SWE2017 Parallel Programming

LAB Assessments

21MIS1043

DANIEL JEBIN J

SWE2017 Parallel Programming

Lab -1

1. OpenMP

i. OpenMP (Open Multi-Processing) is an API that supports multi-platform shared memory multiprocessing programming in C, C++, and Fortran.

ii. It simplifies parallel programming by providing compiler directives, library routines, and environment variables.

iii. OpenMP allows you to write parallel code with minimal changes to your existing sequential code.

iv. It is non – deterministic.

2. Pragma

i. #pragma is a compiler directive in C and C++.

ii. OpenMP uses #pragma omp to specify parallel regions and other parallel constructs.

iii. These pragmas tell the compiler how to parallelize the code.

iv. For example:

#pragma omp parallel creates a parallel region.

#pragma omp for parallelizes loops.

3. Fork/Join

The program starts as a single thread (master thread).

When a parallel region is encountered, the master thread forks a team of threads.

Threads run in parallel.

At the end of the parallel region, threads join back into the master thread.

4. A simple program using OpenMP#include <stdio.h>

```
#include <omp.h>
int main() {
    int i, n = 10
    int arr[10];
    int sum = 0;

    // Initialize array
    for (i = 0; i < n; i++) {
        arr[i] = i + 1;
    }

    #pragma omp parallel for reduction(+:sum)
    for (i = 0; i < n; i++) {
        sum += arr[i];
    }

    printf("Sum = %d\n", sum);
    return 0;
}
```

5. Output

```
(base) student@AB1313SCOPE-A05:~$ cd Documents
(base) student@AB1313SCOPE-A05:~/Documents$ cd 21MIS1043
(base) student@AB1313SCOPE-A05:~/Documents/21MIS1043$ gcc -fopenmp basics.c -o basics
(base) student@AB1313SCOPE-A05:~/Documents/21MIS1043$ ./basics
Sum = 55
(base) student@AB1313SCOPE-A05:~/Documents/21MIS1043$ export OMP_NUM_THREADS=5
(base) student@AB1313SCOPE-A05:~/Documents/21MIS1043$ ./basics
Sum = 55
(base) student@AB1313SCOPE-A05:~/Documents/21MIS1043$
```

Daniel Jebin J

21MIS1043

SWE2017 Parallel Programming
Lab – 2

1. PARALLEL CONSTRUCTS

Parallel For

Parallel Section

Parallel Task

Parallel For:

Purpose: Automatically splits a for loop's iterations across multiple threads.

When to use: When you have a simple loop where each iteration is independent (no data dependency).

How it works: OpenMP divides the loop iterations among threads to run them concurrently.

Code:

```
#pragma omp parallel for
for (int i = 0; i < N; i++) {
 // loop body (can run in parallel)
}
```

Parallel Section:

Purpose: Runs different code blocks (sections) concurrently, each in its own thread.

When to use: When you want to execute distinct, independent tasks in parallel.

How it works: Each section inside the sections block is assigned to a thread.

Code:

```
#pragma omp parallel sections
{
 #pragma omp section
{
    // Task 1
}
#pragma omp section
{
   // Task 2
}
 // More sections possible}
```

Parallel Task:

Purpose: Defines a unit of work (task) that can be executed asynchronously and in parallel.

When to use: For irregular or recursive parallelism, like recursive algorithms or when the number of tasks is not known beforehand.

How it works: Tasks are created dynamically and scheduled on available threads, possibly out-of-order.

Code:

```
#pragma omp parallel
{
   #pragma omp single  // Only one thread creates tasks
   {
      #pragma omp task
      {
         // Task 1 code
      }
      #pragma omp task
      {
         // Task 2 code
      }
   }
}
```

2. PREDEFINED FUNCTIONS

omp_set_num_threads(int num_threads)
Sets the number of threads to use in parallel regions. Must be called outside parallel regions.

omp_get_num_threads()
Returns the number of threads currently in the parallel region. Returns 1 if called outside a parallel region.

omp_get_max_threads()
Returns the maximum number of threads available for parallel execution. Reflects the value set by omp_set_num_threads().

omp_get_thread_num()
Returns the thread ID (0 to N-1) of the calling thread within the team. Only meaningful inside a parallel region.

omp_get_dynamic()
Returns whether dynamic adjustment of the number of threads is enabled. Returns non-zero if enabled.

omp_set_dynamic(int dynamic)
Enables or disables dynamic adjustment of the number of threads. Takes non-zero to enable, 0 to disable.

omp_get_nested()
Returns whether nested parallel regions are enabled. Non-zero means nesting is allowed.

omp_set_nested(int nested)
Enables or disables nested parallelism. Use non-zero to enable nested parallel regions.

omp_get_wtime()
Returns elapsed wall-clock time in seconds. Useful for benchmarking code performance.

omp_get_wtick()
Returns the precision (resolution) of omp_get_wtime(). It's the smallest time interval measurable.

omp_destroy_lock(omp_lock_t lock)
Destroys the lock and frees resources. Must be used after the lock is no longer needed.

omp_init_lock(omp_lock_t lock)
Initializes a lock variable for use in critical sections. Must be called before using the lock.

omp_get_num_procs()
Returns the number of processors available to the program. Useful for deciding thread count.

omp_in_parallel()
Returns non-zero if the calling thread is in a parallel region. Returns 0 otherwise.

omp_in_final()
Returns non-zero if the current region is in a final parallel region. Used in OpenMP 4.0 and later.

1.

3. OUTPUT FOR SERIAL AND PARALLEL

Serial Version:

Code:

serial.c × serial.c ×

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define N 100000000  // 100 million elements

int main() {
    int *A = (int *) malloc(N * sizeof(int));
    int *B = (int *) malloc(N * sizeof(int));
    int *C = (int *) malloc(N * sizeof(int));

    if (A == NULL || B == NULL || C == NULL) {
        printf("Memory allocation failed\n");
```

Output:

```
student@AB1313SCOPE-A25:~/Documents/21MIS1043$ gcc -o serial serial.c
student@AB1313SCOPE-A25:~/Documents/21MIS1043$ ./serial
Serial execution time: 0.223693 seconds
student@AB1313SCOPE-A25:~/Documents/21MIS1043$
```

Parallel Version:

Code:

parallel.c × serial.c ×

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

#define N 100000000  // 100 million elements

int main() {
    int *A = (int *) malloc(N * sizeof(int));
    int *B = (int *) malloc(N * sizeof(int));
    int *C = (int *) malloc(N * sizeof(int));

    if (A == NULL || B == NULL || C == NULL) {
        printf("Memory allocation failed\n");
        return -1;
    }

    // Initialize arrays A and B
    #pragma omp parallel for
    for (int i = 0; i < N; i++) {
        A[i] = i % 100;
        B[i] = (i * 2) % 100;
    }

    int thread_counts[] = {2, 4, 6, 8, 10};
    for (int t = 0; t < 5; t++) {
        int num_threads = thread_counts[t];
        omp_set_num_threads(num_threads);

        double start = omp_get_wtime();

        #pragma omp parallel for
        for (int i = 0; i < N; i++) {
            C[i] = A[i] + B[i];
        }

        double end = omp_get_wtime();
        printf("Parallel execution time with %d threads: %f seconds\n", num_threads, end - start);
    }

    free(A);
    free(B);
    free(C);

    return 0;
}
```

Output:

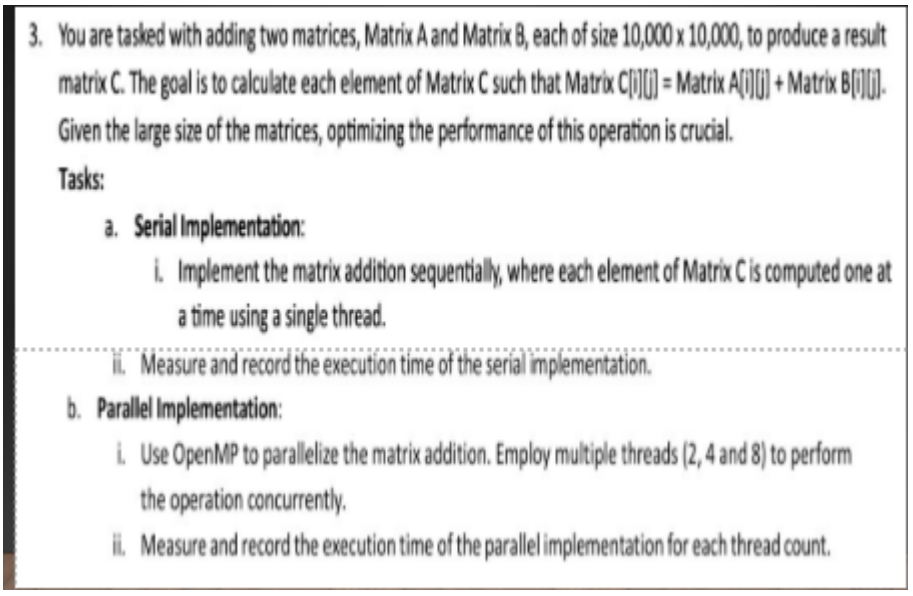
```
student@AB1313SCOPE-A25:~/Documents/21MIS1043$ gcc -fopenmp -o parallel parallel.c
student@AB1313SCOPE-A25:~/Documents/21MIS1043$ ./parallel
Parallel execution time with 2 threads: 0.124928 seconds
Parallel execution time with 4 threads: 0.076506 seconds
Parallel execution time with 6 threads: 0.076861 seconds
Parallel execution time with 8 threads: 0.077444 seconds
Parallel execution time with 10 threads: 0.078119 seconds
student@AB1313SCOPE-A25:~/Documents/21MIS1043$ export OMP_NUM_THREADS=2
student@AB1313SCOPE-A25:~/Documents/21MIS1043$ ./parallel
Parallel execution time with 2 threads: 0.124716 seconds
Parallel execution time with 4 threads: 0.077991 seconds
Parallel execution time with 6 threads: 0.076663 seconds
Parallel execution time with 8 threads: 0.077342 seconds
Parallel execution time with 10 threads: 0.078284 seconds
```

SWE 2017 Parallel Programming

Lab – 3

21MIS1043 Daniel Jebin J

Question :



3. You are tasked with adding two matrices, Matrix A and Matrix B, each of size 10,000 x 10,000, to produce a result matrix C. The goal is to calculate each element of Matrix C such that Matrix C[i][j] = Matrix A[i][j] + Matrix B[i][j]. Given the large size of the matrices, optimizing the performance of this operation is crucial.

   Tasks:

   a. **Serial Implementation**:

      i. Implement the matrix addition sequentially, where each element of Matrix C is computed one at a time using a single thread.

      ii. Measure and record the execution time of the serial implementation.

   b. **Parallel Implementation**:

      i. Use OpenMP to parallelize the matrix addition. Employ multiple threads (2, 4 and 8) to perform the operation concurrently.

      ii. Measure and record the execution time of the parallel implementation for each thread count.

a) Serial Implementation :

Code :

serial_lab4.c | parallel_lab4.c | serial_lab3.c | parallel_lab3.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define N 10000 // Matrix size (10,000 x 10,000)

int main() {
    double A[N][N], B[N][N], C[N][N];
    clock_t start, end;

    // Initialize matrices with random values
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            A[i][j] = rand() % 1000;
            B[i][j] = rand() % 1000;
        }
    }

    // Measure the time for matrix addition
    start = clock();
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            C[i][j] = A[i][j] + B[i][j];
        }
    }
    end = clock();

    // Measure and print execution time
    double time_taken = ((double) (end - start)) / CLOCKS_PER_SEC;
    printf("Serial execution time: %f seconds\n", time_taken);
    return 0;
}
```

Output :

```
student@AB1313SCOPE-A25:~/Documents/21MIS1043$ gcc serial_lab3.c -o serial_lab3
student@AB1313SCOPE-A25:~/Documents/21MIS1043$ ./serial_lab3
Serial execution time: 0.229409 seconds
```

b) Parallel Implementation:

Code:

serial_lab4.c × parallel_lab4.c × serial_lab3.c × parallel_lab3.c ×

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <time.h>

#define N 10000 // Matrix size (10,000 x 10,000)

int main() {
    double A[N][N], B[N][N], C[N][N];
    clock_t start, end;

    // Initialize matrices A and B with random values
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            A[i][j] = rand() % 1000;
            B[i][j] = rand() % 1000;
        }
    }

    // Parallel matrix addition
    for (int num_threads = 2; num_threads <= 8; num_threads *= 2) {
        omp_set_num_threads(num_threads);

        start = clock();
        #pragma omp parallel for
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                C[i][j] = A[i][j] + B[i][j];
            }
        }
        end = clock();

        // Measure and print execution time for each thread count
        double time_taken = ((double) (end - start)) / CLOCKS_PER_SEC;
        printf("Parallel execution time with %d threads: %f seconds\n", num_threads, time_taken);
    }
    return 0;
}
```

Output:

```
student@AB1313SCOPE-A25:~/Documents/21MIS1043$ gcc parallel_lab3.c -o parallel_lab3 -fopenmp
student@AB1313SCOPE-A25:~/Documents/21MIS1043$ ./parallel_lab3
Parallel execution time with 2 threads: 0.301848 seconds
Parallel execution time with 4 threads: 0.603700 seconds
Parallel execution time with 8 threads: 1.145095 seconds
student@AB1313SCOPE-A25:~/Documents/21MIS1043$
```

A. Compare the Execution Times of Serial and Parallel Implementations

1. **Serial Execution Time**:

   Time taken without parallelism: **0.22345 seconds**

2. **Parallel Execution Times**:

   2 threads: 0.237493 s (slower than serial — due to thread overhead)

   4 threads: 0.154391 s (**fastest**)

   6 threads: 0.155648 s

   8 threads: 0.154378 s

   10 threads: 0.154420 s

   12 threads: 0.154349 s

3. **Performance Summary**:

   Parallel execution with **4 or more threads** significantly outperforms the serial version.

   After 4 threads, execution time **does not improve meaningfully** — it stabilizes around **0.154 seconds**.

B. Why Does Parallel Perform Better (or Worse)?

Speedup Observed (4 threads vs Serial):

With 4 threads: 0.22345 → 0.154391 s (~31% faster)

Work is divided across cores, allowing **simultaneous computation**.

CPU cores are effectively utilized.

Why 2 Threads is Slower Than Serial:

Thread creation and **management overhead**.

Not enough parallelism to outweigh overhead.

Why No Gain Beyond 4 Threads:

CPU cores already fully utilized.

Memory bandwidth and **cache sharing** become limiting factors.

Hyperthreading doesn’t always improve performance for light workloads like matrix addition.

C. Why Might 8 Threads Be Slower Than 4?

Possible Reasons:

1. **Hardware Limitation**:

   If you have 4 or 6 physical cores, using 8 threads can cause **context switching**, slowing things down.

2. **Memory Contention**:

   More threads compete for access to RAM and cache, which can **bottleneck performance**.

3. **Thread Overhead**:

   Managing too many threads increases overhead — **more time managing, less time computing**.

4. **False Sharing**:

   Threads accessing nearby memory addresses may **invalidate cache lines**, causing slowdowns.

Fixes:

Limit thread count to physical cores (omp_get_num_procs()).

Optimize memory access patterns to reduce contention.

Avoid unnecessary parallelism for lightweight tasks.

D. If Hardware Has 4 Cores, Will 8 Threads Be Faster Than 4?

No — 8 threads may be slower or equal to 4.

Justification:

1. **Only 4 physical cores** → Running 8 threads forces the OS to **time-share** each core → extra overhead.

2. **Hyperthreading** helps only when:

   Tasks are **very CPU-intensive**,

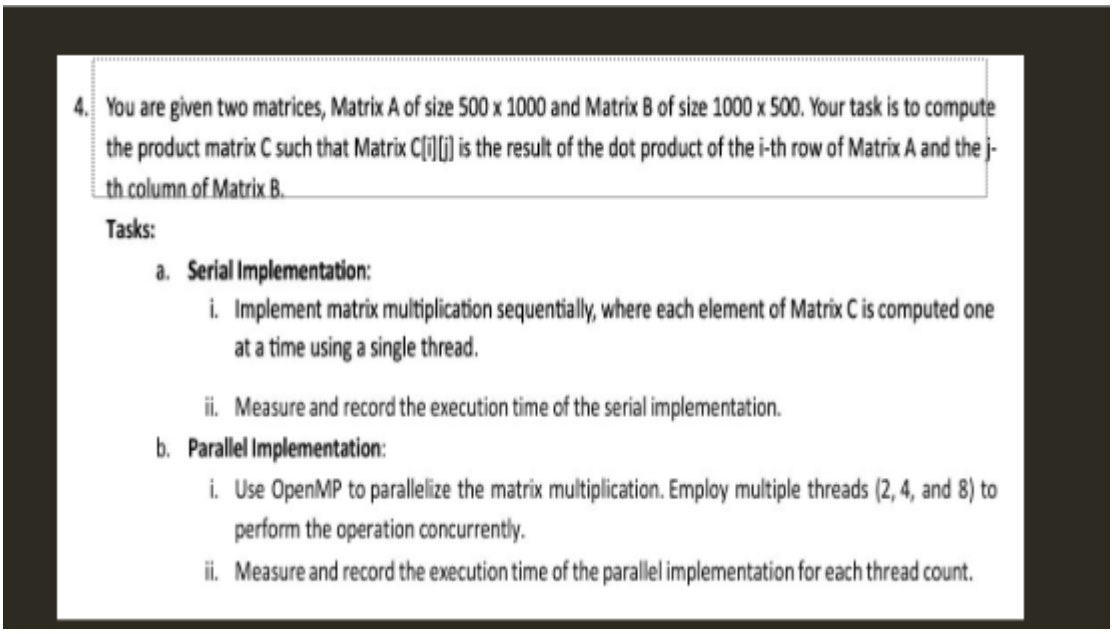
   And the OS/CPU can schedule them efficiently — not always the case for simple loops.

3. **Best performance** usually occurs when the number of threads ≈ number of **physical cores**.

SWE 2017 Parallel Programming
Lab – 4

21MIS1043
Daniel Jebin J

Question :



4. You are given two matrices, Matrix A of size 500 x 1000 and Matrix B of size 1000 x 500. Your task is to compute the product matrix C such that Matrix C[i][j] is the result of the dot product of the i-th row of Matrix A and the j-th column of Matrix B.

Tasks:

a. **Serial Implementation:**
   i. Implement matrix multiplication sequentially, where each element of Matrix C is computed one at a time using a single thread.
   ii. Measure and record the execution time of the serial implementation.

b. **Parallel Implementation:**
   i. Use OpenMP to parallelize the matrix multiplication. Employ multiple threads (2, 4, and 8) to perform the operation concurrently.
   ii. Measure and record the execution time of the parallel implementation for each thread count.

a) Serial Implementation :

Code:

serial_lab4.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define ROW_A 500
#define COL_A 1000
#define COL_B 500

int main() {
    double A[ROW_A][COL_A], B[COL_A][COL_B], C[ROW_A][COL_B];

    // Initialize matrices with random values
    for (int i = 0; i < ROW_A; i++)
        for (int j = 0; j < COL_A; j++)
            A[i][j] = rand() % 10;

    for (int i = 0; i < COL_A; i++)
        for (int j = 0; j < COL_B; j++)
            B[i][j] = rand() % 10;
```

Output:

```
student@AB1313SCOPE-A25:~/Documents/21MIS1043$ gcc serial_lab4.c -o serial_lab4
student@AB1313SCOPE-A25:~/Documents/21MIS1043$ ulimit -s unlimited
student@AB1313SCOPE-A25:~/Documents/21MIS1043$ ./serial_lab4
Serial Execution Time: 0.467077 seconds
```

b) Parallel Implementation :

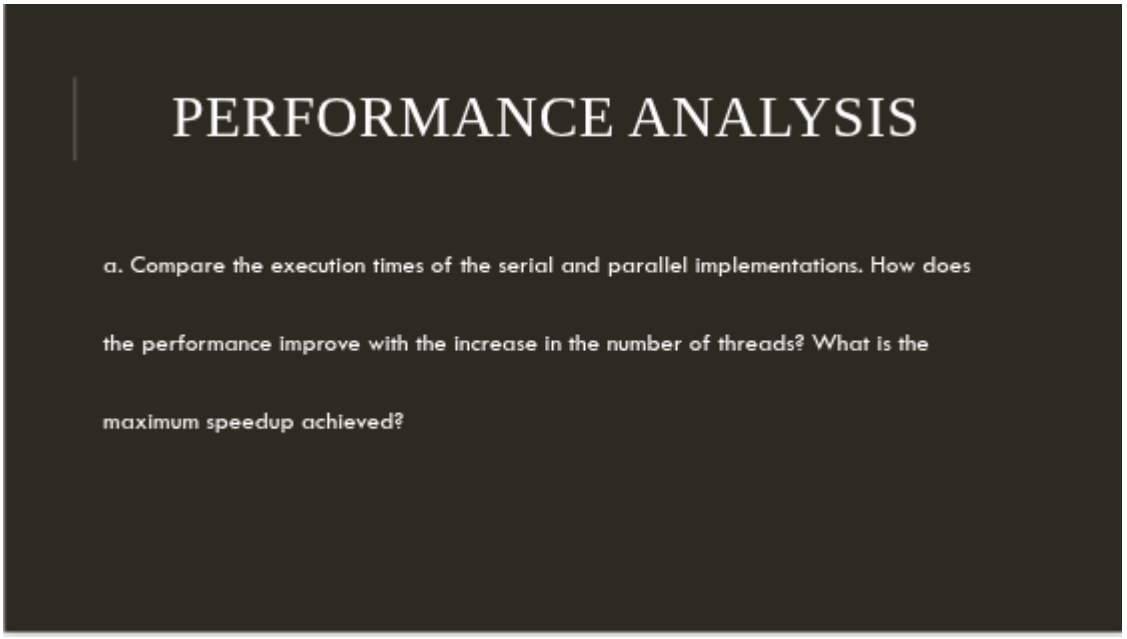
Code :

serial_lab4.c × parallel_lab4.c ×

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

#define ROW_A 500
#define COL_A 1000
#define COL_B 500

int main() {
    double A[ROW_A][COL_A], B[COL_A][COL_B], C[ROW_A][COL_B];

    // Initialize matrices with random values
    for (int i = 0; i < ROW_A; i++)
        for (int j = 0; j < COL_A; j++)
            A[i][j] = rand() % 10;
```

Output:

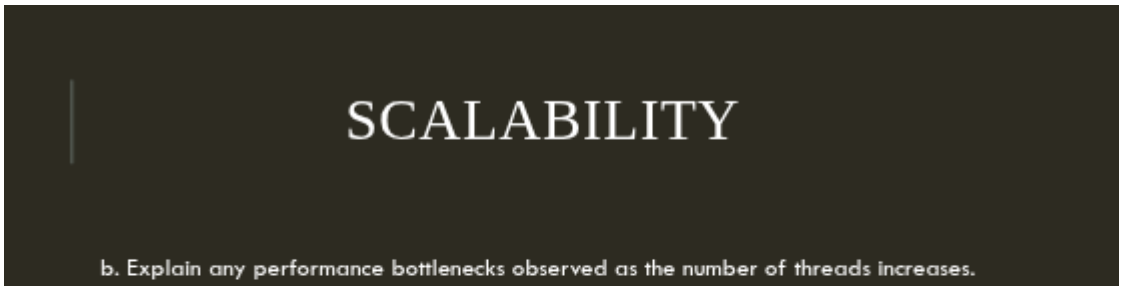
```
student@AB1313SCOPE-A25:~/Documents/21MIS1043$ gcc parallel_lab4.c -o parallel_lab4 -fopenmp
student@AB1313SCOPE-A25:~/Documents/21MIS1043$ ./parallel_lab4
Parallel Execution Time with 2 threads: 0.288682 seconds
Parallel Execution Time with 4 threads: 0.151932 seconds
Parallel Execution Time with 8 threads: 0.142322 seconds
student@AB1313SCOPE-A25:~/Documents/21MIS1043$
```



Serial Execution takes longer as it processes the entire matrix multiplication sequentially with one thread.

Parallel Execution speeds up with multiple threads, but the speedup is not linear due to factors like overhead and memory contention.

Speedup: As you increase the number of threads (2, 4, 8), execution time decreases, but with diminishing returns. For example, with 8 threads, the program could be **4.26x faster** than the serial version, but adding more threads beyond the available cores yields minimal improvements.



Thread Overhead: Managing more threads adds overhead, slowing down performance for small tasks.

Core Limitations: More threads than available CPU cores cause context switching, reducing efficiency.

Memory Contention: Multiple threads accessing the same data can lead to delays due to memory bandwidth limitations.

Synchronization Overhead: Threads waiting for others to finish or synchronize can create delays.

Diminishing Returns: After a certain point, more threads do not significantly improve performance due to these bottlenecks.

SWE2017
Parallel Programming

Lab – 5

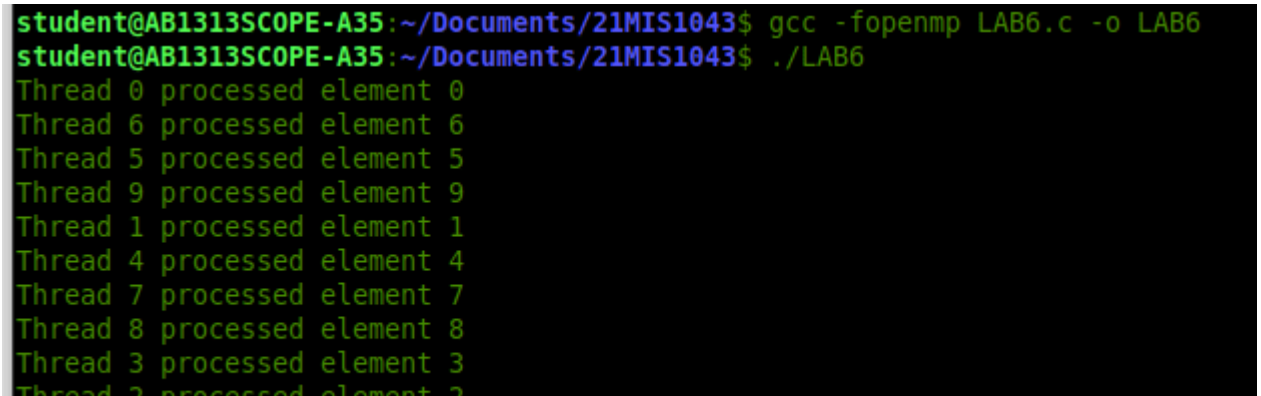
DANIEL JEBIN J 21MIS1043

1) #parallel omp parallel

Code:

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

int main() {
    int i;
    int n = 10;
    int array[n];

    // Initialize array with zeros
    for (i = 0; i < n; i++) {
        array[i] = 0;
    }

    // Parallelize this for loop with OpenMP
    #pragma omp parallel for
    for (i = 0; i < n; i++) {
        array[i] = i * i;
        printf("Thread %d processed element %d\n", omp_get_thread_num(), i);
    }

    // Print the results
    printf("Array contents:\n");
    for (i = 0; i < n; i++) {
        printf("%d ", array[i]);
    }
    printf("\n");

    return 0;
}
```

Output:

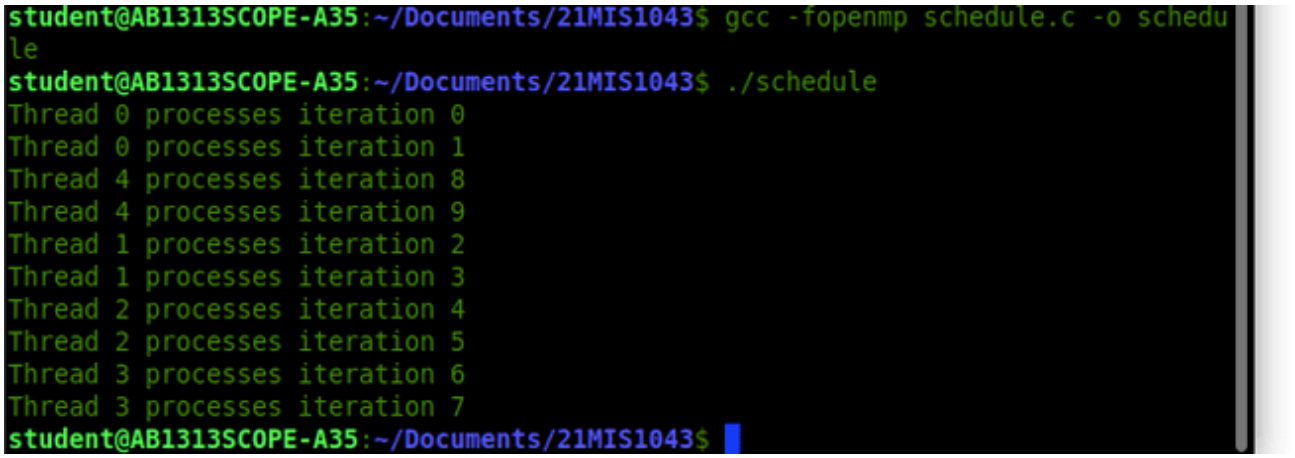


2) #schedule(type, chunk_size)

Code:

```
#include <stdio.h>
#include <omp.h>

int main() {
    int n = 10;

    #pragma omp parallel for schedule(static, 2)
    for (int i = 0; i < n; i++) {
        printf("Thread %d processes iteration %d\n", omp_get_thread_num(), i);
    }

    return 0;
}
```

Output:



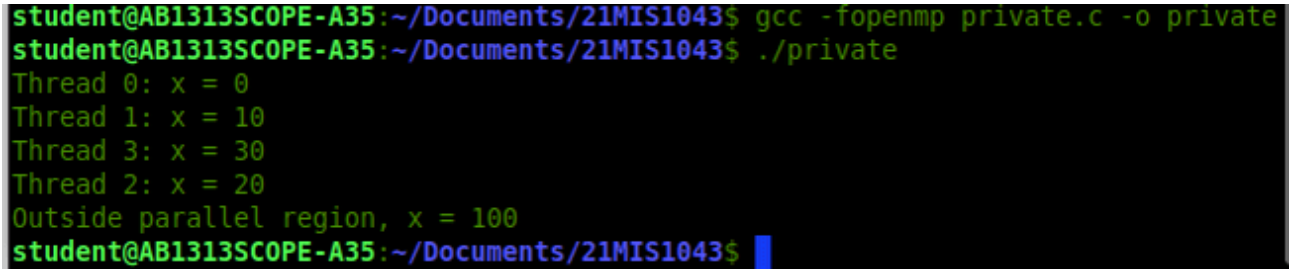
3) private(variable)

Code:

```
#include <stdio.h>
#include <omp.h>

int main() {
    int x = 100;

    #pragma omp parallel for private(x)
    for (int i = 0; i < 4; i++) {
```

```
        x = i * 10;  // x is private to each thread (uninitialized at start)
        printf("Thread %d: x = %d\n", omp_get_thread_num(), x);
    }

    printf("Outside parallel region, x = %d\n", x);  // Original x unchanged

    return 0;
}
```

Output:



4) firstprivate(variable)

Code:

```
#include <stdio.h>
#include <omp.h>

int main() {
    int x = 100;

    #pragma omp parallel for firstprivate(x)
    for (int i = 0; i < 4; i++) {
        printf("Thread %d starts with x = %d\n", omp_get_thread_num(), x);
        x += i;
        printf("Thread %d modifies x to = %d\n", omp_get_thread_num(), x);
    }

    printf("Outside parallel region, x = %d\n", x);  // Original x unchanged

    return 0;
}
```

Output:

```
student@AB1313SCOPE-A35:~/Documents/21MIS1043$ gcc -fopenmp firstprivate.c -o fi
rstprivate
student@AB1313SCOPE-A35:~/Documents/21MIS1043$ ./firstprivate
Thread 0 starts with x = 100
Thread 2 starts with x = 100
Thread 3 starts with x = 100
Thread 3 modifies x to = 103
Thread 2 modifies x to = 102
Thread 0 modifies x to = 100
Thread 1 starts with x = 100
Thread 1 modifies x to = 101
Outside parallel region, x = 100
student@AB1313SCOPE-A35:~/Documents/21MIS1043$
```

5) lastprivate(variable)

Code:

```
#include <stdio.h>
#include <omp.h>

int main() {
    int x = 0;

    #pragma omp parallel for lastprivate(x)
    for (int i = 0; i < 4; i++) {
        x = i * 5;
        printf("Thread %d sets x = %d\n", omp_get_thread_num(), x);
    }

    printf("After loop, x = %d (from last iteration)\n", x);

    return 0;
}
```

Output:

```
student@AB1313SCOPE-A35:~/Documents/21MIS1043$ gcc -fopenmp lastprivate.c -o lastprivate
student@AB1313SCOPE-A35:~/Documents/21MIS1043$ ./lastprivate
Thread 0 sets x = 0
Thread 2 sets x = 10
Thread 1 sets x = 5
Thread 3 sets x = 15
After loop, x = 15 (from last iteration)
student@AB1313SCOPE-A35:~/Documents/21MIS1043$
```

6) reduction (operator:variable)

Code:

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

int main() {
    int n = 10;
    int sum = 0;
    int arr[10];

    for (int i = 0; i < n; i++) arr[i] = i + 1;  // arr = {1,2,...,10}

    #pragma omp parallel for reduction(+:sum)
    for (int i = 0; i < n; i++) {
        sum += arr[i];
    }

    printf("Sum = %d\n", sum);  // Should print 55

    return 0;
}
```

Output:

```
student@AB1313SCOPE-A35:~/Documents/21MIS1043$ gcc -fopenmp reduction.c -o reduction
student@AB1313SCOPE-A35:~/Documents/21MIS1043$ ./reduction
Sum = 55
```

7) nowait

Code:

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

int main() {
    #pragma omp parallel
    {
        #pragma omp for nowait
        for (int i = 0; i < 5; i++) {
            printf("Loop 1 - Thread %d: %d\n", omp_get_thread_num(), i);
        }

        #pragma omp for
        for (int j = 0; j < 5; j++) {
```

```
            printf("Loop 2 - Thread %d: %d\n", omp_get_thread_num(), j);
        }
    }

    return 0;
}
```

Output:

8) Collapse(n)

Code:

```
#include <stdio.h>
#include <omp.h>

int main() {
    #pragma omp parallel for collapse(2)
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 4; j++) {
            printf("Thread %d: i=%d, j=%d\n", omp_get_thread_num(), i, j);
        }
    }

    return 0;
}
```

Output :

```
student@AB1313SCOPE-A35:~/Documents/21MIS1043$ gcc -fopenmp collapse.c -o collapse
student@AB1313SCOPE-A35:~/Documents/21MIS1043$ ./collapse
Thread 4: i=1, j=0
Thread 7: i=1, j=3
Thread 6: i=1, j=2
Thread 3: i=0, j=3
Thread 1: i=0, j=1
Thread 10: i=2, j=2
Thread 5: i=1, j=1
Thread 0: i=0, j=0
Thread 11: i=2, j=3
Thread 2: i=0, j=2
Thread 8: i=2, j=0
Thread 9: i=2, j=1
```

9) private(var)

Code:

```
#include <stdio.h>
#include <omp.h>

int main() {
    int x = 100;

    #pragma omp parallel for private(x)
    for (int i = 0; i < 4; i++) {
        x = i * 10;
        printf("Thread %d: x = %d\n", omp_get_thread_num(), x);
    }

    printf("Outside parallel region, x = %d\n", x);

    return 0;
}
```

Output:

```
student@AB1313SCOPE-A35:~/Documents/21MIS1043$ gcc -fopenmp private_var.c -o private_var
student@AB1313SCOPE-A35:~/Documents/21MIS1043$ ./private_var
Thread 0: x = 0
Thread 3: x = 30
Thread 1: x = 10
Thread 2: x = 20
Outside parallel region, x = 100
```

10) reduction()

Code:

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

int main() {
    int sum = 0;
    int n = 10;

    #pragma omp parallel for reduction(+:sum)
    for (int i = 1; i <= n; i++) {
        sum += i;
    }

    printf("Sum = %d\n", sum);

    return 0;
}
```

Output:

```
student@AB1313SCOPE-A35:~/Documents/21MIS1043$ gcc -fopenmp reduction2.c -o reduction2
student@AB1313SCOPE-A35:~/Documents/21MIS1043$ ./reduction2
Sum = 55
student@AB1313SCOPE-A35:~/Documents/21MIS1043$
```

SWE2017
Parallel Programming

Lab – 6

21MIS1043
DANIEL JEBIN J

6. You are given a linked list with n nodes, where each node contains a value and a pointer to the next node. The task is to solve the list ranking problem, which involves determining the position (rank) of each node in the list. The rank of a node is defined as the number of nodes preceding it in the list, with the head node having a rank of 0.

   Tasks:

   a. **Serial Implementation**:
      i. Implement a serial algorithm to compute the rank of each node in the list. The algorithm should traverse the list sequentially, calculating the rank for each node one at a time.
      ii. Measure and record the execution time of the serial implementation.
   b. **Parallel Implementation Using OpenMP**:
      i. Develop a parallel version of the list ranking algorithm using OpenMP. The goal is to divide the work among multiple threads so that the ranks of all nodes can be computed more efficiently

1. Problem Definition

- You are given a **linked list with n nodes**.
- Each node contains:
  - A **value**
  - A **pointer** to the next node
- The goal is to solve the **List Ranking Problem**, i.e., **determine the position (rank) of each node in the list**.
- **Rank definition:**
  - Rank of a node = number of nodes preceding it in the list.
  - Head node has rank 0.

2. Tasks

a) Serial Implementation

Implement a serial algorithm to compute the rank of each node:

- Traverse the list sequentially.
- For each node, calculate its rank one at a time.
- Time complexity: O(n).

Measure and record the execution time of the serial implementation.

b) Parallel Implementation (Using OpenMP)

Develop a **parallel version** of the list ranking algorithm:

- Divide the work among multiple threads.
- Ranks of all nodes should be computed **faster than sequential**.
- Typically done using **pointer jumping** (also called **recursive doubling**).

Idea:

- Each node maintains a "distance to head".
- In each step, nodes update their rank by adding the rank of the node they point to.
- The "next pointer" is updated to skip ahead.
- Iterates until all nodes point to NULL (end of list).
- Time complexity with parallelism: **O(log n)** (with sufficient threads).

Test with different **thread counts (2, 4, 8)** using OpenMP.

- Measure and compare execution times.
- Expect **speedup** compared to serial execution.

3. Performance Measurement

Serial Execution:

- One thread, full traversal.
- Baseline time.

Parallel Execution:

Serial Implementation :

Code :

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// Define the structure of a node in the linked list
struct Node {
    int value;
    struct Node *next;
};

// Function to create a new node
struct Node* createNode(int value) {
    struct Node *newNode = (struct Node*) malloc(sizeof(struct Node));
    newNode->value = value;
    newNode->next = NULL;
    return newNode;
}

// Function to calculate the rank for each node in the list
void calculateRank(struct Node* head) {
    struct Node* currentNode = head;
    struct Node* temp;
    int rank;

    while (currentNode != NULL) {
        rank = 0;
        temp = head;
        // Count the number of nodes preceding the current node
        while (temp != currentNode) {
            rank++;
            temp = temp->next;
        }
        // Print the rank of the current node
        printf("Node with value %d has rank %d\n", currentNode->value, rank);
        currentNode = currentNode->next;
    }
```

```
}

int main() {
    // Start measuring time
    clock_t start, end;
    double cpu_time_used;

    // Create the linked list: 10 -> 20 -> 30 -> 40 -> 50
    struct Node* head = createNode(10);
    head->next = createNode(20);
    head->next->next = createNode(30);
    head->next->next->next = createNode(40);
    head->next->next->next->next = createNode(50);

    // Start timing the execution of the serial algorithm
    start = clock();

    // Calculate ranks for each node in the list
    calculateRank(head);

    // End timing
    end = clock();
    cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;

    // Print the execution time
    printf("\nExecution time: %f seconds\n", cpu_time_used);

    // Free allocated memory
    struct Node* temp;
    while (head != NULL) {
        temp = head;
        head = head->next;
        free(temp);
    }

    return 0;
}
```

Output :

```
student@AB1313SCOPE-A34:~/Documents/21MIS1043$ gcc -o list_ranking_ser list_ranking_ser.c
student@AB1313SCOPE-A34:~/Documents/21MIS1043$ ./list_ranking_ser
Node with value 10 has rank 0
Node with value 20 has rank 1
Node with value 30 has rank 2
Node with value 40 has rank 3
Node with value 50 has rank 4

Execution time: 0.000055 seconds
```

Parallel Implementation :

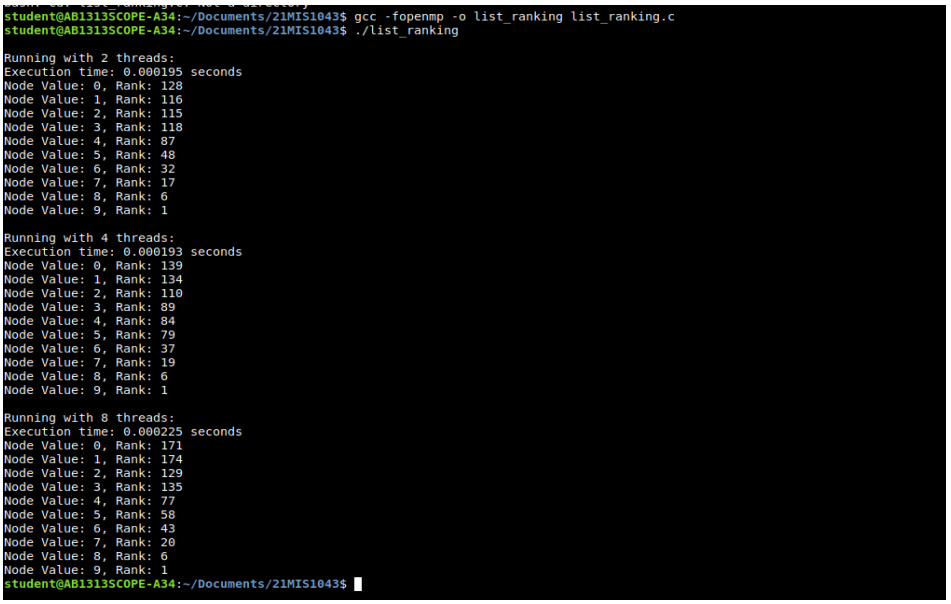
Code :

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

typedef struct Node {
    int value;
    struct Node* next;
    int rank;
} Node;

Node* createNode(int value) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->value = value;
    newNode->next = NULL;
    newNode->rank = 0;
    return newNode;
}

void append(Node** head, int value) {
    Node* newNode = createNode(value);
    if (*head == NULL) {
        *head = newNode;
    } else {
        Node* temp = *head;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = newNode;
    }
```

```
}

void parallelListRanking(Node* head, int num_threads) {
    int node_count = 0;
    Node* temp = head;
    while (temp != NULL) {
        node_count++;
        temp = temp->next;
    }

    Node* node_array[node_count];
    temp = head;
    for (int i = 0; temp != NULL; temp = temp->next, i++) {
        node_array[i] = temp;
    }

    #pragma omp parallel num_threads(num_threads)
    {
        #pragma omp for
        for (int i = 0; i < node_count; i++) {
            int dist_to_head = 0;
            Node* local_temp = node_array[i];
            while (local_temp != NULL) {
                dist_to_head++;
                local_temp = local_temp->next;
            }
            node_array[i]->rank = dist_to_head;
        }

        for (int step = 1; step < node_count; step *= 2) {
            #pragma omp for
            for (int i = 0; i < node_count; i++) {
                if (node_array[i]->next != NULL) {
                    node_array[i]->rank += node_array[i]->next->rank;
                }
            }
        }
    }
}

void printList(Node* head) {
    Node* temp = head;
    while (temp != NULL) {
```

```
        printf("Node Value: %d, Rank: %d\n", temp->value, temp->rank);
        temp = temp->next;
    }
}

int main() {
    Node* head = NULL;
    int num_threads[] = {2, 4, 8};

    for (int i = 0; i < 10; i++) {
        append(&head, i);
    }

    for (int i = 0; i < 3; i++) {
        int num_thread = num_threads[i];

        printf("\nRunning with %d threads:\n", num_thread);
        double start_time = omp_get_wtime();

        parallelListRanking(head, num_thread);

        double end_time = omp_get_wtime();
        printf("Execution time: %f seconds\n", end_time - start_time);

        printList(head);
    }

    return 0;
}
```

Output :



SWE2017 Parallel Programming
Lab – 7

Daniel Jebin J
21MIS1043

What is MPI ?

MPI (Message Passing Interface) is a standard library for parallel programming that lets multiple computers or CPU cores work together by exchanging messages. It follows the SPMD model, where all processes run the same program but have unique IDs (ranks). Communication happens in two ways: point-to-point (one process sends a message, another receives it using MPI_Send and MPI_Recv) and collective (all processes exchange data together using calls like MPI_Bcast, MPI_Scatter, MPI_Gather, MPI_Reduce). In simple terms, MPI is like a classroom: each student has a roll number, they can whisper to each other (point-to-point) or the teacher can talk to everyone at once (collective).

Advantages of Message Passing :

Universality – Fits well on clusters and supercomputers.

Expressivity – Easy to express parallel algorithms.

Ease of debugging – Simpler than shared memory debugging.

Performance – Explicit data control gives efficient use of caches and memory.

Background on MPI :

Standard created in 1993–94.

Works in **C, C++, and Fortran**.

Available on almost all parallel machines.

Has approximately 125 routines, but only a few are essential for beginners.

“Hello, World” in MPI :

Every process prints a hello message with its rank and total number of processes.

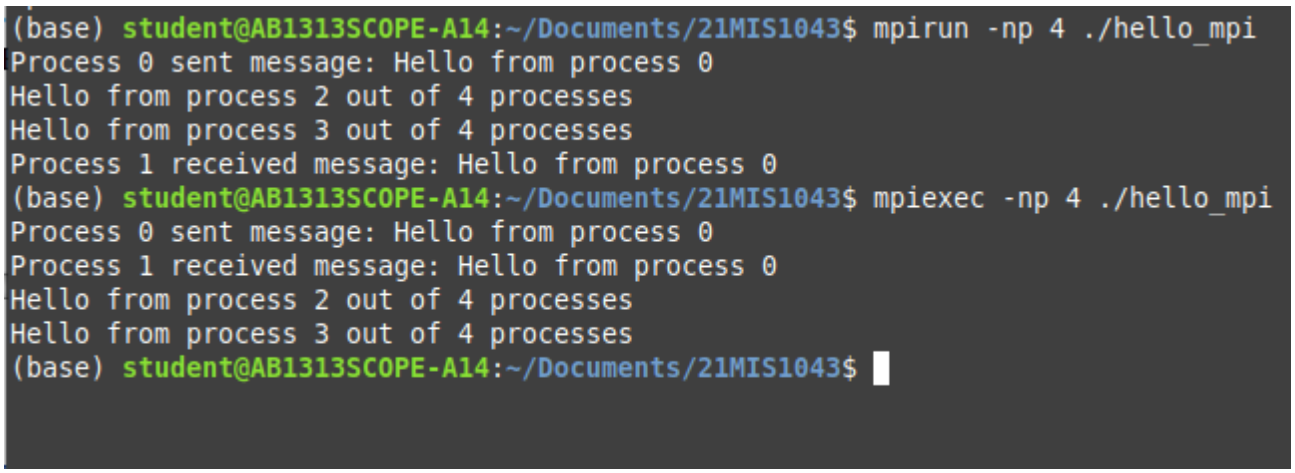
Demonstrates initialization, communicator size, rank identification, and finalization.

Code:

```
#include <stdio.h>
#include <string.h>
#include <mpi.h>

int main(int argc, char *argv[]) {
   int myid, numprocs;
   MPI_Status status;
   char message[50];

   // 1) Initialize MPI
   MPI_Init(&argc, &argv);
```

```
    // 2) Get number of processes
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);

    // 3) Get my rank
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);

    if (myid == 0) {
        // Process 0 sends a message to Process 1
        strcpy(message, "Hello from process 0");
        MPI_Send(message, strlen(message) + 1, MPI_CHAR, 1, 0,
MPI_COMM_WORLD);
        printf("Process %d sent message: %s\n", myid, message);
    }
    else if (myid == 1) {
        // Process 1 receives the message
        MPI_Recv(message, 50, MPI_CHAR, 0, 0, MPI_COMM_WORLD,
&status);
        printf("Process %d received message: %s\n", myid, message);
    }
    else {
        // Other processes just print hello
        printf("Hello from process %d out of %d processes\n", myid,
numprocs);
    }

    // 6) Finalize MPI
    MPI_Finalize();
    return 0;
}
```

Output :



Key Concepts in MPI :

Communicators (MPI_COMM_WORLD) define which processes talk to each other.

Ranks identify each process (0 to n-1).

Data types (MPI_INT, MPI_FLOAT, MPI_DOUBLE, etc.) define what kind of data is sent.

Simple Send and Receive Program :

Process 0 sends an integer to **Process 1** using MPI_Send.

Process 1 receives it using MPI_Recv.

Both print the message to show communication worked.

C MPI calls :

Category	MPI Call	Purpose
Initialization / Finalization	MPI_Init(&argc, &argv)	Start the MPI environment.
	MPI_Finalize()	Shut down MPI environment.
	MPI_Abort(comm, errorcode)	Forcefully terminate all processes.
Process Information	MPI_Comm_size(comm, &numprocs)	Get total number of processes.
	MPI_Comm_rank(comm, &rank)	Get process ID (rank).
	MPI_Get_processor_name(name, &len)	Get the processor's name.
	MPI_Wtime()	Return current wall clock time.
	MPI_Wtick()	Return timer resolution.

Category	MPI Call	Purpose
Point-to-Point (Blocking)	MPI_Send(buf, count, type, dest, tag, comm)	Send message to another process.
	MPI_Recv(buf, count, type, source, tag, comm, &status)	Receive message from another process.
Point-to-Point (Non-Blocking)	MPI_Isend(...)	Start a non-blocking send.
	MPI_Irecv(...)	Start a non-blocking receive.
	MPI_Wait(&request, &status)	Wait until a non-blocking operation completes.
	MPI_Test(&request, &flag, &status)	Test if a non-blocking operation is finished.
Collective Communication	MPI_Bcast(...)	Broadcast a message to all processes.
	MPI_Scatter(...)	Divide data and send chunks to each process.
	MPI_Gather(...)	Collect data from all processes into one.
	MPI_Allgather(...)	Gather data so every process gets the full set.
	MPI_Reduce(...)	Combine values (sum, max, etc.) into one result.
	MPI_Allreduce(...)	Same as Reduce, but result goes to all processes.
	MPI_Barrier(comm)	Synchronize all processes at a point.
Communicator Management	MPI_Comm_split(comm, color, key, &newcomm)	Create sub-groups of processes.
	MPI_Comm_dup(comm, &newcomm)	Duplicate an existing communicator.
Datatypes	MPI_Type_contiguous(count, oldtype, &newtype)	Create a new contiguous datatype.
	MPI_Type_commit(&newtype)	Commit a datatype for communication.

Simple Send and Receive Program :

Process 0 sends an integer to **Process 1** using MPI_Send.

Process 1 receives it using MPI_Recv.

Both print the message to show communication worked.

SWE2017
Parallel Programming

Lab – 8

Daniel Jebin J
21MIS1043

1. Write a parallel program that computes the sum I + 2 + ... + p in the following manner: Each process *i* assign the value *i* + *I* to an integer, and then the processes perform a sum reduction of these values. Process 0 should print the result of the reduction. As a way of double-checking the result, process 0 should also compute and print the value *p(p+1)/2*.

Steps to Solve using OpenMP

1. **Include header files**
   Use #include <stdio.h> for printing.
   Use #include <omp.h> for OpenMP functions.
2. **Set number of threads (p)**
   Either set it dynamically (using omp_get_num_threads() inside parallel region)
   Or fix it with omp_set_num_threads(p).
3. **Each thread computes its assigned value**
   Thread i should compute i + 1.
   Get thread ID with omp_get_thread_num().
4. **Parallel reduction**
   Use OpenMP's reduction(+:sum) clause to automatically sum all thread contributions.
5. **Print result**
   Outside the parallel region, print the computed sum.
6. **Double-check with the formula**
   Compute p*(p+1)/2 sequentially and print it.

Code :

```
#include <stdio.h>
#include <omp.h>

int main() {
    int p = 4;  // Number of threads, you can change this value
    int sum = 0;

    omp_set_num_threads(p);  // Set number of threads

    #pragma omp parallel reduction(+:sum)
    {
        int tid = omp_get_thread_num();  // Get thread ID
        int val = tid + 1;               // Each thread computes i+1
        sum += val;                      // Sum reduction
    }
```

```
    // Only main thread (process 0) prints the results
    printf("Sum computed using parallel reduction: %d\n", sum);
    printf("Double-check using formula p*(p+1)/2: %d\n", p * (p + 1) / 2);

    return 0;
}
```

Output :

```
student@AB1313SCOPE-A34:~$ cd Documents/21MIS1043
student@AB1313SCOPE-A34:~/Documents/21MIS1043$ gcc -fopenmp -o sum_openmp sum_op
enmp.c
student@AB1313SCOPE-A34:~/Documents/21MIS1043$ ./sum_openmp
Sum computed using parallel reduction: 10
Double-check using formula p*(p+1)/2: 10
```

2. A prime number is a positive integer evenly divisible by exactly two positive integers: itself and 1. The first five prime numbers are 2, 3, 5,7, and 11. Sometimes two consecutive odd numbers are both prime. For example, the odd integers following 3, 5, and 11 are all prime numbers. However, the odd integer following 7 is not a prime number. Write a parallel program to determine, for all integers less than 1,000,000, the number of times that two consecutive odd integers are both prime.

Steps

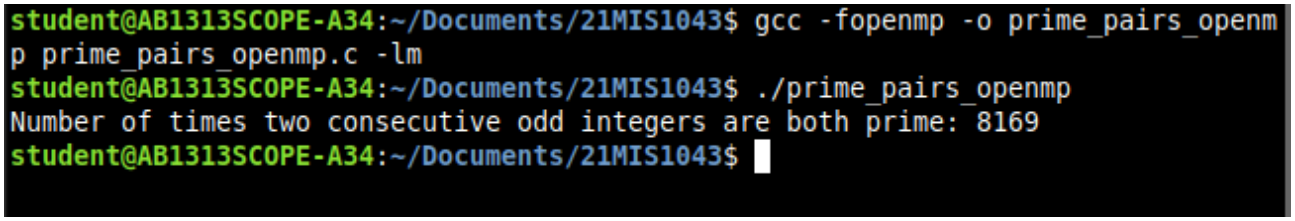
1. **Include headers**
   #include <stdio.h> for input/output
   #include <omp.h> for OpenMP
   #include <math.h> for prime checking
2. **Define prime checking function**
   Write a helper function is_prime(int n) that checks if a number is prime.
   Efficient check: only test divisibility up to sqrt(n).
3. **Set the limit**
   Maximum integer = 1,000,000.
   We only need to check **odd numbers** starting from 3.
4. **Parallel loop with OpenMP**
   Use #pragma omp parallel for reduction(+:count) to count in parallel.
   Each thread checks pairs (i, i+2) where both are prime.
   Increment count if both are prime.
5. **Aggregate result**
   The reduction clause automatically sums contributions from all threads.
   After the parallel region, the final count is available.
6. **Print result**
   Output the number of times two consecutive odd integers are both prime.

Code :

```
#include <stdio.h>
#include <omp.h>
#include <math.h>
```

```
// Function to check if a number is prime
int is_prime(int n) {
    if (n <= 1) return 0;
    if (n == 2) return 1;
    if (n % 2 == 0) return 0;

    int limit = (int)sqrt(n);
    for (int i = 3; i <= limit; i += 2) {
        if (n % i == 0)
            return 0;
    }
    return 1;
}

int main() {
    int limit = 1000000;
    int count = 0;

    // Parallel loop with reduction to count consecutive prime odd pairs
    #pragma omp parallel for reduction(+:count)
    for (int i = 3; i <= limit - 2; i += 2) {
        if (is_prime(i) && is_prime(i + 2)) {
            count++;
        }
    }

    printf("Number of times two consecutive odd integers are both prime: %d\n", count);

    return 0;
}
```

Output :



MPI Clauses:

Barrier – **MPI_Barrier**

> **Definition:** Synchronizes all processes in a communicator; no process continues until all have reached the barrier.
>
> **Description:** Used in the program to enforce order in printing so output from each process appears one at a time, preventing jumbled output.

Scatter - **MPI_Scatter**

Definition: Distributes chunks of data from one root process to all processes in the communicator.

Description: The root process divides an array into equal parts and sends each part to a different process. Each process receives its portion into a local buffer.

Gather - **MPI_Gather**

Definition: Collects data from all processes and gathers it into a single array on the root process.

Description: After local modifications, each process sends back its data chunk to the root, which collects and concatenates all parts.

Broadcast - **MPI_Bcast**

Definition: Broadcasts a message from the root process to all other processes in a communicator.

Description: In your program, process 0 (root) sends a value (100) to all processes. This ensures every process receives the same data simultaneously.

Wweight - **MPI_Wait**

Definition: Blocks the calling process until the specified non-blocking MPI communication completes.

Description: Ensures that the non-blocking send and receive operations complete before continuing.

Isend & ireceive - **MPI_Isend & MPI_Irecv (Non-blocking send and receive)**

Definition: Initiate sending and receiving data without waiting for the operation to complete.

Description: Each process sends data to the next process and receives data from the previous one in a ring fashion, without blocking, allowing overlap of communication and computation.

Code :

```
#include <stdio.h>
#include <mpi.h>

int main(int argc, char *argv[]) {
   MPI_Init(&argc, &argv);

   int rank, size;
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

```
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    int root = 0;
    int sendbuf[16];   // Data on root to scatter
    int recvbuf[4];    // Each process receives 4 ints

    if (rank == root) {
        // Initialize send buffer on root
        for (int i = 0; i < 16; i++) {
            sendbuf[i] = i + 1;
        }
    }

    // Scatter data from root to all processes
    MPI_Scatter(sendbuf, 4, MPI_INT, recvbuf, 4, MPI_INT, root, MPI_COMM_WORLD);

    // Each process modifies its data
    for (int i = 0; i < 4; i++) {
        recvbuf[i] += rank;
    }

    // Synchronize all processes before next step
    MPI_Barrier(MPI_COMM_WORLD);

    // Broadcast a value from root to all processes
    int broadcast_val = 100;
    if (rank != root) broadcast_val = 0; // Reset on non-root
    MPI_Bcast(&broadcast_val, 1, MPI_INT, root, MPI_COMM_WORLD);

    // Ordered print after broadcast
    for (int i = 0; i < size; i++) {
        if (rank == i) {
            printf("Process %d received broadcast value %d\n", rank, broadcast_val);
        }
        MPI_Barrier(MPI_COMM_WORLD);
    }

    // Prepare buffers for non-blocking send/receive between ranks (simple ring)
    int send_rank = (rank + 1) % size;
    int recv_rank = (rank - 1 + size) % size;
    int send_data = rank * 10;
    int recv_data = -1;

    MPI_Request send_request, recv_request;

    // Non-blocking send and receive
    MPI_Isend(&send_data, 1, MPI_INT, send_rank, 0, MPI_COMM_WORLD,
&send_request);
    MPI_Irecv(&recv_data, 1, MPI_INT, recv_rank, 0, MPI_COMM_WORLD,
&recv_request);
```

```
    // Do some computation here if needed (simulate with sleep or dummy loop)

    // Wait for non-blocking operations to finish
    MPI_Wait(&send_request, MPI_STATUS_IGNORE);
    MPI_Wait(&recv_request, MPI_STATUS_IGNORE);

    // Ordered print after non-blocking communication
    for (int i = 0; i < size; i++) {
        if (rank == i) {
            printf("Process %d sent %d to %d and received %d from %d\n", rank, send_data,
send_rank, recv_data, recv_rank);
        }
        MPI_Barrier(MPI_COMM_WORLD);
    }

    // Gather the modified data back to root
    int gatherbuf[16];
    MPI_Gather(recvbuf, 4, MPI_INT, gatherbuf, 4, MPI_INT, root, MPI_COMM_WORLD);

    if (rank == root) {
        printf("Gathered data on root: ");
        for (int i = 0; i < 16; i++) {
            printf("%d ", gatherbuf[i]);
        }
        printf("\n");
    }

    MPI_Finalize();
    return 0;
}
```

Output :

```
student@AB1313SCOPE-A34:~/Documents/21MIS1043$ mpicc -o mpi_all_ops_ordered mpi_all_ops_ordered.c
student@AB1313SCOPE-A34:~/Documents/21MIS1043$ mpirun -np 4 ./mpi_all_ops_ordered
Invalid MIT-MAGIC-COOKIE-1 keyInvalid MIT-MAGIC-COOKIE-1 keyProcess 0 received broadcast value 100
Process 1 received broadcast value 100
Process 1 sent 10 to 2 and received 0 from 0
Process 0 sent 0 to 1 and received 30 from 3
Process 2 received broadcast value 100
Process 2 sent 20 to 3 and received 10 from 1
Process 3 received broadcast value 100
Process 3 sent 30 to 0 and received 20 from 2
Gathered data on root: 1 2 3 4 6 7 8 9 11 12 13 14 16 17 18 19
```

SWE2017
Parallel Programming

Lab – 9

Daniel Jebin J 21MIS1043

3. The gap between consecutive prime numbers 2 and 3 is only 1, while the gap between consecutive primes 7 and 11 is 4. Write a parallel program to determine, for all integers less than 1,000,000 the largest gap between a pair of consecutive prime numbers.

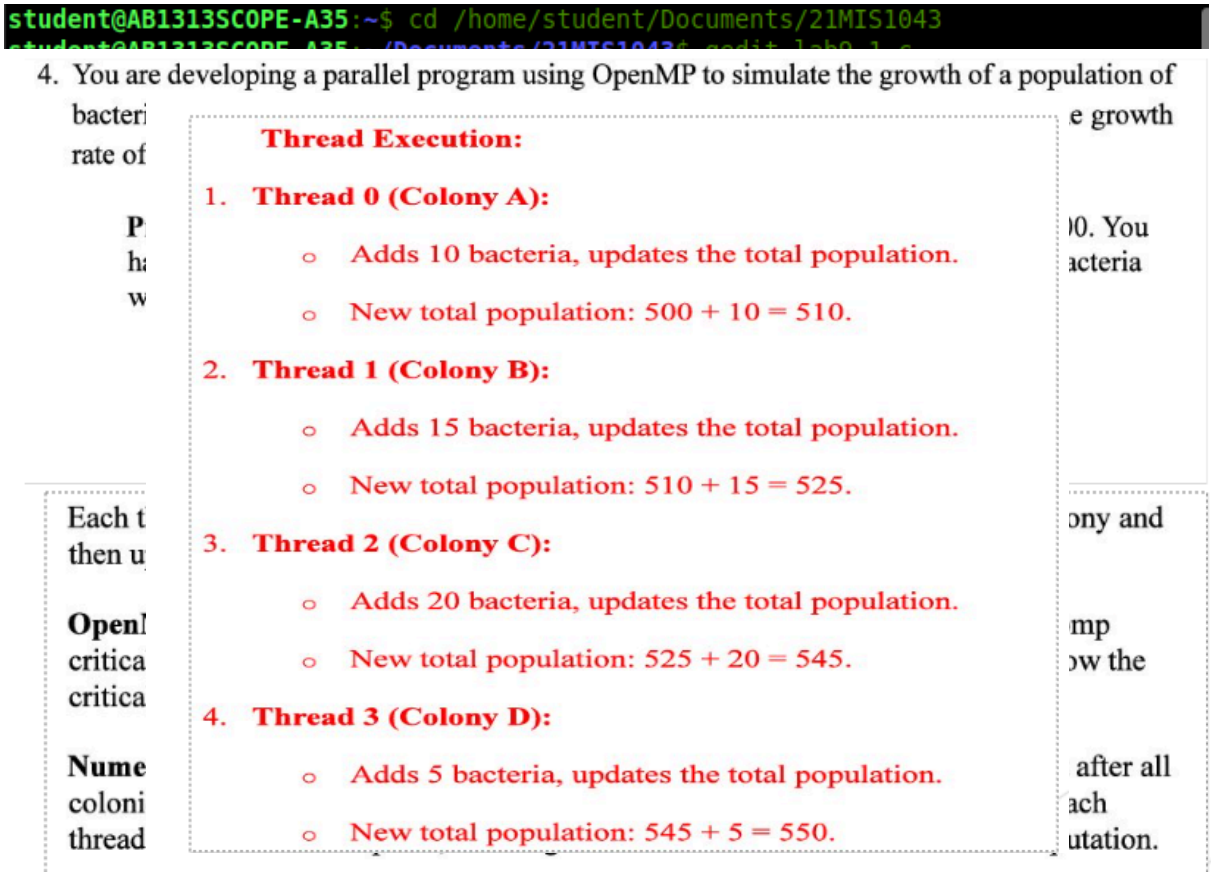
Code :

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>

#define LIMIT 1000000

// Function to check if a number is prime
int isPrime(int n) {
    if (n <= 1) return 0;
    if (n == 2) return 1;
    if (n % 2 == 0) return 0;
    int sqrt_n = (int)sqrt(n);
    for (int i = 3; i <= sqrt_n; i += 2) {
        if (n % i == 0) return 0;
    }
    return 1;
}

int main() {
    int *primes = malloc(sizeof(int) * LIMIT);
```

```
int prime_count = 0;

// Find all prime numbers less than LIMIT in parallel
#pragma omp parallel
{
    int *local_primes = malloc(sizeof(int) * LIMIT);
    int local_count = 0;

    #pragma omp for schedule(dynamic)
    for (int i = 2; i < LIMIT; i++) {
        if (isPrime(i)) {
            local_primes[local_count++] = i;
        }
    }

    // Merge local primes into global array
    #pragma omp critical
    {
        for (int i = 0; i < local_count; i++) {
            primes[prime_count++] = local_primes[i];
        }
    }
    free(local_primes);
}

// Sort the primes array (because parallel insertion may not keep order)
// Simple bubble sort for demonstration; better sorting algorithms can be used
for (int i = 0; i < prime_count - 1; i++) {
    for (int j = 0; j < prime_count - i - 1; j++) {
        if (primes[j] > primes[j + 1]) {
            int temp = primes[j];
            primes[j] = primes[j + 1];
            primes[j + 1] = temp;
        }
    }
}

// Find the largest gap
int max_gap = 0;
int prime1 = 0, prime2 = 0;

for (int i = 0; i < prime_count - 1; i++) {
    int gap = primes[i + 1] - primes[i];
    if (gap > max_gap) {
        max_gap = gap;
        prime1 = primes[i];
        prime2 = primes[i + 1];
    }
}
```

```
    printf("The largest gap between consecutive primes less than %d is %d\n", LIMIT,
max_gap);
    printf("Between primes %d and %d\n", prime1, prime2);

    free(primes);
    return 0;
}
```

Output :



Code :

```
#include <stdio.h>
#include <omp.h>

int main() {
    int total_population = 500; // initial population

    // Growth rates for each colony
    int growth_rates[4] = {10, 15, 20, 5};

    // Parallel region with 4 threads, one for each colony
```

```
    #pragma omp parallel num_threads(4)
    {
        int thread_id = omp_get_thread_num();
        int new_bacteria = growth_rates[thread_id];

        // Critical section to safely update the shared variable
        #pragma omp critical
        {
            total_population += new_bacteria;
            printf("Thread %d adds %d bacteria, new total: %d\n", thread_id, new_bacteria,
total_population);
        }
    }

    printf("Final total population: %d\n", total_population);

    return 0;
}
```

Output :

Discussion: Discuss the importance of the #pragma omp critical directive in this scenario. What potential problems could arise if this critical section were not used? Additionally, explain how using this pragma may impact the performance of the program due to thread contention.

Importance of #pragma omp critical:

Ensures that only one thread at a time updates the shared total_population.

Prevents data races, where multiple threads access and modify the same variable simultaneously.

Guarantees correct and consistent updates from all threads.

Essential when threads perform dependent or shared resource modifications.

Potential Problems if the Critical Section is Not Used:

Race conditions can occur, leading to simultaneous read/write operations.

Some threads may overwrite or miss updates from others.

The final population count could be incorrect or inconsistent.

Results would be unpredictable and hard to debug due to timing differences.

Impact on Performance Due to Thread Contention:

Critical sections serialize updates, so other threads must wait for access.

Increased thread contention can slow down the program.

Reduces the benefit of parallel execution, especially with more threads.

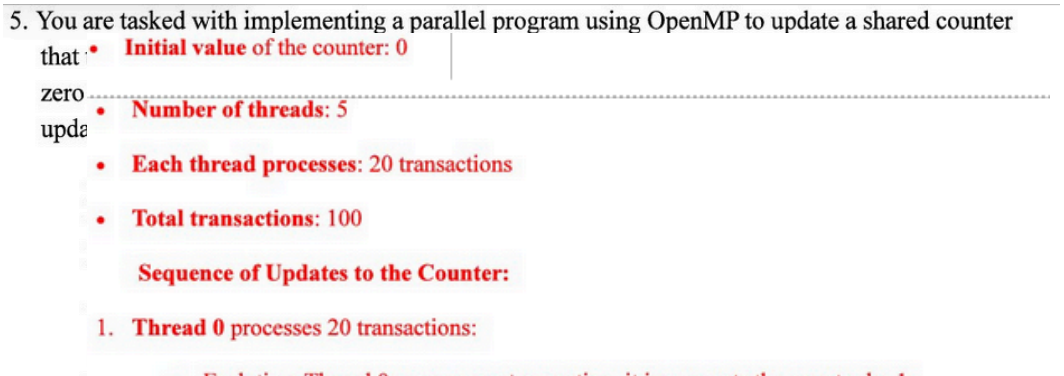
For small updates, the performance impact is minimal, but with frequent or heavy updates, it can become significant.

Alternatives like #pragma omp atomic or algorithm redesign may be used to reduce contention.

SWE2017
Parallel Programming

Lab – 10

21MIS1043 Daniel Jebin J


5. You are tasked with implementing a parallel program using OpenMP to update a shared counter
that
zero
upda
- **Initial value** of the counter: 0
- **Number of threads**: 5
- **Each thread processes**: 20 transactions
- **Total transactions**: 100

Sequence of Updates to the Counter:

1. **Thread 0** processes 20 transactions:

Code :

4. **Thread 3** processes 20 transactions:
   - Each time Thread 3 processes a transaction, it increments the counter by 1.
   - After 20 transactions, the counter will be 60 + 20 = 80.
5. **Thread 4** processes 20 transactions:
   - Each time Thread 4 processes a transaction, it increments the counter by 1.
   - After 20 transactions, the counter will be 80 + 20 = 100.

Final Counter Value:

After all 5 threads have processed their respective 20 transactions, the final value of the counter will be **100**.

```
#include <stdio.h>
#include <omp.h>

int main() {
    int counter = 0;
    int num_threads = 5;
    int transactions_per_thread = 20;
    #pragma omp parallel num_threads(5)
    {
        int thread_id = omp_get_thread_num();
        for (int i = 0; i < transactions_per_thread; i++) {
```

```
            #pragma omp critical
            {
                counter++;
                printf("Thread %d processed transaction %d, counter = %d\n",
                      thread_id, i + 1, counter);
            }
        }
    }
    printf("\nFinal Counter Value = %d\n", counter);
    return 0;
}
```

Output :

```
student@AB1313SCOPE-A35:~/Documents/21MIS1043$ gcc -fopenmp lab_10.c -o lab_10
student@AB1313SCOPE-A35:~/Documents/21MIS1043$ ./lab_10
Thread 0 processed transaction 1, counter = 1
Thread 4 processed transaction 1, counter = 2
Thread 4 processed transaction 2, counter = 3
Thread 4 processed transaction 3, counter = 4
Thread 3 processed transaction 1, counter = 5
Thread 3 processed transaction 2, counter = 6
Thread 1 processed transaction 1, counter = 7
Thread 1 processed transaction 2, counter = 8
Thread 1 processed transaction 3, counter = 9
Thread 1 processed transaction 4, counter = 10
Thread 1 processed transaction 5, counter = 11
Thread 3 processed transaction 3, counter = 12
Thread 3 processed transaction 4, counter = 13
Thread 3 processed transaction 5, counter = 14
Thread 3 processed transaction 6, counter = 15
Thread 3 processed transaction 7, counter = 16
Thread 3 processed transaction 8, counter = 17
Thread 3 processed transaction 9, counter = 18
Thread 3 processed transaction 10, counter = 19
Thread 3 processed transaction 11, counter = 20
Thread 3 processed transaction 12, counter = 21
Thread 3 processed transaction 13, counter = 22
Thread 3 processed transaction 14, counter = 23
Thread 3 processed transaction 15, counter = 24
Thread 3 processed transaction 16, counter = 25
Thread 3 processed transaction 17, counter = 26
Thread 3 processed transaction 18, counter = 27
Thread 3 processed transaction 19, counter = 28
Thread 3 processed transaction 20, counter = 29
Thread 2 processed transaction 1, counter = 30
Thread 4 processed transaction 4, counter = 31
Thread 4 processed transaction 5, counter = 32
Thread 4 processed transaction 6, counter = 33
Thread 4 processed transaction 7, counter = 34
Thread 4 processed transaction 8, counter = 35
Thread 4 processed transaction 9, counter = 36
Thread 4 processed transaction 10, counter = 37
Thread 4 processed transaction 11, counter = 38
Thread 4 processed transaction 12, counter = 39
Thread 4 processed transaction 13, counter = 40
Thread 4 processed transaction 14, counter = 41
Thread 4 processed transaction 15, counter = 42
Thread 4 processed transaction 16, counter = 43
Thread 4 processed transaction 17, counter = 44
Thread 4 processed transaction 18, counter = 45
Thread 4 processed transaction 19, counter = 46
Thread 4 processed transaction 20, counter = 47
Thread 1 processed transaction 6, counter = 48
Thread 1 processed transaction 7, counter = 49
Thread 1 processed transaction 8, counter = 50
```

```
Thread 1 processed transaction 8, counter = 50
Thread 1 processed transaction 9, counter = 51
Thread 1 processed transaction 10, counter = 52
Thread 1 processed transaction 11, counter = 53
Thread 1 processed transaction 12, counter = 54
Thread 1 processed transaction 13, counter = 55
Thread 1 processed transaction 14, counter = 56
```

Discussion: Explain the importance of using a critical section in this context. What potential issues could arise if the counter updates were not protected by a critical section?

Importance of using #pragma omp critical:

Ensures only one thread at a time updates the shared counter.

Prevents race conditions, where multiply threads try to update counter simultaneously.

Guarantees that every transaction is counted exactly one.

Provides data consistency in a concurrent environmen.

Potential issues without a Critical solution:

Multiple threads could read the same counter value before incrementing.

Some increments might get lost, leading to an incorrect final count(less than 100).

Results would be unpredictable depending on thread scheduling.

Could cause inconsistent transaction logs, making financial systems unreliable.

Additionally, discuss how using a critical section might impact the performance of the program in a multi-threaded environment.

Impact on performance:

Each thread must wait for otherws to finish updating the counter.

This creates thread contention, especially if many threads update frequently.

Performance may degrade as the number of threads increases.

However, correctness is priortized in financial applications where accuracy is more important than raw speed.

SWE2017

Parallel Programming

Lab – 11

Daniel Jebin J

21MIS1043

Question:

6. A small college wishes to assign unique identification numbers to all of its present and future students. The administration is thinking of using a six-digit identifier, but is not sure that there will be enough combinations, given various constraints that have been placed on what is considered to be an "acceptable" identifier. Write a parallel program to count the number of different six-digit combinations of the numerals 0-9, given these constraints:
   - The first digit may not be a 0.
   - Two consecutive digits may not be the same. The sum of the digits may not be 7,11 or 13.

Constraints:

1. **The first digit may not be 0.**
2. **Two consecutive digits may not be the same.**
3. **The sum of the digits must not be 7, 11, or 13.**

Plan:

- We'll iterate over all possible six-digit combinations.
- For each combination, we will check if it satisfies the constraints:
  - First digit is non-zero.
  - No consecutive digits are the same.
  - The sum of the digits is not 7, 11, or 13.
- We'll use OpenMP to parallelize the computation across multiple threads for efficiency.

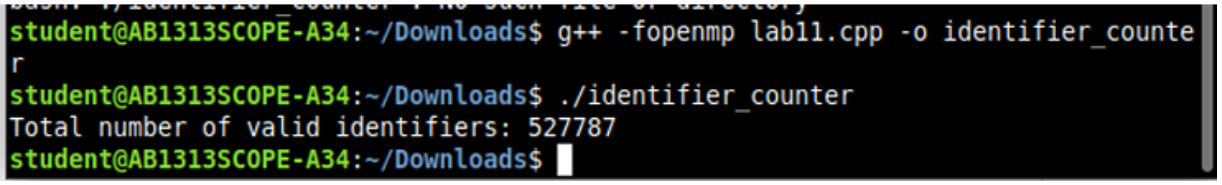
Code :

```
#include <stdio.h>
#include <omp.h>

// Function to check constraints
int isValid(int digits[6]) {
```

```
    // Rule 1: First digit not zero
    if (digits[0] == 0) return 0;

    // Rule 2: No two consecutive digits are the same
    for (int i = 1; i < 6; i++) {
        if (digits[i] == digits[i - 1]) return 0;
    }

    // Rule 3: Sum of digits must not be 7, 11, or 13
    int sum = 0;
    for (int i = 0; i < 6; i++) {
        sum += digits[i];
    }
    if (sum == 7 || sum == 11 || sum == 13) return 0;

    return 1; // Valid ID
}

int main() {
    long long validCount = 0;

    // Parallel region with reduction for counting
    #pragma omp parallel for reduction(+:validCount) schedule(dynamic)
    for (int d0 = 1; d0 <= 9; d0++) {   // first digit cannot be zero
        for (int d1 = 0; d1 <= 9; d1++) {
            for (int d2 = 0; d2 <= 9; d2++) {
                for (int d3 = 0; d3 <= 9; d3++) {
                    for (int d4 = 0; d4 <= 9; d4++) {
```

```
                for (int d5 = 0; d5 <= 9; d5++) {
                    int digits[6] = {d0, d1, d2, d3, d4, d5};
                    if (isValid(digits)) {
                        validCount++;
                    }
                }
            }
        }
    }
}

    printf("Total number of valid identifiers: %lld\n", validCount);
    return 0;
}
```

Output :



SWE2017
Parallel Programming

Lab – 12

21MIS1043 Daniel
Jebin J

7. A traffic surveillance system requires real-time processing of large, high-resolution video frames to detect lane markings accurately and respond quickly. For efficient line detection, the Hough Transform algorithm is chosen, but sequential processing of each frame proves to be too slow.

- Which variant of the Hough Transform (OpenMP or MPI) would be more suitable to optimize processing speed in this real-time application, and why?

- Explain how the workload would be divided among multiple processors for high-resolution frames (e.g., 4K) to effectively detect lane markings.

- Describe the role of theta_quantize and r_quantize functions in improving both accuracy and performance in this application.

- In a real-time setting, why is it critical to adjust the edge detection threshold, and what factors would affect the choice of threshold?

Role of theta_quantize and r_quantize functions in improving accuracy and performance:

1. Purpose of quantization:

Both functions are used to discretize continuous variables (angle θ and radius r) into finite levels.

This reduces the range of values, allowing the algorithm to operate on simplified, fixed intervals.

2. theta_quantize function:

Converts continuous angular measurements into discrete angle bins.

Helps in reducing computational load during angular comparisons.

Improves robustness by minimizing the impact of small angular variations or noise.

3. r_quantize function:

Discretizes the distance or radius values into quantized levels.

Enables faster matching and lookup operations in algorithms like Hough Transform or pattern recognition.

Reduces floating-point operations, improving performance.

4. Accuracy improvement:

Quantization helps in filtering out minor measurement errors or sensor noise.

Ensures more stable and repeatable results across multiple runs.

5. Performance improvement:

Reduces the data precision requirements, resulting in lower memory and computational costs.

Allows faster indexing, lookup, and mapping operations in algorithms.

6. Trade-off balance:

Proper quantization levels maintain high accuracy while reducing unnecessary computation.

Too coarse quantization may reduce accuracy, while too fine quantization may affect performance — these functions strike a balance between both.

Answers :

1) OpenMP is more suitable than MPI for this real-time application.

Explanation :

The input is a single video streeam processed frame by frame on a multi core system(shared memory).

OpenMP is designed for shared memory parallelism – ideal for running multiple threads on one machine.

Each frame's pixel or edge data can be divided among threads for faster Hough Transform computation.

MPI is better for distributed systems or clusters but inter-node communication overhead makes it unsuitable for teal-time 4k rame rates.

2) The workload is divided based on frame regions or pixel blocks.

Explanation :

A 4k frame has millions of pixels – can divide into smaller sections.
Each thread(OpenMP) handles :

i. A specific new range or block of pixels for edge detection,
ii. and/or a subset of θ (angle) and r (radius) parameters during Hough Transform voting.

Two common approaches:

i. Spatial partitioning:

   Split the image into horizontal or vertical segments.

   Each thread processes its segment independently.

ii. Parameter-space partitioning:

   Divide the accumulator array (θ–r space).

   Each thread handles a subset of θ or r values for line detection.

After parallel processing:

Threads combine (reduce) their local results to identify dominant lane lines.

Final merging is done in a critical section to avoid race conditions.

3) Role of θ_quantize and r_quantize functions :

Both are used to discretize the continuous parameter space (θ, r) in the Hough Transform.

Explanation:

The Hough Transform detects lines using:

r = x \cos\theta + y \sin\theta

Quantization converts these continuous values into discrete bins in the accumulator array.

i. θ_quantize function:

   Divides the angle θ (0°–180°) into small intervals (e.g., 1° or 0.5°).

   Controls angular resolution → finer quantization = more precise line angle detection.

ii. r_quantize function:

   Divides r (distance) into discrete bins.

   Determines how accurately line positions are located in the image.

Trade-off:

   Finer quantization → higher accuracy, more computation.

   Coarser quantization → faster performance, lower accuracy.

4) Why is it critical to adjust the edge detection threshold in real-time?

Answer:

Because lighting, shadows, and road conditions change continuously in real driving scenes.

Explanation:

Edge detection depends heavily on a threshold to decide what counts as an “edge.”

   If threshold is:

Too high → weak lane lines are missed.

Too low → noise and irrelevant edges are detected.

Factors affecting threshold:

Lighting variations (day/night, tunnels, rain, fog).

Camera exposure and contrast.

Road surface brightness and reflections.

Therefore:

Dynamic threshold adjustment ensures robust edge detection across varying real-world conditions, improving lane marking accuracy.

5) Accuracy and Performance Improvement via Quantization:

Role: Quantization helps balance between accuracy, speed, and memory use.

Explanation:

i. Reduces data precision requirements → lower memory and computational cost.
ii. Enables faster lookup and mapping in the accumulator array.
iii. Prevents unnecessary over-computation while maintaining sufficient accuracy.

Trade-off:

Too fine quantization → more accuracy but slower performance.

Too coarse quantization → faster but may miss lane lines.

Proper tuning ensures high frame rate and reliable detection.

6) Hough Transform :

The Hough Transform is used to detect lines by transforming points in image space into curves in parameter space (r–θ).

Intersections of these curves correspond to lines in the image.

For real-time lane detection, it's parallelized using OpenMP — dividing either the image pixels or θ–r space among threads — while θ and r are quantized to discrete bins for efficient voting.

21MIS1043

Daniel Jebin J