**Task Level Papalism using OpenMP**

Let's consider a more practical example where Task 1 and Task 2 are independent of each other, but Task 3 is dependent on the completion of both Task 1 and Task 2. This kind of scenario is common when tasks involve data processing that needs to be done in parallel, and the results from two independent tasks are combined in a third task.

In OpenMP, we can achieve this using dependencies with the depend clause in the #pragma omp task directive. This ensures that Task 3 will not start until both Task 1 and Task 2 have completed.

**Practical Example:**

#include <stdio.h>

#include <omp.h>

void task1(int \*result) {

printf("Task 1 started\n");

// Simulate some computation

for (int i = 0; i < 100000000; i++);

\*result = 10; // Simulate some result from task 1

printf("Task 1 finished\n");

}

void task2(int \*result) {

printf("Task 2 started\n");

// Simulate some computation

for (int i = 0; i < 100000000; i++);

\*result = 20; // Simulate some result from task 2

printf("Task 2 finished\n");

}

void task3(int result1, int result2) {

printf("Task 3 started\n");

// Simulate combining results from task 1 and task 2

int combined\_result = result1 + result2;

printf("Task 3 finished. Combined result: %d\n", combined\_result);

}

int main() {

int result1 = 0, result2 = 0;

// Set the number of threads to use

omp\_set\_num\_threads(4);

// Parallel region where tasks will be defined

#pragma omp parallel

{

#pragma omp single // Ensures that only one thread enters this section

{

// Create independent tasks

#pragma omp task shared(result1)

task1(&result1);

#pragma omp task shared(result2)

task2(&result2);

// Create a dependent task that depends on both task1 and task2

#pragma omp task depend(in: result1) depend(in: result2) depend(out: result1)

task3(result1, result2);

}

}

return 0;

}

**Explanation:**

1. **Task 1 and Task 2**:
   * task1 and task2 are independent tasks. They simulate some computation (in this case, a simple loop) and store results in result1 and result2 respectively.
2. **Task 3**:
   * task3 depends on the results from both task1 and task2. It combines result1 and result2 to produce a final result.
3. **Dependencies**:
   * #pragma omp task depend(in: result1) ensures that task3 will wait for task1 to finish and for the result1 to be updated.
   * #pragma omp task depend(in: result2) ensures that task3 will also wait for task2 to finish and for result2 to be updated.
   * #pragma omp task depend(out: result1) indicates that task3 will modify result1 in some way (though here it is simply reading the value, in practice, result1 could be updated by task 3).
4. **#pragma omp single**:
   * Ensures that the task creation happens from a single thread. All tasks are created within the single block, and the tasks themselves will execute in parallel as available threads are assigned.

**How it works:**

* task1 and task2 execute independently in parallel.
* task3 will wait for both task1 and task2 to finish, as indicated by the depend clauses, before it starts executing.
* The program prints the start and finish messages for each task, and task3 outputs the combined result after both task1 and task2 are completed.

**Output (Order may vary based on thread scheduling):**

Task 1 started

Task 2 started

Task 1 finished

Task 2 finished

Task 3 started

Task 3 finished. Combined result: 30

**Notes:**

* The tasks are scheduled in parallel, and task3 will only start after both task1 and task2 are complete.
* The depend clause in OpenMP ensures proper synchronization and scheduling, avoiding race conditions when task3 depends on the results of the other tasks.

This is a basic example of how task dependencies can be modeled in OpenMP for tasks that need to wait for other tasks to complete before they can start their execution.

**Barrier**

A **barrier** in OpenMP is a synchronization construct that ensures all threads in a parallel region reach a specific point in the code before they can continue executing. In other words, all threads must meet at the barrier before any thread can proceed past it. This is useful when you need to ensure that all threads have completed their work up to a certain point before continuing.

Here’s a simple example demonstrating how a barrier works in OpenMP:

### Example with Barrier:

#include <stdio.h>

#include <omp.h>

void task1() {

printf("Task 1: Thread %d is working...\n", omp\_get\_thread\_num());

// Simulate some work with a loop

for (int i = 0; i < 100000000; i++);

printf("Task 1: Thread %d is done\n", omp\_get\_thread\_num());

}

void task2() {

printf("Task 2: Thread %d is working...\n", omp\_get\_thread\_num());

// Simulate some work with a loop

for (int i = 0; i < 100000000; i++);

printf("Task 2: Thread %d is done\n", omp\_get\_thread\_num());

}

int main() {

// Set the number of threads to use

omp\_set\_num\_threads(4);

// Parallel region

#pragma omp parallel

{

task1(); // Thread will execute task1

// Barrier to synchronize threads

#pragma omp barrier

task2(); // After barrier, all threads will execute task2

}

return 0;

}

### Explanation:

1. **#pragma omp parallel**: This is the parallel region, where multiple threads are created.
2. **Task 1**: Each thread executes task1(), and we simulate some work by running a simple loop.
3. **#pragma omp barrier**: This is the barrier. All threads in the parallel region will reach this point, and they will not proceed to task2() until all threads have completed task1(). The barrier ensures synchronization.
4. **Task 2**: After the barrier, all threads will execute task2(), but only after every thread has finished executing task1().

### Output (may vary in order):

Task 1: Thread 0 is working...

Task 1: Thread 1 is working...

Task 1: Thread 2 is working...

Task 1: Thread 3 is working...

Task 1: Thread 3 is done

Task 1: Thread 1 is done

Task 1: Thread 2 is done

Task 1: Thread 0 is done

Task 2: Thread 0 is working...

Task 2: Thread 1 is working...

Task 2: Thread 2 is working...

Task 2: Thread 3 is working...

Task 2: Thread 3 is done

Task 2: Thread 1 is done

Task 2: Thread 2 is done

Task 2: Thread 0 is done

### How it works:

* Each thread first executes task1.
* When all threads reach the barrier (#pragma omp barrier), they all stop at the barrier point, and no thread can proceed until every thread has reached that point.
* Once all threads have reached the barrier, they can all proceed to task2, and all threads will start task2 after the barrier is released.

### Use of Barriers:

Barriers are useful when you need to ensure that certain sections of code are fully completed by all threads before moving on to subsequent work. For example, after one phase of computations, you might need to synchronize all threads before starting a second phase that depends on the results of the first.

In this example, the barrier ensures that task1 is completed by all threads before any thread starts task2.