CS 301
High-Performance Computing

Lab 4 - Parallel Time Evaluation of Codes

Rakshit Pandhi (202201426)
Kalp Shah (202201457)

March 20, 2025

Contents

1 Problem A -Integration of a function using Trapezoidal rule.

1.1 Brief description of the problem

In the problem A, we need to integrate a function using the trapezoid rule. Trapezoidal numerical integration is a method to approximate the definite integral of a function by dividing the area under the curve into trapezoids and summing their areas. It is simple, efficient and its accuracy improves with more trapezoids.

1.2 Description of Algorithm

Let $f(x)$ be a continuous function on the interval $[a, b]$. Now divide the interval $[a, b]$ into n equal sub-intervals with each of width,

$$\Delta x = \frac{b-a}{n},$$

such that,

$$a = x_0 < x_1 < x_2 < x_3 < \ldots < x_n = b.$$

Then the Trapezoidal Rule formula for approximating the definite integral $\int_a^b f(x)\,dx$ is given by,

$$\int_a^b f(x)\,dx \approx T_n = \frac{\Delta x}{2}\left[f(x_0) + 2f(x_1) + \ldots + 2f(x_{n-1}) + f(x_n)\right].$$

1.3 Complexity of the Algorithm

1.4 Serial

The number of subintervals used in the calculation for Problem A1, which includes integrating a function using the trapezoidal method, is n, and the serial complexity is O(n). This indicates that the time it takes for the method to finish will grow linearly with n as the number of subintervals does. Because the method only must be run once through the data to calculate the integration, this complexity is considered as linear. An algorithm's serial complexity gives an estimate of how long it will take to execute as the size of the input grows.

1.5 Parallel

- The loop is divided among P threads in a static manner, where each thread processes approximately $\frac{N}{P}$ iterations.
- Each iteration still performs $O(1)$ work (evaluation of $f(x)$ and summation).
- Thus, the per-thread complexity is $O(N/P)$.

Synchronization Overhead

- The `reduction(+:sum,flops)` operation introduces $O(\log P)$ overhead for summing the partial results across threads.

1.6 Compute to Memory Access Ratio

Numerical integration is generally considered compute-bound because the primary limiting factor in its performance is the number of calculations required to evaluate the function at different points across the integration interval, rather than the amount of memory needed to store data; most of the time is spent performing arithmetic operations, not waiting for data to be transferred from memory.

1.7 Memory Bound versus Compute Bound

From calculation we can see that the CMA is nearly 1 , suggesting algorithm spends as much time in computing as in memory access. Hence we can say that this is a memory bound problem.

1.8 Hardware Details

1.8.1 Hardware Details for LAB207 PCs

- Architecture: x86_64
- CPU op-mode(s): 32-bit, 64-bit
- Byte Order: Little Endian
- CPU(s): 12
- On-line CPU(s) list: 0-3
- Thread(s) per core: 2
- Core(s) per socket: 6
- Socket(s): 1
- NUMA node(s): 1
- Vendor ID: GenuineIntel
- CPU family: 6
- Model: 60
- Model name: Intel Core i5-12500
- Stepping: 3
- CPU MHz: 799.992
- CPU max MHz: 4.6G
- CPU min MHz: 800.0000
- BogoMIPS: 6584.55
- Virtualization: VT-x

- L1d cache: 288K
- L1i cache: 192K
- L2 cache: 7.5M
- L3 cache: 18M
- NUMA node0 CPU(s): 0-3

1.8.2 Hardware Details for HPC Cluster

- Architecture: x86_64
- CPU op-mode(s): 32-bit, 64-bit
- Byte Order: Little Endian
- CPU(s): 24
- On-line CPU(s) list: 0-23
- Thread(s) per core: 2
- Core(s) per socket: 6
- Socket(s): 2
- NUMA node(s): 2
- Vendor ID: GenuineIntel
- CPU family: 6
- Model: 63
- Model name: Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz
- Stepping: 2
- CPU MHz: 1419.75
- BogoMIPS: 4804.69
- Virtualization: VT-x
- L1d cache: 32K
- L1i cache: 32K
- L2 cache: 256K
- L3 cache: 15360K
- NUMA node0 CPU(s): 0-5,12-17
- NUMA node1 CPU(s): 6-11,18-23

1.9 Input Parameters,Output,Accuracy

Input Parameters:

- $f(x)$: The function to be integrated.
- a: The lower limit of integration.
- b: The upper limit of integration.
- n: The number of sub-intervals.

Output

- The final answer after the integration.

Accuracy

- Accuracy is proportional to the number of computations performed.
- Therefore for smaller n accuracy may come low and vice versa.
- We can measure the accuracy by calculating the error if we known with the exact value of the integral.

$$\text{Error} = |T_n - I|$$

1.10 Algorithm Time versus Problem Size

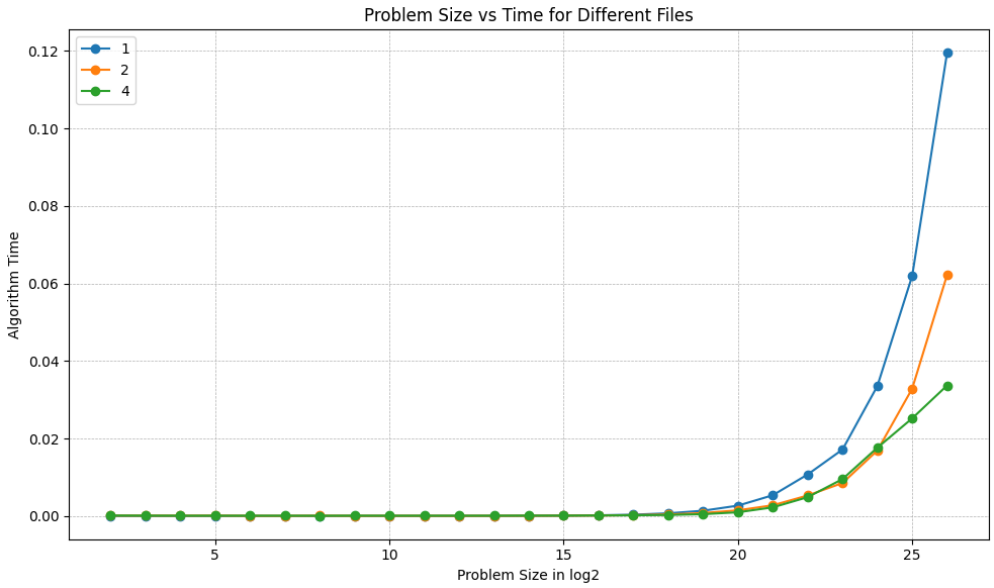

Figure 1: Screenshot from LAB207 PC

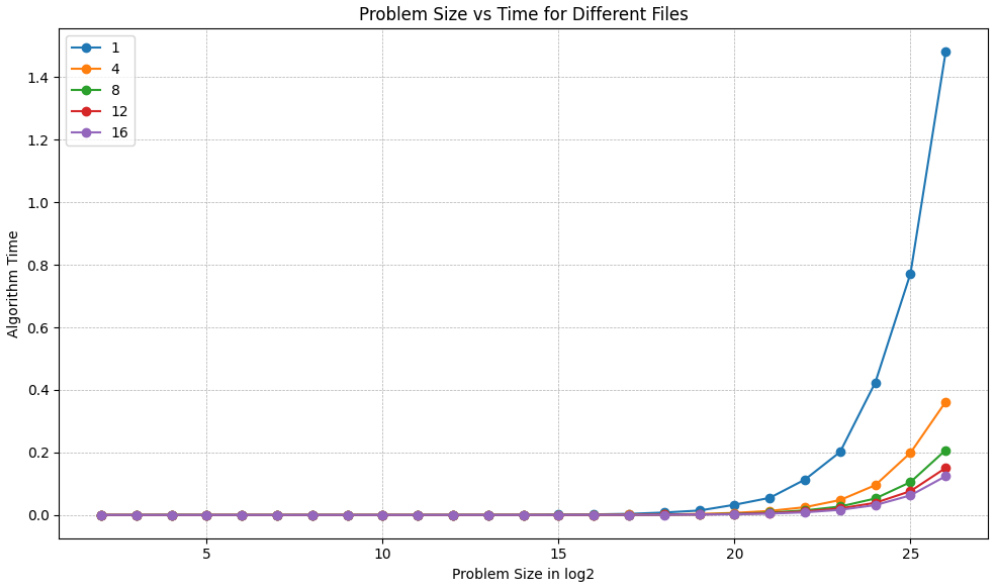

Figure 2: Screenshot from HPC Cluster

1.11 Speedup

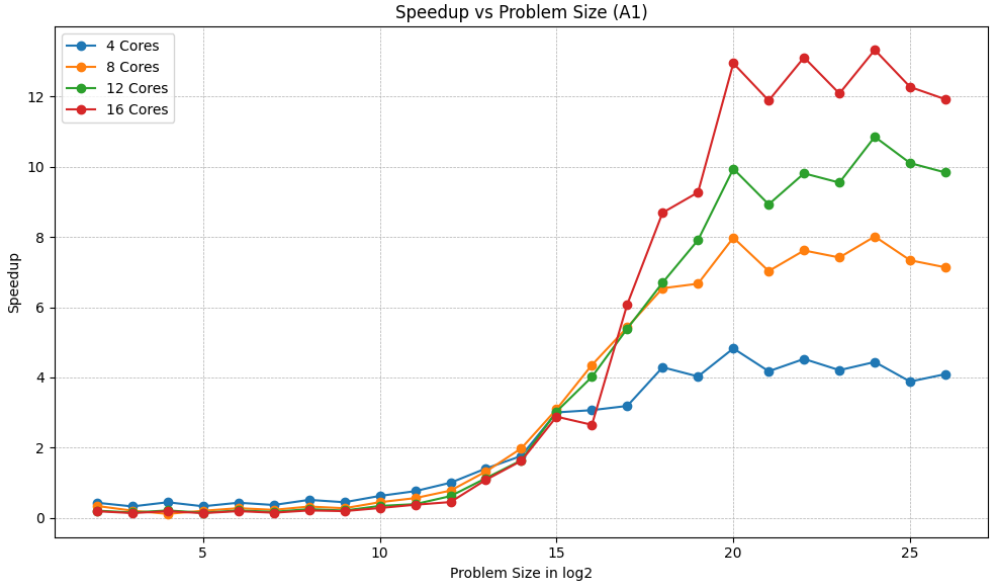

Figure 3: Screenshot from HPC Cluster

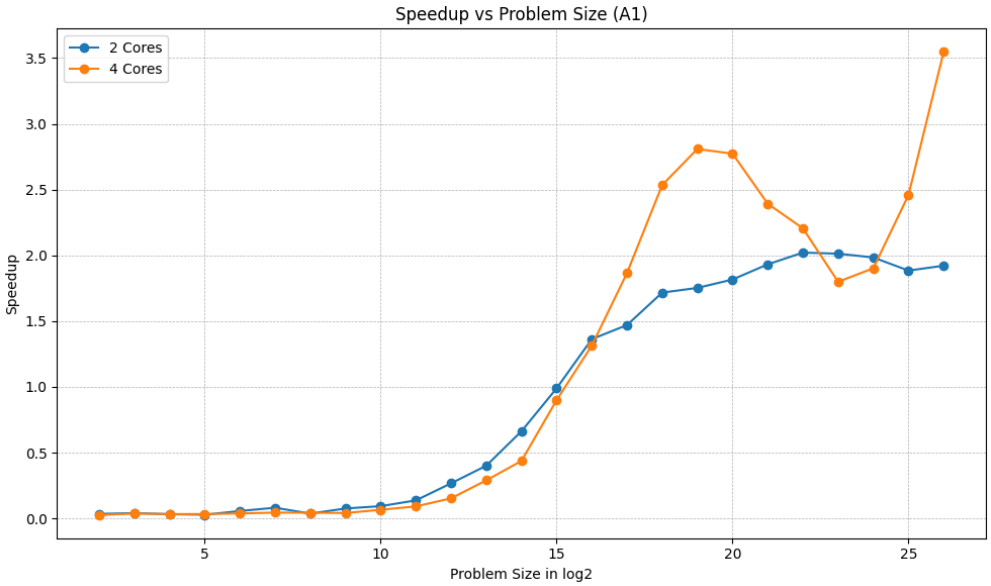

Figure 4: Screenshot from LAB207 PC

Theoretical speedup should be 1/number of processors. But it is not seen practically because of parallel overheads and all other reasons.

1.12 Inference

HPC Cluster and LAB207 PC takes almost same time up to size 20, for running the code due to lack of parallel algorithm. After that as the size further increases as the cache size of Lab PC is greater than HPC Cluster it takes relatively lesser time than the HPC Cluster.

2 Problem B -Summation of 2 vectors and addition of constant [A(i)+B(i)] + k

2.1 Brief description of the problem

The problem involves computing the element-wise summation of two vectors A and B, followed by adding a constant k to each resulting element. Mathematically, this can be expressed as:

$$C(i) = A(i) + B(i) + k$$

where i is the index of the vector elements.

2.2 Description of Algorithm

- **Initialize**: Vectors A, B, and constant k.
- **Compute**: For each index i, calculate $C(i) = A(i) + B(i) + k$.
- **Output**: Return vector C.

2.3 Complexity of the Algorithm

2.4 Serial

The time complexity of this algorithm is $O(N)$, where N is the length of the vectors. This is because the algorithm performs a constant amount of work—addition and assignment—for each element in the vectors. Since the loop iterates exactly N times, the total time taken grows linearly with the size of the input. The space complexity is also $O(N)$ because a new vector C of length N is created to store the results.

2.5 Parallel

The loop iterations are distributed across P threads. Each thread approximately handles $\frac{N}{P}$ iterations. Each iteration performs $O(1)$ work.

Thus, the per-thread complexity is:

$$O\left(\frac{N}{P}\right)$$

Synchronization Overhead

Since each thread independently computes a subset of elements without dependencies, there is no need for reduction or explicit synchronization. However, thread creation, scheduling, and memory access contention introduce an overhead of:

$$O(\log P)$$

Total Parallel Complexity

$$O\left(\frac{N}{P} + \log P\right)$$

2.6 Compute to Memory Access Ratio

In one iteration there are 2 flop i.e. 2 additions. And per iteration there are 3 memory access 1 for A(read) ,1 for B(read) and 1 for C(write), assuming reading constant k does not take time. So for a random N total flops would be 2N and memory accesses would be 3N giving CMA of

$$\frac{2}{3}$$

.

2.7 Memory Bound versus Compute Bound

This problem is memory-bound because the number of memory accesses is relatively high compared to the number of computations.

2.8 Hardware Details

2.8.1 Hardware Details for LAB207 PCs

- Architecture: x86_64
- CPU op-mode(s): 32-bit, 64-bit
- Byte Order: Little Endian
- CPU(s): 12
- On-line CPU(s) list: 0-3
- Thread(s) per core: 2
- Core(s) per socket: 6
- Socket(s): 1
- NUMA node(s): 1
- Vendor ID: GenuineIntel
- CPU family: 6
- Model: 60
- Model name: Intel Core i5-12500
- Stepping: 3
- CPU MHz: 799.992
- CPU max MHz: 4.6G
- CPU min MHz: 800.0000
- BogoMIPS: 6584.55
- Virtualization: VT-x
- L1d cache: 288K
- L1i cache: 192K
- L2 cache: 7.5M
- L3 cache: 18M
- NUMA node0 CPU(s): 0-3

2.8.2 Hardware Details for HPC Cluster

- Architecture: x86_64
- CPU op-mode(s): 32-bit, 64-bit
- Byte Order: Little Endian
- CPU(s): 24
- On-line CPU(s) list: 0-23
- Thread(s) per core: 2
- Core(s) per socket: 6
- Socket(s): 2
- NUMA node(s): 2
- Vendor ID: GenuineIntel
- CPU family: 6
- Model: 63
- Model name: Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz
- Stepping: 2
- CPU MHz: 1419.75
- BogoMIPS: 4804.69
- Virtualization: VT-x
- L1d cache: 32K
- L1i cache: 32K
- L2 cache: 256K
- L3 cache: 15360K
- NUMA node0 CPU(s): 0-5,12-17
- NUMA node1 CPU(s): 6-11,18-23

2.9 Input Parameters,Output,Accuracy

Input Parameters

- **Vector** A: Array of numbers of length N. Example: $A = [a_1, a_2, \ldots, a_N]$.
- **Vector** B: Array of numbers of length N. Example: $B = [b_1, b_2, \ldots, b_N]$.
- **Constant** k: A scalar value added to each element after summation.

Output

- **Vector** C: Resultant vector of length N where each element is calculated as:

$$C(i) = A(i) + B(i) + k$$

  Example Output: $C = [c_1, c_2, \ldots, c_N]$.

Accuracy Check

- **C**ompare the output against expected results for known inputs.
- **E**nsure accuracy by using a tolerance level:

$$|C(i) - (A(i) + B(i) + k)| < \epsilon$$

  where ϵ is a small value, e.g., 10^{-9}.

2.10 Algorithm Time versus Problem Size

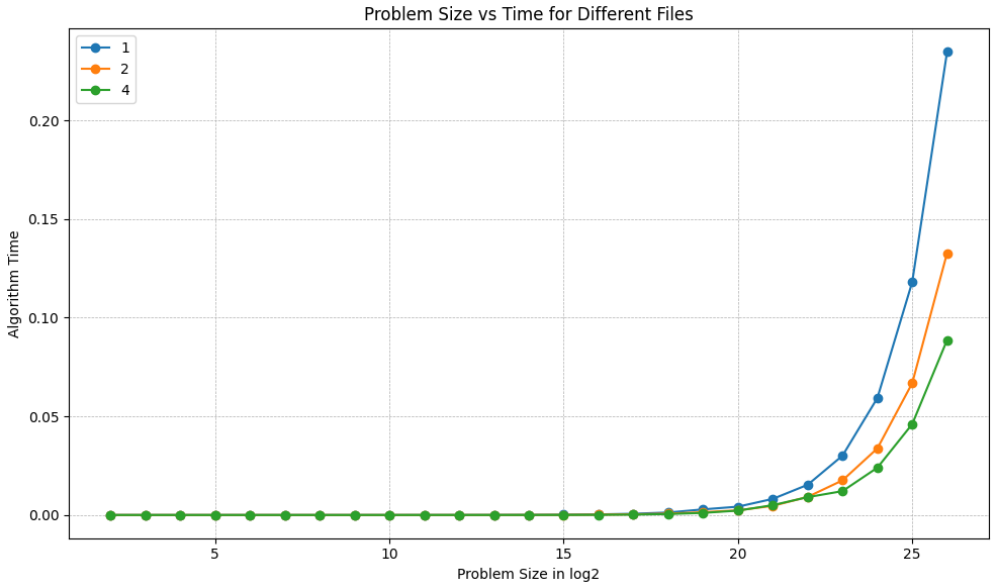

Figure 5: Screenshot from LAB PC

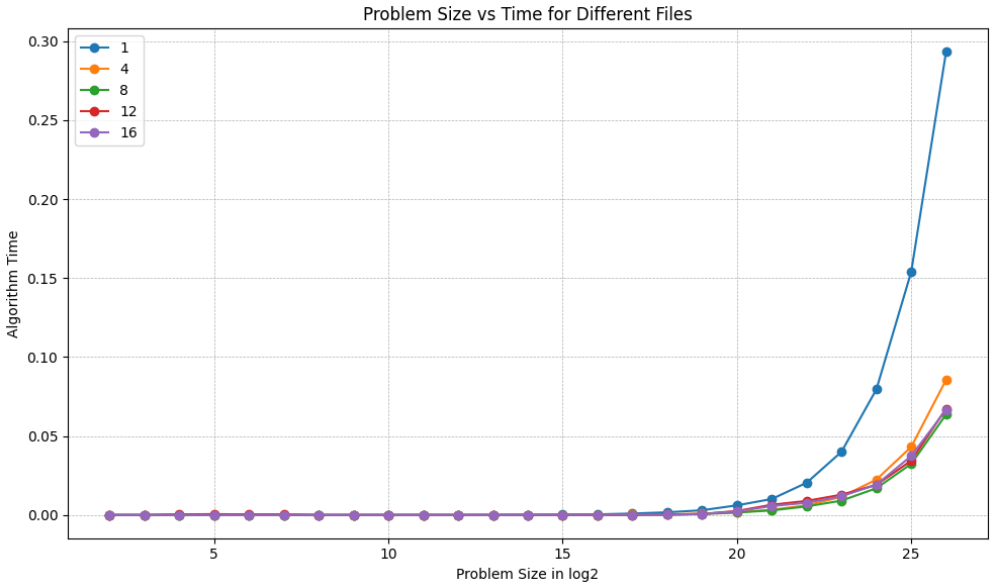

Figure 6: Screenshot from HPC Cluster

2.11 Speedup

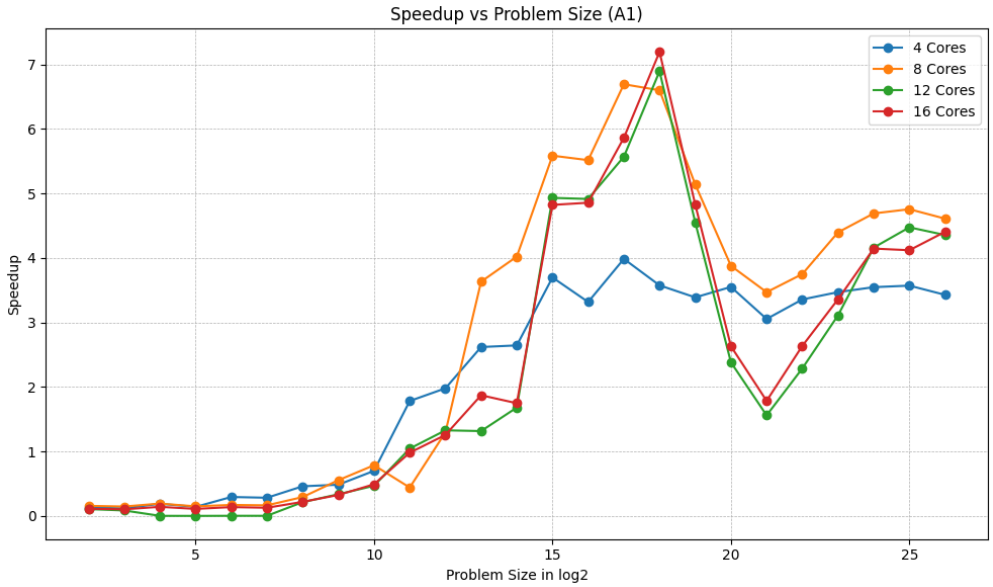

Figure 7: Screenshot from HPC Cluster

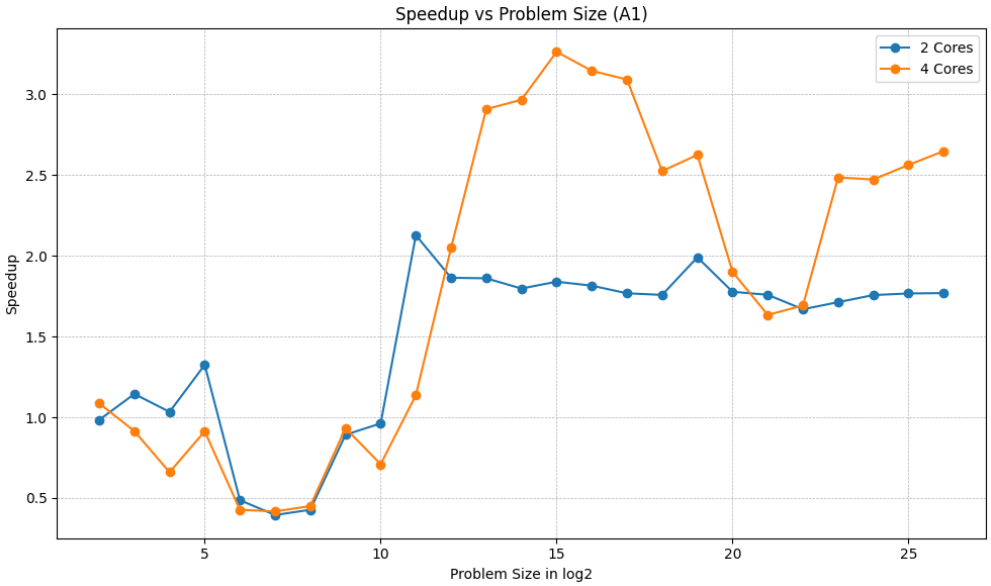

Figure 8: Screenshot from LAB207 PC

Theoretical speedup should be 1/number of processors. But it is not seen practically because of parallel overheads and all other reasons.

2.12 Inference

HPC Cluster and LAB207 PC takes almost same time up to size 20, for running the code due to lack of parallel algorithm. After that as the size further increases as the cache size of Lab PC is greater than HPC Cluster it takes relatively lesser time than the HPC Cluster.

3 Problem C-Sorting - Quick Sort

3.1 Brief description of the problem

Quick Sort is a comparison-based sorting algorithm that uses the divide-and-conquer approach. It selects a "pivot" element from the array and partitions the other elements into two groups: those less than the pivot and those greater. The sub-arrays are then sorted recursively. This process continues until the entire array is sorted.

3.2 Description of Algorithm

1. **Choose Pivot:** Select a pivot element from the array. This can be the first element, the last element, a random element, or the median.

2. **Partitioning:** Rearrange the array such that all elements less than the pivot are on its left, and all greater elements are on its right.

3. **Recursive Sort:** Recursively apply the same process to the left and right sub-arrays.

4. **Base Case:** Recursion ends when the sub-array has fewer than two elements, as they are already sorted.

3.3 Complexity of the Algorithm

3.4 Serial

The average time complexity of Quick Sort is:

$$O(n \log n)$$

However, in the worst case (e.g., when the pivot is the smallest or largest element), the time complexity can degrade to:

$$O(n^2)$$

3.5 Parallel

Synchronization Overhead

The task parallelism introduced using `#pragma omp task` incurs overhead due to:

- **Task creation and scheduling**: Each recursive function call generates new tasks, leading to additional overhead.
- **Dependency management**: Since recursive calls depend on partitioning results, an implicit barrier exists at the end of the parallel region.
- **Load balancing issues**: If partitions are unbalanced, some threads finish earlier than others, causing inefficiencies. This introduces a synchronization overhead of $O(\log P)$.

Total Parallel Complexity

QuickSort has a sequential complexity of $O(N \log N)$. When parallelized using P threads, the total complexity is given by:

$$O\left(\frac{N}{P} \log N + \log P\right)$$

where:

- $O(N/P \log N)$ represents the partitioning work distributed across P processors.
- $O(\log P)$ accounts for task scheduling and thread synchronization overhead.

3.6 Compute to Memory Access Ratio

On average, the number of memory accesses is roughly proportional to the number of compute operations.

In QuickSort, the ratio is roughly balanced, but the algorithm tends to be compute-bound because:

The number of comparisons and swaps grows with the input size.

Memory accesses are typically sequential or localized, making them relatively efficient (especially with caching).

3.7 Memory Bound versus Compute Bound

QuickSort is compute-bound because its performance is primarily determined by the number of comparisons and swaps, which are CPU-intensive operations. Memory usage is minimal and does not typically become the limiting factor as it is an in place sorting algorithm.

3.8 Hardware Details

3.8.1 Hardware Details for LAB207 PCs

- Architecture: x86_64
- CPU op-mode(s): 32-bit, 64-bit
- Byte Order: Little Endian
- CPU(s): 12
- On-line CPU(s) list: 0-3
- Thread(s) per core: 2
- Core(s) per socket: 6
- Socket(s): 1
- NUMA node(s): 1
- Vendor ID: GenuineIntel
- CPU family: 6
- Model: 60
- Model name: Intel Core i5-12500
- Stepping: 3
- CPU MHz: 799.992
- CPU max MHz: 4.6G
- CPU min MHz: 800.0000
- BogoMIPS: 6584.55
- Virtualization: VT-x
- L1d cache: 288K
- L1i cache: 192K
- L2 cache: 7.5M
- L3 cache: 18M
- NUMA node0 CPU(s): 0-3

3.8.2 Hardware Details for HPC Cluster

- Architecture: x86_64
- CPU op-mode(s): 32-bit, 64-bit
- Byte Order: Little Endian
- CPU(s): 24
- On-line CPU(s) list: 0-23
- Thread(s) per core: 2
- Core(s) per socket: 6
- Socket(s): 2
- NUMA node(s): 2
- Vendor ID: GenuineIntel
- CPU family: 6
- Model: 63
- Model name: Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz
- Stepping: 2
- CPU MHz: 1419.75
- BogoMIPS: 4804.69
- Virtualization: VT-x
- L1d cache: 32K
- L1i cache: 32K
- L2 cache: 256K
- L3 cache: 15360K
- NUMA node0 CPU(s): 0-5,12-17
- NUMA node1 CPU(s): 6-11,18-23

3.9 Input Parameters,Output,Accuracy

Input Parameters

- A: Array of n elements to be sorted
- l: Low index, indicating the starting position of the array or subarray
- h: High index, indicating the ending position of the array or subarray

Output

- A': Sorted array in non-decreasing order

Accuracy The output array satisfies the following conditions:

1. $A'[i] \leq A'[i+1],\ \forall i$
2. A' is a permutation of A
3. Relative order of identical elements is preserved (if stable)

3.10 Algorithm Time versus Problem Size

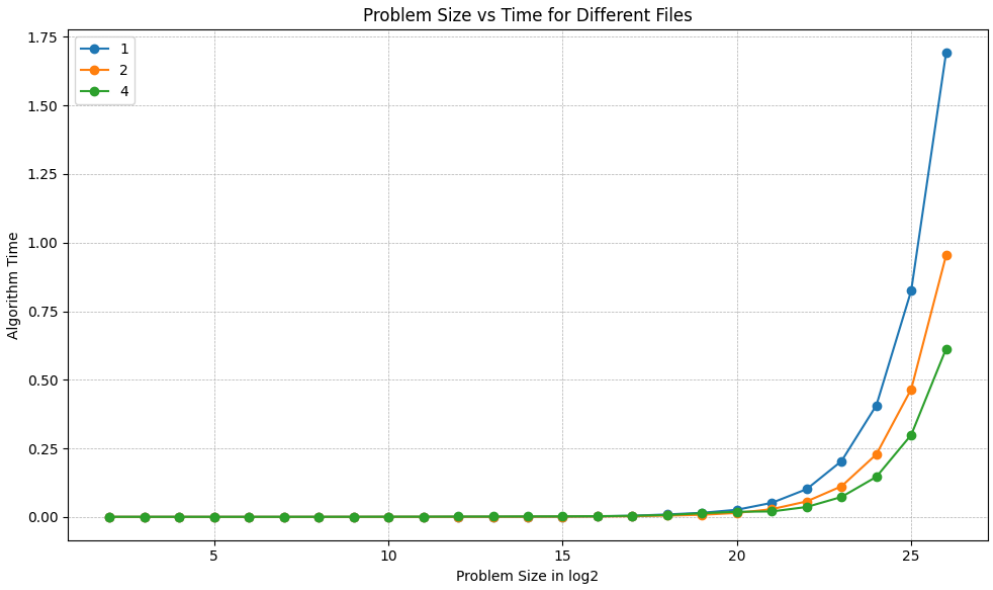

Figure 9: Screenshot from LAB PC

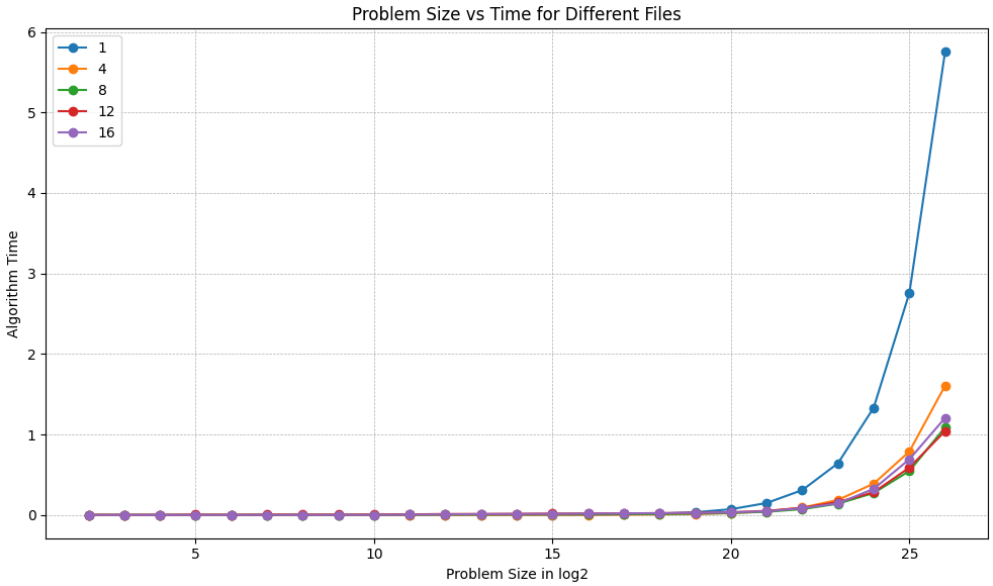

Figure 10: Screenshot from HPC Cluster

3.11 Speedup

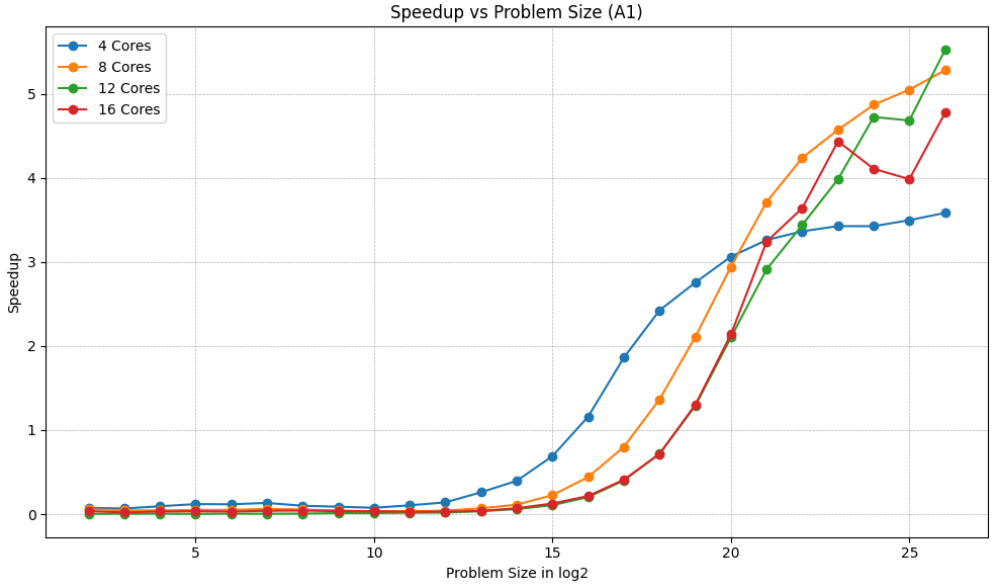

Figure 11: Screenshot from HPC Cluster

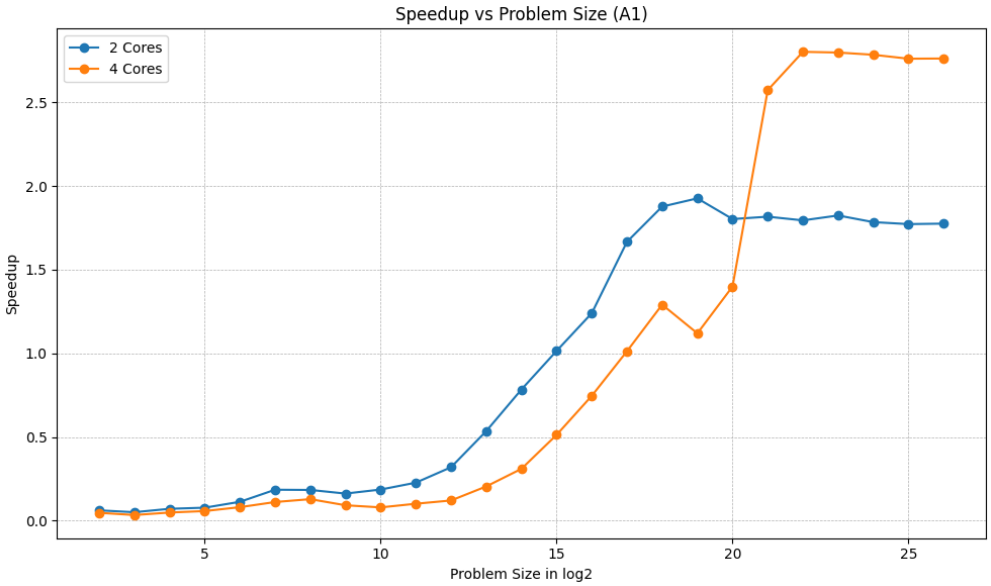

Figure 12: Screenshot from LAB207 PC

Theoretical speedup should be 1/number of processors. But it is not seen practically because of parallel overheads and all other reasons.

3.12 Inference

Firstly the plot of the HPC Cluster is for 2 runs only as it takes too much time for the computation. The plot for the LAB PC is for 5 runs. For this problem the maxsize has been taken 25 in order to reduce the wait time and miss in caches.

Therefore, we can see that even though there are less runs in case of the cluster still it is slower as compared to LAB PC because of compute bound and small cache size of the cluster as compared to the LAB PC.

4 Conclusions

- We can conclude that Quicksort and Numerical Integration are relatively a compute bound problem, whereas vector addition in memory bound.
- Also commenting the algorithm time versus problem size we can say that the HPC Cluster more or less gives the same plot as the LAB PC, just that because of higher cache size LAB PC outeperforms slightly.
- OpenMP enables efficient parallelization by automatically distributing workload across multiple threads, allowing computations to be executed concurrently rather than sequentially. This significantly reduces execution time by utilizing the full potential of multi-core processors.
- Commands like `#pragma omp parallel for` ensure that loop iterations are divided among threads, while `#pragma omp task` allows independent tasks to execute in parallel, enhancing

performance without requiring complex manual thread management. Additionally, OpenMP provides built-in synchronization mechanisms such as `#pragma omp reduction`, which ensures safe aggregation of results while minimizing bottlenecks.