Parallel and Distributed Computing

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

Shared Memory Programming Models

OpenMP

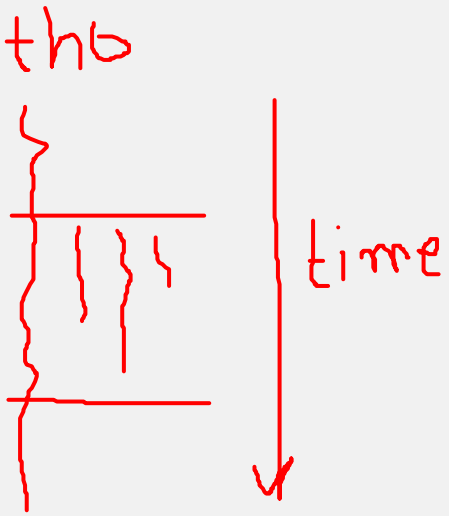
General Rules about Directives

- They always apply to the next statement, which must be a structured block.
- Examples
  - ```
    #pragma omp construct [clause …]
    statement
    ```
  - ```
    #pragma omp construct [clause …]
    { statement1; statement2; statement3; }
    ```

OpenMP Parallel Region

- Defines a parallel region, which is code that will be executed by multiple threads in parallel.

```
#pragma omp parallel [clauses]
{
    code_block
}
```



- (Optional) Zero or more clauses.
- Each thread executes the same code.
- Each thread waits at the end.

Creating Parallel Regions

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

int main() {
    #pragma omp parallel
      printf("Hello World");
}
```

- pragmas are case sensitive
- Same work by all threads

#pragma omp parallel: This directive creates a team of threads. By default, the number of threads is determined by:

1. **Environment variable** OMP_NUM_THREADS (if set).
2. **num_threads()** clause if provided.
3. **Default**: If neither is specified, OpenMP typically uses the number of available CPU cores.

Creating Parallel Regions

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

int main() {
    #pragma omp parallel num_threads(4)
    {
        int i = omp_get_thread_num();
        printf("Hello from thread %d\n", i);
    }
}
```

```
Hello from thread 0
Hello from thread 1
Hello from thread 2
Hello from thread 3
```

- This code is non-deterministic and will produce different results on different runs
- The output of the code will depend on how the threads interleave their operations and when the printf statement is executed.

Creating Parallel Regions

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

int main() {
int x = 5;
#pragma omp parallel num_threads(4)
    {
     x++;
    }
 printf("Value of x is %d\n", x);
}
```

Parallel Regions

```
double D[1000]; // Every value of D is 1
#pragma omp parallel num_threads(5)
{
  int i; double sum = 0;
  for (i=0; i<1000; i++)
  sum += D[i];
  printf("Thread %d computes %f\n",omp_get_thread_num(),sum);
}
```

```
Thread 0 computes 1000.000000
Thread 1 computes 1000.000000
Thread 2 computes 1000.000000
Thread 3 computes 1000.000000
Thread 4 computes 1000.000000
```

(Note: The order of output may vary due to concurrent execution.)

Example

```
int sum = 0;
#pragma omp parallel num_threads(5)
{
    int i; int sum = 0;
    for (i = 0; i < 1000; i++) sum += 1;
    printf("Thread %d computes %d\n", omp_get_thread_num(), sum);
}
    printf("Sum = %d\n", sum);
```

In this code snippet, each thread computes the sum of 1000 elements, each initialized to 1, independently. However, there's a local variable sum declared inside the parallel region, which shadows the outer sum variable. Therefore, the outer sum variable remains uninitialized and its value is unpredictable.

Getting Threads to do Different Things

- Through explicit thread identification (as in Pthreads).
- Through work-sharing directives.

Thread Identification

```
int omp_get_thread_num()
int omp_get_num_threads()
```

- Gets the thread id.
- Gets the total number of threads.

Example

```
int main() {

    #pragma omp parallel num_threads(4)
    {
            // Get the thread ID
            int thread_id = omp_get_thread_num();

            // Get the total number of threads
            int num_threads = omp_get_num_threads();

        printf("Hello from thread %d out of %d threads\n", thread_id, num_threads);
    }
    return 0;
}
```

```
Hello from thread 0 out of 4 threads
Hello from thread 1 out of 4 threads
Hello from thread 2 out of 4 threads
Hello from thread 3 out of 4 threads
```

Example

```
void doThis() {
printf("Thread %d is executing doThis()", omp_get_thread_num());
}

void doThat() {
printf("Thread %d is executing doThat()", omp_get_thread_num());
}
int main() {
#pragma omp parallel num_threads(4)
{
   if( omp_get_thread_num() >1)
       doThis();
   else
       doThat();
} return 0;}
```

```
Thread 0 is executing doThat()
Thread 1 is executing doThat()
Thread 2 is executing doThis()
Thread 3 is executing doThis()
```

Work Sharing Directives

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