HPC LAB 05

PRN: 21510042

Omkar Rajesh Auti

CSE

Github Link:

https://github.com/omkarauti11/HPC_LAB

Q1. Write an OpenMP program such that, it should print the name of your family members, such that the names should come from different threads/cores. Also print the respective job id.

Code:

```
#include <iostream>
#include <vector>

int main() {
int N = 1000000;
std::vector<int> array(N, 1);
int sum = 0;

for (int i = 0; i < N; ++i) {
sum += array[i];
}

std::cout << "Sequential sum: " << sum << std::endl;
return 0;
}
```

Output:

```
ubuntu@ubuntu-VirtualBox:~/Documents/Assignment05$ g++ -fopenmp -o a 05_01_a.cpp
ubuntu@ubuntu-VirtualBox:~/Documents/Assignment05$ ./a
Thread ID 3: Diana
Thread ID 1: Bob
Thread ID 2: Charlie
Thread ID 0: Alice
ubuntu@ubuntu-VirtualBox:~/Documents/Assignment05$
```

Q2. Write an OpenMP program such that, it should print the sum of square of the thread id's. Also make sure that, each thread should print the square value of their thread id.

Code:

```
#include <iostream>

#include <omp.h>

int main() {
int sum = 0;

#pragma omp parallel reduction(+:sum)
{
int tid = omp_get_thread_num();
int square = tid * tid;
sum += square;
std::cout << "Thread ID " << tid << " square: " << square << std::endl;
}

std::cout << "Sum of squares: " << sum << std::endl;
return 0;
}
```

Output:

```
ubuntu@ubuntu-VirtualBox:~/Documents/Assignment05$ g++ -fopenmp -o a 05_02_a.cpp
ubuntu@ubuntu-VirtualBox:~/Documents/Assignment05$ ./a
Thread ID 1 square: 1
Thread ID 0 square: 0
Sum of squares: 1
ubuntu@ubuntu-VirtualBox:~/Documents/Assignment05$
```

Q3. Consider a variable called “Aryabhatta” declared as 10 (i.e int Arbhatta=10).Write an OpenMP program which should print the result of multiplication of thread id and value of the above variable.

Note*: The variable “Aryabhatta” should be declared as private

code:

```
#include <iostream>

#include <omp.h>

int main() {
int Aryabhatta = 10;

#pragma omp parallel private(Aryabhatta)
{
int tid = omp_get_thread_num();
int result = tid * Aryabhatta;
std::cout << "Thread ID " << tid << " result: " << result << std::endl;
}

return 0;
}
```

Output:

```
ubuntu@ubuntu-VirtualBox:~/Documents/Assignment05$ g++ -fopenmp -o a 05_03_a.cpp
ubuntu@ubuntu-VirtualBox:~/Documents/Assignment05$ ./a
Thread ID 1 result: 1919448176
Thread ID 0 result: 0
```

Q4. Write an OpenMP program that calculates the partial sum of the first 20 natural numbers using parallelism. Each thread should compute a portion of the sum by iterating through a loop. Implement the program using the lastprivate clause to ensure that the final total sum is correctly computed and printed outside the parallel region.

Hint:

1.Utilize OpenMP directives to parallelize the summation process.

2.Ensure that each thread has its private copy of partial sum.

3.Use the lastprivate clause to assign the value of the last thread's partial sum to the final total sum after the parallel region.

Code:

```
#include <iostream>

#include <omp.h>

int main() {
int total_sum = 0;
const int N = 20;
int partial_sum = 0;

#pragma omp parallel private(partial_sum)
{
#pragma omp for
for (int i = 1; i <= N; ++i) {
partial_sum += i;
}

#pragma omp critical
total_sum += partial_sum;
}

std::cout << "Total sum: " << total_sum << std::endl;
return 0;
}
```

Output:

```
ubuntu@ubuntu-VirtualBox:~/Documents/Assignment05$ g++ -fopenmp -o a 05_04_a.cpp
ubuntu@ubuntu-VirtualBox:~/Documents/Assignment05$ ./a
Total sum: 29387
```

Q5. Consider a scenario where you have to parallelize a program that performs matrix multiplication using OpenMP. Your task is to implement parallelization using both static and dynamic scheduling, and compare the execution time of each approach.

Note*:

Practical No 5

- Implement a serial version of matrix multiplication in C/C++.
- Parallelize the matrix multiplication using OpenMP with static scheduling.
- Parallelize the matrix multiplication using OpenMP with dynamic scheduling.
- Measure the execution time of each parallelized version for various matrix sizes.
- Compare the execution times and discuss the advantages and disadvantages of static and dynamic scheduling in this context.

Code :

```
#include <iostream>

#include <vector>
#include <omp.h>

void matrix_multiply_static(const std::vector<std::vector<int>>& A,
const std::vector<std::vector<int>>& B,
std::vector<std::vector<int>>& C) {
int n = A.size();
#pragma omp parallel for schedule(static)
for (int i = 0; i < n; ++i) {
for (int j = 0; j < n; ++j) {
C[i][j] = 0;
for (int k = 0; k < n; ++k) {
C[i][j] += A[i][k] * B[k][j];
}
}
}
}

void matrix_multiply_dynamic(const std::vector<std::vector<int>>& A,
const std::vector<std::vector<int>>& B,
std::vector<std::vector<int>>& C) {
int n = A.size();
#pragma omp parallel for schedule(dynamic)
for (int i = 0; i < n; ++i) {
for (int j = 0; j < n; ++j) {
C[i][j] = 0;
for (int k = 0; k < n; ++k) {
C[i][j] += A[i][k] * B[k][j];
}
}
}
```

```
}

int main() {
int n = 100; // Adjust size as needed
std::vector<std::vector<int>> A(n, std::vector<int>(n, 1));
std::vector<std::vector<int>> B(n, std::vector<int>(n, 1));
std::vector<std::vector<int>> C(n, std::vector<int>(n));

double start_time = omp_get_wtime();
matrix_multiply_static(A, B, C);
double end_time = omp_get_wtime();
std::cout << "Static scheduling time: " << (end_time - start_time) << " seconds" << std::endl;

start_time = omp_get_wtime();
matrix_multiply_dynamic(A, B, C);
end_time = omp_get_wtime();
std::cout << "Dynamic scheduling time: " << (end_time - start_time) << " seconds" << std::endl;

return 0;
}
```

Output:

```
ubuntu@ubuntu-VirtualBox:~/Documents/Assignment05$ g++ -fopenmp -o a 05_05_a.cpp
ubuntu@ubuntu-VirtualBox:~/Documents/Assignment05$ ./a
Static scheduling time: 0.0203766 seconds
Dynamic scheduling time: 0.00705895 seconds
ubuntu@ubuntu-VirtualBox:~/Documents/Assignment05$
```

Q6. Write a Parallel C program which should print the series of 2 and 4. Make sure both should be executed by different threads !

Code:

```
#include <iostream>

#include <omp.h>

int main() {
#pragma omp parallel num_threads(2)
{
```

```
int tid = omp_get_thread_num();
if (tid == 0) {
std::cout << "Thread ID " << tid << ": 2" << std::endl;
} else if (tid == 1) {
std::cout << "Thread ID " << tid << ": 4" << std::endl;
}
}
return 0;
}
```

Output:

```
ubuntu@ubuntu-VirtualBox:~/Documents/Assignment05$ g++ -fopenmp -o a 05_06_a.cpp
ubuntu@ubuntu-VirtualBox:~/Documents/Assignment05$ ./a
Thread ID 1: 4
Thread ID 0: 2
ubuntu@ubuntu-VirtualBox:~/Documents/Assignment05$
```

Q7. Consider a scenario where you have a shared variable total_sum that needs to be updated concurrently by multiple threads in a parallel program. However, concurrent updates to this variable can result in data races and incorrect results. Your task is to modify the program to ensure correct synchronization using OpenMP's critical and atomic constructs.

Note*:

- Implement a simple parallel program in C that initializes an array of integers and calculates the sum of its elements concurrently using OpenMP.
- Identify potential issues with concurrent updates to the total_sum variable in the parallelized version of the program.
- Modify the program to use OpenMP's critical/atomic directive to ensure synchronized access to the total_sum variable.
- Measure and compare the performance of synchronized versions against the unsynchronized implementation.

Code:

```
#include <iostream>

#include <omp.h>

int main() {
const int N = 1000;
```

```
int array[N];
int total_sum = 0;

for (int i = 0; i < N; ++i) {
array[i] = 1;
}

#pragma omp parallel
{
int local_sum = 0;
#pragma omp for
for (int i = 0; i < N; ++i) {
local_sum += array[i];
}

#pragma omp critical
total_sum += local_sum;
}

std::cout << "Total sum: " << total_sum << std::endl;
return 0;
}
```

Output:

```
ubuntu@ubuntu-VirtualBox:~/Documents/Assignment05$ g++ -fopenmp -o a 05_07_a.cpp
ubuntu@ubuntu-VirtualBox:~/Documents/Assignment05$ ./a
Total sum: 1000
ubuntu@ubuntu-VirtualBox:~/Documents/Assignment05$
```

Q8. Consider a scenario where you have a large array of integers, and you need to find the sum of all its elements in parallel using OpenMP. The array is shared among multiple threads, and parallelism is needed to expedite the computation process. Your task is to write a parallel program that calculates the sum of all elements in the array using OpenMP's reduction clause.

Note*:

- Implement a sequential version of the program that calculates the sum of all elements in the array without using any parallelism.
- Identify potential bottlenecks and limitations of the sequential implementation in handling large arrays efficiently.

Practical No 5

- Modify the program to utilize OpenMP's reduction clause to parallelize the summation process across multiple threads.
- Test the program with different array sizes and thread counts to evaluate its scalability and performance.
- Discuss the advantages of using the reduction clause for parallel summation and its impact on program efficiency.

Code:

```
#include <iostream>

#include <omp.h>

int main() {
const int N = 1000;
int array[N];
int total_sum = 0;

for (int i = 0; i < N; ++i) {
array[i] = 1;
}

#pragma omp parallel
{
int local_sum = 0;
#pragma omp for
for (int i = 0; i < N; ++i) {
local_sum += array[i];
}

#pragma omp critical
total_sum += local_sum;
}

std::cout << "Total sum: " << total_sum << std::endl;
return 0;
}
```

Output:

```
ubuntu@ubuntu-VirtualBox:~/Documents/Assignment05$ g++ -fopenmp -o a 05_08_a.cpp
ubuntu@ubuntu-VirtualBox:~/Documents/Assignment05$ ./a
Sequential sum: 1000000
ubuntu@ubuntu-VirtualBox:~/Documents/Assignment05$
```