**MULTITHREADING**

**THREAD CREATION AND JOINING**

**CODE**

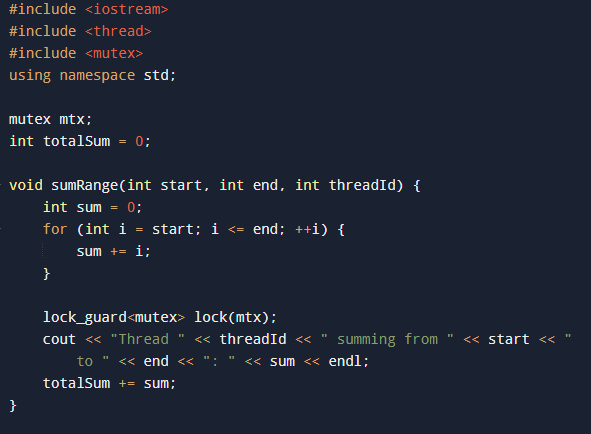


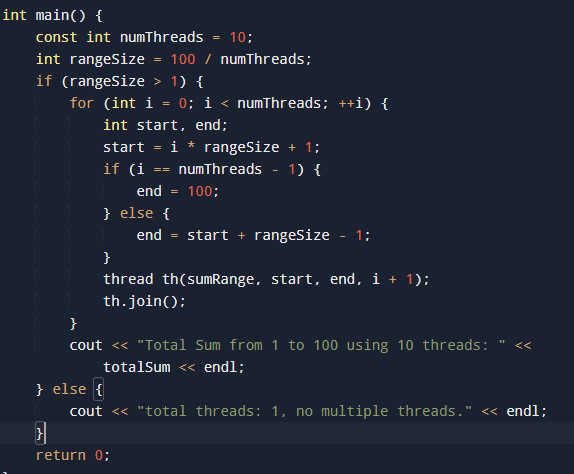
**OUTPUT**

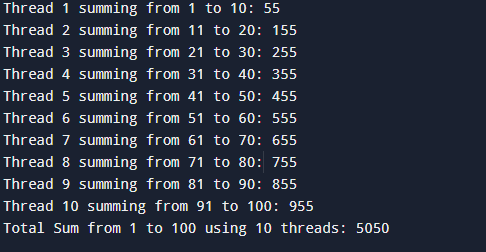


**SUM OF NUMBERS FROM 1 TO 100 USING MULTITHREADING**

**CODE**





**OUTPUT**

**OPENMP**

**PTHREADS VS OPENMP PERFORMANCE**

**CODE**

#include <pthread.h>

#include <sched.h>

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <omp.h>

#include <time.h>

#define NUM\_THREADS 4

#define N 100000000 *// Size of the array*

*// Global array*

int array[N];

*// Mutex for thread synchronization*

pthread\_mutex\_t mutex\_sum = PTHREAD\_MUTEX\_INITIALIZER;

long long sum\_pthread = 0;

*// Function to set CPU affinity to core 0 (same core for all threads)*

void set\_cpu\_affinity\_same\_core() {

    cpu\_set\_t cpuset;

    CPU\_ZERO(&cpuset); *// Initialize cpuset to be empty*

    CPU\_SET(0, &cpuset); *// Assign thread to core 0*

    pthread\_t current\_thread = pthread\_self();

    if (pthread\_setaffinity\_np(current\_thread, sizeof(cpu\_set\_t), &cpuset) != 0) {

        perror("pthread\_setaffinity\_np");

        exit(EXIT\_FAILURE);

    }

    printf("Thread %lu is now running on core 0\n", current\_thread);

}

*// Function to set CPU affinity to different cores (0, 1, 2, ...)*

void set\_cpu\_affinity(int *core\_id*) {

    cpu\_set\_t cpuset;

    CPU\_ZERO(&cpuset);

    CPU\_SET(*core\_id*, &cpuset);

    pthread\_t current\_thread = pthread\_self();

    if (pthread\_setaffinity\_np(current\_thread, sizeof(cpu\_set\_t), &cpuset) != 0) {

        perror("pthread\_setaffinity\_np");

        exit(EXIT\_FAILURE);

    }

    printf("Thread %lu is now running on core %d\n", current\_thread, *core\_id*);

}

*// Function to simulate work (sum array values)*

void\* sum\_pthread\_func(void\* *arg*) {

    int thread\_id = \*((int\*)*arg*);

*// Bind threads to cores (for same-core or different cores)*

    if (thread\_id < NUM\_THREADS) {

        set\_cpu\_affinity(thread\_id); *// Bind thread to different cores (option 2)*

    } else {

        set\_cpu\_affinity\_same\_core(); *// Bind all threads to core 0 (option 1)*

    }

    long long local\_sum = 0;

    for (int i = thread\_id \* (N / NUM\_THREADS); i < (thread\_id + 1) \* (N / NUM\_THREADS); i++) {

        local\_sum += array[i];

    }

*// Locking shared sum*

    pthread\_mutex\_lock(&mutex\_sum);

    sum\_pthread += local\_sum;

    pthread\_mutex\_unlock(&mutex\_sum);

    return NULL;

}

*// Function to initialize the array with random values*

void init\_array() {

    for (int i = 0; i < N; i++) {

        array[i] = rand() % 1000;

    }

}

int main() {

    init\_array();

*// Time comparison*

    struct timespec start, end;

*// Option 1: Pthreads on Same Core*

    printf("\n=== Pthreads on Same Core ===\n");

    clock\_gettime(CLOCK\_REALTIME, &start);

    pthread\_t threads\_same\_core[NUM\_THREADS];

    int thread\_ids\_same\_core[NUM\_THREADS] = {0, 0, 0, 0};

*// Create threads and bind all to core 0*

    for (int i = 0; i < NUM\_THREADS; i++) {

        if (pthread\_create(&threads\_same\_core[i], NULL, sum\_pthread\_func, (void\*)&thread\_ids\_same\_core[i]) != 0) {

            perror("pthread\_create");

            exit(EXIT\_FAILURE);

        }

    }

    for (int i = 0; i < NUM\_THREADS; i++) {

        pthread\_join(threads\_same\_core[i], NULL);

    }

    clock\_gettime(CLOCK\_REALTIME, &end);

    printf("Pthread sum (same core): %lld\n", sum\_pthread);

    printf("Pthread (same core) execution time: %lf seconds\n",

           (end.tv\_sec - start.tv\_sec) + (end.tv\_nsec - start.tv\_nsec) / 1e9);

*// Reset the sum for the next experiment*

    sum\_pthread = 0;

*// Option 2: Pthreads on Different Cores*

    printf("\n=== Pthreads on Different Cores ===\n");

    clock\_gettime(CLOCK\_REALTIME, &start);

    pthread\_t threads\_different\_cores[NUM\_THREADS];

    int thread\_ids\_different\_cores[NUM\_THREADS] = {0, 1, 2, 3};

*// Create threads and assign them to different cores*

    for (int i = 0; i < NUM\_THREADS; i++) {

        if (pthread\_create(&threads\_different\_cores[i], NULL, sum\_pthread\_func, (void\*)&thread\_ids\_different\_cores[i]) != 0) {

            perror("pthread\_create");

            exit(EXIT\_FAILURE);

        }

    }

    for (int i = 0; i < NUM\_THREADS; i++) {

        pthread\_join(threads\_different\_cores[i], NULL);

    }

    clock\_gettime(CLOCK\_REALTIME, &end);

    printf("Pthread sum (different cores): %lld\n", sum\_pthread);

    printf("Pthread (different cores) execution time: %lf seconds\n",

           (end.tv\_sec - start.tv\_sec) + (end.tv\_nsec - start.tv\_nsec) / 1e9);

*// Reset the sum for the next experiment*

    sum\_pthread = 0;

*// Option 3: OpenMP Parallel Execution*

    printf("\n=== OpenMP Parallel Execution ===\n");

    clock\_gettime(CLOCK\_REALTIME, &start);

    long long sum\_openmp = 0;

*// Parallelize the summing of the array using OpenMP*

    #pragma omp parallel for reduction(+:sum\_openmp)

    for (int i = 0; i < N; i++) {

        sum\_openmp += array[i];

    }

    clock\_gettime(CLOCK\_REALTIME, &end);

    printf("OpenMP sum: %lld\n", sum\_openmp);

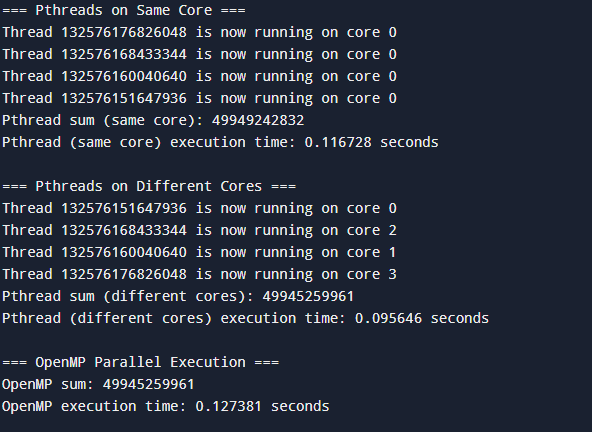
    printf("OpenMP execution time: %lf seconds\n",

           (end.tv\_sec - start.tv\_sec) + (end.tv\_nsec - start.tv\_nsec) / 1e9);

    return 0;

}

**OUTPUT**



**SUMMATION USING OPENMP**

**CODE:**

#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();

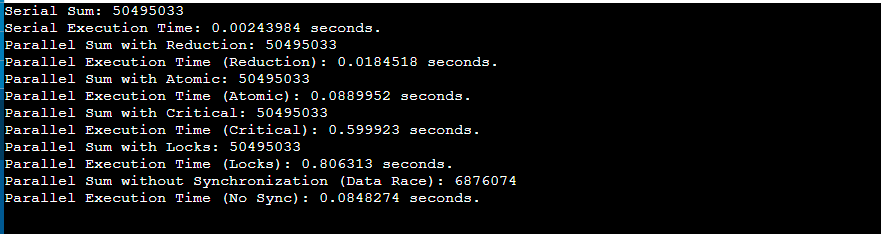
    std::cout << "Parallel Sum without Synchronization (Data Race): " << sum << std::endl;

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

    return 0;

}

**OUTPUT:**



**COMPARISON TABLE**

| **Synchronization Method** | **Sum Correct?** | **Execution Time (s)** | **Remarks** |
| --- | --- | --- | --- |
| Serial Execution | Yes | 0.00243984 | Baseline for comparison |
| Parallel Sum with Reduction | Yes | 0.0184518 | Fast, minimal overhead |
| Parallel Sum with Atomic | Yes | 0.0889952 | Moderate overhead |
| Parallel Sum with Critical | Yes | 0.599523 | High overhead |
| Parallel Sum with Locks | Yes | 0.806313 | Highest overhead |
| No Synchronization (Data Race) | **No** | 0.0848274 | Fast but incorrect result (Data Race) |

**ADVANTAGES AND DRAWBACKS FOR EACH SYNCHRONIZATION METHOD:**

**REDUCTION:**

Advantage: Fast and efficient for aggregations.

Drawback: Limited to specific operations (e.g., sum, max).

**ATOMIC:**

Advantage: Avoids full locks with minimal overhead.

Drawback: Only suitable for simple, single-variable updates.

**CRITICAL:**

Advantage: Easy to use for protecting critical sections.

Drawback: Slows execution as only one thread can enter at a time.

**LOCKS:**

Advantage: Flexible and can protect multiple variables or complex logic.

Drawback: High overhead and potential for deadlocks.

**NO SYNCHRONIZATION:**

Advantage: Fastest execution without any coordination overhead.

Drawback: Unsafe due to race conditions, leading to incorrect results.

**SUM OF N NUMBERS USING OPEN MP**

**CODE**

#include <iostream>

#include <omp.h>

using namespace std;

int main() {

    int N = 100;

    int sum = 0;

*// Parallel loop with atomic to prevent race conditions*

    #pragma omp parallel for shared(sum)

    for (int i = 1; i <= N; i++) {

        #pragma omp atomic

        sum += i;

    }

    cout << "Final sum = " << sum << endl;

    return 0;

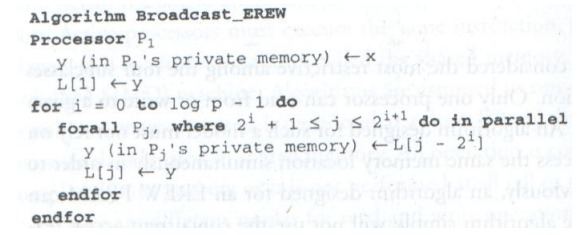
}

**OUTPUT**

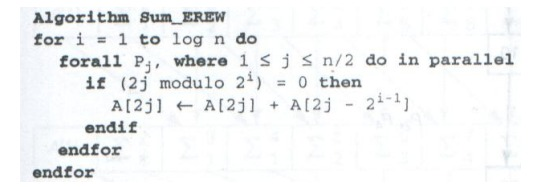


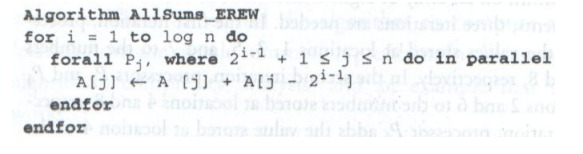
**PRAM ALGORITHMS**

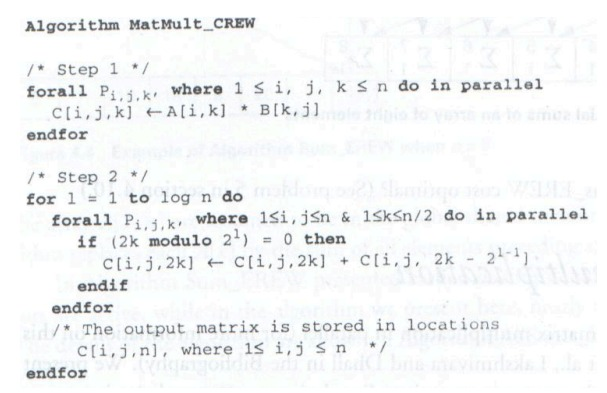
**MULTIPLE ACCESSES ON EREW**

****

**COMPUTING SUM OF AN ARRAY ON EREW PRAM**

**COMPUTING ALL PARTIAL SUM**

**MATRIX MULTIPLICATION (CREW PRAM)**

****