Parallel and Distributed Computing

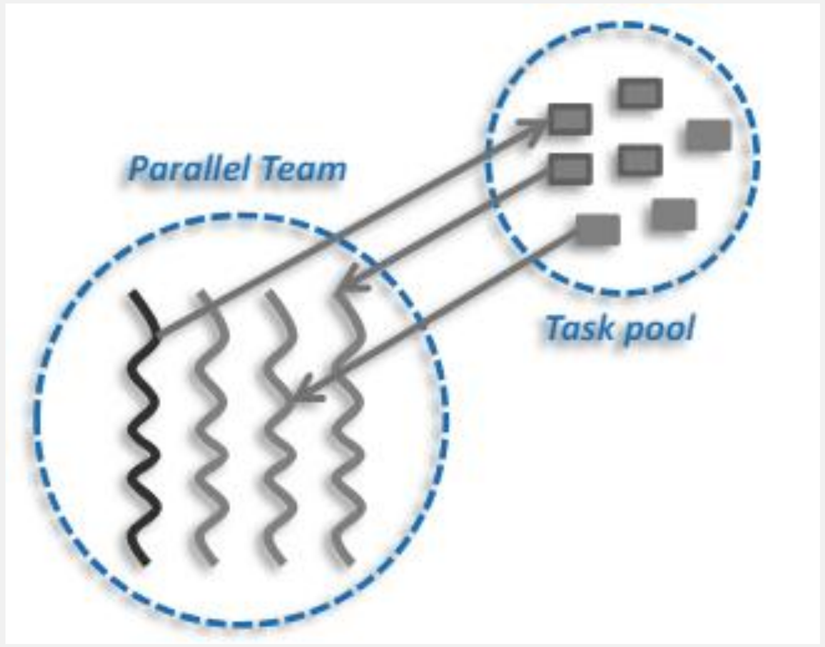
Dr. Ali Sayyed
Department of Computer Science
National University of Computer & Emerging Sciences

Shared Memory Programming Models

OpenMP

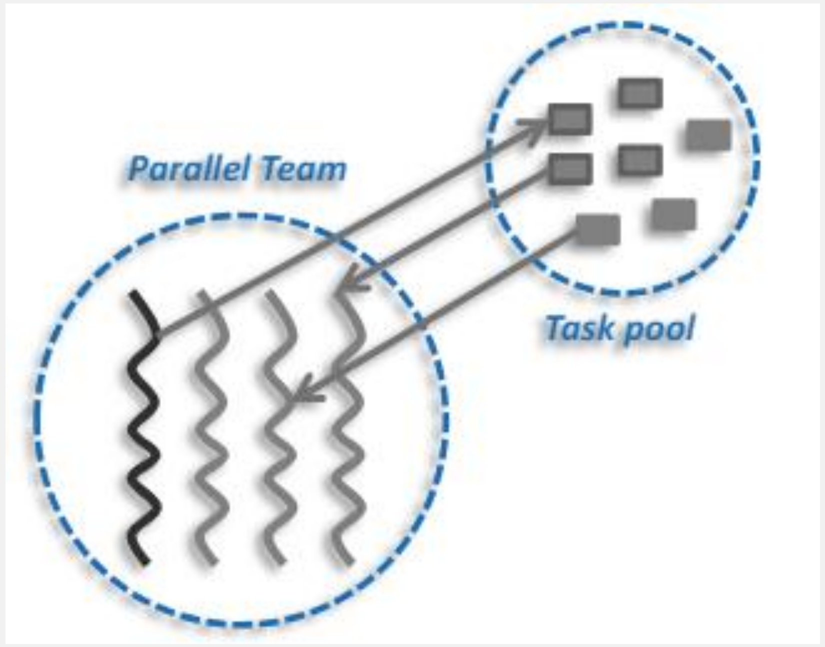
OpenMP Tasks

- In OpenMP, **`tasks`** provide a way to divide a program into discrete units of work, which can be executed independently by different threads.
- Particularly **useful for irregular problems** (problems without loops, unbounded loops, recursive algorithms, etc.) where the **workload distribution** might vary **dynamically**.
- Each task can be **executed as soon as** any **thread is available** to do so.



OpenMP Tasks

- **Task Creation:** Each time a thread encounters `#pragma omp task`, a new task is created.
- **Task Execution:** Tasks are placed in a task pool and can be picked up by any available thread.
- **Synchronization:** Use `#pragma omp taskwait` to make a thread wait until all child tasks have completed.



Example

```
void process(int i) {
    printf("Processing element %d by thread %d\n", i, omp_get_thread_num());
}

int main() {
    int n = 10;

    // Start parallel region
    #pragma omp parallel
    {

        #pragma omp single
        {
            for (int i = 0; i < n; i++) {
                #pragma omp task
                {
                    process(i);  // Each task processes one element
                }
            }
        }  // End of single
    }  // End of parallel region

    return 0;
}
```

```
Processing element 0 by thread 1
Processing element 3 by thread 2
Processing element 5 by thread 3
Processing element 1 by thread 1
```

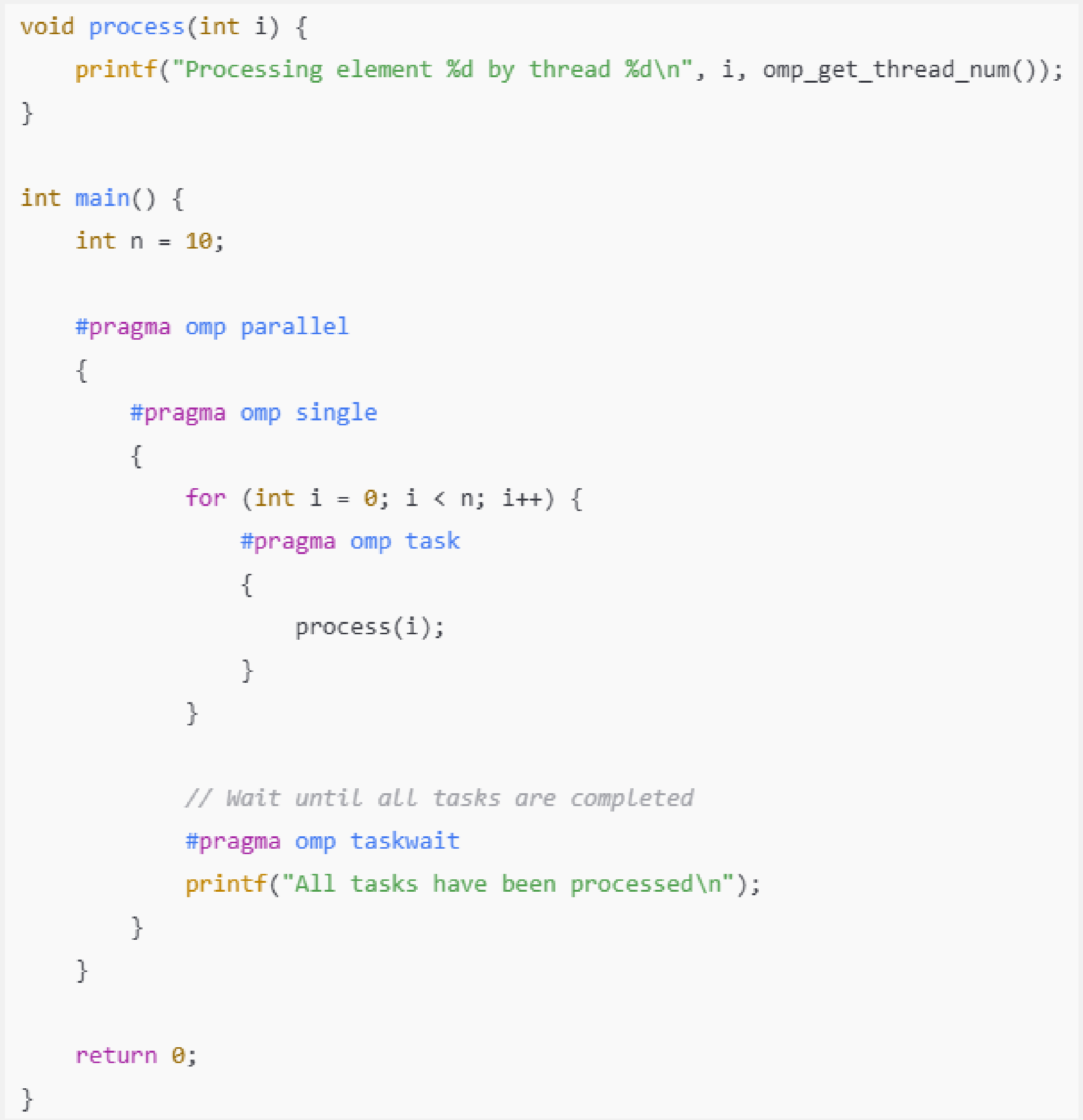
This starts a parallel region; Without this #pragma, thread #0 will have to do everything

Ensures that only one thread creates the tasks; Without this #pragma, each thread will create tasks

Defines a task for each iteration of the loop

when tasks are created inside a #pragma omp single region, any thread within the team of threads can be assigned to execute those tasks.

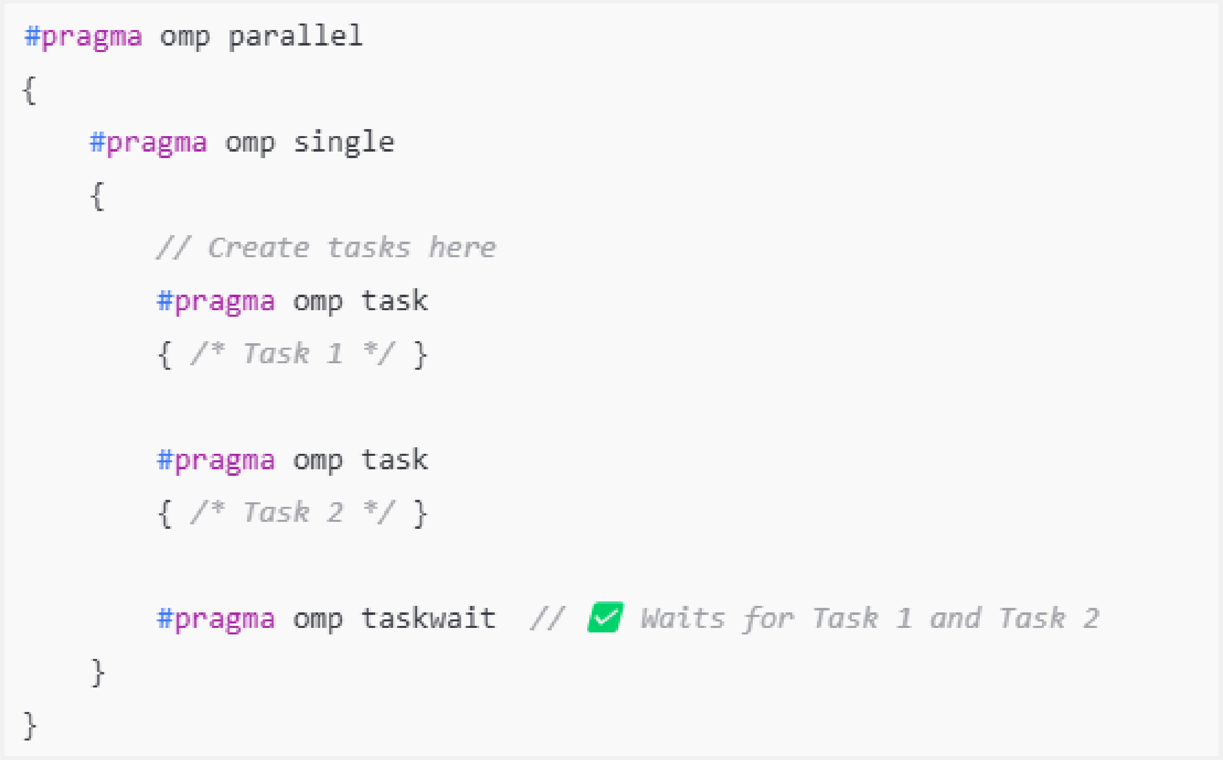
Example with `taskwait`



```c
void process(int i) {
    printf("Processing element %d by thread %d\n", i, omp_get_thread_num());
}

int main() {
    int n = 10;

    #pragma omp parallel
    {
        #pragma omp single
        {
            for (int i = 0; i < n; i++) {
                #pragma omp task
                {
                    process(i);
                }
            }

            // Wait until all tasks are completed
            #pragma omp taskwait
            printf("All tasks have been processed\n");
        }
    }

    return 0;
}
```

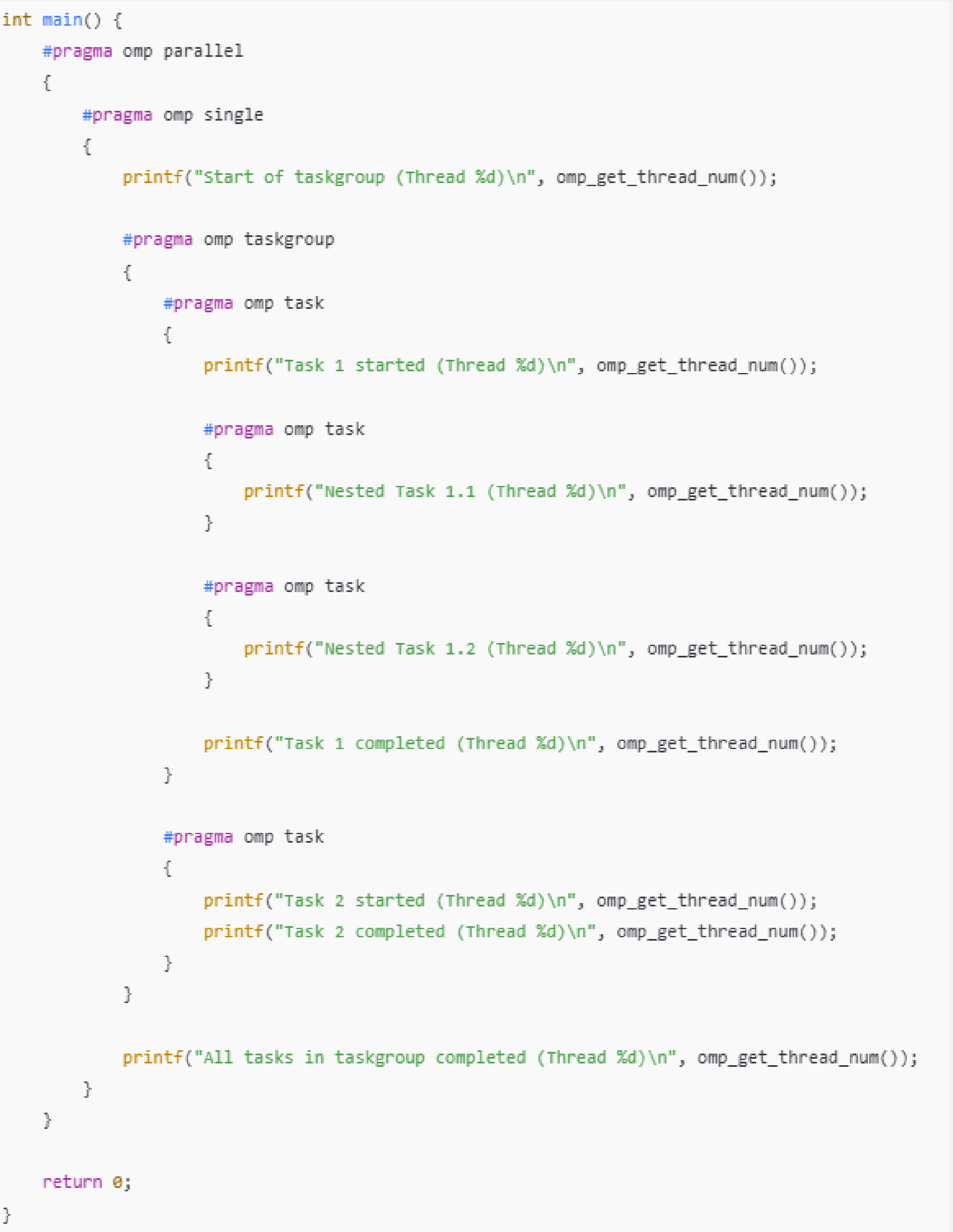
- #pragma omp taskwait ensures that "All tasks have been processed" is printed only **after all tasks finish execution**.
- The #pragma omp taskwait directive should be **inside the structured block where tasks are created**.
- Otherwise, it **would have no effect**, as the parallel region (and thus all tasks) would have already finished executing.

Example with `taskwait`



```
#pragma omp parallel
{
    #pragma omp single
    {
        // Create tasks here
        #pragma omp task
        { /* Task 1 */ }

        #pragma omp task
        { /* Task 2 */ }

        #pragma omp taskwait  // ✅ Waits for Task 1 and Task 2
    }
}
```

- #pragma omp taskwait ensures that "All tasks have been processed" is printed only **after all tasks finish execution**.
- The #pragma omp taskwait directive should be **inside the structured block where tasks are created**.
- Otherwise, it **would have no effect**, as the parallel region (and thus all tasks) would have already finished executing.



```c
int main() {
    #pragma omp parallel
    {
        #pragma omp single
        {
            printf("Start of taskgroup (Thread %d)\n", omp_get_thread_num());

            #pragma omp taskgroup
            {
                #pragma omp task
                {
                    printf("Task 1 started (Thread %d)\n", omp_get_thread_num());

                    #pragma omp task
                    {
                        printf("Nested Task 1.1 (Thread %d)\n", omp_get_thread_num());
                    }

                    #pragma omp task
                    {
                        printf("Nested Task 1.2 (Thread %d)\n", omp_get_thread_num());
                    }

                    printf("Task 1 completed (Thread %d)\n", omp_get_thread_num());
                }

                #pragma omp task
                {
                    printf("Task 2 started (Thread %d)\n", omp_get_thread_num());
                    printf("Task 2 completed (Thread %d)\n", omp_get_thread_num());
                }
            }

            printf("All tasks in taskgroup completed (Thread %d)\n", omp_get_thread_num());
        }
    }

    return 0;
}
```

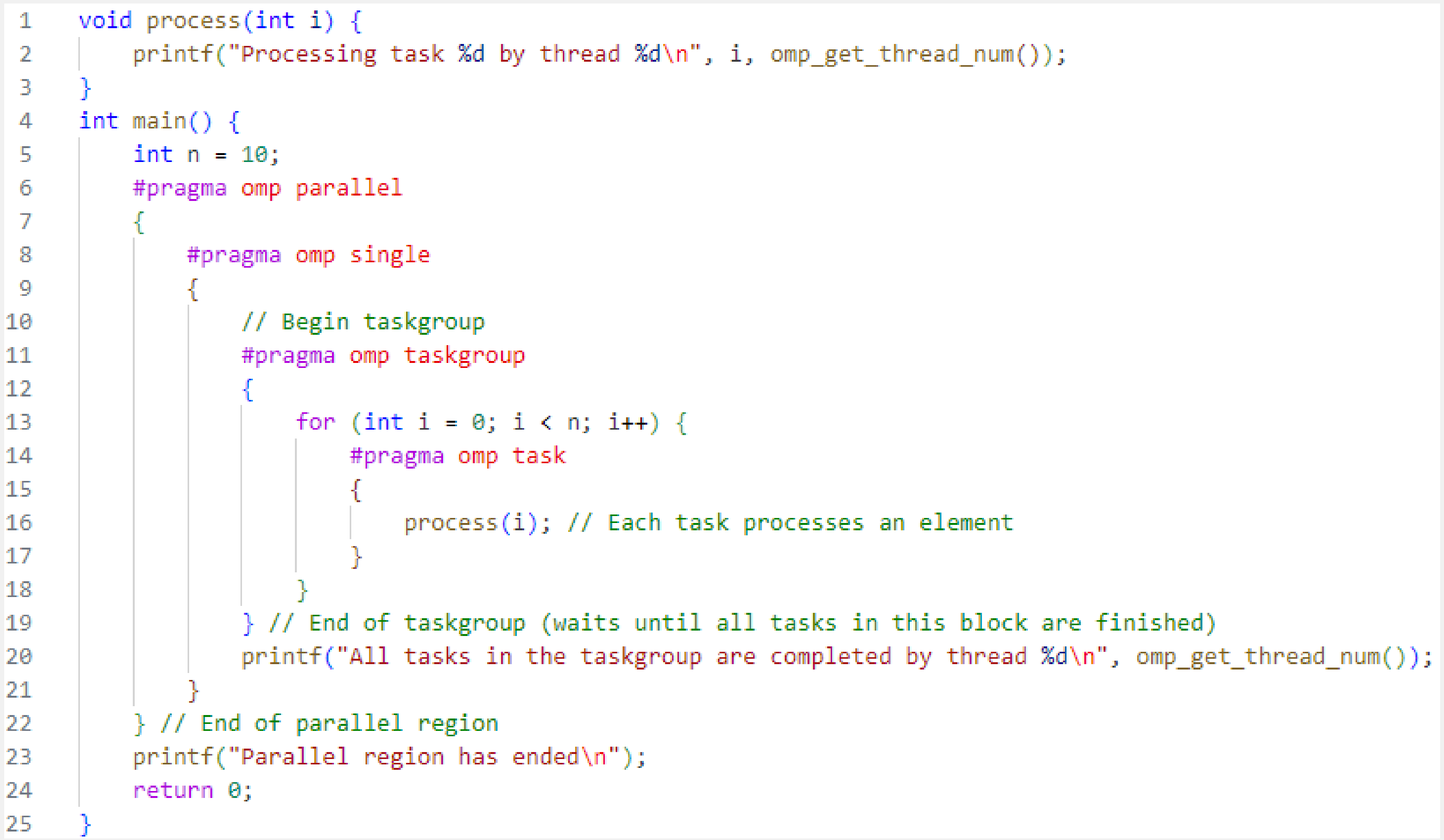
Example with `taskgroup`

Wait for all tasks created inside this block, including any nested tasks, before moving on."

What Happens?

- One thread enters single.
- Inside it, we create a taskgroup.
- Two top-level tasks (1 & 2) are created.
- Task 1 creates two nested tasks.
- The taskgroup will wait for all of them, including the nested ones, before continuing.

Example with `taskgroup`



```c
void process(int i) {
    printf("Processing task %d by thread %d\n", i, omp_get_thread_num());
}
int main() {
    int n = 10;
    #pragma omp parallel
    {
        #pragma omp single
        {
            // Begin taskgroup
            #pragma omp taskgroup
            {
                for (int i = 0; i < n; i++) {
                    #pragma omp task
                    {
                        process(i); // Each task processes an element
                    }
                }
            } // End of taskgroup (waits until all tasks in this block are finished)
            printf("All tasks in the taskgroup are completed by thread %d\n", omp_get_thread_num());
        }
    } // End of parallel region
    printf("Parallel region has ended\n");
    return 0;
}
```

More Examples

1st Code

```
#pragma omp parallel num_threads(3)
     {
             #pragma omp single
             {
                 printf(“A “);
                #pragma omp task
                 {printf(“B ");}
                #pragma omp task
                 {printf(“C ");}
             }
         }
```

When you run this code, you might see something like "A B C “ or “A C B”. The order of "B" and "C" might vary because they are executed asynchronously as tasks. "A " is printed first since it's not within any task region.

2nd Code

```
#pragma omp parallel num_threads(3)
 {
     #pragma omp single
     {
         printf(“A “);
         #pragma omp task
         {printf(“B ");}
         #pragma omp task
         {printf(“C ");}
         #pragma omp taskwait
        printf(“D“);
     }
 }
```

The final output might be "A B C D“ or “A C B D”. "B " and "C " are printed asynchronously. The taskwait pragma ensures that "D" is printed at the end.

#pragma parallel with Variables

- In parallel region, **default behavior** is that all variables are **shared** except loop index
- All threads read and write the same memory location for each variable
- This is ok if threads are accessing different elements of an array
- Problem if threads write same scalar or array element
- **Loop index is private**, so each thread has its own copy
- Here's an example where a shared variable, i2, could cause a problem.

```
ifirst = 10;
#pragma omp parallel for
for(i = 1; i <= imax; i++){
   i2 = 2*i;
   j[i] = ifirst + i2;
}
```

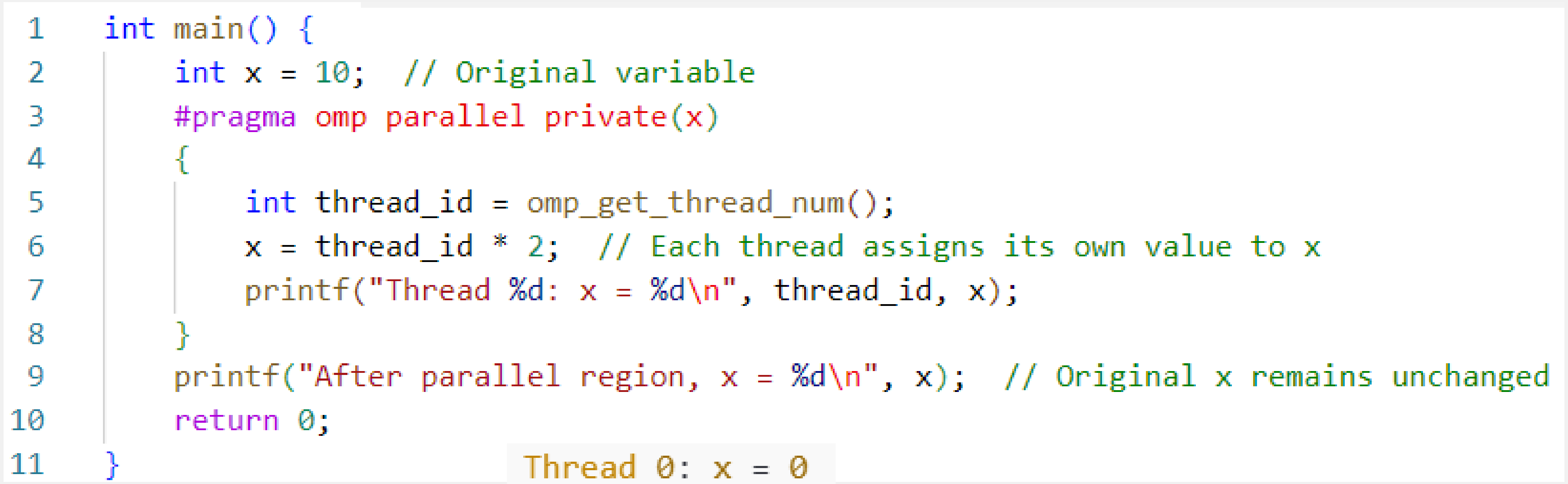
– Thread 1 calculates i2 = 2 for i = 1.
– Before Thread 1 updates j[1], the OS suspends thread 1.
– Now thread 2 calculates i2 = 4 for i = 2.
– Thread 2 updates j[2] = 10 + 4 = 14.
– Meanwhile, the OS resumes the execution of Thread 1.
– Thread 1 updates j[1] using ifirst + i2, but i2 is now 4 (calculated by Thread 2), so j[1] = 10 + 4 = 14, which is incorrect. Thread 1 should have used i2 = 2 for i = 1.

#pragma parallel with private clause

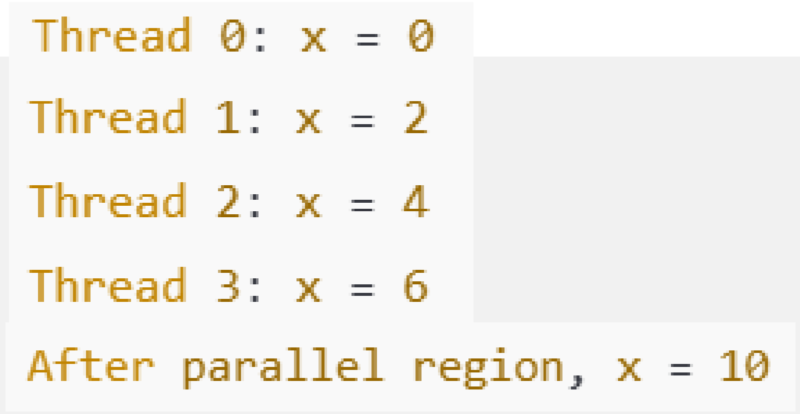
- `Private` clause creates local copies of the specified variable for each thread
- There is no connection between the original variable and the private copies

```
ifirst = 10;
#pragma omp parallel for private(i2)
for(i = 1; i <= imax; i++){
   i2 = 2*i;
   j[i] = ifirst + i2;
}
```

#pragma parallel with private Example



```c
int main() {
    int x = 10;  // Original variable
    #pragma omp parallel private(x)
    {
        int thread_id = omp_get_thread_num();
        x = thread_id * 2;  // Each thread assigns its own value to x
        printf("Thread %d: x = %d\n", thread_id, x);
    }
    printf("After parallel region, x = %d\n", x);  // Original x remains unchanged
    return 0;
}
```



```
Thread 0: x = 0
Thread 1: x = 2
Thread 2: x = 4
Thread 3: x = 6
After parallel region, x = 10
```

More example on Private Variables

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

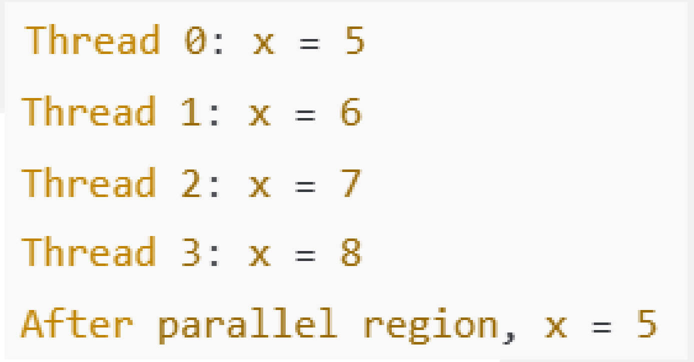
- Swaps the values in a and b.
- Loop-carried dependence on tmp.
- Easily fixed by privatizing tmp.

```
#pragma omp parallel for
   private( tmp )
for( i=0; i<n; i++ ) {
   tmp = a[i];
   a[i] = b[i];
   b[i] = tmp;
}
```

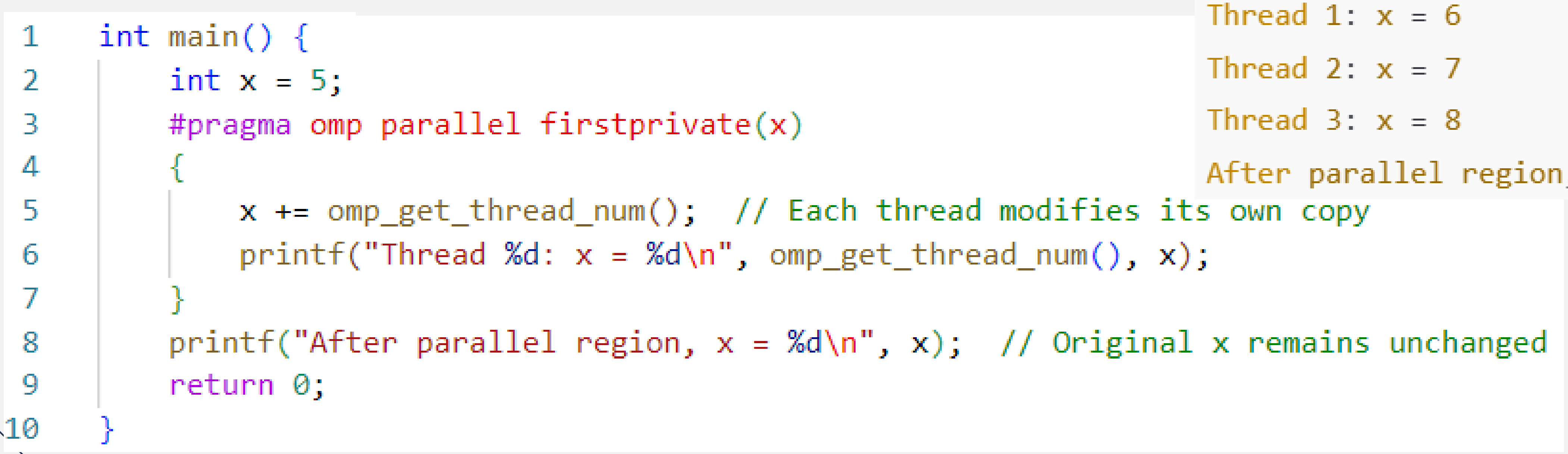
- Removes dependence on tmp.

`#pragma parallel` with `firstprivate`

- The `firstprivate` clause initializes each thread's private copy of a variable to the value the variable had before entering the parallel region.
- This is useful if each thread needs to work with the initial value of a variable



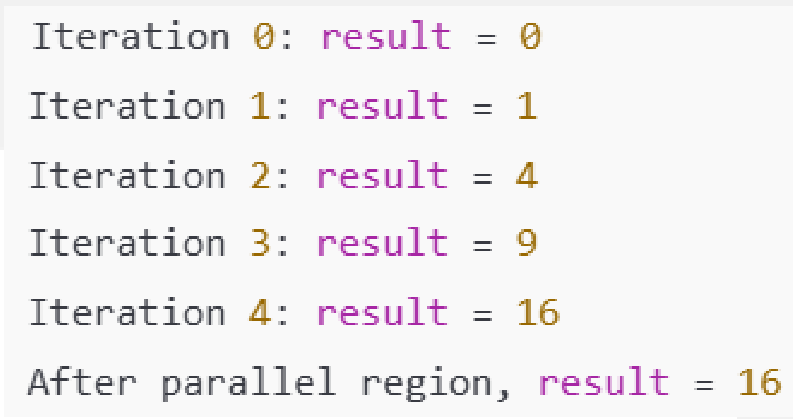
```
Thread 0: x = 5
Thread 1: x = 6
Thread 2: x = 7
Thread 3: x = 8
After parallel region, x = 5
```



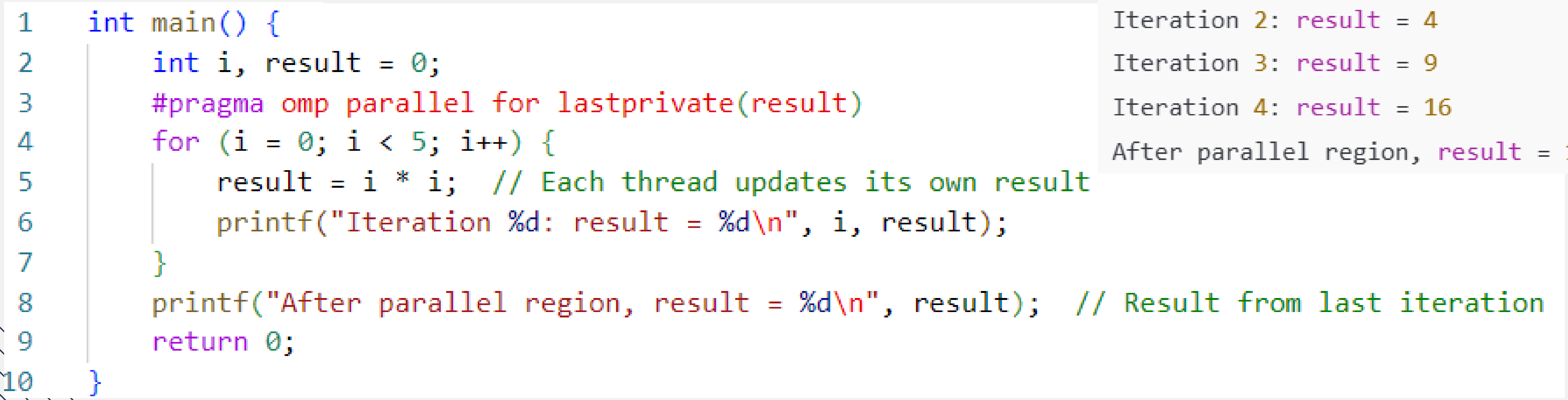
```c
int main() {
    int x = 5;
    #pragma omp parallel firstprivate(x)
    {
        x += omp_get_thread_num();  // Each thread modifies its own copy
        printf("Thread %d: x = %d\n", omp_get_thread_num(), x);
    }
    printf("After parallel region, x = %d\n", x);  // Original x remains unchanged
    return 0;
}
```

#pragma parallel with lastprivate

- The lastprivate clause ensures that the last iteration of a loop (or the last thread in a parallel section) assigns its value to the shared variable after the parallel region ends.



```
Iteration 0: result = 0
Iteration 1: result = 1
Iteration 2: result = 4
Iteration 3: result = 9
Iteration 4: result = 16
After parallel region, result = 16
```



```c
int main() {
    int i, result = 0;
    #pragma omp parallel for lastprivate(result)
    for (i = 0; i < 5; i++) {
        result = i * i;  // Each thread updates its own result
        printf("Iteration %d: result = %d\n", i, result);
    }
    printf("After parallel region, result = %d\n", result);  // Result from last iteration
    return 0;
}
```

Last Value with `lastprivate`

- The **lastprivate** clause does not depend on **which thread runs the last iteration**. Instead, **it is concerned with the logical last iteration** of the loop.
- OpenMP **ensures** that the **final value** of the variable **from the last logical iteration is copied** out to the shared variable after the parallel region ends.
- The actual thread that performs the last iteration can vary based on the schedule type and thread workload, but **lastprivate ensures that the correct final value is stored consistently**.

Last Value with `lastprivate`

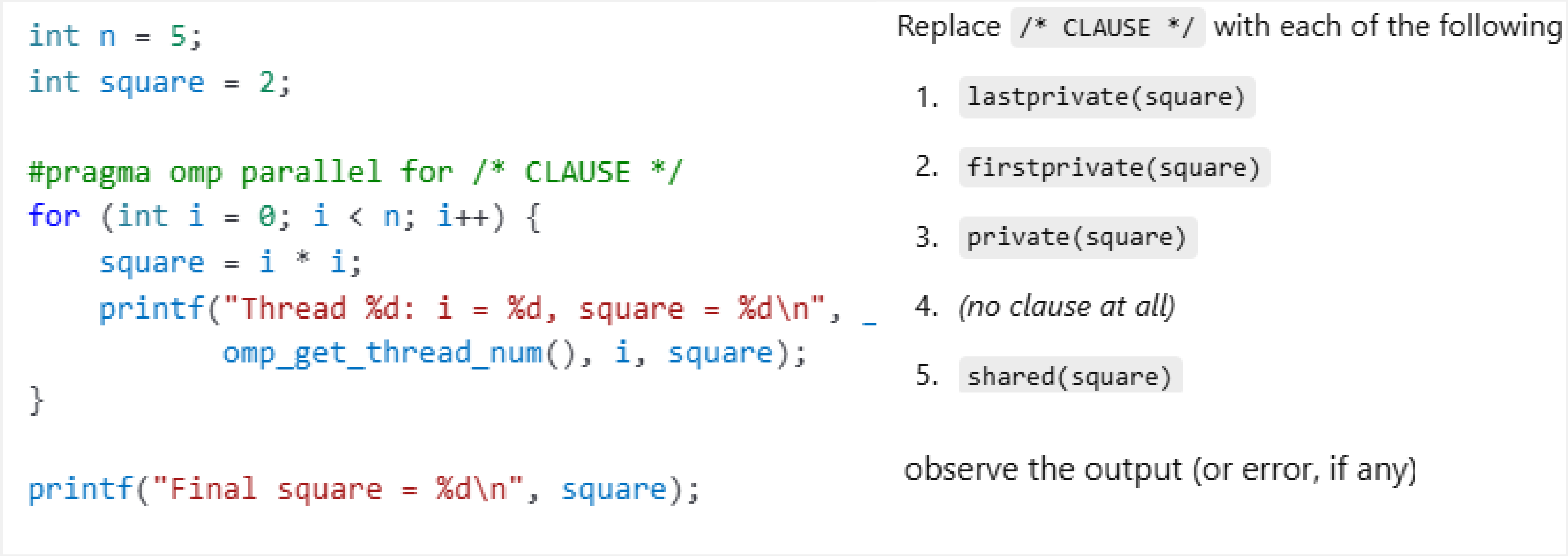
- If you use **lastprivate without a for loop**, it will behave a bit differently because, outside of a loop context, there is no "last iteration."

- Instead, OpenMP will treat the lastprivate variable according to the **last thread** that completes its execution within the parallel region.

- Since thread execution order can vary, the **final value** of result after the parallel region could **differ** between runs.

- Using lastprivate without a for loop is **less common** because of this **unpredictability**.

Example



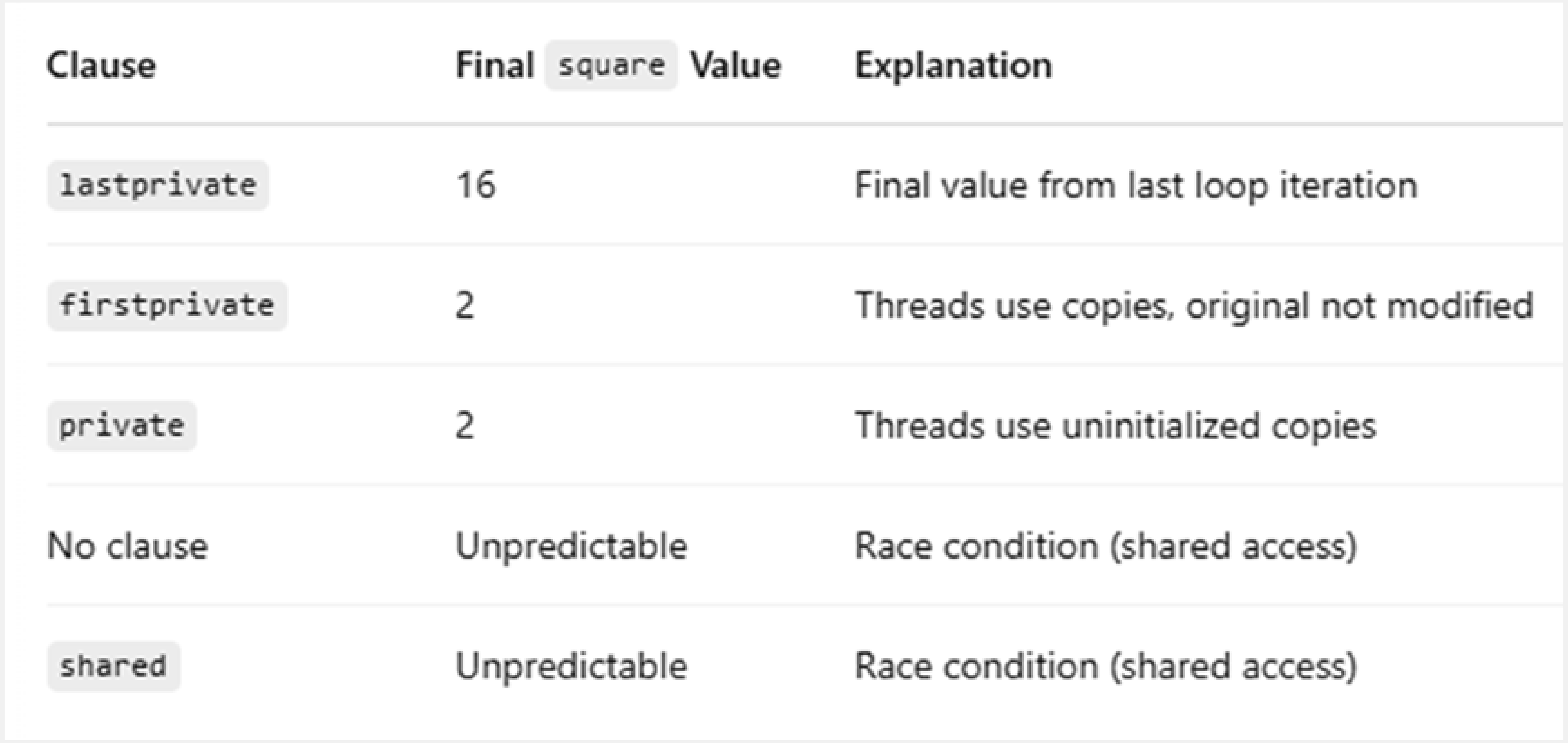
```c
int n = 5;
int square = 2;

#pragma omp parallel for /* CLAUSE */
for (int i = 0; i < n; i++) {
    square = i * i;
    printf("Thread %d: i = %d, square = %d\n", _
           omp_get_thread_num(), i, square);
}

printf("Final square = %d\n", square);
```

Replace `/* CLAUSE */` with each of the following

1. `lastprivate(square)`
2. `firstprivate(square)`
3. `private(square)`
4. *(no clause at all)*
5. `shared(square)`

observe the output (or error, if any)

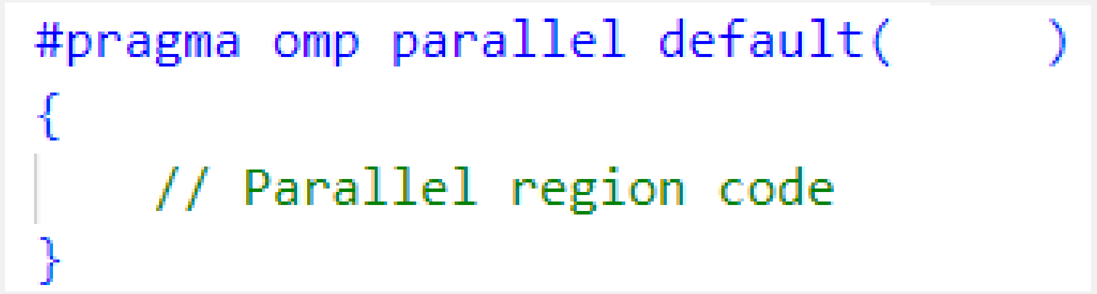
Solution



Clause	Final `square` Value	Explanation
`lastprivate`	16	Final value from last loop iteration
`firstprivate`	2	Threads use copies, original not modified
`private`	2	Threads use uninitialized copies
No clause	Unpredictable	Race condition (shared access)
`shared`	Unpredictable	Race condition (shared access)

Default Clause

- In OpenMP, the **default clause** is used to specify the default data-sharing behavior of variables within a parallel region.
- The default clause sets whether variables are **shared**, **private**, or have no default (none).
- This helps control how variables are treated in parallel regions without needing to individually specify private or shared for each variable.



```
#pragma omp parallel default(     )
{
    // Parallel region code
}
```

Types of Default Clause

- **`default(shared)`**
  - All variables are shared among threads unless explicitly specified otherwise (e.g., marked as private).
- **`default(private)`**
  - All variables in the parallel region are treated as private.
  - Each thread gets its own independent copy of variables.
- **`default(none)`**
  - Forces you to explicitly declare all variable data-sharing attributes.
  - If a variable's data-sharing attribute is not explicitly declared, the compiler will generate an error.
  - This is useful for safe coding because it avoids unintentional sharing.

#pragma parallel with reduction

- The **reduction** clause in OpenMP is used to perform reductions on variables in parallel regions.
- A reduction operation takes a variable that all threads update, and **combines the results** from each thread into a single final result once the parallel region completes.
- This is useful for operations that accumulate results, like summing values, finding a minimum or maximum, or performing logical operations across multiple threads.

```
#pragma omp parallel for reduction(operator: variable)
{
    // Parallel region where 'variable' is used in a reduction operation
}
```

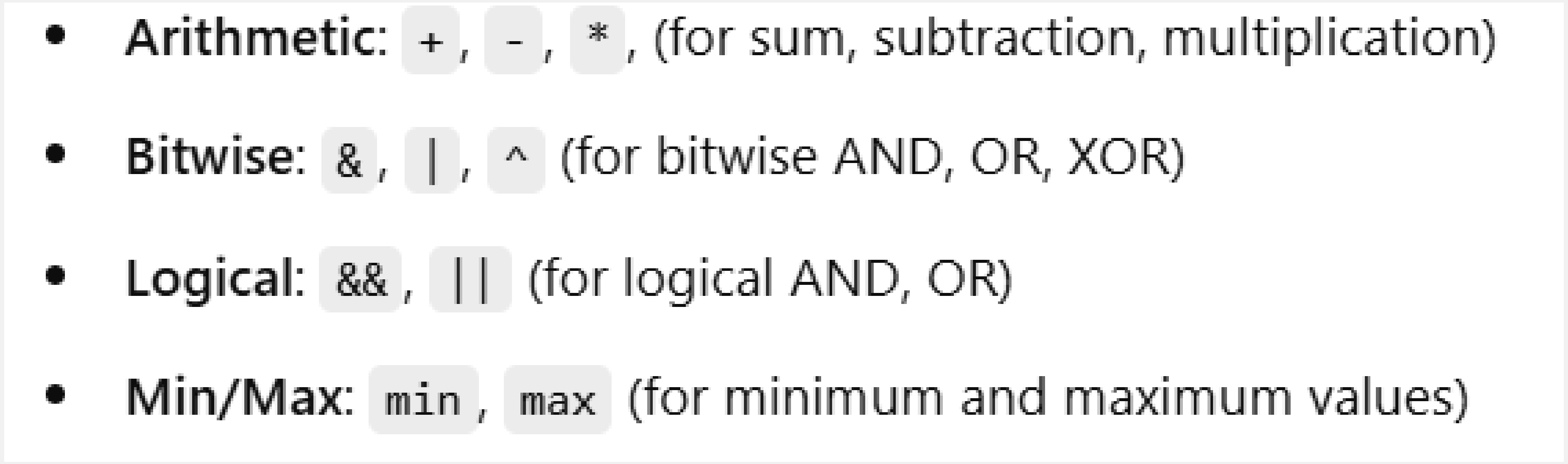
#pragma parallel with reduction

- **operator**: The type of reduction operation to apply. Common operators include +, *, -, &, |, ^, &&, and ||.
- **variable**: The variable on which the reduction operation is applied.
- When a variable is specified with the reduction clause, it is assigned a **private status**.
- The variables in list **must be used with this operator** in the loop.
- The variables are **automatically initialized** to sensible values.

```
#pragma omp parallel for reduction(operator: variable)
{
    // Parallel region where 'variable' is used in a reduction operation
}
```

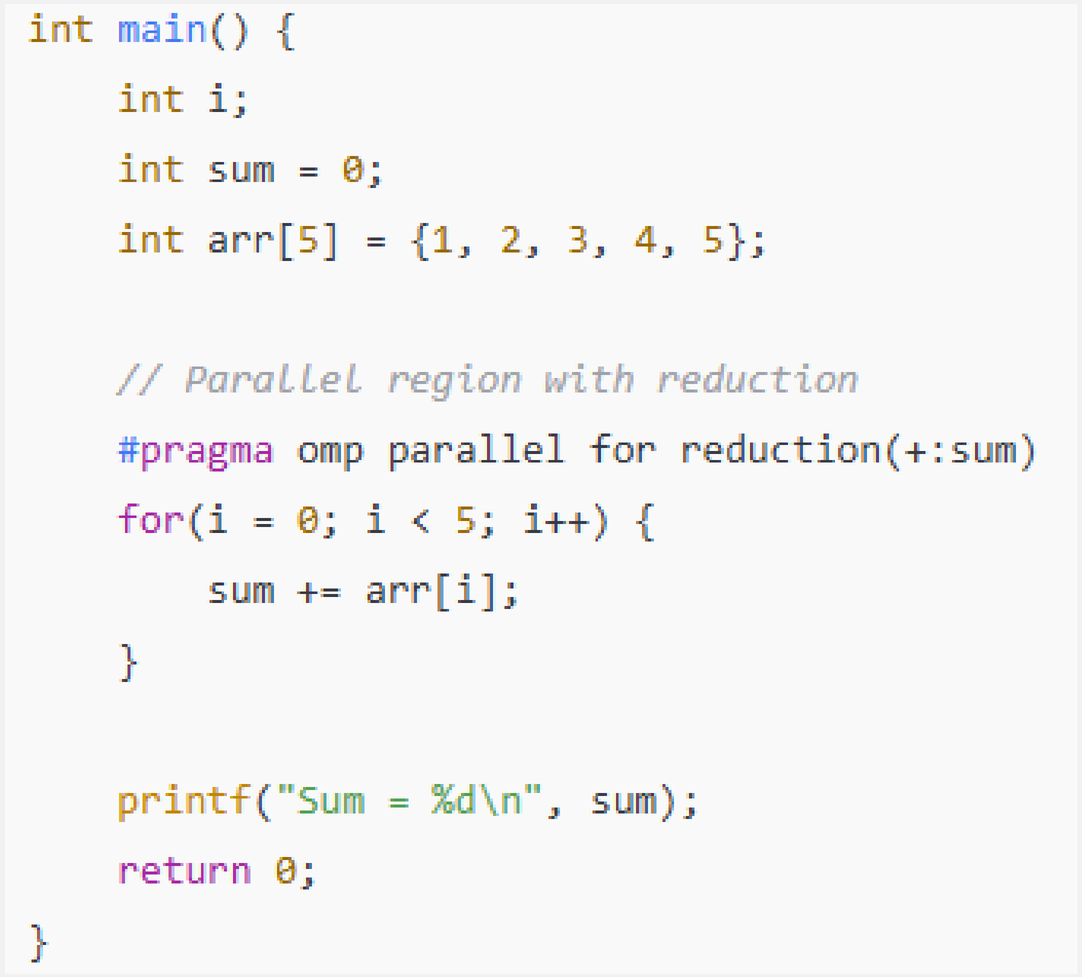
Operator	Initial Value
+	0
*	1
-	0

Supported Reduction Operators



- **Arithmetic:** `+`, `-`, `*`, (for sum, subtraction, multiplication)
- **Bitwise:** `&`, `|`, `^` (for bitwise AND, OR, XOR)
- **Logical:** `&&`, `||` (for logical AND, OR)
- **Min/Max:** `min`, `max` (for minimum and maximum values)

Example of reduction with Summation



```c
int main() {
    int i;
    int sum = 0;
    int arr[5] = {1, 2, 3, 4, 5};

    // Parallel region with reduction
    #pragma omp parallel for reduction(+:sum)
    for(i = 0; i < 5; i++) {
        sum += arr[i];
    }

    printf("Sum = %d\n", sum);
    return 0;
}
```

- Create a **local copy** of sum for each thread.
- Perform the **addition** (+) inside each thread.
- After the loop, OpenMP **combines** all local sum values into the global sum.

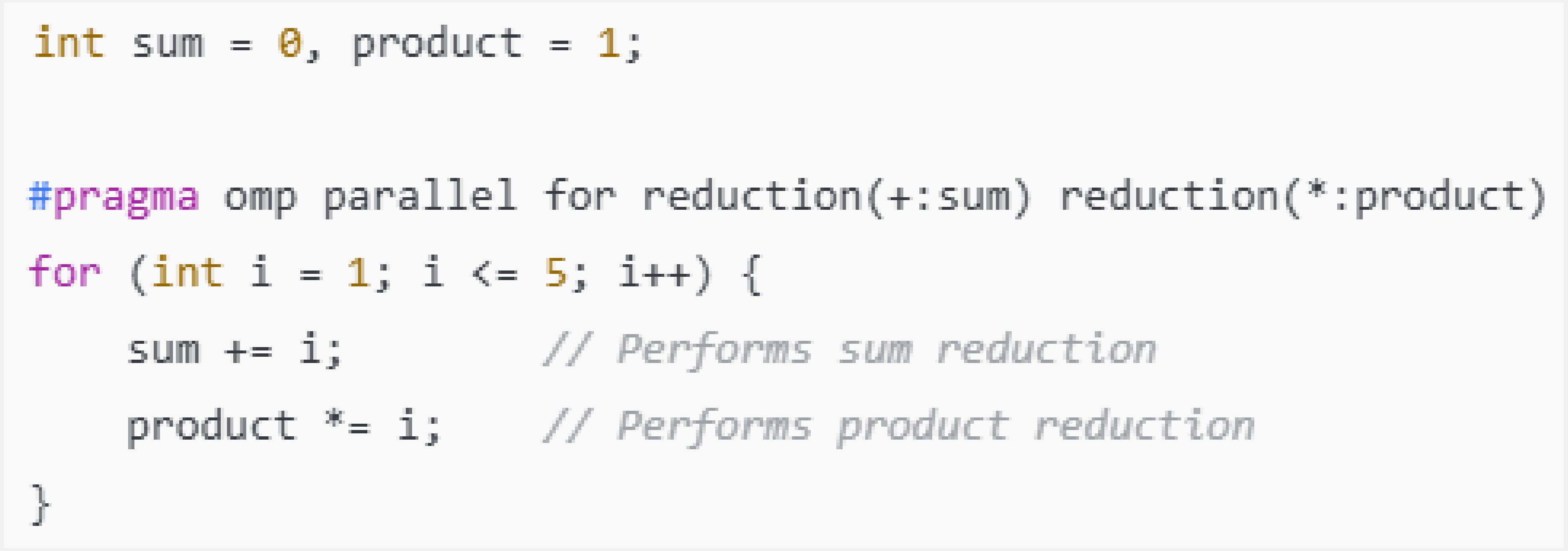
- **Sum = 15** (because 1 + 2 + 3 + 4 + 5 = 15)

Example of reduction with 2 variables

```
#pragma omp parallel for reduction(+: a, y)
for (i = 0; i < n; i++) {
    a += b[i];  // Operation on array b
    y = y + c[i];  // Operation on array c
}
```

At the end of the region for which the reduction clause was specified, the original object is updated to reflect the result of combining its original value with the final value of each of the private copies using the operator specified.

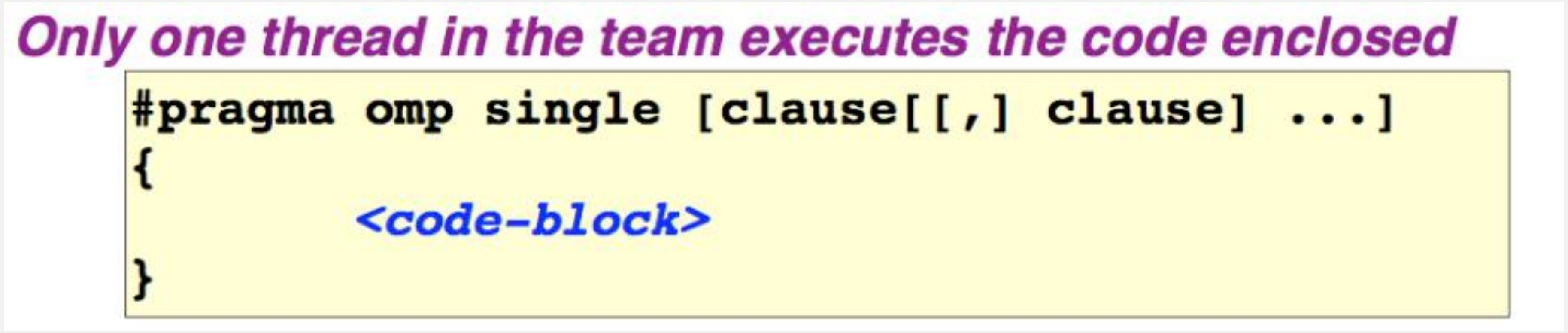
Example of reduction with 2 operators



```
int sum = 0, product = 1;


#pragma omp parallel for reduction(+:sum) reduction(*:product)
for (int i = 1; i <= 5; i++) {
    sum += i;          // Performs sum reduction
    product *= i;      // Performs product reduction
}
```

- In the latest specification of OpenMP, you can use the reduction clause with more than one operator, but each operator must be applied to a different variable.
- You cannot apply multiple operators to the same variable in one reduction. Each variable can only appear once in all reduction clauses for a single directive.

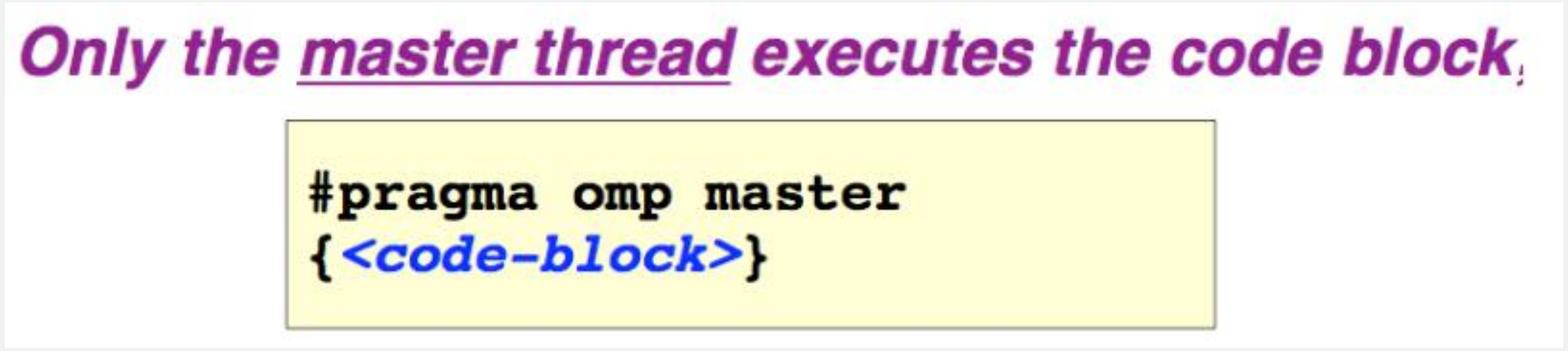
single and master Construct



Only one thread in the team executes the code enclosed

```
#pragma omp single [clause[[,] clause] ...]
{
        <code-block>
}
```

- Useful when only one thread should do a task like I/O or initialization.

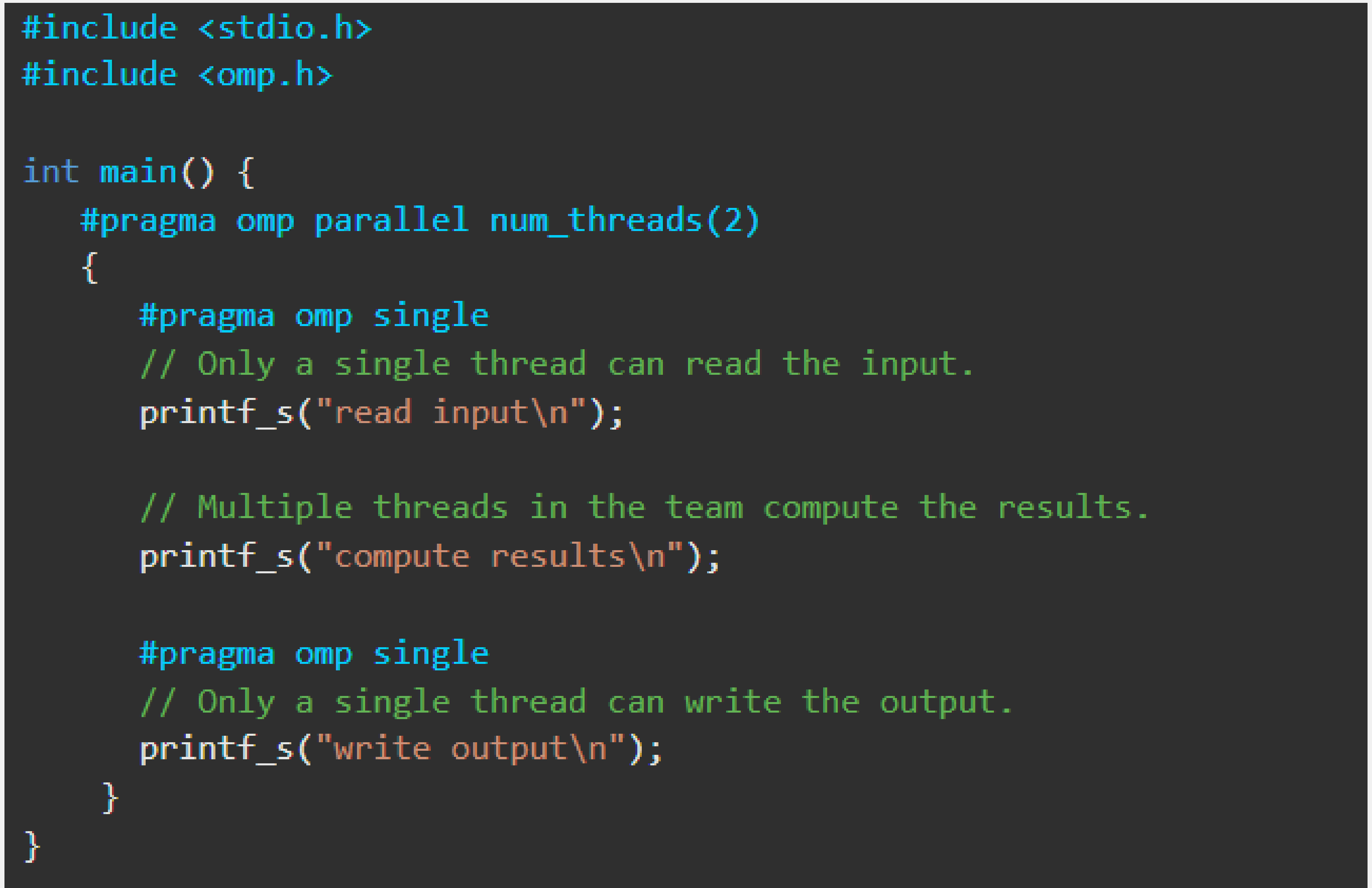


Only the master thread executes the code block

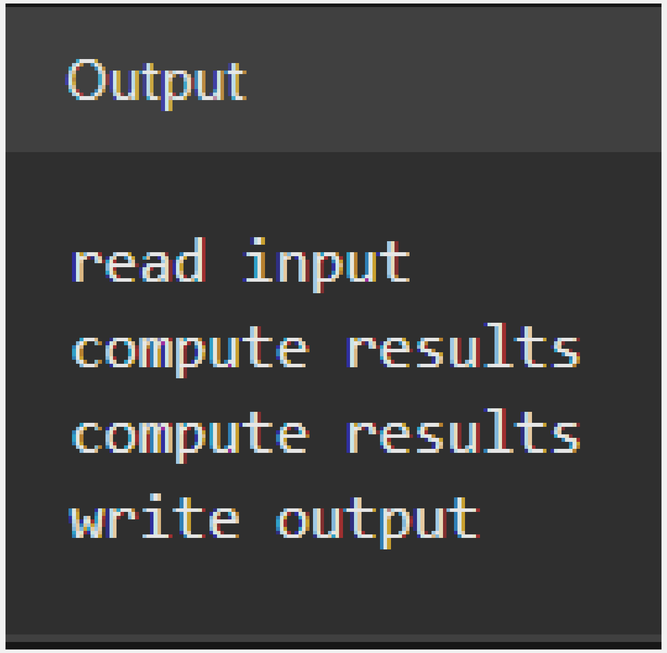
```
#pragma omp master
{<code-block>}
```

- Useful when you want a specific thread (master) to handle a task.

Example



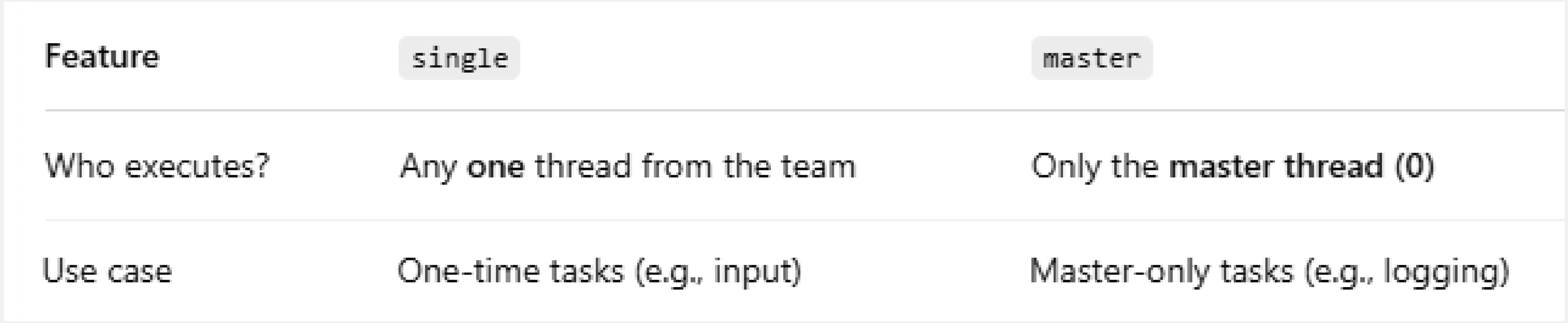
```
#include <stdio.h>
#include <omp.h>

int main() {
   #pragma omp parallel num_threads(2)
   {
      #pragma omp single
      // Only a single thread can read the input.
      printf_s("read input\n");

      // Multiple threads in the team compute the results.
      printf_s("compute results\n");

      #pragma omp single
      // Only a single thread can write the output.
      printf_s("write output\n");
    }
}
```



Output

```
read input
compute results
compute results
write output
```

single and master Construct



Feature	`single`	`master`
Who executes?	Any **one** thread from the team	Only the **master thread** (0)
Use case	One-time tasks (e.g., input)	Master-only tasks (e.g., logging)

Conditional Parallelism

- Oftentimes, parallelism is only useful if the problem size is sufficiently big.

- In OpenMP, conditional parallelism allows you to decide at runtime whether a parallel region should actually run in parallel or just run in serial (single thread), based on a condition.

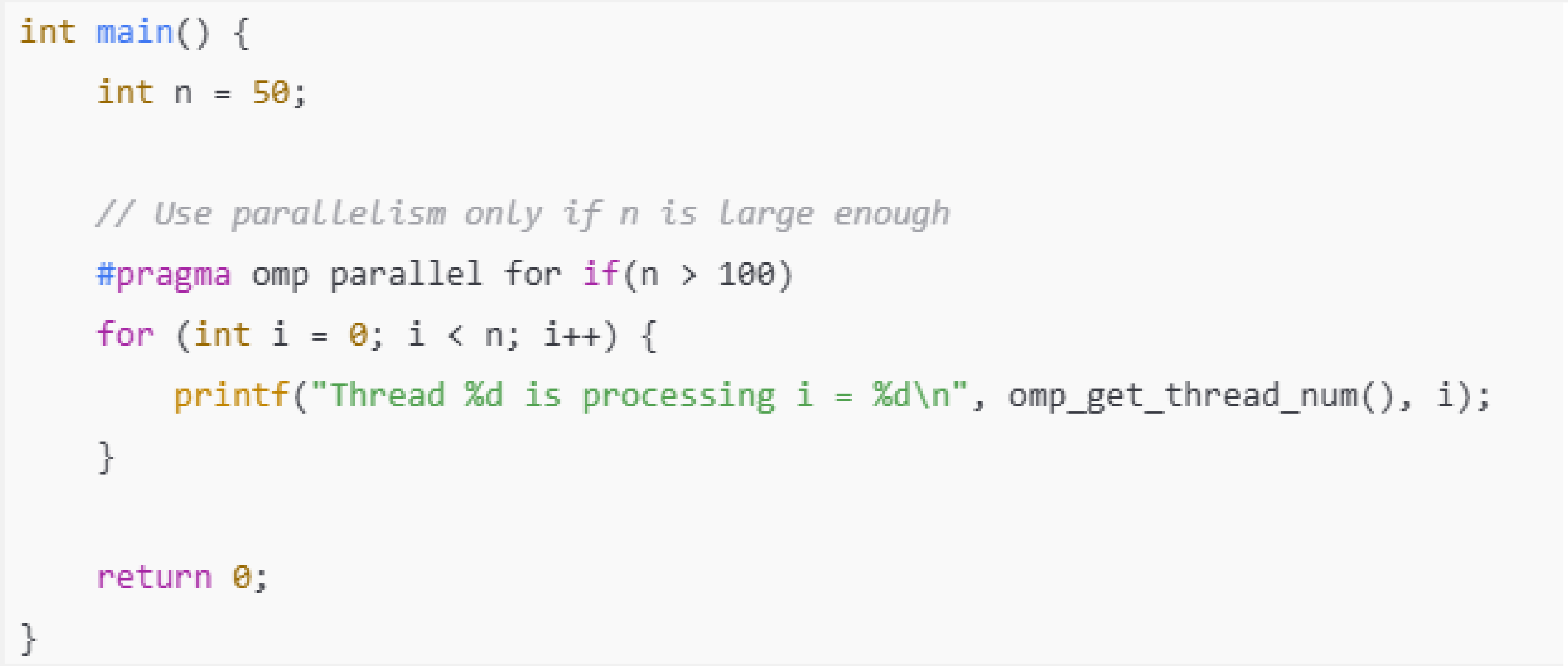
- This is done using the if clause with `#pragma omp parallel`.

Conditional Parallelism: Specification

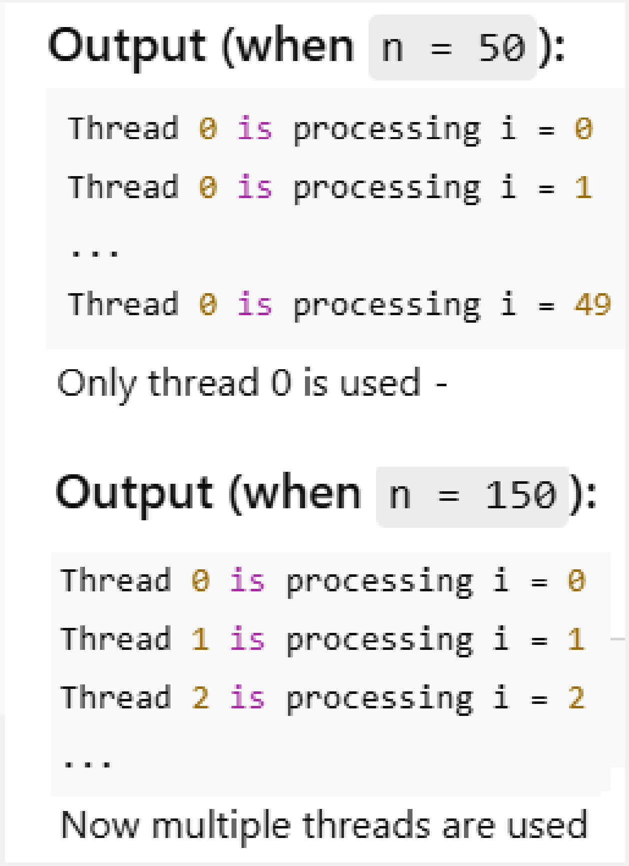
```
#pragma omp parallel if( expression )
#pragma omp for if( expression )
#pragma omp parallel for if( expression )
```

- Execute in parallel if expression is true, otherwise execute sequentially.

Conditional Parallelism: Example



```c
int main() {
    int n = 50;

    // Use parallelism only if n is large enough
    #pragma omp parallel for if(n > 100)
    for (int i = 0; i < n; i++) {
        printf("Thread %d is processing i = %d\n", omp_get_thread_num(), i);
    }

    return 0;
}
```



Output (when `n = 50`):

```
Thread 0 is processing i = 0
Thread 0 is processing i = 1
...
Thread 0 is processing i = 49
```

Only thread 0 is used -

Output (when `n = 150`):

```
Thread 0 is processing i = 0
Thread 1 is processing i = 1
Thread 2 is processing i = 2
...
```

Now multiple threads are used

- If n > 100, OpenMP will run the loop in parallel.
- If n <= 100, the loop will run sequentially.
- This avoids the overhead of creating threads when the work (n) is small.

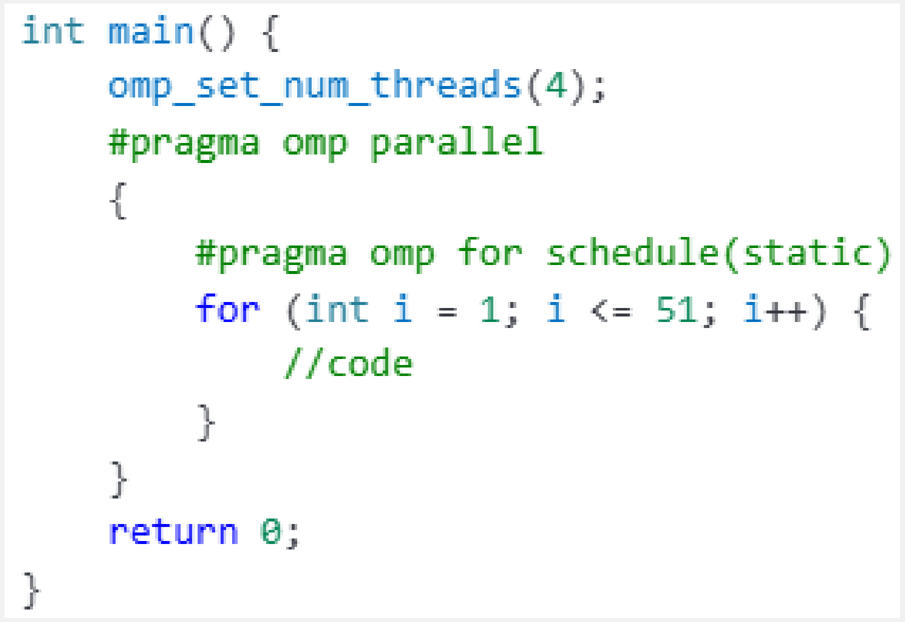
Schedule Clause

- A parallel region has **at least one barrier**, at its end, and may have **additional** barriers within it. At each barrier, the other members of the team must **wait** for the **last thread to arrive**. To minimize this wait time, shared work should be distributed so that all threads arrive at the barrier at about the same time.

- The schedule clause of the for directive deals with the assignment of iterations to threads.
- The general form of the schedule directive is
  - `schedule(scheduling_class[, parameter]).`
- OpenMP supports four scheduling classes: **static**, **dynamic**, **guided** and **runtime**

Static Schedule

- **Behavior:** Iterations are divided into equal-sized chunks, and each chunk is assigned to a thread before the loop starts. Once assigned, the chunks do not change during execution.
- **Best for:** Uniform workload (all iterations take roughly the same time).
- **Example:** loop runs from 1 to 51, 4 threads

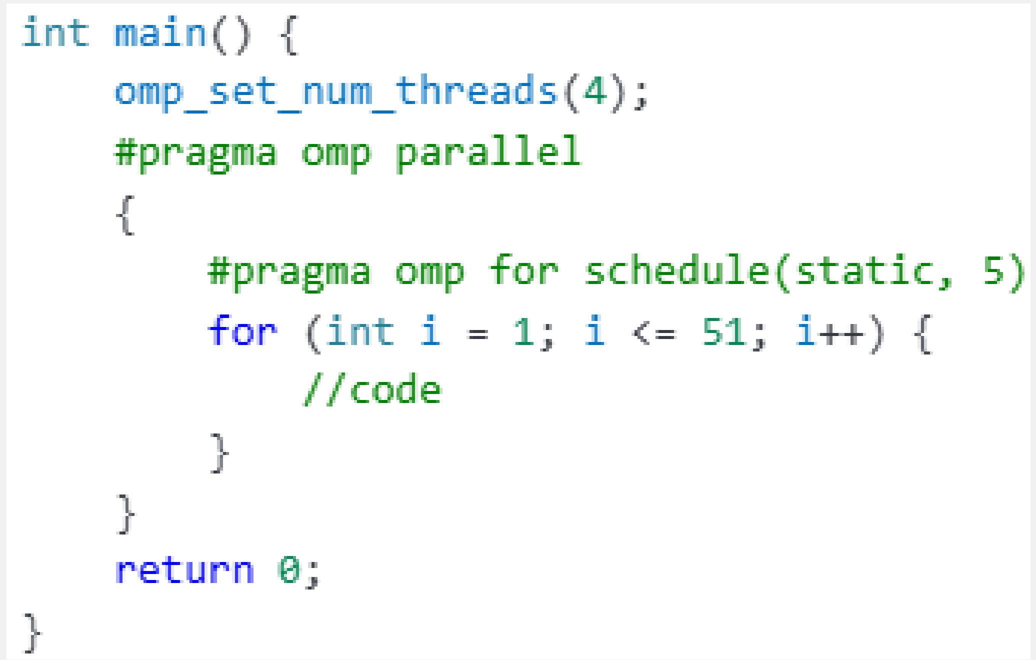


```
int main() {
    omp_set_num_threads(4);
    #pragma omp parallel
    {
        #pragma omp for schedule(static)
        for (int i = 1; i <= 51; i++) {
            //code
        }
    }
    return 0;
}
```

thread	indices	no. indices
0	1-13	13
1	14-26	13
2	27-39	13
3	40-51	12

Static Schedule

- Number of indices assigned to each thread at a time is called the chunk size
- can be modified with the SCHEDULE clause
- **Example:** loop runs from 1 to 51, 4 threads but chunk size is 5
- Here chunks are assigned to the threads in the team in a round-robin fashion in the order of the thread number



```
int main() {
    omp_set_num_threads(4);
    #pragma omp parallel
    {
        #pragma omp for schedule(static, 5)
        for (int i = 1; i <= 51; i++) {
            //code
        }
    }
    return 0;
}
```

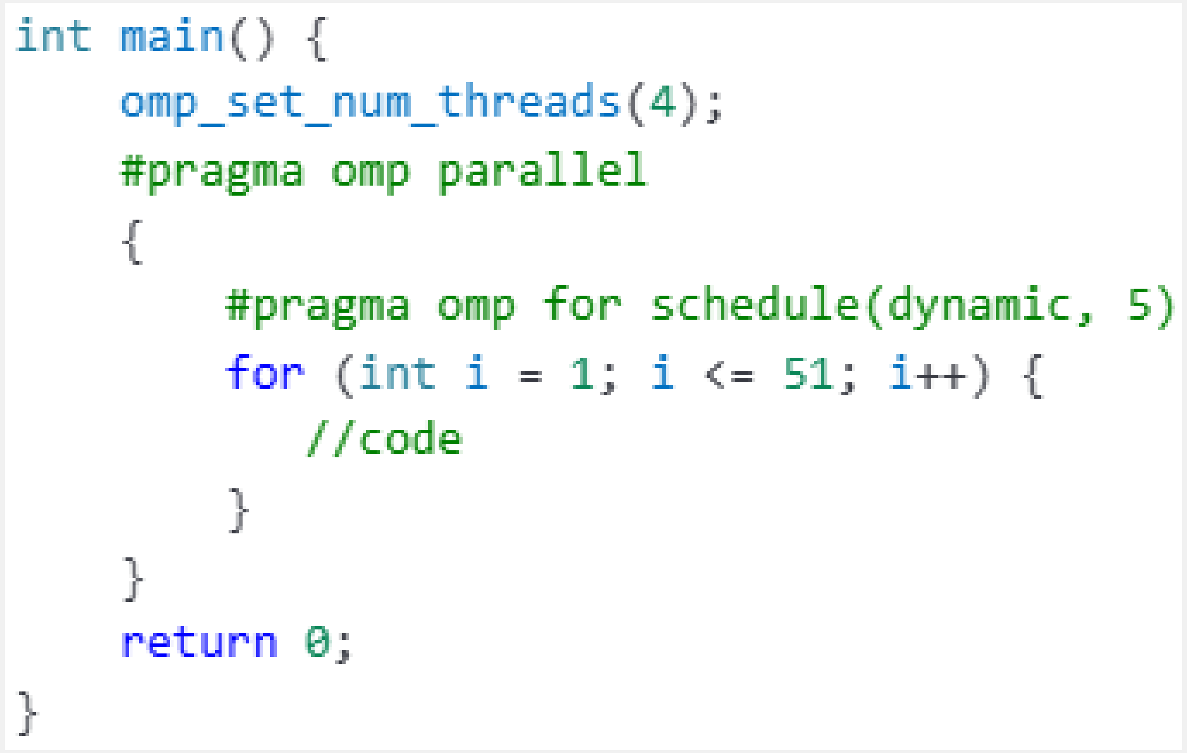
thread	chunk 1 indices	chunk 2 indices	chunk 3 indices	no. indices
0	1-5	21-25	41-45	15
1	6-10	26-30	46-50	15
2	11-15	31-35	51	11
3	16-20	36-40	-	10

Dynamic Schedule

- `#pragma omp for schedule(dynamic, Chunk_Size)`
- **Chunk_Size** is optional and **Default Chunk Size** is 1
- Iterations are divided into chunks of a **Chunk_Size**.
- Threads are initially assigned one **Chunk_Size** at a time
- Thread takes the next available chunk once it finishes its current chunk.
- Dynamic scheduling assigns loop iterations to threads at runtime.
- This is helpful when the time to process each iteration is uneven or unpredictable.

Dynamic Schedule Example

Unlike static, where chunks are pre-assigned in round-robin order, dynamic lets faster threads do more work, which can lead to better load balancing.



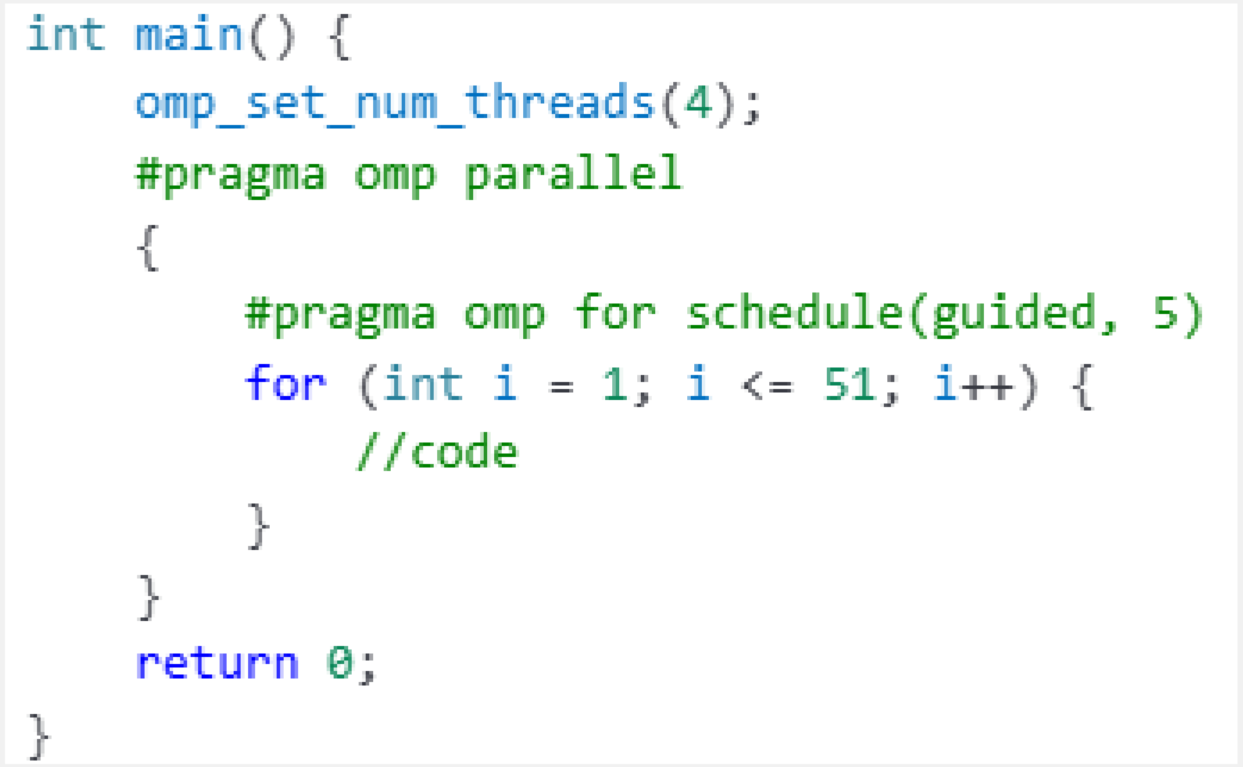
```
int main() {
    omp_set_num_threads(4);
    #pragma omp parallel
    {
        #pragma omp for schedule(dynamic, 5)
        for (int i = 1; i <= 51; i++) {
           //code
        }
    }
    return 0;
}
```

Guided Schedule

- **`#pragma omp for schedule(dynamic, Min_Chunk_Size)`**
- **Min_Chunk_Size** is optional and **Default Min_Chunk Size** is 1
- Guided scheduling starts by giving large chunks to threads.
- As threads finish their work, the chunk size gradually decreases.
- These decreasing chunk sizes will eventually fall to `Min_Chunk_Size`
- It's useful when the beginning of the loop is heavier and we want better load balancing as the work progresses.

Guided Schedule Example

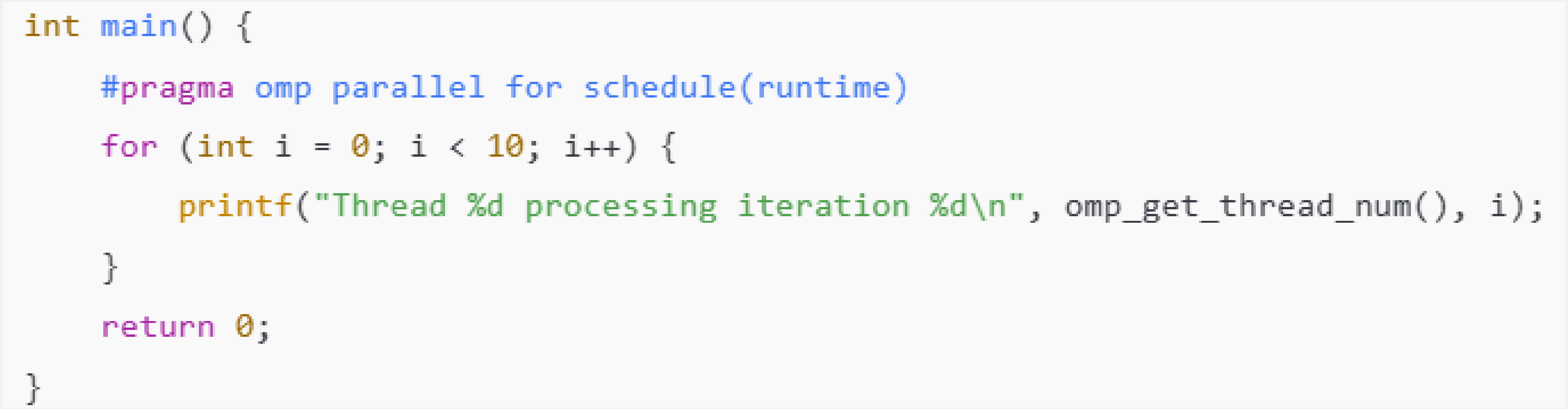
- OpenMP assigns large chunks at the beginning → fewer threads are idle.
- As the loop progresses (and the work per iteration decreases), it assigns smaller chunks, so threads finish at around the same time.



```
int main() {
    omp_set_num_threads(4);
    #pragma omp parallel
    {
        #pragma omp for schedule(guided, 5)
        for (int i = 1; i <= 51; i++) {
            //code
        }
    }
    return 0;
}
```

Runtime Schedule

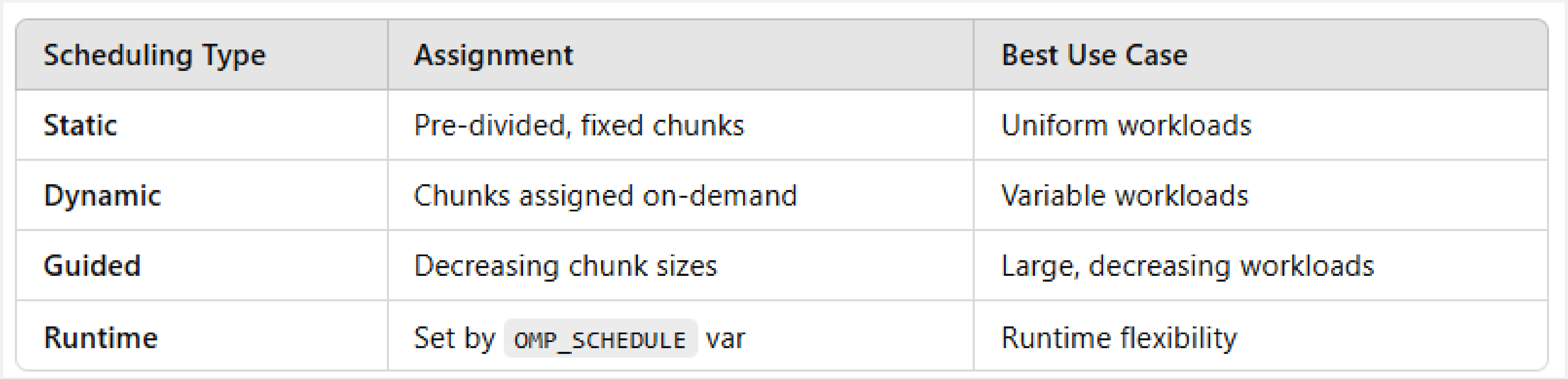
- **Behavior**: The scheduling is determined by the OMP_SCHEDULE environment variable at runtime.
- **Best for**: When the scheduling policy needs to be decided at runtime.



```c
int main() {
    #pragma omp parallel for schedule(runtime)
    for (int i = 0; i < 10; i++) {
        printf("Thread %d processing iteration %d\n", omp_get_thread_num(), i);
    }
    return 0;
}
```

- e.g., export OMP_SCHEDULE="dynamic,2" before running the program to use a dynamic schedule with chunk size 2.

Summary



Scheduling Type	Assignment	Best Use Case
Static	Pre-divided, fixed chunks	Uniform workloads
Dynamic	Chunks assigned on-demand	Variable workloads
Guided	Decreasing chunk sizes	Large, decreasing workloads
Runtime	Set by `OMP_SCHEDULE` var	Runtime flexibility

Synchronization

- Synchronization: A technique that (a) imposes order of execution, and/or (b) provides protected access to shared data
  - **High Level Synchronization:**
    - Barrier
    - Taskwait
    - Critical
    - atomic
  - **Low Level Synchronization:**
    - locks

OpenMP’s Critical Directive

- critical directive ensure that **only one thread** executes a specific block of code at a time.
- This is helpful when multiple threads need to access shared resources in a way that could cause issues if done simultaneously.

```c
int main() {
    int myVariable = 0;

    #pragma omp parallel num_threads(4)
    {
        #pragma omp critical
        {
            myVariable += 1;
            printf("Thread is %d : Variable is %d\n", omp_get_thread_num(), myVariable);
        }
    }

    printf("Final value of the shared variable: %d\n", myVariable);
    return 0;
}
```

```
Thread is 2 : Variable is 1
Thread is 1 : Variable is 2
Thread is 3 : Variable is 3
Thread is 0 : Variable is 4
Final value of the shared variable: 4
```

Atomic

- Similar to critical, but it's faster and lighter
- With critical, you incur significant overhead every time a thread enters and exits the critical section
- An atomic operation has much lower overhead. It takes advantage of the hardware support if available .
- Applies to single line following the atomic directive

```c
int main() {
    int counter = 0;

    #pragma omp parallel num_threads(4)
    {
        #pragma omp atomic
        counter += 1;

        printf("Thread %d updated the counter.\n", omp_get_thread_num());
    }

    printf("Final counter value: %d\n", counter);
    return 0;
}
```

Barriers

- All 4 threads print "before barrier".
- They then wait at the #pragma omp barrier.
- Only when all threads reach the barrier, they continue and print "after barrier".

```
int main() {
    #pragma omp parallel num_threads(4)
    {
        int tid = omp_get_thread_num();

        printf("Thread %d before barrier\n", tid);

        #pragma omp barrier

        printf("Thread %d after barrier\n", tid);
    }

    return 0;
}
```

```
Thread 0 before barrier
Thread 2 before barrier
Thread 1 before barrier
Thread 3 before barrier
Thread 2 after barrier
Thread 0 after barrier
Thread 1 after barrier
Thread 3 after barrier
```

Locks

- OpenMP locks are **synchronization** mechanisms used to **control access** to **shared resources** in parallel regions.
- When a lock is owned by a thread, other threads cannot execute the locked region
- **Step 1: Declare**
  - Simple locks (omp_lock_t) are the most commonly used type.
- **Step 2: Initialize**
  - Locks are created and initialized using omp_init_lock
- **Step 3: Acquire**
  - To acquire a lock, a thread calls omp_set_lock
- **Step 4: Release**
  - When a thread no longer needs the lock, it releases it using omp_unset_lock
- **Step 5: Destroy**
  - After a lock is no longer needed, it should be destroyed to release any resources associated with it. This is done using omp_destroy_lock

Locks

```
omp_lock_t myLock;                    // the lock object
omp_init_lock(&myLock);               // initialize to unset
#pragma omp parallel
{
        omp_set_lock(&myLock);        // set lock (wait if already locked)
        // do some work
        omp_unset_lock(&myLock);      // unset lock
}
omp_destroy_lock(&myLock);            // destroy/deallocate lock
```

Locks vs Critical vs Atomic

- **Locks** provide **general-purpose synchronization** and can be used to protect arbitrary sections of code or data structures. They offer **flexibility** but may **incur higher overhead** due to lock acquisition and release.

- **Critical sections are simpler** to use than locks and are **suitable for protecting small, frequently accessed sections of code or variables**. They are **automatically managed** by the OpenMP runtime system.

- **Atomic operations are specialized** for **specific operations on scalar variables** and are **more efficient** than locks or critical sections for those operations. However, they are **limited in scope** and cannot protect larger sections of code or more complex data structures.

Profiling

- An analysis that may measure usage of instructions in terms of their frequency or duration of execution. The goal of profiling is to aid program optimization.

```
double start_time = omp_get_wtime();
#pragma omp parallel [...]
// code
double time = omp_get_wtime() - start_time;
```

- **double start_time = omp_get_wtime();**: This line initializes a variable start_time of type double to store the start time of the parallel section.
- **double time = omp_get_wtime() - start_time;**: After the parallel section of code, this line calculates the total execution time of the parallel region. It subtracts the start time (start_time) obtained at the beginning of the parallel section from the current time
- can be used to analyze the performance of the parallelized code.