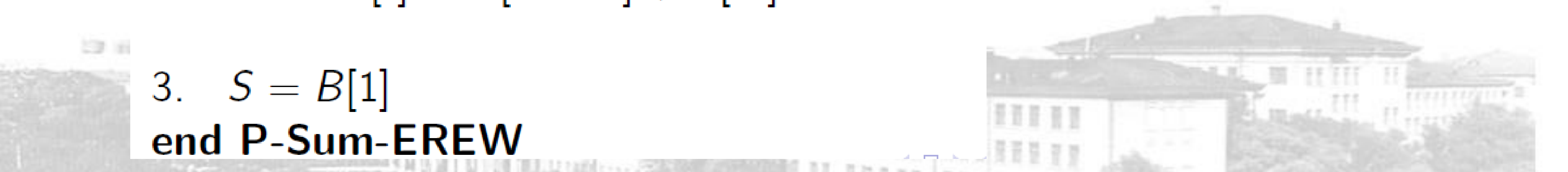
PRAM Algorithm 1—求和

- PRAM SUM algorithm?

```
begin P-Sum-EREW
Input: $n = 2^k$ numbers stored in Array $A[1..n]$
Output: $S = \sum_{i=1}^{n} A[i]$
Note $k = \log_2 n$

begin
1.  for $i = 1...n$ in parallel do
        B[i]=A[i]

2.  for $h = 1..k$ do
        for $1 \le i \le n/2^h$ in parallel do
            $B[i] = B[2i-1] + B[2i]$

3.  $S = B[1]$
end P-Sum-EREW
```

a	b	c	d	e	f	g	h
a+b	c+d	e+f	g+h				
a+b+c+d	e+f+g+h						
a+b+c+d+e+f+g+h							

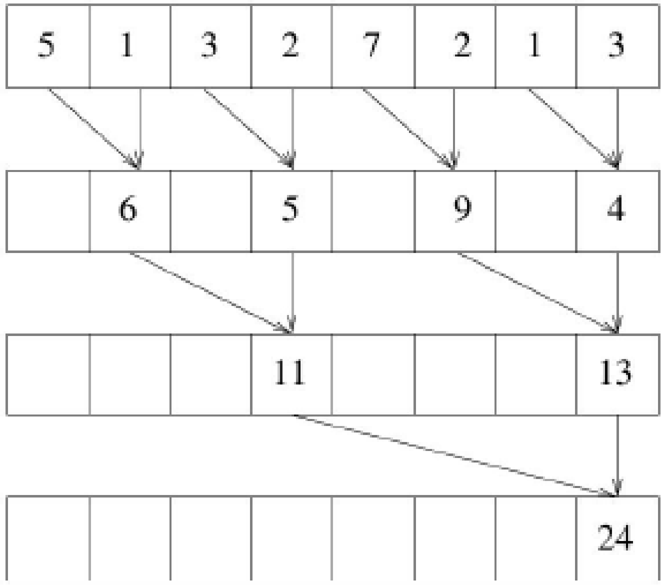
1	2	3	4	5	6	7	8
3	7	11	15				
10	26						
36							



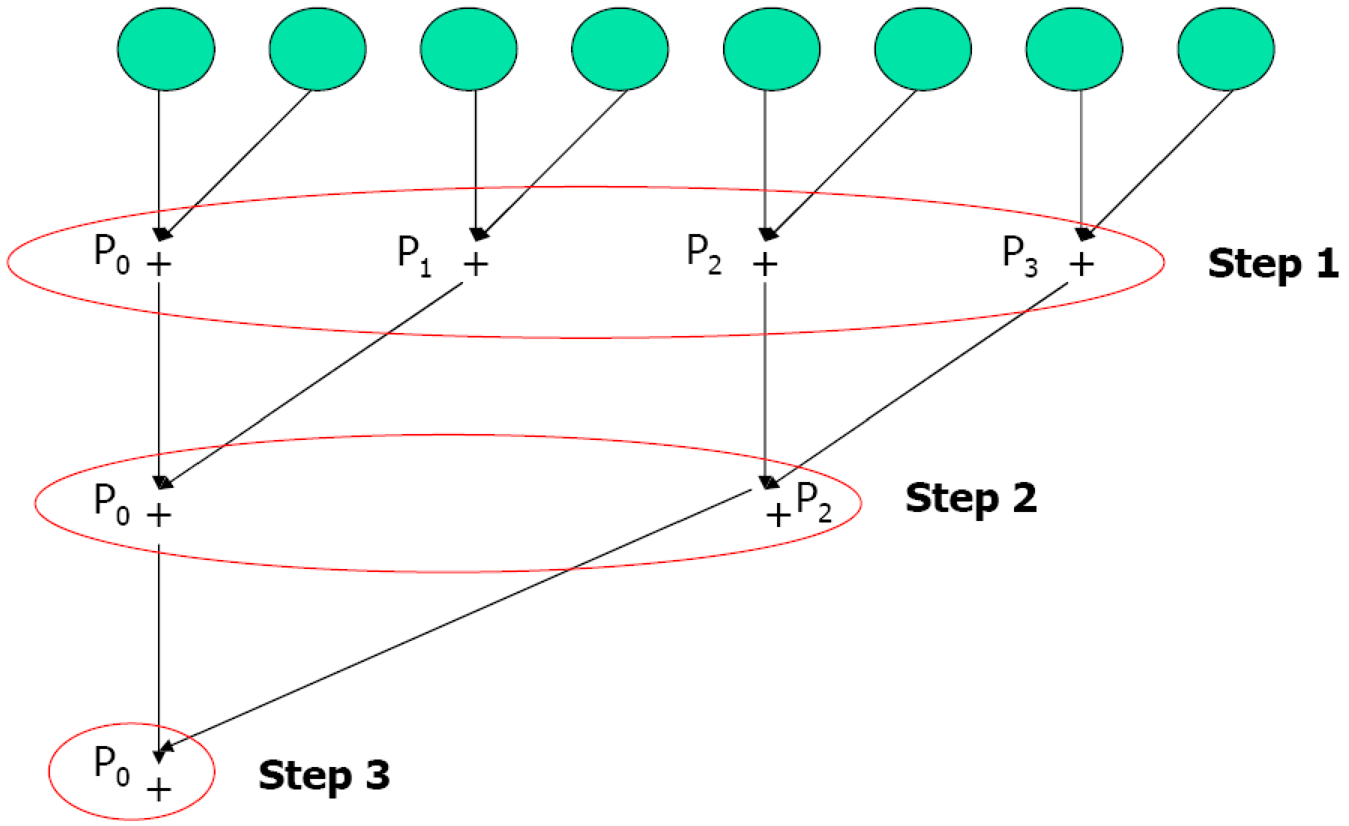
PRAM Algorithm 1—求和

- PRAM SUM algorithm? Indexing Methods

```
for $1 \le i \le n$ in parallel do
    for all $j$, $1 \le j \le \log_2 n$ do
        if $i \mod 2^j = 0$ then
            $B[i] = B[i] + B[i - 2^{j-1}]$
```



PRAM Algorithm 1—求和



PRAM Algorithm 1—求和

- Time complexity: log(n) steps
- Parallelism: n/2 processors
- Speed-up (vs sequential algorithm): n/log(n)

PRAM Algorithm 2—广播数据

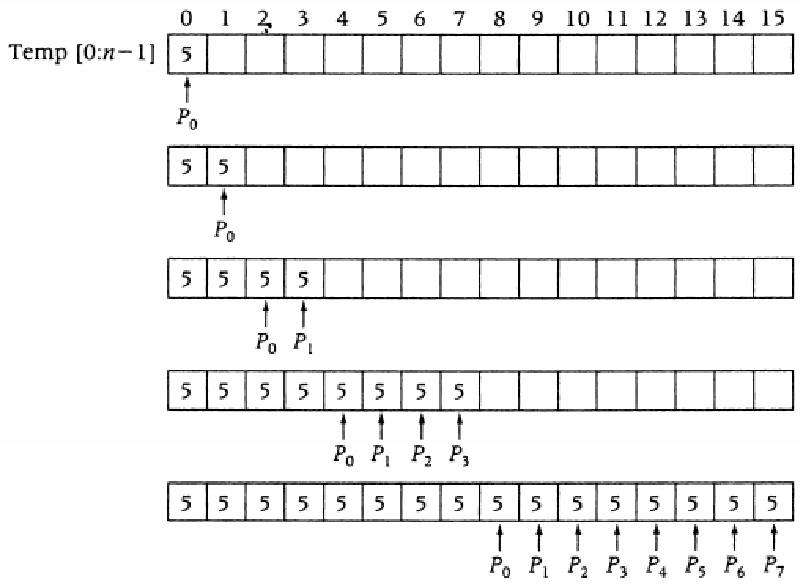
- 一个数据，传给所有处理器
- EREW
  - double the number of processors that have the value in each steps
  - Log(P) steps
- CREW
  - Broadcaster sends value to shared memory
  - All processors read from shared memory
  - O(1) steps

PRAM Algorithm 2—广播数据

EREW Broadcast

Array $A[1..n]$ of size $n = 2^k$
Input X value to be broadcast to array A
(fill array A with entries value X)
Output $A[i] = X, i = 1, \ldots, n$

```
$A[1] = X$
For $i = 0 \ldots k - 1$
    begin
    For all $j$, $2^i + 1 \le j \le 2^{i+1}$ in parallel do
        $A[j] = A[j - 2^i]$
    end-for-loop
end EREW-broadcast
```

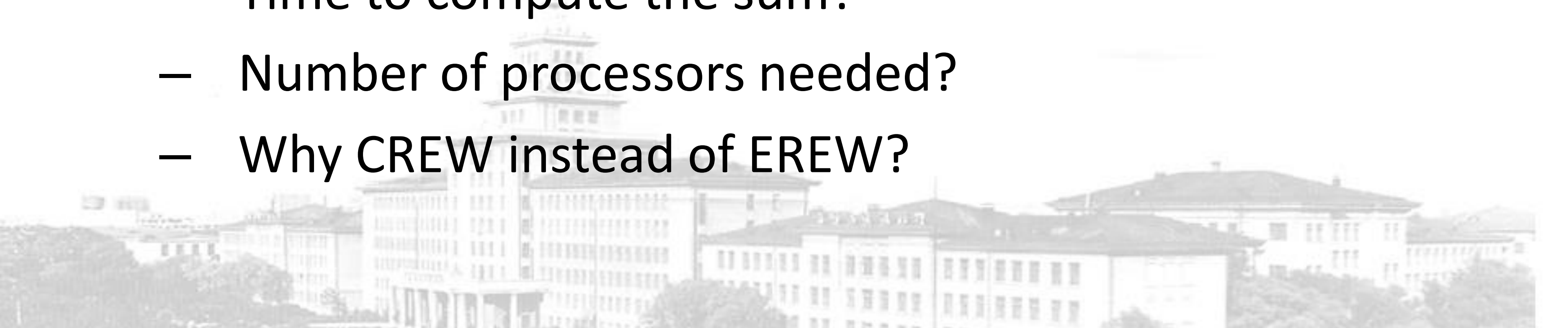


PRAM Algorithm 3—向量计算

- Given an n x n matrix A and a column vector X = (x[0], x[1], …, x[n-1]), B = A X
- Sequential code:

```
For(i=0; i<n; i++) for (j=0; j<n; j++) B[i] += A[i][j] * X[j];
```

- CREW PRAM algorithm
  - Time to compute the product?
  - Time to compute the sum?
  - Number of processors needed?
  - Why CREW instead of EREW?



PRAM Algorithm 3—向量计算

- Given an n x n matrix A and a column vector X = (x[0], x[1], …, x[n-1]), B = A X

begin P-Matrix-Vector-Mult

Input: Matrix A, $n \times n$, Vector B, $n \times 1$

Output: Vector Y, $n \times 1$, $Y = A * B$ Note: $n = 2^k$

```
1.  This step is concurrent read on $B[j]$ for all $i$
    for $i, j = 1..n$ in parallel do
        $C[i,j] = A[i,j] * B[j]$

2.  for $i = 1..n$ in parallel do
        for $h = 1..k$ do
            for $1 \le j \le n/2^h$ in parallel do
                $C[i,j] = C[i,2j-1] + C[i,2j]$

3.  for $i = 1..n$ in parallel do
        $Y[i] = C[i,1]$
end P-M-V-Mult-alg
```

Sequential

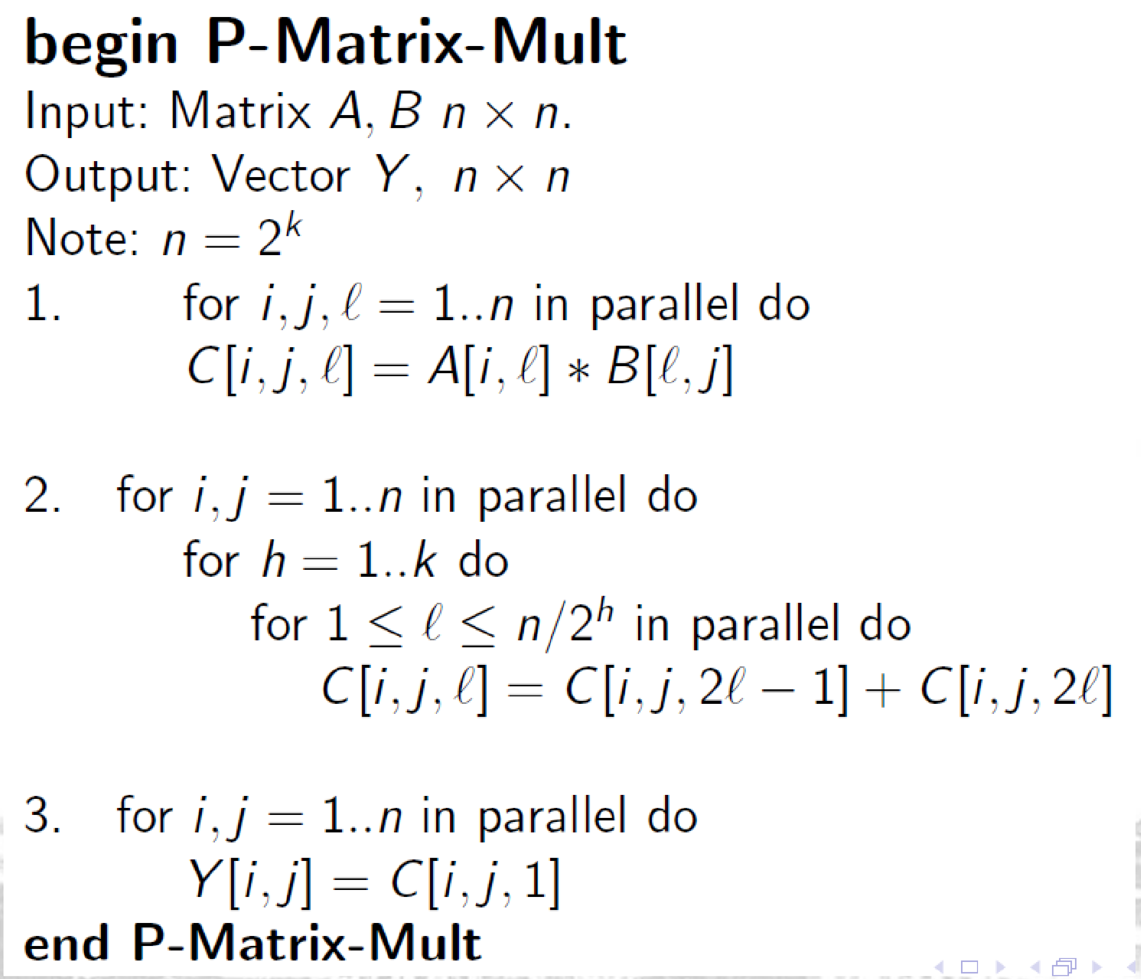
$$\begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} \times \begin{vmatrix} b_1 \\ b_2 \end{vmatrix} = \begin{vmatrix} a_{11}b_1 + a_{12}b_2 \\ a_{21}b_1 + a_{22}b_2 \end{vmatrix}$$

Parallel

$$\begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} \times \begin{vmatrix} b_1 \\ b_2 \end{vmatrix} \longrightarrow \begin{vmatrix} a_{11}b_1 & a_{12}b_2 \\ a_{21}b_1 & a_{22}b_2 \end{vmatrix} \longrightarrow$$

PRAM Algorithm 4—矩阵乘法

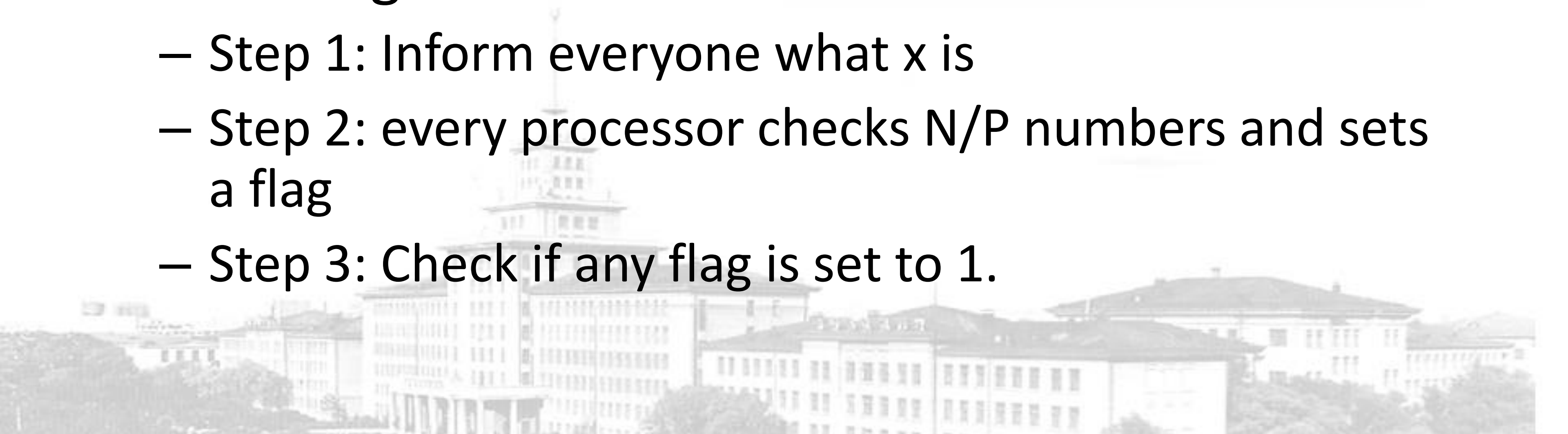
- CREW PRAM algorithm?



```
begin P-Matrix-Mult
Input: Matrix $A, B$ $n \times n$.
Output: Vector $Y$, $n \times n$
Note: $n = 2^k$
1.      for $i, j, \ell = 1..n$ in parallel do
        $C[i, j, \ell] = A[i, \ell] * B[\ell, j]$

2.   for $i, j = 1..n$ in parallel do
        for $h = 1..k$ do
            for $1 \leq \ell \leq n/2^h$ in parallel do
                $C[i, j, \ell] = C[i, j, 2\ell - 1] + C[i, j, 2\ell]$

3.   for $i, j = 1..n$ in parallel do
        $Y[i, j] = C[i, j, 1]$
end P-Matrix-Mult
```

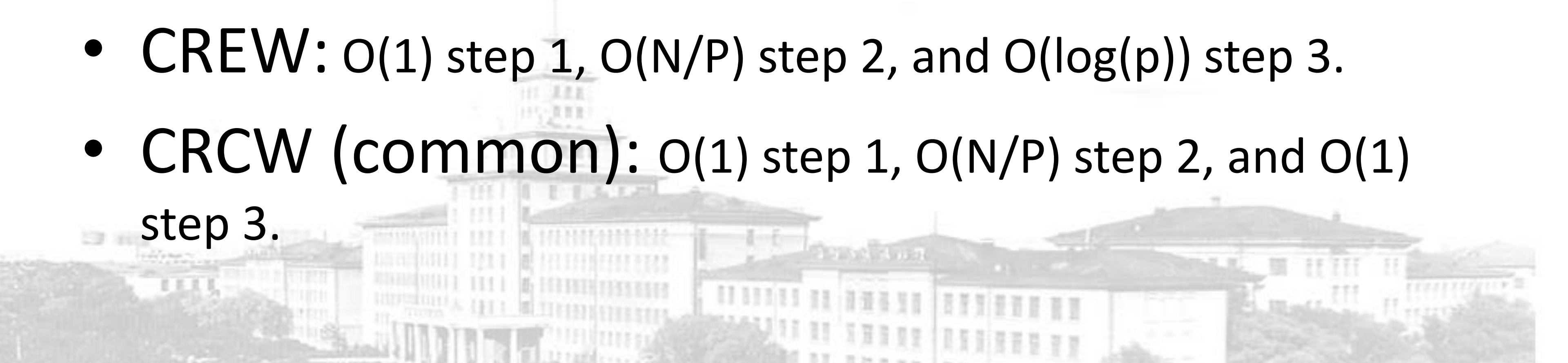
PRAM Algorithm 5—搜索

- P processors PRAM with unsorted N numbers (P<=N)
- Does x exist in the N numbers?
- p_0 has x initially, p_0 must know the answer at the end.
- PRAM Algorithm:
  - Step 1: Inform everyone what x is
  - Step 2: every processor checks N/P numbers and sets a flag
  - Step 3: Check if any flag is set to 1.



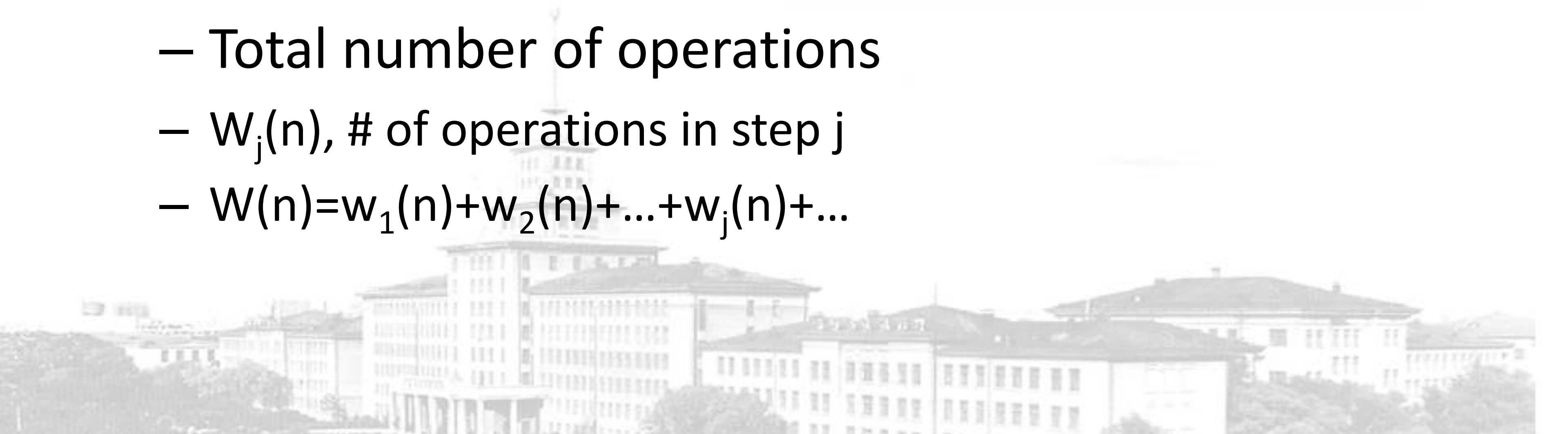
PRAM Algorithm 5—搜索

- PRAM Algorithm:
  - Step 1: Inform everyone what x is
  - Step 2: every processor checks N/P numbers and sets a flag
  - Step 3: Check if any flag is set to 1.
- EREW: O(log(p)) step 1, O(N/P) step 2, and O(log(p)) step 3.
- CREW: O(1) step 1, O(N/P) step 2, and O(log(p)) step 3.
- CRCW (common): O(1) step 1, O(N/P) step 2, and O(1) step 3.



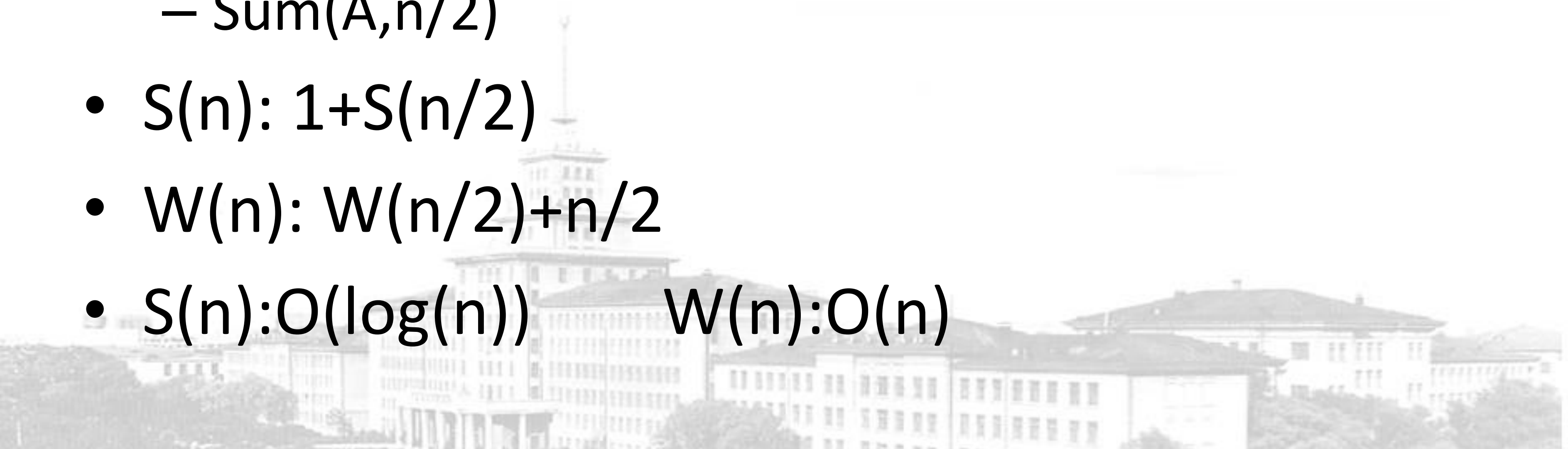
PRAM —Work-Time Paradigm

- Associate two complexity measures with a PRAM Algorithm
- S(n): time complexity
  - Total number of steps
- W(n): work complexity
  - Total number of operations
  - $W_j(n)$, # of operations in step j
  - $W(n)=w_1(n)+w_2(n)+\ldots+w_j(n)+\ldots$



PRAM —Work-Time Paradigm

- Sum(A,n)
  - If(n==1)
    - Return A[1]
  - For 1<=i<=n/2 in parallel do
    - A[i]=A[2i-1]+A[2i]
  - Sum(A,n/2)
- S(n): 1+S(n/2)
- W(n): W(n/2)+n/2
- S(n):O(log(n))   W(n):O(n)

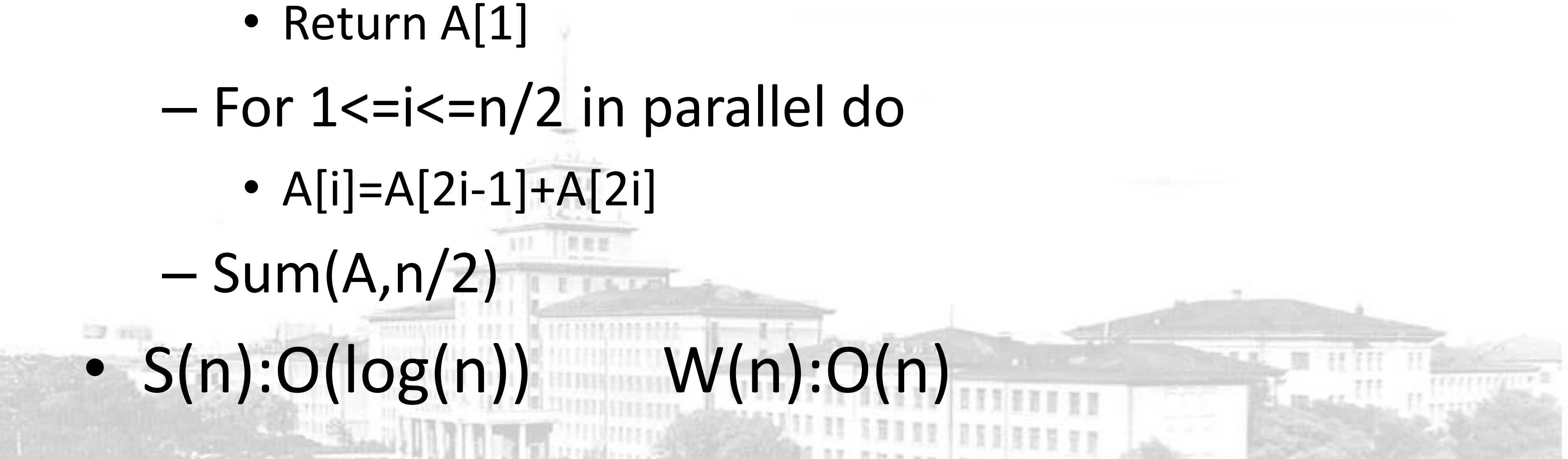


PRAM —Brent's Scheduling Principle

A parallel algorithm with step complexity S(n) and work complexity W(n) can be simulated on a p-processor PRAM in no more than $T_C(n,p) = W(n)/p + S(n)$ parallel steps

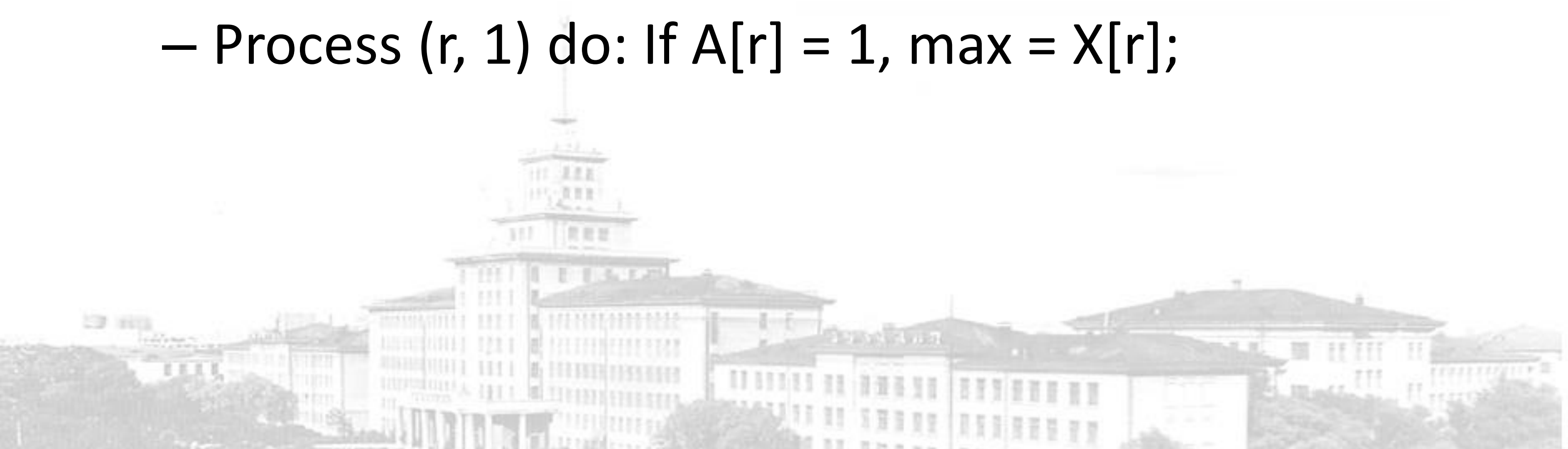
- S(n) could be thought of as the length of the "critical path"

- Sum(A,n)
  - If(n==1)
    - Return A[1]
  - For 1<=i<=n/2 in parallel do
    - A[i]=A[2i-1]+A[2i]
  - Sum(A,n/2)
- S(n):O(log(n))   W(n):O(n)



PRAM Algorithm 6—最大值

- N个元素，求最大值
- CRCW algorithm with O(1) time using N^2 processors
  - – Processor (r, 1) do A[s] = 1
  - – Process (r,s) do if (X[r] < X[s]) A[r] = 0;
  - – Process (r, 1) do: If A[r] = 1, max = X[r];



PRAM Algorithm 6—最大值

- O(loglog*n*) time and O(*n*loglog*n*) processors
  - Partition data into $n^{1/2}$ pieces
  - 递归计算，每一份的最大值
  - 进一步，利用上一个算法求解最大值

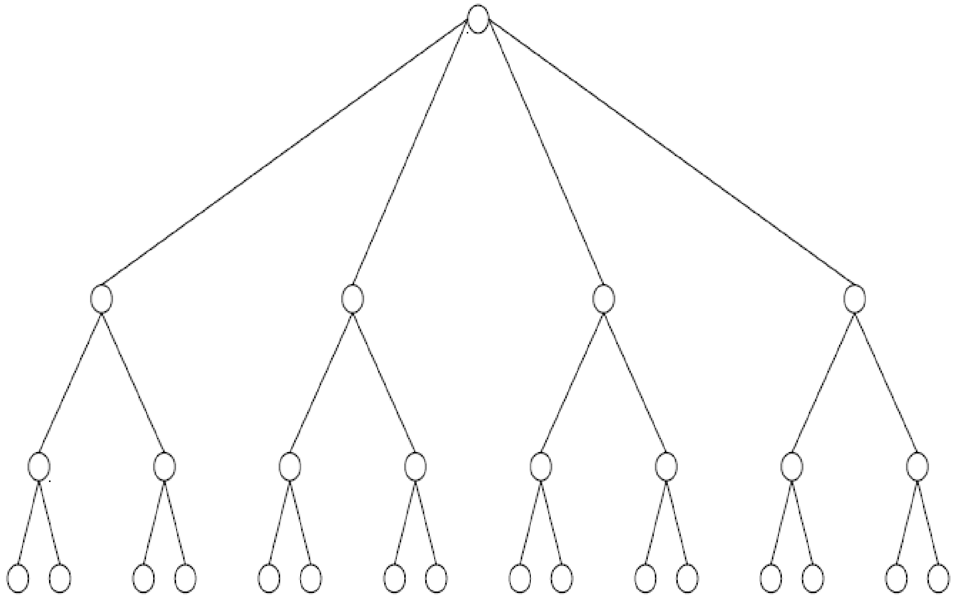
$$T(n) \le T(\sqrt{n}) \ + \ c_1; \quad W(n) \le \sqrt{n}W(\sqrt{n}) \ + \ c_2 n$$

$$T(n) \ = \ O(\log\log n)$$

$$W(n) \ = \ O(n \log\log n)$$

PRAM Algorithm 6—最大值

- O(loglog*n*) time and O(*n*loglog*n*) processors
  - Partition data into $n^{1/2}$ pieces



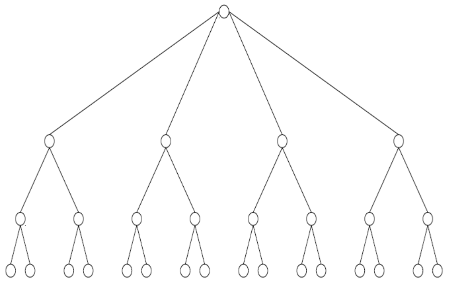
Doubly-logarithmic trees (n=16)

Balanced trees Techniques

- $n = 2^{2^h}$ leaves
- $s \leq \log\log n - 1$ $\textit{has}\ 2^{2^{h-s-1}}\ \textit{children}$
- $s = \log\log n$ $\textit{two children}$
- Node with s <=loglog*n*-1, it has $2^{2^{h-s}}$ leaves, has children with square root size $2^{2^{h-s-1}} = \sqrt{2^{2^{h-s}}}$

PRAM Algorithm 6—最大值

- O(loglog*n*) time and O(*n*loglog*n*) processors
  - Partition data into $n^{1/2}$ pieces



- $n = 2^{2^h}$ leaves
- $s \leq \log\log n - 1$

  has $2^{2^{h-s-1}}$ *children*
- $s = \log\log n$

  two children
- Node with s <=loglog*n*-1, it has $2^{2^{h-s}}$ leaves, has children with square root size

  $2^{2^{h-s-1}} = \sqrt{2^{2^{h-s}}}$

- For each level s, computing max ⬆
- For each node v in s-1 level

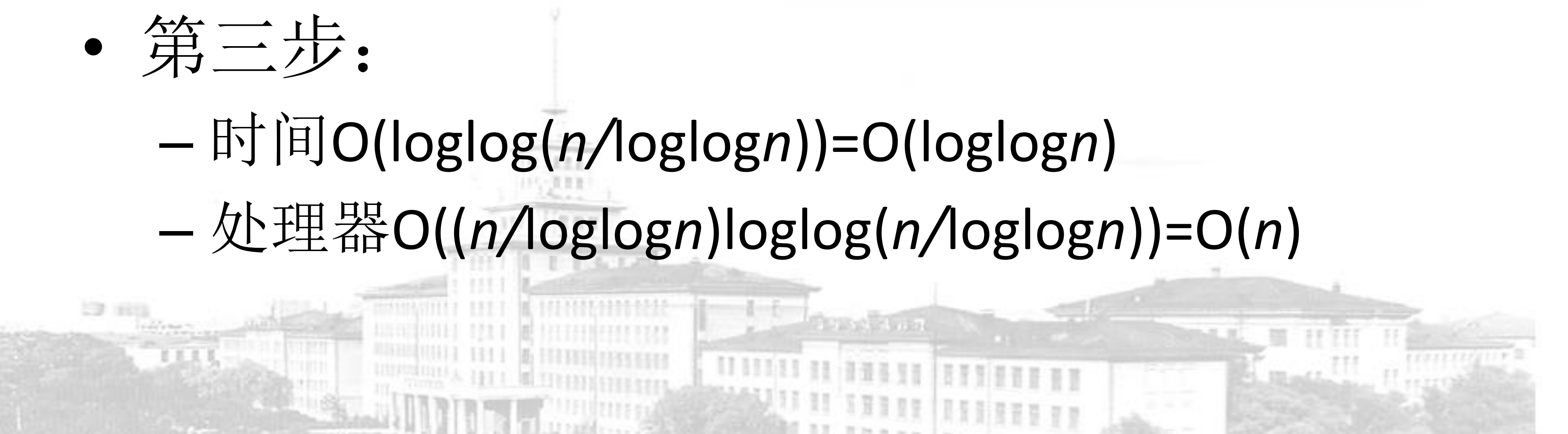
$2^{2^{h-s-2}}$ children
- v can be obtained by O(1)

  $O((2^{2^{h-s-2}})^2) = O(2^{2^{h-s-1}})$
- Total # of v

  n/(levels # of v)

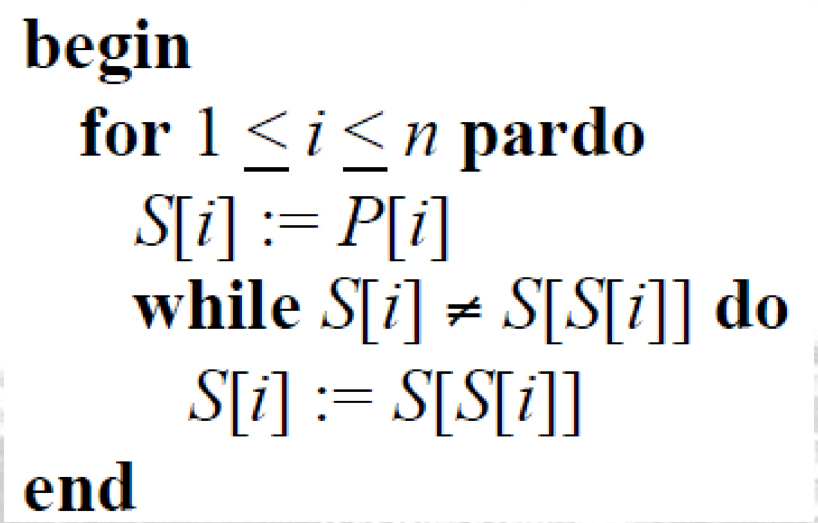
PRAM Algorithm 6—最大值

- O(loglog*n*) time and O(*n*) processors
  - Partition data into loglog*n* pieces
  - 线性比较，求解每一份的最大值
  - 进一步，利用上一个算法求解最大值
- 第二步：O(loglog*n*) time and O(*n*) processors
- 第三步：
  - 时间O(loglog(*n*/loglog*n*))=O(loglog*n*)
  - 处理器O((*n*/loglog*n*)loglog(*n*/loglog*n*))=O(*n*)



PRAM Algorithm 7—求root

- Given an array *P* of *n* integers, P[i]=j if and only if (i,j) is an edge in the forest. A vertex is root if and only if P[i]=[i]
- Output: S[i] is the root of the tree containing I
- Sequential: O(n)

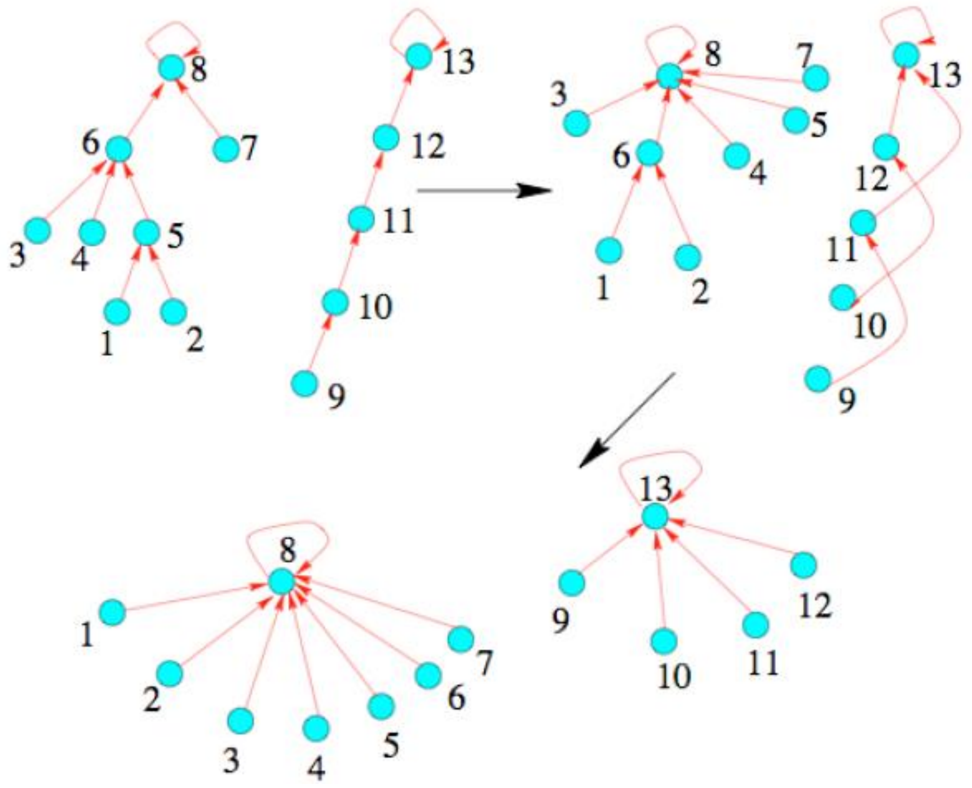


```
begin
  for $1 \leq i \leq n$ pardo
    $S[i] := P[i]$
    while $S[i] \neq S[S[i]]$ do
      $S[i] := S[S[i]]$
end
```

PRAM Algorithm 7—求root

```
begin
  for $1 \leq i \leq n$ pardo
    $S[i] := P[i]$
    while $S[i] \neq S[S[i]]$ do
      $S[i] := S[S[i]]$
end
```

Pointer jumping



$T(n) = O(\log h)$ with h the maximum height of trees
$W(n) = O(n \log h)$

PRAM Algorithm 8—直方图

- Given L[1,…,n] of integers in [1,k], where *k*=log*n*, find how many times each integer occurs in L.
- That is compute a histogram R[1,…,k], such that each R[i] means there are R[i] items in L with value i.

PRAM Algorithm 8—直方图

- Sequential: O(*n*)

```
$R[1:k] \leftarrow 0$
**for** $i = 1$ **to** $n$ **do**
    $R[L[i]] \leftarrow R[L[i]] + 1$
**enddo**
```

PRAM Algorithm 8—直方图

- Parallel: T(n)=O(log*n*), W(n)=O(nk)=O(*n*log*n*)

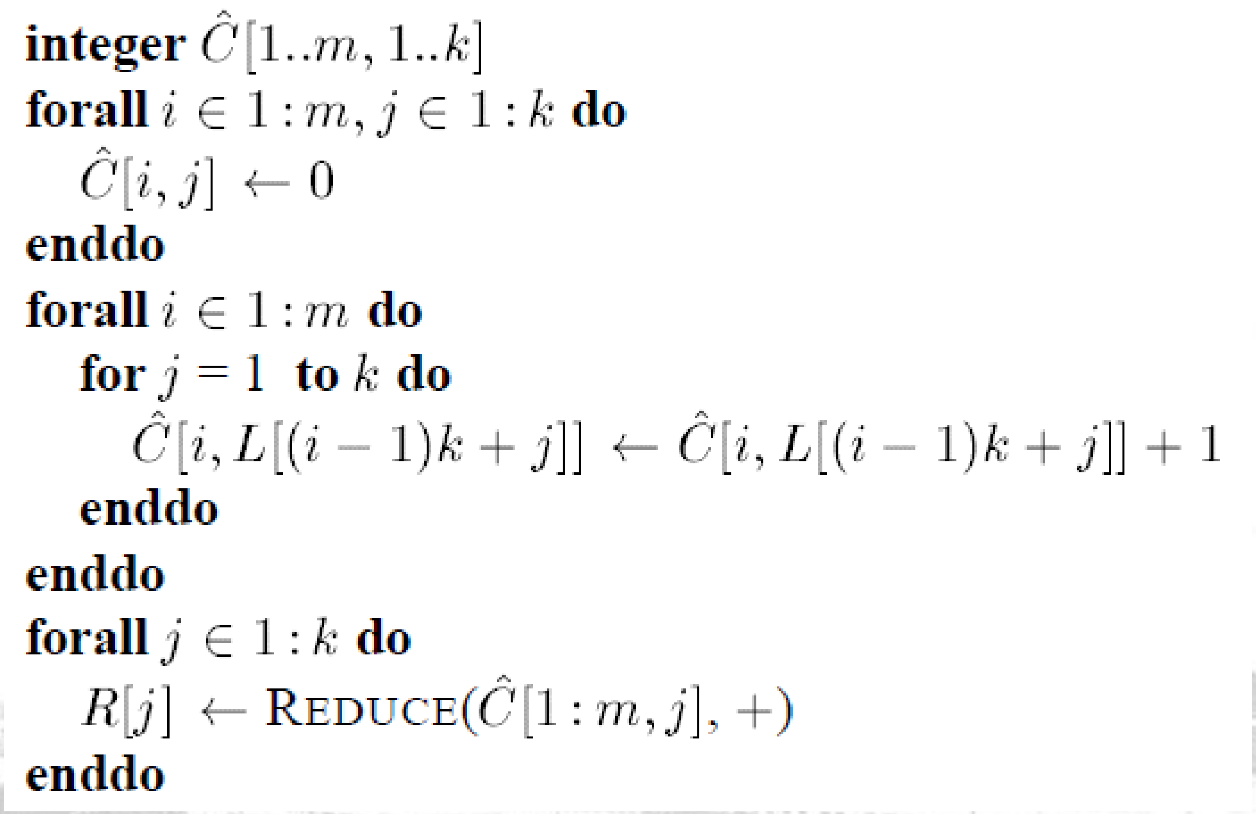
```
forall $i \in 1:n, j \in 1:k$ do
    $C[i,j] \leftarrow 0$
enddo
forall $i \in 1:n$ do
    $C[i, L[i]] \leftarrow 1$
enddo
forall $j \in 1:k$ do
    $R[j] \leftarrow \textsc{Reduce}(C[1:n, j], +)$
enddo
```

$$C_{i,j} = \begin{cases} 1 & \text{if } L[i] = j \\ 0 & \text{otherwise} \end{cases}$$

PRAM Algorithm 8—直方图

- Parallel: T(n)=O(log*n*), W(n)=O(mk)=O(*n*)
  - m=*n*/log*n*

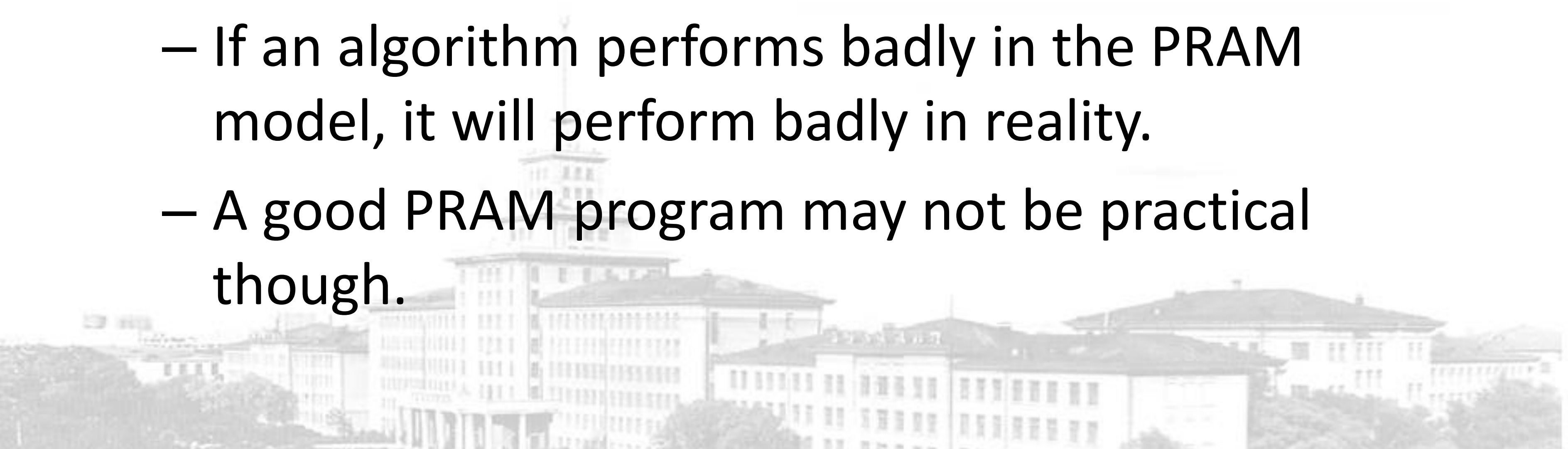
Algorithm cascading



```
integer $\hat{C}[1..m, 1..k]$
forall $i \in 1:m, j \in 1:k$ do
    $\hat{C}[i,j] \leftarrow 0$
enddo
forall $i \in 1:m$ do
    for $j = 1$ to $k$ do
        $\hat{C}[i, L[(i-1)k+j]] \leftarrow \hat{C}[i, L[(i-1)k+j]] + 1$
    enddo
enddo
forall $j \in 1:k$ do
    $R[j] \leftarrow \textsc{Reduce}(\hat{C}[1:m, j], +)$
enddo
```

PRAM strengths

- Natural extension of RAM
- It is simple and easy to understand
  – Communication and synchronization issues are hided.
- Can be used as a benchmark
  – If an algorithm performs badly in the PRAM model, it will perform badly in reality.
  – A good PRAM program may not be practical though.

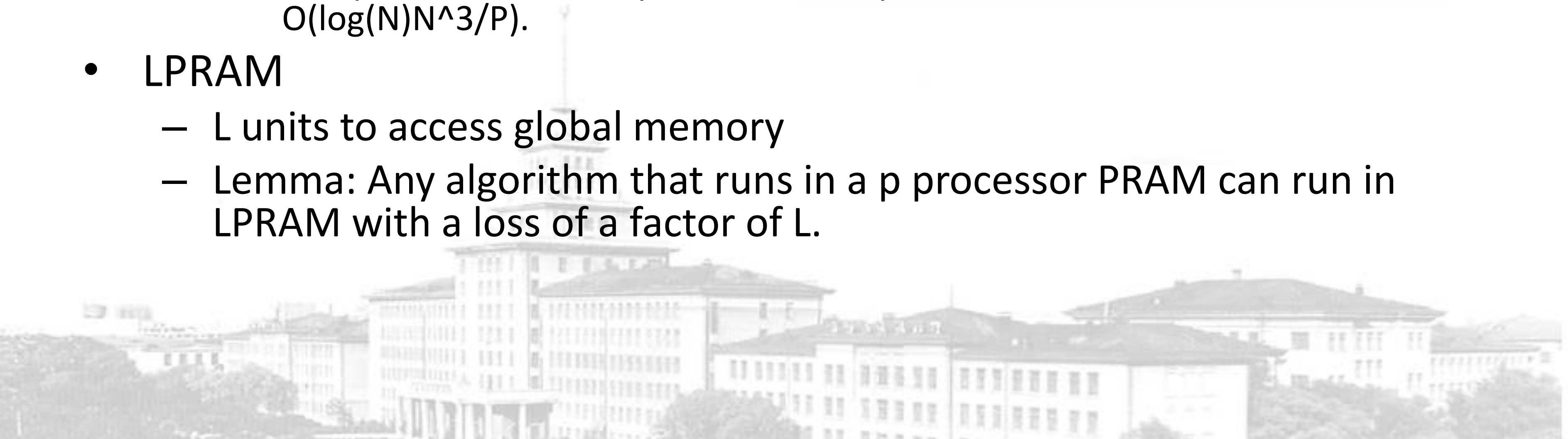


PRAM weaknesses

- Model inaccuracies
  - Unbounded local memory (register)
  - All operations take unit time
  - Processors run in lock steps
- Unaccounted costs
  - Non-local memory access
  - Latency
  - Bandwidth
  - Memory access contention

PRAM variations

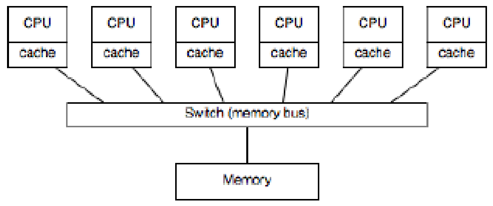
- Bounded memory PRAM, PRAM(m)
  - In a given step, only m memory accesses can be serviced.
  - Lemma: Assume $m'<m$. Any problem that can be solved on a p-processor and m-cell PRAM in t steps can be solved on a $max(p,m')$-processor m'-cell PRAM in $O(tm/m')$ steps.
- Bounded number of processors PRAM
  - Lemma: Any problem that can be solved by a p processor PRAM in t steps can be solved by a p' processor PRAM in t = O(tp/p') steps.
    - E.g. Matrix multiplication PRAM algorithm with time complexity O(log(N)) on N^3 processors→ on P processors, the problem can be solved in O(log(N)N^3/P).
- LPRAM
  - L units to access global memory
  - Lemma: Any algorithm that runs in a p processor PRAM can run in LPRAM with a loss of a factor of L.



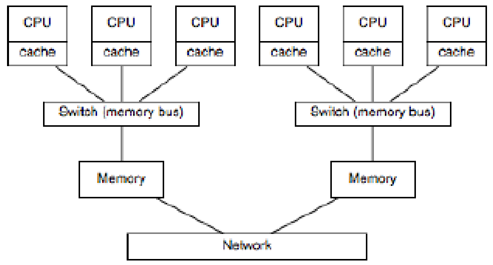
PRAM summary

- The RAM model is widely used.
- PRAM is simple and easy to understand
  - This model never reachs beyond the algorithm community.
  - It is getting more important as threaded programming becomes more popular.
- The BSP (bulk synchronous parallel) model is another try after PRAM.
  - Asynchronously progress
  - Model latency and limited bandwidth

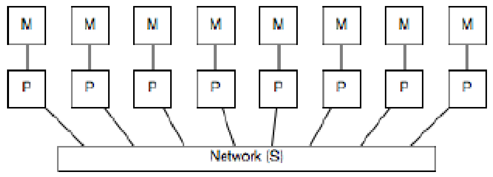
Parallel Algorithm—OpenMP



Shared memory UMA machine with a single bus



Shared memory NUMA machine with memory b



DSM

- Shared Memory
- OpenMP is a portable implementation of common parallel constructs for shared memory machines
- #pragma omp directive_name

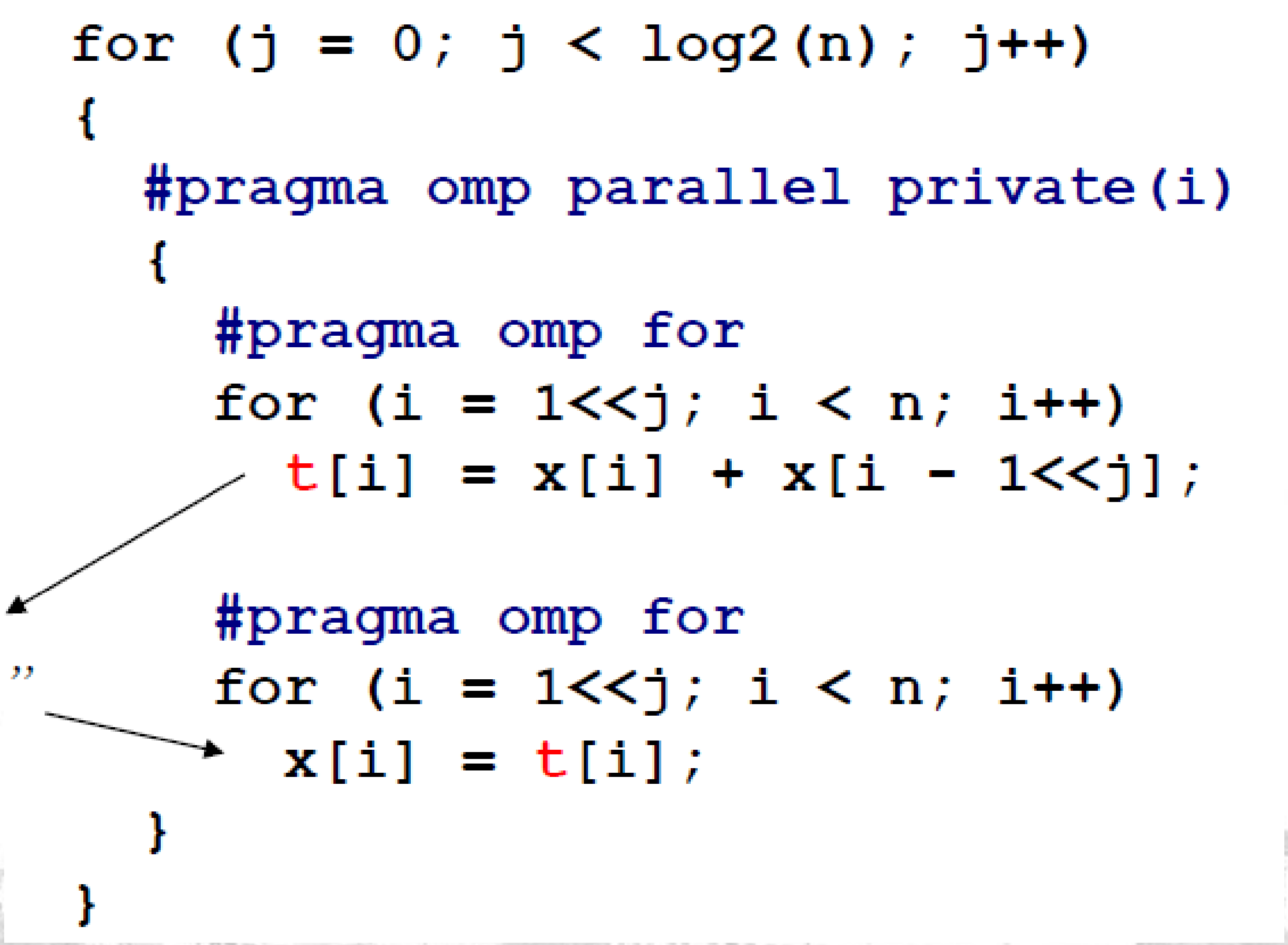
  statement_block

#pragma omp atomic

#pragma omp flush(variables)

#pragma omp barrier

Parallel Algorithm—OpenMP



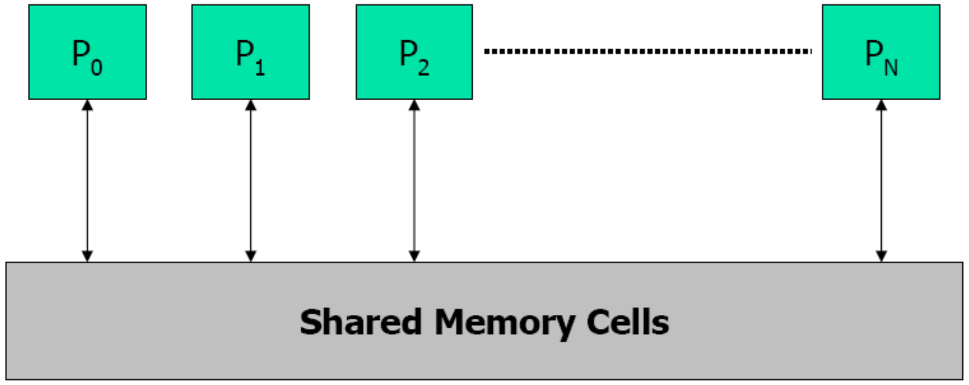
```
for (j = 0; j < log2(n); j++)
{
  #pragma omp parallel private(i)
  {
    #pragma omp for
    for (i = 1<<j; i < n; i++)
      t[i] = x[i] + x[i - 1<<j];

    #pragma omp for
    for (i = 1<<j; i < n; i++)
      x[i] = t[i];
  }
}
```

Parallel Algorithm—MPI

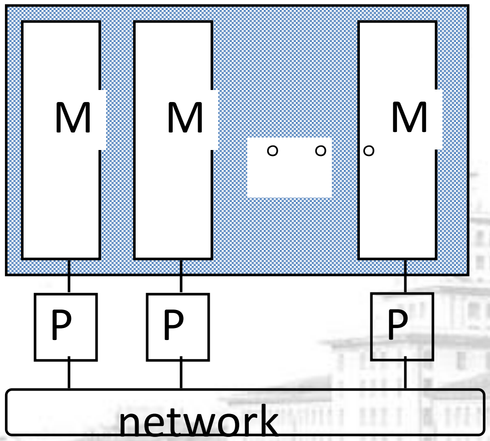
- Message Passing
- Message Passing Interface (MPI) is an API and protocol standard with portable implementations

```
main(int argc, char *argv[])
{
  int myrank;
  int value = 123;
  MPI_Status status;

  MPI_Init(&argc, &argv);

  MPI_Comm_Rank(MPI_COMM_WORLD, &myrank);

  if (myrank == 0)
    MPI_Send(&value, 1, MPI_INT, 1, MPI_ANY_TAG, MPI_COMM_WORLD);
  else if (myrank == 1)
    MPI_Recv(&value, 1, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status);

  MPI_Finalize();
}
```

LogP model



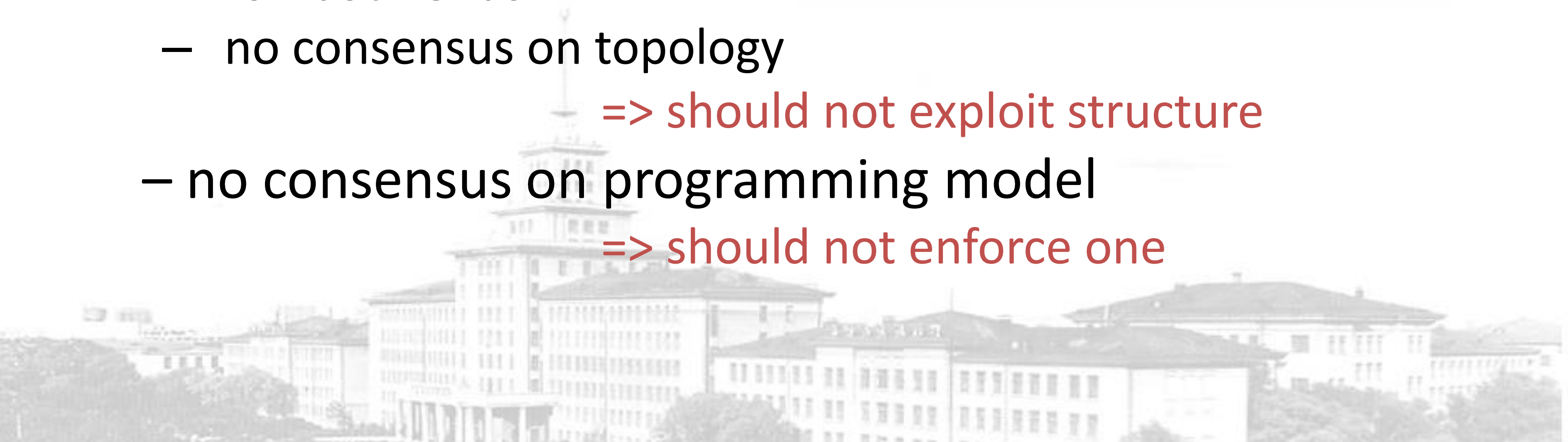
PRAM model: shared memory



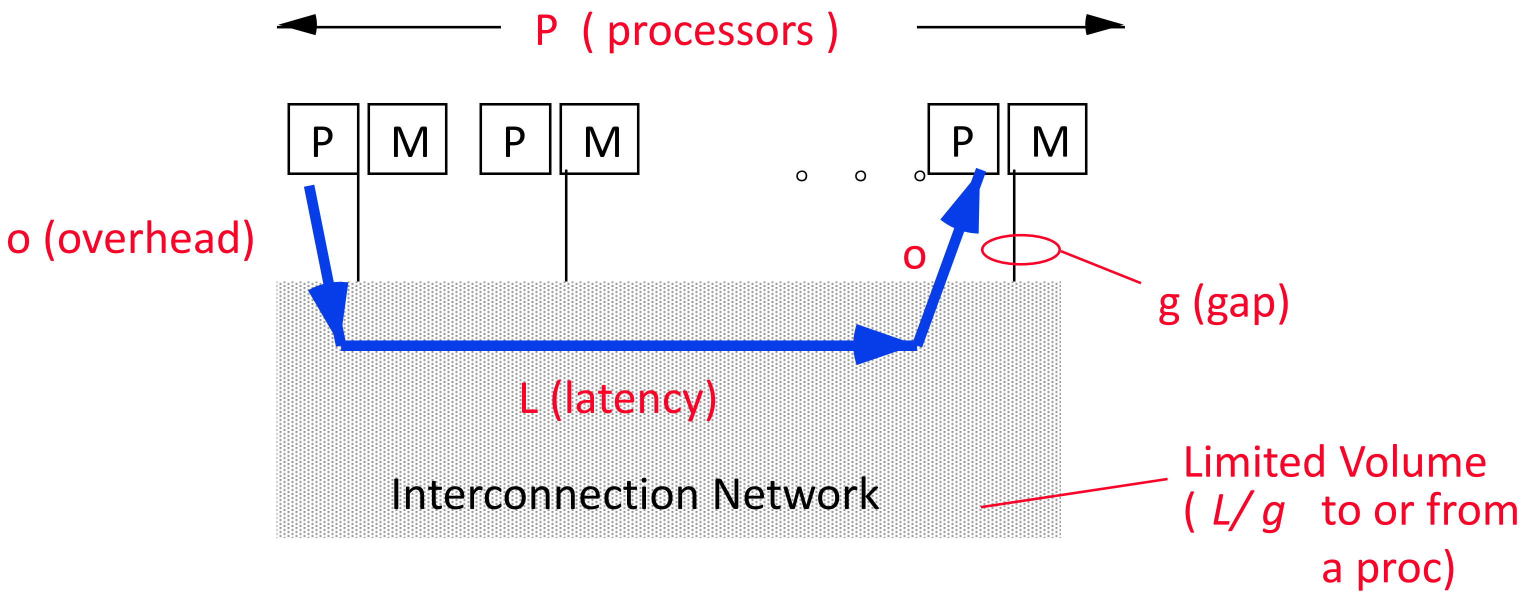
- Common MPP organization: complete machine connected by a network.
- LogP attempts to capture the characteristics of such organization.

Deriving LogP model

- Processing
  – powerful microprocessor => P
- Communication
  + significant latency => L
  + limited bandwidth => g
  + significant overhead => o
    - on both ends
  – no consensus on topology
    => should not exploit structure
  – no consensus on programming model
    => should not enforce one



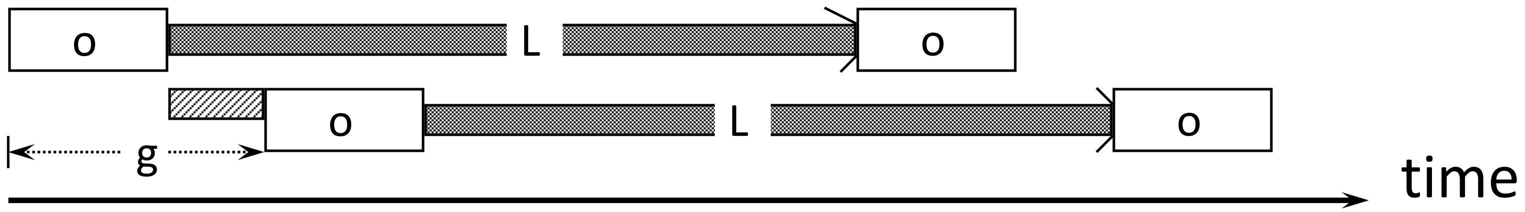
LogP



- Latency in sending a (small) mesage between modules
- overhead felt by the processor on sending or receiving msg
- gap between successive sends or receives (1/BW)
- Processors

Using the model

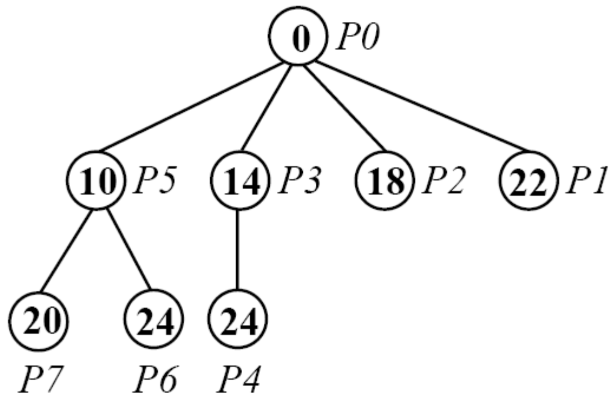
- Two processors send n words to each other:
  - *2o + L + g(n-1)*



- Assumes no network contention
- Can under-estimate the communication time.

Develop efficient broadcast algorithm based on the LogP model

- Broadcast a single datum to P-1 processors



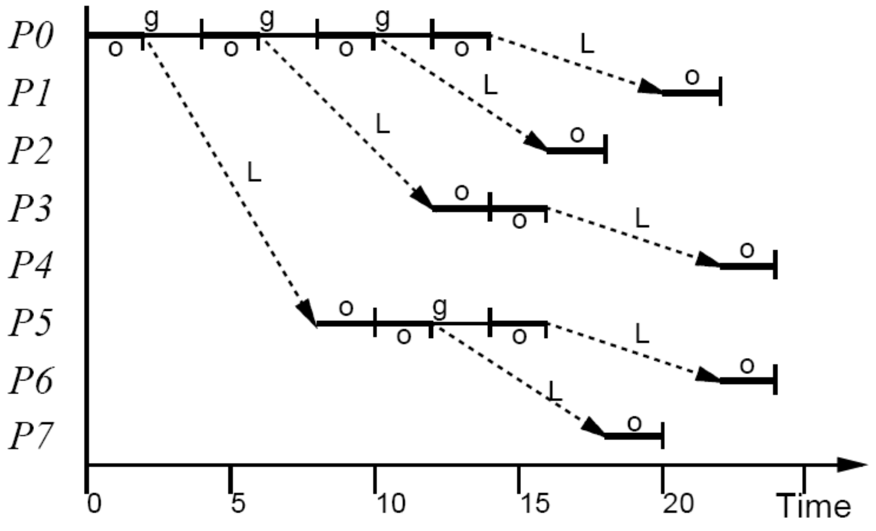


Figure 3: *Optimal broadcast tree for $P = 8, L = 6, g = 4, o = 2$ (left) and the activity of each processor over time (right). The number shown for each node is the time at which it has received the datum and can begin sending it on. The last value is received at time 24.*

Strengths of the LogP model

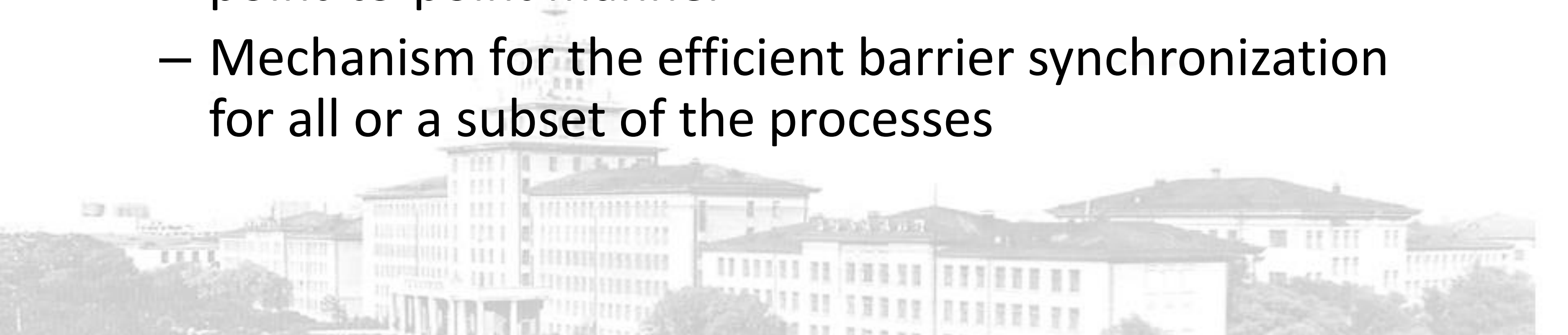
- Simple, 4 parameters
- Can easily be used to guide the algorithm development, especially algorithms for communication routines.
  - This model has been used to analyze many collective communication algorithms.

Weaknesses of the LogP model

- Accurate only at the very low level (machine instruction level)
  - Inaccurate for more practical communication systems with layers of protocols (e.g. TCP/IP)
  - Many variations.
    - LogP family models: LogGP, logGPC, pLogP, etc
      - Making the model more accurate and more complex

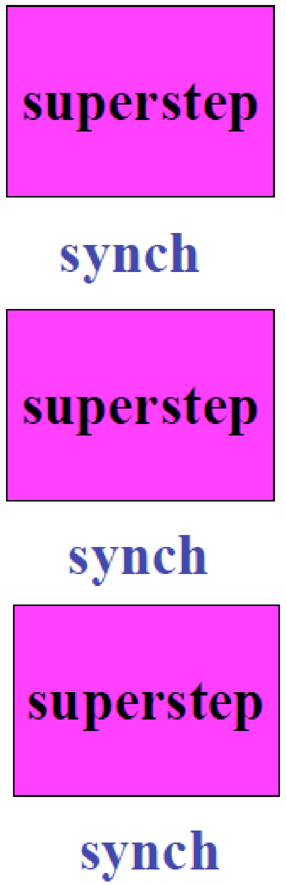
BSP (Bulk synchronous Parallel)

- The BSP abstract computer is a bridging model for designing parallel algorithms
  - Something between hardware and programming model.
- A BSP computer consists of
  - A set of processor-memory pairs
  - A communication network that delivers messages in a point-to-point manner
  - Mechanism for the efficient barrier synchronization for all or a subset of the processes



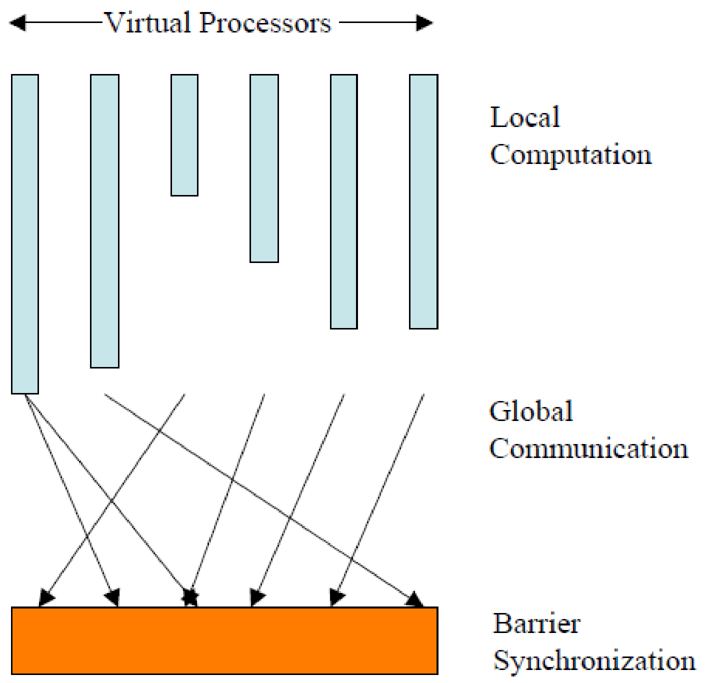
BSP programs

- BSP programs composed of supersteps
  - In each superstep consists of three ordered stages:
    - Computation (up to a certain unit)
    - Communication
    - Barrier synchronization



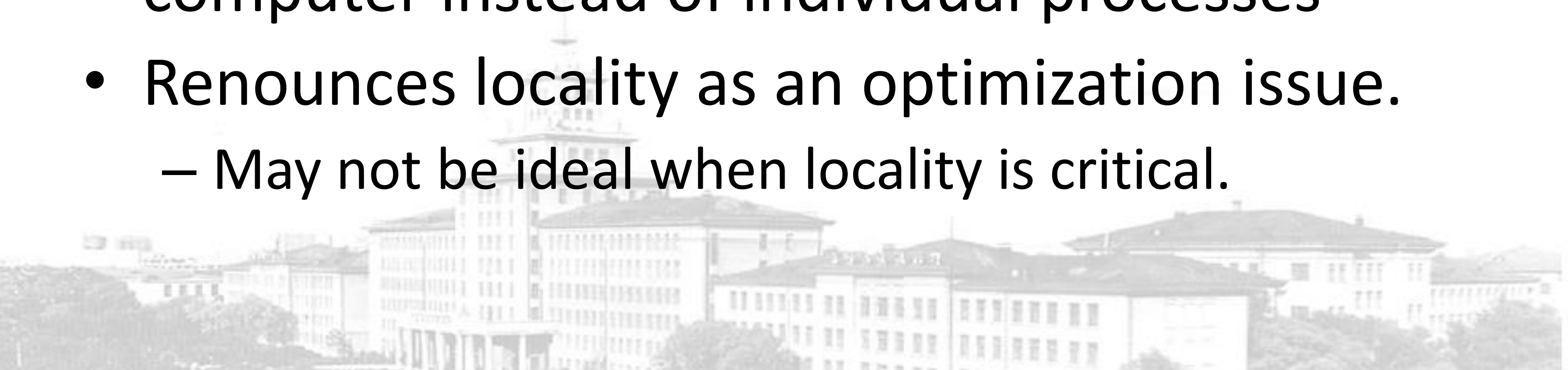
BSP programs

- Vertical structure
  - A sequence of supersteps
    - Local computation
    - Communication
    - Barrier synchronization
- Horizonal structure
  - Concurrency among a fixed number of virtual processors
  - Processes do not have a particular order
  - Locality plays no role
  - P = number of processors



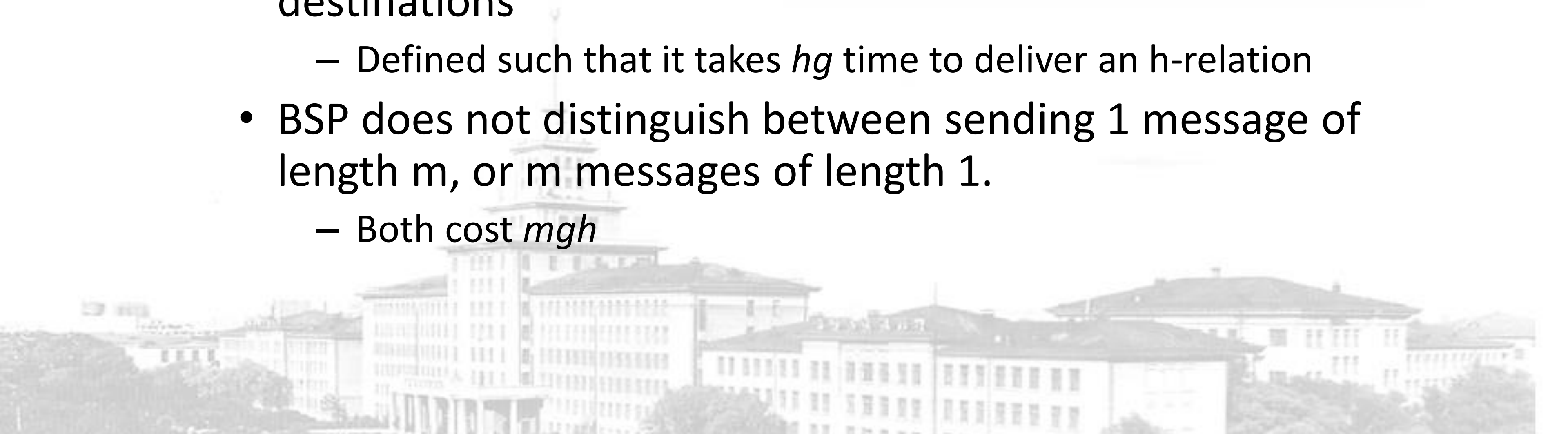
BSP programming style

- Properties
  - Simple to write programs
  - Independent of target architecture
  - Performance of the model is predictable
- Considers computation and communication at the level of the entire program and executing computer instead of individual processes
- Renounces locality as an optimization issue.
  - May not be ideal when locality is critical.



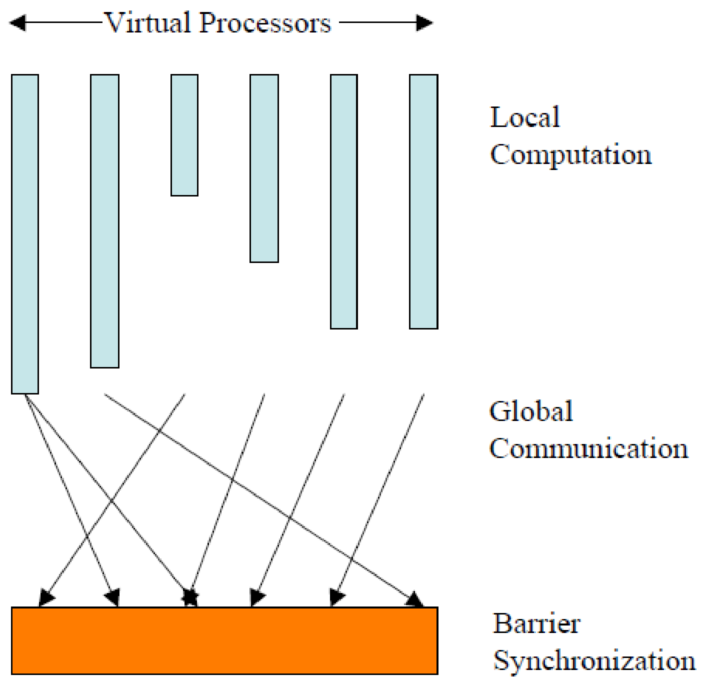
BSP communications

- BSP considers communication *en masse*
  - bound the total time to deliver the whole set of data in a superstep.
    - *h*-relation: the maximum number of incoming or outgoing messages per processor
    - Parameter *g* measures the permeability of the network to continuous traffic addressed to uniformly random destinations
      - Defined such that it takes *hg* time to deliver an h-relation
    - BSP does not distinguish between sending 1 message of length m, or m messages of length 1.
      - Both cost *mgh*



BSP barrier synchronization

- The cost has two parts:
  - Variation in completion time of computation step
  - The cost of reach globally consistent state in all processors
- Cost is captured by parameter l
  - Lower bound on l is a function of the diameter of the networks



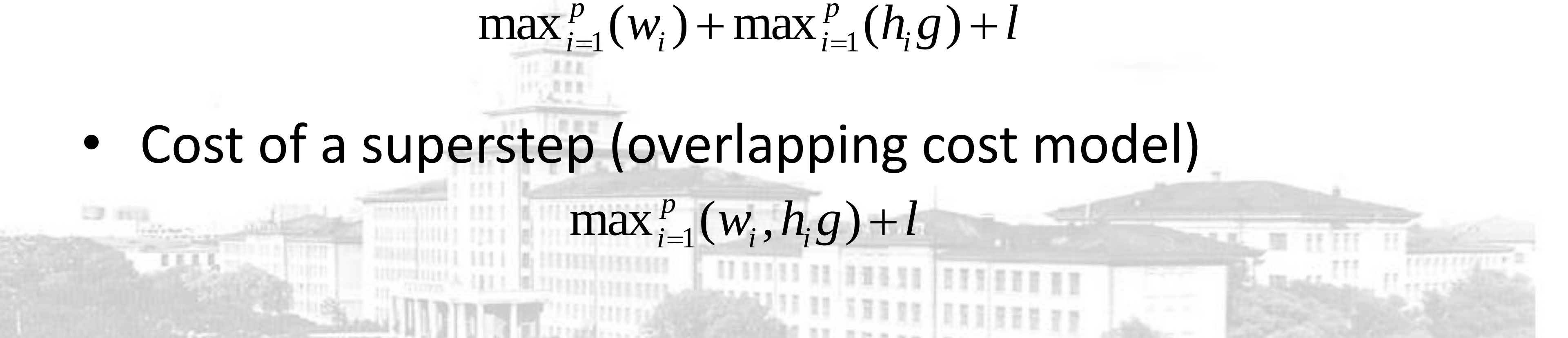
Predictability of the BSP model

- A BSP computer is modeled by:
  - P: number of processors
  - S: processor computation speed (flops/s), used to calibrate g and l
  - l: synchronization periodicity; minimal number of time steps between successive synchronization operations
  - g: the cost of communication so that an h-relation is realized within *gh* steps.
- Cost of a super step (standard cost model)

$$\max_{i=1}^{p}(w_i) + \max_{i=1}^{p}(h_i g) + l$$

- Cost of a superstep (overlapping cost model)

$$\max_{i=1}^{p}(w_i, h_i g) + l$$

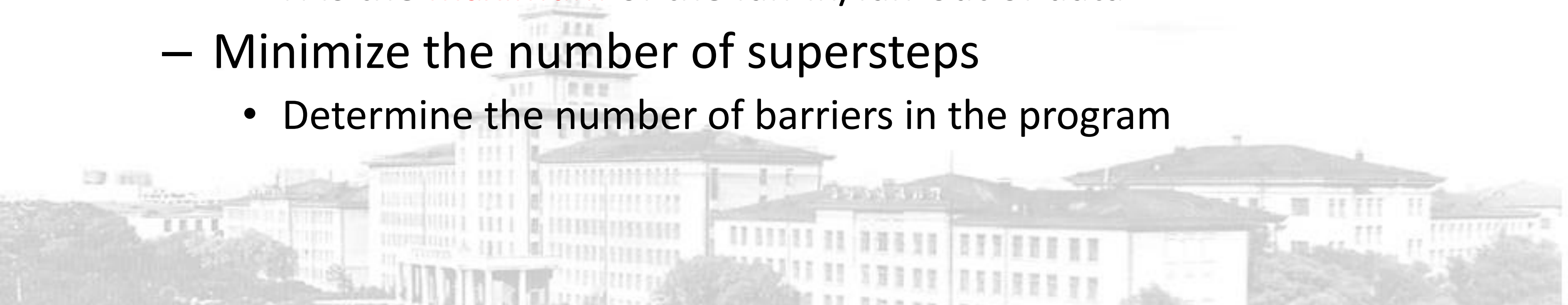


Cost of a BSP algorithm

- The sum of the costs of all S supersteps

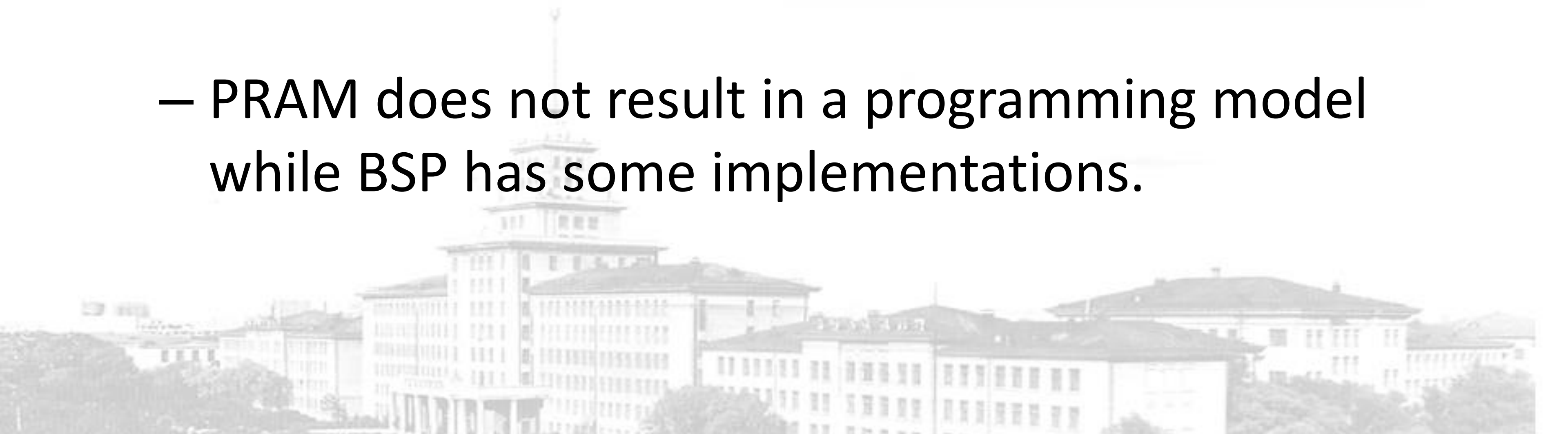
$$W + Hg + Sl = \sum_{s=1}^{S} w_s + g\sum_{s=1}^{S} h_s + Sl$$

- Strategies used in writing efficient BSP programs
  - Balance the computation in each superstep between processes
    - W is the maximum of all computation times in different processors
  - Balance the communication between processes
    - H is the maximum of the fan-in/fan-out of data
  - Minimize the number of supersteps
    - Determine the number of barriers in the program



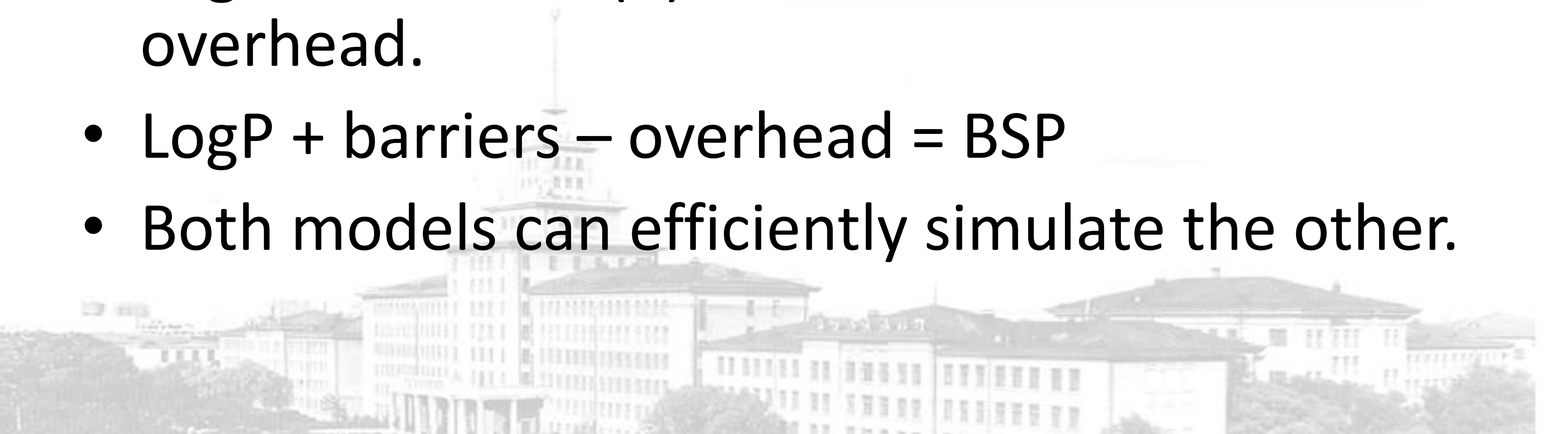
BSP and PRAM

- BSP is a generalization of PRAM
  - Processes in a superstep can have different computation time
  - Communication and synchronization costs are explicitly taken into consideration
  - PRAM does not result in a programming model while BSP has some implementations.



BSP and LogP

- Communication in LogP has a “local” view, based on per pair performance, communication in BSP has a “global” view, based on the performance for the whole program
- LogP has a term (o) for the communication overhead.
- LogP + barriers – overhead = BSP
- Both models can efficiently simulate the other.



PRAM, BSP, LogP summary

- All are fairly simple and can be used to guide parallel algorithm development.

- Simplicity is necessary to be useful for guiding algorithm development, but results in inaccuracy for performance modeling.
  - Many extensions have proposed to refine the models: trade simplicity for accuracy.