EE/CSCI 451: Parallel and Distributed Computation

Lecture #4

8/27/2020

Viktor Prasanna

prasanna@usc.edu

ceng.usc.edu/~prasanna

University of Southern California

USC University of Southern California

Policies and Procedures

- Specify your current time zone and your background info.
  - Please fill the Google form: https://forms.gle/yqxoUa7NC7bMq5yv6

Grades (anxiety?)

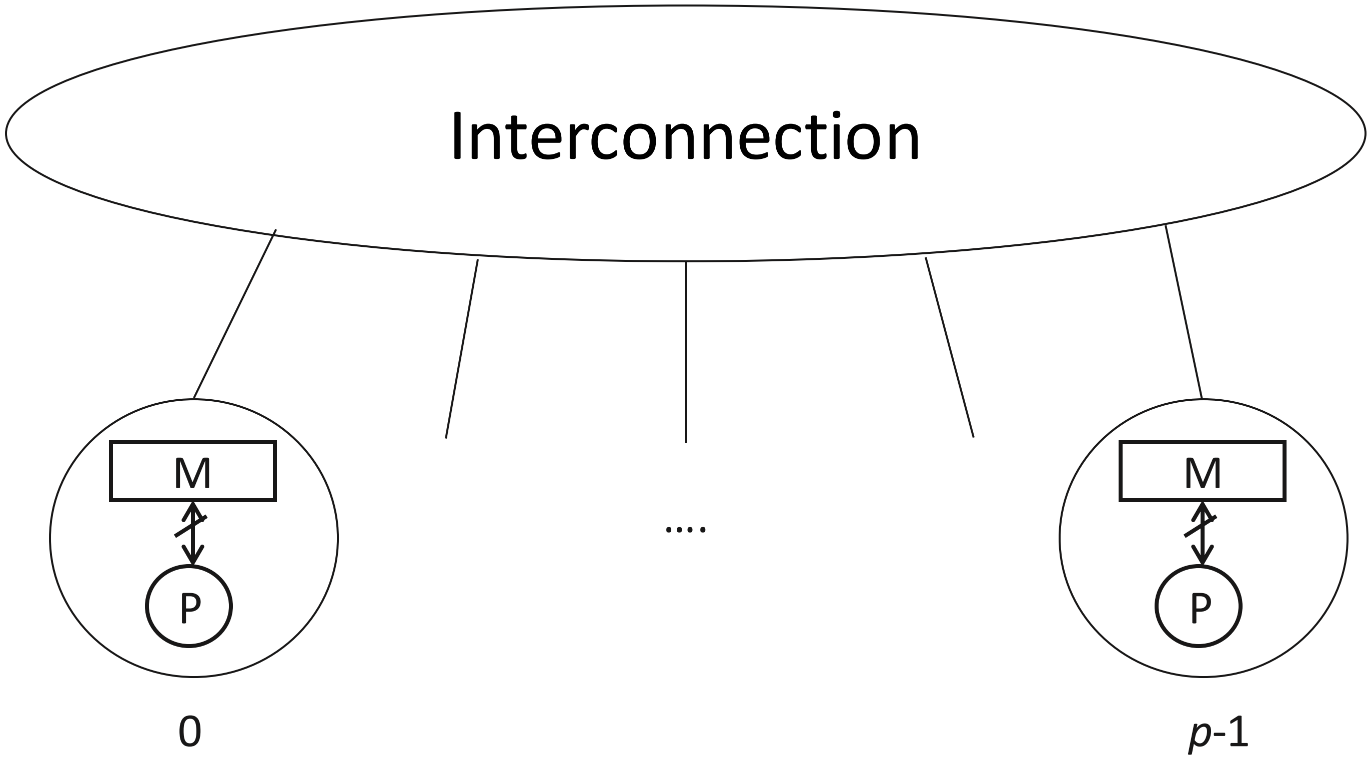
Grade Distribution: Fall 2019

- Midterm 1:
  - A & A- (70-100 pts) : 2 (4%)
  - B & B- (40-70 pts): 33 (73%)
  - C (30-40 pts): 9 (20%)
  - D (0-30pts): 1 (3%)
- Midterm 2:
  - A & A- (80-100 pts) : 5 (11%)
  - B & B- (50-80 pts): 34 (76%)
  - C (0-50 pts): 6 (13%)
- Final grade:
  - A & A- : 33 (73%)
  - B & B- : 11 (24%)
  - C : 1 (3%)

Outline

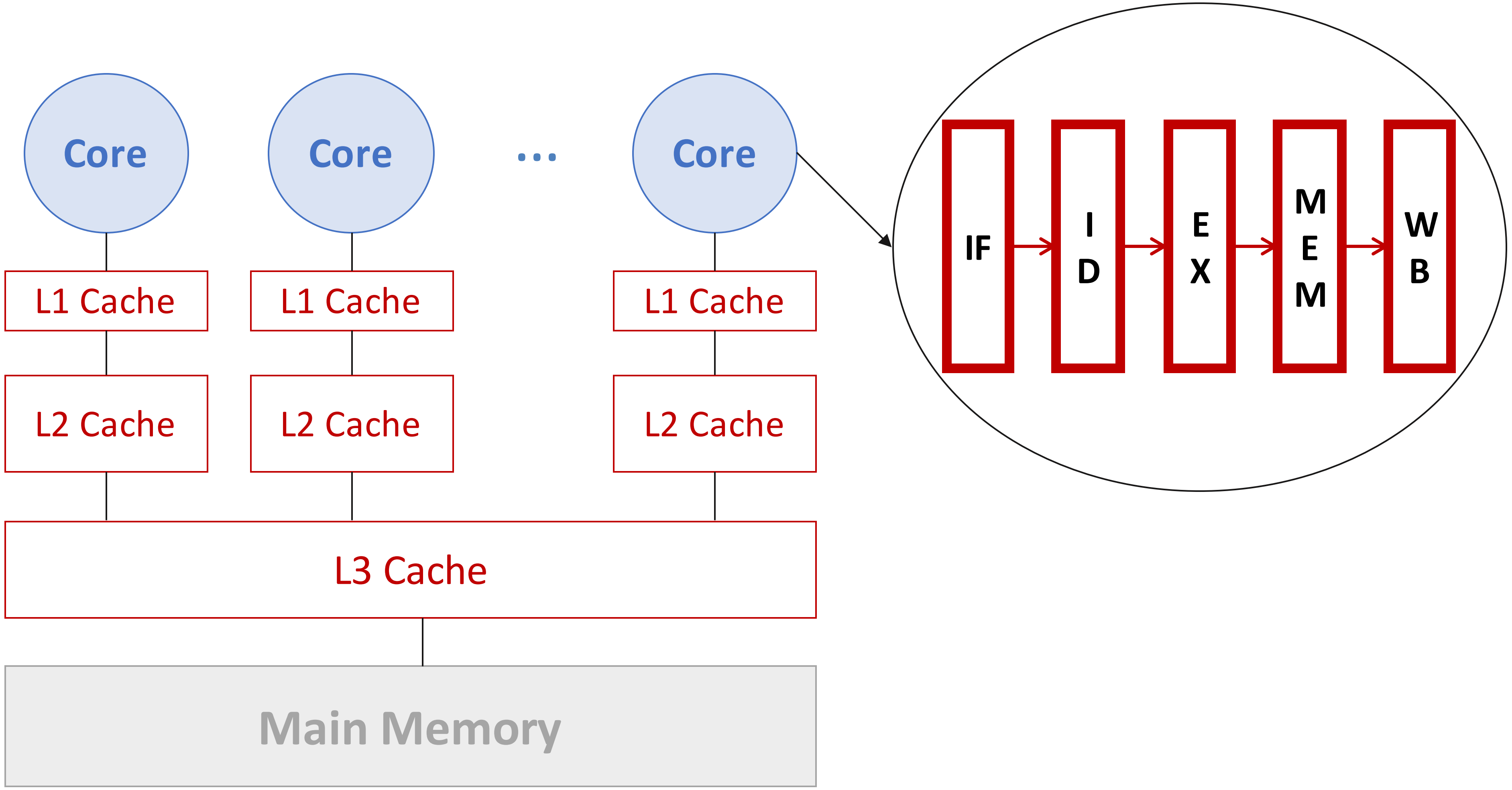
- From last class
  - Memory system
    - Latency, Bandwidth: performance implications
    - Cache: impact on performance
    - Data layout
    - Impact of memory system on performance
      - Random/streaming access
      - Cache
    - Latency hiding
- Today
  - Shared memory programing model
  - Scalability of a parallel solution
  - A simple model of shared memory parallel computation
  - Example shared memory programs

Generic Parallel Architecture



p nodes

Node Architecture



Parallelism

- In each node (multiple cores in parallel)
- Across the p nodes (p nodes in parallel)
- Implicit parallelism

Distributed Shared Memory

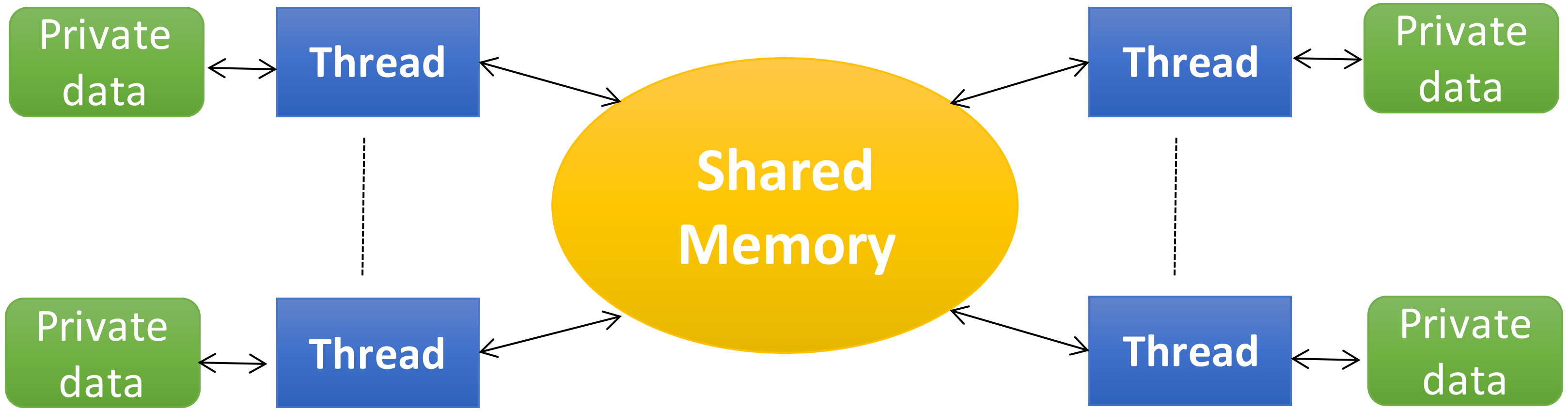
- The address space is distributed across the local external memory of the nodes
- Access to shared data provided by the architecture
    - Load R, X

May be local or in remote node

- Access to (local and remote) data is implicit

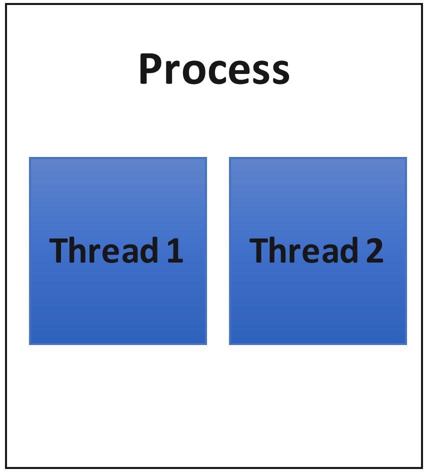
Shared Address Space Programming (1)

- All threads have access to the same global, shared memory
- Threads can also have their own private data
- Programmer in responsible for synchronizing access (protecting) globally shared data (to ensure correctness of the program)



Shared Address Space Programming (2)

- Thread
  - An independent stream of instructions
  - Can be scheduled to run concurrently and/or independently by the operating system
  - Exists within a process and uses the process resources
  - Lightweight: most of the overhead already been accounted for through the creation of the process
  - Example: POSIX Threads (Pthreads)



Shared Address Space Programming (3)

- Example 1
  - $n \times n$ Matrix multiplication

$\boldsymbol{i}$ from 1 to n
    $\boldsymbol{j}$ from 1 to n
        $C(\boldsymbol{i},\boldsymbol{j})$ = dot_product ($\boldsymbol{A}(\boldsymbol{i},:),\boldsymbol{B}(:,\boldsymbol{j})$)

  - n^2 independent iterations → n^2 threads (no dependency among threads)

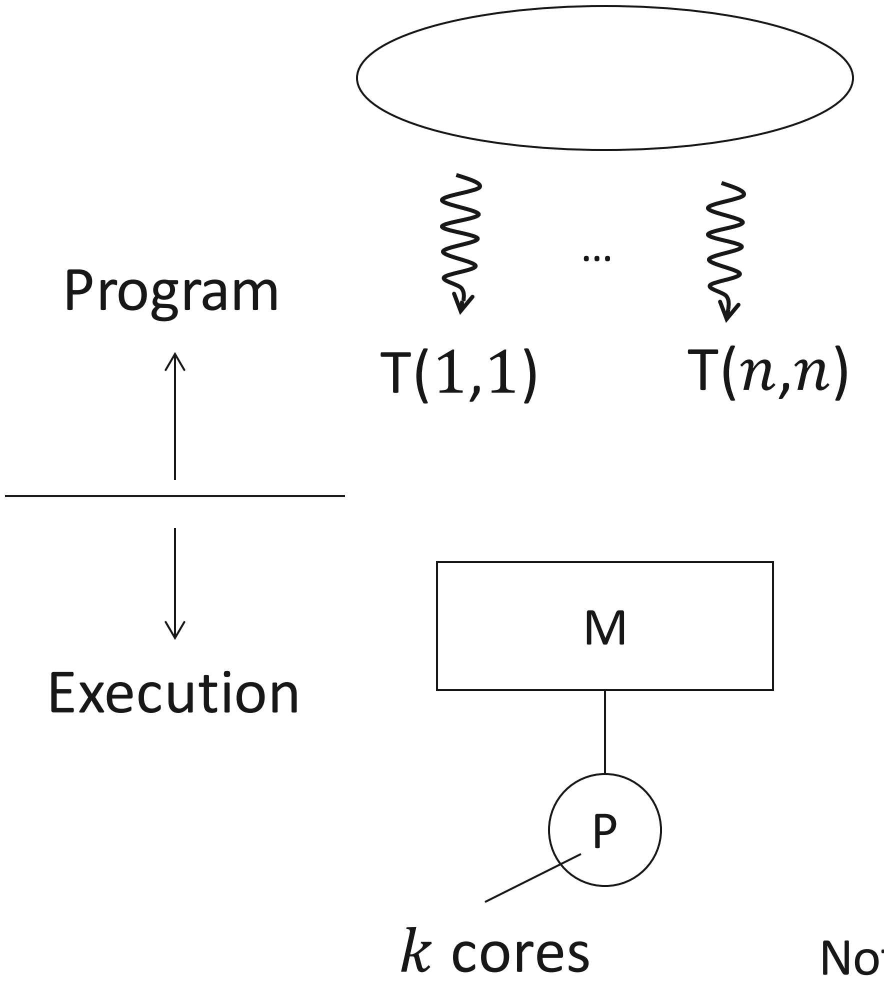
Thread 1 $T(1,1)$ $\boxed{C(1,1) = \text{dot\_product}(A(1,:), B(:,1))}$

⋮

Thread n^2 $T(n,n)$ $\boxed{C(n,n) = \text{dot\_product}(A(n,:), B(:,n))}$

Run concurrently

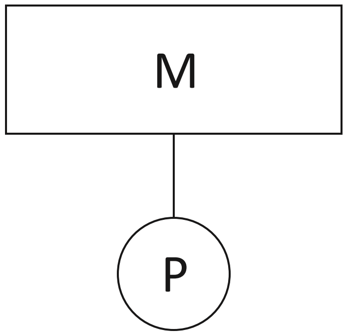
Shared Address Space Programming (4)



Note: Can execute on one core

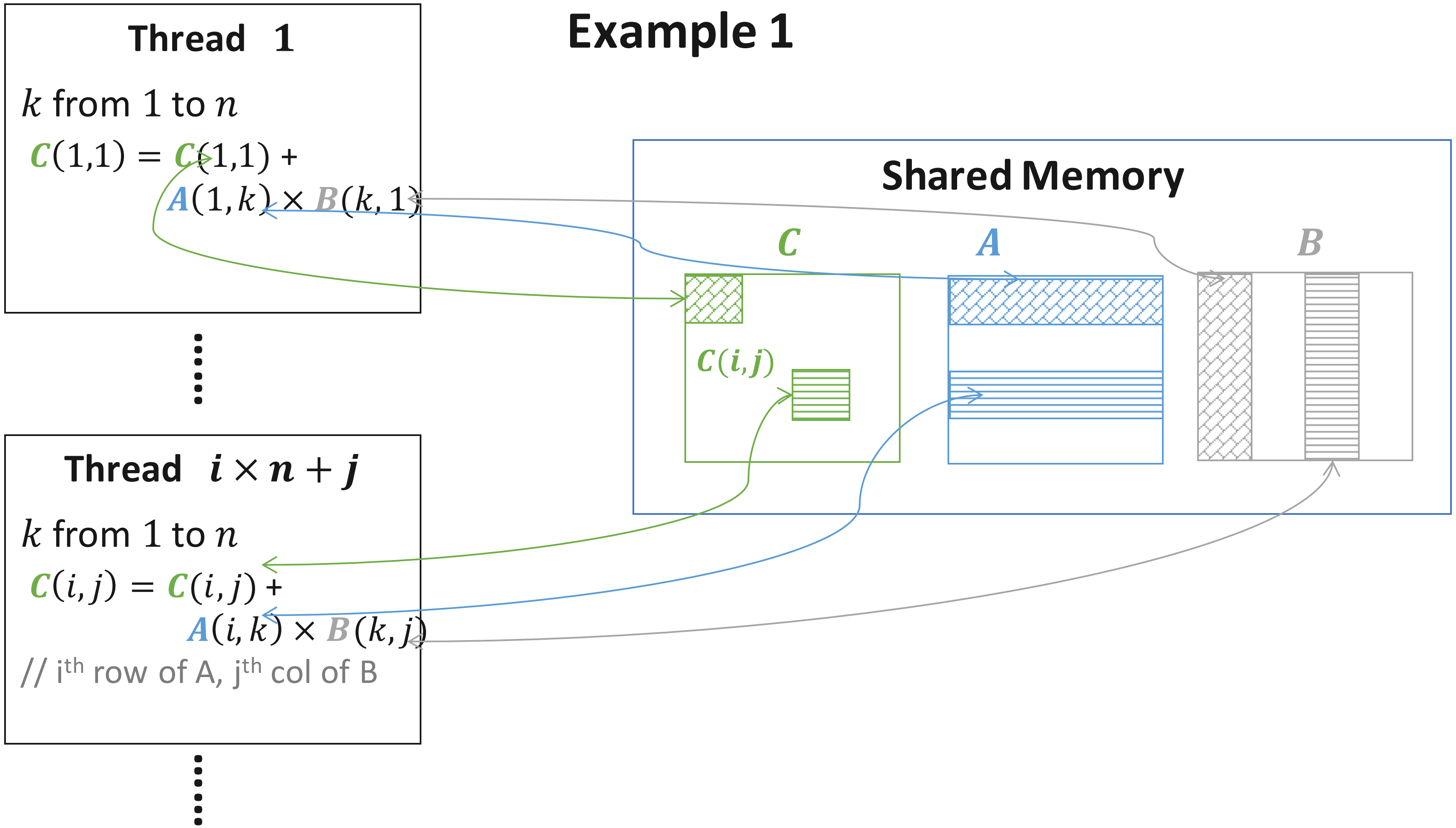
Shared Address Space Programming (5)

- Assume $k = n^2$
- At time t, all cores execute the same instruction ??
  - ➢No Guarantee
- Execution of the threads is **asynchronous** even if all the cores have the same clock
  - OS scheduling
  - Cache state



k cores

Shared Address Space Programming (6)



Shared Address Space Programming (7)

Note:

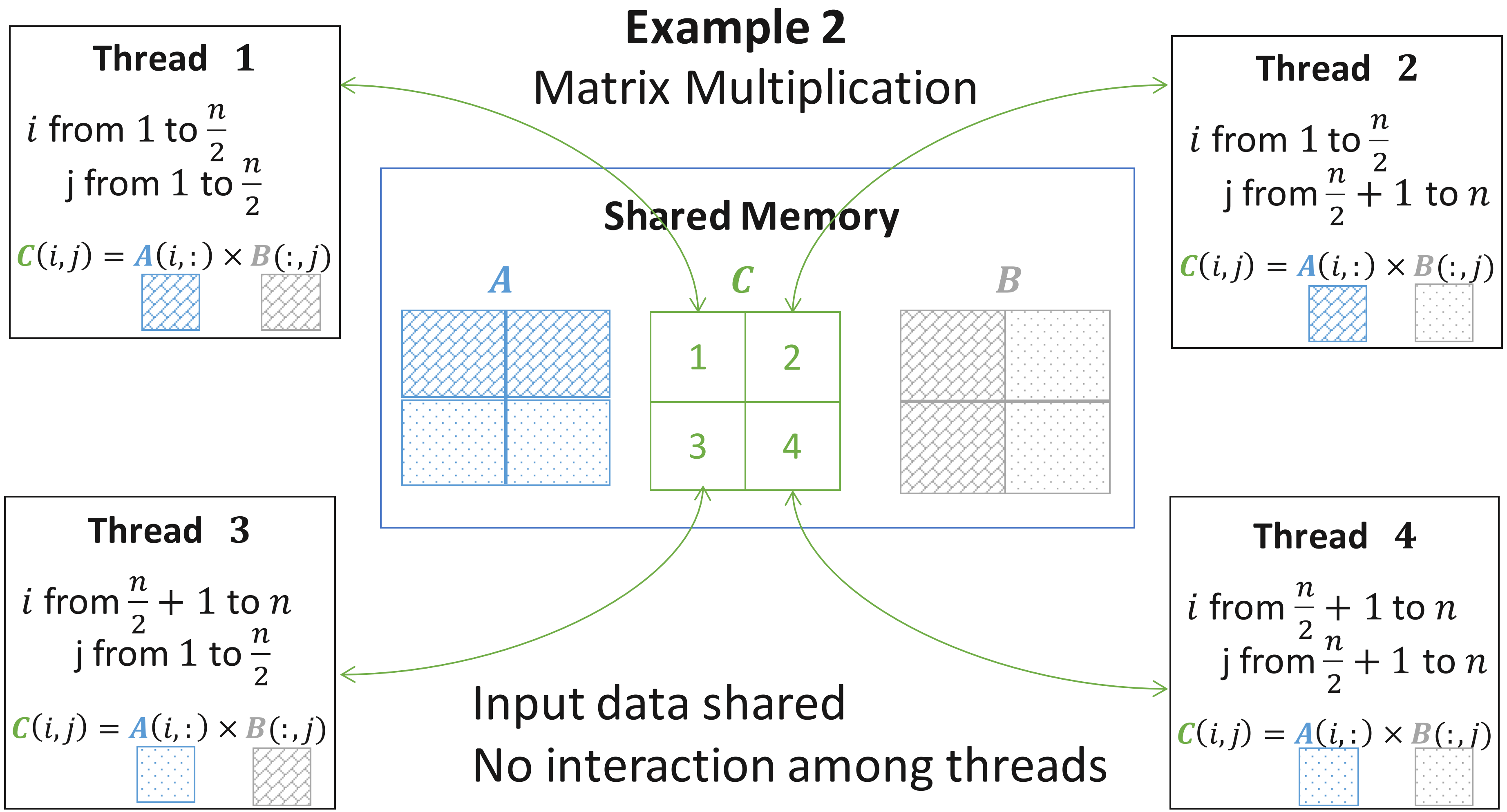
- Input data (A, B matrix) shared by the threads
- Output data (C matrix) **not** shared by threads

  Each element of output matrix is computed (updated) by only one thread

  => **no** dependencies between threads

  => **no** coordination is needed between the threads
- Very simple shared address space program (Embarrassingly parallel)
- Memory access cost?
  - If k threads execute ($k \leq n^2$), $2k$ concurrent memory accesses by k threads

Shared Address Space Programming (8)



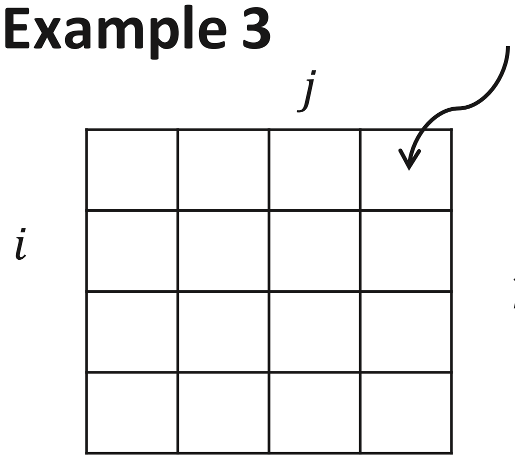
Input data shared

No interaction among threads

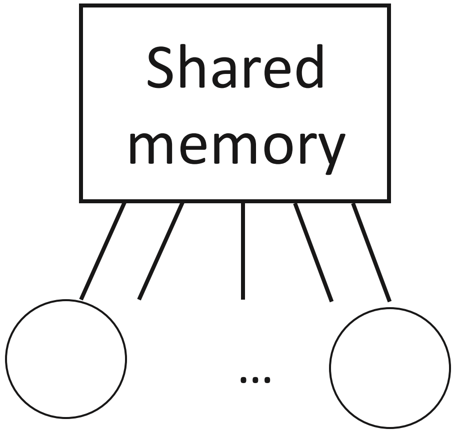
Shared Address Space Programming (9)

Example 3

block size $\frac{n}{p} \times \frac{n}{p}$



$p \times p$ mesh
of threads



T $(1,1)$ T(p,p)

p^2 threads

T(i,j) computes $(i,j)^{th}$ output block
Each block of size $\frac{n}{p} \times \frac{n}{p}$

Thread →

Do i = 1 to p
    Do j = 1 to p
        T(i,j) computes $(i,j)^{th}$ output block
    end
end

Shared Address Space Programming (10)

Thread (i, j)

Output block $C(i, j) \leftarrow 0$

Do k = 1 to p

Read Block (i, k) of A

Read Block (k, j) of B

Perform matrix multiplication using the blocks

Update output block $C(i, j)$

end

Block size = $\frac{n}{p} \times \frac{n}{p}$

Total # of threads = p^2

Shared Address Space Programming (11)

Thread (i, j) performs $p(\frac{n}{p})^3$ Add & Multiply operations = $2\frac{n^3}{p^2}$ operations

Total # of ops performed by all the threads $= p^2\ (\frac{2n^3}{p^2})$

$= 2n^3$ operations

Total data fetched from shared memory

$$= p^2 \left(p \cdot \underbrace{\left[\frac{n}{p} \times \frac{n}{p} \right]}_{\text{Block size}} \cdot 2 \right)$$

$$= 2\underbrace{p}_{\text{Total \# of threads}} n^2$$

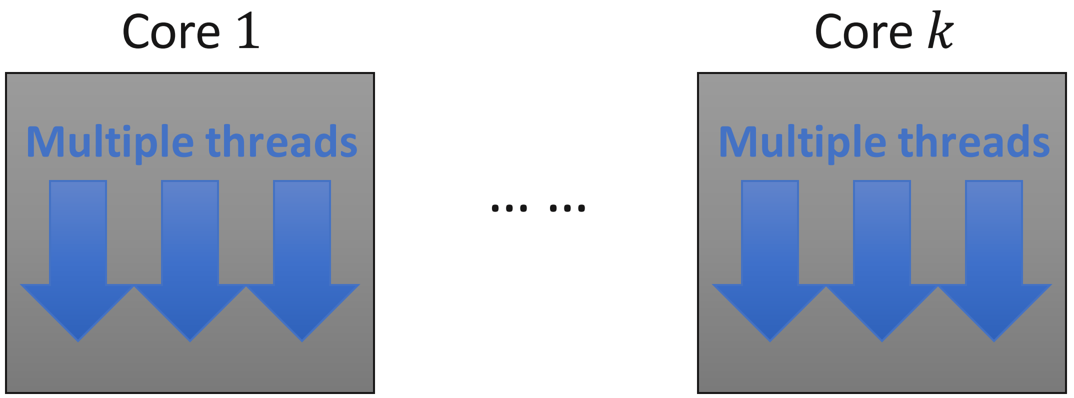
Total # of threads

Block size

Threads on Multi-core platform

- Operating system scheduler maps threads/ processes to cores.
- We can bind a thread to a specific core.
- Multiple threads can run on the same core.
- More threads → resource contention ↑; thread switch overhead ↑; cache hit ratio ↓

→ Can hide access latency



Scalability (1)

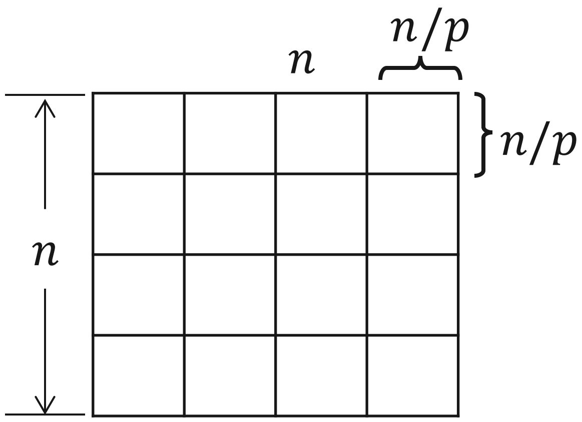
- Poor Scalability?
  - Communication Cost?
  - Coordination among the processors?
- Example: $n \times n$ matrix multiplication

  $C \leftarrow A \times B \quad A, B$ are $n \times n$ matrices

  p^2 processors

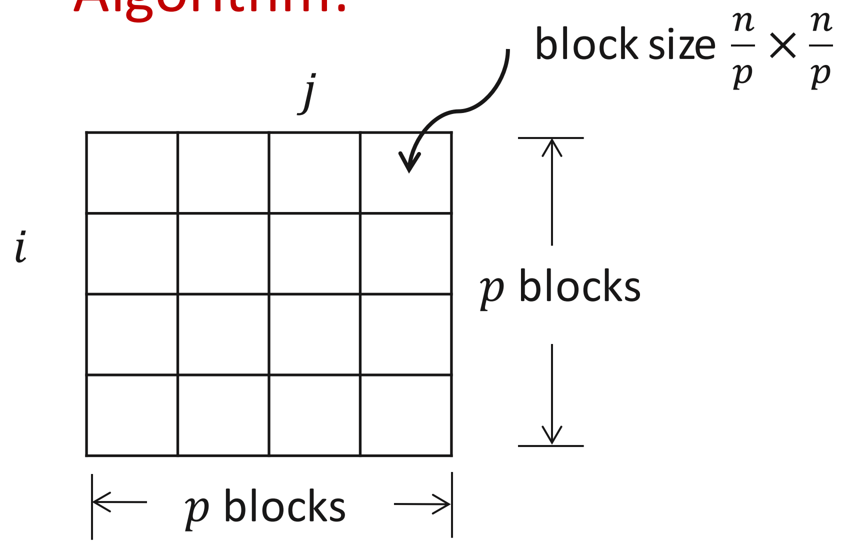
  Each processor computes a $n/p \times n/p$ sub-matrix of C

(See Example 3 earlier)

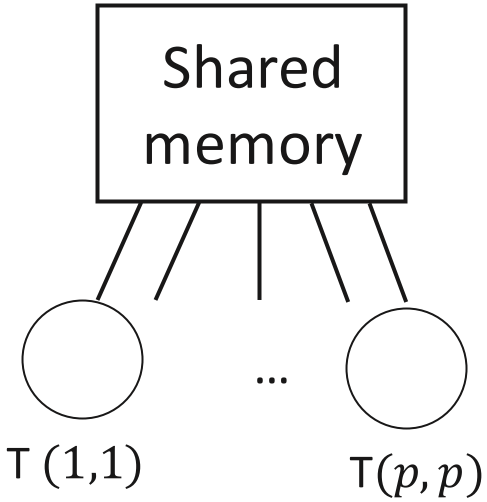


Scalability (2)

Algorithm:



T(i, j) computes $(i, j)^{th}$ output block
Each block of size $\frac{n}{p} \times \frac{n}{p}$



p^2 threads

Thread

```
Do i = 1 to p
    Do j = 1 to p
        T(i, j) computes (i, j)^th output block
    end
end
```

Scalability (3)

Time to compute $(i, j)^{\text{th}}$ output block

$= p \times 2(\frac{n}{p})^3$

$= \frac{2n^3}{p^2}$

Scalability (4)

Total volume of data communicated

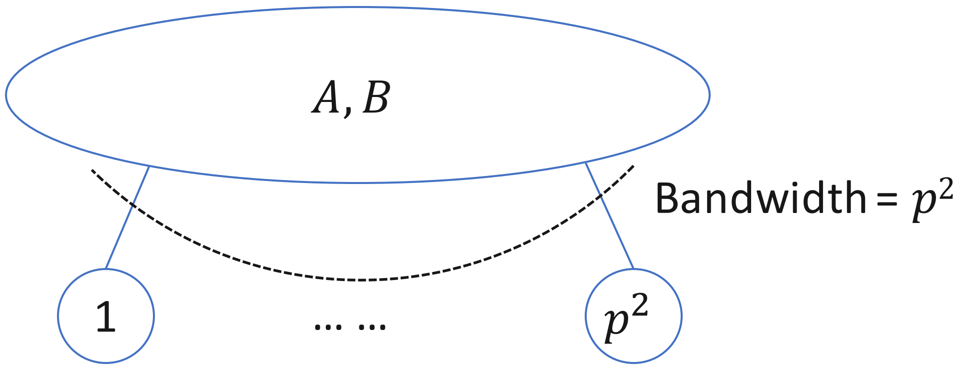
$$= p^2 \times (n/p \times n/p) \times p \times 2$$
$$= 2p \times n^2$$

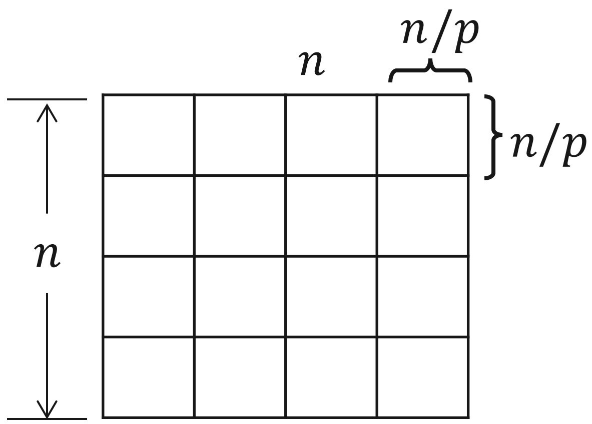
Suppose bandwidth between shared memory and the processors = p^2

- p^2 threads access p^2 values (1 value/PE) in 1 unit of time

Time for communication $\propto n^2/\mathrm{p}$

Time for computation $\propto n^3/p^2$





Scalability (5)

Total time = Computation time $+$ Communication time

$$= \frac{n^3}{p^2} + \frac{n^2}{p}$$

$$= \frac{n^2}{p}(n/p) \; + \frac{n^2}{p}(1)$$

Suppose Total number of processors $p^2 \leq n^2$

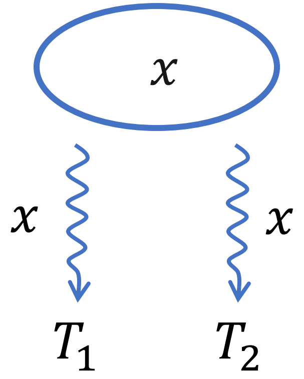
Time for computation dominates time for communication

Parallel time $\propto \dfrac{\text{Serial Time}}{\text{\# of processors}}$

Scalable Solution

Shared access and Synchronization

1. Shared variable access

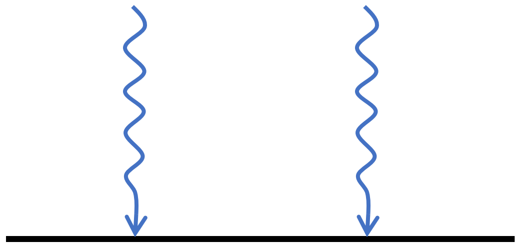


Update shared variable x
Speed of execution of threads?
Order of execution?

Produce correct output independent of execution speed of T_1 and T_2

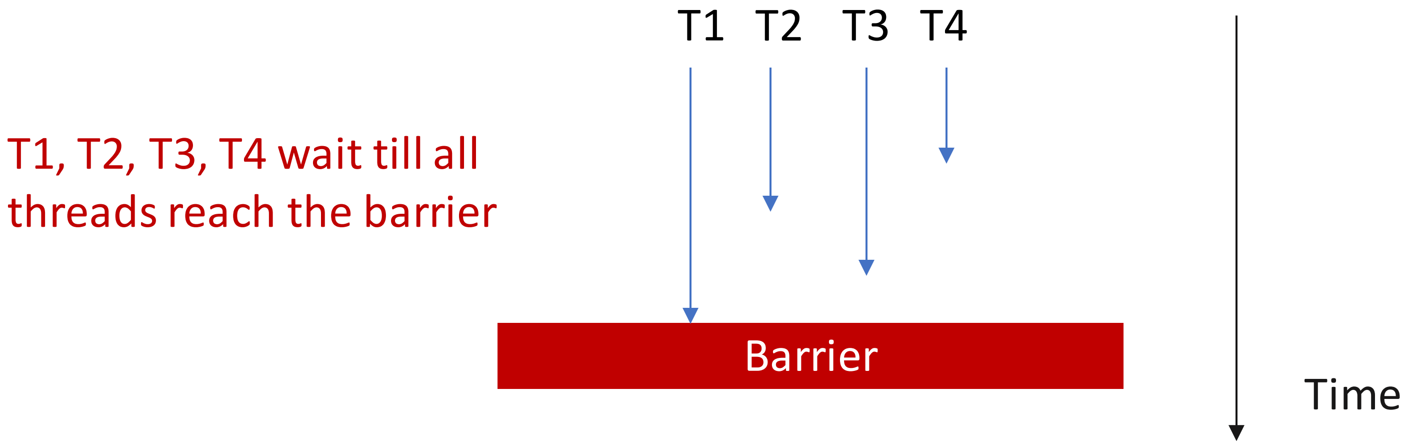
2. Synchronization

Threads should agree they all reached an
(agreed upon) execution state before proceeding further



Barrier (1)

- Synchronization method
- Barrier objects can be created at certain places in the program
- Any thread which reaches the barrier stops until all the threads have reached the barrier



Barrier (2)

Example:

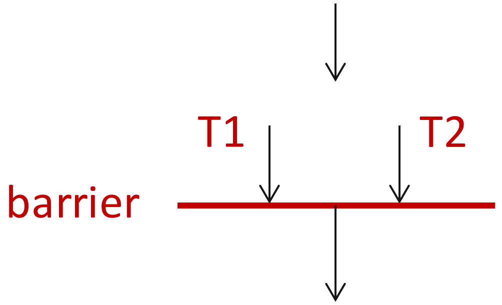
Find max in array A[0:N-1]:

CreateThread(find_max,A[0], A[N/2-1], max1) {T1}

CreateThread(find_max,A[N/2], A[N-1], max2) {T2}

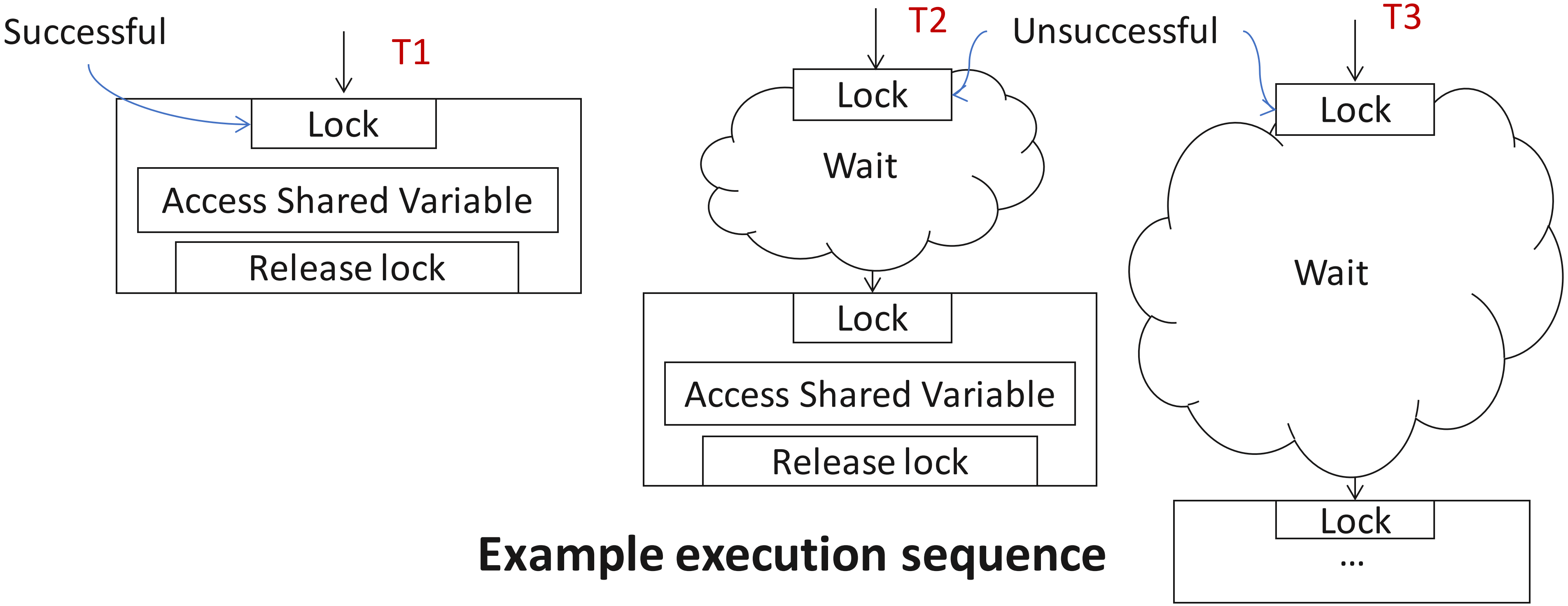
Barrier()

Return max(max1, max2)



Shared Variable Access (1)

- Threads acquire lock to modify a shared variable
- Release lock when done
- Only 1 thread can acquire a lock at any time



Example execution sequence

Shared Variable Access (2)

Example: Find max between two threads

Each thread has a local value $i, j \geq 0$

Initialize Max (Max is a shared variable) $= 0$

Thread 1

1. Acquire_lock(Max)
2. If (i > Max)
3. Max = i
4. Release_lock(Max)

Thread 2

Acquire_lock(Max)
If (j > Max)
Max = j
Release_lock(Max)

Shared Variable Access (3)

Possible execution sequence depends on execution speed of threads

Max $\leftarrow$ 0	Max $\leftarrow$ 0
Thread 1	Thread 2
Thread 2	Thread 1

In both cases correct output produced

Shared Variable Access (4)

Example: Find max between two threads

Each thread has a local value $i, j \geq 0$

Initialize Max (Max is a shared variable) $= 0$

Suppose lock is not used

Thread 1 (T_1)	Thread 2 (T_2)
1. If (i > Max)	1. If (j > Max)
2. Max = i	2. Max = j

Suppose $i = 10, j = 20$

Possible execution sequence that will produce incorrect result:

	Core 1 (T_1)	Core 2 (Faster) (T_2)	
		T_2 Ins 1	Max = 0
Max = 0	T_1 ins 1		
		T_2 Ins 2	Max = 20
Max = 10	T_1 Ins 2		

Final result: Max = 10

Shared Variable Access (5)

Atomic Instruction

- Ex. Acquire_lock, Release_lock
- Single Instruction, once execution starts it is completed without interruption
- If i > Max then Max = i is not an atomic instruction

Thread 1 (T_1)	Thread 2 (T_2)
1. If (i > Max)	1. If (j > Max)
2. Max = i	2. Max = j

- Example code for Thread 1

```
1            LD R1, Max
2            CMP R1, adrs of i
3            IF Flag JMP Update
4            Exit
5    Update: STORE Max, R1
```

Possible execution sequence

Suppose $i = 10, j = 20$

Core 1 (T_1)	Core 2 (T_2) Faster
	1
1	2
	3
2	5
3	
5	

Max = 10 ← incorrect

Implementing Lock (1)

Test_and_Set Instruction ← Atomic Instruction

(Only one thread can execute this instruction at any time)

Test_and_Set (*lock*)

Meaning of *lock* = 0 → Shared variable associated with *lock* is **not** being used

lock = 1 → Shared variable associated with *lock* is being used

Execution of Test_and_Set (*lock*): Returns value of *lock* into a Reg (R0)

Sets *lock* to 1

Atomic

Implementing Lock (2)

Initially *lock* = 0

Acquire_lock(*lock*)

Spin: Test_and_Set(*lock*)

If R0 = 1, go to Spin

Spinning (waiting)

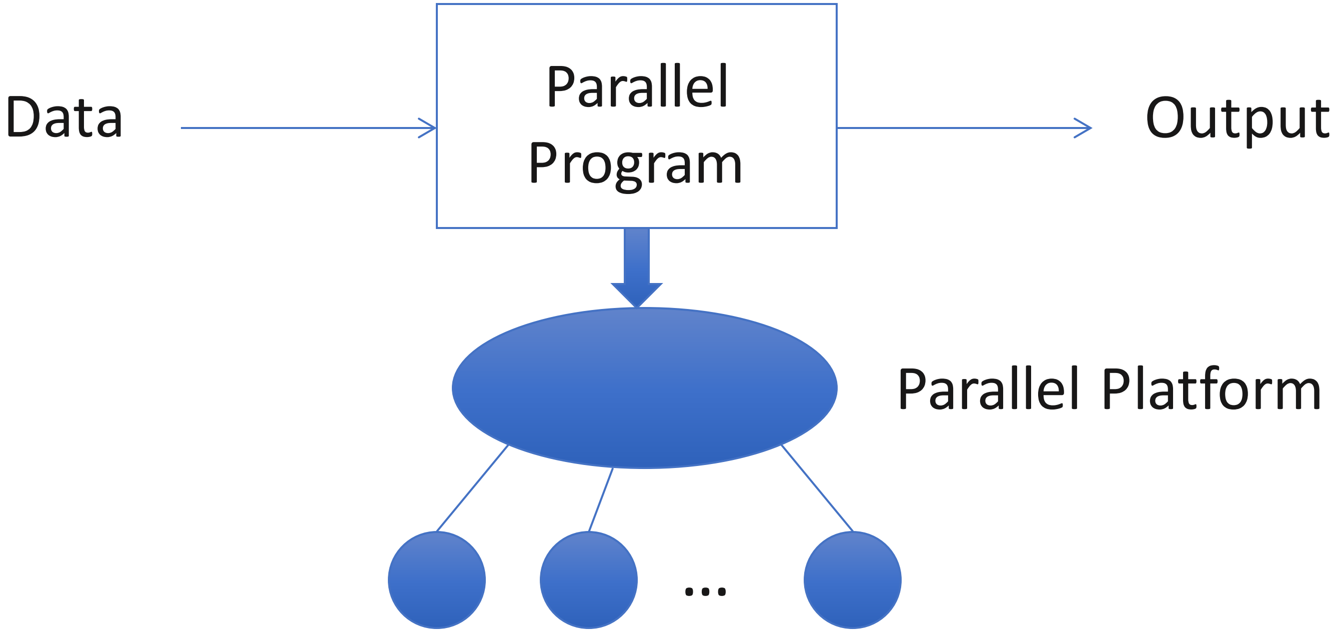
Atomic instructions

Release_lock(*lock*)

Set *lock* to 0

Atomic instruction

Correct Parallel Program



For **all** data inputs, for **all** execution sequences, correct output is produced

Order in which instructions are executed

T_1	T_2
I_1	J_1
I_2	J_2
I_3	J_3
I_4	J_4

Example execution sequence

I_1	J_1	
	J_2	
I_2		
I_3	J_3	
I_4		
	J_4	Time

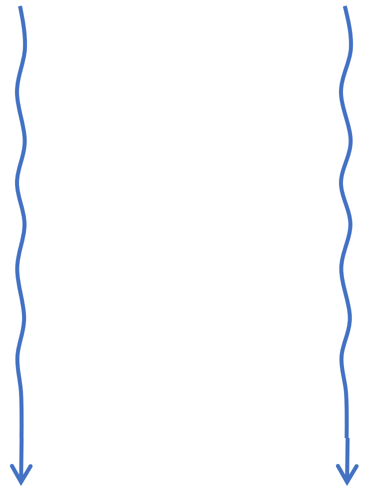
Execution sequence

- Can be dependent on input data
- For the same input data, can result in a different execution sequence on different parallel platforms
- For the same input data, can result in different execution sequence on the same parallel platform when executed again

Deadlock May Occur

Thread 1	Thread 2
Lock(x)	Lock(y)
Lock(y)	Lock(x)
Access x,y	Access x,y

Race Condition



Output depends on the rate at which the two parallel threads execute (for a given input)

⇒ leads to incorrect output

Matrix Multiplication using Shared Variable (1)

$$C \leftarrow A \times B$$

$$n \times n$$

Thread(i, j) accesses A(i, j) {owns A(i, j)} i. e., local data

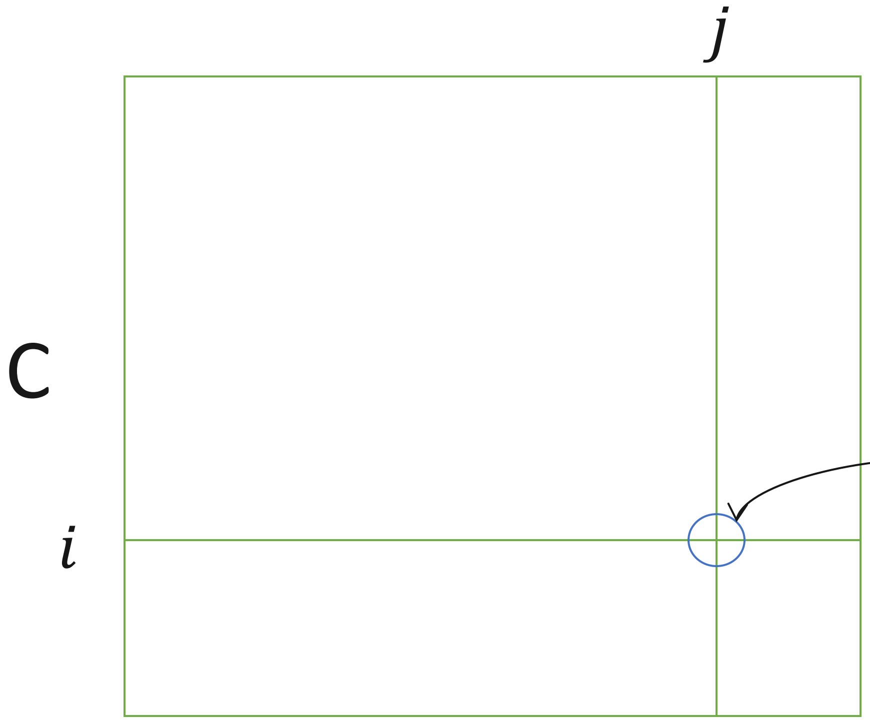
All other data is shared

Each thread is responsible to update all outputs $C(i, k)$, $1 \leq k \leq n$, to which A(i, j) contributes to

Eg. $C(i, 1)$ = A$(i, 1) *$ B$(1,1)$+ … $\boxed{\text{A}(i, j)}$ $*$ B$(j, 1)$+ … A$(i, n) *$ B$(n, 1)$

Contributes to $C(i, k)$, $1 \leq k \leq n$

Matrix Multiplication using Shared Variable (2)



Shared variable updated by Thread (i, k), $1 \le k \le n$

Matrix Multiplication using Shared Variable (3)

Thread$(\boldsymbol{i}, \boldsymbol{j})$

//Uses local data *A(i,j)* to update **all** *C(i,k)*

Do k from 1 to n

Acquire_lock ($C(i, k)$)

$C(i, k) \leftarrow C(i, k) + A(i, j) * B(j, k)$

Release_lock ($C(i, k)$)

End

Matrix Multiplication using Shared Variable (4)

Main Program

Initialize *C(i,j)* to 0

Create n^2 threads

Wait for all threads to complete

End

Matrix Multiplication using Shared Variable (5) Correctness?

Main Program

Initialize *C(i,j)* to 0

Create n^2 threads

End

Thread$(\boldsymbol{i}, \boldsymbol{j})$

Do k from 1 to n

Acquire_lock $(C(i,k))$

$C(i,k) \leftarrow C(i,k) + A(i,j) * B(j,k)$

Release_lock $(C(i,k))$

End

Main Program

Create n^2 threads

End

Thread$(\boldsymbol{i}, \boldsymbol{j})$

Initialize *C(i,j)* to 0

Do k from 1 to n

Acquire_lock $(C(i,k))$

$C(i,k) \leftarrow C(i,k) + A(i,j) * B(j,k)$

Release_lock $(C(i,k))$

End

Matrix Multiplication using Shared Variable (6) Correctness?

Main Program

Create n^2 threads

Wait for all threads to complete

End

Thread$(\boldsymbol{i}, \boldsymbol{j})$

Initialize *C(i,j)* to 0

Barrier

Do k from 1 to n

Acquire_lock ($C(i,k)$)

$C(i,k) \leftarrow C(i,k) + A(i,j) * B(j,k)$

Release_lock ($C(i,k)$)

End

Matrix Multiplication using Shared Variable (7)

Notes:

- Each output $C(p,q)$ is shared by n threads
- Order in which $C(p,q)$ is updated can vary (depends on how the threads are scheduled and execution speed of the processors)
- In principle, any $C(p,q)$ can be updated by n threads in any of the $n!$ ways

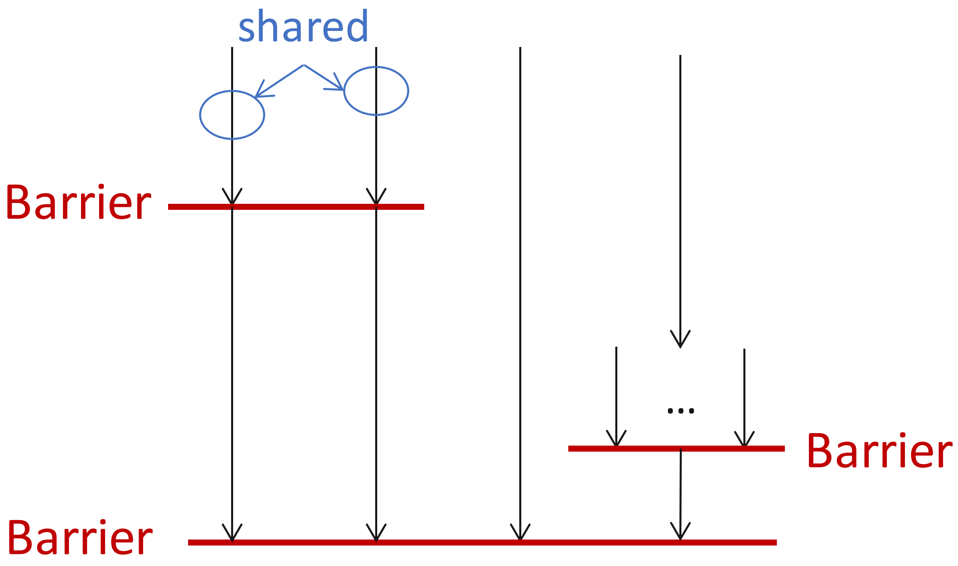
Programmer (Application Developer)'s Responsibility

- Ensure shared memory access is coordinated
- Ensure there is no race condition
- Insert appropriate barrier(s)
- Ensure there is no deadlock

Pthreads Program Structure

Initialize

Create threads



Complete

Asynchronous Execution

- No global clock coordinating execution
- Order of execution of instructions depends on
    - input data
    - scheduling algorithm (OS)
    - speed of the processors
    - speed of communication network

Summary

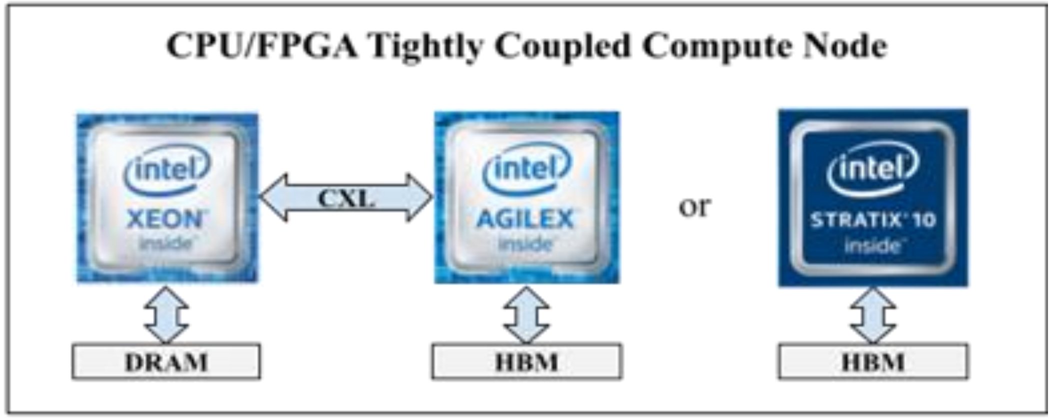
- Shared memory programming model (Pthreads)
  - Thread
  - Thread vs. core
  - Shared variables, local variables
  - Asynchronous execution
  - Programming abstractions
    - Lock
    - Barrier
  - Coordination
  - Overheads
  - Correctness

- Examples of shared memory programs, syntax,… (Discussion session)

Tightly coupled CPU/FPGA accelerators

- Programming model
  - One (or more) CPU thread(s) to coordinate tasks running on FPGAs.
  - One (or more) CPU thread(s) for CPU compute
  - Explicit API for synchronization and communication.

- Programming language support
  - Host:
    - OpenCL
    - OneAPI
  - FPGA:
    - HLS (Vivado HLS; OpenCL)
    - Verilog/VHDL



CPU / FPGA threads interaction