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 <omp.h>

#include <stdio.h>

int main()

{

char \*family[] = {"Shreyas", "Pratima", "Mummy", "Pappa", "Didi"};

int num\_of\_threads = 5;

#pragma omp parallel num\_threads(num\_of\_threads)

{

int thread\_id = omp\_get\_thread\_num();

if (thread\_id < num\_of\_threads)

{

printf("Thread ID: %d - Family Member: %s\n", thread\_id, family[thread\_id]);

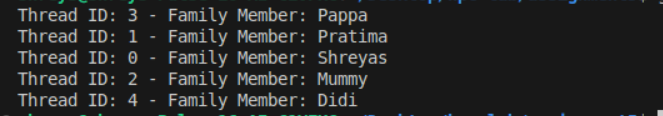
}

}

return 0;

}

OUTPUT :



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.

github token : ghp\_tEdw05j7tBETdgBqo3mTC0cf8t1pGP3NmRm5

CODE :

#include<omp.h>

#include<stdio.h>

int main(){

int sum\_of\_id =0;

int num\_thread=4;

#pragma omp parallel num\_threads(num\_thread)

{

int thread\_id = omp\_get\_thread\_num();

int square = thread\_id\*thread\_id;

sum\_of\_id += (square);

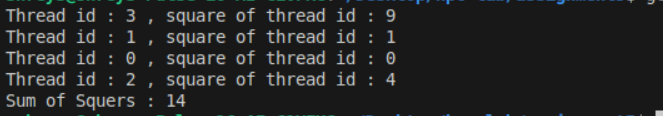
printf("Thread id : %d , square of thread id : %d\n" , thread\_id , square);

}

printf("Sum of Squers : %d\n " , sum\_of\_id);

}

OUTPUT :



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<omp.h>

#include<stdio.h>

int main(){

int Arybhatta =10;

int num\_thread = 5;

#pragma omp parallel num\_threads(num\_thread) private(Arybhatta)

{

Arybhatta=10;

int thread\_id = omp\_get\_thread\_num();

int value = Arybhatta \* thread\_id;

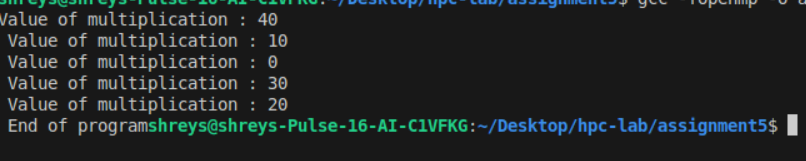
printf("Value of multiplication : %d\n " ,value);

}

printf("End of program");

}

OUTPUT :



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

Code :

#include <stdio.h>

#include <omp.h>

int main() {

int sum = 0; // Final sum to be computed

int partial\_sum = 0; // Each thread's partial sum

int num\_thread = 10;

#pragma omp parallel num\_threads(num\_thread)

{ partial\_sum =0;

int thread\_id = omp\_get\_thread\_num();

#pragma omp for lastprivate(partial\_sum)

for (int i = 1; i <= 20; i++) {

partial\_sum += i;

}

sum = partial\_sum;

printf("The value of Sum : %d for thread id : %d\n" , partial\_sum,thread\_id);

}

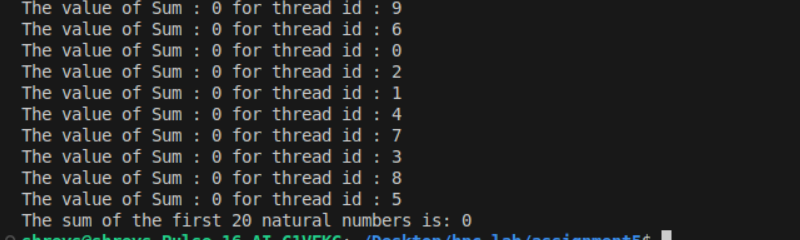
// The last thread's partial sum is assigned to the final sum

printf("The sum of the first 20 natural numbers is: %d\n", sum);

return 0;

}

OUTPUT:



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.

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\*:**

* 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 <omp.h>

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#define N 1000 // Matrix size

void matrix\_multiply\_serial(int \*\*matA, int \*\*matB, int \*\*result) {

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

for (int j = 0; j < N; j++) {

result[i][j] = 0;

for (int k = 0; k < N; k++)

result[i][j] += matA[i][k] \* matB[k][j];

}

}

void matrix\_multiply\_static(int \*\*matA, int \*\*matB, int \*\*result) {

#pragma omp parallel for schedule(static)

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

for (int j = 0; j < N; j++) {

result[i][j] = 0;

for (int k = 0; k < N; k++)

result[i][j] += matA[i][k] \* matB[k][j];

}

}

void matrix\_multiply\_dynamic(int \*\*matA, int \*\*matB, int \*\*result) {

#pragma omp parallel for schedule(dynamic)

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

for (int j = 0; j < N; j++) {

result[i][j] = 0;

for (int k = 0; k < N; k++)

result[i][j] += matA[i][k] \* matB[k][j];

}

}

int main() {

// Dynamically allocate matrices

int \*\*matA = malloc(N \* sizeof(int \*));

int \*\*matB = malloc(N \* sizeof(int \*));

int \*\*result = malloc(N \* sizeof(int \*));

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

matA[i] = malloc(N \* sizeof(int));

matB[i] = malloc(N \* sizeof(int));

result[i] = malloc(N \* sizeof(int));

}

// Initialize matrices with random values

srand(time(0));

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

for (int j = 0; j < N; j++) {

matA[i][j] = rand() % 10;

matB[i][j] = rand() % 10;

}

// Measure time for serial

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

matrix\_multiply\_serial(matA, matB, result);

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

printf("Serial time: %f seconds\n", serial\_time);

// Measure time for static scheduling

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

matrix\_multiply\_static(matA, matB, result);

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

printf("Static scheduling time: %f seconds\n", static\_time);

// Measure time for dynamic scheduling

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

matrix\_multiply\_dynamic(matA, matB, result);

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

printf("Dynamic scheduling time: %f seconds\n", dynamic\_time);

// Free allocated memory

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

free(matA[i]);

free(matB[i]);

free(result[i]);

}

free(matA);

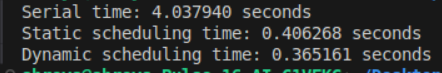
free(matB);

free(result);

return 0;

}

OUTPUT :



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 <omp.h>

#include <stdio.h>

int main() {

#pragma omp parallel num\_threads(2)

{

int thread\_id = omp\_get\_thread\_num();

if (thread\_id == 0) {

printf("Thread %d: Series of 2\n", thread\_id);

for (int i = 1; i <= 5; i++)

printf("%d ", 2 \* i);

printf("\n");

} else if (thread\_id == 1) {

printf("Thread %d: Series of 4\n", thread\_id);

for (int i = 1; i <= 5; i++)

printf("%d ", 4 \* i);

printf("\n");

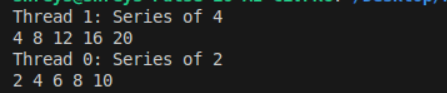
}

}

return 0;

}

OUTPUT :



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 <omp.h>

#include <stdio.h>

#define N 1000000

int arr[N];

void sumWithoutSync() {

int total\_sum = 0;

#pragma omp parallel for

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

total\_sum += arr[i]; // This will lead to a data race

}

printf("Total Sum without synchronization: %d\n", total\_sum);

}

void sumWithCritical() {

int total\_sum = 0;

#pragma omp parallel for

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

#pragma omp critical

{

total\_sum += arr[i]; // Critical section ensures only one thread can modify total\_sum at a time

}

}

printf("Total Sum with critical section: %d\n", total\_sum);

}

void sumWithAtomic() {

int total\_sum = 0;

#pragma omp parallel for

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

#pragma omp atomic

total\_sum += arr[i]; // Atomic ensures the update is done safely without using full locks

}

printf("Total Sum with atomic: %d\n", total\_sum);

}

int main() {

for (int i = 0; i < 100; i++) arr[i] = i + 1;

double stime = omp\_get\_wtime();

sumWithoutSync();

double etime = omp\_get\_wtime();

printf("Time for unsynchronized version: %lf seconds\n", (etime - stime) );

stime = omp\_get\_wtime();

sumWithCritical();

etime = omp\_get\_wtime();

printf("Time for critical version: %lf seconds\n", (etime - stime) );

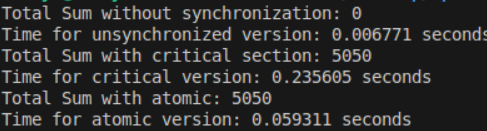
stime = omp\_get\_wtime();

sumWithAtomic();

etime = omp\_get\_wtime();

printf("Time for atomic version: %lf seconds\n", (etime - stime) );

}

OUTPUT :

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.
* 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 <omp.h>

#include <stdio.h>

#include <time.h>

int main() {

int array[1000]; // Large array

for (int i = 0; i < 1000; i++) array[i] = i + 1;

// Sequential sum

int sequential\_sum = 0;

double start\_time = omp\_get\_wtime(); // Start time for sequential sum

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

sequential\_sum += array[i];

double sequential\_time = omp\_get\_wtime() - start\_time; // Elapsed time for sequential sum

printf("Sequential sum: %d, Time taken: %f seconds\n", sequential\_sum, sequential\_time);

// Parallel sum using reduction

int parallel\_sum = 0;

start\_time = omp\_get\_wtime(); // Start time for parallel sum

#pragma omp parallel for reduction(+:parallel\_sum)

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

parallel\_sum += array[i];

double parallel\_time = omp\_get\_wtime() - start\_time; // Elapsed time for parallel sum

printf("Parallel sum using reduction: %d, Time taken: %f seconds\n", parallel\_sum, parallel\_time);

return 0;

}

OUTPUT :

