Parallel and Distributed Computing

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

Shared Memory Programming Models

OpenMP

Work Sharing Directives

- Always occur within a parallel region directive.
- Common are
  - parallel for
  - parallel section
  - Parallel task

OpenMP Parallel For

```
#pragma omp parallel
    #pragma omp for
    for( … ) { … }
```

- The **parallel** directive creates a parallel region where multiple threads are spawned.
- The **for** directive divides the iterations of the loop among the threads created in the parallel region
- OpenMP automatically handles distributing these iterations to balance the workload between threads.
- **All threads wait at the end of the parallel for.**

Example

```c
int main() {
    int N = 8;

    #pragma omp parallel
    {
        #pragma omp for
        for (int i = 0; i < N; i++) {
            printf("Iteration %d is executed by thread %d\n", i, omp_get_thread_num());
        }
    }

    return 0;
}
```

```
Iteration 0 is executed by thread 1
Iteration 1 is executed by thread 0
Iteration 2 is executed by thread 2
Iteration 3 is executed by thread 3
Iteration 4 is executed by thread 1
Iteration 5 is executed by thread 0
Iteration 6 is executed by thread 2
Iteration 7 is executed by thread 3
```

Default number of Threads

- If you do not **explicitly** specify the number of threads in an OpenMP program.

- OpenMP typically creates a number of threads equal to the number of **available CPU cores** on the system, although this can vary depending on the **OpenMP runtime** and **environment**.

- If **OMP_NUM_THREADS** is set, OpenMP will use this value as the default number of threads in all parallel regions where you don't explicitly set a thread count.

Implicit Barrier

- In OpenMP all threads wait at the end of a parallel for loop by default. This waiting point is called an **implicit barrier**.
- When you use the #pragma omp for directive inside a #pragma omp parallel region, OpenMP divides the loop iterations among the threads.
- Once a thread finishes its assigned iterations, it waits at an implicit barrier at the end of the for loop until all threads have completed their iterations.
- Only after all threads reach this barrier will they proceed to execute any code that follows the loop.

Implicit Barrier Example

```c
int main() {
    #pragma omp parallel
    {
        #pragma omp for
        for (int i = 0; i < 8; i++) {
            printf("Iteration %d executed by thread %d\n", i, omp_get_thread_num());
        }

        // Code here executes after all threads complete the loop
        if (omp_get_thread_num() == 0) {
            printf("All threads completed the loop\n");
        }
    }
    return 0;
}
```

```
Iteration 0 executed by thread 0
Iteration 1 executed by thread 1
Iteration 2 executed by thread 2
...
Iteration 7 executed by thread 3
All threads completed the loop
```

- The loop output will vary because of parallel execution, but **the message "All threads completed the loop"** will only print after all threads have completed their work.

Removing the Implicit Barrier: nowait

- If you want threads to proceed without waiting for each other at the end of the for loop, you can add the nowait clause
- With nowait, threads don’t wait for each other, so some threads may reach the "end of the loop" message before others complete the for loop.

```
int main() {
    #pragma omp parallel
    {
        #pragma omp for nowait
        for (int i = 0; i < 8; i++) {
            printf("Iteration %d executed by thread %d\n", i, omp_get_thread_num());
        }

        // This message might print before all threads complete the loop
        printf("Thread %d reached the end of the loop\n", omp_get_thread_num());
    }
    return 0;
}
```

Multiple for Loops in a Single parallel Region

```c
int main() {
    #pragma omp parallel
    {
        // First parallelized for loop
        #pragma omp for
        for (int i = 0; i < 4; i++) {
            printf("First loop, iteration %d executed by thread %d\n", i, omp_get_thread_num());
        }

        // Implicit barrier here: all threads wait before moving to the next loop

        // Second parallelized for loop
        #pragma omp for
        for (int j = 0; j < 4; j++) {
            printf("Second loop, iteration %d executed by thread %d\n", j, omp_get_thread_num());
        }

        // Implicit barrier here as well, but end of parallel region
    }

    return 0;
}
```

Multiple for Loops in a Single parallel Region

- If you want to remove the **barrier** between loops, you can add the nowait clause to the first for loop

```c
int main() {
    #pragma omp parallel
    {
        // First parallelized for loop
        #pragma omp for
        for (int i = 0; i < 4; i++) {
            printf("First loop, iteration %d executed by thread %d\n", i, omp_get_thread_num());
        }

        // Implicit barrier here: all threads wait before moving to the next loop

        // Second parallelized for loop
        #pragma omp for
        for (int j = 0; j < 4; j++) {
            printf("Second loop, iteration %d executed by thread %d\n", j, omp_get_thread_num());
        }

        // Implicit barrier here as well, but end of parallel region
    }

    return 0;
}
```

```
First loop, iteration 0 executed by thread 1
First loop, iteration 1 executed by thread 2
First loop, iteration 2 executed by thread 3
First loop, iteration 3 executed by thread 0
Second loop, iteration 0 executed by thread 1
Second loop, iteration 1 executed by thread 2
Second loop, iteration 2 executed by thread 3
Second loop, iteration 3 executed by thread 0
```

A Useful Shorthand

```
#pragma omp parallel
#pragma omp for
for ( ; ; ) { … }
```

is equivalent to

```
#pragma omp parallel for
for ( ; ; ) { … }
```

This is more concise and generally preferred when you're only parallelizing a single for loop, as it combines the parallel and for directives into one line, making the code easier to read.

Note the Difference between …

```
#pragma omp parallel
{
   #pragma omp for
   for( ; ; ) { … }
   #pragma omp for
   for( ; ; ) { … }
}
```

```
#pragma omp parallel for
for( ; ; ) { … }
#pragma omp parallel for
for( ; ; ) { … }
```

Comparision

- **Single Parallel Region**:
  - Only one #pragma omp parallel directive is used, meaning that a **single team** of threads is created at the start and reused for both for loops.
  - Each #pragma omp for directive has **an implicit barrier** at the end
  - Since **threads** are **created only once** at the start of the parallel region, this can be more efficient
- **Separate Parallel Regions:**
  - Each #pragma omp parallel for creates its own **separate parallel region**, meaning **threads are created for each loop** individually
  - Since each loop is in its own parallel region, the **threads do not wait** for each other between the two for loops
  - Each #pragma omp parallel for **may involve overhead for creating and destroying threads for each loop**.

Comparision

- **Single** #pragma omp parallel region with multiple for:
  - Efficient for sequential parallel loops with implicit barriers between them.
  - Lower overhead because threads are created once.
  - When there is thread-private data that needs to persist.
- **Multiple** #pragma omp parallel for regions:
  - Independent or unrelated loops, where each loop can run in isolation.
  - When thread-private data is not needed across loops.
  - When you want to adjust the number of threads differently for each loop.

Parallel Sections Directive

```
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
        { … }
        #pragma omp section
        { … }
        …
    }
}
```

- The sections construct is a non-iterative worksharing construct that contains a set of structured blocks that are to be distributed among and executed by the threads in a team.
- Each structured block is executed once by one of the threads in the team.

- A section directive must not appear outside the lexical extent of the sections directive.

Example of work-sharing “sections”

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

int main() {
    // Define two variables to hold results
    int result1 = 0;
    int result2 = 0;

    // Start parallel region with sections
    #pragma omp parallel sections
    {
        // Section 1
        #pragma omp section
        {
            result1 = 10;
            printf("Section 1 executed by thread %d, result1 = %d\n", omp_get_thread_num(), result1);
        }

        // Section 2
        #pragma omp section
        {
            result2 = 20;
            printf("Section 2 executed by thread %d, result2 = %d\n", omp_get_thread_num(), result2);
        }
    } // End of parallel sections

    printf("Final results: result1 = %d, result2 = %d\n", result1, result2);
    return 0;
}
```

```
Section 1 executed by thread 0, result1 = 10
Section 2 executed by thread 1, result2 = 20
Final results: result1 = 10, result2 = 20
```

Example of sections with nowait

- the **nowait** clause can be used with sections to indicate that **threads don’t need to wait** for all sections to complete before moving on to the next code block.
- By default, when using sections, there is an implicit barrier at the end, meaning all threads wait until every section is finished.
- Adding **nowait removes this barrier**, allowing threads to continue without waiting.
- Because there’s no barrier, code following the sections block (like the print statements) may execute before all sections are complete.

Example of `sections` with `nowait`

```
#pragma omp parallel

  {
    #pragma omp sections nowait
    {
      #pragma omp section
        for (i=0; i<n-1; i++)
            b[i] = (a[i] + a[i+1])/2;

      #pragma omp section
        for (i=0; i<n; i++)
            d[i] = 1.0/c[i];

    } /*-- End of sections --*/

  } /*-- End of parallel region --*/
```

Sections Example

Considering following scenario:

- p = pcibus(); 1
- n = networkCard(p);
- w = wifiCard(p); 2
- s = ssh(n,w);
- h = http(n,w); 3
- f = ftp(n,w);

- n, w can be executed in parallel
- s, h, and f can be executed in parallel

Sections Example

```
p = pcibus();
#pragma omp parallel sections num_threads(2)
{
    #pragma omp section
    n = networkCard(p);
    #pragma omp section
    w = wifiCard(p);
}
#pragma omp parallel sections num_threads(3)
{
    #pragma omp section
    s = ssh(n,w);
    #pragma omp section
    h = http(n,w);
    #pragma omp section
    f = ftp(n,w);
}
```

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.