EE/CSCI 451: Parallel and Distributed Computation

Lecture #5

9/1/2020

Viktor Prasanna

prasanna@usc.edu

ceng.usc.edu/~prasanna

University of Southern California

Outline

- Announcement
  - PHW2 released today, due 9/14 AOE
  - HW2 to be released on Thursday 9/3, due 9/10 AOE
- From last class
  - Shared memory programing model
  - Scalability of a parallel solution
- Today
  - A simple model of shared memory parallel computation
  - Example shared memory programs
  - OpenMP
    - OpenMP programming model
    - OpenMP directives
    - Examples

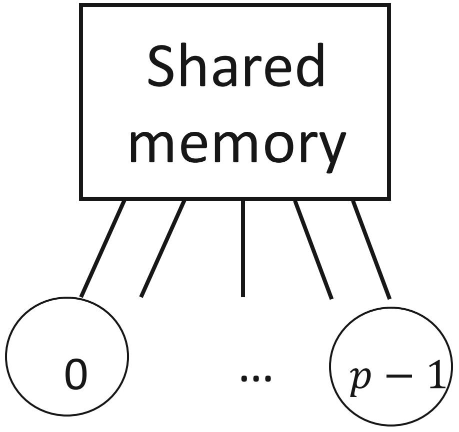
A Simple Model of Shared Address Space Parallel Machine (PRAM) (1)

1 unit of time = Local access
shared memory access

Synchronous model

Parallel time = total number of cycles

Pthreads programming model?
Asynchronous shared memory ??



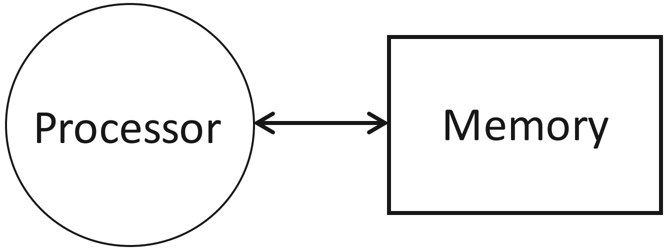
p processors

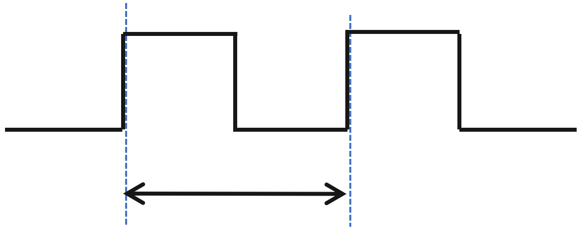
PRAM (2)

Random Access Machine (RAM)

- Random Access
  - Access to **any** memory location (Direct access)
- Time
  - Access to memory ⟶ 1 unit of time
  - Arithmetic/Logic operation ⟶ 1 unit of time
- Serial time complexity $T_s(n)$
  - Example:

    Merge Sort $T_s(n) = O(n \cdot \log n)$

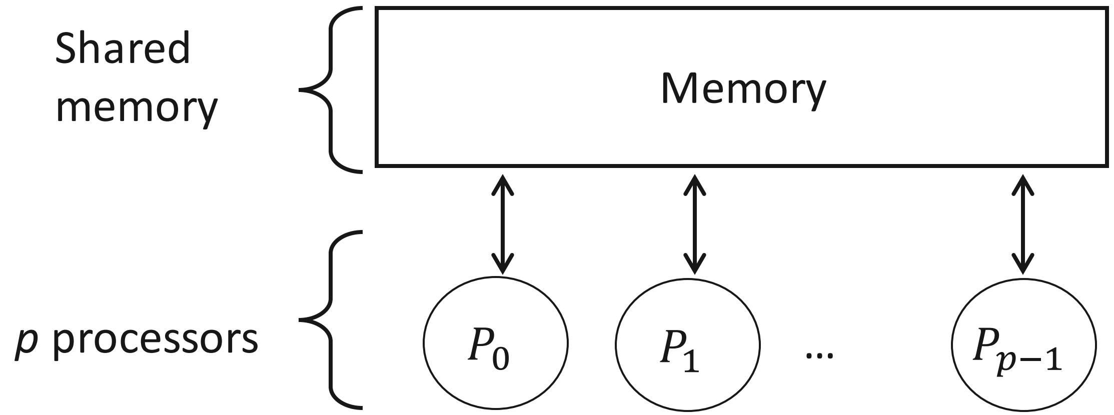




1 unit of time

PRAM (3)

Parallel Random Access Machine (PRAM)



Random Access Machines, executing in parallel using a common clock

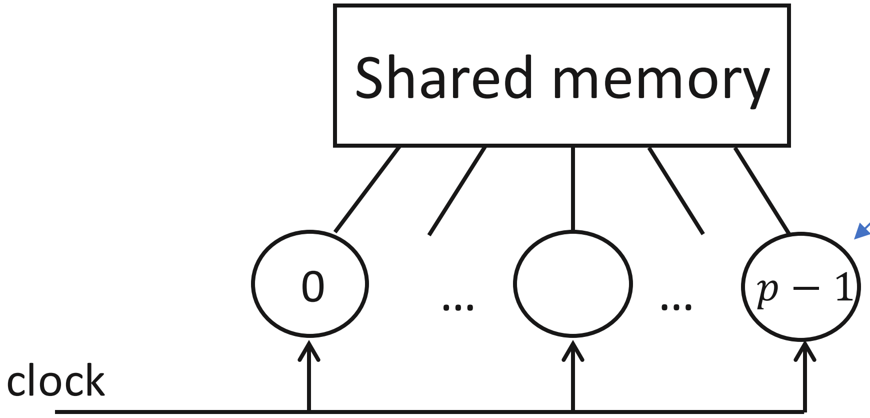
PRAM (4)

- Parallel Random Access Machine (PRAM)
  - Use shared memory for communication between processors
  - Synchronous Model (Global clock)

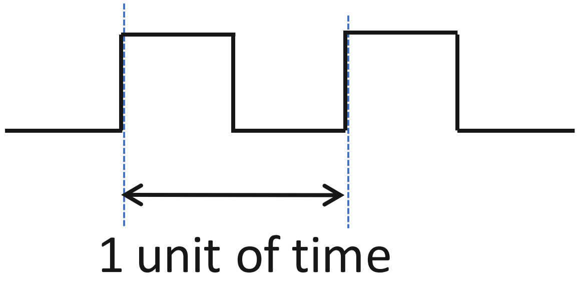
- Time
  - Access to **any** memory location → 1 unit of time
  - Execute one instruction in each processor (arithmetic/logic operation) → 1 unit of time

PRAM (5)

PRAM is a synchronous model



Each processor is a RAM
Local program acting
on data in shared memory



Synchronous execution

For all i, i-th instruction in the execution sequence is executed in the i-th cycle by **all** the processors

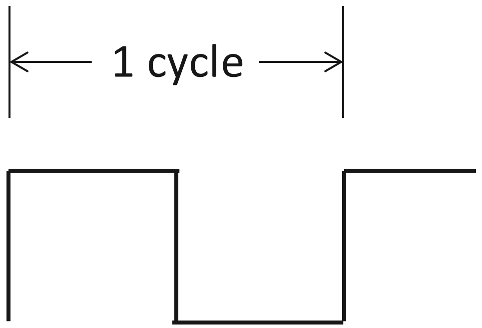
PRAM (6)

PRAM features

p = # of Processors

- In each cycle, p locations can be accessed (in parallel)
- All communication overheads ignored
- Ideal shared memory
- In one cycle

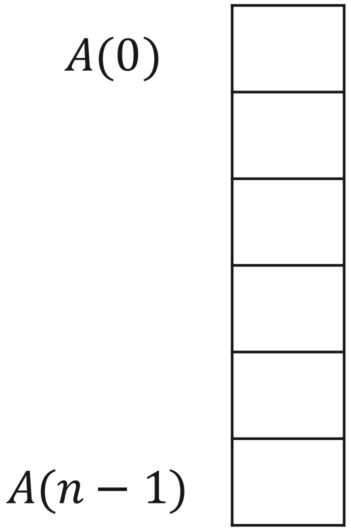
  in each processor
  - Read (from shared memory)
  - Compute (using data in the processor)
  - Write (into shared memory)
- The code executed by different processors can be different
- Lock step execution
- p operations in parallel



Adding on PRAM (1)

Simple shared memory algorithm for adding n numbers

Output $= \sum_{i=0}^{n-1} A(i)$ in $A(0)$

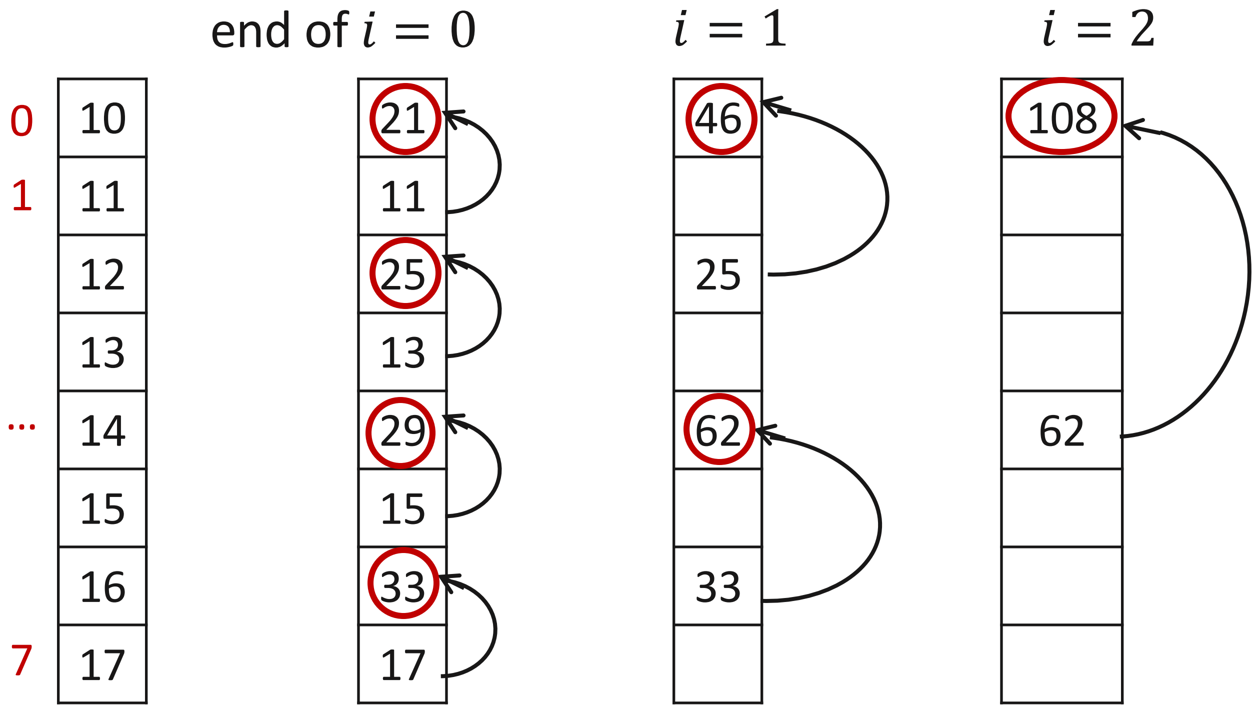


Adding on PRAM (2)

Key Idea

For $n = 8$

i = iteration #



Active processors = $\{0, 2, 4, 6\}$ $\{0, 4\}$ $\{0\}$

Processor index = $\{0, 1, 2, 3\} \times 2$ $\{0, 1\} \times 4$ $\{0\} \times 8$

Adding on PRAM (3)

Example: $n = 8$

Active processors

iteration # (i) → (time)

Processor # (j) ↓

j \ i	0	1	2
0	✓	✓	✓
2	✓		
4	✓	✓	
6	✓		

Total # of additions performed $= \frac{n}{2} + \frac{n}{4} + \cdots 1$

$= n - 1$

$=$ Total # of additions performed by a serial program

Adding on PRAM (4)

- Algorithm

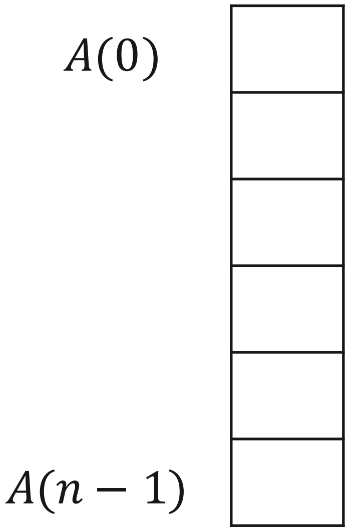
Program in processor $j, 0 \le j \le n-1$

1. Do $i = 0$ to $\log_2 n - 1$
2. If $j = k \cdot 2^{i+1}$, for some $k \in N$
   then $A(j) \leftarrow A(j) + A(j + 2^i)$
3. end

Note:

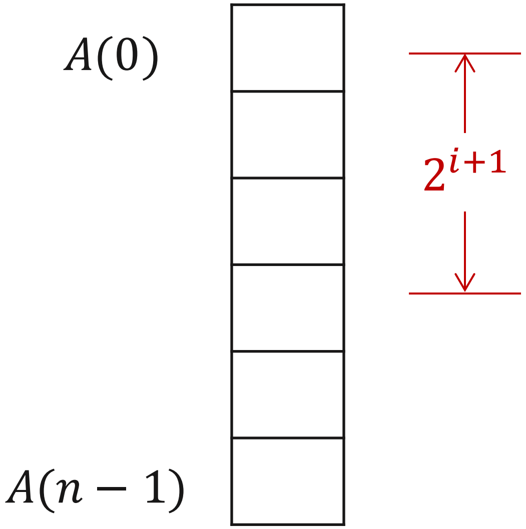
A is shared among all the processors
Synchronous operation [For ex. all the processors execute instruction 2 during the same cycle, $\log_2 n$ time]
N = set of natural numbers = {0, 1, …}

Parallel time = $O(\log n)$



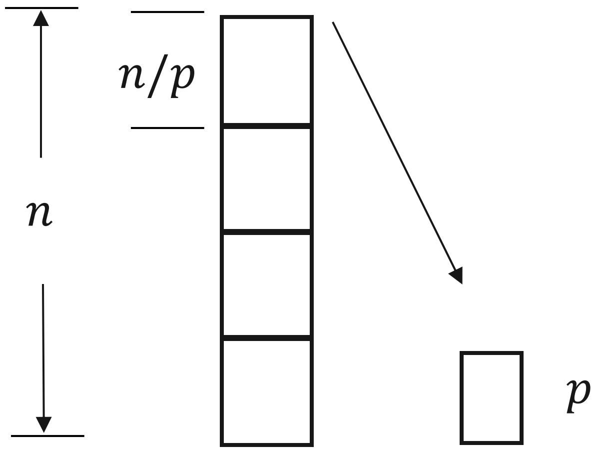
Adding on PRAM (5)

- i^{th} iteration:
  - data which are 2^{i+1} distance apart added
  - $\frac{n}{2^{i+1}}$ partial results produced



Addition on PRAM, $p < n$ processors

1. Add within block of size $\frac{n}{p}$
2. Apply Algorithm 1



T_p = Parallel Time = $O(\frac{n}{p} + \log p)$

T_s = Serial Time = $O(n)$

Synchronous and Asynchronous

Synchronous (e.g. PRAM)

Clock			
0	$y \leftarrow 0$	0	$x \leftarrow 0$
1-500	Do $i = 0$ to 499 $y \leftarrow y + A(i)$ End	1-500	Do $i = 500$ to 999 $x \leftarrow x + A(i)$ End
501	$y \leftarrow x + y$	501	STOP
502	OUTPUT y		
503	STOP		

Asynchronous (e.g. Pthreads)

$y \leftarrow 0$	$x \leftarrow 0$
Do $i = 0$ to 499	Do $i = 500$ to 999
$y \leftarrow y + A(i)$	$x \leftarrow x + A(i)$
End	End
Barrier	Barrier
$y \leftarrow x + y$	

Addition: Pthreads model?

Instruction execution NOT synchronized

Thread *j*

1. Do $i = 0$ to $\log_2 n - 1$
2. If $j = k \cdot 2^{i+1}$, for same $k \in N$
   then $A(j) \leftarrow A(j) + A(j + 2^i)$
3. BARRIER
4. end

Number of active threads in iteration $i = \frac{2^n}{2^{i+1}}$

Complete each iteration before proceeding to next iteration

Within each iteration threads may execute asynchronously

Correct output?

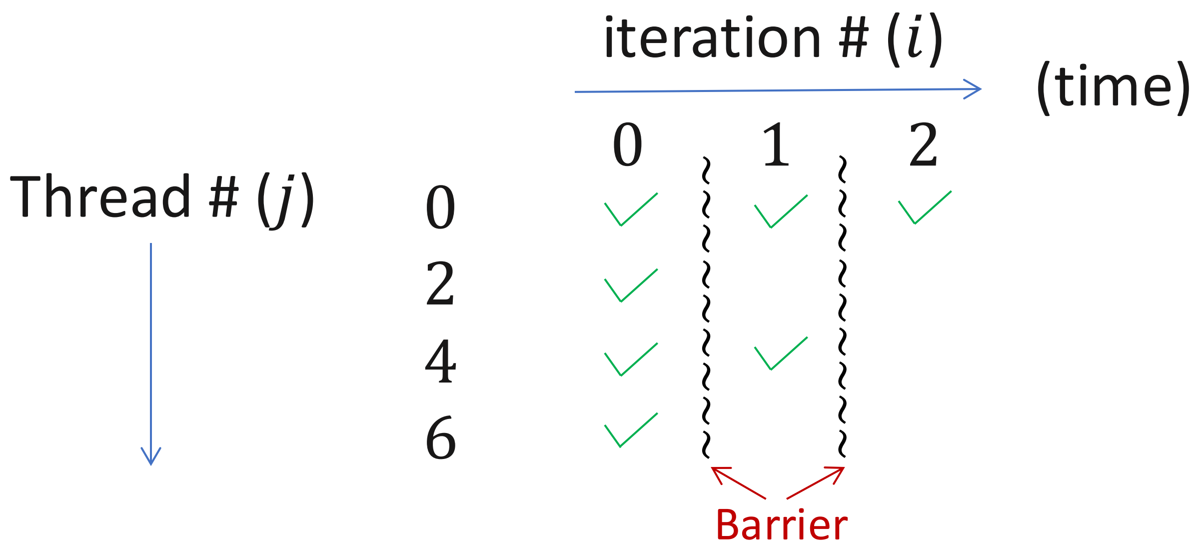
How to measure time?

Addition using Pthreads

Example: $n = 8$

Total # of threads = n

Active threads (threads doing arithmetic operations)



Addition using Pthreads, $\boldsymbol{p} < \boldsymbol{n}$ threads (1)

Each thread is allocated ${}^{n}/_{p}$ data values

Thread $\boldsymbol{j}$ $\quad 0 \le j < p$

Compute $B(j) \leftarrow$ sum of values allocated to it

BARRIER

i is a local variable to each thread

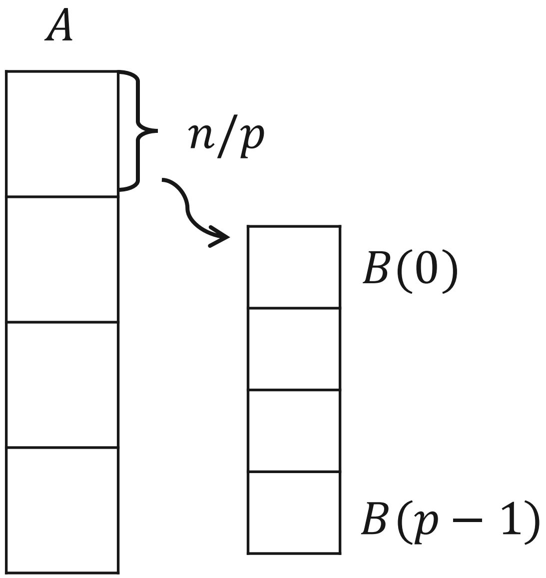
Do $i = 0$ to $\log_2 p - 1$

If $j = k \cdot 2^{i+1}$, for some $k \in N$

then $B(j) \leftarrow B(j) + B(j + 2^i)$

BARRIER

end



Addition using Pthreads, $\boldsymbol{p < n}$ threads (2)

- There are p threads
- All threads execute in parallel
- Correct output is produced for all inputs independent of thread execution speed

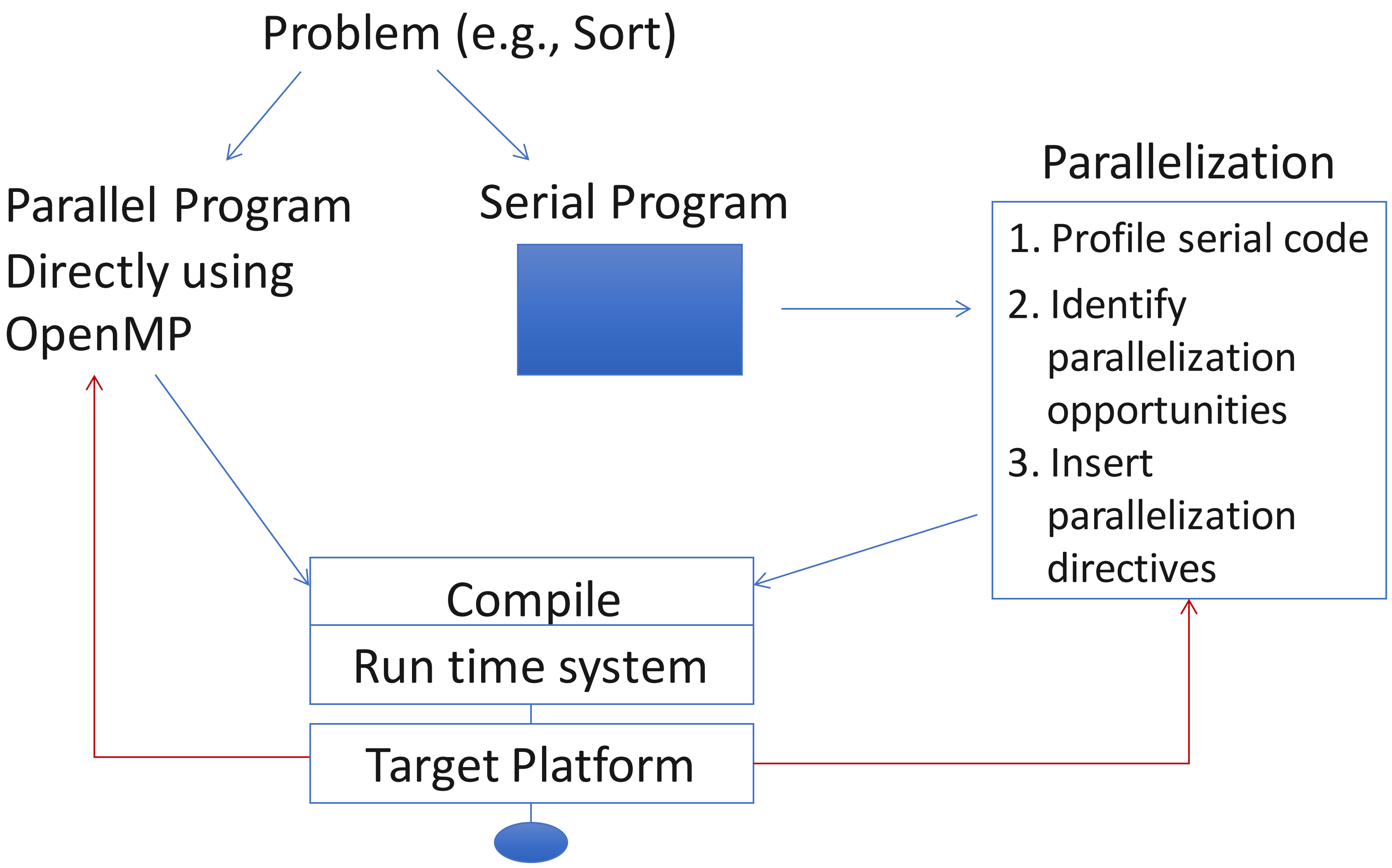
Note: Threads can execute asynchronously

Variable access latency to shared variables okay

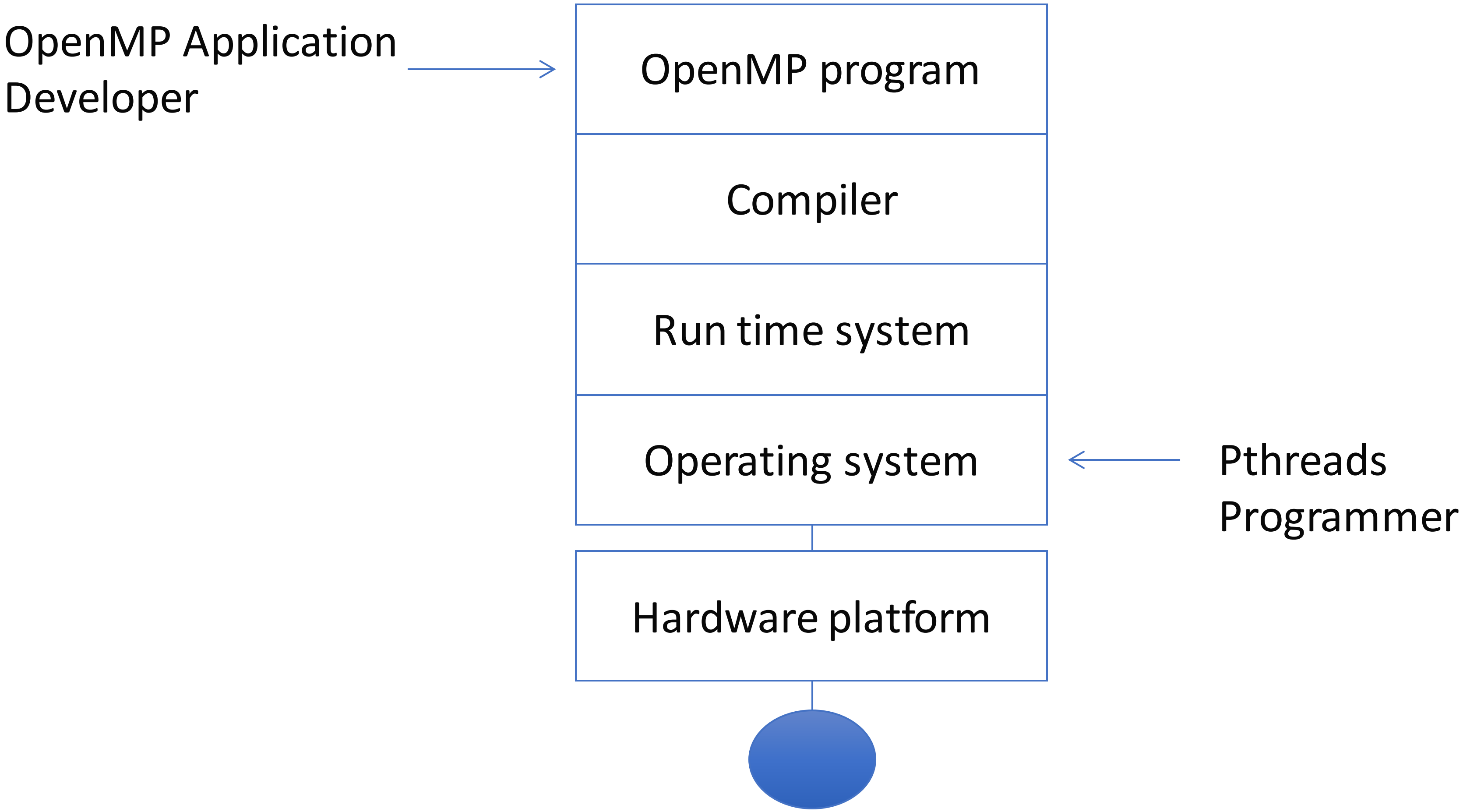
OpenMP (Open Multi-Processing)

- Application programming interface (API) for **shared memory parallel** programming
- A portable (and scalable) programming model (can be used with C, C++, Fortran and various parallel platforms)
- Outcome of standardization efforts
- Consists of compiler directives, library routines, and environment variables
- Higher level of abstraction than Pthreads

Two Uses of OpenMP



OpenMP vs. Pthreads



OpenMP Programming Model (1)

- Shared memory, threads based parallelism
- Explicit parallelism
  - Explicit (not automatic) programming model, offering the programmer full control over parallelization
  - Parallelization can be taking a serial program and inserting compiler directives
- Underlying hardware may or may not provide hardware support for shared memory

OpenMP Programming Model (2)

- Directive based parallel programming

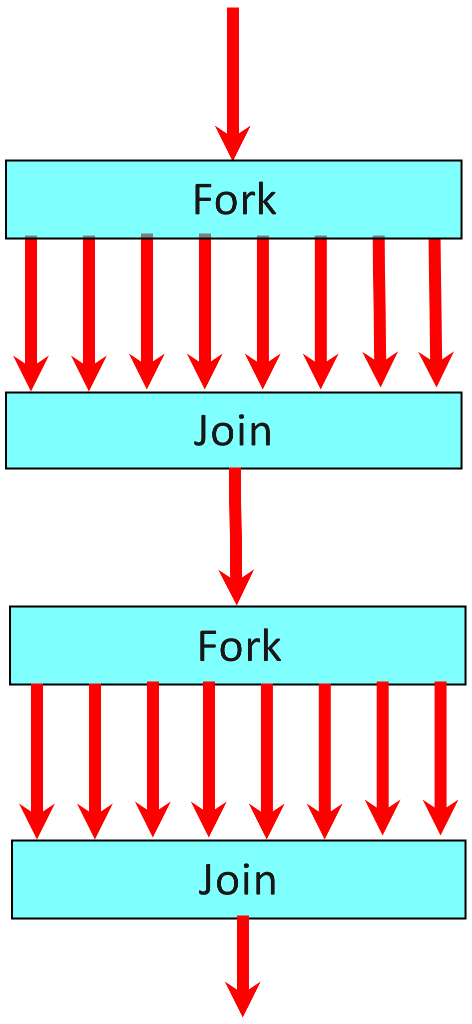
```
#pragma omp directive [clause list]
/* structured block */
```

Structured block = a section of code which is grouped together

- OpenMP directives are not part of a programming language
  - Can be included in various programming languages (e.g., C, C++, Fortran)

OpenMP Programming Model (3)

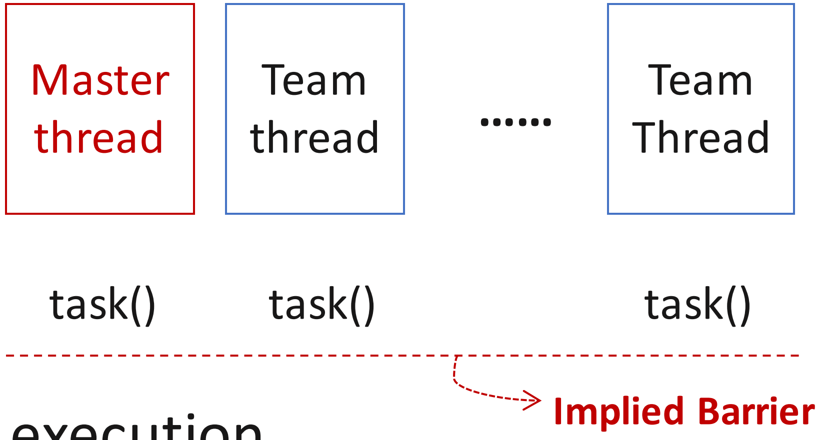
- Fork - Join model:
  - **Fork**: The master thread creates a team of parallel threads
  - **Join**: When the team threads complete the statements in the parallel region, they synchronize and terminate, leaving only the master thread



OpenMP Directives (1)

- OpenMP executes serially until **parallel** directive
- **parallel** region construct
  - Same code will be executed by multiple threads
  - Fundamental OpenMP parallel construct
  - Example

```
#pragma omp parallel [clause list]
{
        task();
}
```



- Clause list specifies parameters of parallel execution
  - # of threads, private variables, shared variables

OpenMP Directives (2)

- Work-Sharing Constructs
  - **Divide** the execution of the enclosed code region among the members
  - Implied barrier at the end of a work sharing construct
  - Types of Work-Sharing Constructs:
    - **for** - shares iterations of a loop across the team. Represents a type of "data parallelism".
    - **sections** - breaks work into separate, discrete sections. Each section is executed by a thread. Represents a type of "functional parallelism".

OpenMP Directives (3)

Loop parallelism

- Most programs have loops
- Rich source for parallelization/optimization
- 90/10 rule
- Many scheduling strategies and compile time and run time optimizations

OpenMP Directives (4)

for directive example: $N \times N$ matrix multiplication, $C = A \times B$

Serial

```
for (i=0; i<N; i++) {
    for (j=0; j<N; j++) {
        c(i,j) = 0;
        for (k=0; k<N; k++)
            c(i, j) += a(i, k) ∗ b(k, j);
    }
}
```

Static scheduling of loops

```
#pragma omp parallel num_threads (4)
  #pragma omp for schedule(static)
  for (i=0; i<N; i++) {
     for (j=0; j<N; j++) {
        c(i,j) = 0;
        for (k=0; k<N; k++)
           c(i, j) += a(i, k) ∗ b(k, j);
     }
  }
```

Static scheduling splits the iteration space into equal chunks of size (specified in the clause list) and assigns them to threads in a round-robin fashion

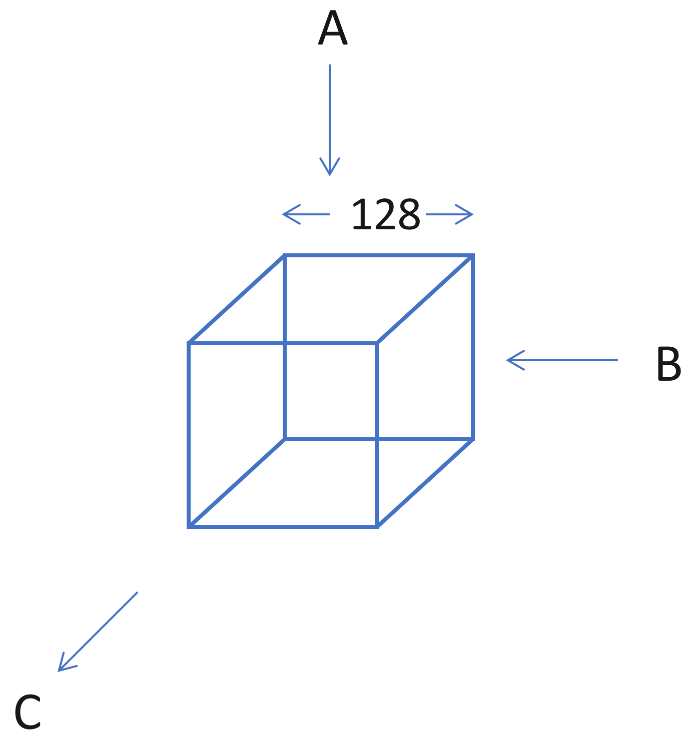
OpenMP Directives (5)

Iteration Space

3 level nested loop for matrix multiplication

dim = 128

```
for (i = 0; i < dim; i++) {
      for (j = 0; j < dim; j++) {
           c(i,j) = 0;
           for (k = 0; k < dim; k++)
                 c(i, j) += a(i, k) * b(k, j);
      }
  }
```



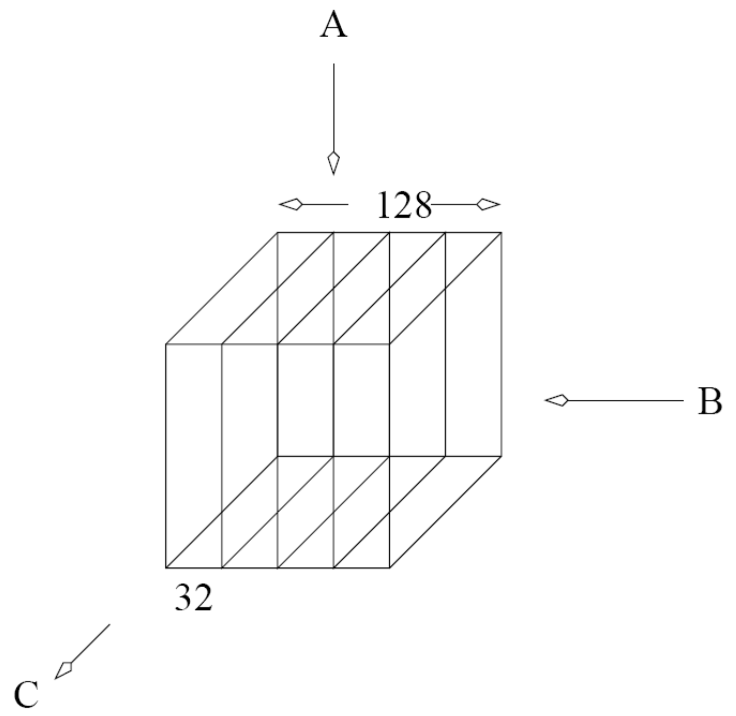
OpenMP Directives (5)

Iteration Space

3 level nested loop for matrix multiplication

dim = 128, 4 threads

```
#prama omp parallel   num_threads (4)
    #pragma omp for schedule(static)
    for (i = 0; i < dim; i++) {
        for (j = 0; j < dim; j++) {
            c(i,j) = 0;
            for (k = 0; k < dim; k++)
                c(i, j) += a(i, k) * b(k, j);
        }
    }
```



Note: Iteration space can be partitioned in many ways ← compiler optimizations

OpenMP Directives (7)

Scheduling iteration space

$$\text{Iterations} \xrightarrow{\text{map}} \text{threads}$$

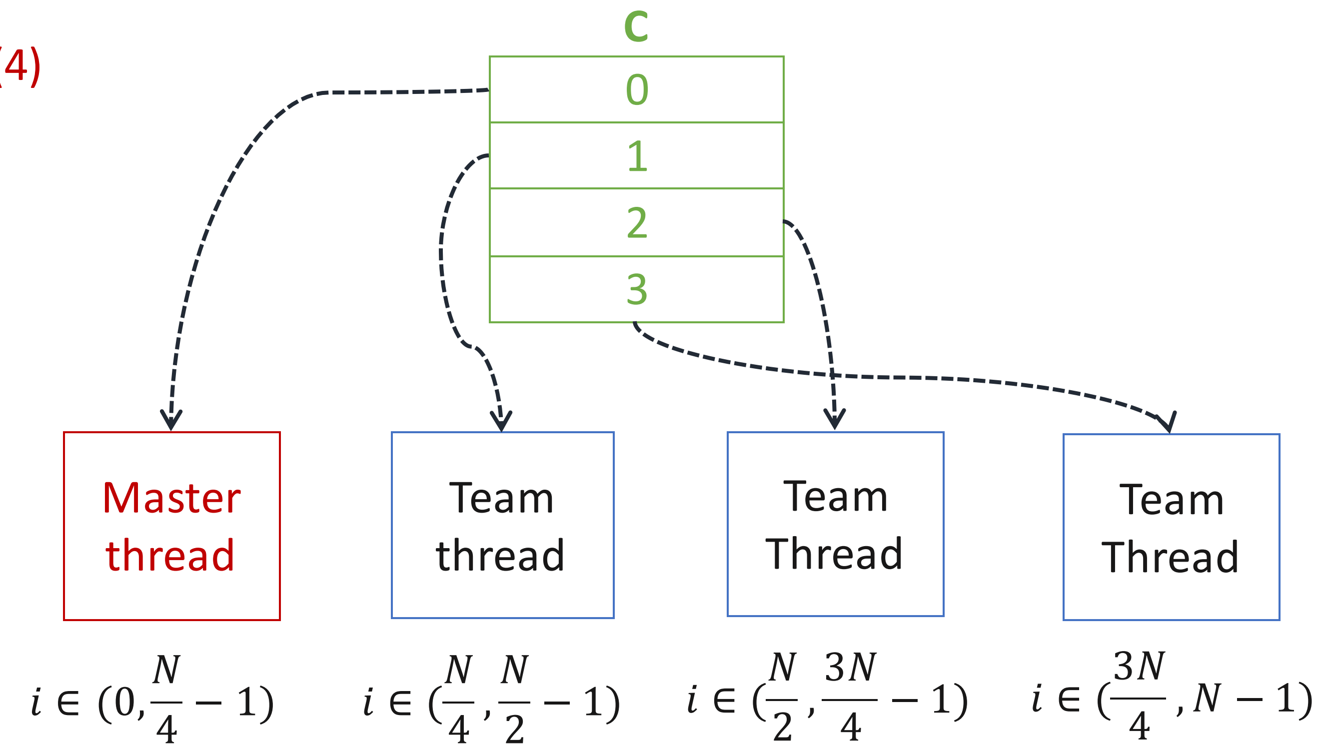
- **Static**: partition into equal sized chunks, map at compile time
- **Dynamic**: partition into equal sized chunks, run time system assigns them to threads (for ex. as the threads become idle)
- **Guided**: Vary the chunk size as the (loop) computation proceeds to ensure load balance among the threads
- **Runtime**: scheduling decisions left to the run time system

OpenMP Directives (8)

- **for** directive example (Data/Loop Parallelism):

  $N \times N$ matrix multiplication: $C = A \times B$

```
#pragma omp parallel num_threads(4)
{
        #pragma omp for
                for (i=0; i< N; i++)
                    Calculate_block()
}
```

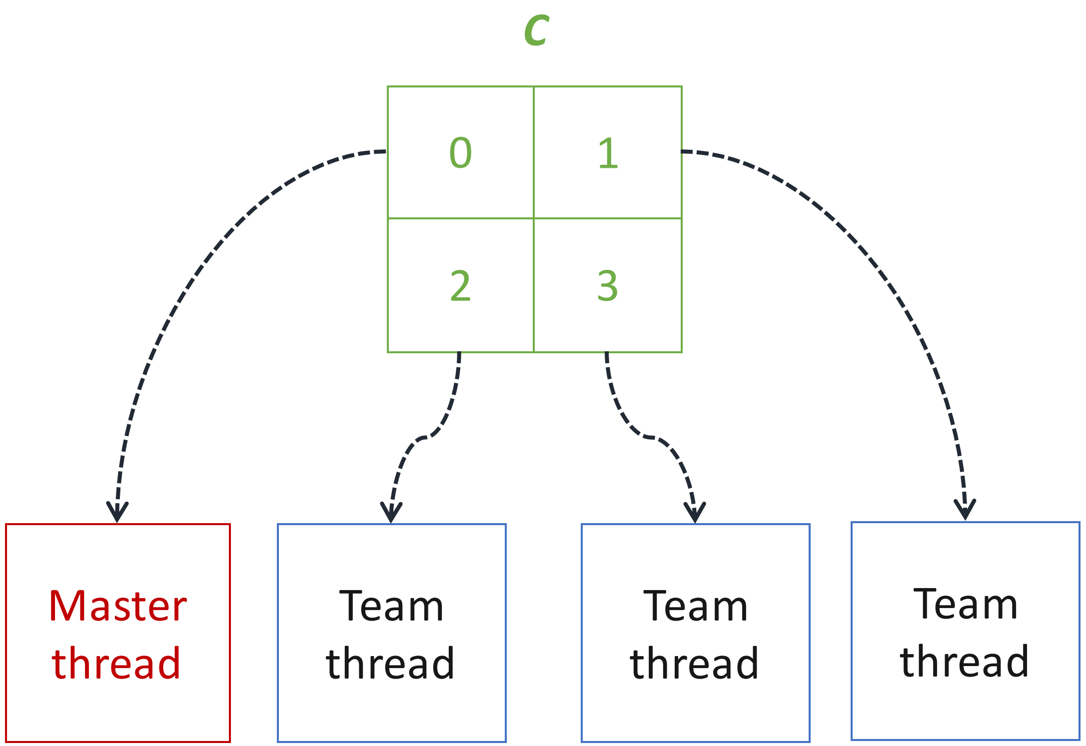


OpenMP Directives (9)

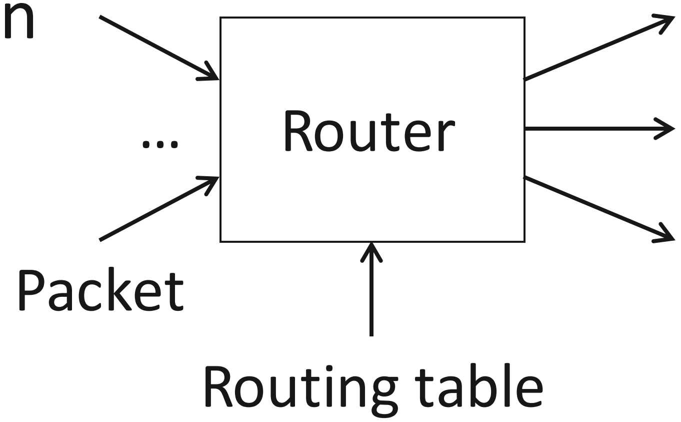
- **sections** example (Task Parallelism):

  $N \times N$ matrix multiplication: $C = A \times B$

```
#pragma omp parallel num_threads(4)
{
    #pragma omp section
      Caculate_block0;
    #pragma omp section
      Caculate_block1;
    #pragma omp section
      Caculate_block2;
    #pragma omp section
      Caculate_block3;
}
```



Example (1)

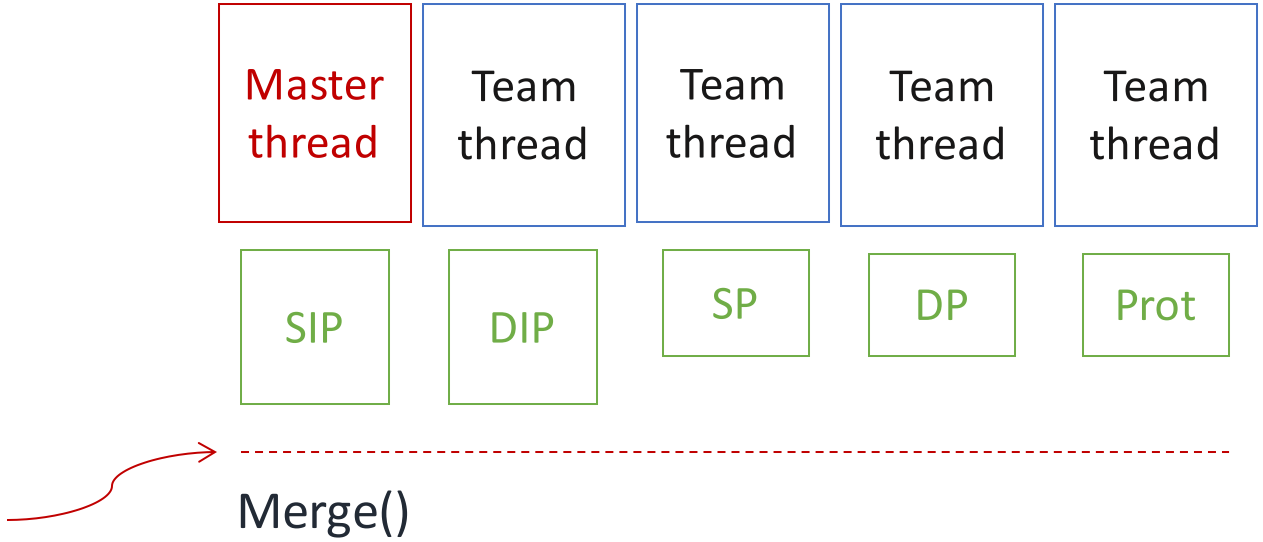


Source IP	Destination IP	Source Port	Destination Port	Protocol	Action
175.77.88.155/32	119.16.158.230/32	123 - 123	0 - 65535	TCP	Act 0
152.175.65.32/28	39.240.26.229/32	0 - 65535	750 - 760	TCP	Act 1
17.21.12.0/23	224.0.0.0/5	0 - 65535	0 - 65535	UDP	Act 2

Example (2)

```
#pragma omp parallel num_threads (5)
 {
      #pragma omp section
          search_SIP();
      #pragma omp section
          search_DIP();
      #pragma omp section
          search_SP();
      #pragma omp section
          search_DP();
      #pragma omp section
          search_Prot();
      #pragma omp barrier
      #pragma omp single
         Merge();
 }
```

Example: 5-field packet classification



More Examples (1)

Sort an N-element Array

#pragma omp parallel num_threads (2)

{

#pragma omp section

Sort(0, $\frac{N}{2} - 1$);

#pragma omp section

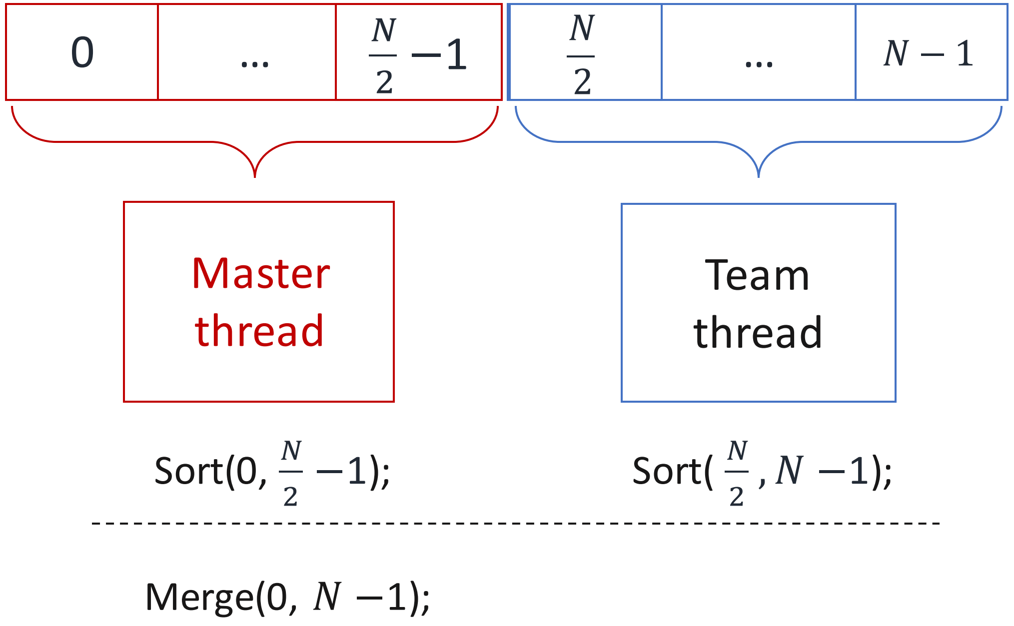
Sort($\frac{N}{2}$, $N - 1$);

#pragma omp barrier

#pragma omp single

Merge(0, $N - 1$);

}



More Examples (2)

Find min. element in an N-element Array

Serial Program

min = A[0];

For i = 1 to $N-1$

if (min > A[i])

min = A[i];

End

OpenMP Program

```
#pragma omp parallel num_threads(2)
{
    #pragma omp section
        min1 = Min (0, N/2 −1);
    #pragma omp section
        min2 = Min (N/2, N −1);
    #pragma omp barrier
    #pragma omp single
        min = Min (min1, min2);
}
```

Summary

OpenMP

- Simple high level abstraction for shared address space programming
- Directives to the compiler
- Specify: # of threads, conditions for parallelization, local/global variables
- Serial code → incremental parallelization
- Loop parallelization, task parallelization
- Static analysis → parallel code that executes with the aid of a run time system
- Run time system: data management, layout, communication, mapping, optimizations, …

Backup Slides

Resources

- Slides from CMU introducing multi-core architecture, thread-level parallelism, cache & cache coherence, etc.
  - https://www.cs.cmu.edu/~fp/courses/15213-s06/lectures/27-multicore.pdf
- A recent publication, light reading on single-core and multi-core machines in intro section, comparisons and measurements are made on two real processors
  - https://aircconline.com/ijcsit/V10N1/10118ijcsit01.pdf
- A Professor's webpage talking about multi-core, parallelism, perf. model (very high-level)
  - https://www.cse.wustl.edu/~jain/cse567-11/ftp/multcore/#sec1
- A youtube video, talking about single & multi core architecture, Intel sandy bridge & AMD interlagos
  - https://www.youtube.com/watch?v=crZwPhNjNiU