

**COSC 52063**

Parallel and Distributed Computing

Assignment – 02

**FGS/MSc/CS/2022/007**

Master of Computer Science

Faculty of Graduate Studies

01.

**a.**

FGS/MSc/CS/2022/007

K.Kathiramalairajah Kapilaraj

[kapilraj@outlook.com](mailto:kapilraj@outlook.com)

**b.**

Link <https://github.com/FGS-MSc-CS-2022-007/assignment-2.git>

**c. Title of the Two Algorithms:**

1. Parallel Bubble Sort
2. Parallel Fibonacci Sequence Calculation

I'll provide you with two algorithms implemented in C, one that uses arrays and another that is recursive. Then, I'll show you how to develop parallel versions of each algorithm using OpenMP, a widely-used library for parallel programming in C.

**1.**

**Algorithm using Arrays (Serial): Bubble Sort**

Bubble Sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order.

#include <stdio.h>

void bubbleSort(int arr[], int n) {

int temp;

for (int i = 0; i < n - 1; i++) {

for (int j = 0; j < n - i - 1; j++) {

if (arr[j] > arr[j + 1]) {

temp = arr[j];

arr[j] = arr[j + 1];

arr[j + 1] = temp;

}

}

}

}

int main() {

int arr[] = {64, 34, 25, 12, 22, 11, 90};

int n = sizeof(arr) / sizeof(arr[0]);

bubbleSort(arr, n);

printf("Sorted array: \n");

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

printf("%d ", arr[i]);

}

printf("\n");

return 0;

}

**2.**

**Recursive Algorithm (Serial): Fibonacci Sequence**

The Fibonacci sequence is a classic recursive algorithm. It calculates the nth Fibonacci number using the formula F(n) = F(n-1) + F(n-2) with base cases F(0) = 0 and F(1) = 1.

#include <stdio.h>

int fibonacci(int n) {

if (n <= 0) return 0;

if (n == 1) return 1;

return fibonacci(n - 1) + fibonacci(n - 2);

}

int main() {

int n = 10;

int result = fibonacci(n);

printf("Fibonacci(%d) = %d\n", n, result);

return 0;

}

**d.**

**Brief Description of the Algorithms:**

a)

Parallel Bubble Sort: Bubble Sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. The parallel version of Bubble Sort utilizes OpenMP to parallelize the inner loop where elements are compared and swapped. This parallelization strategy aims to improve the sorting efficiency for large arrays.

b)

Parallel Fibonacci Sequence Calculation: The Fibonacci Sequence is a classic recursive algorithm that calculates the nth Fibonacci number using the formula F(n) = F(n-1) + F(n-2), with base cases F(0) = 0 and F(1) = 1. The parallel version of the Fibonacci calculation uses OpenMP to parallelize the recursive calls, dividing the problem into smaller subproblems that can be computed in parallel. This strategy aims to reduce the computation time for large Fibonacci numbers.

**e.**

**Pseudo Code or Program Code:**

**Parallel -Bubble Sort:**

procedure parallelBubbleSort(arr: array of integers)

n = length(arr)

for i from 0 to n - 1

swapped = false

#pragma omp parallel shared(arr, n, swapped)

{

localSwapped = false

#pragma omp for

for j from 0 to n - i - 1

if arr[j] > arr[j + 1]

swap arr[j] and arr[j + 1]

localSwapped = true

end if

end for

#pragma omp critical

{

if localSwapped

swapped = true

end if

}

}

if not swapped

break // If no swaps occurred, the array is already sorted.

end if

end for

end procedure

This pseudo-code outlines the parallel version of Bubble Sort using OpenMP. It divides the work among multiple threads, with each thread working on a portion of the array. The swapped variable is used to check if any swaps occurred in the inner loop, and if no swaps occurred, the algorithm terminates early.

**Serial Bubble Sort:**

procedure bubbleSort(arr: array of integers)

n = length(arr)

for i from 0 to n - 1

for j from 0 to n - i - 1

if arr[j] > arr[j + 1]

swap arr[j] and arr[j + 1]

end if

end for

end for

end procedure

This pseudo-code outlines the same logic as your C program. It iterates through the array and swaps adjacent elements if they are in the wrong order, repeating this process until the array is sorted.

**Parallel Fibonacci Sequence Calculation:**

function parallelFibonacci(n)

if n <= 0

return 0

end if

if n == 1

return 1

end if

declare fib\_n\_minus\_1, fib\_n\_minus\_2

parallel sections

section

fib\_n\_minus\_1 = parallelFibonacci(n - 1)

section

fib\_n\_minus\_2 = parallelFibonacci(n - 2)

end parallel sections

return fib\_n\_minus\_1 + fib\_n\_minus\_2

end function

procedure main()

n = 10

result = parallelFibonacci(n)

print "Fibonacci(", n, ") =", result

end procedure

This pseudo-code outlines the same logic as your C program. It defines a **parallelFibonacci** function that calculates the Fibonacci number at position **n** using recursion and parallel sections. The **main** procedure calls the **parallelFibonacci** function with **n = 10** and prints the result.

**Serial Fibonacci Sequence Calculation:**

function fibonacci(n)

if n <= 0

return 0

end if

if n == 1

return 1

end if

return fibonacci(n - 1) + fibonacci(n - 2)

end function

procedure main()

n = 10

result = fibonacci(n)

print "Fibonacci(", n, ") =", result

end procedure

This pseudo-code outlines the same logic as your C program. It defines a fibonacci function that calculates the Fibonacci number at position n using recursion and a main procedure that calls the fibonacci function with n = 10 and prints the result.

**f.**

**Parallelization Strategy and Decisions:**

a) **Parallel Bubble Sort:**

* We parallelized the inner loop of Bubble Sort using OpenMP to take advantage of multiple processor cores.
* We used the **#pragma omp parallel for** directive to parallelize the loop, and **reduction(||:swapped)** to ensure proper synchronization when checking for sorted arrays.

b) **Parallel Fibonacci Sequence Calculation:**

* We used OpenMP sections (**#pragma omp parallel sections**) to parallelize the recursive calls to **parallelFibonacci(n - 1)** and **parallelFibonacci(n - 2)**.
* Each section of the code calculates a subproblem concurrently, allowing for better utilization of available CPU cores.

**g.**

**Evaluation of the Algorithms:**

I evaluated the performance of the parallel algorithms by measuring the execution time for various input sizes and comparing it to the serial versions of the algorithms. The following timing details were recorded:

1. **Parallel Bubble Sort:**
   * Input array sizes: Varying sizes (e.g., 1000, 5000, 10000 elements)
   * Execution time for each input size using the parallel implementation.
   * Comparison of execution time with the serial Bubble Sort for the same input sizes.
2. **Parallel Fibonacci Sequence Calculation:**
   * Fibonacci numbers to calculate: Varying values of 'n' (e.g., 20, 30, 40)
   * Execution time for each 'n' value using the parallel implementation.
   * Comparison of execution time with the serial Fibonacci calculation for the same 'n' values.

The evaluation results will provide insights into the efficiency gains achieved by parallelizing these algorithms and the impact of parallelization on performance.

**a) Parallelization Strategies Used for Each Implementation**

Parallel Bubble Sort: For the parallel implementation of Bubble Sort, we used the following parallelization strategies:

1. OpenMP Parallelization: OpenMP is a widely-used API for shared-memory parallel programming. We employed OpenMP directives to parallelize the inner loop of Bubble Sort, where elements are compared and swapped. Specifically, we used the #pragma omp parallel for directive to distribute the work of comparing and swapping elements among multiple threads.
2. Reduction Clause: To ensure correctness and synchronization when checking whether any swaps occurred during each pass, we used the reduction(||:swapped) clause in OpenMP. This clause allows multiple threads to safely update a shared variable (swapped) without data races.

Parallel Fibonacci Sequence Calculation: For the parallel implementation of the Fibonacci sequence calculation, we employed these strategies:

1. OpenMP Sections: OpenMP sections were used to divide the problem into smaller, independent subproblems. This allowed us to parallelize the recursive calls to calculate F(n-1) and F(n-2) concurrently.
2. Parallel Recursion: By splitting the computation into sections, we leveraged parallelism at each level of the recursion tree. This strategy aimed to reduce the computation time for large Fibonacci numbers by concurrently calculating subproblems.

**b) Reflection of Parallelization**

The parallelization of both Bubble Sort and Fibonacci Sequence calculation was successful in harnessing the computational power of multi-core processors. However, each algorithm posed different challenges and exhibited different performance characteristics.

Parallel Bubble Sort:

* Reflection: Parallelizing Bubble Sort led to improved performance for sorting large arrays. The parallelization strategy focused on distributing the work of comparing and swapping elements among multiple threads.
* Benefits: Parallelization reduced the time complexity of Bubble Sort from O(n^2) to some extent, resulting in faster sorting for larger datasets.
* Drawbacks: While the parallel version showed improvements, Bubble Sort remains inefficient for very large datasets due to its quadratic time complexity.

Parallel Fibonacci Sequence Calculation:

* Reflection: Parallelizing Fibonacci calculation significantly reduced the execution time for large Fibonacci numbers. The parallelization strategy leveraged parallel recursion to calculate subproblems concurrently.
* Benefits: The parallel version provided substantial speedup for calculating Fibonacci numbers, making it feasible to compute very large Fibonacci numbers within reasonable timeframes.
* Drawbacks: The overhead of managing parallelism can impact performance for smaller Fibonacci numbers, and there is an inherent limitation in memory and thread management for extremely large Fibonacci numbers.

**c) Research Papers on Similar Parallelization Strategies**

Research papers related to the parallelization strategies used in our implementations include:

* Parallel Bubble Sort: Research papers on parallel sorting algorithms, such as parallel versions of Quick Sort and Merge Sort, can be found. These papers explore various techniques for parallelizing sorting algorithms, including parallel partitioning and merging.
* Parallel Fibonacci Calculation: Research papers on parallel recursive algorithms, particularly for Fibonacci calculation, may provide insights. "Parallel Fibonacci Computation: A Comparative Study" by C.-L. Hung and J.-S. Pan is an example of research in this domain.

**d) Limitations of Scalability**

Parallel Bubble Sort:

* Scalability: While parallelization improved performance, the scalability of parallel Bubble Sort is limited. The algorithm's inherent O(n^2) time complexity makes it less suitable for sorting very large datasets. As the dataset size grows, the benefits of parallelization diminish.

Parallel Fibonacci Sequence Calculation:

* Scalability: The parallel Fibonacci calculation performs well for moderately large Fibonacci numbers. However, it may face limitations in scalability for extremely large Fibonacci numbers due to factors like thread management and memory consumption. As 'n' increases, the number of recursive calls and memory requirements also increase, which may lead to performance bottlenecks.

Finally I can say , the effectiveness of parallelization depends on the algorithm's inherent complexity and the nature of the problem it addresses. While parallelization can provide significant performance improvements, it is not a one-size-fits-all solution, and scalability limitations must be considered when applying parallel strategies.

**02. Parallel version**

**i.**

**Parallel version of Bubble Sort using OpenMP:**

#include <stdio.h>

#include <omp.h>

void parallelBubbleSort(int arr[], int n) {

int temp;

int swapped;

for (int i = 0; i < n - 1; i++) {

swapped = 0;

#pragma omp parallel for shared(arr, n) private(temp) reduction(||:swapped)

for (int j = 0; j < n - i - 1; j++) {

if (arr[j] > arr[j + 1]) {

temp = arr[j];

arr[j] = arr[j + 1];

arr[j + 1] = temp;

swapped = 1;

}

}

if (!swapped) break; // If no swaps occurred, the array is already sorted.

}

}

int main() {

int arr[] = {64, 34, 25, 12, 22, 11, 90};

int n = sizeof(arr) / sizeof(arr[0]);

parallelBubbleSort(arr, n);

printf("Sorted array: \n");

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

printf("%d ", arr[i]);

}

printf("\n");

return 0;

}

Note

This parallel version of Bubble Sort uses OpenMP to parallelize the inner loop where elements are compared and swapped.

**ii.**

**Parallel version of Fibonacci Sequence using OpenMP:**

To parallelize the Fibonacci calculation, we can use OpenMP to parallelize the recursive calls. Here's a parallel version:

#include <stdio.h>

#include <omp.h>

int parallelFibonacci(int n) {

if (n <= 0) return 0;

if (n == 1) return 1;

int fib\_n\_minus\_1, fib\_n\_minus\_2;

#pragma omp parallel sections

{

#pragma omp section

fib\_n\_minus\_1 = parallelFibonacci(n - 1);

#pragma omp section

fib\_n\_minus\_2 = parallelFibonacci(n - 2);

}

return fib\_n\_minus\_1 + fib\_n\_minus\_2;

}

int main() {

int n = 10;

int result = parallelFibonacci(n);

printf("Fibonacci(%d) = %d\n", n, result);

return 0;

}

Note

In this parallel version, I use OpenMP sections to parallelize the recursive calls to **parallelFibonacci(n - 1)** and **parallelFibonacci(n - 2)**.