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.

#include <omp.h>

#include <stdio.h>

int main() {

    char\* family\_members[] = {

        "Abhijeet",

        "Rohan",

        "Shashank",

        "Rohit"

    };

    int num\_members = sizeof(family\_members) / sizeof(family\_members[0]);

    omp\_set\_num\_threads(num\_members);

    #pragma omp parallel

    {

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

        printf("Thread %d: Family member: %s\n", thread\_id, family\_members[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.

#include <omp.h>

#include <stdio.h>

int main() {

    int num\_threads;

    int sum\_of\_squares = 0;

    omp\_set\_num\_threads(4);

    #pragma omp parallel

    {

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

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

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

        #pragma omp atomic

        sum\_of\_squares += square;

        #pragma omp single

        num\_threads = omp\_get\_num\_threads();

    }

    printf("Sum of squares of thread IDs = %d\n", sum\_of\_squares);

    return 0;

}

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

#include <omp.h>

#include <stdio.h>

int main() {

    int Aryabhatta = 10;

    #pragma omp parallel private(Aryabhatta)

    {

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

        Aryabhatta = 10;

        int result = thread\_id \* Aryabhatta;

        printf("Thread %d: Result of %d \* %d = %d\n", thread\_id, thread\_id, Aryabhatta, result);

    }

    return 0;

}

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 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.
* #include <omp.h>
* #include <stdio.h>
* #include <stdlib.h>
* #include <time.h>
* #define MATRIX\_SIZE 100
* void matrix\_multiply\_serial(int n, int \*\*A, int \*\*B, int \*\*C) {
* 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\_static(int n, int \*\*A, int \*\*B, int \*\*C) {
* #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(int n, int \*\*A, int \*\*B, int \*\*C) {
* #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];
* }
* }
* }
* }
* void initialize\_matrices(int n, int \*\*A, int \*\*B) {
* srand(time(NULL));
* for (int i = 0; i < n; i++) {
* for (int j = 0; j < n; j++) {
* A[i][j] = rand() % 10;
* B[i][j] = rand() % 10;
* }
* }
* }
* void print\_execution\_time(const char \*label, double start, double end) {
* printf("%s execution time: %f seconds\n", label, end - start);
* }
* int main() {
* int n = MATRIX\_SIZE;
* int \*\*A = (int \*\*)malloc(n \* sizeof(int \*));
* int \*\*B = (int \*\*)malloc(n \* sizeof(int \*));
* int \*\*C = (int \*\*)malloc(n \* sizeof(int \*));
* for (int i = 0; i < n; i++) {
* A[i] = (int \*)malloc(n \* sizeof(int));
* B[i] = (int \*)malloc(n \* sizeof(int));
* C[i] = (int \*)malloc(n \* sizeof(int));
* }
* initialize\_matrices(n, A, B);
* // Serial Matrix Multiplication
* double start = omp\_get\_wtime();
* matrix\_multiply\_serial(n, A, B, C);
* double end = omp\_get\_wtime();
* print\_execution\_time("Serial", start, end);
* // Parallel Matrix Multiplication with Static Scheduling
* start = omp\_get\_wtime();
* matrix\_multiply\_static(n, A, B, C);
* end = omp\_get\_wtime();
* print\_execution\_time("Static scheduling", start, end);
* // Parallel Matrix Multiplication with Dynamic Scheduling
* start = omp\_get\_wtime();
* matrix\_multiply\_dynamic(n, A, B, C);
* end = omp\_get\_wtime();
* print\_execution\_time("Dynamic scheduling", start, end);
* // Clean up memory
* for (int i = 0; i < n; i++) {
* free(A[i]);
* free(B[i]);
* free(C[i]);
* }
* free(A);
* free(B);
* free(C);
* return 0;
* }



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

#include <omp.h>

#include <stdio.h>

int main() {

    int num\_terms = 10;

    #pragma omp parallel

    {

        #pragma omp single

        {

            #pragma omp task

            {

                for (int i = 0; i < num\_terms; i++) {

                    printf("2 ");

                }

                printf("\n");

            }

            #pragma omp task

            {

                for (int i = 0; i < num\_terms; i++) {

                    printf("4 ");

                }

                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.

#include <omp.h>

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#define ARRAY\_SIZE 1000000

void initialize\_array(int \*array, int size) {

    for (int i = 0; i < size; i++) {

        array[i] = rand() % 10;

    }

}

void sum\_array\_unsynchronized(int \*array, int size) {

    int total\_sum = 0;

    #pragma omp parallel for

    for (int i = 0; i < size; i++) {

        total\_sum += array[i];

    }

    printf("Unsynchronized total sum: %d\n", total\_sum);

}

void sum\_array\_critical(int \*array, int size) {

    int total\_sum = 0;

    #pragma omp parallel

    {

        int local\_sum = 0;

        #pragma omp for

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

            local\_sum += array[i];

        }

        #pragma omp critical

        {

            total\_sum += local\_sum;

        }

    }

    printf("Critical section total sum: %d\n", total\_sum);

}

void sum\_array\_atomic(int \*array, int size) {

    int total\_sum = 0;

    #pragma omp parallel

    {

        #pragma omp for

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

            #pragma omp atomic

            total\_sum += array[i];

        }

    }

    printf("Atomic operation total sum: %d\n", total\_sum);

}

int main() {

    int \*array = (int \*)malloc(ARRAY\_SIZE \* sizeof(int));

    srand(time(NULL));

    initialize\_array(array, ARRAY\_SIZE);

    double start = omp\_get\_wtime();

    sum\_array\_unsynchronized(array, ARRAY\_SIZE);

    double end = omp\_get\_wtime();

    printf("Unsynchronized execution time: %f seconds\n", end - start);

    start = omp\_get\_wtime();

    sum\_array\_critical(array, ARRAY\_SIZE);

    end = omp\_get\_wtime();

    printf("Critical section execution time: %f seconds\n", end - start);

    start = omp\_get\_wtime();

    sum\_array\_atomic(array, ARRAY\_SIZE);

    end = omp\_get\_wtime();

    printf("Atomic operation execution time: %f seconds\n", end - start);

    free(array);

    return 0;

}

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.

#include <omp.h>

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#define MAX\_SIZE 100000000 // Define the maximum size of the array

// Function to initialize the array with random values

void initialize\_array(int \*array, int size) {

    for (int i = 0; i < size; i++) {

        array[i] = rand() % 100;

    }

}

// Function to calculate the sum sequentially

long long sum\_array\_sequential(int \*array, int size) {

    long long total\_sum = 0;

    for (int i = 0; i < size; i++) {

        total\_sum += array[i];

    }

    return total\_sum;

}

// Function to calculate the sum in parallel using OpenMP reduction clause

long long sum\_array\_parallel(int \*array, int size) {

    long long total\_sum = 0;

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

    {

        #pragma omp for

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

            total\_sum += array[i];

        }

    }

    return total\_sum;

}

int main() {

    int \*array;

    int sizes[] = {100000, 1000000, 10000000, 50000000};

    int num\_sizes = sizeof(sizes) / sizeof(sizes[0]);

    int num\_threads[] = {1, 2, 4, 8};

    int num\_thread\_counts = sizeof(num\_threads) / sizeof(num\_threads[0]);

    array = (int \*)malloc(MAX\_SIZE \* sizeof(int));

    for (int s = 0; s < num\_sizes; s++) {

        int size = sizes[s];

        initialize\_array(array, size);

        double start = omp\_get\_wtime();

        long long sequential\_sum = sum\_array\_sequential(array, size);

        double end = omp\_get\_wtime();

        printf("Sequential sum for size %d: %lld, Time: %f seconds\n", size, sequential\_sum, end - start);

        for (int t = 0; t < num\_thread\_counts; t++) {

            int threads = num\_threads[t];

            omp\_set\_num\_threads(threads);

            start = omp\_get\_wtime();

            long long parallel\_sum = sum\_array\_parallel(array, size);

            end = omp\_get\_wtime();

            printf("Parallel sum for size %d with %d threads: %lld, Time: %f seconds\n", size, threads, parallel\_sum, end - start);

        }

    }

    free(array);

    return 0;

}

Output:

