# Intro to OpenMP (shared memory programming)

## Lecture 5

Sunita Chandrasekaran
Associate Professor, University of Delaware
PDC Summer School
Aug 2023

# 3 Ways to Program CPU-GPU Heterogeneous Architecture

**Applications**

| Standard Languages & Libraries | Directives | Platform-Specific Programming Languages |
| --- | --- | --- |
| Kokkos, alpaka, Raja | OpenACC<br>OpenMP | CUDA, OpenCL |
| Accelerated Standard C++ and Fortran | Incremental Performance Optimization | Maximize Performance for Most Important Kernels, e.g., with CUDA or OpenCL |

**Interoperability Needed Across Models**

# OpenACC/OpenMP syntax

C/C++

```
#pragma acc directive clauses
<code>
```

C/C++

```
#pragma omp directive clauses
<code>
```

Fortran

```
!$acc directive clauses
<code>
```

Fortran

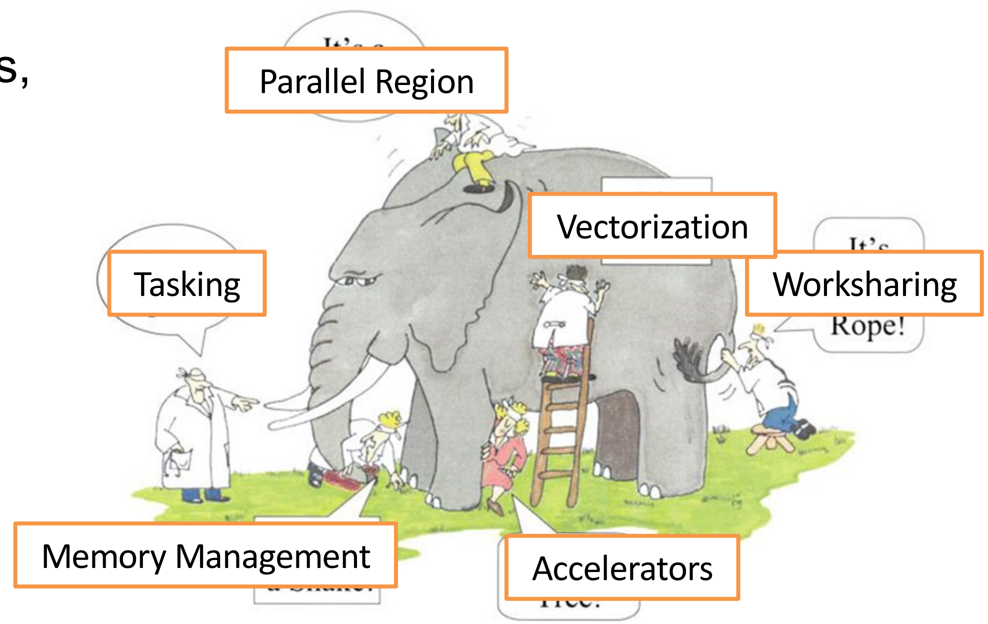
```
!$omp directive clauses
<code>
```

- A ***pragma*** in C/C++ gives instructions to the compiler on how to compile the code. Compilers that do not understand a particular pragma can freely ignore it.
- A ***directive*** in Fortran is a specially formatted comment that instructs the compiler to compile the code.
- ***"acc"*** informs the compiler that what will come is an OpenACC directive
- ***"omp"*** informs the compiler that what will come is an OpenMP directive
- ***Directives*** are commands in OpenACC for altering our code.
- ***Clauses*** are specifiers or additions to directives.

# What is OpenMP?

- De-facto standard Application Programming Interface (API) to write <u>shared memory parallel</u> applications in C, C++, and Fortran
- Consists of Compiler Directives, Runtime routines and Environment variables

- Version 5.0 has been released at SC 2018
- Version 5.2 has been released at SC 2021

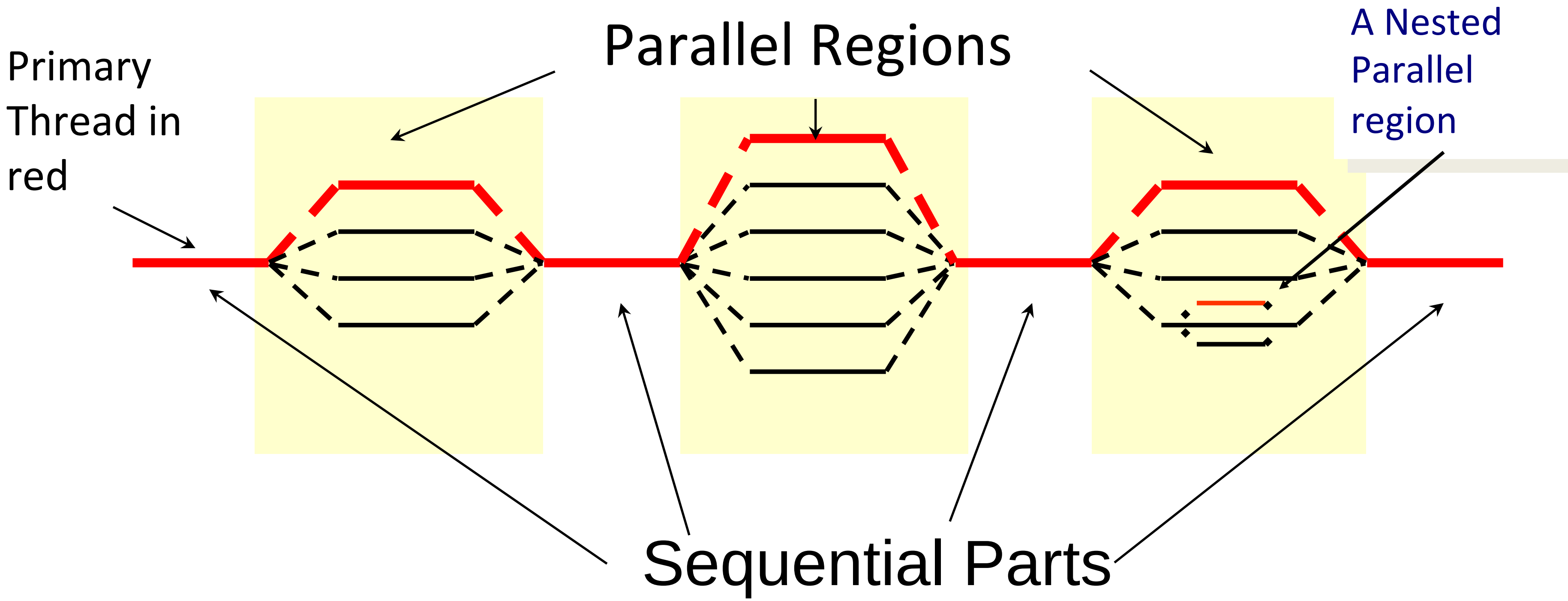



Slide Courtesy: Christian Terboven, Michael Klemm, Bronis R. de Supinski

# OpenMP Programming Model

## Fork-Join Parallelism:

- **The Primary thread** spawns a **team of threads** as needed.
- Parallelism added incrementally until performance goals are met: i.e. the sequential program evolves into a parallel program.



# The Worksharing Constructs

- *The work is distributed over the threads*
- *Must be enclosed in a parallel region*
- *Must be encountered by all threads in the team, or none at all*
- *No implied barrier on entry*
- *Implied barrier on exit (unless the nowait clause is specified)*
- *A work-sharing construct does not launch any new threads*

```
#pragma omp for
{
    ....
}
```

```
#pragma omp sections
{
        ....
}
```

```
#pragma omp single
{
        ....
}
```

Slide Courtesy: Christian Terboven, Michael Klemm, Bronis R. de Supinski

# Example: Hello world

- Write a multithreaded program where each thread prints “hello world”.

```
void main()
{



     int ID = 0;
     printf(“ hello(%d) ”, ID);
     printf(“ world(%d) \n”, ID);

}
```

# Example: Hello world Solution

Tell the compiler to pack code into a function, fork the threads, and join when done …

```
#include "omp.h"
void main()
{
#pragma omp parallel
 {
    int ID = omp_get_thread_num();
    printf(" hello(%d) ", ID);
    printf(" world(%d) \n", ID);
  }
}
```

OpenMP include file

Parallel region with default number of threads

End of the Parallel region

Runtime library function to return a thread ID.

What would actually be printed from this parallel program?

# Example output: Hello world

Tell the compiler to pack code into a function, fork the threads, and join when done …

```
#include "omp.h"
void main()
{
#pragma omp parallel
 {
    int ID = omp_get_thread_num();
    printf(" hello(%d) ", ID);
    printf(" world(%d) \n", ID);
  }
}
```

OpenMP include file

Parallel region with default number of threads

End of the Parallel region

Runtime library function to return a thread ID.

**Sample Output:**

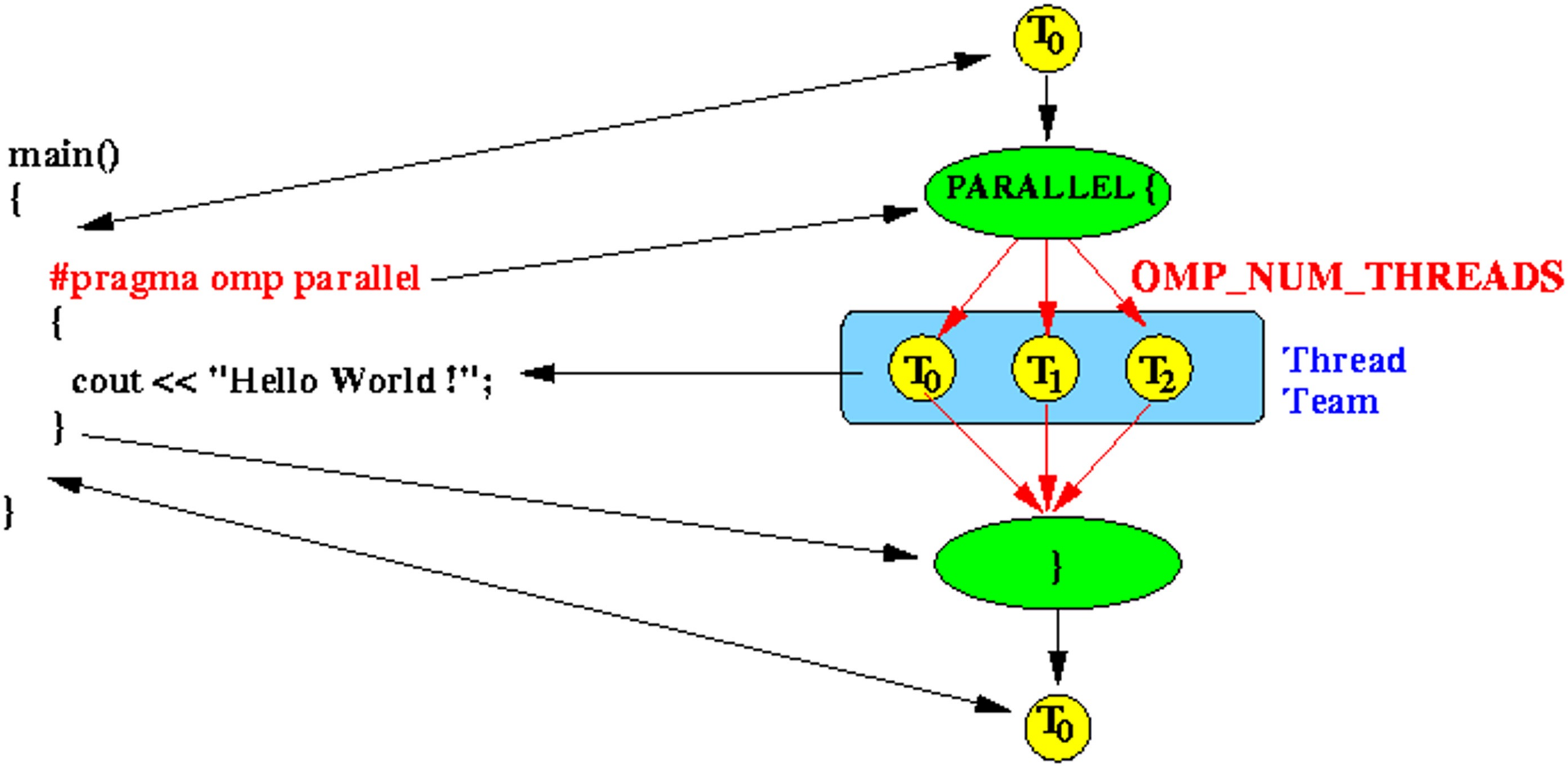
hello(1) hello(0) world(1)

world(0)

hello (3) hello(2) world(3)

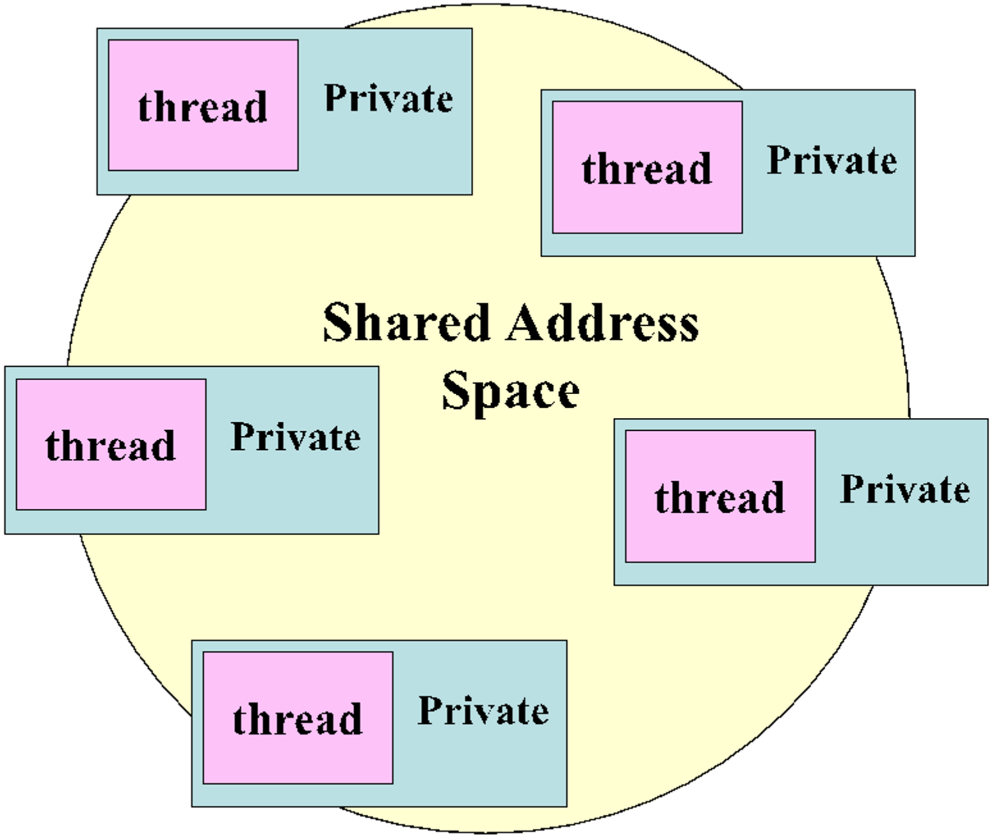
world(2)

# What is happening under the hood?



# A shared memory program

- **An instance of a program:**
  - One process and lots of threads.
  - Threads interact through reads/writes to a shared address space.
  - OS scheduler decides when to run which threads … interleaved for fairness.
  - Synchronization to assure every legal order results in correct results.



# Data Scoping Clauses

One can selectively change storage attributes for variables using the following clauses

- Private
- Shared
- Default (none)

For example ….

```
#pragma omp parallel for
default(shared) private(a, b)
```

# Private Clause

- The private(list) clause declares that all the variables in list are private.
- **b** is a private variable. When a variable is declared private, OpenMP replicates this variable and assigns its local copy to each thread.
- *Note – loop iteration variable is private by default*

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

# Private Clause

For example:

```
int p = 0;
// the value of p is 0

#pragma omp parallel private(p)
{
    // the value of p is undefined
    p = omp_get_thread_num();
    // the value of p is defined
    ...
}
// the value of p is undefined
```

- The behavior of private variables is sometimes unintuitive.
- Let us assume that a private variable has a value before a parallel region.
- However, the value of the variable at the beginning of the parallel region is undefined.
- Additionally, the value of the variable is undefined also after the parallel region.

# Shared Clause

- The default (shared) clause sets the data-sharing attributes of all variables in the construct to shared.
- Shared variables where a single copy of the variable exist and all threads access that single copy
- a, b, c and n are shared variables.

```
int a, b, c, n;
...

#pragma omp parallel for default(shared)
for (int i = 0; i < n; i++)
{
    // using a, b, c
}
```

# Shared Clause

- Another usage of default(shared) clause is to specify the data-sharing attributes of the majority of the variables and then additionally define the private variables.

```
int a, b, c, n;

#pragma omp parallel for default(shared) private(a, b)
for (int i = 0; i < n; i++)
{
    // a and b are private variables
    // c and n are shared variables
}
```

# Implicit Rules

- How many variables do you see?

4

- The data-sharing attribute of variables, which are declared outside the parallel region, is usually shared. What are those variables?

n, a

- The loop iteration variables, however, are private by default.

i

- The variables which are declared locally within the parallel region are private.

b

```
int i = 0;
int n = 10;
int a = 7;

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

# Default (none)

- The default(none) clause forces a programmer to explicitly specify the data-sharing attributes of all variables.
- A distracted programmer might write the following piece of code

```
int n = 10;
std::vector<int> vector(n);
int a = 10;

#pragma omp parallel for default(none) shared(n, vector)
for (int i = 0; i < n; i++)
{
    vector[i] = i * a;
}
```

# Default (none)

- And get the following errors

```
error: 'a' not specified in enclosing parallel
        vector[i] = i * a;
                        ^
error: enclosing parallel
    #pragma omp parallel for default(none) shared(n, vector)
            ^
```

# Default (none)

- The reason for the unhappy compiler is that the programmer used default(none) clause and then she/he forgot to explicitly specify the data-sharing attribute of a.
- The correct version of the program would be

```
int n = 10;
std::vector<int> vector(n);
int a = 10;

#pragma omp parallel for default(none) shared(n, vector, a)
for (int i = 0; i < n; i++)
{
    vector[i] = i * a;
}
```

# Some practices to remember

- always write parallel regions with the default(none) clause
  - Compiler might give you an error, but then that will make you revisit your code
- declare private variables inside parallel regions whenever possible
  - This guideline improves the readability of the code and makes it clearer.

# Data Scoping Clauses

One can selectively change storage attributes for variables using the following clauses

- Private
- Shared
- Default (none)
- Lastprivate
- Firstprivate

For example ....

```
#pragma omp parallel for default(shared) private(a, b)
```

# Lastprivate

- firstprivate and lastprivate are just different variations of *private*
  - *lastprivate* Keep the last value of the variable, after the parallel region
  - When a lastprivate variable is passed to a parallelized for loop,
    - threads work on uninitialized copies but,
    - at the end of the parallelized for loop, the thread in charge of the last iteration sets the value of the original variable to that of its own copy.

**#pragma omp parallel for lastprivate(val)**

**(you will use 'for' if you have a for loop, you won't need the 'for' if you are not using lastprivate in a for loop)**

# firstprivate

- Firstprivate
  - a clause that contains the variables that each thread in the OpenMP parallel region will have an identical copy of
  - These copies are INITIALIZED with the value of the original variable passed to the clause
    - By contrast, private variables DO NOT
  - While the threads work on initialized copies, whatever modification is made to their copies is not reflected onto the original value of that variable after the parallel region

**#pragma omp parallel for firstprivate(val)**

**(you will use 'for' if you have a for loop, you won't need the 'for' if you are not using lastprivate in a for loop)**

# What we have covered so far with OpenMP

Directives

- Parallel
- Parallel for (work sharing directive)

Data Scoping clauses

- Private
- Shared
- Default (none)
- Lastprivate
- Firstprivate

# What we have covered so far with OpenMP

Directives

- Parallel
- Parallel for (work sharing directive)

Data Scoping clauses

- Private
- Shared
- Default (none)
- Lastprivate
- Firstprivate

Other Clauses

- Reduction

Synchronization Constructs

- Critical
- Atomic
- Barrier

# Reduction clause

Parallel tasks often produce some quantity that needs to be summed or otherwise combined.

| | | | | | |
|---|---|---|---|---|---|
| #pragma | omp | parallel | for | reduction | (+: sum) |

# Reduction operators

**C/C++ Reduction Operands**

| Operator | Initial value |
|---|---|
| + | 0 |
| * | 1 |
| - | 0 |
| & | ~0 |
| \| | 0 |
| ^ | 0 |
| && | 1 |
| \|\| | 0 |

# OpenMP reduction clause

```
/* C/C++ Example */

#pragma omp parallel for reduction (+: sum)
for(i=1; i<=n; i++){
        sum = sum + a(i)
}
```

# How do threads interact?

- OpenMP is a multi-threading, shared address model
  - Threads communicate by sharing variables
- Unintended sharing of data causes race conditions
  - race condition: when the program’s outcome changes as the threads are scheduled differently.
- To control race conditions
  - Use synchronization to protect data conflicts
- Synchronization is expensive so
  - Change how data is accessed to minimize the need for synchronization.

# OpenMP Synchronization constructs

- High level synchronization:
  - critical
  - Atomic
  - barrier
  - ordered
- Low level synchronization
  - flush
  - locks (both simple and nested)

# OpenMP Synchronization

- High level synchronization:
  - critical
  - Atomic
  - barrier
  - ordered
- Low level synchronization
  - flush
  - locks (both simple and nested)

# OpenMP Critical construct

```
#pragma omp critical
```

# OpenMP *critical* construct

- The *critical* construct provides a means to ensure that multiple threads do not attempt to update the same shared data simultaneously.
- The enclosed code block will be executed by only one thread at a time.

# Downside of critical construct

- Critical clause can severely slow down performance
  - due to serialization of the execution causing threads to "queue" to enter the critical region,
  - as well as introducing large lock-management overheads required to manage the critical region.

# What's the alternative?

```
#pragma omp parallel for
for ( int i = 0; i < Ni; i++ ) {

#pragma omp critical
        sum += array[i];
}


#pragma omp parallel for reduction(+:sum)
for ( int i = 0; i < Ni; i++ ) {
        sum += array[i];
}
```

# OpenMP *atomic* construct

- The *atomic* construct ensures
  - that a specific storage location is accessed atomically as it name suggests,
  - rather than exposing it to the possibility of multiple, simultaneous reading and writing threads that may result in indeterminate values.

# OpenMP atomic construct

| | | |
|---|---|---|
| #pragma | omp | atomic |

# Downside of atomic construct

- Performance
- Price is synchronization
- 2 threads must synchronize to avoid race conditions, a.k.a. threads are serialized
- Serialization of memory accesses disables parallelism

# What's the alternative?

```
#pragma omp parallel for
for ( int i = 0; i < Ni; i++ ) {

#pragma omp atomic
        sum += array[i];
}


#pragma omp parallel for reduction(+:sum)
for ( int i = 0; i < Ni; i++ ) {
        sum += array[i];
}
```

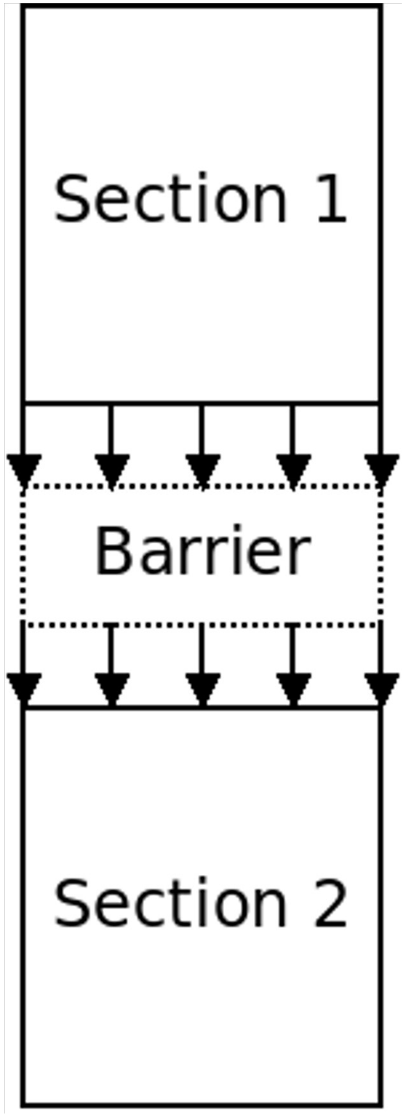
# So what's the difference?

critical vs atomic

- Atomic uses hardware instructions
- Atomic does not use lock/unlock on entering/exiting the line of code
- Lower overhead

# OpenMP barrier construct

- The barrier construct, which is a stand-alone directive, specifies an explicit synchronization barrier at the point at which the construct appears.
- The barrier applies to the innermost enclosing parallel region, forcing every thread that belong to the team of that parallel region to complete any pending explicit task.
- Only once all threads of that team satisfy this criterion will they be allowed to continue their execution beyond the barrier.



```c
int main(int argc, char* argv[])
{
    // Use 4 threads when we create a parallel region
    omp_set_num_threads(4);

    // Create the parallel region
    #pragma omp parallel
    {
        // Threads print their first message
        printf("[Thread %d] I print my first message.\n", omp_get_thread_num());

        // Make sure all threads have printed their first message before moving on.
        #pragma omp barrier

        // One thread indicates that the barrier is complete.
        #pragma omp single
        {
            printf("The barrier is complete, which means all threads have printed their first message.\n");
        }

        // Threads print their second message
        printf("[Thread %d] I print my second message.\n", omp_get_thread_num());
    }

    return EXIT_SUCCESS;
}
```

# OpenMP

Directives
- Parallel
- Parallel for (work sharing directive)

Data Scoping Clauses
- Private
- Shared
- Default (none)
- Lastprivate
- Firstprivate
- Reduction

Synchronization Constructs
- Critical
- Atomic
- Barrier

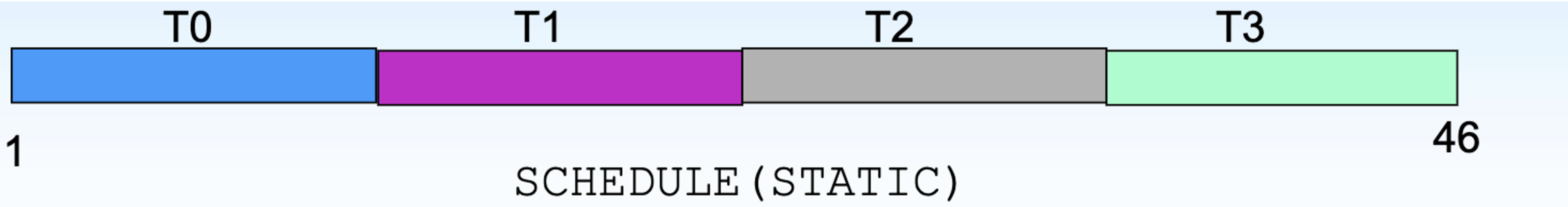
Scheduling clauses
- Static
- Dynamic
- Guided
- Auto
- Runtime variables

# Why scheduling matters?

- Improve distribution of work across threads available
- Address load imbalances and adjust work distribution
- With a goal to keep all processors busy for about the same amount of time and/or at best do not leave threads to be idle
- Access memory contiguously; offers better data locality

# Static Scheduling - Definition

- static[,chunk]: Distribute statically the loop iterations in batched of chunk size in a round-robin fashion.
- Statically - means that the distribution is done before entering the loop



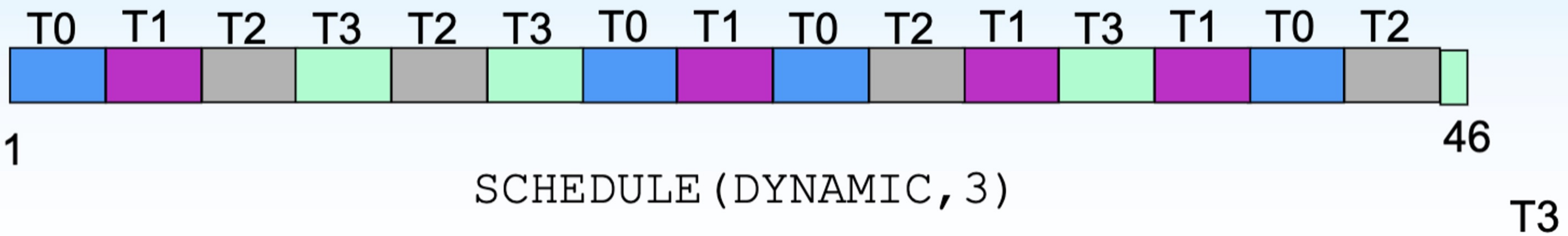
# Static Scheduling – A sample code

A static schedule can be non-optimal, however. This is the case when the different iterations take different amounts of time. Each loop iteration sleeps for a number of seconds equal to the iteration number:

```c
#include <unistd.h>
#include <stdlib.h>
#include <omp.h>
#include <stdio.h>

#define THREADS 4
#define N 16

int main() {
    #pragma omp parallel for schedule(static) num_threads(THREADS)
    for (int i = 0; i < N; i++) {
        /* wait for i seconds */
        sleep(i);

        printf("Thread %d has completed iteration %d.\n", omp_get_thread_num(), i);
    }

    /* all threads done */
    printf("All done!\n");
    return 0;
}
```

# Dynamic Scheduling - Definition

- dynamic[,chunk]: Distribute the loop iterations among the threads by batches of chunk size with a first-come-first-served policy, until no batch remains.
- If not specified, chunk is set to 1



# Dynamic Scheduling – A sample code

Here, OpenMP assigns one iteration to each thread. When the thread finishes, it will be assigned the next iteration that hasn't been executed yet.

```
#include <unistd.h>
#include <stdlib.h>
#include <omp.h>
#include <stdio.h>

#define THREADS 4
#define N 16

int main() {
    #pragma omp parallel for schedule(dynamic) num_threads(THREADS)
    for (int i = 0; i < N; i++) {
        /* wait for i seconds */
        sleep(i);

        printf("Thread %d has completed iteration %d.\n", omp_get_thread_num(), i);
    }

    /* all threads done */
    printf("All done!\n");
    return 0;
}
```

# Scheduling summary – Part 1

- The default for schedule is **implementation defined**.
  - On many environments it is static but can also be dynamic or could very well be auto.
- For loops where each iteration takes roughly equal time a.k.a balanced loops – what scheduling would you use?
  - static schedules work best, as they have little overhead.
- Choosing the best schedule depends on understanding your loop.

# Scheduling summary – Part 2

- For loops where each iteration can take very different amounts of time or varying workloads, what scheduling would you use?
  - dynamic schedules, work best as the work will be split more evenly across threads
- Specifying chunks, or using a guided schedule provide a trade-off between the two.
  - But beware that the first iteration might be the most expensive
- Choosing the best schedule depends on understanding your loop.

# Scheduling summary – Part 3

- When you have iterations taking an unpredictable amount of time, what scheduling kind would you use?
  - Dynamic
  - Need load balancing
- Downside of guided scheduling
  - Some threads would take excessive amount of time at the beginning and not well balanced in general