CS 301
High-Performance Computing

Lab 2 - Performance evaluation of codes

Problem C2: Sorting - Merge Sort.

Jay Dobariya (202101521)
Akshar Panchani (202101522)

January 31, 2024

Contents

1 Introduction

We would use the merge sort strategy to sort this code performance in order to determine its complexity and its specific high performance computing perspective. The article goes on to provide a case study and mathematical concepts.

2 Hardware Details

2.1 Hardware Details for LAB207 PCs

- Architecture: x86_64
- CPU op-mode(s): 32-bit, 64-bit
- Byte Order: Little Endian
- CPU(s): 6
- On-line CPU(s) list: 0-5
- Thread(s) per core: 1
- Core(s) per socket: 6
- Socket(s): 1
- NUMA node(s): 1
- Vendor ID: GenuineIntel
- CPU family: 6
- Model: 155
- Model name: Intel(R) Core(TM) i5-8500 CPU @ 3.00GHz
- Stepping: 10
- CPU MHz: 799.992
- CPU max MHz: 4100.0000
- CPU min MHz: 800.0000
- BogoMIPS: 6000.00
- Virtualization: VT-x
- L1d cache: 192KB
- L1i cache: 192KB
- L2 cache: 1.5MB
- L3 cache: 9MB
- NUMA node0 CPU(s): 0-5

2.2 Hardware Details for HPC Cluster (Node gics1)

- 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: 2642.4378
- 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 Problem C2

3.1 Brief description with its algorithmic approach

In this work, the performance code is analyzed using the merge sort approach. The objective is to analyze its efficiency and temporal complexity on platforms for high-performance computing (HPC). When it comes to parallel computing, merge sort offers a methodical way to comprehend the behavior and scalability of the code.

1. **Divide:** If the array has one or zero elements, it is already sorted. Otherwise, divide the array into two halves.
2. **Conquer:** Recursively apply merge sort to each half of the array.
3. **Merge:** Merge the two sorted halves to produce a single sorted array. This involves comparing elements from each half and combining them in sorted order.

3.2 The complexity of the algorithm (serial)

To implement the merge sort method, two sorted arrays can be merged using a helper function. Merge sort has a $O(N \log N)$ time complexity, where N is the array's element count.

3.3 Profiling using HPC Cluster (with gprof)

The screenshots of profiling using the HPC Cluster are given below

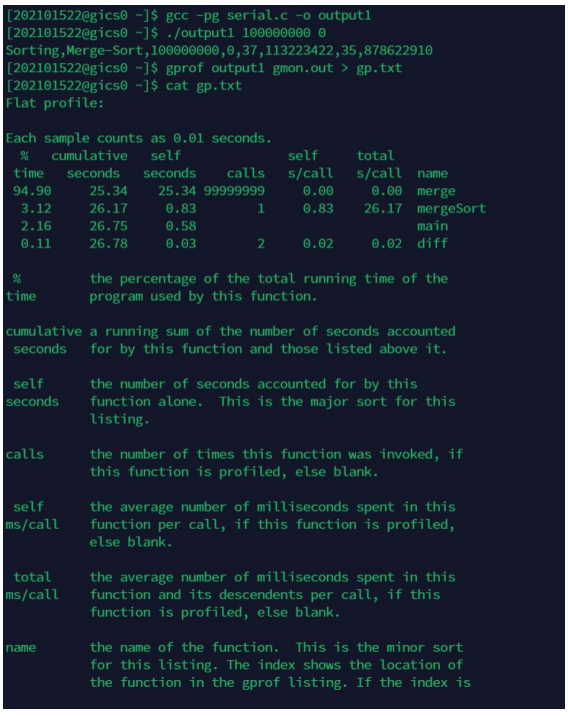

Figure 1: Screenshot of terminal from HPC Cluster

3.4 Graph of Problem Size vs Runtime

3.4.1 Graph of Problem Size vs Runtime for LAB207 PCs

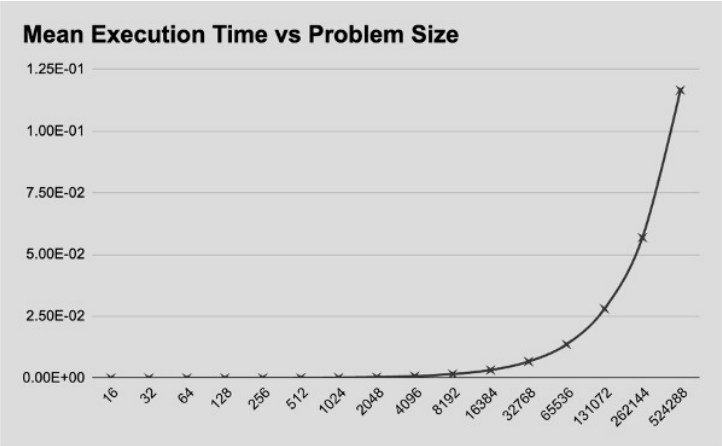

Figure 2: Total mean execution time (Algorithm time) vs Problem size plot for **problem size 10^6 (Hardware: LAB207 PC, Problem: MERGE_SORT)**.

3.4.2 Graph of Problem Size vs Runtime for HPC Cluster

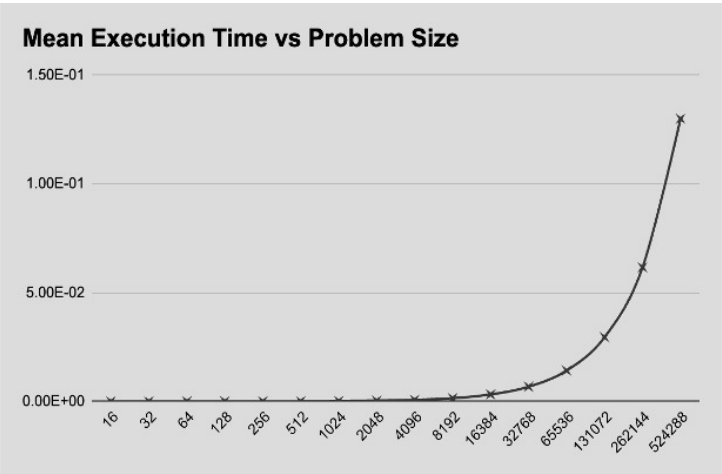

Figure 3: Total mean execution time (Algorithm time) vs Problem size plot for **problem size 10^6 (Hardware: HPC Cluster, Problem: MERGE_SORT)**.

4 Conclusions

- The length of time required to perform Merge Sort rises exponentially with the complexity of the issue for both the LAB PC and the cluster systems.
- There is a trade-off between the size of the issue and the amount of time required to execute the algorithm; for smaller problems, the method's time consumption makes up a lower fraction of the overall time; but, as the problem gets larger, the algorithm's contribution to the total time grows.