

# COL380: Introduction to Parallel & Distributed Programming

# We will Study ...

- Concurrency
- Parallelism
- Distributed computing

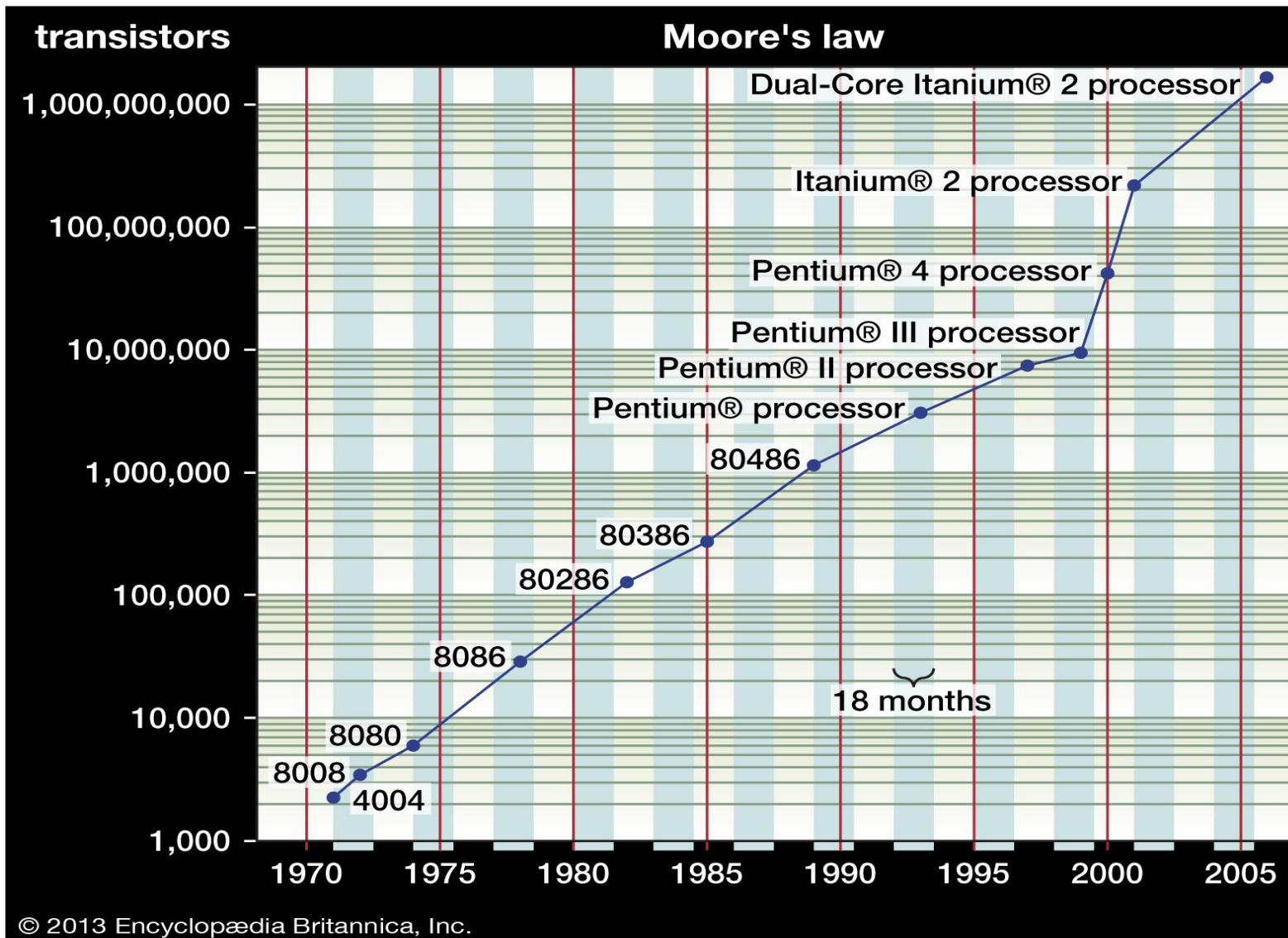
# Evaluation

- Assignments      40%,
- Minor-1            15%,
- Minor-2            15%,
- Major              30%

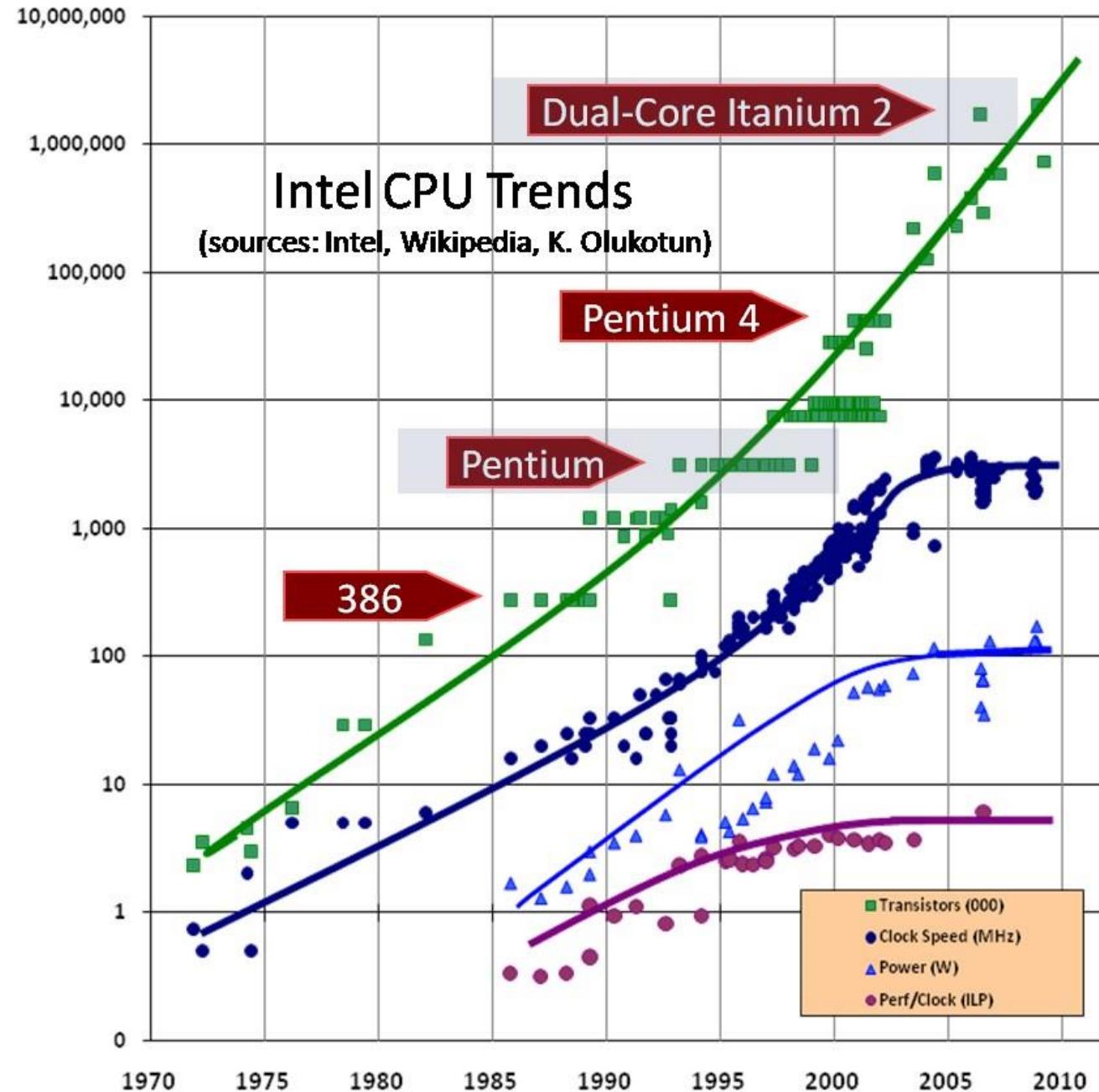
Plagiarism is unacceptable. Offenders will be penalized by a failing grade.

# Moore's Law

the number of transistors in a dense integrated circuit (IC) doubles about every two years.



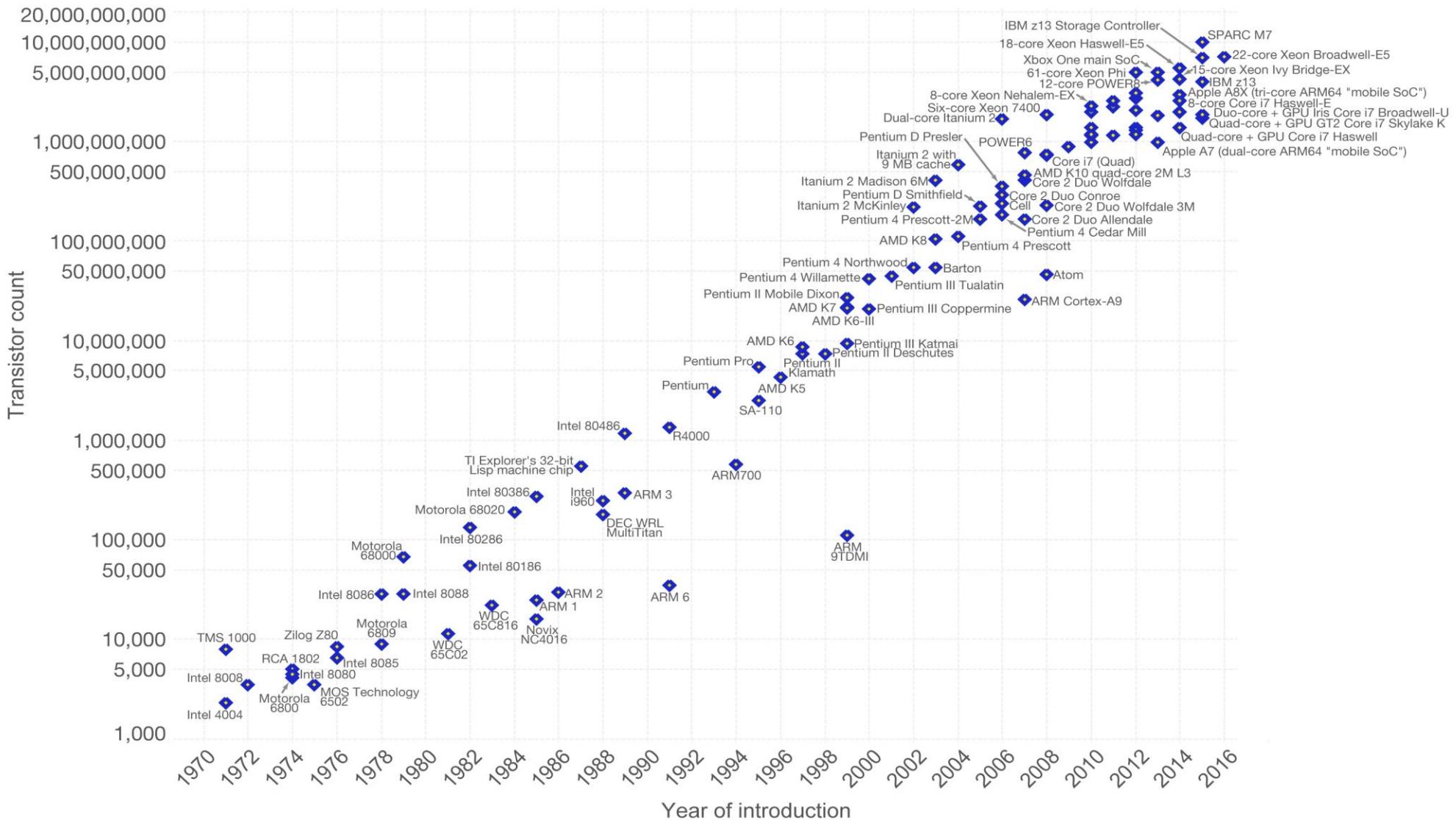
# CPU Trends



# CPU Trends

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

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](https://en.wikipedia.org/wiki/Transistor_count))

The data visualization is available at [OurWorldInData.org](http://OurWorldInData.org). There you find more visualizations and research on this topic.

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

# Sequential Computation

## Bottleneck of Sequential Computation

Single processor performance increases with the as transistor density increases.

Increase in transistor density increases power consumption and result in heating problem.

Heating problem results in unreliable computation.

**Additional technique to improve performance:** *Parallelism*

# Benefits

Parallelism increase computation power

Many important applications

- Climate modeling
- Protein folding
- Drug discovery
- Data analysis
- ...

# Parallelism and Parallel Computing

Can a system automatically parallelize a sequential program?

Specific cases

- Automatic parallelization by compiler
- Instruction level parallelism in architecture

*Cannot exploit all possible opportunities*

Parallel Programming: Device parallel algorithm and program that solves a problem in more efficient manner

# Example

Problem: *Compute n values and sum them together.*

## Sequential Program

```
int sum = 0;  
for (i = 0; i < n; i++) {  
    x = f(i);  
    sum = sum+x;  
}
```

# Parallel Program: Compute Partial Sum

Assumption: p processors where  $p \ll n$

```
my_sum = 0;  
my_first i = . . . ;  
my_last i = . . . ;  
for (my_i = my_first i; my_i < p_last_i; my_i++) {  
    my_x = f(i);  
    my_sum += my_x;  
}
```

1,4,3, 9,2,8, 5,1,1, 6,2,7, 2,5,0, 4,1,8, 6,5,1, 2,3,9,

Core	0	1	2	3	4	5	6	7
my_sum	8	19	7	15	7	13	12	14

# Parallel Program: Accumulate Partial Sum

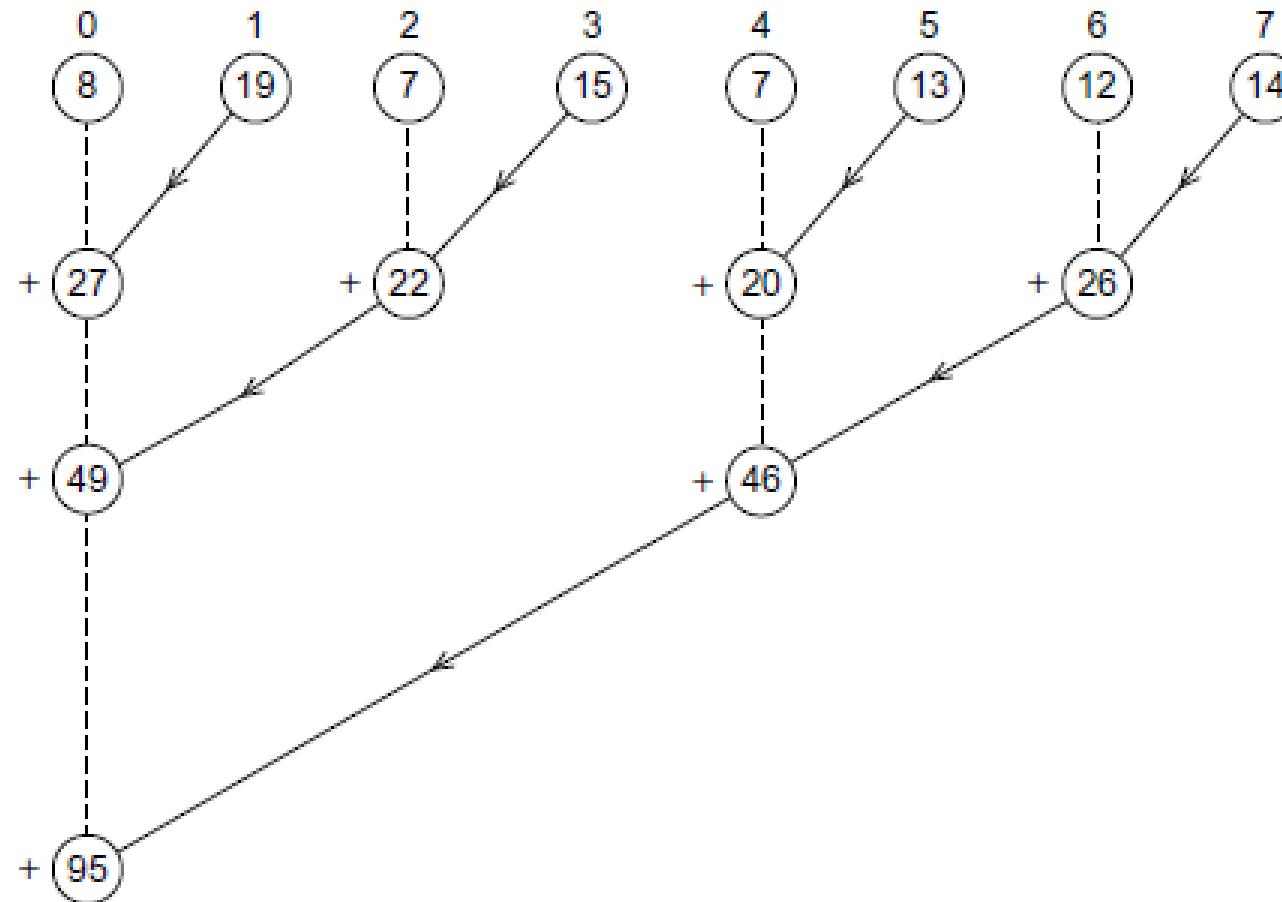
```
if (I'm the master core) {  
    sum = my_sum;  
    for (each core other than myself) {  
        receive value from core;  
        sum += value;  
    }  
} else { send my_sum to the master; }
```

Core	0	1	2	3	4	5	6	7
my_sum	8	19	7	15	7	13	12	14

$$\begin{aligned} & 8 + 19 + 7 + 15 + 7 + 13 + 12 + 14 \\ & = 95 \end{aligned}$$

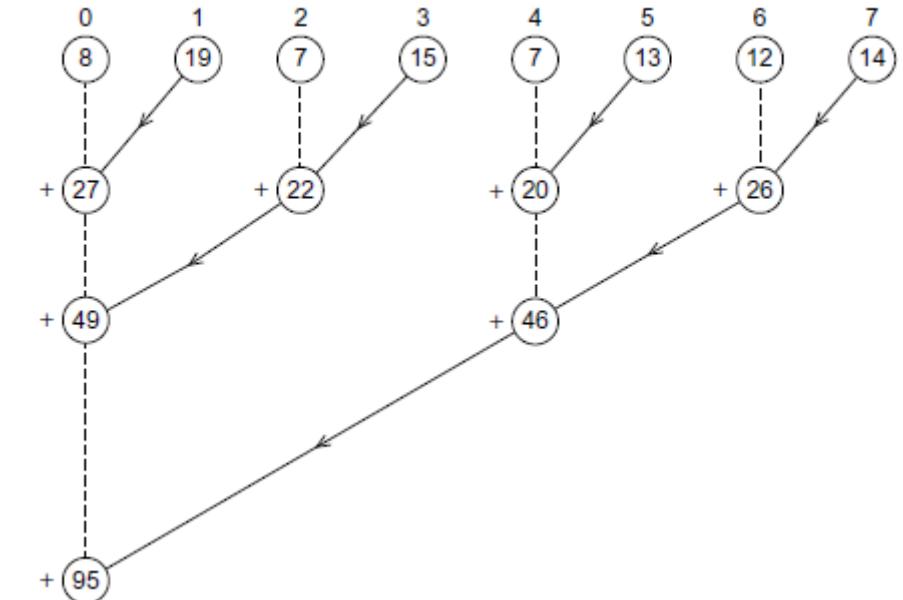
# Parallel Program: Second Attempt

*No constraint on the number of available processors*



# Comparing two Attempts

- (1) Compute partial sum of n/p elements.
- (2) accumulate results of partial sum



Both approaches have same number of addition operations

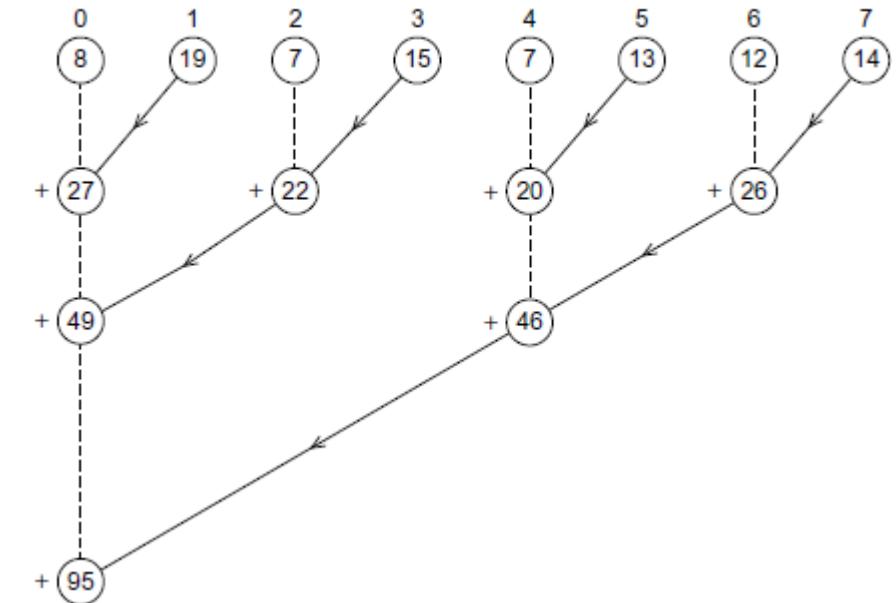
# Comparing two Attempts

## Parallel computation

(1) Compute partial sum of n/p elements.

## Serial computation

(2) accumulate results of partial sum

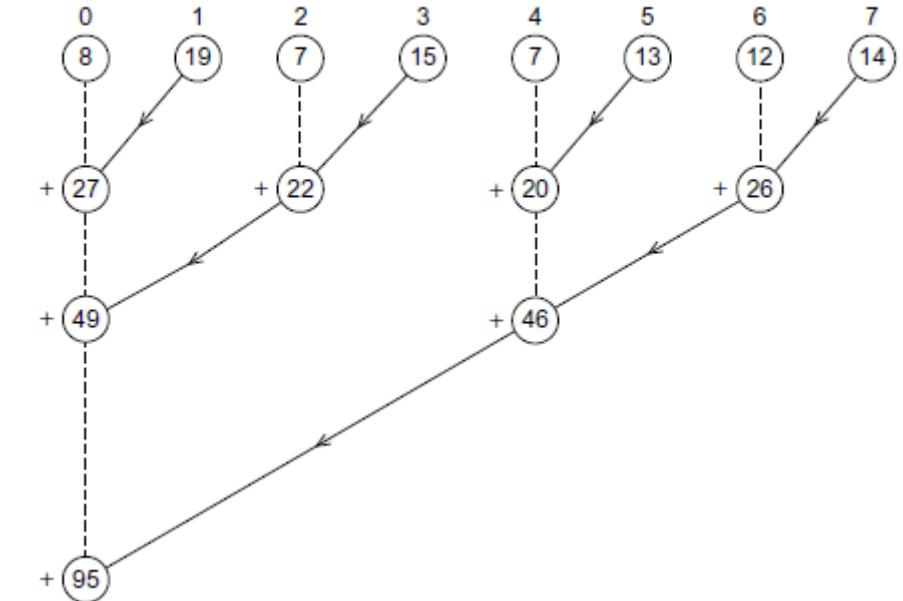


Step (2) of the first approach is serialized

# Comparing two Attempts

(1) Compute partial sum of n/p elements.

(2) accumulate results of partial sum



First approach is closer to the sequential sum program

# Additional Concerns

Communication

- shared memory, message passing, ...

Coordination

- synchronization

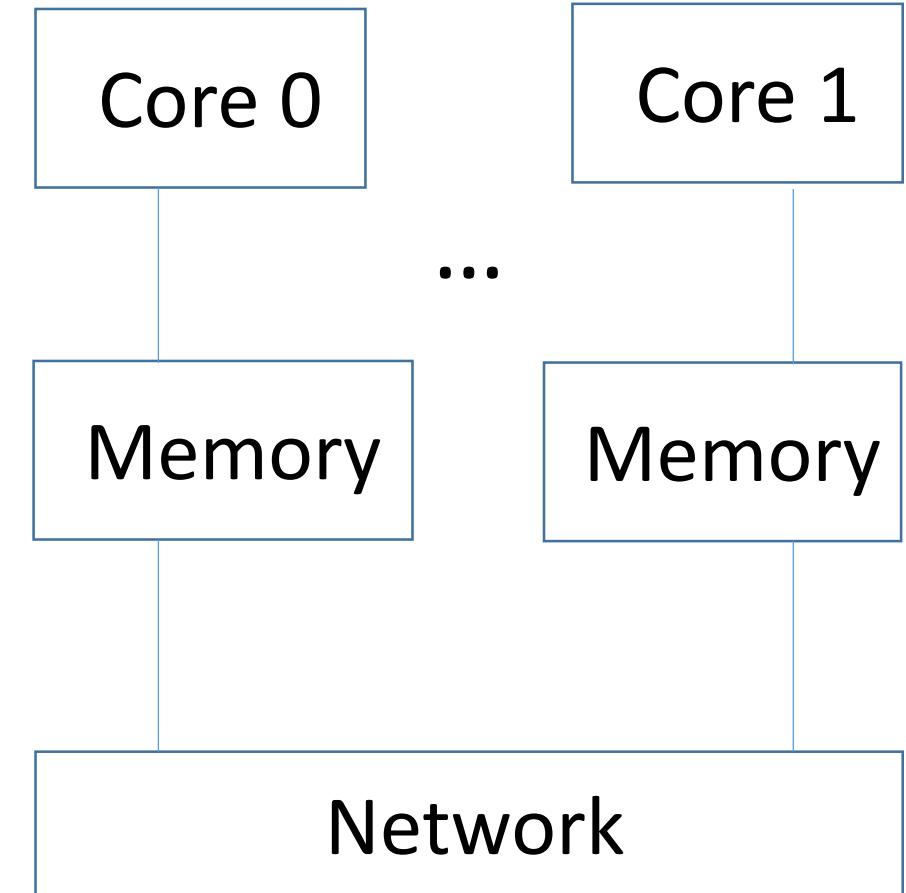
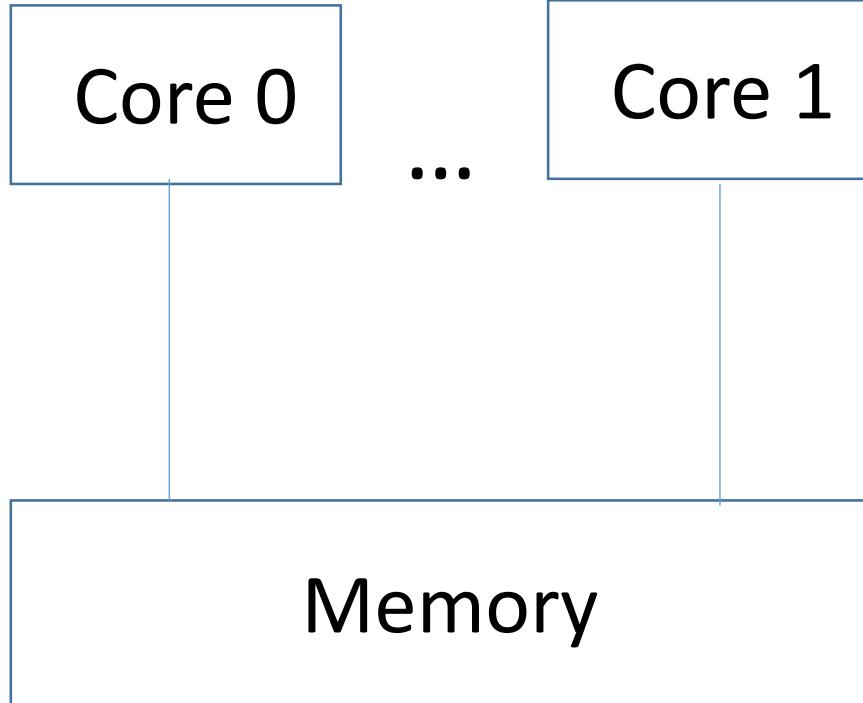
Job distribution

- load balancing

# Different Models of Computation

- Concurrency  
Shared memory programs:  
OpenMP
  - Parallelism  
Message passing:  
MPI
  - Distributed computing

# Memory & Communication Play a Significant Role



# References

## Chapter 1

- An Introduction to Parallel Programming  
by Peter Pacheco.
- Introduction to Parallel Computing, Second Edition  
by Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar

# Performance Analysis

# Recap

- Concurrency, Parallelism, Distributed computing
- Two parallelization approaches for summation computation

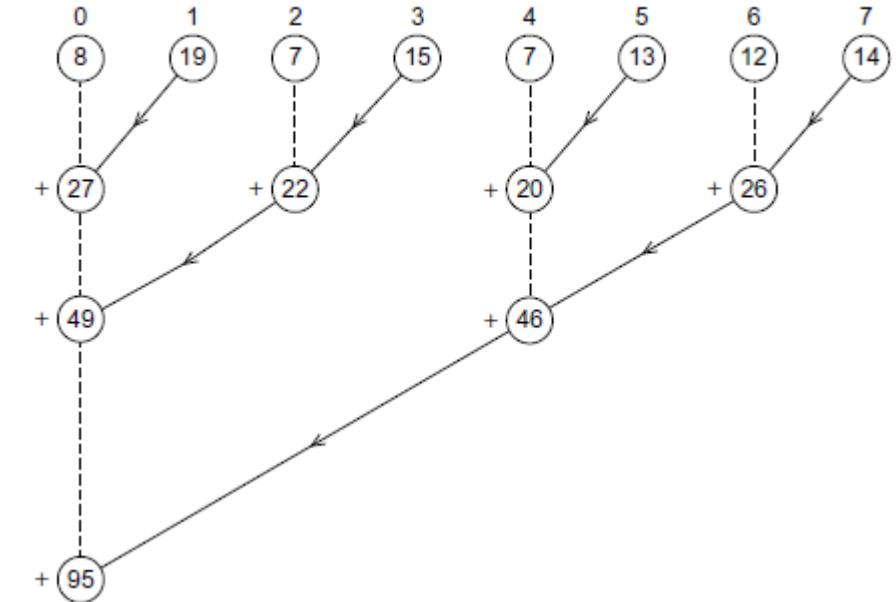
# Comparing two Attempts

## Parallel computation

(1) Compute partial sum of n/p elements.

## Serial computation

(2) accumulate results of partial sum



Step (2) of the first approach is serialized

# Analyzing Performance

Remember the example

1,4,3, 9,2,8, 5,1,1, 6,2,7, 2,5,0, 4,1,8, 6,5,1, 2,3,9,

Core	0	1	2	3	4	5	6	7
my_sum	8	19	7	15	7	13	12	14

What if we have 4 processors?

# Speedup

$T_{\text{serial}}$ : Execution time of the serial program

$T_{\text{parallel}}$ : Execution time of the parallel program

Speedup  $S = T_{\text{serial}} / T_{\text{parallel}}$

# Linear Speedup

Suppose we have  $p$  processors

If  $(T_{\text{parallel}} == T_{\text{serial}} / p)$  then the speedup is linear

In case of linear speedup

$$S = T_{\text{serial}} / T_{\text{parallel}} = p$$

*Linear speedup is ideal but unusual.*

# Speedup Example

**Table 3.6 Speedups of Parallel Matrix-Vector Multiplication**

comm_sz	Order of Matrix				
	1024	2048	4096	8192	16,384
1	1.0	1.0	1.0	1.0	1.0
2	1.8	1.9	1.9	1.9	2.0
4	2.1	3.1	3.6	3.9	3.9
8	2.4	4.8	6.5	7.5	7.9
16	2.4	6.2	10.8	14.2	15.5

Observation: speedup vs p vs problem size

# Efficiency

efficiency:  $E = S / p$  i.e. speedup per processor

$$E = (T_{\text{serial}} / T_{\text{parallel}}) / p$$

$$E = T_{\text{serial}} / (p * T_{\text{parallel}})$$

# Efficiency Example

**Table 3.7** Efficiencies of Parallel Matrix-Vector Multiplication

comm_sz	Order of Matrix				
	1024	2048	4096	8192	16,384
1	1.00	1.00	1.00	1.00	1.00
2	0.89	0.94	0.97	0.96	0.98
4	0.51	0.78	0.89	0.96	0.98
8	0.30	0.61	0.82	0.94	0.98
16	0.15	0.39	0.68	0.89	0.97

# Speedup vs Efficiency

**Table 2.4** Speedups and Efficiencies of a Parallel Program

$p$	1	2	4	8	16
S	1.0	1.9	3.6	6.5	10.8
E = S/p	1.0	0.95	0.90	0.81	0.68

**Table 2.5** Speedups and Efficiencies of a Parallel Program on Different Problem Sizes

	$p$	1	2	4	8	16
Half	S	1.0	1.9	3.1	4.8	6.2
	E	1.0	0.95	0.78	0.60	0.39
Original	S	1.0	1.9	3.6	6.5	10.8
	E	1.0	0.95	0.90	0.81	0.68
Double	S	1.0	1.9	3.9	7.5	14.2
	E	1.0	0.95	0.98	0.94	0.89

*Speedup and efficiency increase with problem size*

# Amdahl's Law

Unless the entire serial program is parallelized, the possible speedup is going to be limited regardless of the number of processors by the sequential component(unparallelized fraction) of a program.

f: parallelized fraction of a program

1-f: sequential component(unparallelized fraction) of a program

$$T_{\text{parallel}} = f * (T_{\text{serial}} / p) + (1-f) * T_{\text{serial}}$$

$$\text{Speedup } S = T_{\text{serial}} / T_{\text{parallel}} = T_{\text{serial}} / (f * (T_{\text{serial}} / p) + (1-f) * T_{\text{serial}})$$

# Example

Let  $T_{\text{serial}} = 20$ ,  $f = 0.9$ , and

$p = 2 \Rightarrow$

$$T_{\text{parallel}} = f * (T_{\text{serial}} / p) + (1-f) * T_{\text{serial}} = 0.9 * 20/2 + 0.1 * 20 = 9 + 2 = 11$$

# Example

Let  $T_{\text{serial}} = 20$ ,  $f = 0.9$ , and

$p = 2 \Rightarrow$

$$T_{\text{parallel}} = f * (T_{\text{serial}} / p) + (1-f) * T_{\text{serial}} = 0.9 * 20/2 + 0.1 * 20 = 9 + 2 = 11$$

$p = 4 \Rightarrow$

$$T_{\text{parallel}} = f * (T_{\text{serial}} / p) + (1-f) * T_{\text{serial}} = 0.9 * 20/4 + 0.1 * 20 = 4.5 + 2 = 6.5$$

# Example

Let  $T_{\text{serial}} = 20$ ,  $f = 0.9$ , and

$p = 2 \Rightarrow$

$$T_{\text{parallel}} = f * (T_{\text{serial}} / p) + (1-f) * T_{\text{serial}} = 0.9 * 20/2 + 0.1 * 20 = 9 + 2 = 11$$

$p = 4 \Rightarrow$

$$T_{\text{parallel}} = f * (T_{\text{serial}} / p) + (1-f) * T_{\text{serial}} = 0.9 * 20/4 + 0.1 * 20 = 4.5 + 2 = 6.5$$

$p = 10 \Rightarrow$

$$T_{\text{parallel}} = f * (T_{\text{serial}} / p) + (1-f) * T_{\text{serial}} = 0.9 * 20/10 + 0.1 * 20 = 1.8 + 2 = 3.8$$

# Example

Let  $T_{\text{serial}} = 20$ ,  $f = 0.9$ , and

$p = 2 \Rightarrow$

$$T_{\text{parallel}} = f * (T_{\text{serial}} / p) + (1-f) * T_{\text{serial}} = 0.9 * 20/2 + 0.1 * 20 = 9 + 2 = 11$$

$p = 4 \Rightarrow$

$$T_{\text{parallel}} = f * (T_{\text{serial}} / p) + (1-f) * T_{\text{serial}} = 0.9 * 20/4 + 0.1 * 20 = 4.5 + 2 = 6.5$$

$p = 10 \Rightarrow$

$$T_{\text{parallel}} = f * (T_{\text{serial}} / p) + (1-f) * T_{\text{serial}} = 0.9 * 20/10 + 0.1 * 20 = 1.8 + 2 = 3.8$$

$p = 20 \Rightarrow$

$$T_{\text{parallel}} = f * (T_{\text{serial}} / p) + (1-f) * T_{\text{serial}} = 0.9 * 20/20 + 0.1 * 20 = 0.8 + 2 = 2.8$$

# Amdahl's Law

unless the entire serial program is parallelized, the possible speedup is going to be very limited regardless of the number of processors.

$$S = T_{\text{serial}} / T_{\text{parallel}} = T_{\text{serial}} / (f * (T_{\text{serial}} / p) + (1-f) * T_{\text{serial}})$$

$$S \approx T_{\text{serial}} / ((1-f) * T_{\text{serial}}) \text{ for a large value of } p$$

$$\text{Also } S \leq T_{\text{serial}} / ((1-f) * T_{\text{serial}})$$

$$\text{Therefore } S \leq 10 \text{ where } T_{\text{serial}} = 20 \text{ and } f = 0.9$$

*Speedup is decided by the sequential/unparallelized version*

# Gustafson-Barsis Law

Time is constant and the problem size increases with the number of processors.

Note:  $T_{\text{serial}} = T_{\text{sequential}} + T_{\text{parallelizable}}$

$$T_{\text{parallel}} \geq T_{\text{sequential}} + T_{\text{parallelizable}} / p$$

$$\text{Speedup } S \leq (T_{\text{sequential}} + T_{\text{parallelizable}}) / (T_{\text{sequential}} + T_{\text{parallelizable}}/p)$$

Let  $r = T_{\text{sequential}} / (T_{\text{sequential}} + T_{\text{parallelizable}}/p)$  and  
 $1-r = (T_{\text{parallelizable}}/p) / (T_{\text{sequential}} + T_{\text{parallelizable}}/p)$

Hence  $T_{\text{sequential}} = r * (T_{\text{sequential}} + T_{\text{parallelizable}}/p)$  and  
 $T_{\text{parallelizable}} = (T_{\text{sequential}} + T_{\text{parallelizable}}/p) * p * (1-r)$

# Gustafson-Barsis Law

Time is constant and the problem size increases with the number of processors.

We replace the values of  $T_{\text{sequential}}$  and  $T_{\text{parallelizable}}$  in the following formula

$$\text{Speedup } S \leq (T_{\text{sequential}} + T_{\text{parallelizable}}) / (T_{\text{sequential}} + T_{\text{parallelizable}}/p)$$

$$\Rightarrow S \leq (r * (T_{\text{sequential}} + T_{\text{parallelizable}}/p) + (T_{\text{sequential}} + T_{\text{parallelizable}}/p) * p * (1-r)) / (T_{\text{sequential}} + T_{\text{parallelizable}}/p)$$

$$\Rightarrow S \leq ((T_{\text{sequential}} + T_{\text{parallelizable}}/p) * (r + (1-r)*p) / (T_{\text{sequential}} + T_{\text{parallelizable}}/p)$$

$$\Rightarrow S \leq r + (1-r)*p \Rightarrow S \leq p + (1-r)*p$$

# Gustafson-Barsis Law

Given a parallel program solving a problem using  $p$  processors,  
let  $r$  be the fraction of total execution time spent in sequential code.

The maximum speedup achievable in this program is  $S \leq p + (1-p)*r$

# Limitations of Amdahl's and Gustafson-Barsis Law

Overhead in Parallelism is not considered.

Parallelism incurs overhead due to communication, mutual exclusion, locks etc

$$T_{\text{parallel}} = (T_{\text{serial}} / p) + T_{\text{overhead}}$$

$$S = T_{\text{serial}} / ((T_{\text{serial}} / p) + T_{\text{overhead}})$$

$$E = S/p = T_{\text{serial}} / (p * ((T_{\text{serial}} / p) + T_{\text{overhead}}))$$

# References

- Chapter 2.6

An Introduction to Parallel Programming

by Peter Pacheco.

- Chapter 7

Parallel programming in C with MPI and OpenMP

by Michael J. Quinn.

# Writing Parallel Program

# Recap

- Speedup
- Efficiency
- Amdahl's Law
- Gustafson-Barsis Law

*Question: Is speedup < 1 possible?*

# Process

An instance of a computer program that is being executed.

- The executable program.
- A block of memory for
  - executable code,
  - *call stack* of active functions,
  - *memory heap* for dynamic memory allocations
- Additional details
  - resources e.g. file descriptors
  - process state
- :

# Multiprocessing

(1)

A multiprocessor system may execute multiple processes in parallel on different processors

(2)

Multiple processes may run on single processor in interleaved fashion.

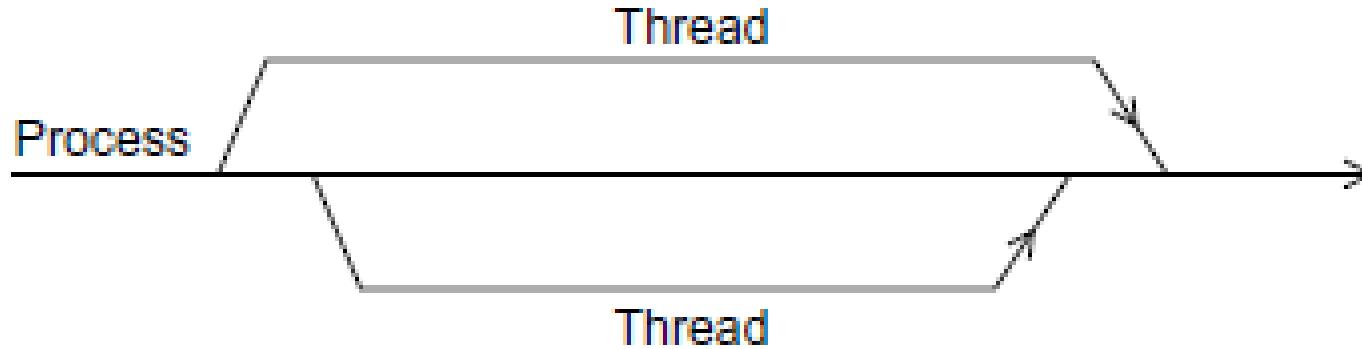
Operating system assigns multiple processes to an available processor

Each process get a time slice to execute on the processor

# Thread

- Contained in a process
- A process may contain multiple threads
- Lightweight than process
- Threads in a program share same executable
- Each thread has its own program counter
- Each thread has its own call stack
- Thread fork and thread join happen inside a process

# Thread

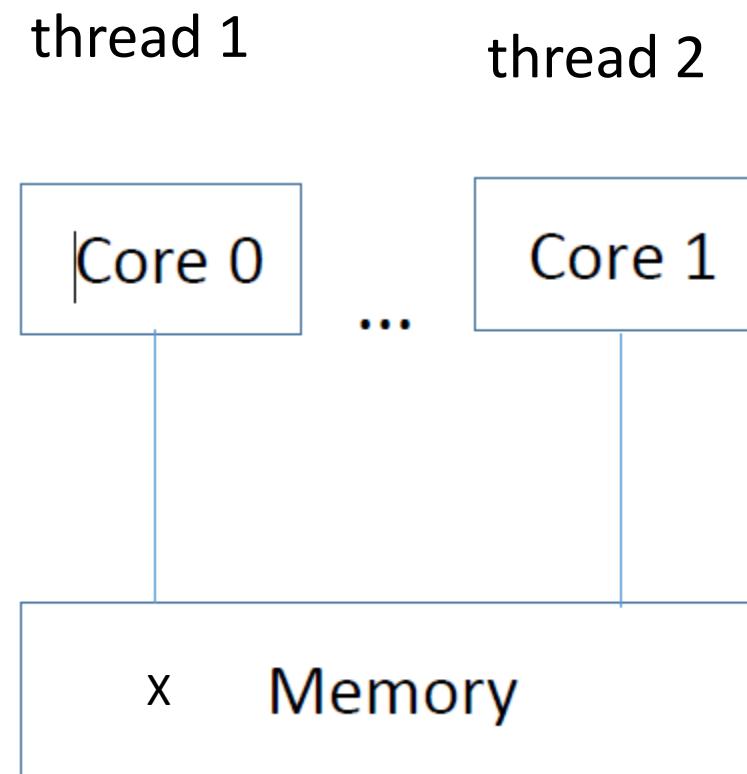


Threads may run in parallel on different processors

Multiple threads may execute on a single processor in time-slices

Threads have local memory and multiple threads share memory  
(unlike processes)

# Assigning Threads to Cores



# Writing a Parallel Program

## Sequential Program

```
for(i=0;i<N;i++) c[i] = a[i] + b[i];
```

The iterations are independent and may execute in parallel

Each thread may execute one or multiple iterations

# Writing a Parallel Program

## Sequential Program

```
for(i=0;i<N;i++) c[i] = a[i] + b[i];
```

The iterations are independent and may execute in parallel.

## Parallel Program

```
parallel for(i=0;i<N;i++) c[i] = a[i] + b[i];
```

*May executes the iterations in parallel*

# Demonstration

- Use OpenMP `parallel for`
- Execution time of the serial and parallel program
- Effects of different parameters on execution time
- Study speedup and efficiency

# Writing Parallel Program-2

# Recap

- Process
- Thread
- Parallel program

# Writing a Parallel Program

Sequential Program

```
int t=0;  
for(i=0;i<N;i++) {  
    t= a[i] + b[i];  
    c[i] = t;  
}
```

Parallel program

```
#pragma omp parallel for  
for(i=0;i<N;i++) {  
    t= a[i] + b[i];  
    c[i] = t;  
}
```

# Private and Shared

**Shared variable:** single instance is shared among all threads

**Private variable:** each thread has its own local copy

## Example

```
int x = 5;
```

```
#pragma omp parallel {  
    int a = x+1;  
}
```

# Private and Shared

**Shared variable:** single instance is shared among all threads

**Private variable:** each thread has its own local copy

## Example

```
int x = 5;
```

```
#pragma omp parallel {  
    int a = x+1;  
}
```

## Example

```
int x = 5;
```

```
#pragma omp parallel private(x){  
    int x = x+1; // bad programming  
}
```

# Rules to Specify Private and Shared Variables

Loop iteration variable is private

```
int a[N]; int i=0;  
:  
#pragma omp parallel for  
for(i=0;i<N;i++) a[i] = i;
```

Better programming practice

```
int a[N];  
:  
#pragma omp parallel for  
for(int i=0;i<N;i++) a[i] = i;
```

# Rules to Specify Private and Shared Variables

Explicit specification of shared and private variables

```
int n; int a; int b;  
:  
#pragma omp parallel for shared(n, a) private(b)  
for (int i = 0; i < n; i++) {  
    int t = b;  
    // b = a + i;  
}
```

Values of a private variable is undefined at the entry and exit of a parallel region.

# Static Schedule

```
int nthreads=10
#pragma omp parallel for shared(a,b,c) private(i) schedule(static)
parallel for(i=0;i<N;i++) c[i] = a[i] + b[i];
```

Distribute the chunk of iterations to threads

thread 1 : iteration 0...(N/10)-1

thread 2: iteration (N/10)...2\*(N/10)-1

:

thread 10: iteration 9\*(N/10)...N-1

# Another Static Schedule

```
#pragma omp parallel for shared(a,b,c) private(i)
schedule(static,size of the chunk=4)
parallel for(i=0;i<64;i++) c[i] = a[i] + b[i];
```

Distribute the chunk of iterations to threads

thread 1 : iteration {0,1,2,3}, {16,17,18,19}, ...

thread 2: iteration {4,5,6,7}, {20,21,22,23}, ...

thread 3 : iteration {8,9,10,11}, {24,25,26,27}, ...

thread 4: iteration {12,13,14,15}, {28,29,30,31}, ...

# Dynamic Schedule

schedule(dynamic, n): Default value of n=1

Each thread

1. executes a chunk of n iterations
2. requests another chunk

No particular order of chunk assignments to threads

# Static vs Dynamic Schedule

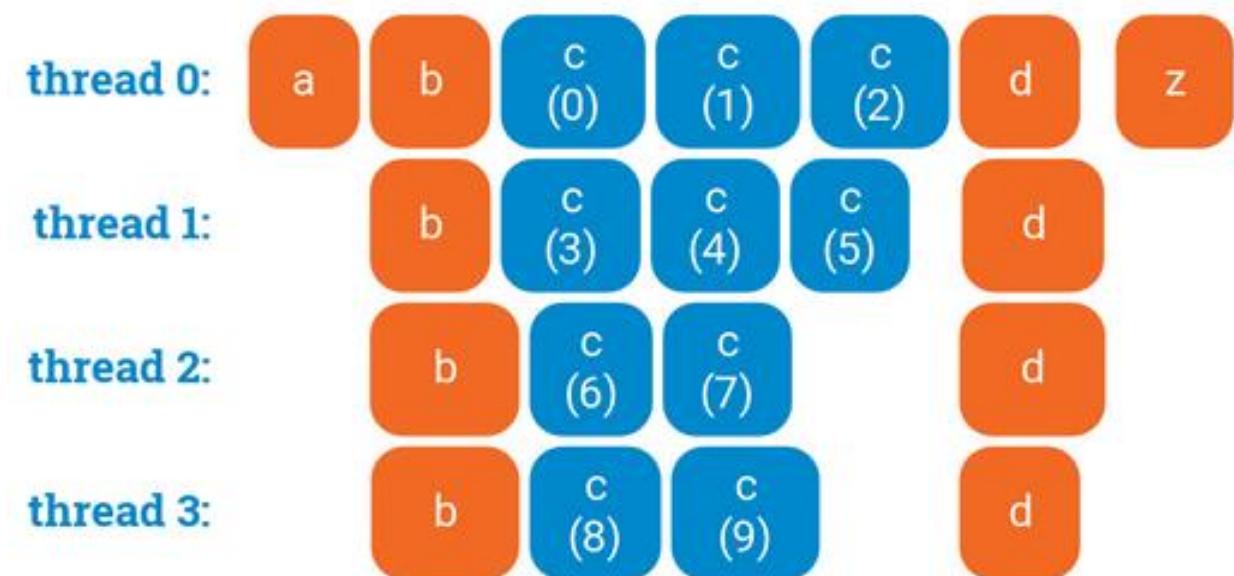
1. Dynamic scheduling is preferred when the iterations are of different computational size
2. Dynamic scheduling incurs runtime overhead unlike static scheduling as distribution is performed during execution

# Waiting in `parallel for`

No synchronization at the beginning of a parallel for loop.

Threads synchronize at the end of a parallel for loop.

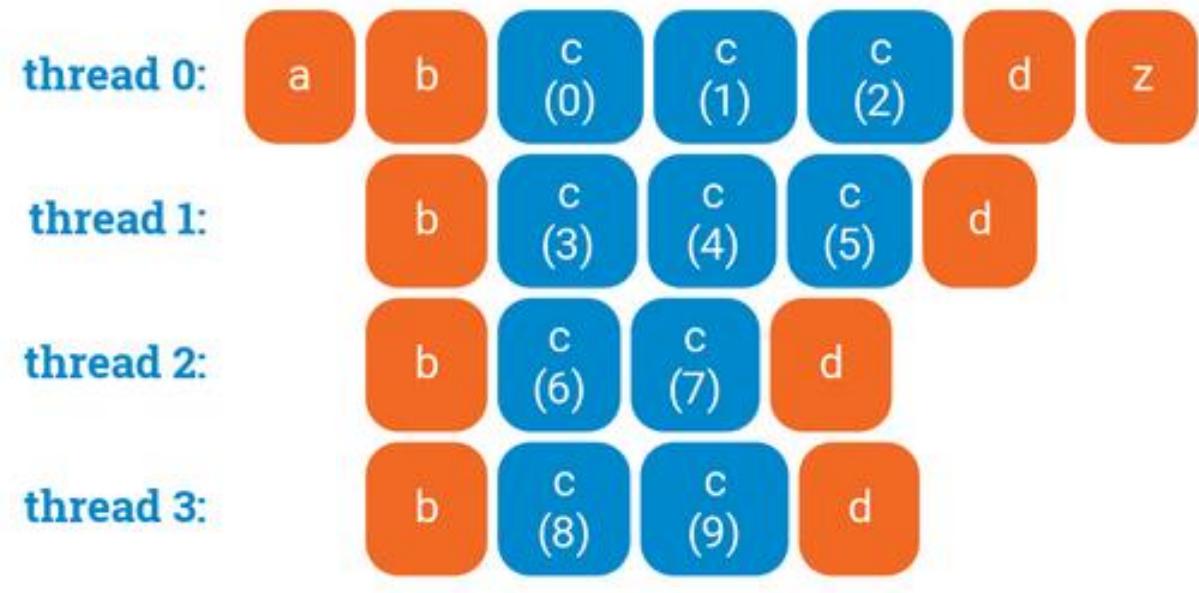
```
a();  
#pragma omp parallel  
{  
    b();  
    #pragma omp for  
    for (int i = 0; i < 10; ++i) {  
        c(i);  
    }  
    d();  
}  
z();
```



# Waiting in `parallel for`

`nowait` removes the synchronization

```
a();
#pragma omp parallel
{
    b();
    #pragma omp for nowait
    for (int i = 0; i < 10; ++i) {
        c(i);
    }
    d();
}
z();
```



# Writing Parallel Program-3

# Recap

- Shared and Private
- Schedule: static, dynamic
- nowait

# Recap: Shared and Private

```
int n; int a; int b;  
:  
#pragma omp parallel for shared(n, a) private(b)  
for (int i = 0; i < n; i++) {  
    int t = b;  
    // b = a + i;  
}
```

# Recap: Schedule

```
// number of threads = 10
#pragma omp parallel for shared(a,b,c) private(i) schedule(static)
parallel for(i=0;i<N;i++)
    c[i] = a[i] + b[i];
```

Distribute the chunk of iterations to threads

thread 1 : iteration 0...(N/10)-1

thread 2: iteration (N/10)...2\*(N/10)-1

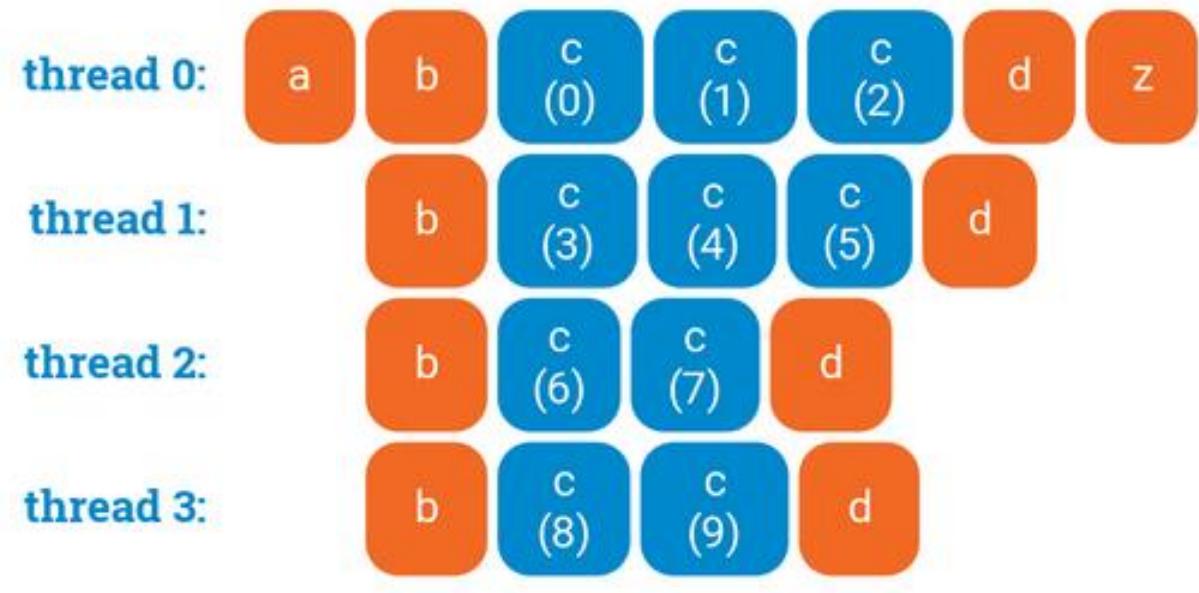
:

thread 10: iteration 9\*(N/10)...N-1

# Waiting in `parallel for`

`nowait` removes the synchronization

```
a();
#pragma omp parallel
{
    b();
    #pragma omp for nowait
    for (int i = 0; i < 10; ++i) {
        c(i);
    }
    d();
}
z();
```



# Q & A Discussion

piazza

Scope of private variable

Nested parallel constructs

# Distribution of Iterations

```
// number of threads = 10
#pragma omp parallel for shared(a,b,c) private(i) schedule(static)
parallel for(i=0;i<N;i++)
    c[i] = a[i] + b[i];
```

Distribute the chunk of iterations to threads

thread 1 : for(int i1=0;i1<N/10;i1++) c[i1] = a[i1] + b[i1];

thread 2: for(int i2=N/10; ;i2<(2\*N)/10; i2++) c[i2] = a[i2] + b[i2];

:

thread 10: for(int i10=(9\*N)/10; i10<N; i10++) c[i10] = a[i10] + b[i10];

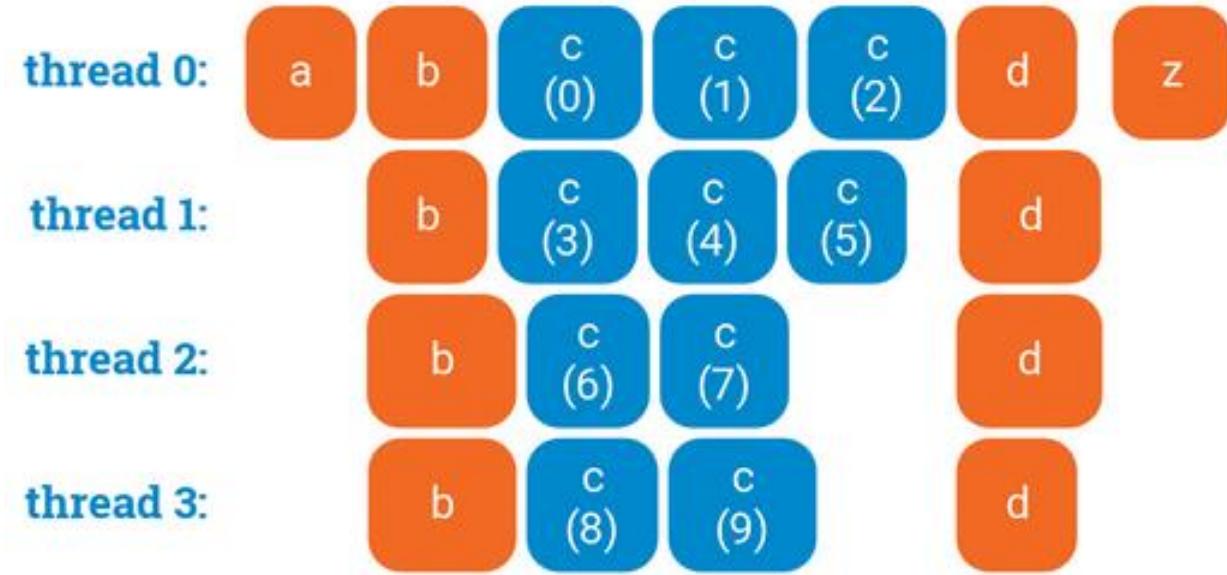
# Distribution of Iterations

```
// number of threads = 10
#pragma omp parallel for shared(a,b,c) private(i) schedule(static, N/10)
for(i=0;i<N;i++)
    c[i] = a[i] + b[i];
```

***What if  $N < 10$  ?***

# Nested Parallelism

```
a();  
#pragma omp parallel  
{  
    b();  
    #pragma omp for  
    for (int i = 0; i < 10; ++i) {  
        c(i);  
    }  
    d();  
}  
z();
```



Can be controlled by `omp_set_nested()`  
`omp_set_nested(0)` // FALSE: no nested parallelization  
`omp_set_nested(1)` // TRUE: nested parallelization

# Synchronization

Orders the completion of code executed by different threads

Several constructs

- barrier

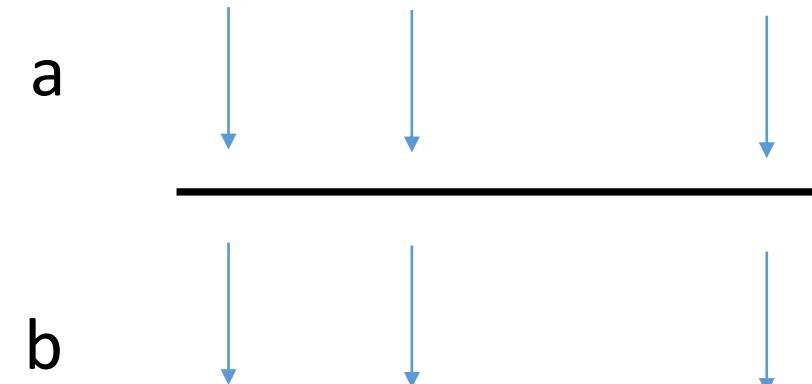
- critical

- :

# Barrier

A program point where all active threads will stop until all threads have arrived at that point.

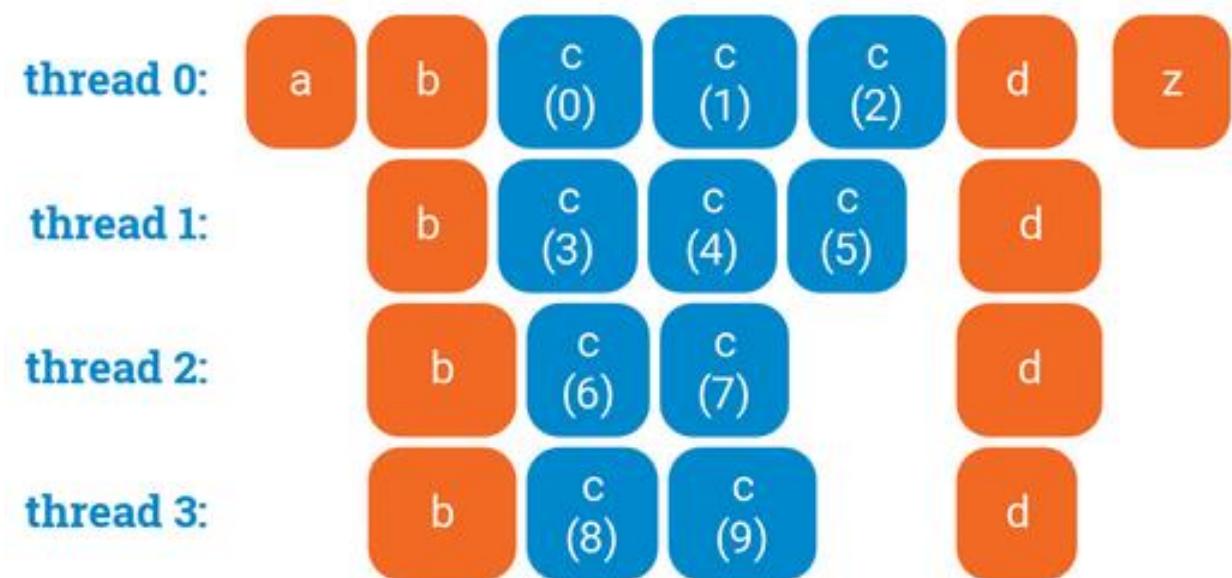
```
#pragma omp parallel {  
    int i = omp_get_thread_num();  
    a[i] = i;  
    #pragma omp barrier  
    b[i] = i;  
}
```



# Implicit Barrier in `parallel for`

Threads synchronize at the end of a parallel for loop.

```
a();  
#pragma omp parallel  
{  
    b();  
    #pragma omp for  
    for (int i = 0; i < 10; ++i) {  
        c(i);  
    }  
    d();  
}  
z();
```

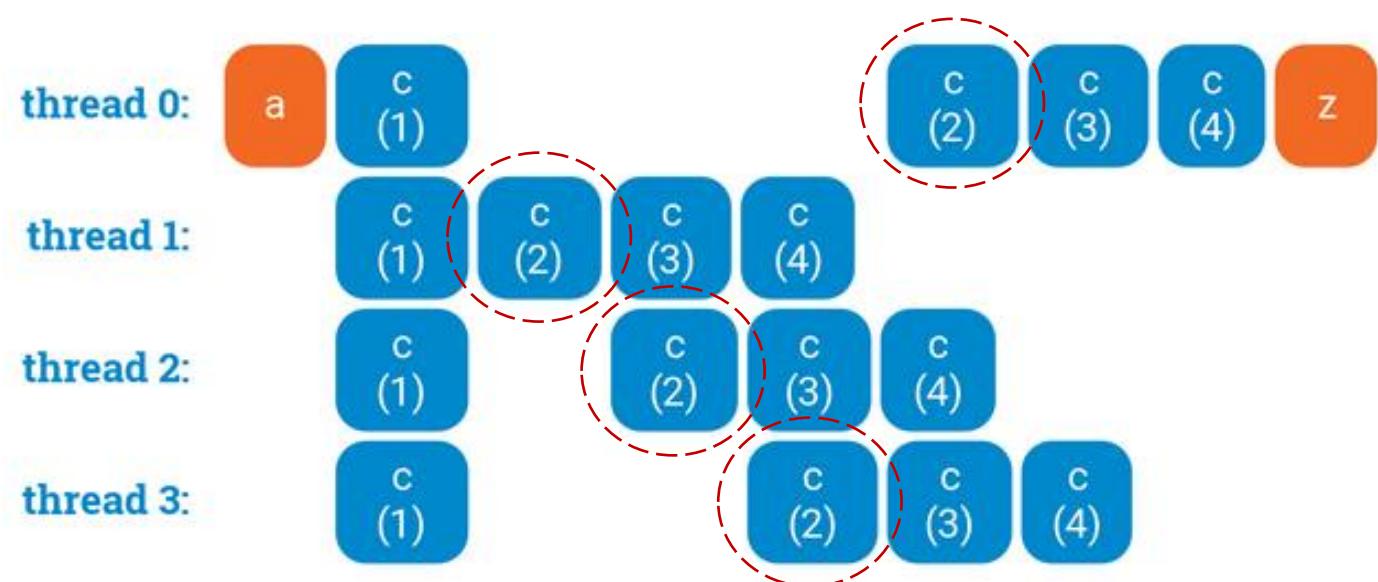


# Critical Section

Only one thread execute a critical section at a time.

Provides mutual exclusion.

```
a();  
#pragma omp parallel  
{  
    c(1);  
    #pragma omp critical  
    {  
        c(2);  
    }  
    c(3);  
    c(4);  
}  
z();
```



# Data Race

Multiple parallel threads access one shared memory location where at least one of these accesses is a write.

The read or written value is undefined

## Example

```
#pragma omp parallel for
for(int i=0;i<N;i++) {
    sum = sum + f(a[i]);
}
```

*Can we fix it?*

# Data Race

Multiple parallel threads access one shared memory location where at least one of these accesses is a write.

The read or written value is undefined

## Example

```
#pragma omp parallel for
for(int i=0;i<N;i++) {
    #pragma omp critical
    {
        sum = sum + f(a[i]);
    }
}
```

# Caveats

- No automatic parallelization; parallelization is programmer specified
- Programmer is responsible to ensure correctness of the parallelized program
- ‘parallel for’ does not work with other loop constructs e.g. while.
- When should we **NOT** parallelize a loop?

- When should we **NOT** parallelize a loop?

Answer: If there are data dependencies across iterations

**Example:**

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

The definition of 'sum' in iteration  $i$  is used in iteration  $(i+1)$

# References

- Chapter 5

An Introduction to Parallel Programming  
by Peter Pacheco.

- Chapter 17

Parallel programming in C with MPI and OpenMP  
by Michael J. Quinn.

- OpenMP Specification.

<https://www.openmp.org/spec-html/5.0/openmp.html>

# Writing Parallel Program-4

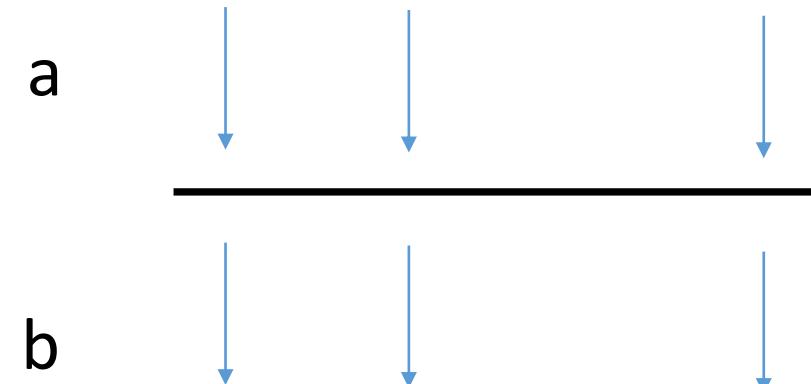
# Recap

- Synchronization (barrier, critical)
- Data race

# Recap: Barrier

A program point where all active threads will stop until all threads have arrived at that point.

```
#pragma omp parallel {  
    int i = omp_get_thread_num();  
    a[i] = i;  
    #pragma omp barrier  
    b[i] = i;  
}
```

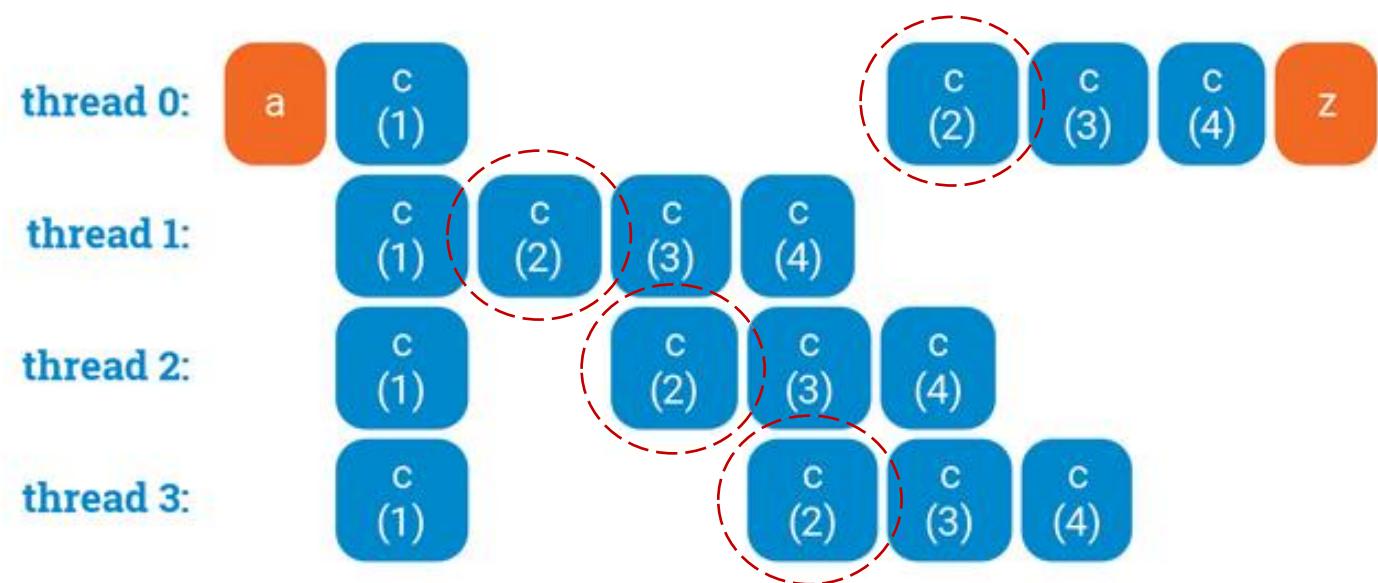


# Recap: Critical Section

Only one thread execute a critical section at a time.

Provides mutual exclusion.

```
a();  
#pragma omp parallel  
{  
    c(1);  
    #pragma omp critical  
    {  
        c(2);  
    }  
    c(3);  
    c(4);  
}  
z();
```



# Recap: Data Race

Multiple parallel threads access one shared memory location where at least one of these accesses is a write.

The read or written value is undefined

## Example

```
#pragma omp parallel for
for(int i=0;i<N;i++) {
    sum = sum + f(a[i]);
}
```

# Reduction

Reducing an expression to a value.

$$\text{sum} = a[0] + a[1] + \dots + a[N-1]$$

*Can we can reduce the following loop?*

```
for(int i=0;i<N;i++) {  
    sum = sum + a[i];  
}
```

**Issues:**

- Data race

# Reduction

Reducing an expression to a value.

$\text{sum} = \text{a}[0] + \text{a}[1] + \dots + \text{a}[\text{N}-1]$

*Can we can reduce the following loop?*

```
for(int i=0;i<N;i++) {  
    sum = sum + a[i];  
}
```

**Issues:**

- Data race
- dependencies across iterations

# Parallel For Reduction

*Can we can reduce the following loop?*

```
#pragma omp parallel for shared(sum, a) reduction(+: sum)
for(int i=0;i<N;i++) {
    sum = sum + a[i];
}
```

Supported reduction operators: +, -, \*, &, |, ^, &&, ||

# Handling Parallel For Reduction

```
#pragma omp parallel for shared(sum, a) reduction(+: sum)
for(int i=0;i<N;i++) {
    sum = sum + a[i];
}
```

- Reduction forks a number of threads
- Iterations are assigned to threads
- Partial sum values are computed.
- Final value is computed by accumulating the partial sum values

- When should we **NOT** parallelize a loop?

Answer: If there are data dependencies across iterations

**Example:**

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

The definition of 'sum' in iteration  $i$  is used in iteration  $(i+1)$

# Data Dependence: general Definition

There is a *data dependence* from statement S1 to statement S2 (statement S2 *depends on* statement S1) if and only if

- 1) both statements access the same memory location and at least one of them is a write and
- 2) there is a feasible run-time execution path from S1 to S2.

# Data Dependency

**True dependence:** written data is read later

$X = a; \dots; b = X;$  //  $X$  is not redefined between S1 and S2

**Anti-dependence:** read data is written later

$b = X; \dots; X = a;$

**Output dependence:** two statements write on the same location

$X=a; \dots; X=b;$

# Data Dependency Across Iterations in a Loop

## Example 1:

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

## Example 2:

```
for(int i=0;i<N;i++)  
    a[0] = i;
```

## Example 3:

```
for(int i=0;i<N;i++){  
    t = a[i];  
    b[i] = t;  
    a[i] = b[i];
```

```
}
```

1.       $t = a[i];$
2.       $b[i] = t;$
3.       $a[i] = b[i];$
4.       $t = a[i+1];$
5.       $b[i+1] = t;$
6.       $a[i+1] = b[i];$

# Data Dependency Across Iterations in a Loop

**Read-Write:** Data is written in one iteration and read in another iteration

**Example:**

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

**Write-Write:** two iterations writing on the same location

**Example:**

```
for(int i=0;i<N;i++)  
    a[0] = i;
```

# Loop index based dependencies

## Example 1:

```
for(int i=0;i<N;i++)  
    a[i+1] = a[i];
```

$a[0] \rightarrow a[1] \rightarrow a[2] \rightarrow \dots \rightarrow a[N-1]$

# Loop index based dependencies

## Example 1:

```
for(int i=0;i<N;i++)  
    a[i+1] = a[i];
```

$a[0] \rightarrow a[1] \rightarrow a[2] \rightarrow \dots \rightarrow a[N-1]$

## Example 2:

```
for(int i=0;i<N;i++)  
    a[i+2] = a[i];
```

$a[0] \rightarrow a[2] \rightarrow a[4] \rightarrow \dots$

$a[1] \rightarrow a[3] \rightarrow a[5] \rightarrow \dots$

# Loop index based dependencies

## Example 1:

```
for(int i=0;i<N;i++)  
    a[i+1] = a[i];
```

$a[0] \rightarrow a[1] \rightarrow a[2] \rightarrow \dots \rightarrow a[N-1]$

## Example 2:

```
for(int i=0;i<N;i++)  
    a[i+2] = a[i];
```

$a[0] \rightarrow a[2] \rightarrow a[4] \rightarrow \dots$   
 $a[1] \rightarrow a[3] \rightarrow a[5] \rightarrow \dots$

## Example 3:

```
for(int i=0;i<N;i+=2)  
    a[i+1] = a[i];
```

$a[0] \rightarrow a[1]$   
 $a[2] \rightarrow a[3]$   
 $a[4] \rightarrow a[5]$

# Data Dependency Across Iterations in a Loop

## Example 1:

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

## Example 2:

```
for(int i=0;i<N;i++)  
    a[0] = i;
```

## Example 3:

```
for(int i=0;i<N;i++){  
    int t = a[i];  
    b[i] = t;  
    a[i] = b[i];  
}
```

*Can we parallelize these loops?*

# References

- OpenMP Specification.

<https://www.openmp.org/spec-html/5.0/openmp.html>

- Chapter 2.2.1, 2.2.2

Optimizing compilers for modern architectures

a dependence-based approach

by Randy Allen, Ken Kennedy

# Dependency

# Recap

- Reduction
- Data dependencies
- Data dependencies across loop iterations

```
S1: a=X;  
    if(a==1)  
S2:  b=Y;
```

# Other Dependencies

- Control dependency

```
S1: a=X;  
if(a==1)  
S2: b=Y;
```

- Address dependency

```
S1: a=X;  
S2: b=Y[a];
```

# Transformation Correctness

## Program Equivalence

Given same input, if two computations produce identical values for output variables on the same inputs then the computations are equivalent.

- (1) program P is transformed to P'
- (2) Program P and P' are equivalent.

The transformation is correct.

correct transformation is semantics preserving

# Correctness of Dependence-Based Transformations

Reordering transformations: S1;S2; --> S2;S1;

A *reordering transformation* changes the order of execution of the code, without adding or deleting any executions of any statements.

## **Reordering transformation and dependencies**

A reordering transformation *preserves* a dependence if it preserves the relative execution order of the source and sink of that dependence.

# Correct of Reordering Transformation

Any reordering transformation that preserves every dependence in a program preserves the meaning of that program.

# Correct of Reordering Transformation

Any reordering transformation that preserves every dependence in a program preserves the meaning of that program.

Is the following correct?

$a=X; Y=a*0; \rightarrow Y=a*0; a=X; \rightarrow Y=0; a=X;$

# True and False Dependence

**True dependence:** dependence cannot be removed

$a=X; Y=a;$

**False dependence:** dependence can be removed

$a=X; Y=a*0;$

*Data dependence analysis is undecidable in general*

*We will not remove false dependence (for now at least)*

# Subtle Questions

Given same input, if two computations produce identical values for output variables on the same inputs then the computations are equivalent.

How to reason about

- Side effect
- Crash/failure
- Non-termination

*We will not cover these issues in this course.*

*If (Interested) goto <https://compcert.org/motivations.html>*

# Dependencies across Multi-Dimensional Accesses

Identifying dependencies across iterations is a challenge

Identifying dependencies are not immediate

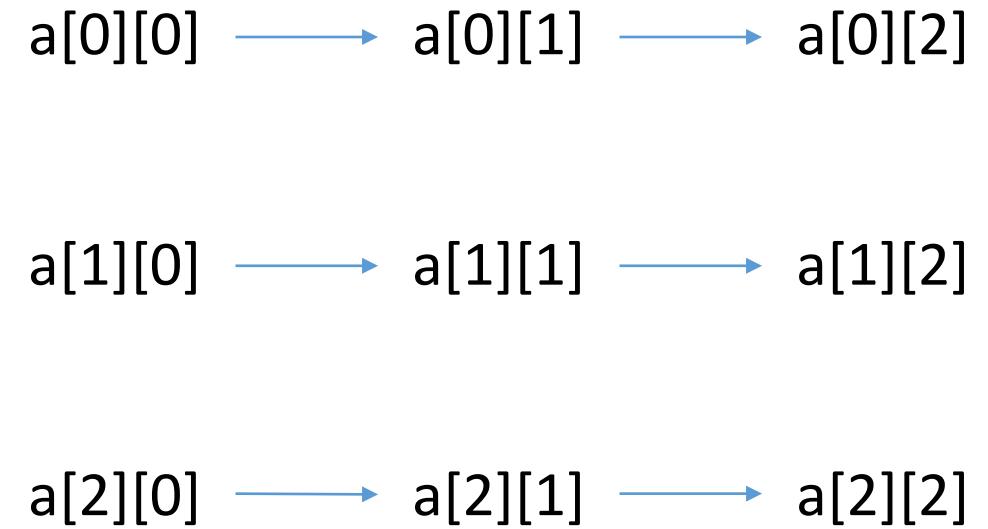
```
for(int p=0;p<N;p++){
    for(int q=0;q<M;q++){
        S1: a[p,q+1] = a[p,q];
    }
}
```

# Dependencies across Multi-Dimensional Accesses

Identifying dependencies across iterations is a challenge

Identifying dependencies are not immediate

```
for(int p=0;p<N;p++){  
    for(int q=0;q<M;q++){  
        S1: a[p,q+1] = a[p,q];  
    }  
}
```



# Distance Vector

If there is a dependence from  $S_1$  on iteration  $i$  to  $S_2$  on iteration  $j$ ;  
then the *dependence distance vector*  $\mathbf{d}(i, j)$  is defined as  $\mathbf{d}(i, j) = j - i$

```
for(int p=0;i<N;p++){
    for(int q=0;q<M;q++){
        S1: a[p,q+1] = a[p,q];
    }
}
```

$$a[0][0] \longrightarrow a[0][1] \longrightarrow a[0][2]$$

$$a[1][0] \longrightarrow a[1][1] \longrightarrow a[1][2]$$

$$a[2][0] \longrightarrow a[2][1] \longrightarrow a[2][2]$$

$$i = (0,0) \text{ and } j = (0,1): \mathbf{d}(i,j) = j - i = (0,1)$$

$$i = (1,0) \text{ and } j = (1,1): \mathbf{d}(i,j) = (0,1)$$

.....

# Direction Vector

If there is a dependence from  $S_1$  on iteration  $i$  and  $S_2$  on iteration  $j$ ; then the *dependence direction vector*  $D(i, j)$  is defined as

$$D(i, j)_k = \begin{cases} "<" & \text{if } d(i, j)_k > 0 \\ "=" & \text{if } d(i, j)_k = 0 \\ ">" & \text{if } d(i, j)_k < 0 \end{cases}$$

$i = (0,0)$  and  $j = (0,1)$ :  $d(i,j) = (0,1) \Rightarrow D(i,j) = (=,<)$

$i = (1,0)$  and  $j = (1,1)$ :  $d(i,j) = (0,1) \Rightarrow D(i,j) = (=,<)$

.....

# Another Example

```
for(int i = 0; i<N; i++)
    for(int j = 0; j<M; j++)
        for(int k = 0; k<L; k++)
            A[i+1][j+1][k] = A[i][j][k] + A[i][j+1][k+1]
        }
    }
}
```

Direction matrices: ( $<$ ,  $<$ ,  $=$ ), ( $<$ ,  $=$ ,  $>$ )

# References

- Chapter 2.2.3, 2.2.4

Optimizing compilers for modern architectures

a dependence-based approach

by Randy Allen, Ken Kennedy

# Dependency Analysis

# Recap

- Program Equivalence
- Data dependencies across loop iterations

# Data Dependencies in Nested Loop

## Iteration Number

loop index: I

Lower and Upper Bound: L and U

Step size: S

$$\text{iteration number } i = (I - L + 1) / S$$

### Example:

L=1; U=10; S=2;

```
for(int I=L; I<=U; I+=S) {
```

...

}

Suppose, L= 1, U = 10, I = 7, S =2

$$i = (7 - 1 + 1)/2 = 4$$

# Data Dependencies in Nested Loop

## Iteration Vector

Consider a nest of  $n$  loops labelled by  $1 \leq k \leq n$  from the outer to the inner loop.

The *iteration vector*  $i = \{i_1, i_2, \dots, i_n\}$  of a particular iteration of the  $n$ -th loop is a vector of integers that contains the iteration numbers for each of the loops in order of nesting level.

### Example:

```
for(int i=1;i<=2;i++) {      // 1
    for (int j=1;j<=2;j++) { // 2
        S;
    }
}
```

$S(2,1)$  is the instance of statement  $S$  in the  
- 2<sup>nd</sup> iteration of the **1**-loop and  
- 1<sup>st</sup> iteration of the **2**-loop

# Data Dependencies in Nested Loop

## Iteration Space

All possible iteration vectors

**Example:**

```
for(int i=1;i<=2;i++) {           // 1           {(1,1), (1,2), (2,1), (2,2)}  
    for (int j=1;j<=2;j++) { // 2  
        S;  
    }  
}
```

# Data Dependencies in Nested Loop

$i_k$  is the  $k$ th element of the vector  $i$

$$i = (2,1)$$

$i[1:k]$  is a  $k$ -vector consisting of the leftmost  $k$  elements of  $i$

$$i_1 = 2, i_2 = 1$$

Iteration  $i$  precedes iteration  $j$ , denoted  $i < j$ , if and only if

$$i[1:2] = (2,1)$$

Given  $i = (2,1)$  and  $j = (2,2)$

1)  $i[1:n-1] < j[1:n-1]$  or

$$\begin{aligned} i &< j \\ (2,1) &< (2,2) \end{aligned}$$

2)  $i[1:n-1] = j[1:n-1]$  and  $i_n < j_n$

# Dependence in Loop Nest

There exists a dependence from statement S1 to statement S2 in a common nest of loops if and only if there exist two iteration vectors  $i$  and  $j$  for the nest, such that

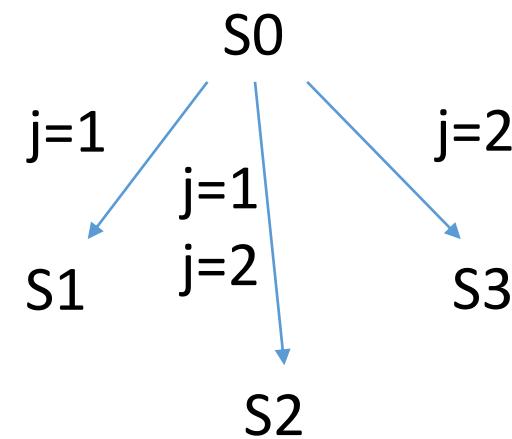
- (1)  $i < j$  or  $i = j$  and there is a path from S1 to S2 in the body of the loop,
- (2) statement S1 accesses memory location  $M$  on iteration  $i$  and statement S2 accesses location  $M$  on iteration  $j$ , and
- (3) one of these accesses is a write.

# Dependence in Loop Nest

There exists a dependence from statement S1 to statement S2 in a common nest of loops if and only if there exist two iteration vectors  $i$  and  $j$  for the nest, such that

- (1)  $i < j$  or  $i = j$  and there is a path from S1 to S2 in the body of the loop,
- (2) statement S1 accesses memory location  $M$  on iteration  $i$  and statement S2 accesses location  $M$  on iteration  $j$ , and
- (3) one of these accesses is a write.

```
for(int i = 1, i<=N; i++){           // 1
    for(int j=1; j<=2; j++){        // 2
        S0: A[i][j] = A[i][j] + B
    }
    S1: T = A(i,1);
    S2: A(i,1) = A(i,2);
    S3: A(i,2) = T;
}
```



# Distance Vector

If there is a dependence from  $S_1$  on iteration  $i$  to  $S_2$  on iteration  $j$ ;  
then the *dependence distance vector*  $d(i, j)$  is defined as  $d(i, j) = j - i$

```
for(int i=1;i<=N;i++){                                j = (2,2,2)  reads a[2][2][2]
    for(int j=1;j<=M;j++){
        for(int k=1; k<=L; k++){
            S1: a[i+1,j,k-1] = a[i,j,k]+1;          i = (1,2,3)  writes a[2][2][2]
        }
    }
}
```

# Direction Vector

If there is a dependence from  $S_1$  on iteration  $i$  and  $S_2$  on iteration  $j$ ;  
then the *dependence direction vector*  $D(i, j)$  is defined as

$$D(i, j)_k = \begin{cases} "<" & \text{if } d(i, j)_k > 0 \\ "=" & \text{if } d(i, j)_k = 0 \\ ">" & \text{if } d(i, j)_k < 0 \end{cases}$$

```
for(int i=1;i<=N;i++){                                j = (2,2,2)  reads a[2][2][2]
    for(int j=1;j<=M;j++){
        for(int k=1; k<=L; k++){
            S1: a[i+1,j,k-1] = a[i,j,k]+1;          i = (1,2,3)  writes a[2][2][2]
        }
    }
}
```

$$d(i,j) = j - i = (1,0,-1)$$
$$D(i,j) = (<, =, >)$$

# Direction Vector and Dependencies

A dependence cannot exist if it has a direction vector such that the leftmost non-“=” component is not “<”

```
for(int i=1;i<=N;i++){                                j = (2,2,2)  reads a[2][2][2]
    for(int j=1;j<=M;j++){
        for(int k=1; k<=L; k++){
            S1: a[i+1,j,k-1] = a[i,j,k]+1;          i = (1,2,3)  writes a[2][2][2]
        }
    }
}
```

$d(i,j) = j - i = (1,0,-1)$

$D(i,j) = (<, =, >)$

There is a data dependence

# Loop-carried Dependencies

There is loop-carried dependence takes place across iterations in a loop.

Statement S2 has a *loop-carried dependence* on statement S1 if and only if

- S1 references location  $M$  on iteration  $i$ , S2 references  $M$  on iteration  $j$  and
- $d(i,j) > 0$  (that is,  $D(i,j)$  contains a “<” as leftmost non “=” component).

The *level* of a loop-carried dependence is the index of the leftmost non-“=” of  $D(i,j)$  for the dependence.

```
for(int i = 1; i<10; i++)
    for(int j = 1; j<10; j++)
        for(int k = 1; k<10; k++)
            S1: A[i][j][k+1] = A[i][j][k];
```

$D(i,j) = (=,=,<)$  for all dependencies

Dependencies level = 3

# Loop Independent Dependencies

Statement S2 has a *loop-independent dependence* on statement S1 if and only if there exist two iteration vectors  $i$  and  $j$  such that:

- (1) Statement S1 refers to memory location  $M$  on iteration  $i$ , S2 refers to  $M$  on iteration  $j$ , and  $i = j$ .
- (2) There is a control flow path from S1 to S2 within the iteration.

```
for(int i=0;i<N;i++){  
    S1: t = a[i];  
    S2: a[i] = b[i];  
    S3: b[i] = t;  
}
```

# General Condition for Loop Dependency

Let  $\alpha$  and  $\beta$  be iteration vectors within the iteration space of the following loop nest

```
for(int i1 = L1; i1<=U1; i1+=S1)
    for(int i2 = L2; i2<=U2; i2+=S2)
        ...
        for(int in = Ln; in<=Un; in+=Sn)
            S1: A(f1(i1,...,in),...,fm(i1,...,in)) = ...
            S2 ... = A(g1(i1,...,in),...,gm(i1,...,in))
        }
        ...
    }
}
```

A dependence exists from S1 to S2 if and only if there exist values of  $\alpha$  and  $\beta$  such that  
(1)  $\alpha$  is lexicographically less than or equal to  $\beta$  and  
(2) the following system of *dependence equations* is satisfied:

$$f_i(\alpha) = g_i(\beta) \text{ for all } i, 1 \leq i \leq m$$

# References

- Chapter 2

Optimizing compilers for modern architectures  
a dependence-based approach

by Randy Allen, Ken Kennedy

# Dependency Testing

# Recap

- Iteration vector
- Distance vector
- Direction vector
- Loop carried dependencies
- Loop independent dependencies

# General Condition for Loop Dependency

Let  $\alpha$  and  $\beta$  be iteration vectors within the iteration space of the following loop nest

```
for(int i1 = L1; i1<=U1; i1+=S1)
    for(int i2 = L2; i2<=U2; i2+=S2)
        ...
        for(int in = Ln; in<=Un; in+=Sn)
            S1: A[f1(i1,...,in)]...[fm(i1,...,in)] = ...
            S2 ... = A[g1(i1,...,in)]...[gm(i1,...,in)]
        }
        ...
    }
}
```

A dependence exists from S1 to S2 if and only if there exist values of  $\alpha$  and  $\beta$  such that  
(1)  $\alpha$  is lexicographically less than or equal to  $\beta$  and  
(2) the following system of *dependence equations* is satisfied:

$$f_i(\alpha) = g_i(\beta) \text{ for all } i, 1 \leq i \leq m$$

# Problem: Dependence Testing

Goal: prove that no dependence exists between given pairs of subscripted references to the same array variable.

- \* no solution for the equation:  $fi(\alpha) = gi(\beta)$  for all  $i$ ,  $1 \leq i \leq m$

array subscripts are linear expressions of the loop index variables.

That is, all subscript expressions are of the form:

$$a_1 * i_1 + a_2 * i_2 + \dots + a_n * i_n + e$$

where  $i_k$  is the index for the loop at nesting level  $k$ ; all  $a_k$ ,  $1 \leq k \leq n$ , integer constants.  
 $e$ : loop-invariant (symbolic) expressions.

Linear subscript: dependency testing  $\Leftrightarrow$  finding the solution of a linear equation

- \* NP-complete.

# Dependency Testing

- There are several dependence based testing techniques
- The techniques address particular types of dependency patterns
- Lamport, **GCD**, Banerjee, I-test, power test, omega test, **delta** test

# GCD Test

```
for(int i1 = L1; i1<=U1;i++)  
    for(int i2 = L2; i2<=U2;i2++)  
        ...  
        for(int in = Ln; in<=Un;in++)  
            S1 A[f(i1,...,in)] = ...;  
            S2 ... = A[g(i1,...,in)];
```

Assume

$$f(x_1, x_2, \dots, x_n) = a_0 + a_1x_1 + \dots + a_nx_n$$
$$g(y_1, y_2, \dots, y_n) = b_0 + b_1y_1 + \dots + b_ny_n$$

$$a_0 + a_1x_1 + \dots + a_nx_n = b_0 + b_1y_1 + \dots + b_ny_n$$

$$a_0 - b_0 + a_1x_1 - b_1y_1 + \dots + a_nx_n - b_ny_n = 0$$

$$a_1x_1 - b_1y_1 + \dots + a_nx_n - b_ny_n = b_0 - a_0$$

**GCD Test.** The equation has a solution if and only if  $\gcd(a_1, \dots, a_n, b_1, \dots, b_n)$  divides  $b_0 - a_0$ .

# Example of GCD Test

if a loop carried dependency exists between  $X[a*i + b]$  and  $X[c*i + d]$   
then  $\text{GCD}(c, a)$  must divide  $(d - b)$ .

```
for (int i = 1;i <= n;i++)
    S1: a[2*i] = b[i] + c[i];
    S2: d[i] = a[2*i-1];
```

Are there  $i_1$  and  $i_2$  such that  $1 \leq i_1 < i_2 \leq n$  and  $2*i_1 = 2*i_2 - 1$  ?

equivalently  $2*i_2 + (-2)*i_1 = 1$

There is an integer solution if and only if  $\text{gcd}(2, -2)$  divides 1

This is not the case, so no dependence.

# False Positive of GCD Test

```
for (int i = L;i <= L+10; i++)
```

```
    S1: a[i] = b[i] + c[i];
```

```
    S2: d[i] = a[i-100];
```

Are there  $i_1$  and  $i_2$  such that  $L \leq i_1 < i_2 \leq L+10$  and  $i_1 = i_2 - 100$  ?

equivalently  $i_2 - i_1 = 100$

There is an integer solution if and only if  $\gcd(1, -1)$  divides 100

Answer: true, so there is a dependence

But there is no actual dependence: false positive!

# Limitations of GCD Test

Ignores loop bounds

Does not provide distance or direction information

If GCD is 1, then the analysis is very conservative

# Delta Test

Source iteration of dependency:  $i$

Sink iteration of dependency:  $i + d$

```
for(int i = 1;i<=N;i++)  
    A[i + 1] = A[i] + B
```

Valid dependency implies  $i+1 = i+d$ .

It implies  $d = 1$ .

Loop-carried dependence with distance vector (1) and direction vector (<)

# Another Example

```
for(int i = 1; i<=100; i++)  
    for(int j=1; j<=100;j++)  
        for(int k = 1, k<=100;k++)  
            A[i+1][j][k] = A[i][j][k+1] + B;
```

$$I+1 = I+ di; \quad J= J+ dj; \quad K= K+ dk + 1$$

- Solutions:  $di = 1; dj = 0; dk = -1$
- Corresponding direction vector:  $(<,=,>)$

# Dependency and Parallelism

It is valid to convert a sequential loop to a parallel loop if the loop carries no dependence.

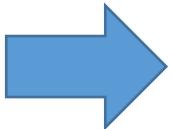
Example:

Loop independent dependence

- Scalar expansion
- Privatization

# Scalar Expansion

```
int a[N], b[N], t;  
for(int i=0;i<N;i++){  
    t = a[i];  
    a[i] = b[i];  
    b[i] = t;  
}
```



```
int a[N], b[N], t[N];  
for(int i=0;i<N;i++){  
    t[i] = a[i];  
    a[i] = b[i];  
    b[i] = t[i];  
}
```

Handles loop independent dependency

**Performance tradeoff:** Temporary variable is converted to an array that result in memory accesses

# Restrictions

- The number of iterations of the loop must be countable; the loop step size must be changed in the loop body.
- The expanded scalar must have no upward exposed uses in the loop

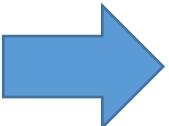
```
for(int i=0;i<N;i++){  
    print(t);  
    t = a[i]; a[i] = b[i]; b[i] = t;  
}
```

- When the scalar is live after the loop, we must move the correct array value into the scalar.
  - a variable is live if it is used later

```
for(int i=0;i<N;i++){  
    t = a[i]; a[i] = b[i]; b[i] = t;  
}  
print(t); // a[N-1]
```

# Privatization

```
for(int i=0;i<N;i++){  
    t = a[i];  
    a[i] = b[i];  
    b[i] = t;  
}
```



```
#pragma omp parallel for ...  
for(int i=0;i<N;i++){  
    int t = a[i];  
    a[i] = b[i];  
    b[i] = t;  
}
```

Create local copies of the variable t

# Restrictions

- The expanded scalar must have no upward exposed uses in the loop

```
for(int i=0;i<N;i++){  
    print(t);  
    t = a[i]; a[i] = b[i]; b[i] = t;  
}
```

- When the scalar is live after the loop, we must move the correct array value into the scalar.

```
for(int i=0;i<N;i++){  
    t = a[i]; a[i] = b[i]; b[i] = t;  
}  
print(t);
```

# Summary: Dependency and Parallelism

It is valid to convert a sequential loop to a parallel loop if the loop carries no dependence.

Automatic parallelization is possible to some extent.

Compilers perform automatic parallelization in certain cases.

Additional transformations may be required to get parallelizable loops.

# References

- Chapter 3 (relevant parts for dependency testing techniques)
- Chapter 6.2 and 6.3 (privatization and scalar expansion)

Optimizing compilers for modern architectures  
a dependence-based approach

by Randy Allen, Ken Kennedy

# Parallelism in Hardware

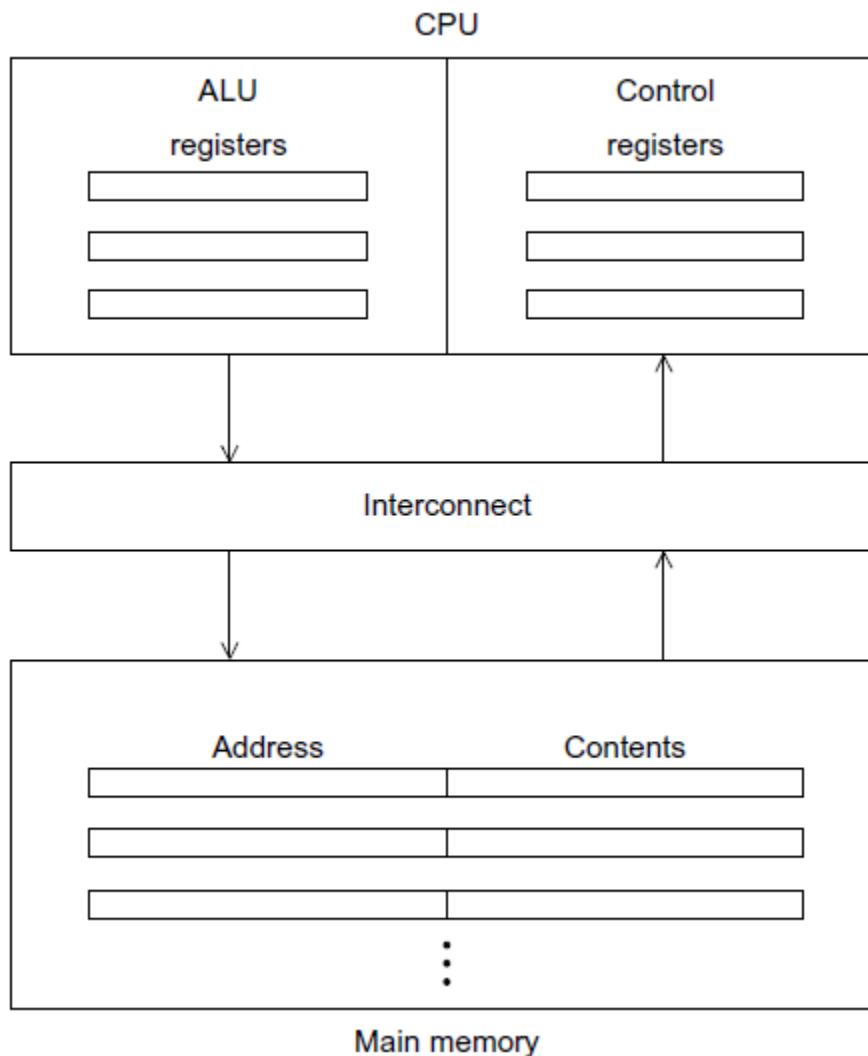
# So Far...

Parallel programming using OpenMP constructs

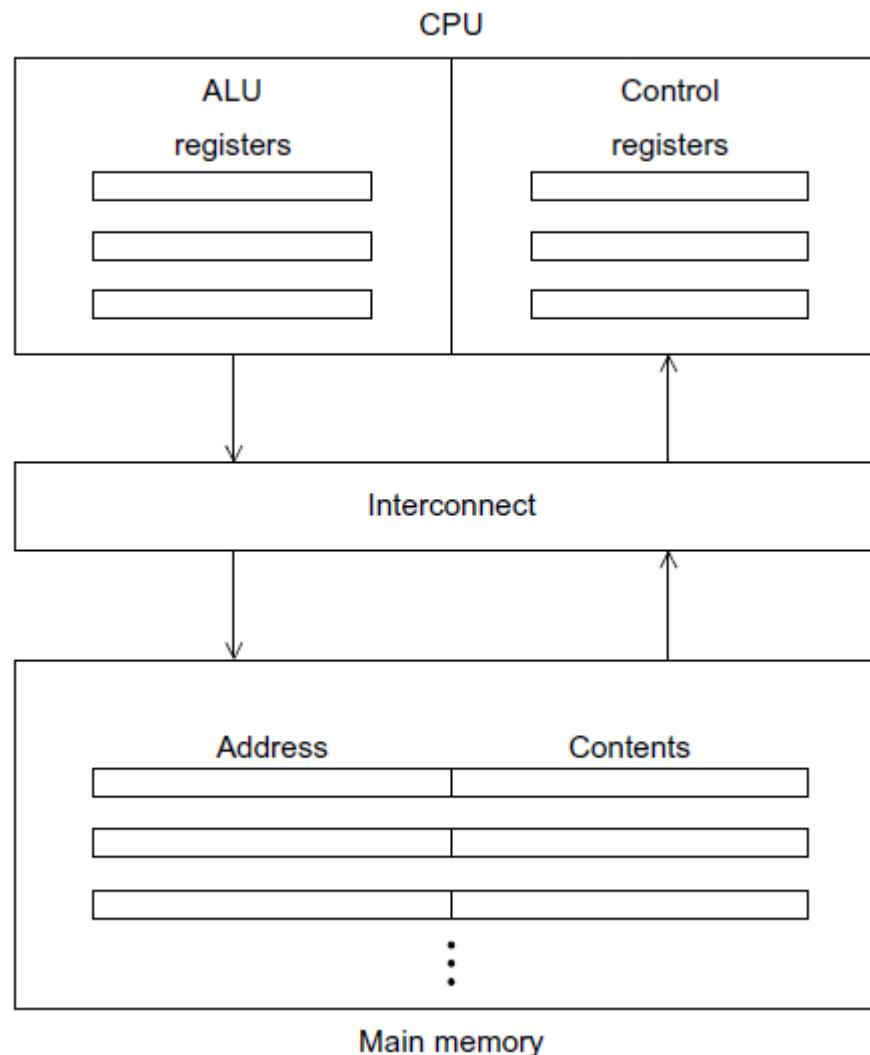
Dependence analyses to identify parallelism

**Question:** How does hardware support parallelism?

# von Neumann architecture



# Improvements on von Neumann architecture



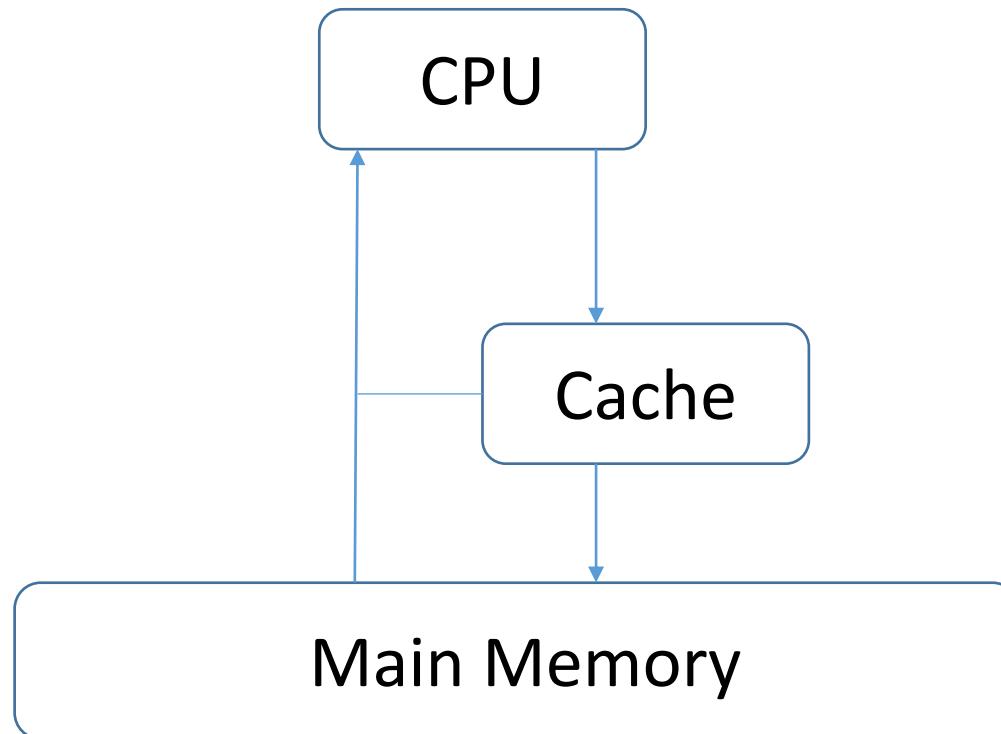
## Major Improvements:

- Cache memory
- Virtual memory
- Low level parallelism

# Cache Memory

An intermediate small, faster memory between CPU and main memory.

CPU can access cache memory significantly faster than main memory.



# Principle of Locality

Cache memory improve temporal and spatial locality.

**Temporal locality.** After accessing one memory location (for instruction or data) computation will access the same location in near future.

**Spatial locality.** After accessing one memory location (for instruction or data) computation will consecutively access a nearby location.

**Example:**

```
int a[1000];
...
sum = 0;
for (i = 0; i < 1000; i++)
    sum += a[i];
```

# Accessing Cache Memory

## Cache Hit and Miss

When a cache is checked for information and the information is available, it's a cache hit. Otherwise, it's a cache miss.

## Cache Block and Cache Lines

Based on principles of locality a memory access will effectively operate on a blocks of memory instead of an individual location.

# Write-through and Write-back Cache

## Write-through cache

writes the data back to memory as soon as it is written to the cache.

## Write-back cache

The data isn't written immediately.

- (1) Mark the updated data in the cache as *dirty*.
- (2) When the cache line is replaced by a new cache line from memory, the dirty line is written to memory.

# Cache mappings

Decided by cache associativity

**Fully associative cache:** a new line can be placed at any location in the cache.

**Direct-mapped cache.** each cache line has a unique location in the cache to which it will be assigned.

**$n$ -way set associative (Intermediate schemes).** each cache line can be placed in one of  $n$  different locations in the cache.

# Example

Table 2.1 Assignments of a 16-line Main Memory to a 4-line Cache

Memory Index	Cache Location		
	Fully Assoc	Direct Mapped	2-way
0	0, 1, 2, or 3	0	0 or 1
1	0, 1, 2, or 3	1	2 or 3
2	0, 1, 2, or 3	2	0 or 1
3	0, 1, 2, or 3	3	2 or 3
4	0, 1, 2, or 3	0	0 or 1
5	0, 1, 2, or 3	1	2 or 3
6	0, 1, 2, or 3	2	0 or 1
7	0, 1, 2, or 3	3	2 or 3
8	0, 1, 2, or 3	0	0 or 1
9	0, 1, 2, or 3	1	2 or 3
10	0, 1, 2, or 3	2	0 or 1
11	0, 1, 2, or 3	3	2 or 3
12	0, 1, 2, or 3	0	0 or 1
13	0, 1, 2, or 3	1	2 or 3
14	0, 1, 2, or 3	2	0 or 1
15	0, 1, 2, or 3	3	2 or 3

Cache associativity Tradeoff:

**Direct mapped cache:**  
good best-case time but  
unpredictable in worst case

**Fully associative cache:**  
the best miss rates, but practical only  
for a small number of entries

Different cache replacement policies  
for n-way associative caches

# Effect of Cache Memory on Programs

```
double A[MAX][MAX], x[MAX], y[MAX];
...
/* Initialize A and x, assign y = 0 */
...
/* First pair of loops */
for (i = 0; i < MAX; i++)
    for (j = 0; j < MAX; j++)
        y[i] += A[i][j] * x[j];
...
/* Assign y = 0 */
/* Second pair of loops */
for (j = 0; j < MAX; j++)
    for (i = 0; i < MAX; i++)
        y[i] += A[i][j] * x[j];
```

C stores two-dimensional arrays in “row-major” order.

MAX=4

Cache Line	Elements of A			
0	A[0][0]	A[0][1]	A[0][2]	A[0][3]
1	A[1][0]	A[1][1]	A[1][2]	A[1][3]
2	A[2][0]	A[2][1]	A[2][2]	A[2][3]
3	A[3][0]	A[3][1]	A[3][2]	A[3][3]

First pair of loops: 4 cache misses

Second pair of loops: 16 cache misses

Result: the first pair of loops are faster

# Virtual Memory

Main memory may not suffice to store:

- a large program,
- a program with large data size,
- multiple programs and data in multitasking OS

Solution: Virtual memory.

- main memory functions as a cache for secondary storage
- main memory access is order of thousands times faster than secondary memory
- virtual memory abstracts the physical addresses of the secondary storage

# Page

- virtual memory operates on pages (blocks of data and instructions)
- pages are of a fixed page size (4 to 16 KB)
- pages contain virtual addresses, not physical addresses of secondary storage
- Page table translates virtual addresses to physical addresses

virtual address = virtual page number + byte offset in the page

Let a virtual address is of 32 bit and page size is 4 KB = 4096 bytes.

Virtual Address											
Virtual Page Number						Byte Offset					
31	30	...	13	12	11	10	...	1	0		
1	0	...	1	1	0	0	...	1	1		

Each byte in the page is identified with 12 bits, since  $2^{12} = 4096$

# Virtual Memory and TLB

**Virtual memory tradeoff:**

Virtual memory to physical memory computation increases the runtime

**Solution:** translation-lookaside buffer or TLB (similar to cache memory)

TLB caches a few entries from the page table in very fast memory.

Principle of spatial and temporal locality is applicable

Improves the page table access speed significantly

**Page fault:** the page is not in main memory, it is in secondary storage. Similar to cache miss.

# References

## Chapter 2

- An Introduction to Parallel Programming  
by Peter Pacheco.

# Parallelism in Hardware

# Recap

- Cache memory
- Virtual memory

# Instruction-level parallelism

multiple functional units executing instructions simultaneously.

**Pipelining** arranges functional units are arranged in stages

**Multiple issue** initiates multiple instructions at the same time

```
float x[1000], y[1000], z[1000];
...
for (i = 0; i < 1000; i++)
    z[i] = x[i] + y[i];
```

**Pipelining:** Fetch of X[1] and Y[1] and computation of  $x[0] + y[0]$  may happen in parallel by different pipeline units.

**Multiple issue:** Fetch of  $x[0], \dots, x[n]$  and  $y[0] \dots y[n]$  can be parallel

# Fine Grained vs Coarse Grained Parallelism

ILP is a fine grained parallelism

Dependencies limit the scope of ILP.

$$\begin{aligned} t &= a+b; \\ r &= X[t]; \end{aligned}$$

$$f[0] = f[1] = 1;$$

```
for (i = 2; i <= n; i++)
```

$$f[i] = f[i-1] + f[i-2];$$

Coarse grain parallelism.

- execute larger/coarser unit of computation in parallel.
- Multithreading computation

# Parallel Hardware

## SIMD machine. Single Instruction Multiple Data

Single control unit and multiple ALU.

```
for (i = 0; i < n; i++)  
    x[i] = x[i]+y[i];
```

# Vector processors

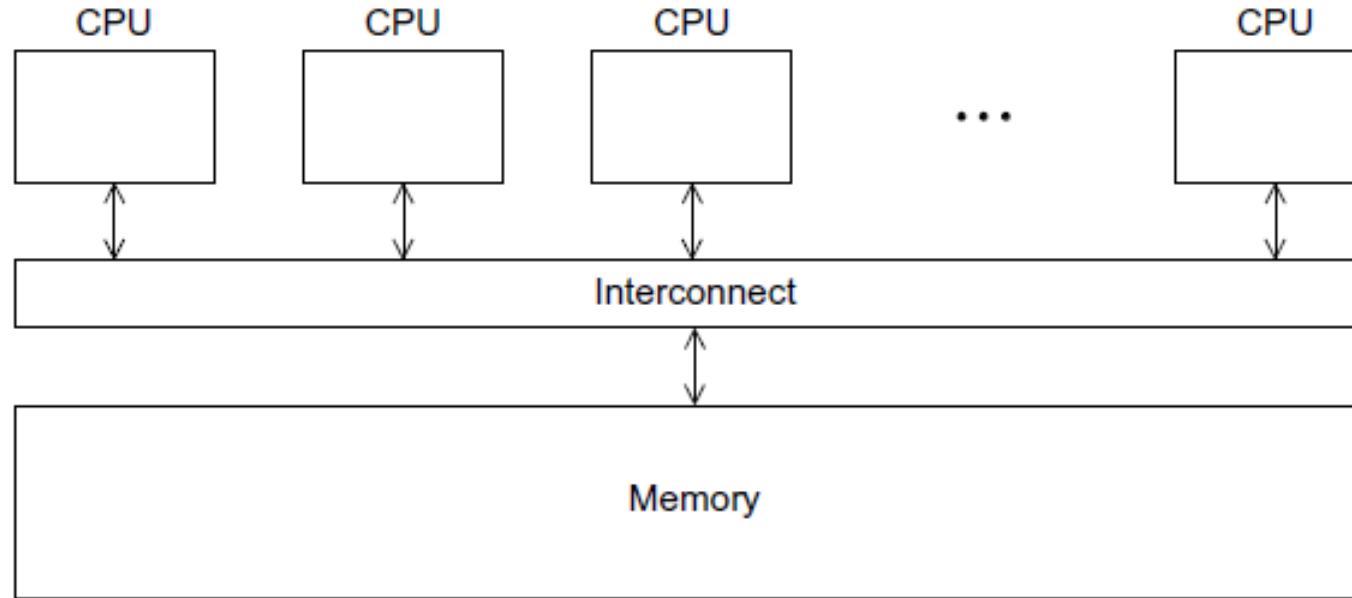
- Vector registers
- Vectorized and pipelined functional units.
- Vector instructions
- multiple “banks” of memory
- Strided memory access: memory access separated by fixed intervals
- Many x86 processor support vector computation
- Compilers generate vector instructions

```
for (i = 0; i < 3; i++)  
    a[i] = b[i] + c[i];
```



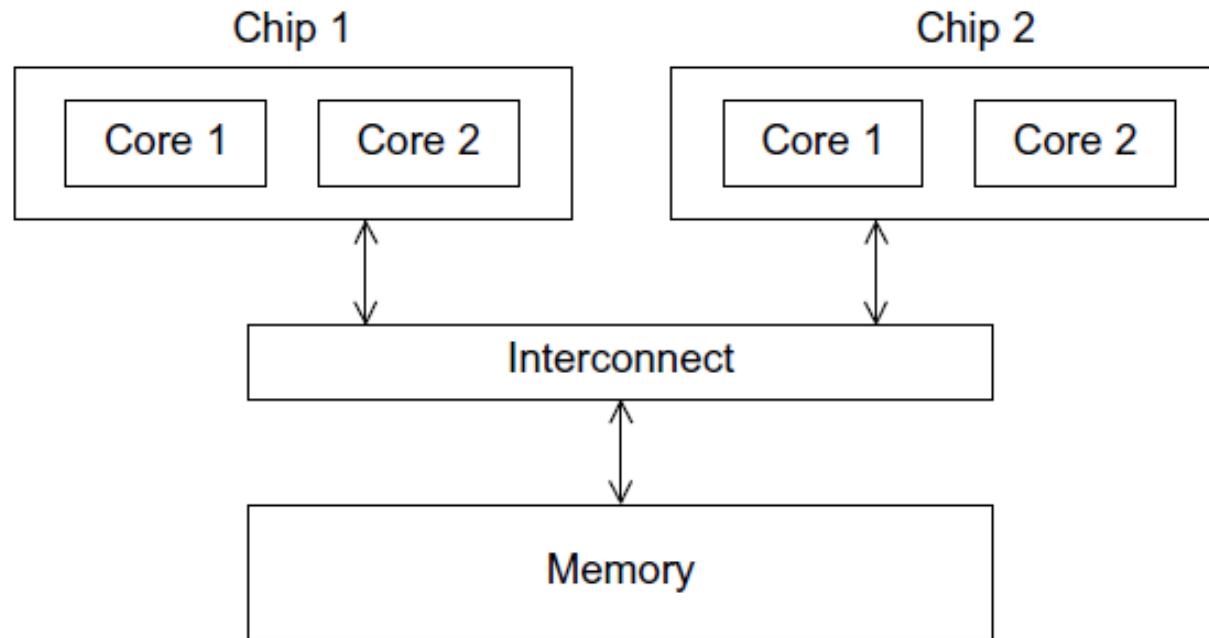
```
vr1 = b[0:3];  
vr2 = c[0:3];  
a[0:3] = vr1 + vr2;
```

# Shared Memory System



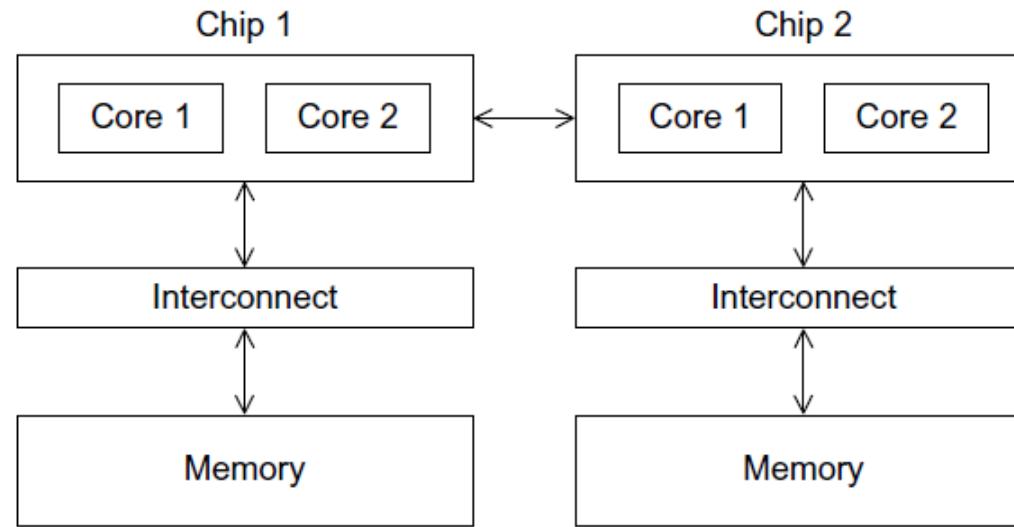
Processors communicate implicitly by accessing shared data residing on shared memory.

# Uniform Memory Access System



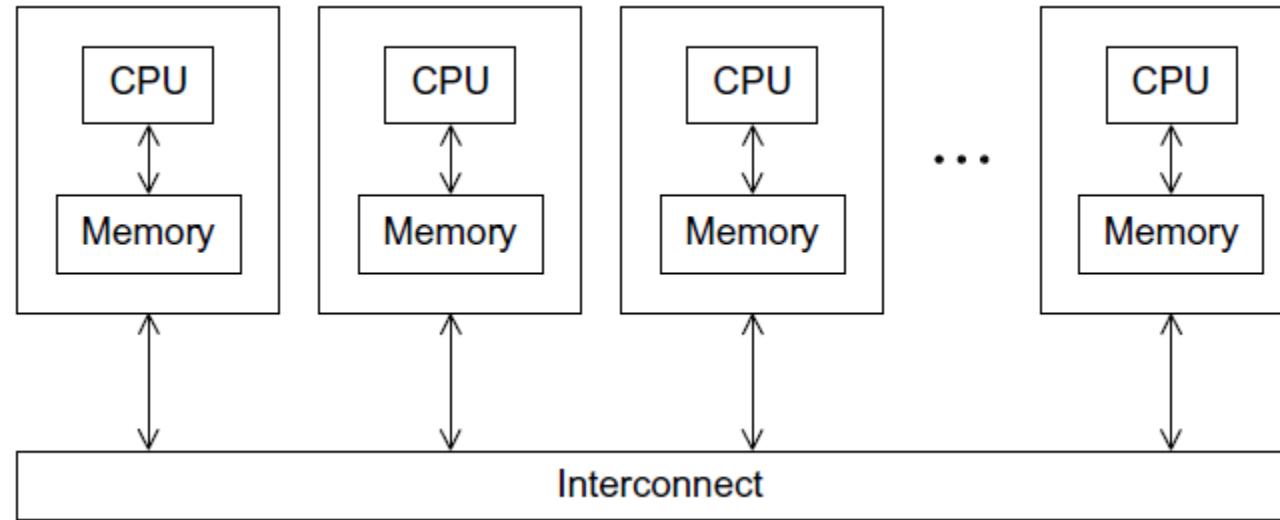
- processors are connected to main memory by interconnects
- Shared memory access time is same for all processor

# Uniform vs Non-Uniform Memory Access



- each processor can have it's block of main memory
- processors access each others' memory through special hardware
- access to other processors' memories may take longer

# Distributed Memory System



- each processor has its own *private* memory
- the processor-memory pairs communicate over an interconnection network
- Communicate by sending messages

# Latency and Bandwidth

Latency is the time that elapses between the source's beginning to transmit the data and the destination's starting to receive the first byte.

Bandwidth is the rate at which the destination receives data after it has started to receive the first byte.

Let

$l$  seconds be the latency of an interconnect and  
 $b$  bytes per second be the bandwidth

Time taken to transmit a message of  $n$  bytes is  $l + (n/b)$

# Cache Coherence

Time	Core 0	Core 1
0	$y_0 = x;$	$y_1 = 3*x;$
1	$x = 7;$	Statement(s) not involving $x$
2	Statement(s) not involving $x$	$z_1 = 4*x;$

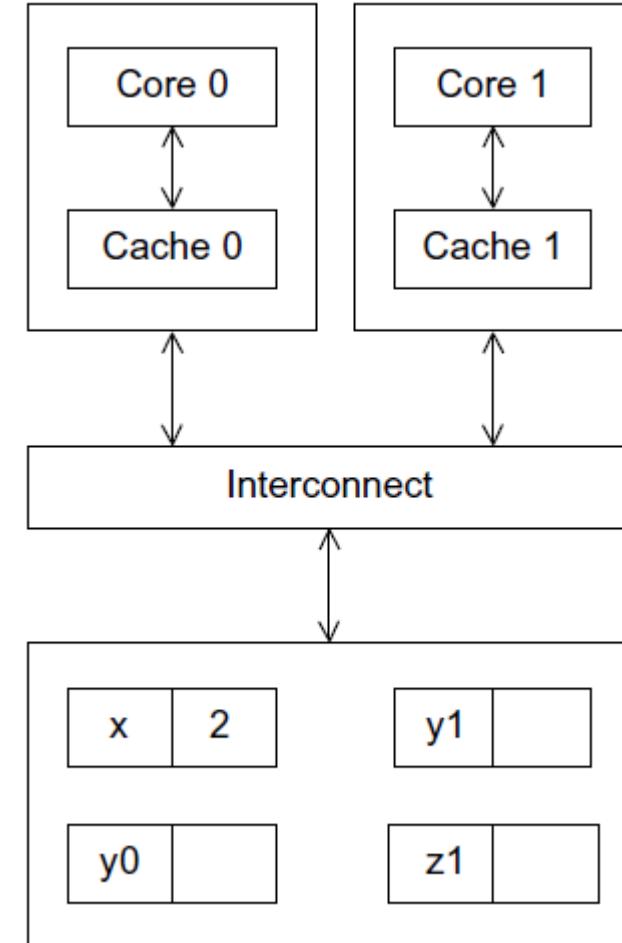
What are the possible values of  $y_1$  and  $z_1$ ?

$$y_1 = 3*2 \rightarrow z_1 = 4*2 = 8 \text{ or } z_1 = 4*7 = 28$$

$$y_1 = 7*3 \rightarrow$$

$$z_1 = 7*3 = 21 \text{ and}$$

$z_1 = 4*2 = 8$  is not possible



# Cache Coherence

x=2

Time	Core 0	Core 1
0	$y_0 = x;$	
1	$x = 7;$	$y_1 = 3*x;$ Statement(s) not involving x
2	Statement(s) not involving x	$z_1 = 4*x;$

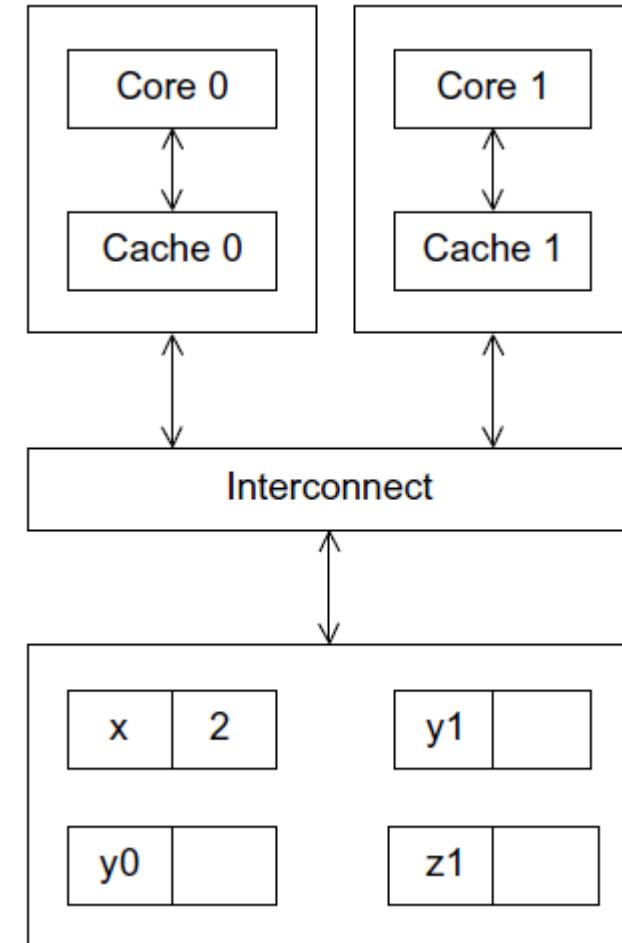
What are the possible values of  $y_1$  and  $z_1$ ?

$$y_1 = 3*2 \rightarrow z_1 = 4*2 = 8 \text{ or } z_1 = 4*7 = 28$$

$$y_1 = 7*3 \rightarrow$$

$$z_1 = 7*3 = 21 \text{ and}$$

$$z_1 = 4*2 = 8 \text{ is not possible}$$



# Cache Coherence Problem

X=1;

X=2;

r=X;

shared main memory

# Snooping Based Cache Coherence Protocol

When the cores share a bus, any signal transmitted on the bus can be observed by all the cores connected to the bus

- core 0 updates the copy of x stored in its cache,
- Core 0 broadcasts this information(cache line that contains x is updated) across the bus
- core 1 is snooping the bus,
- Sees that x (or the cache line that contains x) has been updated
- It can mark its copy of x as invalid.

snooping works with both write-through and write-back caches

# Drawback of Snooping Based Protocol

- Broadcasts are expensive
- Snooping protocol broadcasts every time a variable is updated
- Not scalable, in larger systems due to performance

# Directory-based cache coherence Protocol

The distributed directory stores the status of each cache line.

Each core/memory pair stores the status of the cache lines in its local memory.

A read operation updates the directory entry corresponding to that line with the information about the core

When a variable is updated, the directory is consulted, and the cache controllers of the cores that have that variable's cache line in their caches are invalidated.

Pros. only the cores storing that variable need to be contacted.

$X=Y; \parallel Y=Z; \parallel Z=X;$  // T1{X,Y}, T2{Y,Z}, T3{Z,X}

Cons. Substantial additional storage required for the directory

# False Sharing

CPU caches operate on cache lines, not individual variables

Update on one variable →

mark the cache line as *dirty*, other cores fetch the data from the main memory

```
int i, j, m, n;  
double y[m];  
/* Assign y = 0 */  
  
...  
for (i = 0; i < m; i++)  
    for (j = 0; j < n; j++)  
        y[i] += f(i,j);
```

core\_count=2  
m=8  
doubles are 8 bytes  
cache line 64 bytes

```
int i, j, iter_count;  
/* Shared variables initialized by one core */  
int m, n, core_count; double y[m];  
iter_count = m/core_count;  
  
/* Core 0 */  
for (i = 0; i < iter_count; i++)  
    for (j = 0; j < n; j++)  
        y[i] += f(i,j);  
  
/* Core 1 */  
for (i = iter_count+1; i < 2*iter_count; i++)  
    for (j = 0; j < n; j++)  
        y[i] += f(i,j);
```

# References

## Chapter 2

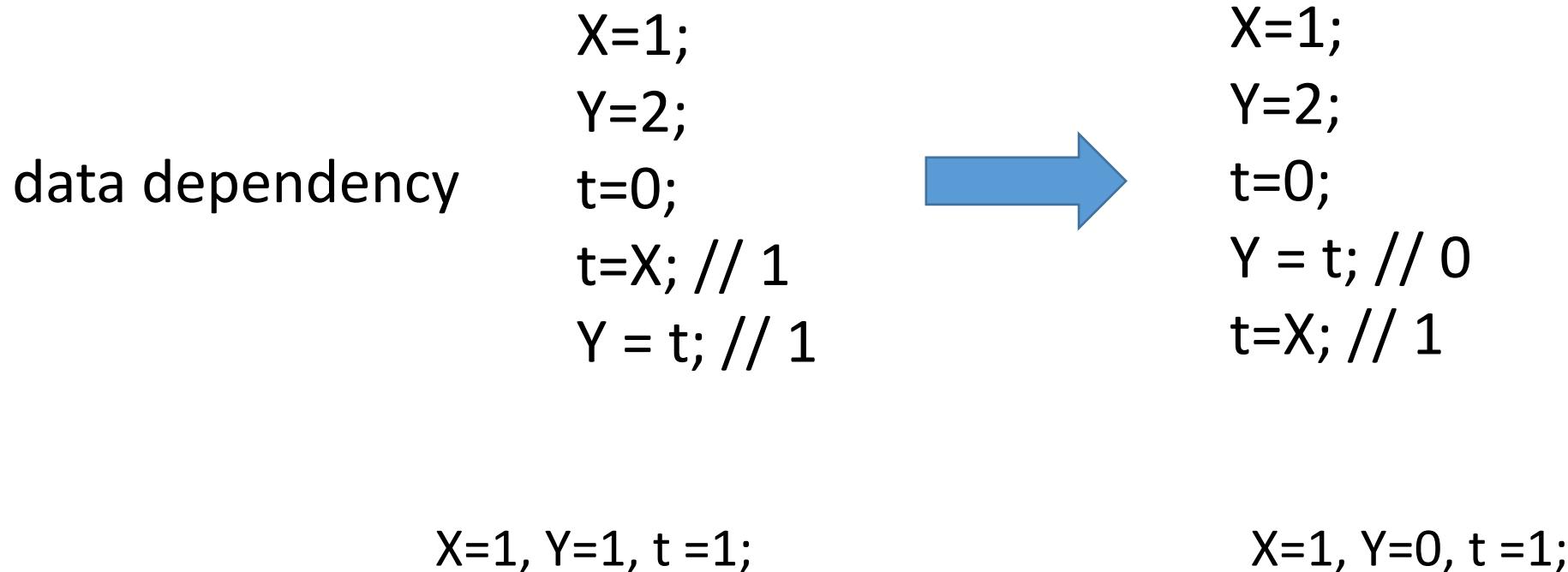
- An Introduction to Parallel Programming  
by Peter Pacheco.

# Parallelism in Hardware

# Recap

- ILP
- SIMD
- Uniform and non-uniform memory access

# Pipelining and Dependencies



# Another Example

a=X;

X=2;

anti

X=2;

a=X;

data

X=2;

X=3;

output

a=X;

b=X;

accessing same  
location

No out of order execution

# Coherence Property

Ensured by the architectures (almost all)

There is a total order on all accesses on a single location/shared variable due to cache coherence property

X=0;

X=1;

a=X;  
X=2;

What are the possible values of a?

a = 0 ? yes

a = 1 ? yes

a = 2 ? no

# Another Example

```
X=0;  
X=2;  
a=X; // 0
```

# Coherence Property

There is a total order on all accesses on a single location/shared variable due to cache coherence property

$x=0$

S1:  $x=1;$

S2:  $x=2;$

S3:  $a=x;$

What are the possible values of  $a$  in S3?

$a = 0$  ? no

$a = 1$  ? yes

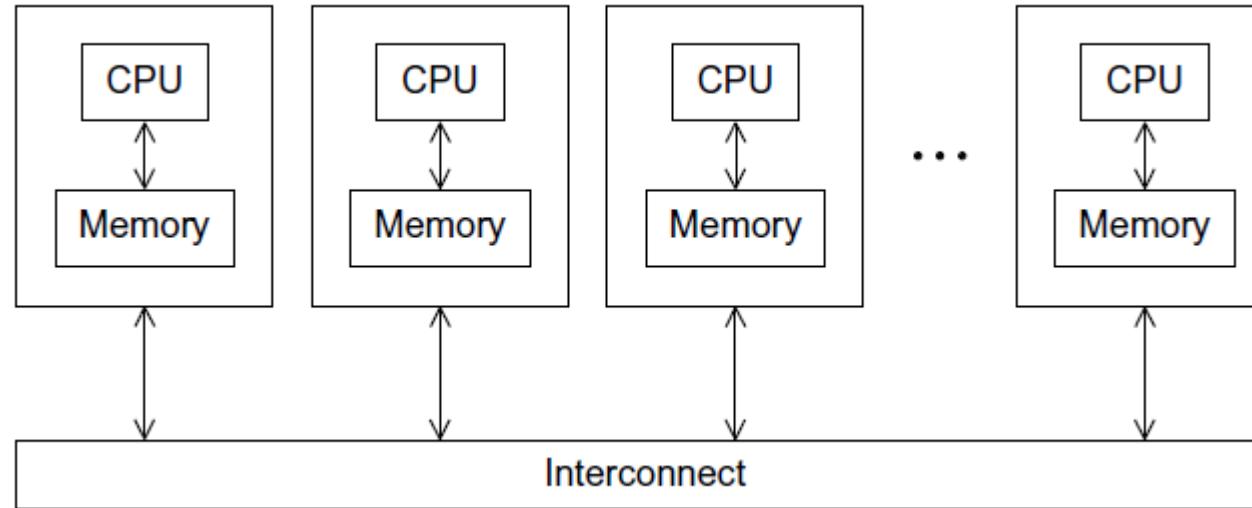
$a = 2$  ? yes

# References

## Chapter 2

- An Introduction to Parallel Programming  
by Peter Pacheco.

# Distributed Programming



- Distributed programming using message passing
- Processes communicate by sending and receiving messages

# Message Passing Interface(MPI)

MPI: message passing interface

A library for send, receive message

Compilation command:

```
mpicc -g -Wall -o <binary name> <filename>.c
```

Execution command:

```
mpiexec -n <number of processes> ./<binary name>
```

# Example

```
const int MAX_STRING = 100;
```

```
int main(void) {
    char greeting[MAX_STRING];
    int comm_sz; /* Number of processes */
    int my_rank; /* My process rank */
```

```
MPI_Init(NULL, NULL);
MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
```

```
if (my_rank != 0) {
    MPI_Send(greeting, strlen(greeting)+1, MPI_CHAR, 0, 0, MPI_COMM_WORLD);
} else {
    for (int q = 1; q < comm_sz; q++) {
        MPI_Recv(greeting, MAX_STRING, MPI_CHAR, q, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }
}
```

```
MPI_Finalize();
return 0;
} /* main */
```

MPI calls are prefixed with **MPI\_**

# MPI Init

**MPI\_Init**: used for initial setup

```
int MPI Init(  
int* argc_p /* in/out */,  
char*** argv_p /* in/out */);
```

- allocate storage for message buffers
- assigns ranks to processes
- define a communicator

# MPI Finalize

**MPI\_Finalize:** completes the MPI programming

```
int MPI_Finalize(void);
```

- any resources allocated for MPI can be freed

# MPI Program Structure

```
...
#include <mpi.h>
...
int main(int argc, char* argv[]) {
    ...
    /* No MPI calls before this */
    MPI_Init(&argc, &argv);
    ...
    MPI_Finalize();
    /* No MPI calls after this */
    ...
    return 0;
}
```

# MPI Communication

Communicators: MPI\_COMM\_WORLD

\* consists of all the processes started by the user

**MPI\_Comm\_rank**

```
int MPI_Comm_rank(MPI_Comm comm /* in - a communicator */,  
                  int my_rank_p /* out */);
```

**MPI\_Comm\_size**

```
int MPI_Comm_size(MPI_Comm comm /* in - a communicator */,  
                  int* comm_sz_p /* out */);
```

# SPMD Programs

SPMD: single program, multiple data

The program launches multiple processes

The program runs with any number of processes

```
if (my_rank != 0) {  
    MPI_Send(greeting, strlen(greeting)+1, MPI_CHAR, 0, 0, MPI_COMM_WORLD);  
} else {  
    for (int q = 1; q < comm_sz; q++) {  
        MPI_Recv(greeting, MAX_STRING, MPI_CHAR, q, 0, MPI_COMM_WORLD,  
MPI_STATUS_IGNORE);  
    }  
}
```

# Communication

```
int MPI_Send(  
    void*           msg_buf_p    /* in */,  
    int             msg_size     /* in */,  
    MPI_Datatype   msg_type    /* in */,  
    int             dest        /* in */,  
    int             tag         /* in */,  
    MPI_Comm       communicator /* in */);
```

msg\_buf\_p, msg\_size, and msg\_type: determine the contents of the message

dest, tag, and communicator: determine the destination of the message.

# Communication

```
int MPI_Send(  
    void*           msg_buf_p      /* in */,  
    int             msg_size       /* in */,  
    MPI_Datatype   msg_type       /* in */,  
    int             dest          /* in */,  
    int             tag           /* in */,  
    MPI_Comm        communicator /* in */);
```

msg\_buf\_p: pointer to message content

msg\_size and msg\_type: determine the amount of data to be sent

# Communication

			MPI datatype	C datatype
int MPI_Send(				
void*	msg_buf_p	/* in */,	MPI_CHAR	<b>signed char</b>
int	msg_size	/* in */,	MPI_SHORT	<b>signed short int</b>
MPI_Datatype	msg_type	/* in */,	MPI_INT	<b>signed int</b>
int	dest	/* in */,	MPI_LONG	<b>signed long int</b>
int	tag	/* in */,	MPI_LONG_LONG	<b>signed long long int</b>
MPI_Comm	communicator	/* in */);	MPI_UNSIGNED_CHAR	<b>unsigned char</b>
			MPI_UNSIGNED_SHORT	<b>unsigned short int</b>
			MPI_UNSIGNED	<b>unsigned int</b>
			MPI_UNSIGNED_LONG	<b>unsigned long int</b>
			MPI_FLOAT	<b>float</b>
			MPI_DOUBLE	<b>double</b>
			MPI_LONG_DOUBLE	
			MPI_BYTE	
			MPI_PACKED	

# Communication

```
int MPI_Send(  
    void*          msg_buf_p    /* in */,  
    int            msg_size     /* in */,  
    MPI_Datatype   msg_type     /* in */,  
    int            dest         /* in */,  
    int            tag          /* in */,  
    MPI_Comm       communicator /* in */);
```

dest: specifies the rank of the process that should receive the message.

tag: distinguishes messages.

MPI\_Comm: specifies the collection of processes or communicator that can send messages to each other.

\* Message is not sent outside the communicator processes.

# Communication

```
int MPI_Recv(  
    void*           msg_buf_p    /* out */,  
    int             buf_size     /* in  */,  
    MPI_Datatype   buf_type    /* in  */,  
    int             source       /* in  */,  
    int             tag          /* in  */,  
    MPI_Comm        communicator /* in  */,  
    MPI_Status*    status_p     /* out */);
```

msg\_buf\_, buf\_size, buf\_type specifies the memory that receives messages.

source, tag, communicator identify the messages.

# Communication

Suppose process *q* calls:

```
MPI_Send(send_buf_p, send_buf_sz, send_type, dest, send_tag, send_comm);
```

Suppose process *r* calls:

```
MPI_Recv(recv_buf_p, recv_buf_sz, recv_type, src, recv_tag, recv_comm, &status);
```

the message sent by *q* can be received by *r* if:

- `recv_comm = send_comm`
- `recv_tag = send_tag`
- `dest = r`, and
- `src = q.`
- `recv_type = send_type`
- `recv_buf_sz >= send_buf_sz`

# Communication

The MPI\_ANY\_SOURCE and MPI\_ANY\_TAG enables a receiver can receive a message without knowing

- the amount of data in the message,
- the sender of the message, or
- the tag of the message.

status\_p argument in the MPI\_Recv() preserves

- status.MPI\_SOURCE
- status.MPI\_TAG

# Semantics of MPI\_Send and MPI\_Recv

## MPI\_Send()

- Adds some data along with the message
- Either buffer the message and return or
- Block until the message is sent
- Implementation dependent (defines a cutoff for message size)

if message size <= cutoff then buffer and return  
else block unless the message is sent

## MPI\_Recv()

- always blocks until a matching message has been received.

# Message Ordering

Two messages from same sender do not reorder

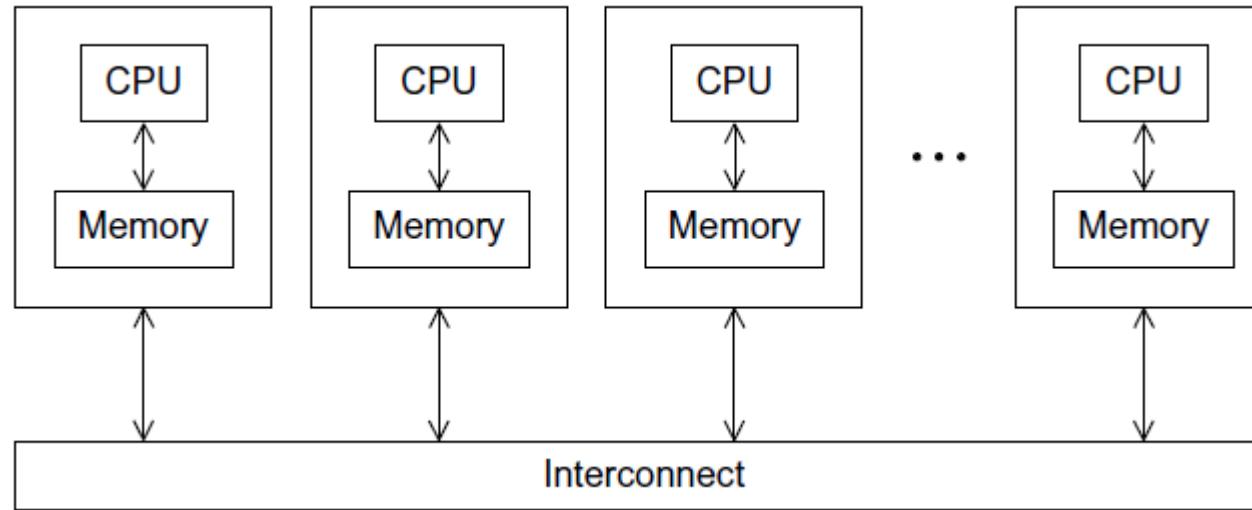
Messages from different processes may reach a receiver in any order.

MPI do not impose any performance restriction on network.

# References

- Chapter 3.1-3.3  
An Introduction to Parallel Programming  
by Peter Pacheco.

# Distributed Programming



- Distributed programming using MPI
- MPI\_Init(), MPI\_Finalize(), MPI\_Send(), MPI\_Recv()

# Message Ordering

Two messages from same sender do not reorder

Messages from different processes may reach a receiver in any order.

MPI do not impose any performance restriction on network.

# Remember the Example

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

Consider that the array  $a[]$  is distributed and each element is accessible to one process each.

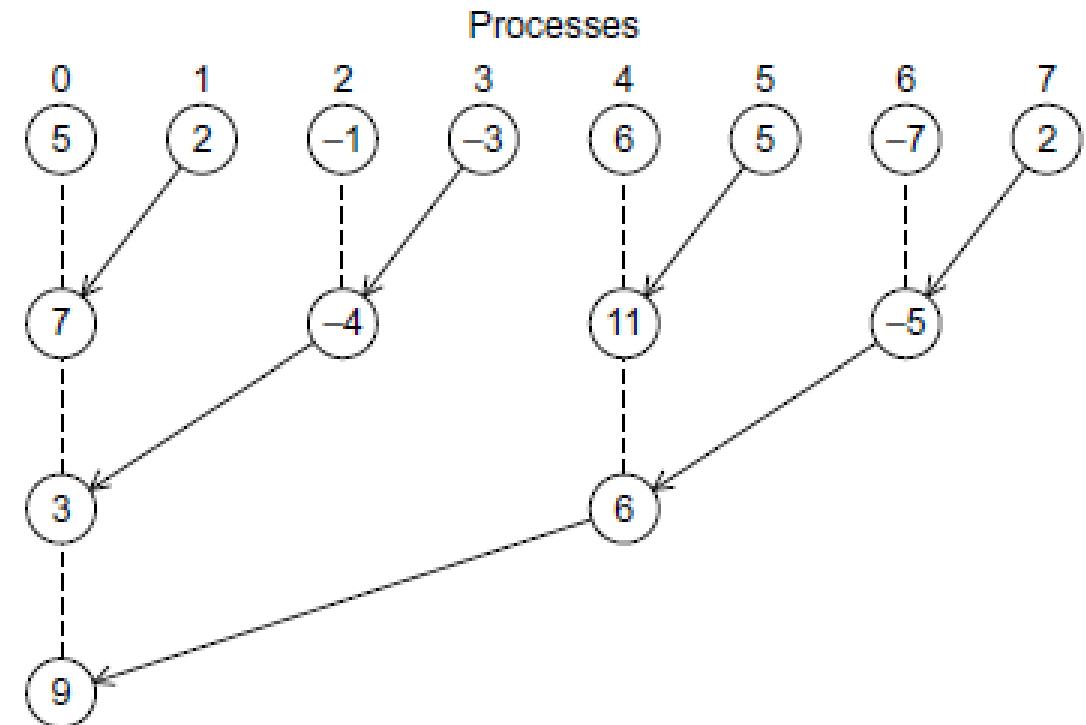
Solution:

Each process communicate the values to process 0 for computation.

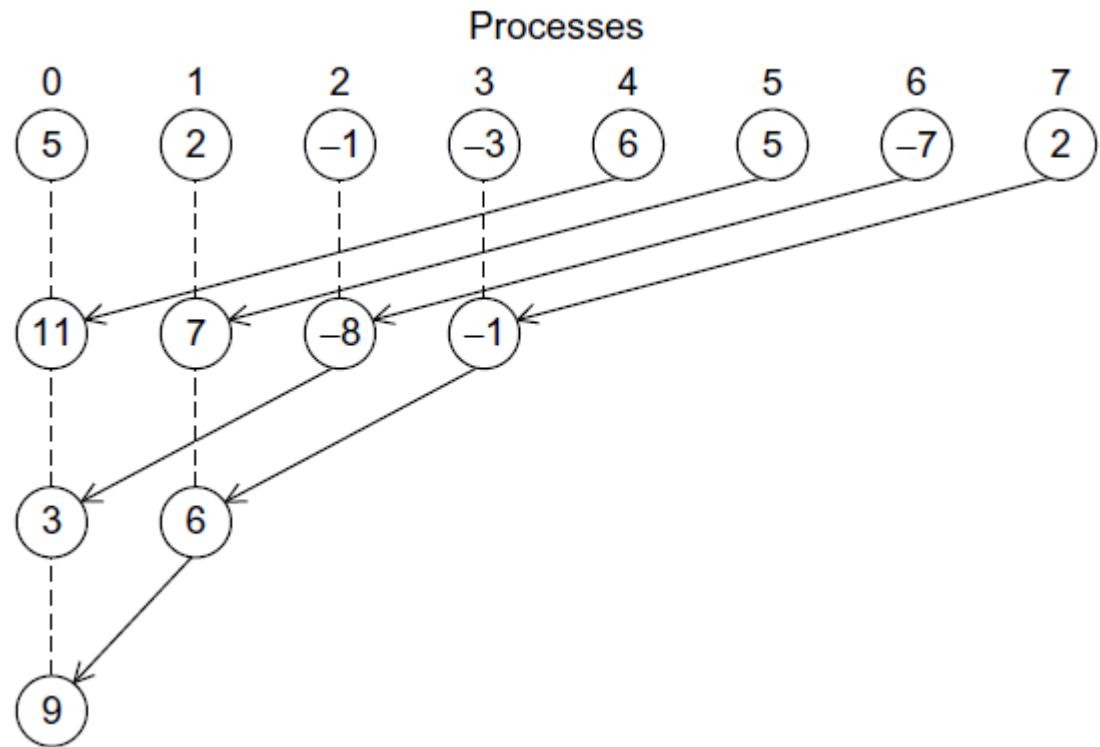
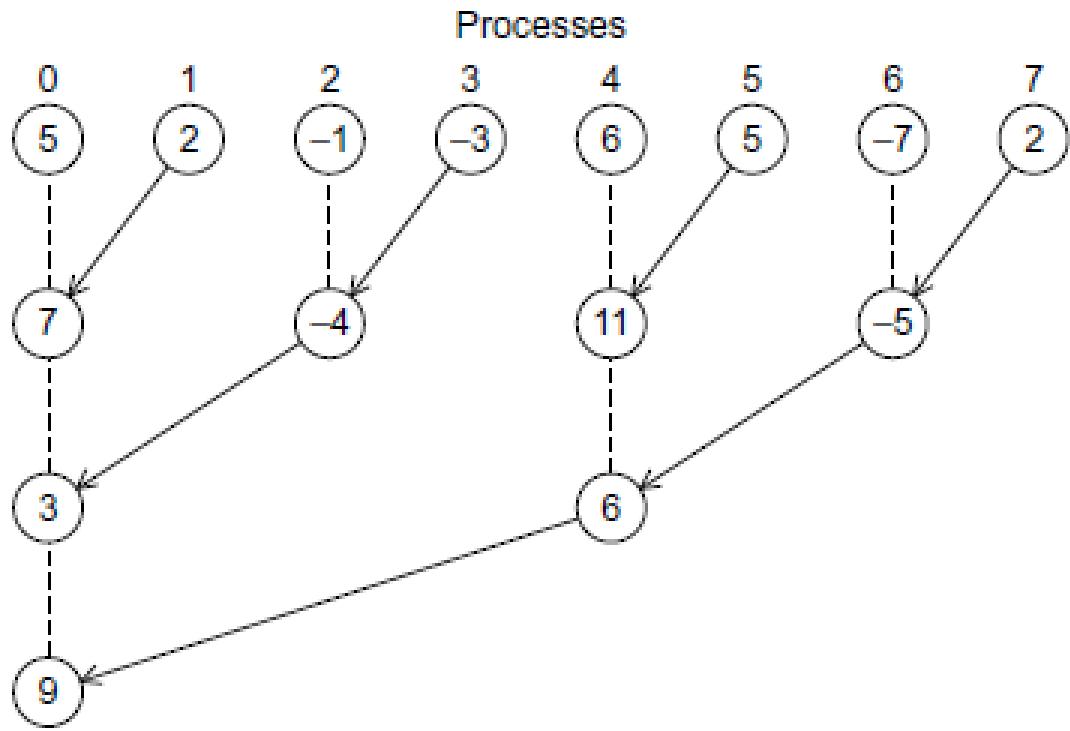
$N-1$  messages and  $N-1$  additions.

# Tree Based Communication

1.
  - a. Processes 2 and 6 send their new values to processes 0 and 4, respectively.
  - b. Processes 0 and 4 add the received values into their new values.
  
2.
  - a. Process 4 sends its newest value to process 0.
  - b. Process 0 adds the received value to its newest value.



# Alternatives



Performance may vary across systems

# Collective Communication

communication functions that involve all the processes in a communicator

```
int MPI_Reduce(  
    void*           input_data_p /* in */,  
    void*           output_data_p /* out */,  
    int             count        /* in */,  
    MPI_Datatype   datatype    /* in */,  
    MPI_Op          operator     /* in */,  
    int             dest_process /* in */,  
    MPI_Comm        comm        /* in */);
```

Operation Value	Meaning
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical and
MPI_BAND	Bitwise and
MPI_LOR	Logical or
MPI_BOR	Bitwise or
MPI_LXOR	Logical exclusive or
MPI_BXOR	Bitwise exclusive or
MPI_MAXLOC	Maximum and location of maximum
MPI_MINLOC	Minimum and location of minimum

# Another Example

```
1 int main(void) {
2     int my_rank, comm_sz, n = 1024, local_n;
3     double a = 0.0, b = 3.0, h, local_a, local_b;
4     double local_int, total_int;
5     int source;
6
7     MPI_Init(NULL, NULL);
8     MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
9     MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
10
11    h = (b-a)/n;          /* h is the same for all processes */
12    local_n = n/comm_sz;   /* So is the number of trapezoids */
13
14    local_a = a + my_rank*local_n*h;
15    local_b = local_a + local_n*h;
16    local_int = Trap(local_a, local_b, local_n, h);
17
18    if (my_rank != 0) {
19        MPI_Send(&local_int, 1, MPI_DOUBLE, 0, 0,
20                  MPI_COMM_WORLD);
21    } else {
22        total_int = local_int;
23        for (source = 1; source < comm_sz; source++) {
24            MPI_Recv(&local_int, 1, MPI_DOUBLE, source, 0,
25                      MPI_COMM_WORLD, MPI_STATUS_IGNORE);
26            total_int += local_int;
27        }
28    }
29
30    if (my_rank == 0) {
31        printf("With n = %d trapezoids, our estimate\n", n);
32        printf("of the integral from %f to %f = %.15e\n",
33              a, b, total_int);
34    }
35    MPI_Finalize();
36    return 0;
37 } /* main */
```

```
16    local_int = Trap(local_a, local_b, local_n, h);
17
18    if (my_rank != 0) {
19        MPI_Send(&local_int, 1, MPI_DOUBLE, 0, 0,
20                  MPI_COMM_WORLD);
21    } else {
22        total_int = local_int;
23        for (source = 1; source < comm_sz; source++) {
24            MPI_Recv(&local_int, 1, MPI_DOUBLE, source, 0,
25                      MPI_COMM_WORLD, MPI_STATUS_IGNORE);
26            total_int += local_int;
27        }
28    }
29
30    MPI_Reduce(&local_int, &total_int, 1,
31                MPI_DOUBLE, MPI_SUM, 0,
32                MPI_COMM_WORLD);
```

# Collective vs. point-to-point communications

All the processes in the communicator must call the same collective function

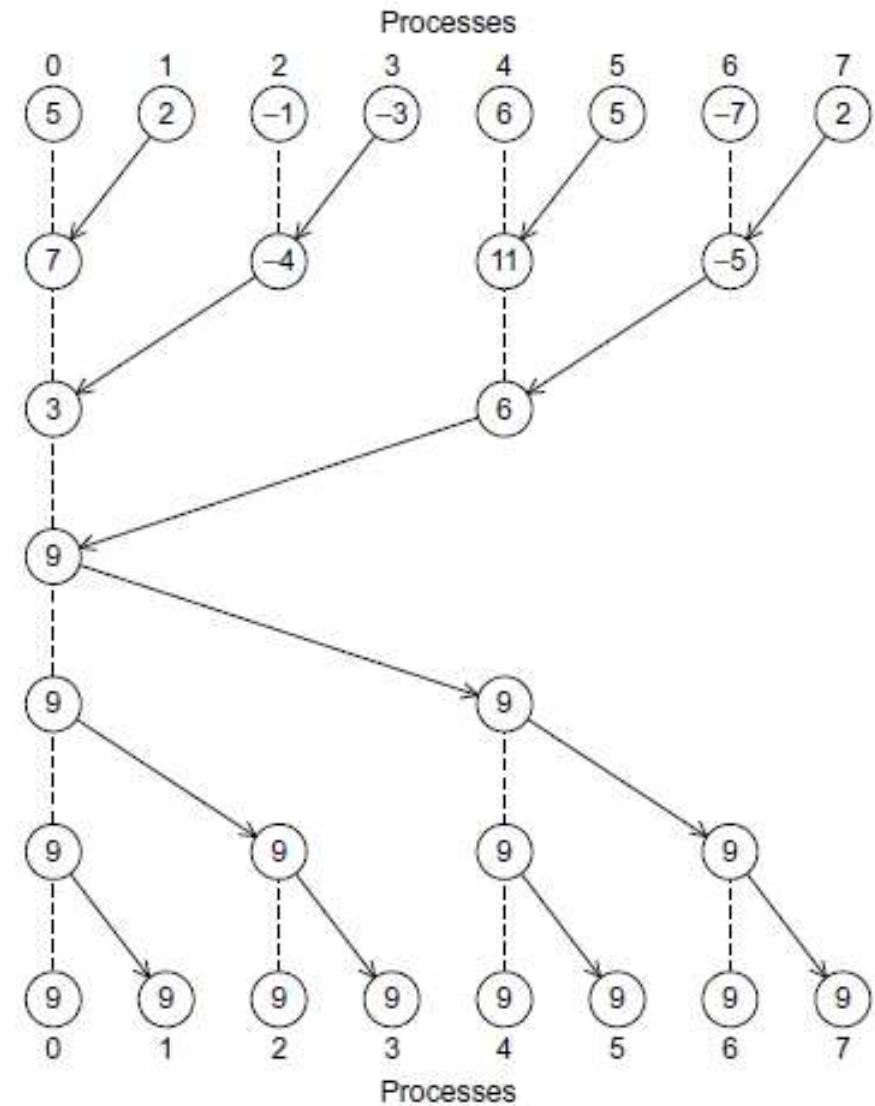
The arguments passed by each process to an MPI collective communication must be “compatible.”

...

# MPI AllReduce

```
int MPI_Allreduce(  
    void*           input_data_p /* in */,  
    void*           output_data_p /* out */,  
    int             count          /* in */,  
    MPI_Datatype   datatype      /* in */,  
    MPI_Op          operator       /* in */,  
    MPI_Comm        comm          /* in */);
```

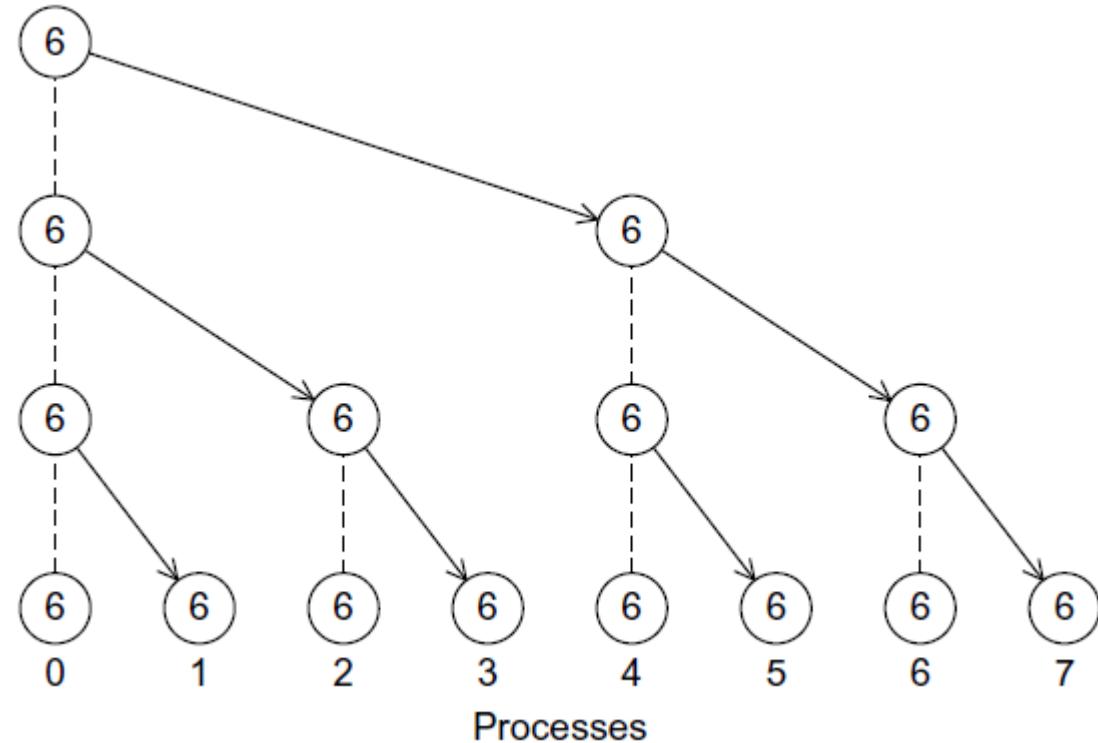
- Reduce to compute the result
- Store the result in all processes



# MPI Broadcast

```
int MPI_Bcast(  
    void*          data_p      /* in/out */,  
    int            count       /* in      */,  
    MPI_Datatype  datatype   /* in      */,  
    int            source_proc /* in      */,  
    MPI_Comm       comm       /* in      */);
```

- Broadcast data\_p to all processes



# Data Partitioning

Block partitioning

Cyclic partitioning

Block-cyclic partitioning

Process	Components									Block-Cyclic			
	Block					Cyclic				Blocksize = 2			
0	0	1	2	3	0	3	6	9	0	1	6	7	
1	4	5	6	7	1	4	7	10	+6'	3	8	9	
2	8	9	10	11	2	5	8	11	4	5	10	11	

# Data Partitioning

MPI\_Scatter performs block partitioning

```
int MPI_Scatter(
    void* send_buf_p /* in */,
    int send_count /* in */,
    MPI_Datatype send_type /* in */,
    void* recv_buf_p /* out */,
    int recv_count /* in */,
    MPI_Datatype recv_type /* in */,
    int src_proc /* in */,
    MPI_Comm comm /* in */);

1 void Read_vector(
2     double local_a[] /* out */,
3     int local_n /* in */,
4     int n /* in */,
5     char vec_name[] /* in */,
6     int my_rank /* in */,
7     MPI_Comm comm /* in */) {
8
9     double* a = NULL;
10    int i;
11
12    if (my_rank == 0) {
13        a = malloc(n*sizeof(double));
14        printf("Enter the vector %s\n", vec_name);
15        for (i = 0; i < n; i++)
16            scanf("%lf", &a[i]);
17        MPI_Scatter(a, local_n, MPI_DOUBLE, local_a, local_n,
18                    MPI_DOUBLE, 0, comm);
19        free(a);
20    } else {
21        MPI_Scatter(a, local_n, MPI_DOUBLE, local_a, local_n,
22                    MPI_DOUBLE, 0, comm);
23    }
24 } /* Read_vector */
```

# Data Partitioning

MPI\_Gather accumulates  
the data

```
int MPI_Gather(  
    void* send_buf_p /* in */,  
    int send_count /* in */,  
    MPI_Datatype send_type /* in */,  
    void* recv_buf_p /* out */,  
    int recv_count /* in */,  
    MPI_Datatype recv_type /* in */,  
    int dest_proc /* in */,  
    MPI_Comm comm /* in */);
```

```
1 void Print_vector(  
2     double local_b[] /* in */,  
3     int local_n /* in */,  
4     int n /* in */,  
5     char title[] /* in */,  
6     int my_rank /* in */,  
7     MPI_Comm comm /* in */) {  
8  
9     double* b = NULL;  
10    int i;  
11  
12    if (my_rank == 0) {  
13        b = malloc(n*sizeof(double));  
14        MPI_Gather(local_b, local_n, MPI_DOUBLE, b, local_n,  
15                    MPI_DOUBLE, 0, comm);  
16        printf("%s\n", title);  
17        for (i = 0; i < n; i++)  
18            printf("%f ", b[i]);  
19        printf("\n");  
20        free(b);  
21    } else {  
22        MPI_Gather(local_b, local_n, MPI_DOUBLE, b, local_n,  
23                    MPI_DOUBLE, 0, comm);  
24    }  
25 } /* Print_vector */
```

# Performance Evaluation

$$T_{\text{parallel}}(n,p) = T_{\text{serial}}(n)/p + T_{\text{overhead}}$$

$$\text{Speedup } S(n,p) = T_{\text{serial}}(n) / T_{\text{parallel}}(n,p)$$

$$\text{Efficiency } E(n,p) = S(n,p) / p$$

# Scalability

A solution/program scalable if the problem size can be increased at a rate so that the efficiency doesn't decrease as the number of processes increase.

comm_sz	Order of Matrix					Efficiency
	1024	2048	4096	8192	16,384	
1	1.00	1.00	1.00	1.00	1.00	
2	0.89	0.94	0.97	0.96	0.98	
4	0.51	0.78	0.89	0.96	0.98	
8	0.30	0.61	0.82	0.94	0.98	
16	0.15	0.39	0.68	0.89	0.97	

# References

- Chapter 3.4-3.6  
An Introduction to Parallel Programming  
by Peter Pacheco

# Models for Parallel Computation

parallel random access machine (PRAM)

- All processors access the same shared memory.
- Processors share a common clock.
- May execute different instructions in each cycle

# PRAM Types

- Exclusive-read, exclusive-write (EREW) PRAM.  
No concurrent read/write operations are allowed.
- Concurrent-read, exclusive-write (CREW) PRAM.  
multiple read accesses to a memory location are allowed.  
Multiple write accesses to a memory location are serialized.
- Exclusive-read, concurrent-write (ERCW) PRAM.  
Multiple write accesses to a memory location are allowed.  
Multiple read accesses are serialized.
- Concurrent-read, concurrent-write (CRCW) PRAM.  
Multiple read and write accesses to a common memory location are allowed.

# Problems with Multiple Accesses

Concurrent accesses in CREW, ERCW, EREW :

- Violates mutual exclusion
- Results in data race

# Protocols for Concurrent Writes

**Common:** concurrent write is allowed if all the values that the processors are attempting to write are identical.

**Arbitrary:** an arbitrary processor is allowed to proceed to write and the rest fail.

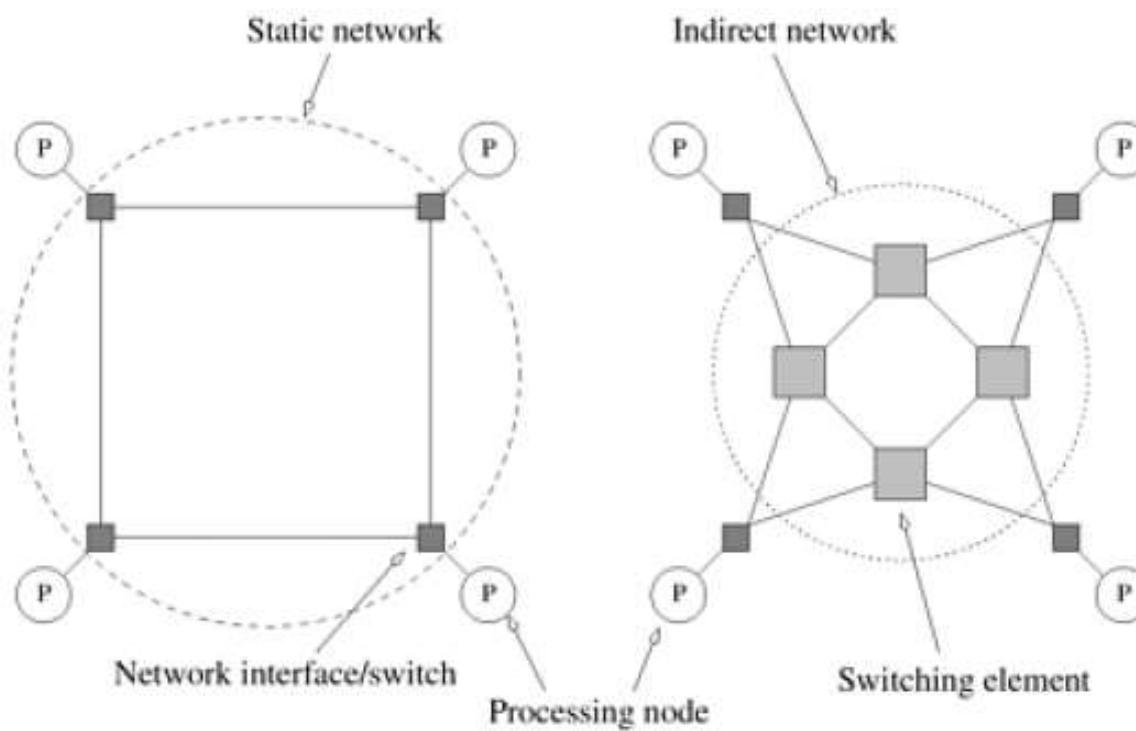
**Priority:** all processors are prioritized and the processor with the highest priority succeeds and the rest fail.

**Sum:** the sum of all the quantities is written

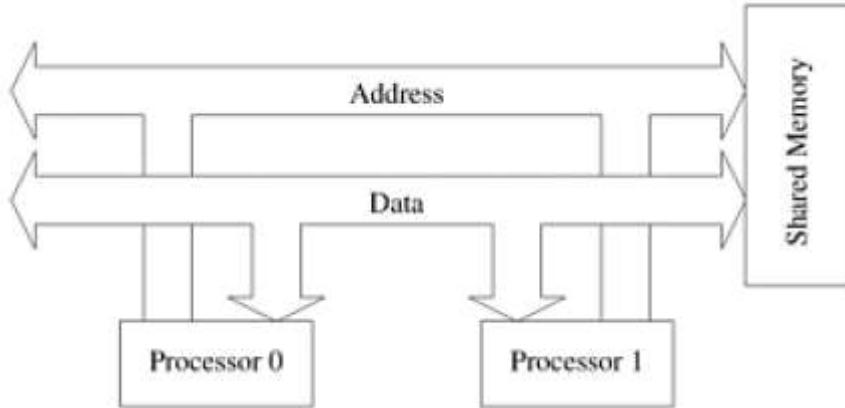
# Interconnect Network

Static, point to point network

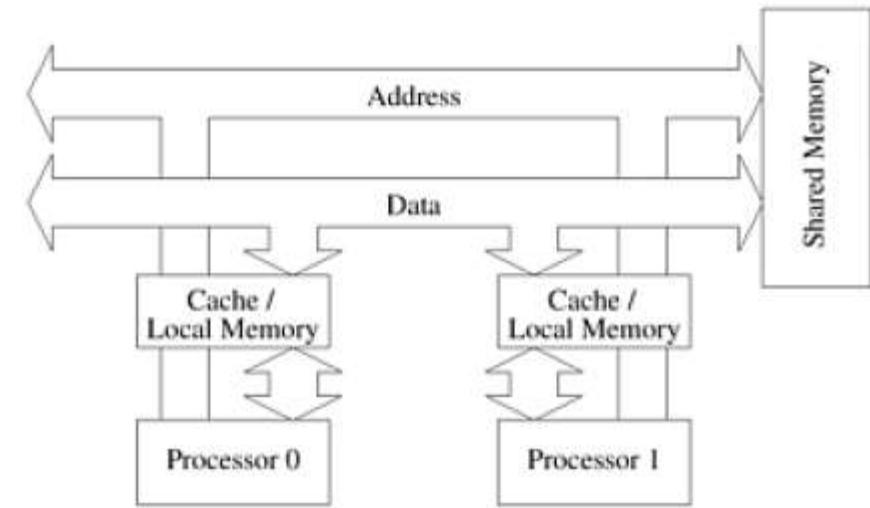
Dynamic/indirect, using switches and communication links



# Network Topologies

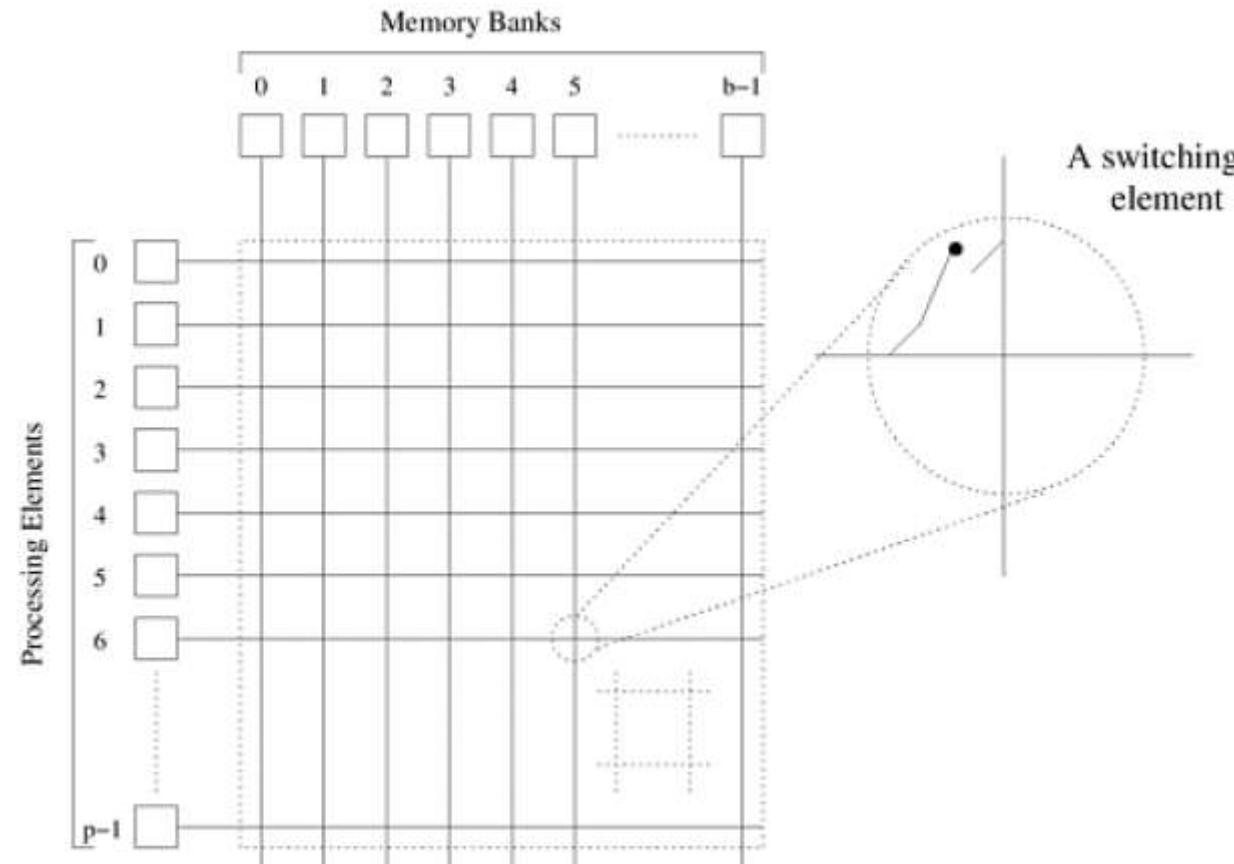


No local cache



With local cache

# CrossBar Network



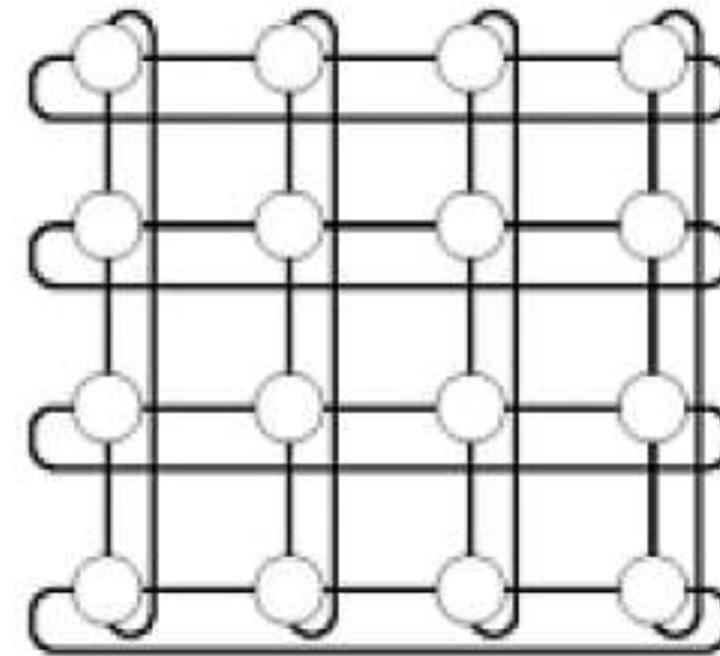
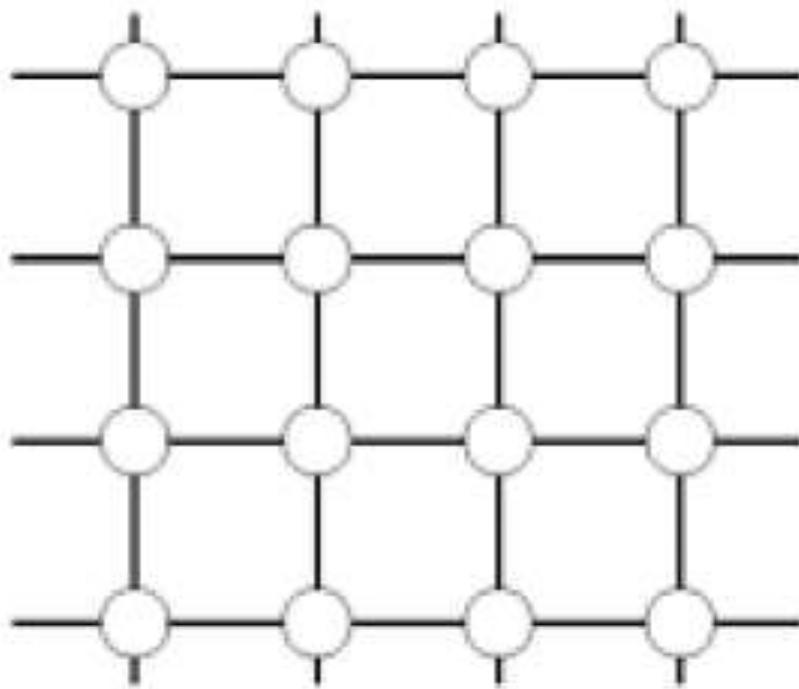
A processor-memory connection does not block another processor-memory connection. Network size is  $p * b$ .

# Linear Array Network



Nodes have left and right neighbor

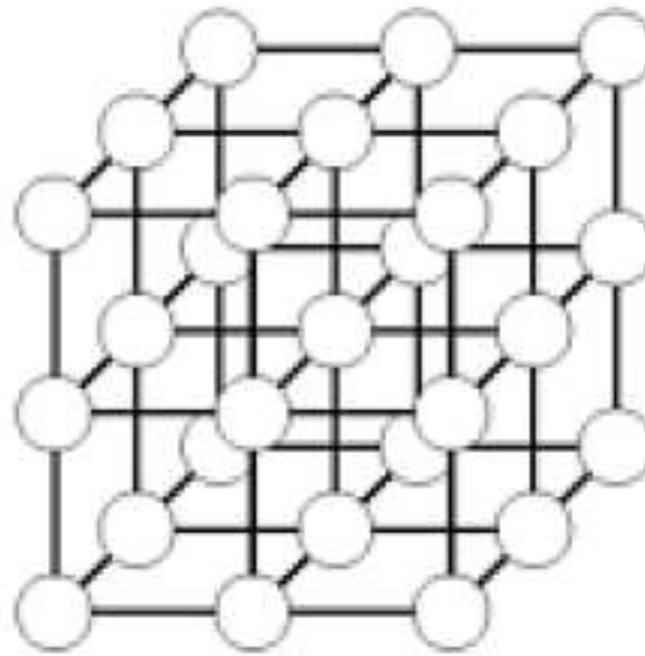
# 2D Mesh Network



$p$  processor  $\rightarrow$  dimension size is  $\text{sqrt}(p)$

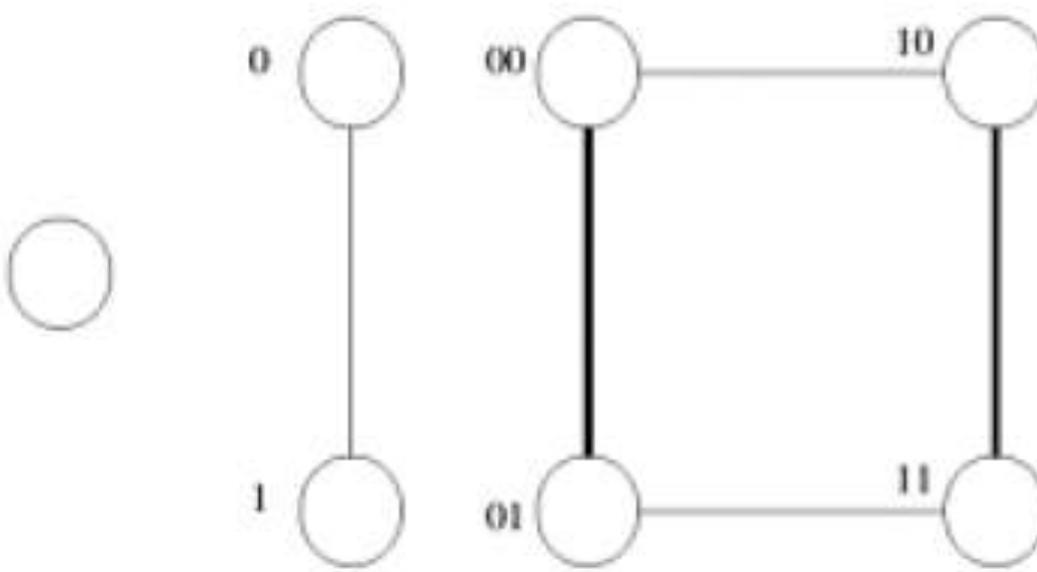
A node in the network is referred by  $(i,j)$  position

# 3D Mesh Network

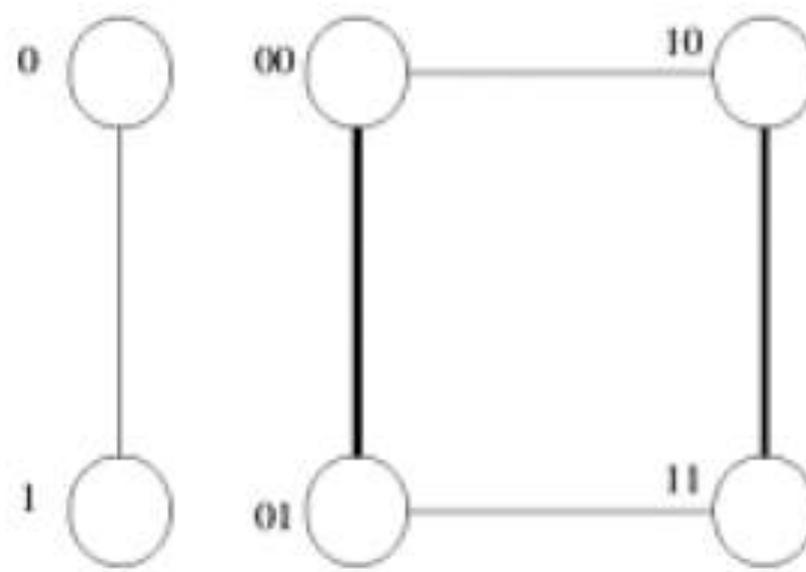


Many 3D applications map naturally to 3D mesh network

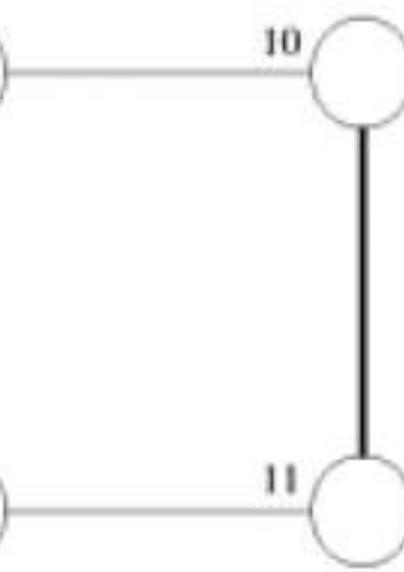
# Hypercube



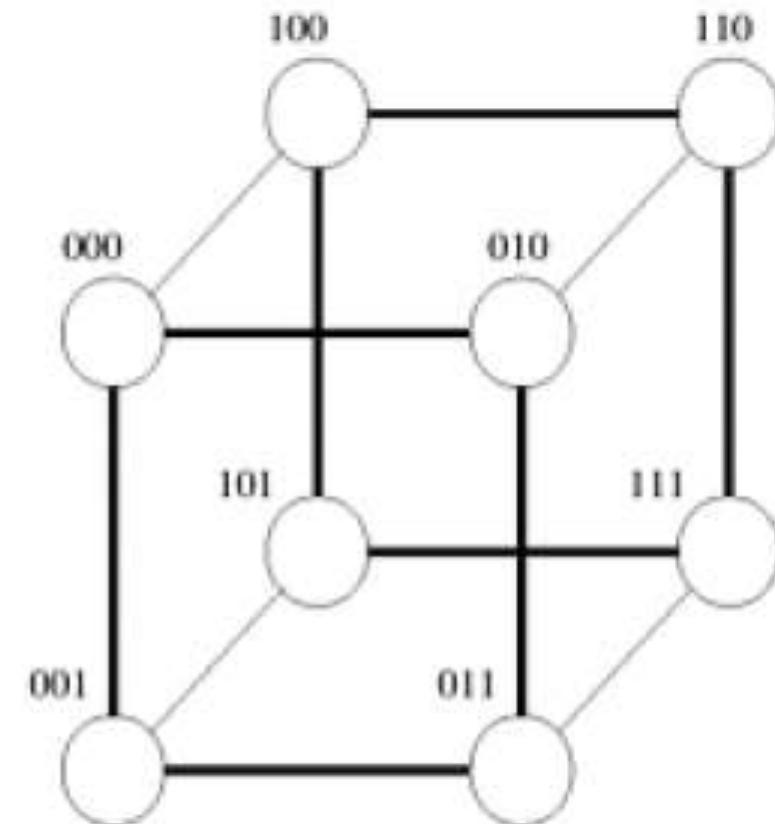
0-D hypercube



1-D hypercube



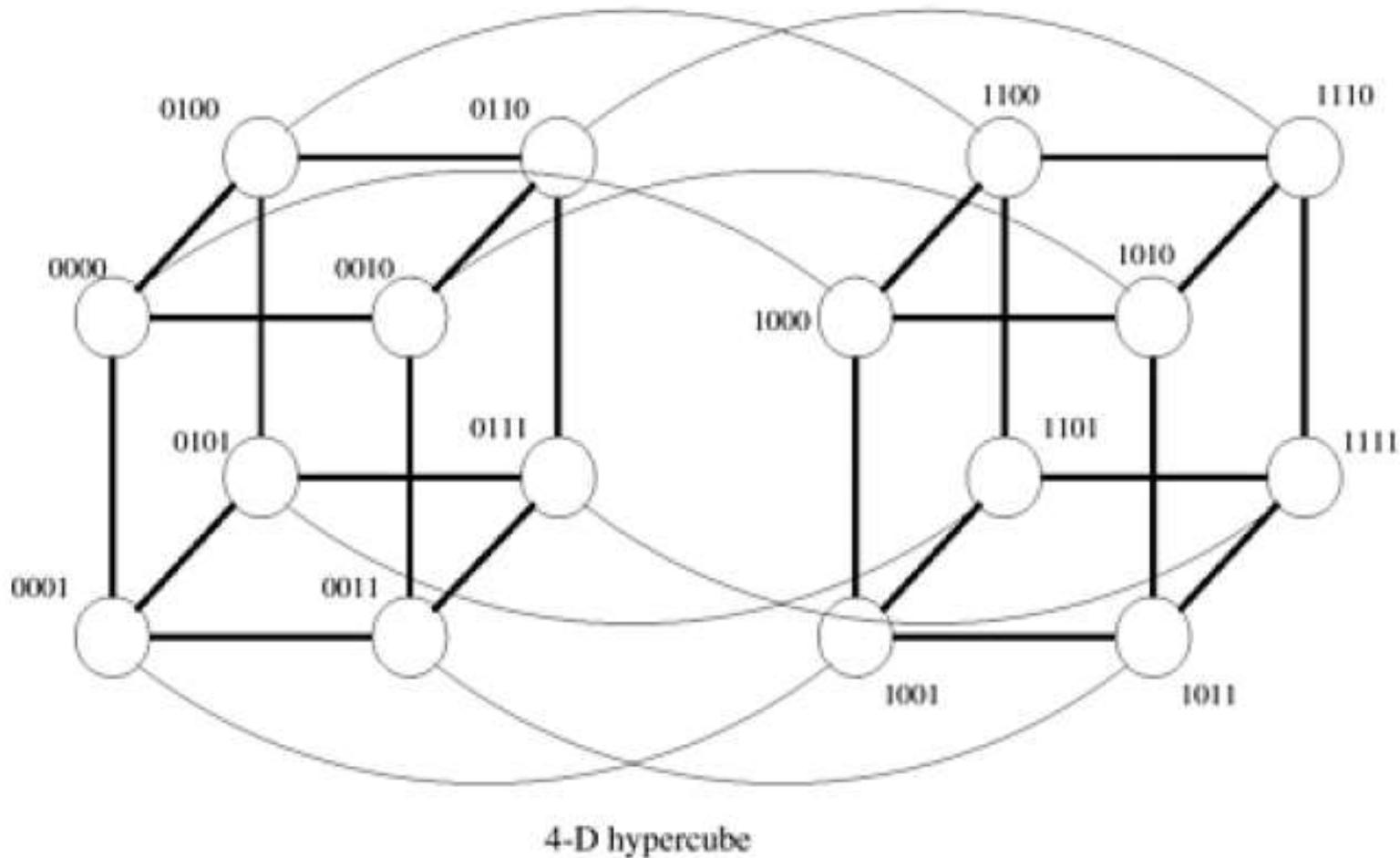
2-D hypercube



3-D hypercube

# Hypercube

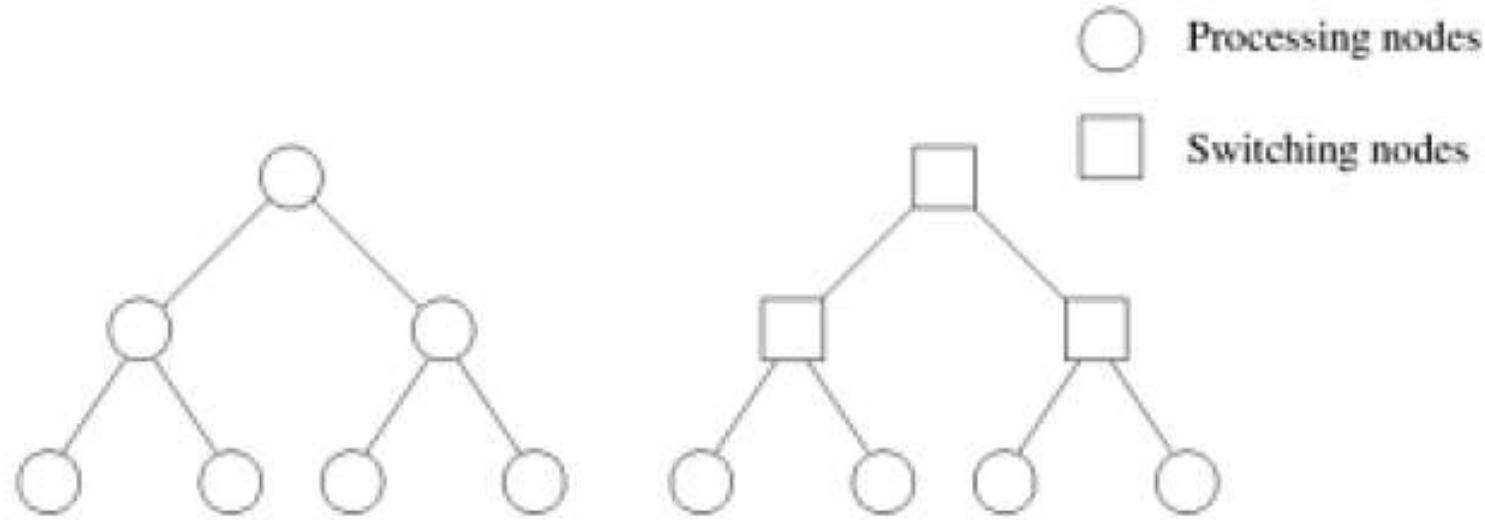
0  
3D-  
Hyper-  
cube



1  
3D-  
Hyper-  
cube

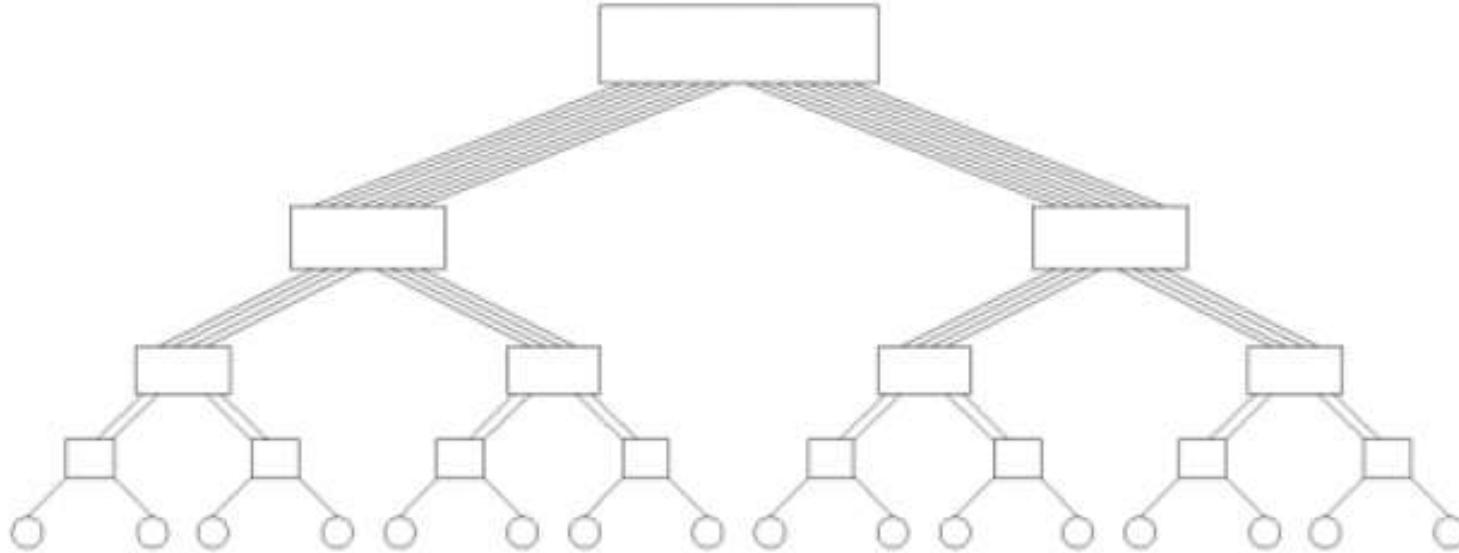
minimum distance = no. of different bits in the two processors

# Tree Network



Suffers from communication bottleneck near the root

# FAT Tree Network



More number of links and switches near the root

# Communication Cost

Communication time =

time to prepare a message for transmission +

time taken for the message to reach from source to destination

Startup time ( $T_s$ )

time required to handle the message at the receiver and sender

Per hop time or node latency ( $T_h$ )

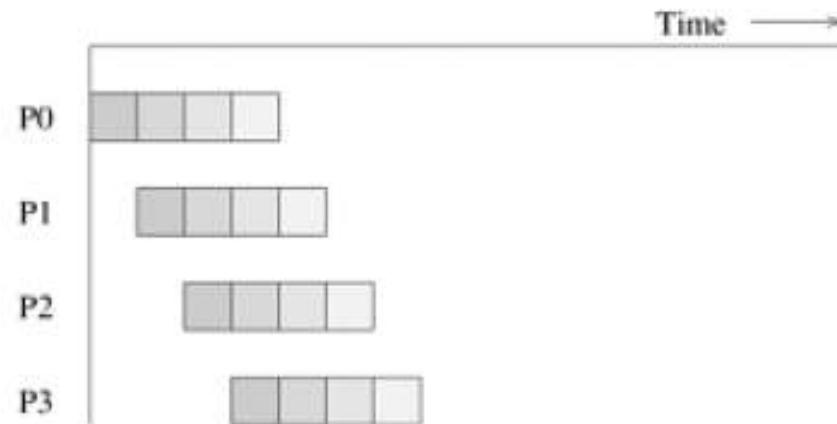
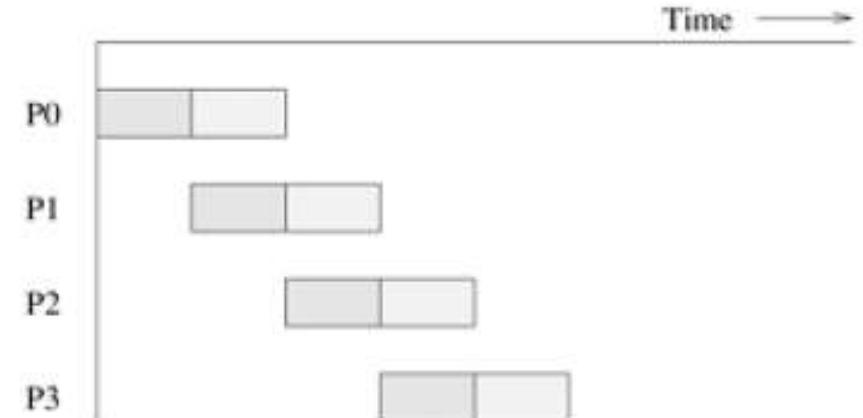
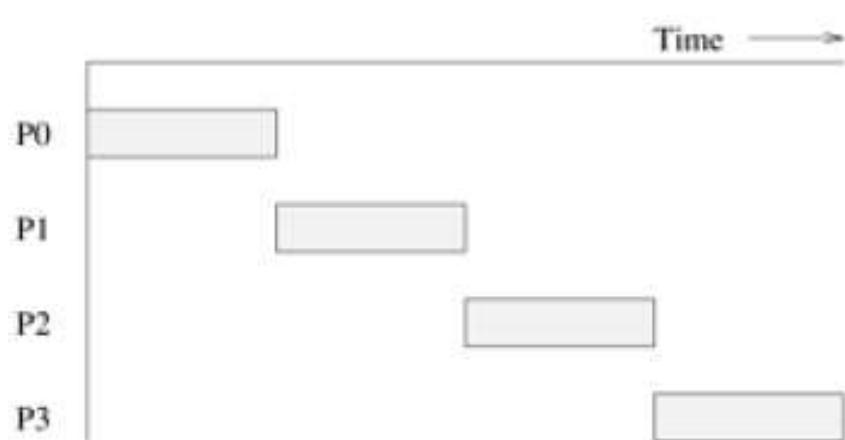
time taken for a message reach from one node to another in the network

Per word transfer time ( $T_w$ )

$T_w = 1/r$  where  $r$  = bandwidth i.e.  $r$  words per seconds

# Store-and-Forward Routing

Each node in the network receives and stores the entire message before forwarding



# Communication Time

Suppose a message of  $m$  word traverses  $l$  links

$$T_{\text{comm}} = T_s + (m * T_w + T_h) * l$$



$$T_{\text{comm}} = T_s + m * T_w * l$$

when  $T_h$  is small

# Message Routing by Packets

A message is broken into packets for transmission

- Lower overhead for packet loss
- Routing can be more efficient

Suppose

message size is  $m$  word,

$r$  is packet size

$s$  is additional information in each packet

$T_{w1}$  is the time taken to create packetize a word of message

$m \cdot T_{w1}$  is the time to packetize the message

# Message Routing by Packets

Network send one word in  $T_w$  seconds

$T_h$  is the delay in each hop and there are  $l$  hops

Time taken for the first packet =  $T_h \cdot l + T_w \cdot (r+s)$

Destination node receives a packet in each  $T_w \cdot (r+s)$  seconds

Number of packets after the first packet:  $(m/r)-1$

Total communication time  $T_{comm}$

$$= T_s + m \cdot T_w + T_h \cdot l + T_w \cdot (r+s) + ((m/r)-1) \cdot T_w \cdot (r+s)$$

$$= T_s + T_h \cdot l + m \cdot T_w \text{ where } T_w = T_w + T_w \cdot (1+s/r)$$

# Cut Through Routing

All packets pass through same path.

No routing information is included in message.

Message is broken into *flits*

$l$  links, per hop time is  $T_h$ ,

Message header takes  $l * T_h$  seconds to reach the destination

$m * T_w$  seconds for the message to reach the destination from source after the header

$$T_{comm} = T_s + l * T_h + m * T_w$$

# Simplified Model

$$T_{\text{comm}} = T_s + l * T_h + m * T_w$$

$T_s \gg T_h$  and  $T_s \gg T_w$

- Aggregate small messages in a big message and communicate
- Reduce the volume of data ( $T_w$ )
- Minimize data transfer distance ( $l$ )

If  $l$  is small then

$$T_{\text{comm}} = T_s + m * T_w$$

# Reference

Chapter 2.4-2.5

Introduction to Parallel Computing

Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar

# Recap

parallel random access machine (PRAM)

- \* EREW, CREW, ERCW, CRCW

Network topology

Communication time

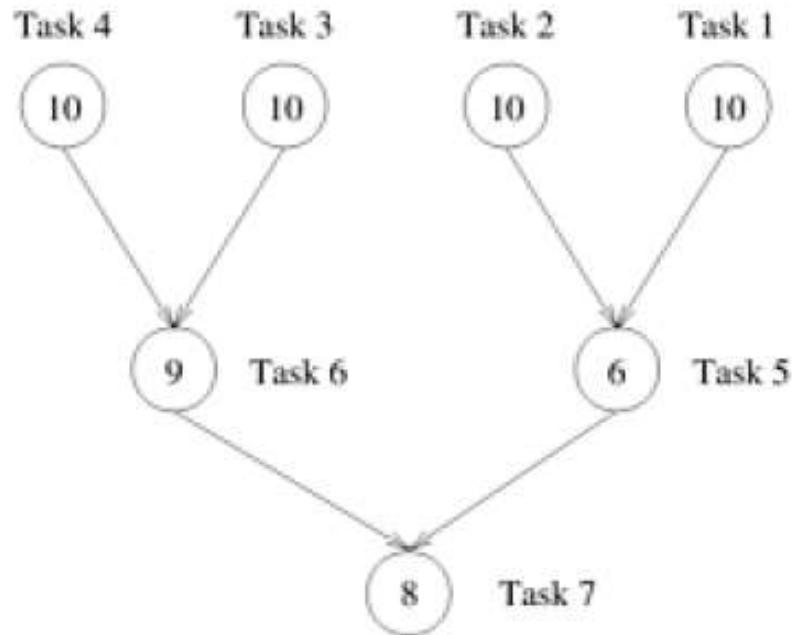
# Principles of Parallel Algorithm Design

- Identify independent(that can be performed concurrently) tasks.
- Map tasks to processes/threads considering task dependencies.
- Distribute data
- Access shared memory safely.
- Manage Synchronization.

# Task Dependency Graph

- nodes are tasks and the directed edges are dependencies amongst them.
- A directed acyclic graph(DAG).
- A task can be executed if all tasks connected to this node by incoming edges have completed.
- Task-dependency graphs can be disconnected.

# Example: Task Graph

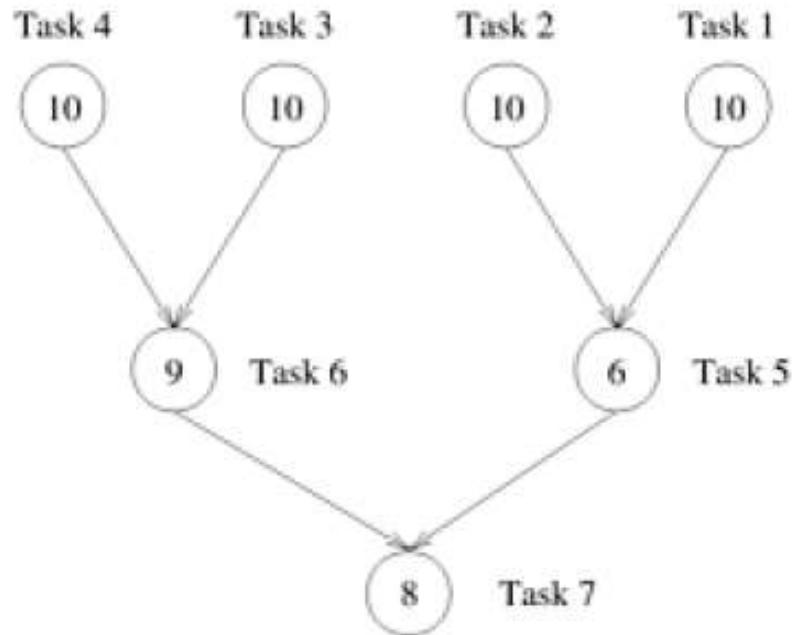


The values in the nodes indicates the amount of work for the task

Critical Path: dependency path with maximum weight from start to finish

Critical path length: maximum weight from start to finish

# Example: Task Graph



average degree of concurrency = total work / critical path length

total work = 63, critical path length = 27

# Task Interaction Graph

An undirected graph that captures the interactions among tasks

A ‘supergraph’ of task dependence graph

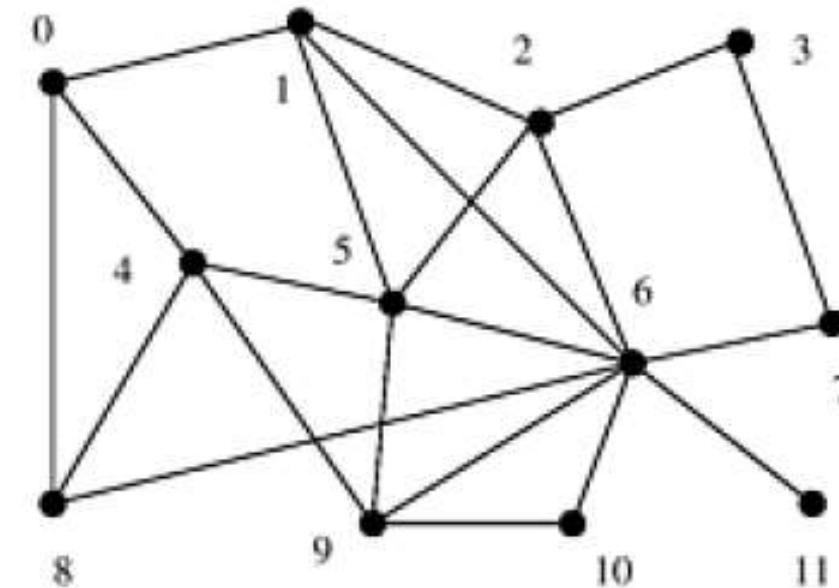
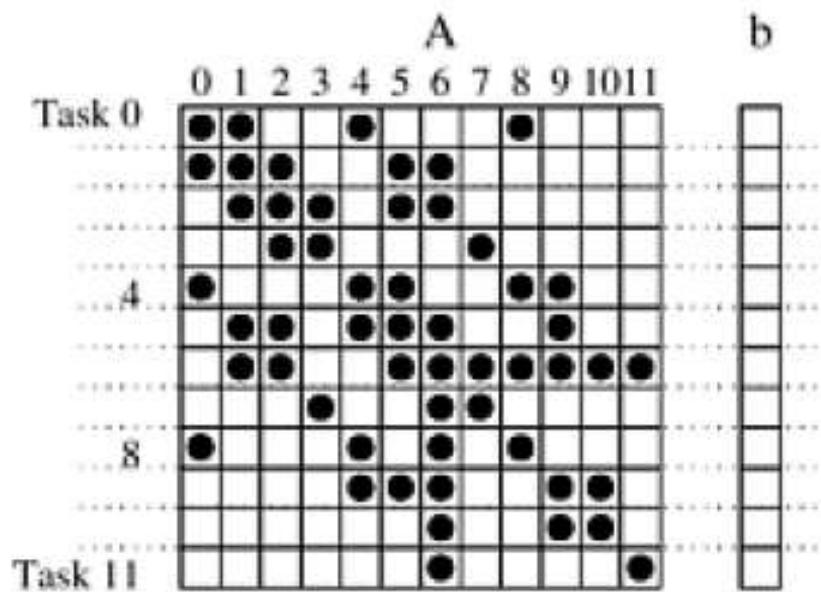
Determined by task decomposition

Used for task mapping on processors

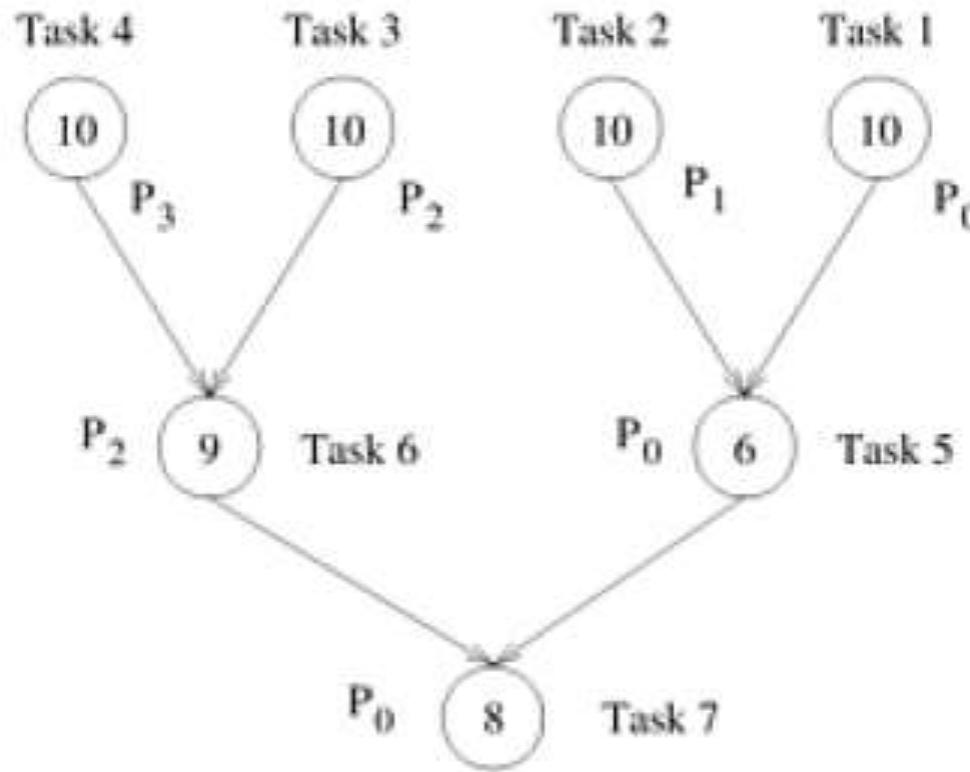
# Example: Task Interaction Graph

Task i computes

$$y[i] = \sum_{j=1}^n (A[i, j] \times b[j])$$



# Map Tasks to Processors



Maximum degree of concurrency is the upper limit of processors

# Data Decomposition Techniques

Recursive decomposition

Data decomposition

Exploratory Decomposition

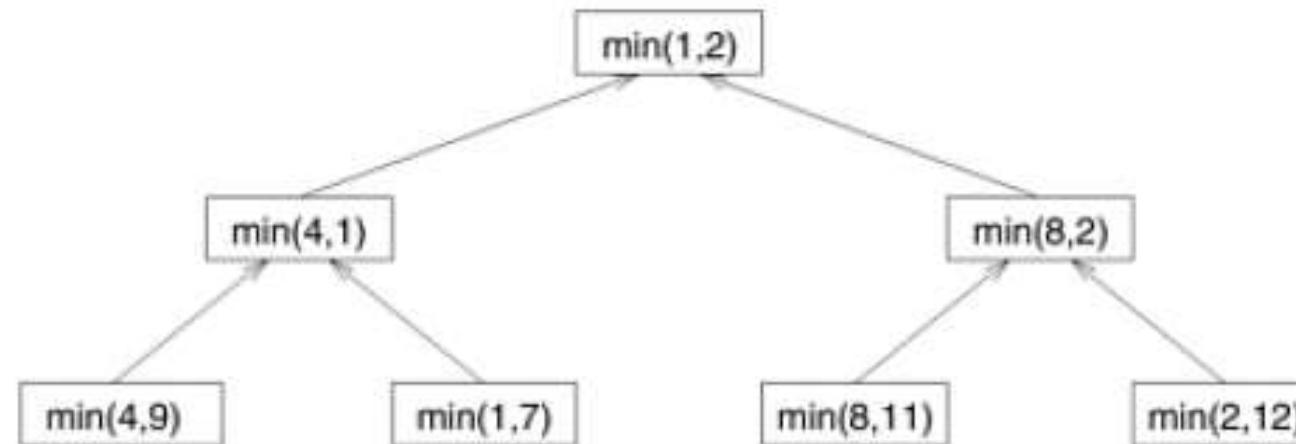
Speculative Decomposition

Hybrid Decompositions

# Example: Recursive Decomposition

Recursive decomposition: solves using divide-and-conquer strategy

Find the minimum element in an array



\* correction: the video slide had a wrong diagram which is fixed here

# Recursive Min Finding

```
1.  procedure RECURSIVE_MIN (A, n)
2.  begin
3.  if (n = 1) then
4.      min := A[0];
5.  else
6.      lmin := RECURSIVE_MIN (A, n/2);
7.      rmin := RECURSIVE_MIN (&(A[n/2]), n - n/2);
8.      if (lmin < rmin) then
9.          min := lmin;
10.     else
11.         min := rmin;
12.     endelse;
13.   endelse;
14.   return min;
15. end RECURSIVE_MIN
```

# Data Decomposition

Data decomposition:

- (i) data is partitioned
- (ii) based on data partitioning computation is decomposed into tasks

Example: matrix multiplication

$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \cdot \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} \rightarrow \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

(a)

$$\text{Task 1: } C_{1,1} = A_{1,1}B_{1,1} + A_{1,2}B_{2,1}$$

$$\text{Task 2: } C_{1,2} = A_{1,1}B_{1,2} + A_{1,2}B_{2,2}$$

$$\text{Task 3: } C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1}$$

$$\text{Task 4: } C_{2,2} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2}$$

# Exploratory decomposition

Parallel search of spaces for solution

Search space is partitioned and then the search each part concurrently

Serial computation may perform better than parallel if the solution is immediate

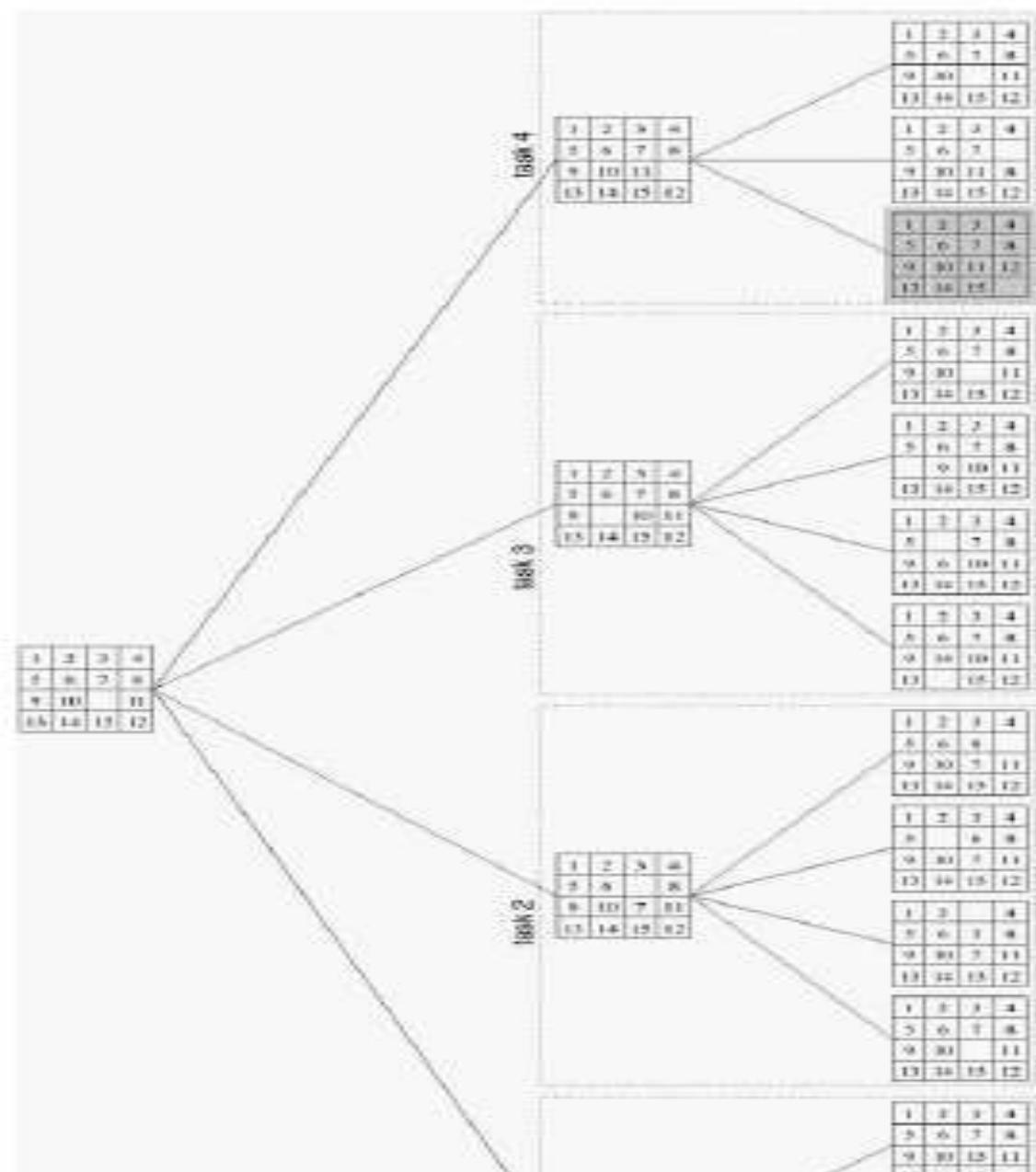
# Example: 15-puzzle

Problem: Find the a sequence of move to reach final configuration from an initial configuration

Serially generate a few level of search configuration

each node is assigned to a task to explore further until at least one of them finds a solution.

if we find a solution then it can inform the others to terminate their searches



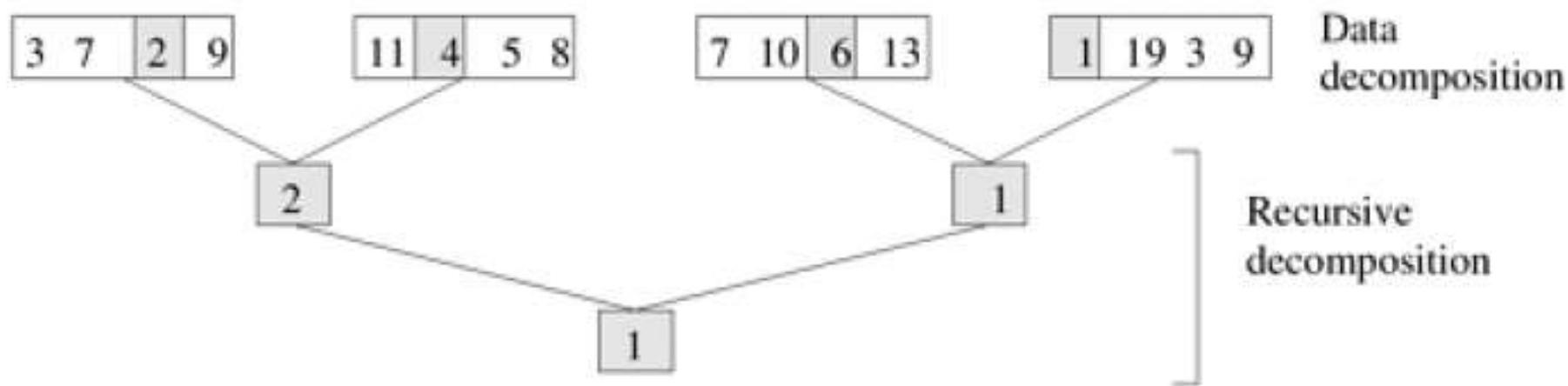
# Speculative Decomposition

Can be performed on `switch` cases.

Parallel execution of (i) switch condition and (ii) switch cases

Wasted computation

# Hybrid Decomposition



# Load Balancing

Goals:

- (1) Balanced task assignment to processors
- (2) Minimal communication between processors

Challenges:

- (1) These two goals may be conflicting e.g.  
assign all tasks on one processor to remove communication cost
- (2) Balanced task partitioning may incur higher communication cost

# Mapping Schemes

Static partitioning: an NP-complete problem in general

- Based on data distribution
  - e.g. block, cyclic, block-cyclic distribution
- Based on static task dependency graph
- Hierarchical

Dynamic

- Task is assigned during execution time
- Task queue

# Parallel Algorithm Models

Data-Parallel Model

Task Model

Task/Work Pool Model

- \* tasks are enqueued in a pool and dequeued for execution

Master-Slave Model

- \* Master process creates the work and assign to the slave processes

Pipeline or Producer-Consumer Model

- \* stream of data is passed on through a succession of processes

# References

Chapter 3

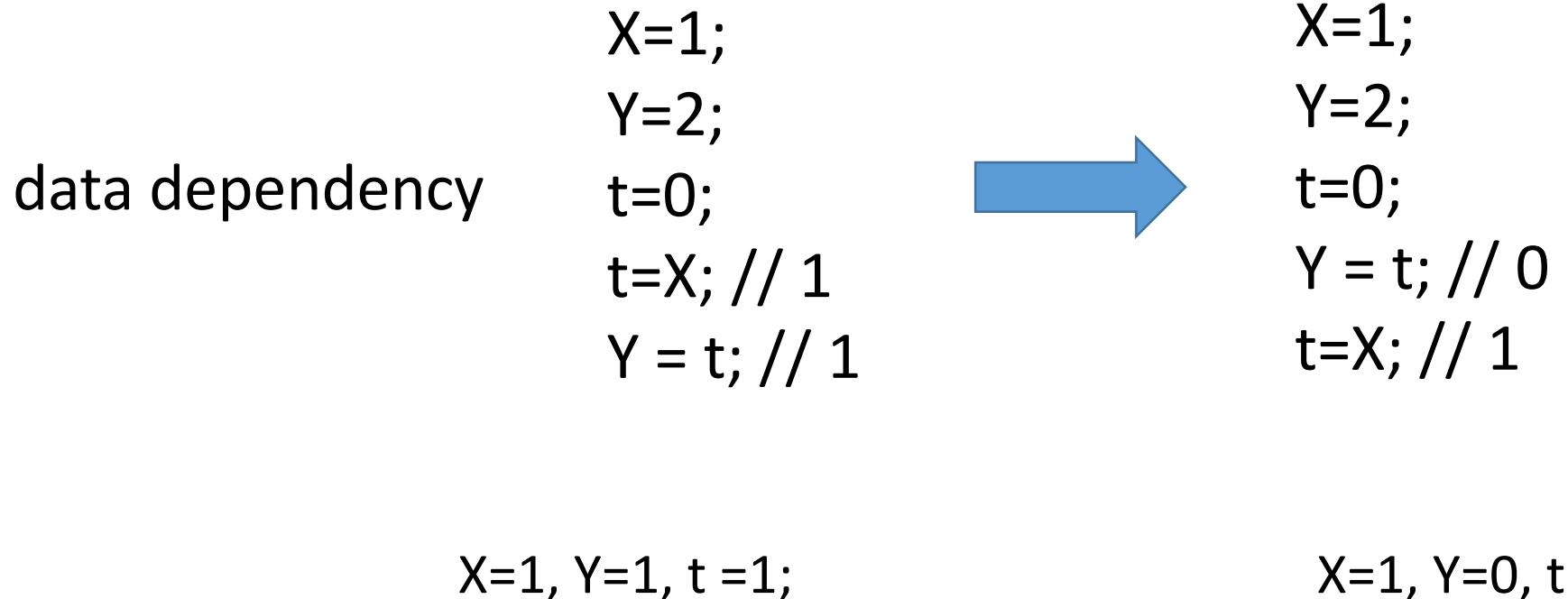
Introduction to Parallel Computing

Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar

# Parallelism in Hardware

# Recap: Program behaviors due to dependencies

architectures preserves dependencies



# Recap: Dependence and Same Location Accesses

a=X;  
X=2;

anti

X=2;  
a=X;

data

X=2;  
X=3;

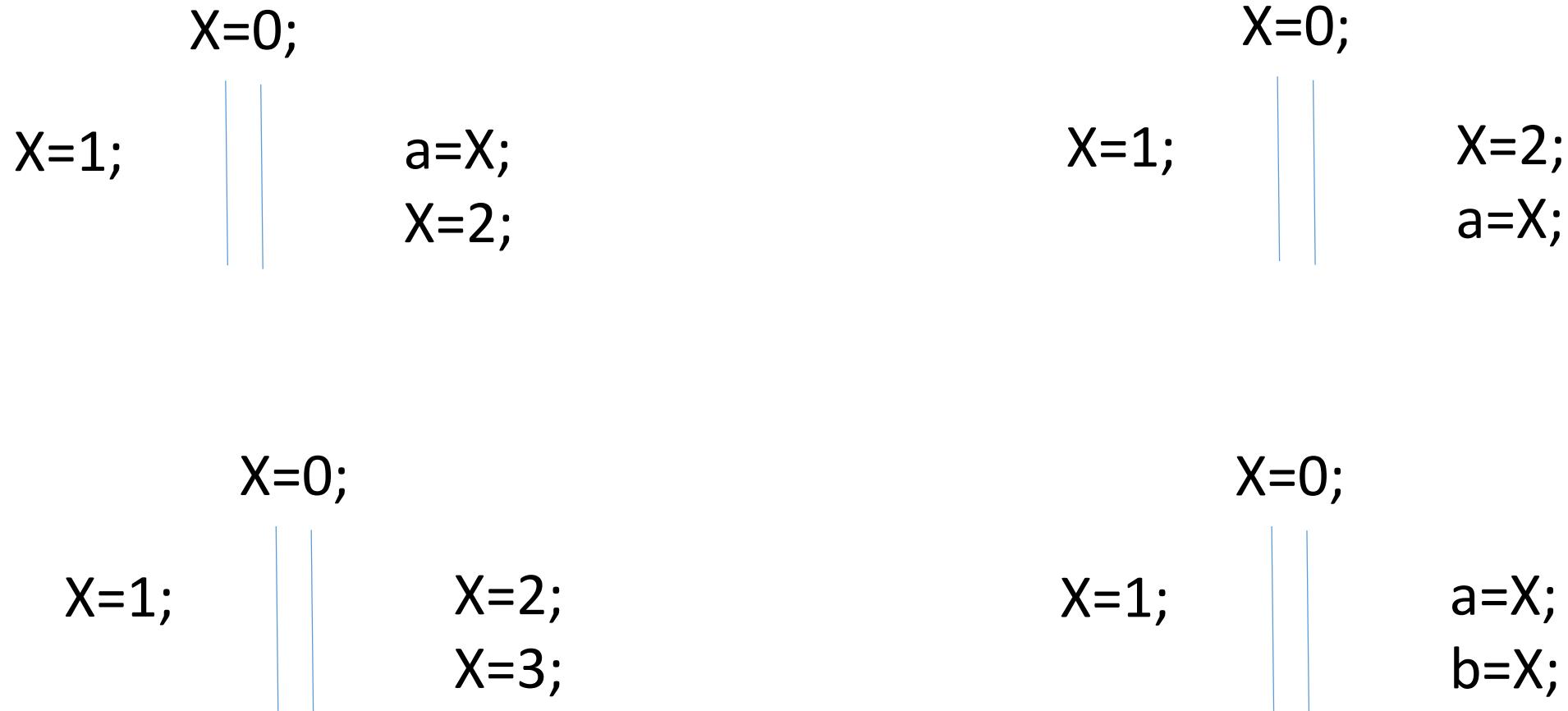
output

a=X;  
b=X;

accessing same  
location

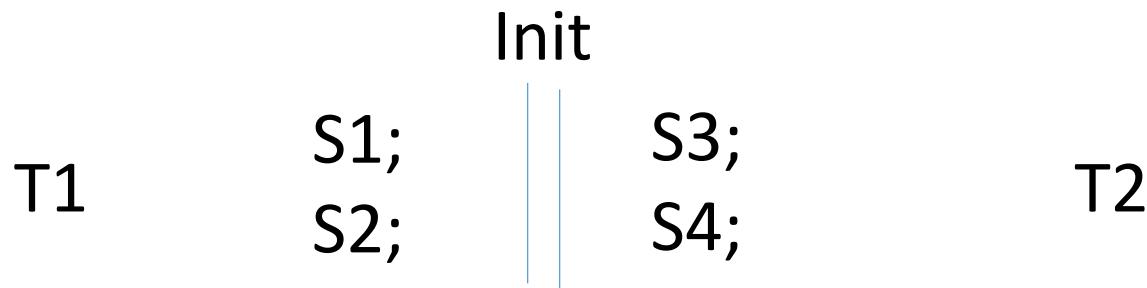
No out of order execution

# Program behaviors in Multithreaded Programs



# Interleaving Execution

- (1) Statements in each thread execute in order.
- (2) Statements from different threads may execute in arbitrary order



Init;S1;S2;S3;S4  
Init;S1;S3;S2;S4  
Init;S1;S3;S4;S2  
Init;S3;S4;S1;S2  
Init;S3;S1;S2;S4  
:

# Coherence Property

X=0;

S1: X=1;



S2: int a=X;     What are the possible values of a?  
S3: X=2;

S1;S2;S3

S2;S1;S3

S2;S3;S1

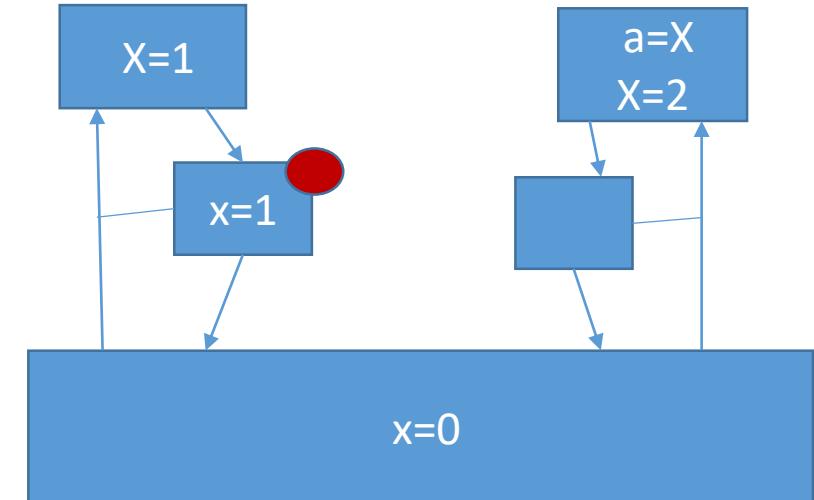
# Coherence Property

X=0;  
S1: X=1;      ||  
  
S1;S2;S3  
S2;S1;S3  
S2;S3;S1

S2: a=X;  
S3: X=2;

What are the possible values of a?

a=1  
a=0  
a=0



# Coherence Property

X=0;

S1: X=1;



S2: a=X;  
S3: X=2;

What are the possible values of a?

S1;S2;S3  
S2;S1;S3  
S2;S3;S1

a=1  
a=0  
a=0

a = 0 ? yes

a = 1 ? yes

a = 2 ? no

# Coherence Property

$x=0$

S1:  $x=1;$

S2:  $x=2;$

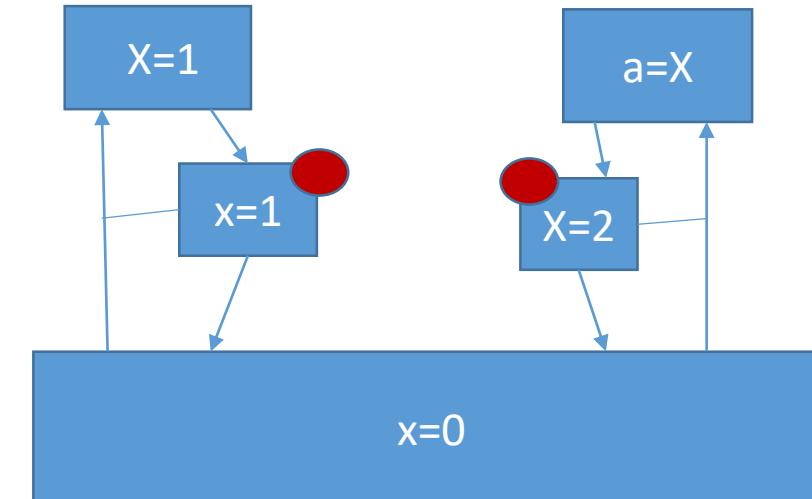
S3:  $a=x;$

What are the possible values of  $a$  in S3?

S1;S2;S3

S2;S1;S3

S2;S3;S1



# Coherence Property

$X=0$

S1:  $X=1;$

S2:  $X=2;$

S3:  $a=X;$

What are the possible values of  $a$  in S3?

S1;S2;S3

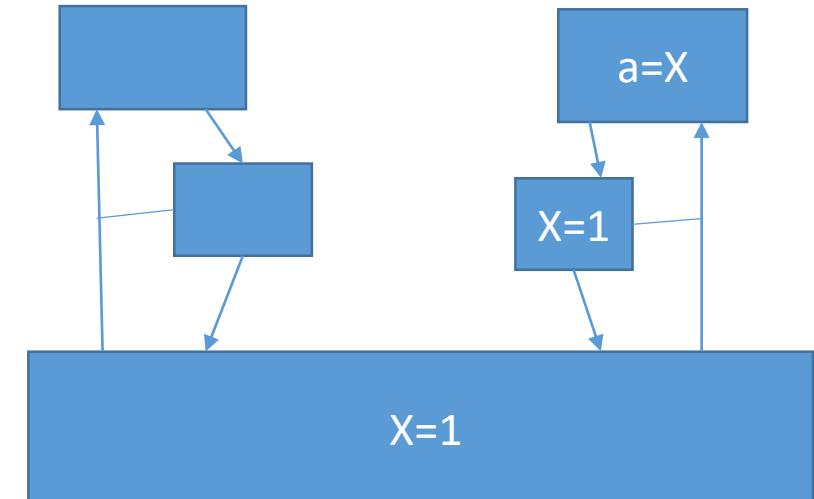
S2;S1;S3

S2;S3;S1

$a=2$

$a=1$

$a=2$



# Coherence Property

X=0

S1: X=1;

S2: X=2;

S3: a=X;

What are the possible values of a in S3?

S1;S2;S3

a=2

a = 0 ? no

S2;S1;S3

a=1

a= 1 ? yes

S2;S3;S1

a=2

a = 2 ? yes

# Coherence Property

X=0

S1: X=1;

S2: X=2;

S3: X=3;

What is the final possible values of X?

S1;S2;S3

S2;S1;S3

S2;S3;S1

# Coherence Property

X=0

S1: X=1;

S2: X=2;

S3: X=3;

S1;S2;S3

X=3

S2;S1;S3

X=3

S2;S3;S1

X=1

# Coherence Property

X=0

S1: X=1;

S2: X=2;

S3: X=3;

S1;S2;S3

X=3

X = 0 ? no

S2;S1;S3

X=3

X = 1 ? yes

S2;S3;S1

X=1

X = 2 ? no

X=3 ? yes

# Coherence Property

X=0

S1: X=1;

S2: a=X;

S3: b=X;

What are the possible values of a and b?

S1;S2;S3

S2;S1;S3

S2;S3;S1

# Coherence Property

$X=0$

S1:  $X=1;$

S2:  $a=X;$

S3:  $b=X;$

What are the possible values of a and b?

$S1;S2;S3$

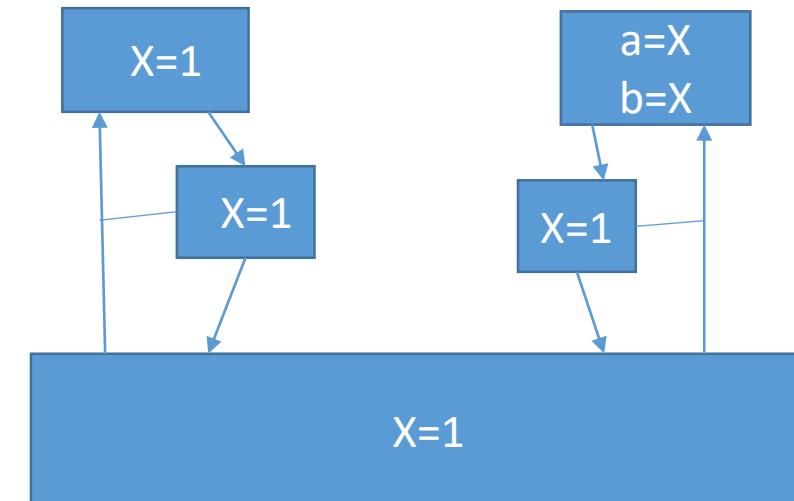
$S2;S1;S3$

$S2;S3;S1$

$a=1, b=1, X=1$

$a=0, b=1, X=1$

$a=0, b=0, X=1$



# Coherence Property

X=0

S1: X=1;

S2: a=X;

S3: b=X;

What are the possible values of a and b?

S1;S2;S3

a=1, b=1

a=1, b=0? no (does not read a stale value)

S2;S1;S3

a=0, b=1

X = 0 ? no

S2;S3;S1

a=0, b=0

X=1;

X=2;

a=X; // 1 not possible

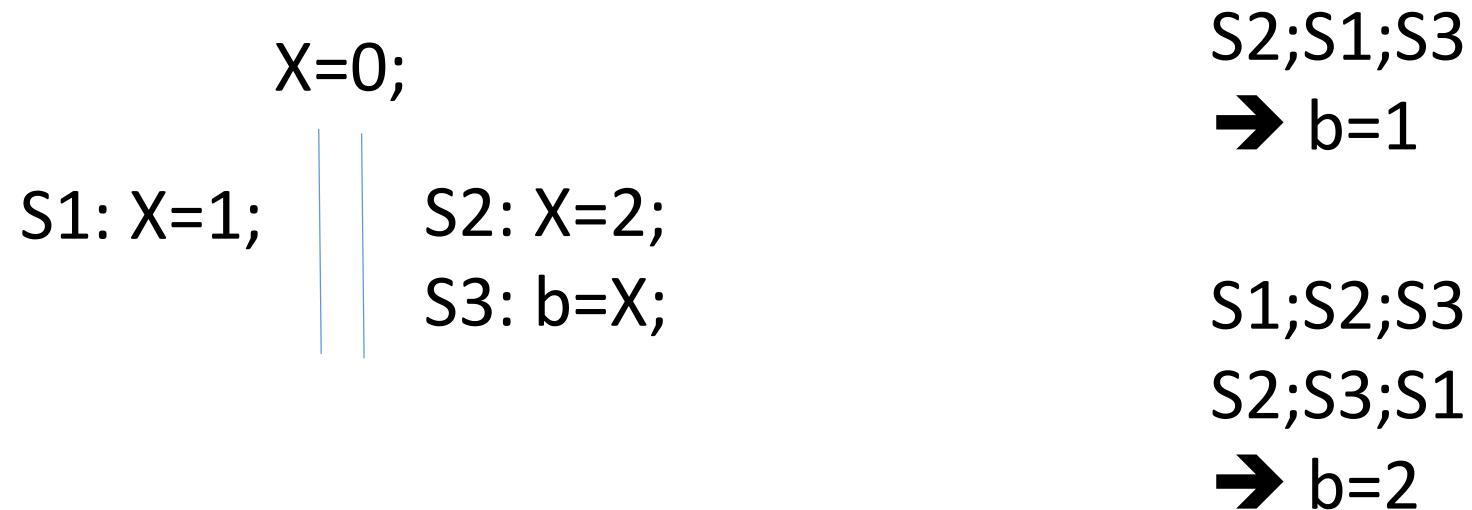
# Parallelism in Hardware

# Recap

Coherence examples

# Recap

## Coherence examples



b = 1 or b = 2 is possible

# Coherence Property

X=0;

S1: X=1;

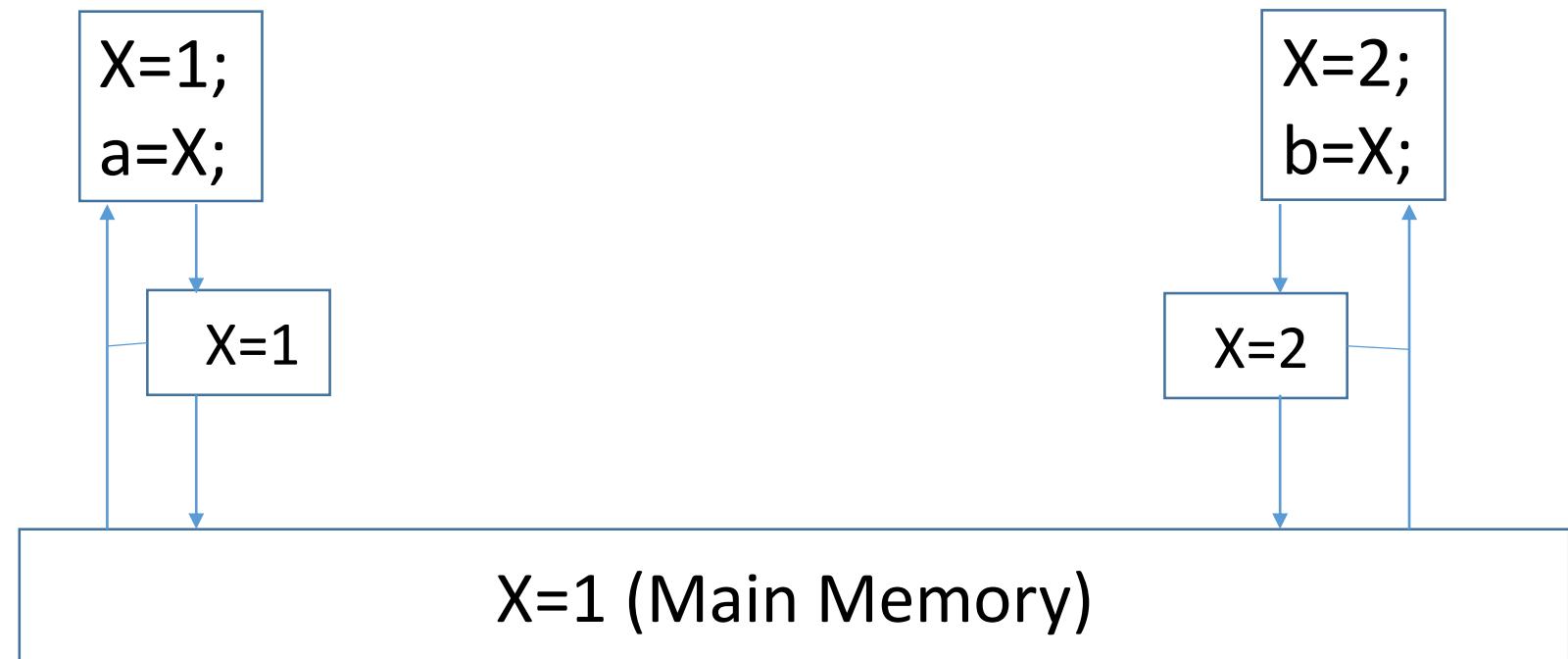
S2: a=X;

S3: X=2;

S4: b=X;

What are the possible values of a and b?

- S1;S2;S3;S4 → a=1, b=2
- S1;S3;S4;S2 → a=2, b=2
- S1;S3;S2;S4 → a=2, b=2
- S3;S4;S1;S2 → a=1, b=2
- S3;S1;S2;S4 → a=1, b=1
- S3;S1;S4;S2 → a=1, b=1



# Cache Coherence

X=0

X=1;   ||    X=2;  
a=X;   ||    b=X;

- ✓ a=1;b=1;
- ✓ a=2,b=2;
- ✓ a=1;b=2;
- ✗ a=2, b=1 is not possible due to cache coherence

S1;S2;S3;S4 → a=1,b=2  
S1;S3;S4;S2 → a=2,b=2  
S1;S3;S2;S4 → a=2,b=2  
S3;S4;S1;S2 → a=1, b=2  
S3;S1;S2;S4 → a=1, b=1  
S3;S1;S4;S2 → a=1, b=1

# Cache Coherence

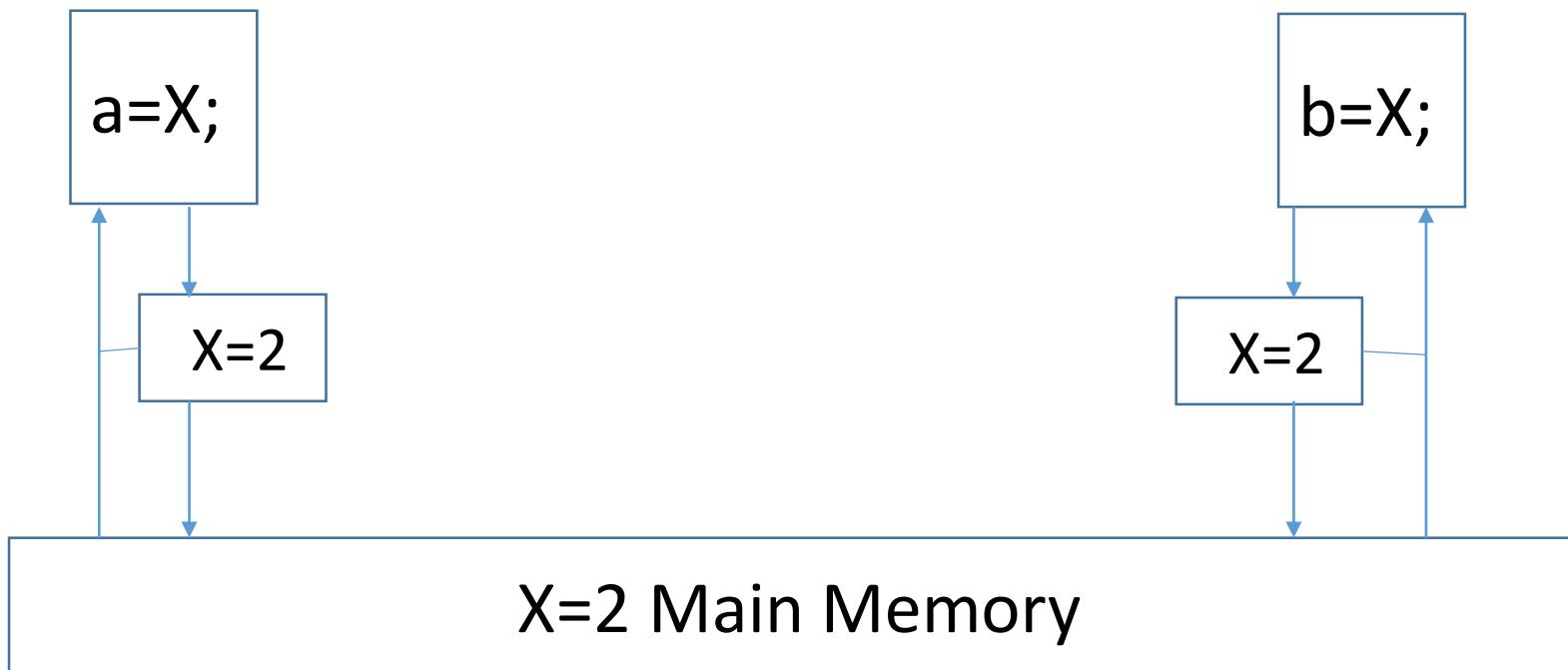
S1: X=1;  
S2: a=X;

S3: X=2;  
S4: b=X;

a=2, b=1 ?

b=1 → S3;S1;S4;S2  
a=2 X

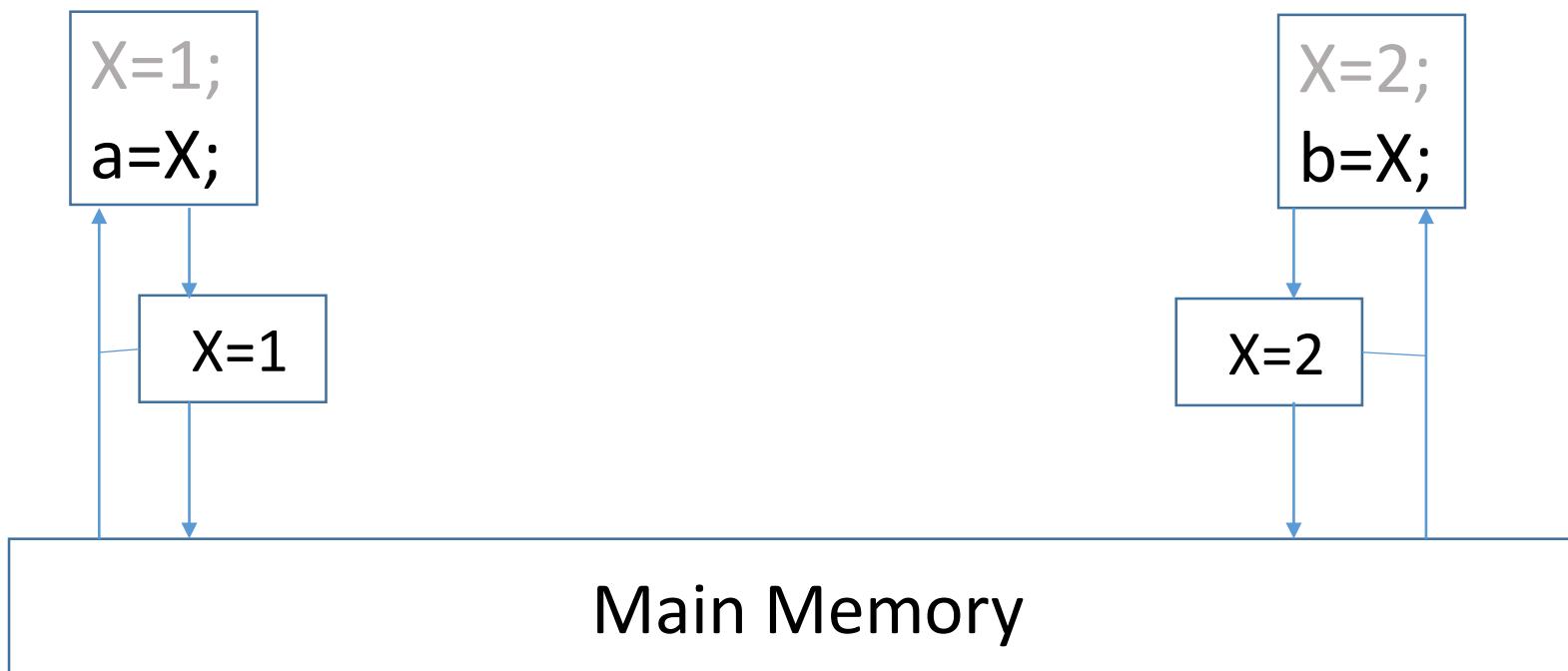
What is the possible values of a and b?



# Cache Coherence

X=1;   ||    X=2;  
a=X;   ||    b=X;

What is the possible values of a and b?



# Cache Coherence Protocol

Keeps the cache updates coherent

Important for multithreaded programs executing on multiple cores

# Snooping Based Cache Coherence Protocol

When the cores share a bus, any signal transmitted on the bus can be observed by all the cores connected to the bus

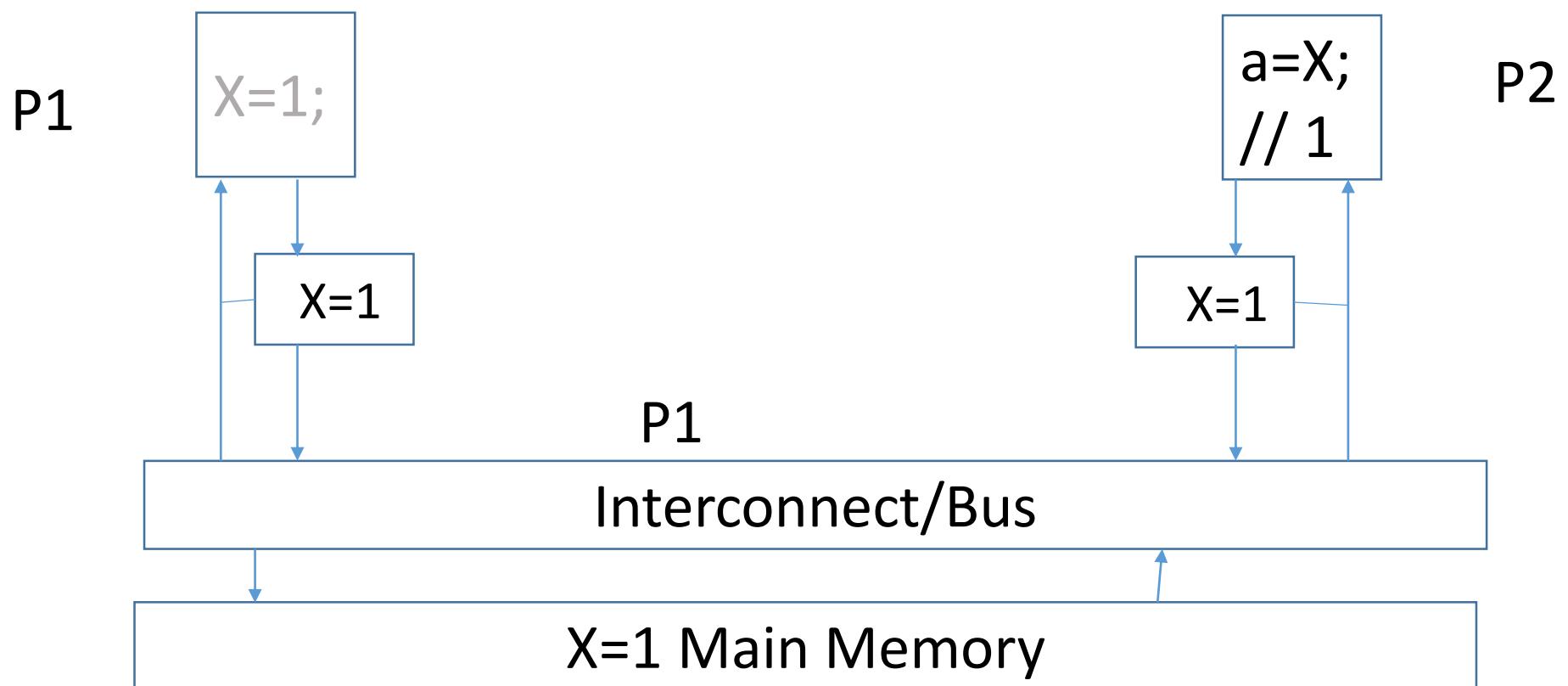
- core 0 updates the copy of x stored in its cache,
- Core 0 broadcasts this information(cache line that contains x is updated) across the bus
- core 1 is snooping the bus,
- Sees that x (or the cache line that contains x) has been updated
- It can mark its copy of x as invalid.

snooping works with both write-through and write-back caches

# Write Invalidate

When a cache block is written:

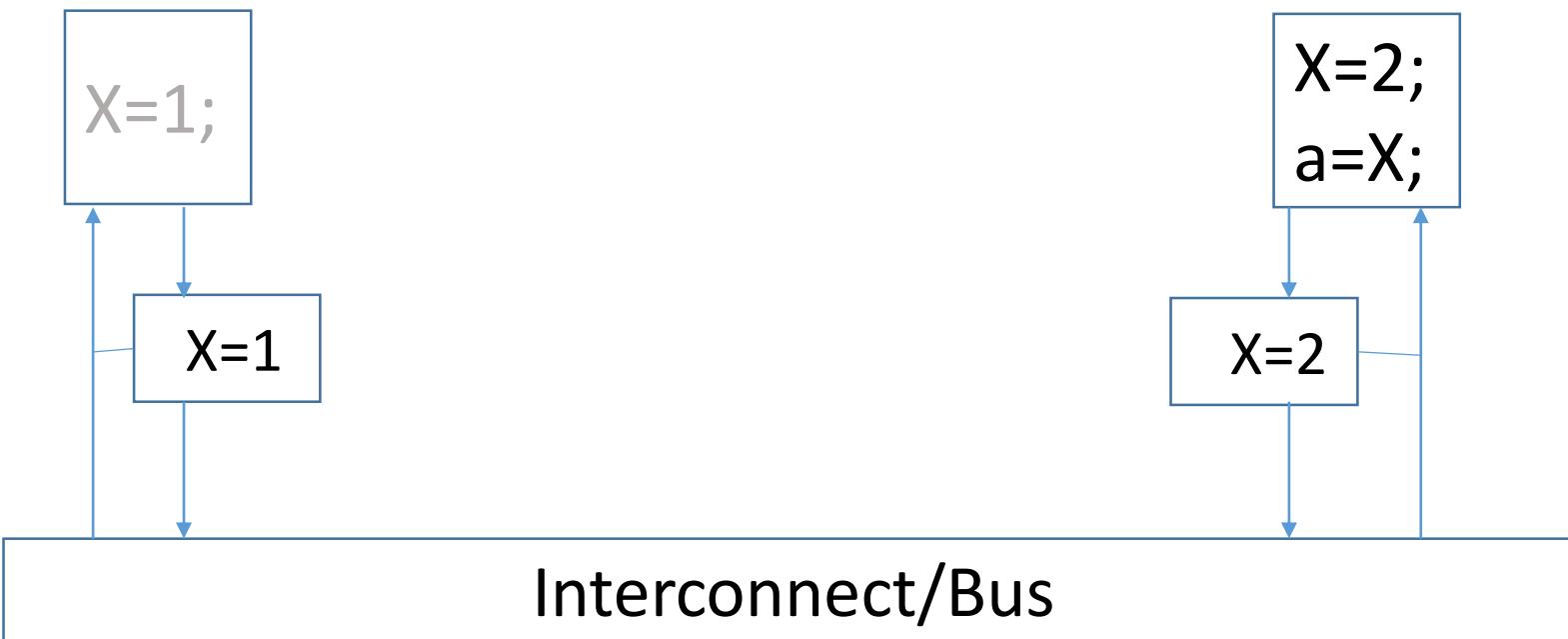
- all the shared copies in the other caches are invalidated through bus snooping.
- only one data copy can be exclusively read and written by a processor.
- All the other copies in other caches are invalidated.



# Write Update

When a cache block is written:

- all the shared copies of the other caches are updated through bus snooping.
- This method broadcasts a write data to all caches throughout a bus.
- It incurs larger bus traffic than write-invalidate protocol.



# Drawback of Snooping Based Protocol

- Broadcasts are expensive
- Snooping protocol broadcasts every time a variable is updated
- Not scalable, in larger systems due to performance

# Directory-based cache coherence Protocol

The distributed directory stores the status of each cache line.

Each core/memory pair stores the status of the cache lines in its local memory.

A read operation updates the directory entry corresponding to that line with the information about the core

When a variable is updated, the directory is consulted, and the cache controllers of the cores that have that variable's cache line in their caches are invalidated.

Pros. only the cores storing that variable need to be contacted.

$X=Y; \parallel Y=Z; \parallel Z=X;$  // T1{X,Y}, T2{Y,Z}, T3{Z,X}

Cons. Substantial additional storage required for the directory

# False Sharing

CPU caches operate on cache lines, not individual variables

Update on one variable →

mark the cache line as *dirty*, other cores fetch the data from the main memory

```
int i, j, m, n;
```

```
...
```

```
double y[m];
```

core\_count=2

```
/* Assign y = 0 */
```

m=8

```
...
```

```
for (i = 0; i < m; i++)
```

doubles are 8 bytes

```
    for (j = 0; j < n; j++)
```

cache line 64 bytes

```
        y[i] += f(i,j);
```

# False Sharing

```
int i, j, iter_count;  
/* Shared variables initialized by one core */  
int m, n, core_count; double y[m];  
iter_count = m/core_count;
```

core\_count=2  
m=8  
doubles are 8 bytes  
cache line 64 bytes

```
/* Core 0 */  
for (i = 0; i < iter_count; i++)  
    for (j = 0; j < n; j++)  
        y[i] += f(i,j);
```

```
/* Core 1 */  
for (i = iter_count+1; i < 2*iter_count; i++)  
    for (j = 0; j < n; j++)  
        y[i] += f(i,j);
```

Y[0], Y[1], ..., Y[7]

Y[0], Y[1], ..., Y[7]



# References

## Chapter 2

- An Introduction to Parallel Programming  
by Peter Pacheco.

## Chapter 5.2 (Details on cache coherence protocols)

- Computer Architecture, Sixth Edition A Quantitative Approach  
by John L. Hennessy, David A. Patterson

# Beyond For-Loop Parallelism in OpenMP

- Concurrent/multithreaded programming
- Task parallelism
- SIMD parallelism
- Pipeline parallelism/ ILP

...

# C++ Multithreading

```
void function_1();
void function_2();
:
std::thread thread_1(function_1); // thread_1 executes function_1()
std::thread thread_2(function_2); // thread_2 executes function_2()

thread_1.join(); // thread_1 is completed
thread_2.join(); // thread_2 is completed

thread_1 and thread_2 execute concurrently
```

# OpenMP Multithreaded Execution

Multiple `section` in a `sections` execute concurrently

```
#pragma omp parallel { // creates a set of threads
    #pragma omp sections { // distributes multiple section between existing threads.
        #pragma omp section { // structured block 1 }
        #pragma omp section { // structured block 2 }
        #pragma omp section { // structured block 3 }
        ...
    }
}
```

# OpenMP Multithreaded Execution

Multiple `section' in a `sections' execute concurrently

- if #section > #threads, some of the threads will execute multiple section
- if #section < #threads, some of the threads will be idle.
- No order of execution of threads
- Implicit barrier at the end of a `sections' construct unless a `nowait' clause is specified.

# Example: Mergesort

```
void mergesort_serial(int a[], int n, int temp[]) {  
    mergesort_serial(a, n/2, temp);  
    mergesort_serial(a + n/2, n - n/2, temp);  
    merge(a, n, temp);  
}
```

1. Divide the unsorted list into 2 halves.
2. Call mergesort recursively on each of the halves and then merge each of the sorted halves.

# Parallel Mergesort

```
int threads = omp_get_thread_num();
```

```
...
```

```
void mergesort_parallel(int a[], int n, int temp[], int threads){
```

```
    if (threads == 1) { mergesort_serial(a, n, temp);}
```

```
    else if (threads > 1) {
```

```
        #pragma omp parallel sections
```

```
{
```

```
        #pragma omp section
```

```
            mergesort_parallel(a, n/2,temp,threads/2);
```

```
        #pragma omp section
```

```
            mergesort_parallel(a + n/2, n - n/2, temp + n/2, threads - threads/2);
```

```
}
```

```
        merge(a, size, temp);
```

```
} // end sections
```

```
}
```

Ref: <https://github.com/serendipity/parallel-j/blob/master/OpenMP/mergeSort-omp.c>

# Atomic Accesses

Shared memory accesses on a variable results in data race if at least once access is a write.

Mutual exclusion is too costly.

The atomic construct ensures that a specific location is accessed atomically

Read: atomically reads a value of a location

Write: atomically writes a value on a location

Update: atomically reads and writes a value of a location

X=0

int a=<sub>atomic</sub> X;    ||    X =<sub>atomic</sub> 1        a = {0, 1}

# Example

```
X=0;  
X++;    ||    X++;
```

The final value: X=2

# Example

X=0;

S1: int a=X; // 0  
S2: a=a+1; //1  
S3: X=a; // 1

S4: int b=X; // 0  
S5: b=b+1; // 1  
S6: X=b; // 1

Possible interleaving S1;S4;S2;S5;S3;S6  
The final value: X=1     *Incorrect !*

# Example

X=0;

S1: int a=\_atomic X;  
S2: a=a+1;  
S3: X=\_atomic t;

S4: int b=\_atomic X;  
S5: b=b+1;  
S6: X=\_atomic b;

No data race.

Possible interleaving S1;S4;S2;S5;S3;S6

The final value: X=1      *Incorrect !*

Solution:  
Use atomic update.

X=0;  
X++;      |      X++;

The final value: X=2

# Memory Orders

seq\_cst  
acq\_rel  
release  
acquire  
relaxed

Write Release:

seq\_cst  
acq\_rel  
release

The operations before a write release must be completed before performing the write release operation

Read Acquire:

seq\_cst  
acq\_rel  
acquire

The operations after a read acquire is performed after the read acquire is completed

relaxed accesses provides no restriction in ordering

# References

- <https://www.openmp.org/spec-html/5.1/openmp.html>

# Memory Consistency

Defines the possible result/outcome of a program

Defines the ordering of execution of shared memory accesses

Captures the effect of out-of-order executions and the effects of caches in a processor

Different consistency models:

- Sequential consistency (SC)
- Total-store-order (TSO)
- Partial store order (PSO)
- Relaxed memory order (RMO)
- :

# Sequential Consistency

"... the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program."

Leslie Lamport,  
"How to Make a Multiprocessor Computer That Correctly Executes  
Multiprocess Programs",  
IEEE Trans. Comput. C-28,9 (Sept. 1979), 690-691.

# *How does concurrent programs execute?*

*“...the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.”* ~ Leslie Lamport

Sequential consistency (SC) ==> Interleaving Execution

# Representative Example: Store Buffer (SB)

X=Y=0

X = 1;

a = Y;

Y = 1;

b = X;

# Program: Store Buffer (SB)

X=Y=0

S1: X = 1;

S2: a = Y;

S3: Y = 1;

S4: b = X;

**Outcome:**

a=0, b=1 (S1;S2;S3;S4)

# Program: Store Buffer (SB)

X=Y=0

S1: X = 1;

S2: a = Y;

S3: Y = 1;

S4: b = X;

**Outcome:**

a=0, b=1 (S1;S2;S3;S4)

a=1, b=1 (S1;S3;S2:S4 , S3;S1;S4;S2, ...)

# Program: Store Buffer (SB)

X=Y=0

S1:X = 1;

S2:a = Y;

S3:Y = 1;

S4:b = X;

**Outcome:**

a=0, b=1 (S1;S2;S3;S4)

a=1, b=1 (S1;S3;S2:S4 , S3;S1;S4;S2, ...)

a=1, b=0 (S3;S4;S1;S2)

a=0, b=0 **X**

# Program: Store Buffer (SB)

X=Y=0

X = 1;

a = Y;

Y = 1;

b = X;

*Let's execute the program...*

# Program: Store Buffer (SB)

X=Y=0

X = 1;

a = Y;

Y = 1;

b = X;

## Outcome:

a=0, b=1

a=1, b=1

a=1, b=0

a=0, b=0

# Program: Store Buffer (SB)

X=Y=0

X = 1;

a = Y;

Y = 1;

b = X;

a=0, b=0

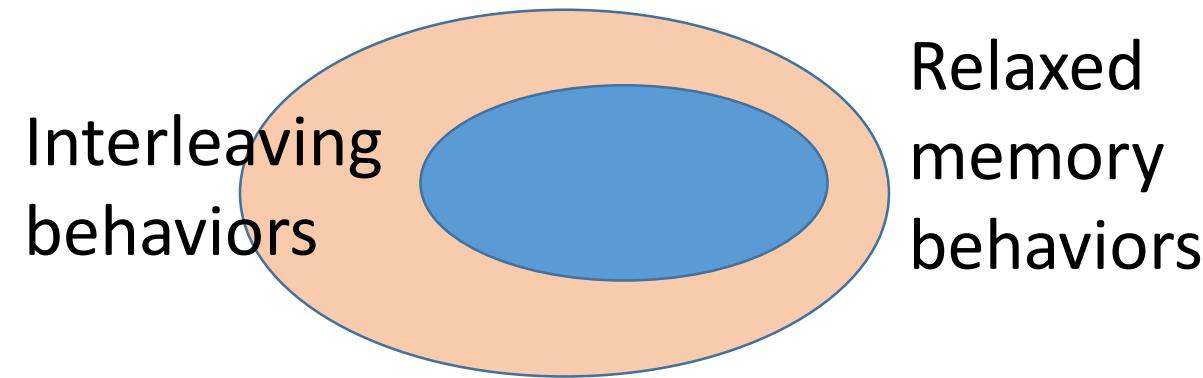
Cannot be explained by thread interleaving.

*Relaxed memory model/consistency/concurrency*

# Relaxed Memory Concurrency

Traditionally: Concurrency = thread interleaving

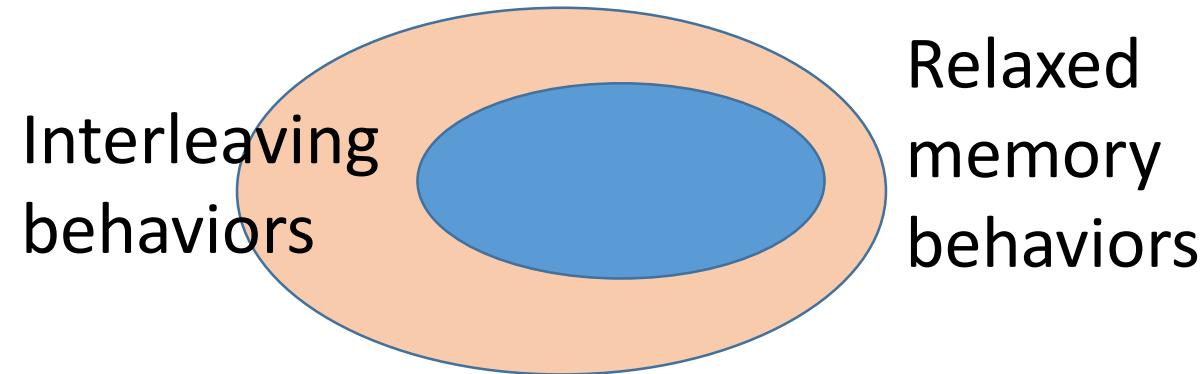
Reality: more behaviors than thread interleaving



# Relaxed Memory Concurrency

Traditionally: Concurrency = thread interleaving

Reality: more behaviors than thread interleaving



## Reasons:

**Compiler:** reorder instructions

**Hardware:**

- Out of order execution
- Data movement/communication is not instantaneous

# Order relaxation

No order for Store – Load

$$X=Y=0$$

$X = 1;$

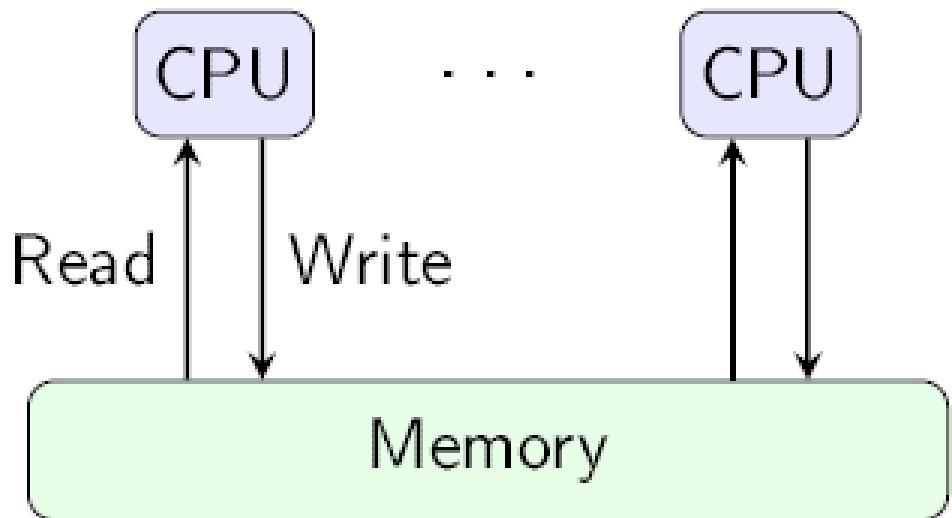
$a = Y;$

$Y = 1;$

$b = X;$

$a=0, b=0$

# Memory Models in Processors

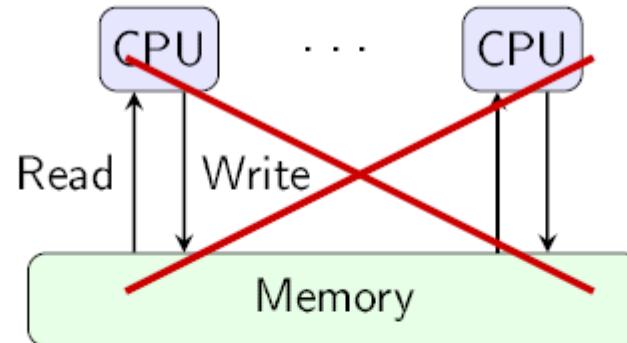


**Sequential Consistency (SC)**

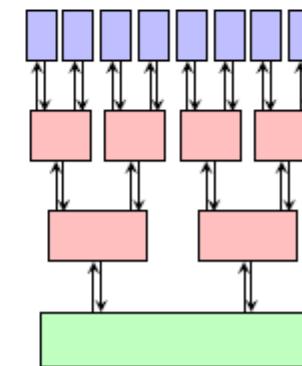
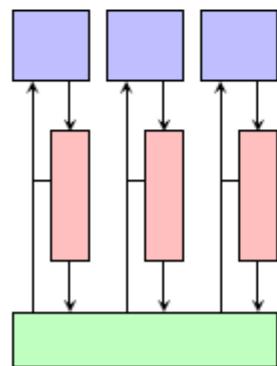
Initially  $X = Y = 0$ ;  
 $X = 1$ ; ||  $Y = 1$ ;  
 $a = Y$ ; ||  $b = X$ ;

- $a = 1, b = 1, \quad X = 1, Y = 1 \quad \checkmark$
- $a = 0, b = 1, \quad X = 1, Y = 1 \quad \checkmark$
- $a = 1, b = 0, \quad X = 1, Y = 1 \quad \checkmark$
- $a = 0, b = 0, \quad X = 1, Y = 1 \quad \text{X}$

# Effect of Caches

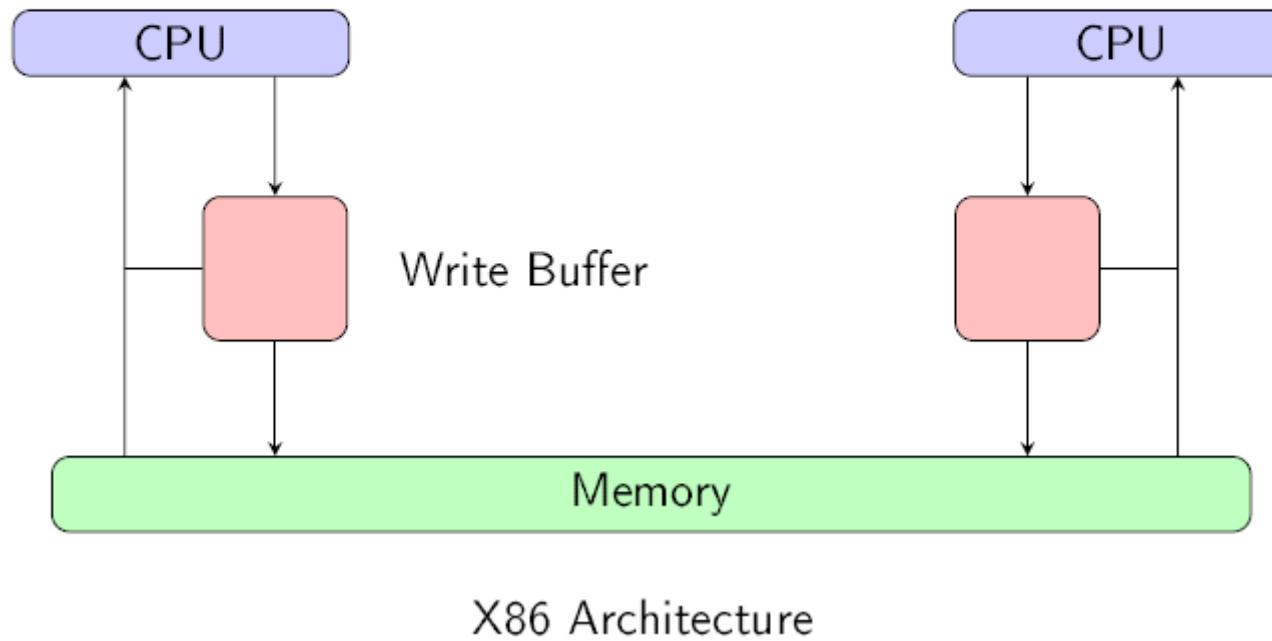


**Sequential Consistency (SC)**  
Relaxed Memory Consistency



# x86 TSO Architecture

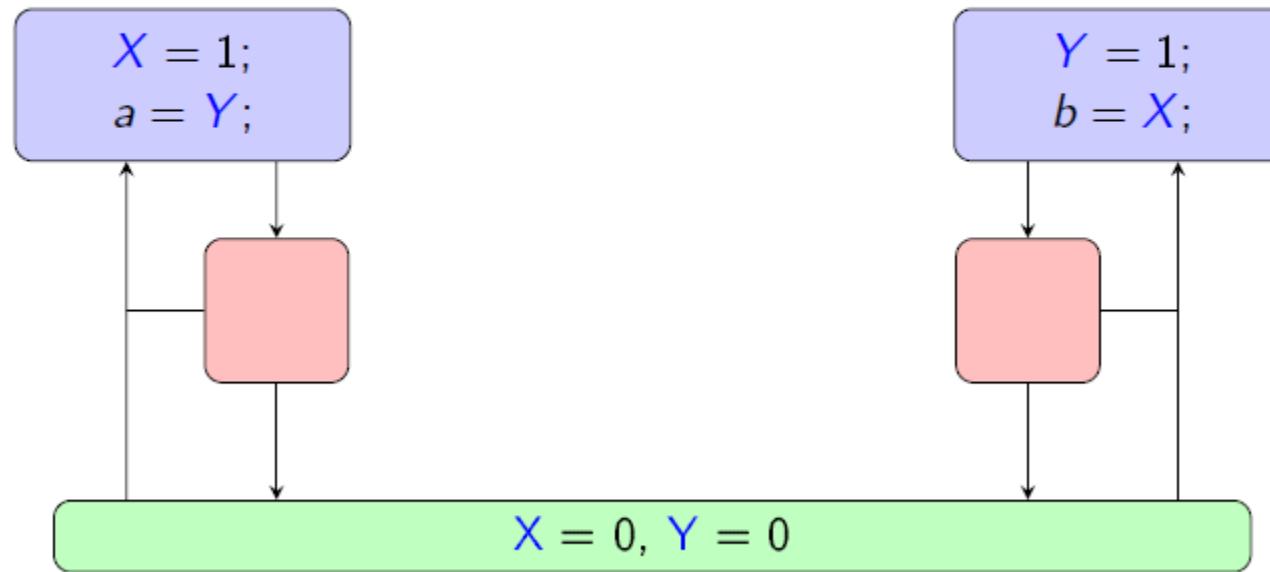
Initially  $X = Y = 0$ ;  
 $X = 1$ ; ||  $Y = 1$ ;  
 $a = Y$ ; ||  $b = X$ ;



# x86 TSO Architecture

Initially  $X = Y = 0$ ;

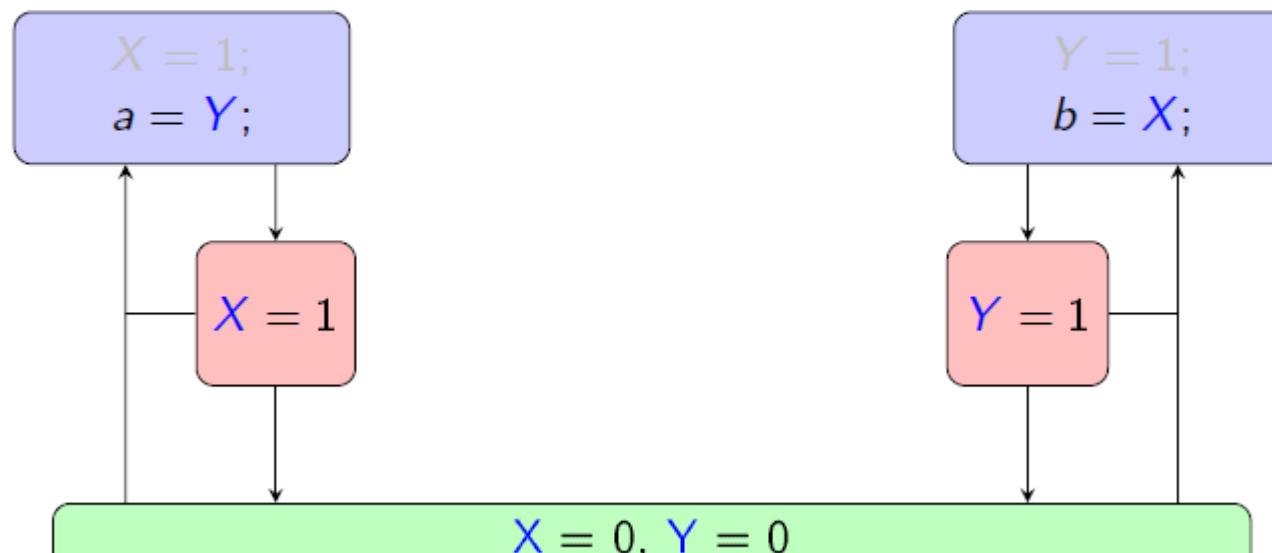
$X = 1$ ; ||  $Y = 1$ ;  
 $a = Y$ ; ||  $b = X$ ;



X86 Architecture

# x86 TSO Architecture

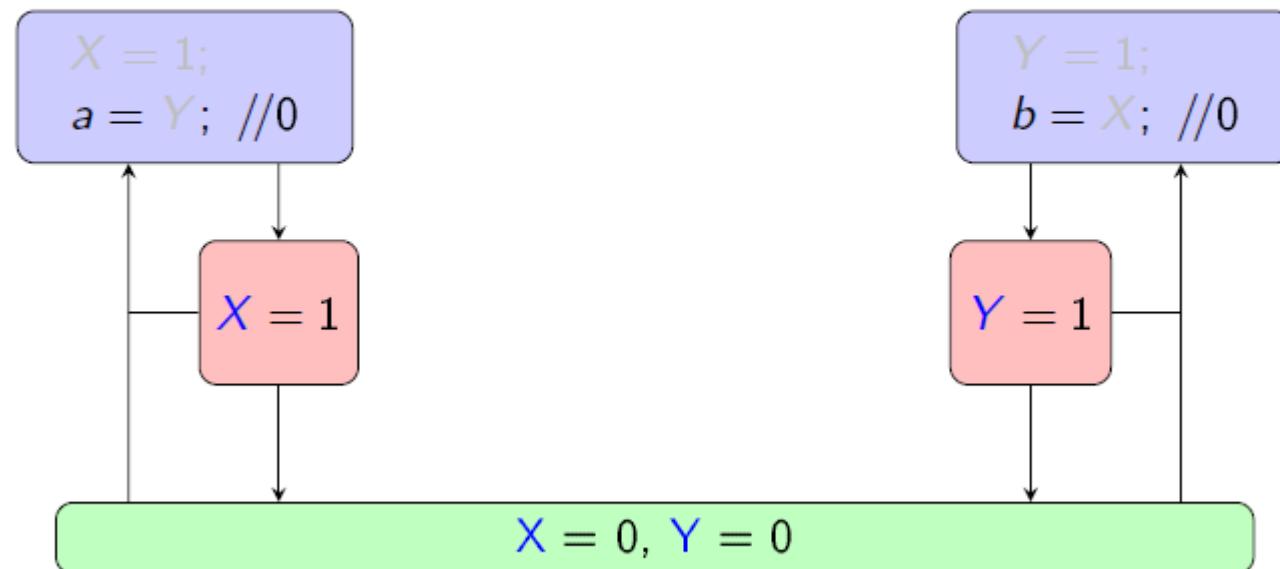
Initially  $X = Y = 0$ ;  
 $X = 1$ ; ||  $Y = 1$ ;  
 $a = Y$ ; ||  $b = X$ ;



X86 Architecture

# x86 TSO Architecture

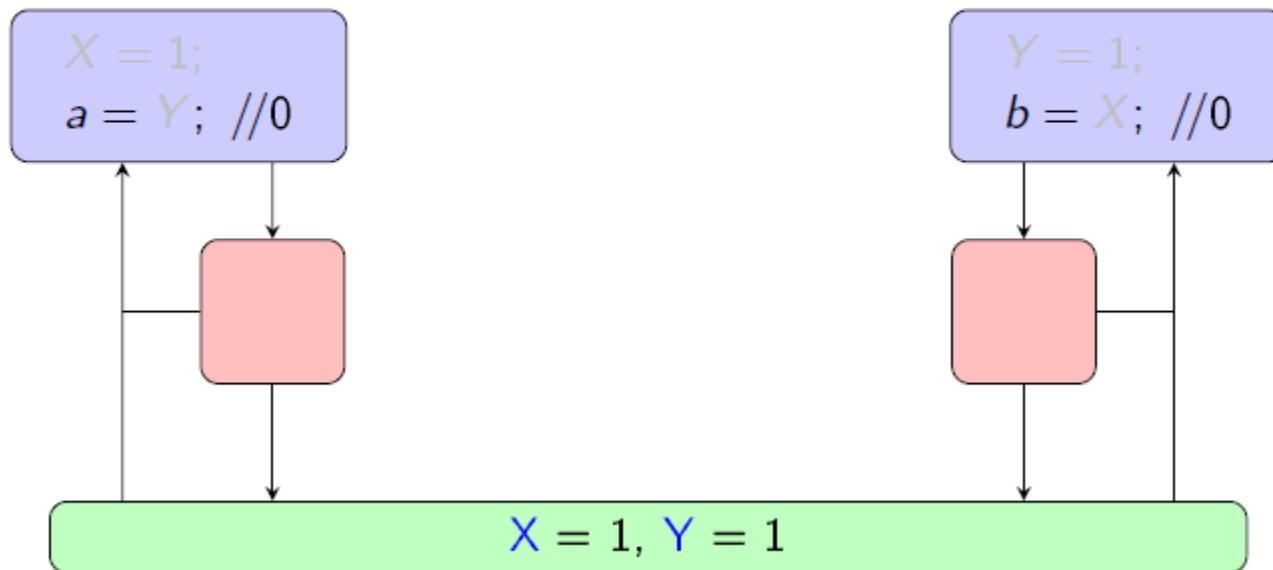
Initially  $X = Y = 0$ ;  
 $X = 1; \parallel Y = 1;$   
 $a = Y; \parallel b = X;$



X86 Architecture

# x86 TSO Architecture

Initially  $X = Y = 0$ ;  
 $X = 1; \parallel Y = 1;$   
 $a = Y; \parallel b = X;$



X86 Architecture

# x86 TSO Architecture

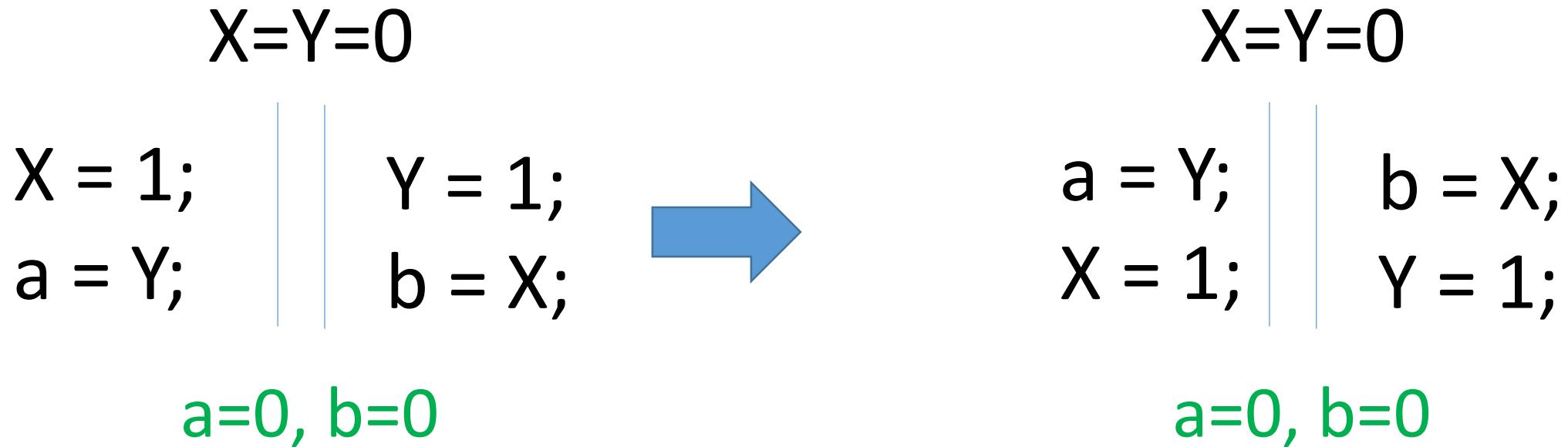
Initially  $X = Y = 0$ ;  
 $X = 1; \parallel Y = 1;$   
 $a = Y; \parallel b = X;$

$a = 1, b = 1, X = 1, Y = 1 \checkmark$   
 $a = 0, b = 1, X = 1, Y = 1 \checkmark$   
 $a = 1, b = 0, X = 1, Y = 1 \checkmark$   
 $a = 0, b = 0, X = 1, Y = 1 \checkmark$

X86 Architecture

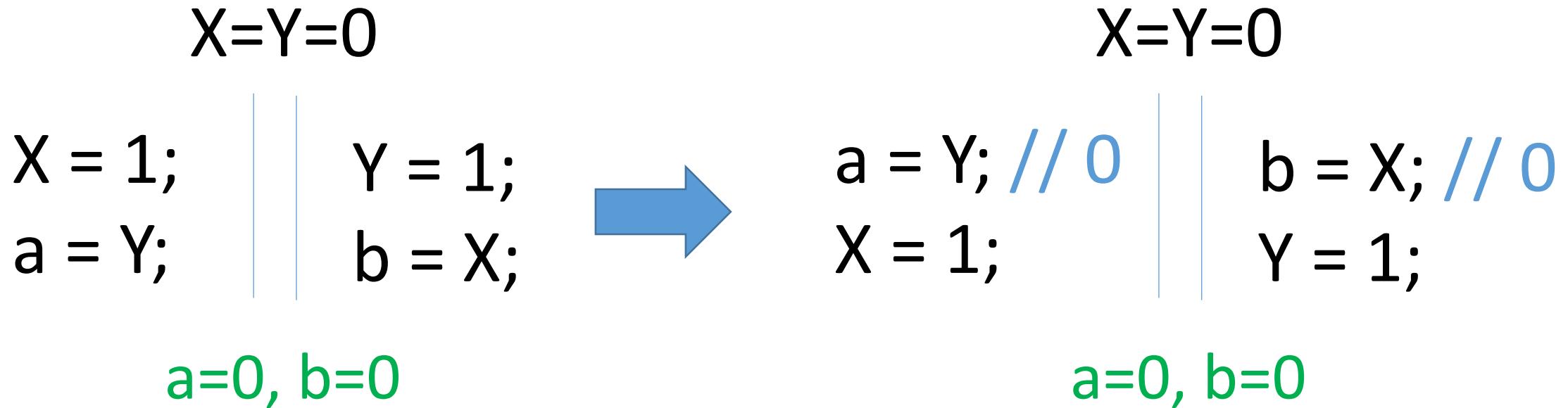
# Order relaxation

No order for Store – Load



# Order relaxation

No order for Store – Load



*Is it allowed by any relaxed memory model?*

# References

## Chapter 5.6 (Memory consistency Models)

- Computer Architecture, Sixth Edition A Quantitative Approach  
by John L. Hennessy, David A. Patterson

# Memory Consistency

Sequential consistency

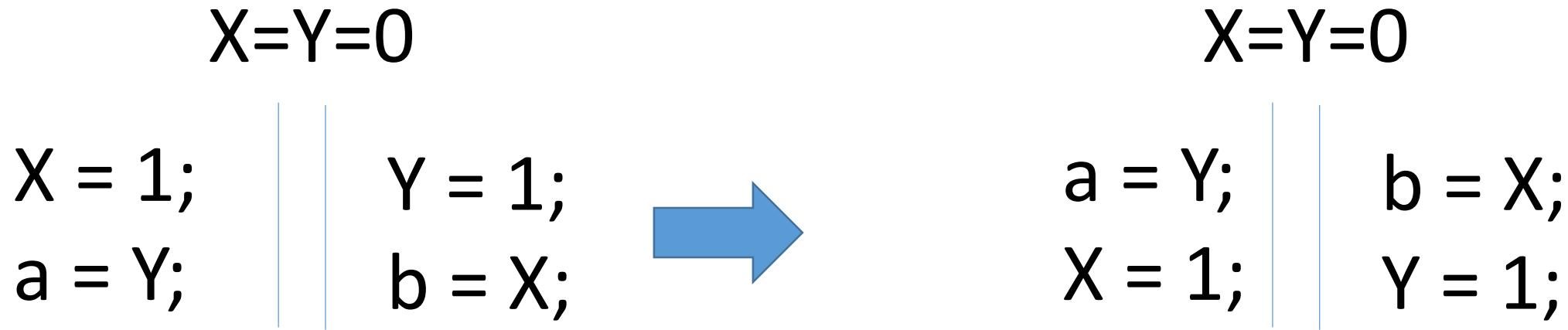
Relaxed consistency

Compiler reordering

Effects of cache

# Store Buffer(SB)

No order for Store – Load



# Total Store Order (TSO)

Relaxation for Store – Load

<b>a;b</b>	<b>Store</b>	<b>Load</b>
<b>Store</b>	<b>N</b>	<b>Y</b>
<b>Load</b>	<b>N</b>	<b>N</b>

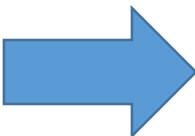
Behavior = transformations+interleaving

# Store Buffer(SB)

X=Y=Z=0

X = 1;  
t=Z;  
a = Y;

Y = 1;  
Z=1;  
b = X;



X=Y=Z=0

t=Z;  
X = 1;  
a = Y;

Y = 1;  
b = X;  
Z=1;



X=Y=Z=0

a=b=0

t=Z; //0  
a = Y; // 0  
X = 1;

b = X; // 0  
Y = 1;  
Z=1;

# Another Program: Message Passing (MP)

X=Y=0

S1: X = 1;

S2: Y = 1;

S3: a = Y;

S4: if(a == 1)

S5: b = X;

*Is a=1, b=0 possible an allowed behavior?*

- SC ?
- TSO ?

S3;S4;S1;S2 → a=0, b= ...

S1;S2;S3;S4;S5 → a=1, b=1

...

# Another Program: Message Passing (MP)

X=Y=0

S1: X = 1;

S2: Y = 1;

S3: a = Y;

S4: if(a == 1)

S5: b = X;

*Is a=1, b=0 possible an allowed behavior?*

- SC ?
- TSO ?

TSO has same behaviors as SC model

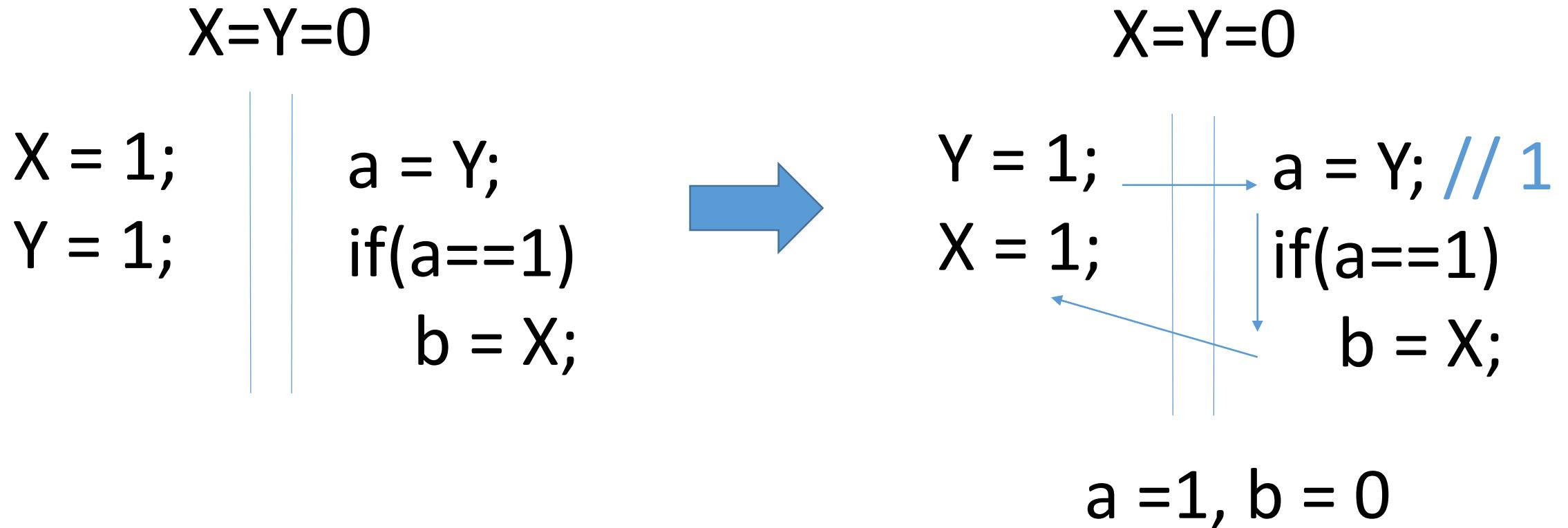
# Another Program: Message Passing (MP)

```
X=Y=0  
  
X = 1;           ||  
Y = 1;           ||  
                  a = Y;  
                  if(a == 1)  
                  b = X;
```

*Is a=1, b=0 possible an allowed behavior?*

- SC **X**
- TSO **X**

# Program: Message Passing (MP)



*Requires Store-Store relaxation*

# Program: Message Passing (MP)

```
A X=NULL; int Y=0;
```

```
X = A();           | a = Y;  
Y = 1;             | if(a==1)  
                  |   b = X;  
                  | // b is NOT NULL
```

```
X=NULL; int Y=0;
```

```
Y = 1;           | a = Y; // 1  
X = A();         | if(a==1)  
                  |   b = X;  
a = 1, b = 0
```

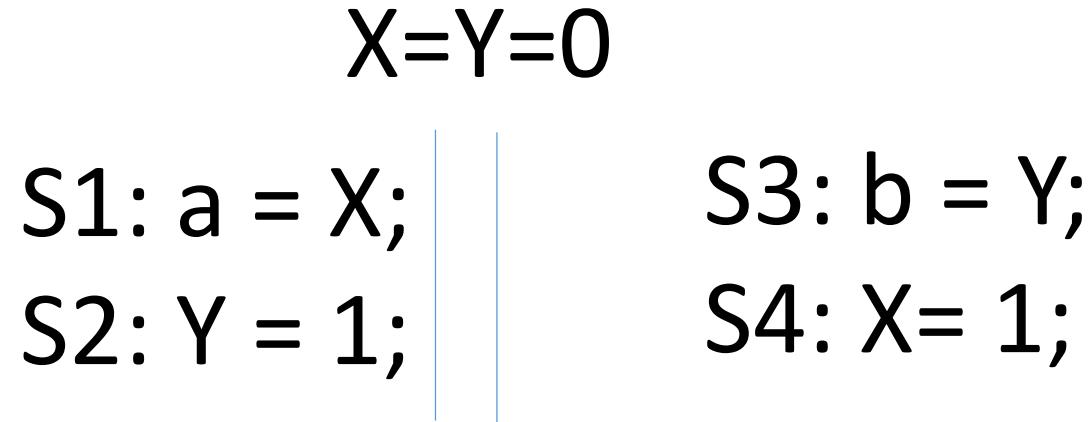
*PSO may reorder the accesses in the first thread*

# Partial Store Order (PSO)

Relaxation for Store – Store, Store - Load

<b>a;b</b>	<b>Store</b>	<b>Load</b>
<b>Store</b>	Y	Y
<b>Load</b>	N	N

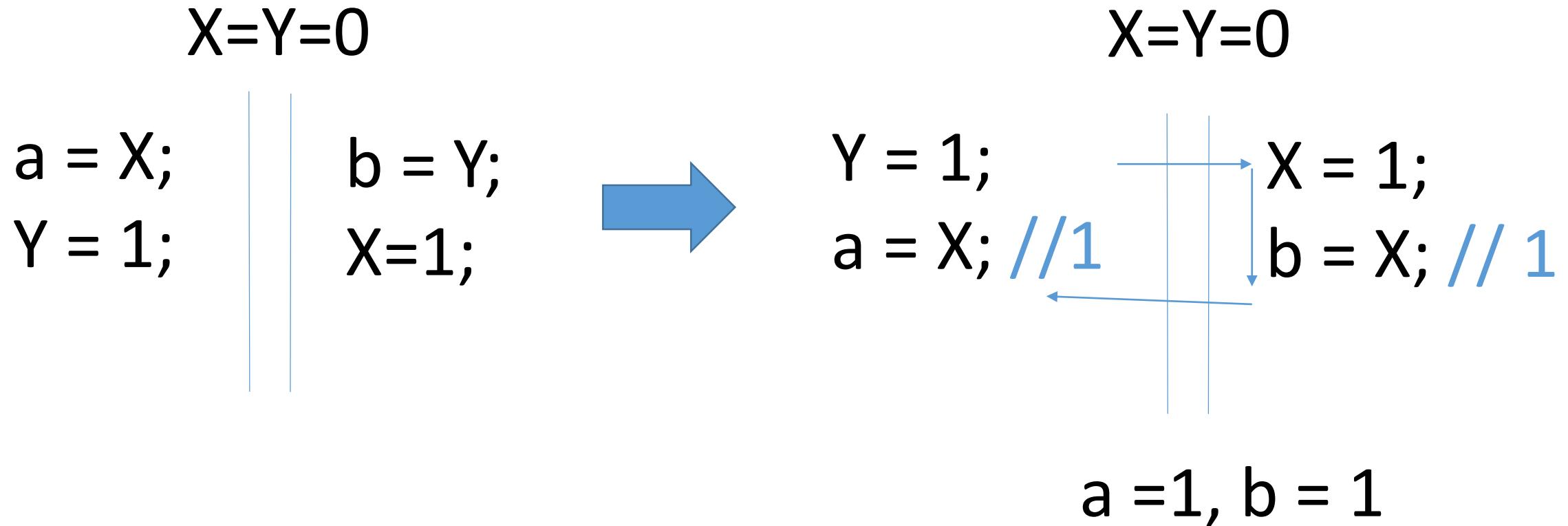
# Yet Another Program: Load Buffering (LB)



*Is  $a=1, b=1$  possible an allowed behavior?*

- SC ?      **X**       $a=1, b=0 \rightarrow S3; S4$
- TSO ?      **X**       $a=0, b=0 \rightarrow S1; S3; \dots$
- PSO ?      **X**       $a=0, b=1 \rightarrow$

# Program: Message Passing (MP)



*Requires Load-Store relaxation*

# Relaxed Memory Order (RMO)

Relaxation for Store – Store

<b>a;b</b>	<b>Store</b>	<b>Load</b>
<b>Store</b>	Y	Y
<b>Load</b>	Y	Y

# Summary

ACC. Pair Relaxation →	Store-Load	Store-Store	Load-Load	Load-Store
Memory Model ↓				
SC	N	N	N	N
TSO	Y	N	N	N
PSO	Y	Y	N	N
RMO	Y	Y	Y	Y

# Release-Acquire

1. All writes are release writes
2. All Reads are acquire reads
3. predecessor of release write cannot be moved after the release write
4. successor of acquire read cannot be moved before the acquire read

a;b	Store	Load
Store	N	Y
Load	N	N

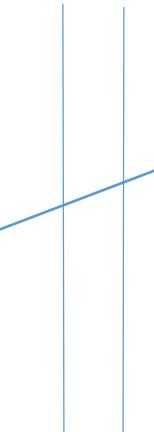
Same restrictions in TSO, but the models are different

# Release-Acquire

1. All writes are release writes
2. All Reads are acquire reads
3. predecessor of release write cannot be moved after the release write
4. successor of acquire read cannot be moved before the acquire read
5. When an acquire read reads from a release  
write it establishes a synchronization

# Program: Message Passing (MP) in RA

```
X=Y=0  
  
X = 1;  
Y = 1;      a = Y;  
            if(a==1)  
            b = X;
```



a = 1, b = 0

# References

## Chapter 5.6 (Memory consistency Models)

- Computer Architecture, Sixth Edition A Quantitative Approach  
by John L. Hennessy, David A. Patterson