MULTITHREADING

THREAD CREATION AND JOINING

CODE

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

void* mythread(void* args) {
    printf("%s\n", (char*) args);
    return NULL;
}
int main(int argc, char *argv[])
{
    pthread_t p1, p2;
    printf("main begin\n");

    pthread_create(&p1, NULL, mythread, "A");
    pthread_create(&p2, NULL, mythread, "B");

    pthread_join(p1, NULL);
    pthread_join(p2, NULL);

    printf("main end\n");
    return 0;
}
```

OUTPUT

```
main begin
A
B
main end
```

SUM OF NUMBERS FROM 1 TO 100 USING MULTITHREADING

CODE

```
#include <iostream>
#include <thread>
#include <mutex>
using namespace std;

mutex mtx;
int totalSum = 0;

void sumRange(int start, int end, int threadId) {
    int sum = 0;
    for (int i = start; i <= end; ++i) {
        sum += i;
    }

    lock_guard<mutex> lock(mtx);
    cout << "Thread " << threadId << " summing from " << start << "
        to " << end << ": " << sum << endl;
    totalSum += sum;
}
```

```
int main() {
    const int numThreads = 10;
    int rangeSize = 100 / numThreads;
    if (rangeSize > 1) {
        for (int i = 0; i < numThreads; ++i) {
            int start, end;
            start = i * rangeSize + 1;
            if (i == numThreads - 1) {
                end = 100;
            } else {
                end = start + rangeSize - 1;
            }
            thread th(sumRange, start, end, i + 1);
            th.join();
        }
        cout << "Total Sum from 1 to 100 using 10 threads: " <<
            totalSum << endl;
    } else {
        cout << "total threads: 1, no multiple threads." << endl;
    }
    return 0;
```

OUTPUT

```
Thread 1 summing from 1 to 10: 55
Thread 2 summing from 11 to 20: 155
Thread 3 summing from 21 to 30: 255
Thread 4 summing from 31 to 40: 355
Thread 5 summing from 41 to 50: 455
Thread 6 summing from 51 to 60: 555
Thread 7 summing from 61 to 70: 655
Thread 8 summing from 71 to 80: 755
Thread 9 summing from 81 to 90: 855
Thread 10 summing from 91 to 100: 955
Total Sum from 1 to 100 using 10 threads: 5050
```

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);
}
```

```c
// 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;
```

```c
    // 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

```
=== Pthreads on Same Core ===
Thread 132576176826048 is now running on core 0
Thread 132576168433344 is now running on core 0
Thread 132576160040640 is now running on core 0
Thread 132576151647936 is now running on core 0
Pthread sum (same core): 49949242832
Pthread (same core) execution time: 0.116728 seconds

=== Pthreads on Different Cores ===
Thread 132576151647936 is now running on core 0
Thread 132576168433344 is now running on core 2
Thread 132576160040640 is now running on core 1
Thread 132576176826048 is now running on core 3
Pthread sum (different cores): 49945259961
Pthread (different cores) execution time: 0.095646 seconds

=== OpenMP Parallel Execution ===
OpenMP sum: 49945259961
OpenMP execution time: 0.127381 seconds
```

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();
```

```cpp
    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:

```
Serial Sum: 50495033
Serial Execution Time: 0.00243984 seconds.
Parallel Sum with Reduction: 50495033
Parallel Execution Time (Reduction): 0.0184518 seconds.
Parallel Sum with Atomic: 50495033
Parallel Execution Time (Atomic): 0.0889952 seconds.
Parallel Sum with Critical: 50495033
Parallel Execution Time (Critical): 0.599923 seconds.
Parallel Sum with Locks: 50495033
Parallel Execution Time (Locks): 0.806313 seconds.
Parallel Sum without Synchronization (Data Race): 6876074
Parallel Execution Time (No Sync): 0.0848274 seconds.
```

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

```cpp
#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

```
Final sum = 5050
```

PRAM ALGORITHMS

MULTIPLE ACCESSES ON EREW

```
Algorithm Broadcast_EREW
Processor P1
  y (in P1's private memory) ← x
  L[1] ← y
for i = 0 to log p - 1 do
  forall Pj, where 2^i + 1 ≤ j ≤ 2^(i+1) do in parallel
    y (in Pj's private memory) ← L[j - 2^i]
    L[j] ← y
  endfor
endfor
```

COMPUTING SUM OF AN ARRAY ON EREW PRAM

```
Algorithm Sum_EREW
for i = 1 to log n do
  forall Pj, where 1 ≤ j ≤ n/2 do in parallel
    if (2j modulo 2^i) = 0 then
      A[2j] ← A[2j] + A[2j - 2^(i-1)]
    endif
  endfor
endfor
```

COMPUTING ALL PARTIAL SUM

```
Algorithm AllSums_EREW
for i = 1 to log n do
  forall Pj, where 2^(i-1) + 1 ≤ j ≤ n do in parallel
    A[j] ← A [j] + A[j - 2^(i-1)]
  endfor
endfor
```

MATRIX MULTIPLICATION (CREW PRAM)

```
Algorithm MatMult_CREW

/* Step 1 */
forall Pi,j,k, where 1 ≤ i, j, k ≤ n do in parallel
  C[i,j,k] ← A[i,k] * B[k,j]
endfor

/* Step 2 */
for 1 = 1 to log n do
  forall Pi,j,k, where 1≤i,j≤n & 1≤k≤n/2 do in parallel
    if (2k modulo 2^1) = 0 then
      C[i,j,2k] ← C[i,j,2k] + C[i,j, 2k - 2^(1-1)]
    endif
  endfor
/* The output matrix is stored in locations
   C[i,j,n], where 1≤ i,j ≤ n   */
endfor
```