

OpenMP

Introduction:

OpenMP (Open Multi-Processing) is an API that allows developers to write parallel code for shared-memory systems. It simplifies the process of creating multithreaded applications in languages like C, C++, and Fortran. The goal of OpenMP is to improve the performance of applications by dividing tasks across multiple processor cores.

- **Parallelism Type:**
 - OpenMP works with **threads** in a **shared-memory** model, meaning all threads operate within the same address space and can directly access shared variables.
- **Ease of Use:**
 - **High-level abstraction:** OpenMP uses compiler directives (`#pragma`) to define parallel regions, making it easy to convert sequential code into parallel code. No need to manually manage threads or processes.
 - It automatically handles thread creation, work division, and synchronization for basic parallelism.
- **Target Use Case:**
 - Best for simple, loop-based parallelism where workload is easily divisible (e.g., parallelizing loops).
- **Synchronization:**
 - Built-in support for synchronization primitives like critical sections, barriers, locks, and reduction.

Objective:

- To introduce students to parallel programming concepts using OpenMP.
 - To demonstrate basic constructs such as parallel regions, synchronization, and reduction.
-

Prerequisites:

- Basic understanding of C/C++ programming.
 - Basic knowledge of multithreading and parallel programming concepts.
-

Lab Steps:

Step1: Setting Up OpenMP

- **Compiling an OpenMP Program:**
 - Demonstrate how to set up OpenMP in a C/C++ environment. For example, when using GCC:

```
gcc -fopenmp program.c -o program
```

- **Run a simple "Hello World" example:**

- Example Code:

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

int main() {
    #pragma omp parallel
    {
        printf("Hello from thread %d\n",
omp_get_thread_num());
    }
    return 0;
}
```

- This simple program will print out which thread is running, introducing them to the concept of threads and parallelism.
-

Step2: Parallelism and Work Distribution

- **Parallel Regions:**
 - When OpenMP uses the `#pragma omp parallel` directive, it creates a **parallel region** in the code. In this parallel region, multiple threads are created, and they execute the block of code inside the parallel region simultaneously. This is one of the simplest ways to start parallel execution in OpenMP.
- **How `#pragma omp parallel` Works:**

1. Thread Creation:

- When the program encounters a `#pragma omp parallel` directive, OpenMP creates a **team of threads**. The number of threads is usually determined by the system or can be explicitly controlled by the programmer (using the `omp_set_num_threads()` function or environment variables).

2. Parallel Region:

- The block of code immediately following the `#pragma omp parallel` directive becomes a **parallel region**, meaning all the threads in the team will execute that block concurrently.
- Every thread runs the **same code** inside the parallel region, but each thread can behave differently if the code depends on the thread's unique identifier (which can be obtained using `omp_get_thread_num()`).

3. Execution by Multiple Threads:

- All threads execute the code within the parallel block **simultaneously**, with each thread having its own instance of local variables declared inside the block. Shared variables (outside the block) can be accessed by all threads unless specific sharing clauses are defined.

4. Thread Synchronization:

- By default, when all the threads finish executing the parallel region, the program resumes **sequential execution** with the master thread (the original thread that started before entering the parallel region).

- **Exercise 1:**

- modify the previous "Hello World" program by adding `omp_set_num_threads(threads_num)` to set thread to use and `omp_get_num_threads()` to display the total number of threads. This will give them an idea of how many threads are being created:

```
omp_set_num_threads(3);

#pragma omp parallel

{

    printf("Thread with previous set %d out of %d\n",
omp_get_thread_num(), omp_get_num_threads());

}

#pragma omp parallel num_threads(5)

{

    printf("Thread %d out of %d\n", omp_get_thread_num(),
omp_get_num_threads());

}
```

- **Parallelizing Loops (Work-sharing):**

- Introduce the `#pragma omp` for directive, which distributes loop iterations among threads. OpenMP automatically divides the loop iterations across available threads.
- Example Code (Parallel Loop):

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

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

- **Exercise 2:**
 - create a program that sums an array using OpenMP with a parallel loop.
-

Step3: Race Conditions

- Here is a basic example of code that demonstrates a race condition:

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

int main() {
    int count = 0;

    #pragma omp parallel for
    for (int i = 0; i < 1000; i++) {
        count++; // Potential race condition
    }

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

- Race Condition in Action:

In a perfect scenario where no race condition occurs, the final value of `count` should be 1000. However, due to the **race condition**, the actual result may be much less than 1000, and it will vary each time the program is executed.

- Why Does the Race Condition Happen?

A race condition occurs here because:

The statement `count++` involves **three steps**:

1. **Read** the current value of `count`.
2. **Increment** the value.
3. **Write** the new value back to `count`.

In a multithreaded environment:

- **Multiple threads** may simultaneously read the value of `count`, increment it, and write the result back, **without knowing** that other threads are doing the same thing.
- For example, two threads might read `count = 5` at the same time, both increment it to 6, and both write 6 back to `count`. This means one of the increments is **lost**, resulting in incorrect behavior.

- **Solution to the Race Condition:**

To avoid race conditions, you can use **synchronization mechanisms** like OpenMP's `critical` section or atomic operations to ensure that only one thread increments the variable at a time.

Fixing the Race Condition:

One simple solution is to use OpenMP's `#pragma omp atomic` directive to make the increment operation atomic, ensuring that it is executed by only one thread at a time.

Fixed Example with atomic:

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

int main() {
    int count = 0;

    #pragma omp parallel for
    for (int i = 0; i < 1000; i++) {
```

```

        #pragma omp atomic
        count++; // Fixed: No race condition
    }

    printf("Final count: %d\n", count);
    return 0;
}

///// or
#pragma omp parallel for
for(int i = 0; i < 1000; i++) {
    #pragma omp critical
    {
        count++;
    }
}

```

Explanation of the Fix:

- By adding `#pragma omp atomic`, we ensure that the increment operation is performed atomically, meaning no two threads can update `count` at the same time.
 - Now the final result will always be 1000, no matter how many threads are running, because the operation is protected from concurrent access.
-

Step4: Reduction

- In OpenMP, the **reduction clause** is used when multiple threads need to update a shared variable, such as a sum, product, or other accumulations, without causing **race conditions**. When using the reduction clause, OpenMP ensures that each thread works on a **local copy** of the shared variable. After all threads complete their execution, OpenMP combines the results from all threads in a **safe and efficient manner** into the shared variable. This prevents race conditions while allowing the threads to perform the accumulation in parallel.
- **How Reduction Works:**
 1. Each thread is given a **private copy** of the shared variable.
 2. Each thread operates on its own local copy during the execution of the parallel region.
 3. Once all threads complete their tasks, OpenMP **reduces** all the local copies into a single result using the specified operator (like `+`, `*`, etc.).
 4. The final result is stored in the original shared variable.
- **Common Operators for Reduction:**
 - `+` (addition)

- * (multiplication)
- - (subtraction)
- & (bitwise AND)
- | (bitwise OR)
- ^ (bitwise XOR)

- **Syntax:** `#pragma omp parallel for reduction(operator:shared_variable)`

- **Example Code Using Reduction:**

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

int main() {
    int sum = 0;

    #pragma omp parallel for reduction(+:sum)
    for (int i = 0; i < 1000; i++) {
        sum += i; // Reduction avoids race condition
    }

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

Explanation of the Code:

- We declare a shared variable `sum` initialized to 0.
- The `#pragma omp parallel for reduction(+:sum)` directive creates a parallel loop where multiple threads calculate the sum.
- The **reduction** clause (`+:sum`) ensures that each thread works on a local copy of `sum`. After the loop finishes, the local sums from each thread are added together and stored in the original `sum` variable.

What Happens Internally:

- **Thread-private copies:** Each thread is given its own copy of the variable `sum`, which it uses to accumulate the values for its iterations of the loop.
 - **Final Reduction:** After all threads complete their iterations, OpenMP **combines** the thread-private sums into the original `sum` variable using the addition (+) operator.
 - This prevents any race condition that would have occurred if multiple threads tried to update `sum` at the same time.
-

Step5: Locks

- In OpenMP, **locks** are used to protect shared variables or critical sections of the code from being accessed by multiple threads at the same time. Locks are typically used in scenarios where threads need to update shared resources (such as variables or data structures), and you want to ensure that only one thread can perform the update at any given moment. This prevents data corruption and ensures **safe access** to shared resources.
- **How Locks Work:**
 - **Acquiring a Lock:** Before a thread can enter a critical section (the part of the code that accesses shared resources), it must acquire the lock. If another thread has already acquired the lock, the current thread will be blocked until the lock becomes available.
 - **Releasing the Lock:** After finishing its task, the thread releases the lock, allowing other threads to acquire it and enter the critical section.
- **Common Lock Functions in OpenMP:**
 - **omp_init_lock(&lock):** Initializes the lock before it is used.
 - **omp_set_lock(&lock):** Acquires the lock. If another thread has the lock, the current thread waits until it is released.
 - **omp_unset_lock(&lock):** Releases the lock, allowing other threads to acquire it.
 - **omp_destroy_lock(&lock):** Destroys the lock after it is no longer needed.
- **Example**

```
#include <omp.h>

#include <stdio.h>

int main() {

    int count = 0;

    omp_lock_t lock;

    // Initialize the lock

    omp_init_lock(&lock);

    // Parallel region with lock to control access to 'count'

    #pragma omp parallel for

    for (int i = 0; i < 1000; i++) {

        // Acquire the lock before modifying 'count'
```



```

        omp_set_lock(&lock);

        count++; // Critical section

        // Release the lock after modification

        omp_unset_lock(&lock);
    }

    // Destroy the lock after use

    omp_destroy_lock(&lock);

    printf("Final count: %d\n", count);

    return 0;
}

```

Step6: OpenMP's Importance

Parallel vs. Sequential Performance Comparison

Example: Summing a Large Array

- **Sequential Approach**

```

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

#define ARRAY_SIZE 100000000 // 100 million elements

int main() {
    int array[ARRAY_SIZE];
    long long sum = 0;

    // Initialize array with values
    for (int i = 0; i < ARRAY_SIZE; i++) {
        array[i] = i + 1; // Example: Summing numbers from 1 to
        ARRAY_SIZE
    }

    // Measure time for sequential sum
    double start = omp_get_wtime();

```

```

        // Sequential sum calculation
        for (int i = 0; i < ARRAY_SIZE; i++) {
            sum += array[i];
        }

        double end = omp_get_wtime();
        printf("Sequential Sum: %lld\n", sum);
        printf("Time taken (Sequential): %f seconds\n", end - start);

        return 0;
    }

```

- **Parallel Approach using OpenMP**

```

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

#define ARRAY_SIZE 100000000 // 100 million elements

int main() {
    int array[ARRAY_SIZE];
    long long sum = 0;

    // Initialize array with values
    for (int i = 0; i < ARRAY_SIZE; i++) {
        array[i] = i + 1; // Example: Summing numbers from 1 to
        ARRAY_SIZE
    }

    // Measure time for parallel sum
    double start = omp_get_wtime();

    // Parallel sum calculation
    #pragma omp parallel for reduction(+:sum)
    for (int i = 0; i < ARRAY_SIZE; i++) {
        sum += array[i];
    }

    double end = omp_get_wtime();
    printf("Parallel Sum: %lld\n", sum);
    printf("Time taken (Parallel): %f seconds\n", end - start);

    return 0;
}

```
