Parallel and Distributed Computing

Name: Shaik Aarif

Regd No: 2200033144

Experiment 1:

Title: Demonstrate various programs in OpenMP, Parallel global sum

Pre-Lab:

1. Explain the purpose of the critical directive in OpenMP and provide a scenario where it is beneficial?

Ans. The critical directive in OpenMP ensures that a specific section of code is executed by only one thread at a time, preventing race conditions when accessing shared resources. It is beneficial in scenarios like updating a shared counter or writing to a log file, where concurrent access could lead to inconsistent or corrupted results. This guarantees thread safety and maintains data integrity in parallel programs.

2. Describe the role of the master directive in OpenMP. In what situations is it commonly used?

Ans. The master directive in OpenMP specifies that a block of code should only be executed by the master thread (thread 0) in a parallel region. It avoids the overhead of synchronization since only one thread executes the block without involving others.

Common Situations:

1. **Input/Output Operations:** To handle I/O tasks like reading data or printing results to ensure orderly output.
2. **Initialization:** To perform setup tasks before other threads start working.
3. **Finalization:** To aggregate results or perform cleanup after parallel computation.

3. Discuss the significance of the default clause in OpenMP. How does it affect the sharing of variables in a parallel region?

Ans. The default clause in OpenMP specifies the default data-sharing attributes for variables within a parallel region. It determines whether variables are shared or private if their attributes are not explicitly defined.

Significance:

1. **Control over variable scope:** It ensures consistency and reduces ambiguity in how variables are handled.
2. **Error prevention:** It helps avoid unintentional sharing or privatization of variables, which could lead to race conditions or incorrect behavior.

Effects:

- default(shared): All variables are shared unless explicitly privatized.
- default(none): Forces explicit declaration of sharing attributes, enhancing code clarity and safety.

4. What is the purpose of the thread private directive in OpenMP? Provide an example of a situation where using this directive is advantageous.

Ans. The threadprivate directive in OpenMP makes global or static variables private to each thread, ensuring each thread has its own independent copy that persists across parallel regions.

Purpose:

- **Thread-local storage:** Allows threads to maintain separate, persistent states for specific variables across parallel regions.

Advantageous Situation:

When implementing thread-specific contexts or maintaining state between parallel regions, such as storing a thread's intermediate computation

results for later use in another parallel section. For example, each thread maintaining its own random number generator state avoids synchronization overhead and ensures reproducibility.

5. Explain the concept of the reduction clause in OpenMP. How does it facilitate parallel computation?

Ans. The reduction clause in OpenMP simplifies parallel computations involving operations like summation, product, or logical conditions by combining partial results from multiple threads into a single value.

How It Facilitates Parallel Computation:

1. **Automatic Aggregation:** Each thread performs its computation on a private copy of the variable, avoiding race conditions.
2. **Efficient Finalization:** OpenMP automatically combines the thread-local results using the specified operator (e.g., +, *, &&) at the end of the parallel region.
3. **Simplifies Coding:** Reduces the need for manual synchronization, making parallel code cleaner and easier to write.

This is particularly useful in scenarios like calculating the total sum of an array or finding the maximum/minimum value across elements.

In-Lab:

1. Basic programs to demonstrate various #pragrams in OpenMP: Parallel • for • private • shared • critical • MASTER Directive • THREADPRIVATE Directive • DEFAULT Clause • FIRSTPRIVATE Clause •LASTPRIVATE Clause.
    - /* Program to Demonstrate #pragrams in OpenMP: Parallel, for */
    - /* Program to Demonstrate private clause */
    - /* Program to Demonstrate shared clause */
    - /* Program to Demonstrate Critical Directive */
    - /* Program to Demonstrate Master Directive */
    - /* Program to Demonstrate threadprivate Clause */

- /* Program to Demonstrate default Clause */
- /* Program to Demonstrate firstprivate clause */
- /* Program to Demonstrate lastprivate clause */

Program:

Parallel, for:

```c
/* Program to Demonstrate parallel, for */
#include <omp.h>
#include <stdio.h>

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

Private clause:

```c
/* Program to Demonstrate private clause */
#include <omp.h>
#include <stdio.h>

int main() {
    int x = 10;
    #pragma omp parallel private(x)
    {
        x = omp_get_thread_num();
        printf("Thread %d has x = %d\n", omp_get_thread_num(), x);
    }
    return 0;
}
```

Shared clause:

```c
/* Program to Demonstrate shared clause */
#include <omp.h>
#include <stdio.h>

int main() {
    int x = 0;
    #pragma omp parallel shared(x)
    {
        #pragma omp critical
        {
            x += omp_get_thread_num();
            printf("Thread %d updates x to %d\n", omp_get_thread_num(), x);
        }
    }
    printf("Final value of x: %d\n", x);
    return 0;
}

```

Critical Directive:

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

int main() {
    int counter = 0;
    #pragma omp parallel
    {
        for (int i = 0; i < 5; i++) {
            #pragma omp critical
            {
                counter++;
                printf("Thread %d increments counter to %d\n", omp_get_thread_num(),
counter);
            }
        }
    }
    printf("Final counter value: %d\n", counter);
    return 0;
}
```

Master Directive:

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

int main() {
    #pragma omp parallel
    {
        #pragma omp master
        {
            printf("Master thread (thread 0) is executing this code.\n");
        }
    }
    return 0;
}
```

threadprivate Clause:

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

int global_var;
#pragma omp threadprivate(global_var)

int main() {
    global_var = 42;
    #pragma omp parallel
    {
        printf("Thread %d has global_var = %d\n", omp_get_thread_num(), global_var);
        global_var = omp_get_thread_num();
    }

    #pragma omp parallel
    {
        printf("Thread %d retains global_var = %d\n", omp_get_thread_num(),
global_var);
    }
    return 0;
}
```

Default Clause:

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

int main() {
    int x = 10, y = 20;
    #pragma omp parallel default(none) shared(x) private(y)
    {
        y = omp_get_thread_num();
        printf("Thread %d: x = %d, y = %d\n", omp_get_thread_num(), x, y);
    }
    return 0;
}
```

firstprivate clause:

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

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

lastprivate clause:

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

int main() {
    int x;
    #pragma omp parallel for lastprivate(x)
    for (int i = 0; i < 5; i++) {
        x = i;
        printf("Thread %d processes iteration %d\n", omp_get_thread_num(), i);
    }
    printf("Last value of x: %d\n", x);
    return 0;
}
```

2. **Write programs to demonstrate SINGLE Directive BARRIER Directive ATOMIC Directive**
   - **/* Program to Demonstrate single Directive */**
   - **/* Program to demonstrate barrier Directive */**
   - **/* Program to Demonstrate atomic Directive */**

Program:

Single Directive:

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

int main() {
    #pragma omp parallel
    {
        #pragma omp single
        {
            printf("Single directive executed by thread %d\n",
omp_get_thread_num());
        }
    }
    return 0;
}
```

Barrier Directive:

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

int main() {
    #pragma omp parallel
    {
        printf("Thread %d reached the barrier.\n", omp_get_thread_num());
        #pragma omp barrier
        printf("Thread %d passed the barrier.\n", omp_get_thread_num());
    }
    return 0;
}
```

Atomic Directive:

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

int main() {
    int sum = 0;
    #pragma omp parallel for
    for (int i = 0; i < 10; i++) {
        #pragma omp atomic
        sum += i;
    }
    printf("Sum of first 10 integers: %d\n", sum);
    return 0;
}
```

VIVA-VOCE Questions:

1. What does the SINGLE directive in OpenMP accomplish, and in what scenarios would you use it?

Ans. The SINGLE directive in OpenMP ensures that a specific block of code is executed by only one thread in a parallel region. It is used when a task should be performed by just one thread, such as initialization or a non-repetitive task. Other threads in the parallel region skip the SINGLE block.

2. Explain the purpose of the BARRIER directive in OpenMP and why it is essential in parallel programming.

Ans. The BARRIER directive in OpenMP synchronizes all threads in a parallel region, ensuring that no thread can proceed past the barrier until all threads reach it. It is essential in parallel programming to coordinate the execution order of threads, prevent race conditions, and ensure data consistency across threads before moving on to the next phase of computation.

3. What is the role of the ATOMIC directive in OpenMP? Provide an example where using ATOMIC is crucial?

Ans. The ATOMIC directive in OpenMP ensures that a specific memory location is updated atomically, preventing data races when multiple threads attempt to modify the same variable simultaneously. It is crucial in scenarios where multiple threads update a shared variable and you need to guarantee that each update is completed without interference from other threads, such as incrementing a global counter.

Example scenario: Incrementing a shared counter in a parallel loop. Without ATOMIC, multiple threads may try to update the counter simultaneously, leading to inconsistent results.

4. How does the BARRIER directive contribute to ensuring correct synchronization in parallel programs?

Ans. The BARRIER directive ensures that all threads in a parallel region synchronize at a specific point, meaning no thread can proceed past the barrier until all threads have reached it. This synchronization is crucial for ensuring that all previous computations are completed before moving on to subsequent tasks, preventing race conditions and ensuring data consistency in parallel programs. It helps maintain the correct order

of operations and guarantees that dependencies between threads are respected.

5. In what situations would you prefer using the SINGLE directive over the MASTER directive, and vice versa?

Ans. The SINGLE directive is used when a specific block of code should be executed by only one thread, but other threads can still proceed with their execution. It is suitable for tasks that need to be performed once, such as initialization, where only one thread is required.

The MASTER directive, on the other hand, ensures that the code block is executed only by the master thread (typically thread 0) and is skipped by all other threads. It is useful for tasks that should be handled exclusively by the master thread, such as I/O operations or managing parallel work.

In short:

- Use SINGLE when any single thread can handle the task.
- Use MASTER when only the master thread should handle the task.