**Reading Assignment : Understanding Shared Variables and Their Protection Mechanisms in OpenMP**

**Objective:**

In this assignment, you will explore how shared variables work in parallel programming using OpenMP. You will learn how to use **reduction**, **atomic**, **critical**, and **locks** to protect shared variables during parallel execution. By comparing these different mechanisms, you will understand how they impact performance and correctness.

**Background:**

In parallel programming, when multiple threads access and modify shared variables, there can be **data races**—situations where the outcome depends on the timing of thread execution. OpenMP provides several methods for managing shared variables safely and efficiently:

* **reduction**: Automatically creates private copies of the variable for each thread and combines the results in a thread-safe manner after the parallel region.
* **atomic**: Ensures that updates to a shared variable are done atomically, preventing data races without requiring heavy synchronization overhead.
* **critical**: Ensures that only one thread at a time can access the protected code block, which is useful for ensuring exclusive access to shared variables.
* **locks**: Provide a manual synchronization mechanism where threads acquire and release locks to access shared variables, but with more overhead.

**Problem Statement:**

You are given an array of integers, and your task is to compute the **sum** of the array in parallel using OpenMP. You will implement the solution using the following approaches and compare their performance:

1. **Parallel Sum with Reduction**: Use OpenMP’s reduction clause to safely compute the sum.
2. **Parallel Sum with Atomic Operations**: Use OpenMP’s atomic clause to prevent data races when updating the sum.
3. **Parallel Sum with Critical Section**: Use OpenMP’s critical directive to protect the shared sum variable.
4. **Parallel Sum with Locks**: Use OpenMP locks (omp\_lock\_t) to ensure that only one thread updates the sum at a time.
5. **Parallel Sum without Synchronization**: Perform parallel summation without any synchronization (to observe the impact of data races).

**Tasks:**

1. **Matrix Setup:**
   * Initialize an array arr[] of size n with random integers. For simplicity, you can initialize the array with random integers between 1 and 100.
2. **Implement the Following Approaches:**
   * **Serial Sum**: First, implement a serial version of the sum of the array, without any parallelism, to establish a baseline.
   * **Parallel Sum with Reduction**: Use OpenMP’s reduction clause to compute the sum. This will automatically handle thread-local variables and combine them.
   * **Parallel Sum with Atomic**: Use OpenMP’s atomic to update the shared sum variable safely, without using critical sections.
   * **Parallel Sum with Critical**: Use OpenMP’s critical section to ensure that only one thread at a time updates the shared sum variable.
   * **Parallel Sum with Locks**: Use OpenMP locks (omp\_lock\_t) to ensure mutual exclusion when updating the shared sum.
   * **Parallel Sum without Synchronization**: Implement the summation without any synchronization and observe the incorrect output due to data races.
3. **Performance Measurement:**
   * Measure the execution time for the serial sum and each parallel version (reduction, atomic, critical, locks, and unsynchronized).
   * Compare the execution times for different parallel implementations.
4. **Results Analysis:**
   * Print the results for the sum of the array and the execution times of each approach.
   * Discuss the performance differences between each approach. Pay attention to which methods are more efficient and why.

**Guidelines:**

* Use omp\_get\_wtime() to measure execution time.
* Use the reduction, atomic, critical, and lock clauses appropriately in the respective parallel regions.
* For the **Parallel Sum without Synchronization**, you should not use any synchronization mechanisms (i.e., atomic, critical, or reduction). You are encouraged to use #pragma omp parallel for without any additional directives.

**Code Template:**

#include <iostream>

#include <vector>

#include <cstdlib>

#include <ctime>

#include <omp.h>

int main() {

int n = 1000000; // Example array size

std::vector<int> arr(n);

int sum = 0; // Shared sum variable

// Initialize the array with random values between 1 and 100

srand(time(0));

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

arr[i] = rand() % 100 + 1;

}

// Serial sum for baseline comparison

double start\_time = omp\_get\_wtime();

int serial\_sum = 0;

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

serial\_sum += arr[i];

}

double end\_time = omp\_get\_wtime();

std::cout << "Serial Sum: " << serial\_sum << std::endl;

std::cout << "Serial Execution Time: " << end\_time - start\_time << " seconds." << std::endl;

// Parallel sum with reduction

start\_time = omp\_get\_wtime();

sum = 0;

#pragma omp parallel for reduction(+:sum)

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

sum += arr[i];

}

end\_time = omp\_get\_wtime();

std::cout << "Parallel Sum with Reduction: " << sum << std::endl;

std::cout << "Parallel Execution Time (Reduction): " << end\_time - start\_time << " seconds." << std::endl;

// Parallel sum with atomic

start\_time = omp\_get\_wtime();

sum = 0;

#pragma omp parallel for

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

#pragma omp atomic

sum += arr[i];

}

end\_time = omp\_get\_wtime();

std::cout << "Parallel Sum with Atomic: " << sum << std::endl;

std::cout << "Parallel Execution Time (Atomic): " << end\_time - start\_time << " seconds." << std::endl;

// Parallel sum with critical section

start\_time = omp\_get\_wtime();

sum = 0;

#pragma omp parallel for

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

#pragma omp critical

sum += arr[i]; // Only one thread at a time can update 'sum'

}

end\_time = omp\_get\_wtime();

std::cout << "Parallel Sum with Critical: " << sum << std::endl;

std::cout << "Parallel Execution Time (Critical): " << end\_time - start\_time << " seconds." << std::endl;

// Parallel sum with locks

start\_time = omp\_get\_wtime();

sum = 0;

omp\_lock\_t lock;

omp\_init\_lock(&lock);

#pragma omp parallel for

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

omp\_set\_lock(&lock);

sum += arr[i];

omp\_unset\_lock(&lock);

}

omp\_destroy\_lock(&lock);

end\_time = omp\_get\_wtime();

std::cout << "Parallel Sum with Locks: " << sum << std::endl;

std::cout << "Parallel Execution Time (Locks): " << end\_time - start\_time << " seconds." << std::endl;

// Parallel sum without synchronization (Data Race)

start\_time = omp\_get\_wtime();

sum = 0;

#pragma omp parallel for

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

sum += arr[i]; // No synchronization, potential data race

}

end\_time = omp\_get\_wtime();

std::cout << "Parallel Sum without Synchronization (Data Race): " << sum << std::endl;

std::cout << "Parallel Execution Time (No Sync): " << end\_time - start\_time << " seconds." << std::endl;

return 0;

}

**Expected Results:**

* **Serial Execution**: This will serve as your baseline. The result should be the correct sum of the array.
* **Parallel Sum with Reduction**: This should give the correct sum with fast execution, as OpenMP handles the combining of results automatically.
* **Parallel Sum with Atomic**: This will also give the correct sum, but with a slight performance cost due to atomic operations ensuring thread-safe updates.
* **Parallel Sum with Critical**: This should give the correct sum, but performance will be slower compared to other approaches because only one thread can access the sum at a time.
* **Parallel Sum with Locks**: This should also give the correct sum, but with more overhead due to the lock acquisition and release in each iteration.
* **Parallel Sum without Synchronization**: This will likely give an incorrect result due to a **data race** (inconsistent updates to the sum variable), and performance should be worse than other methods due to the lack of synchronization.

**Analysis and Discussion:**

After running the program, compare the results for the sum and the execution time for each parallel approach:

* **Compare correctness**: Identify which approaches result in correct sums and which ones do not (the unsynchronized version will show incorrect results).
* **Performance analysis**: Discuss the trade-offs between the different methods. Consider which method would scale better for larger arrays (reduction typically scales the best). Compare the performance of atomic, critical, and locks to reduction, and explain why one might be faster than the other.

**Submission:**

* Submit your source code along with a report that includes:
  + The output of the program (the sum and execution times).
  + A detailed comparison and analysis of the performance of each approach.
  + Discussion of the advantages and drawbacks of each synchronization method (reduction, atomic, critical, locks, and no synchronization).