CS 301
High-Performance Computing

Lab 4 - Performance evaluation of codes

Problem C2: Sorting - Merge Sort.

Jay Dobariya (202101521)
Akshar Panchani (202101522)

February 28, 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 Information about parallel implementation

The provided code implements parallel merge sort using OpenMP directives. Here's a brief explanation of the parallel implementation:

- The program takes two command-line arguments: n (size of the input array) and p (number of processors).
- It initializes an array of size n with random values and performs parallel merge sort on it.
- The merge sort algorithm is parallelized using OpenMP directives. The recursive calls to merge sort for the left and right halves of the array are executed concurrently in parallel sections.
- The number of threads (p) determines the level of parallelism, with each thread executing a subset of the merge sort operations.
- The result is a sorted array in ascending order.

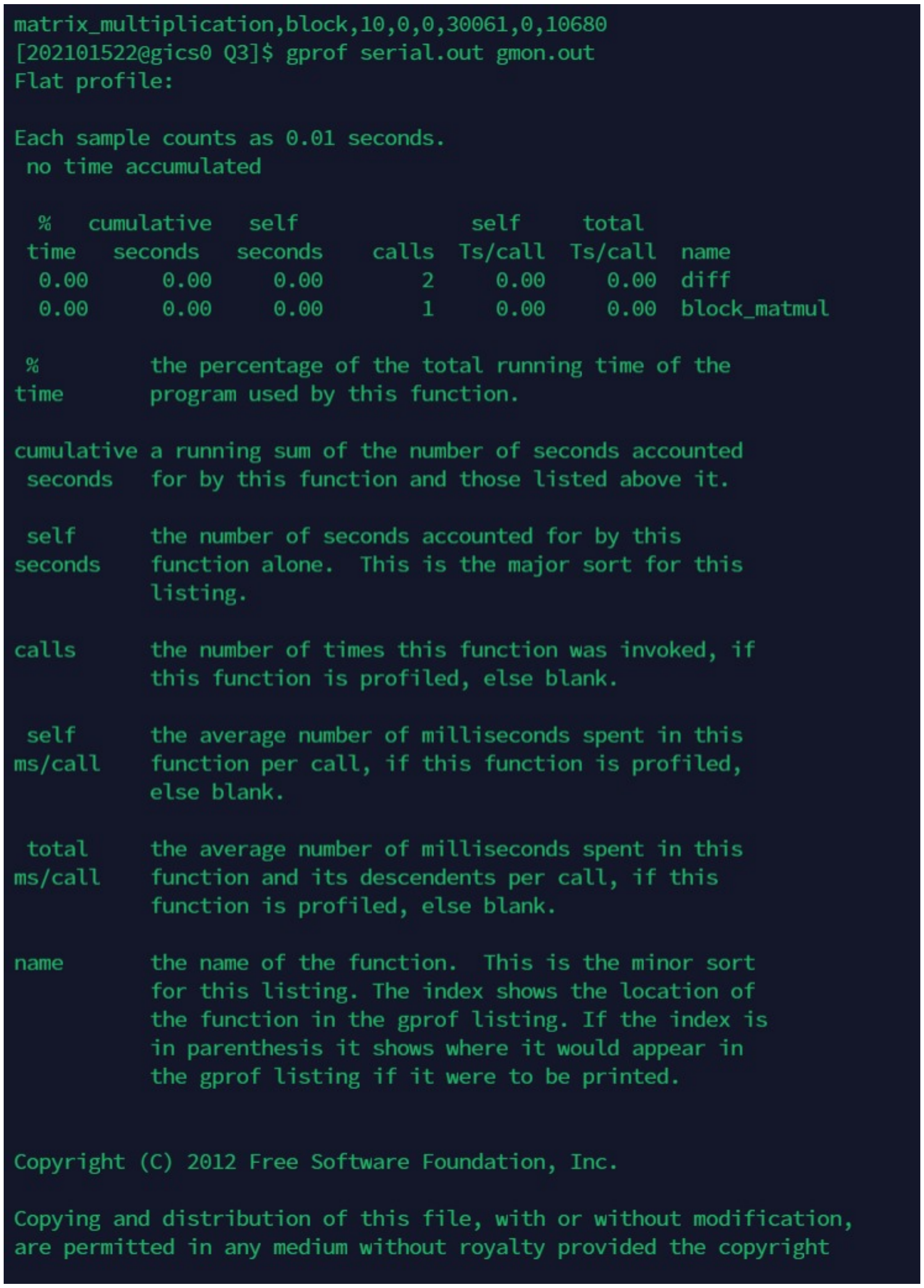
3.4 The complexity of the algorithm (Parallel)

The complexity of the algorithm can be analyzed as follows:

- **Parallelism:** The parallel implementation of merge sort uses OpenMP to execute the recursive calls in parallel sections, allowing for concurrent execution on multiple CPU cores. The number of threads (p) determines the level of parallelism, with each thread executing a subset of the merge sort operations.

3.5 Profiling using HPC Cluster (with gprof)

The screenshots of profiling using the HPC Cluster are given below



```
matrix_multiplication,block,10,0,0,30061,0,10680
[202101522@gics0 Q3]$ gprof serial.out gmon.out
Flat profile:

Each sample counts as 0.01 seconds.
 no time accumulated

  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
  0.00      0.00     0.00        2     0.00     0.00  diff
  0.00      0.00     0.00        1     0.00     0.00  block_matmul

 %         the percentage of the total running time of the
time       program used by this function.

cumulative a running sum of the number of seconds accounted
 seconds   for by this function and those listed above it.

 self      the number of seconds accounted for by this
seconds    function alone.  This is the major sort for this
           listing.

calls      the number of times this function was invoked, if
           this function is profiled, else blank.

 self      the average number of milliseconds spent in this
ms/call    function per call, if this function is profiled,
           else blank.

 total     the average number of milliseconds spent in this
ms/call    function and its descendents per call, if this
           function is profiled, else blank.

name       the name of the function.  This is the minor sort
           for this listing. The index shows the location of
           the function in the gprof listing. If the index is
           in parenthesis it shows where it would appear in
           the gprof listing if it were to be printed.


Copyright (C) 2012 Free Software Foundation, Inc.

Copying and distribution of this file, with or without modification,
are permitted in any medium without royalty provided the copyright
```

Figure 1: Screenshot of terminal from HPC Cluster

3.6 Graph of Problem Size vs Runtime

3.6.1 Graph of Problem Size vs Total-time for LAB207 PCs

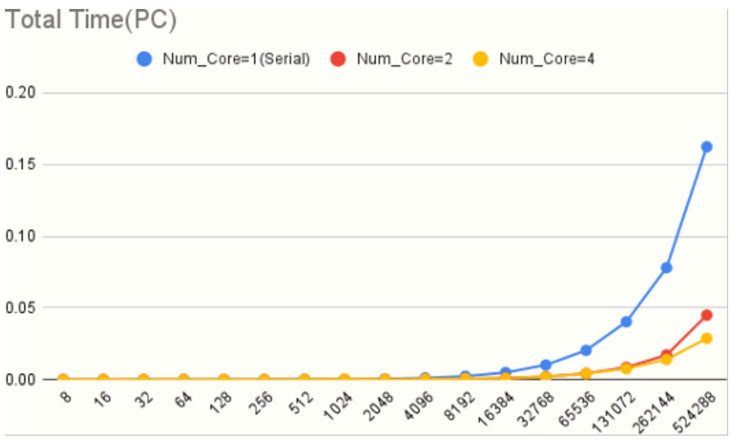


Figure 2: Total mean execution time (Total time) vs Problem size plot for **problem size 10^6 (Hardware: LAB207 PC, Problem: MERGE_SORT)**.

3.6.2 Graph of Problem Size vs Algorithm-time for LAB207 PCs

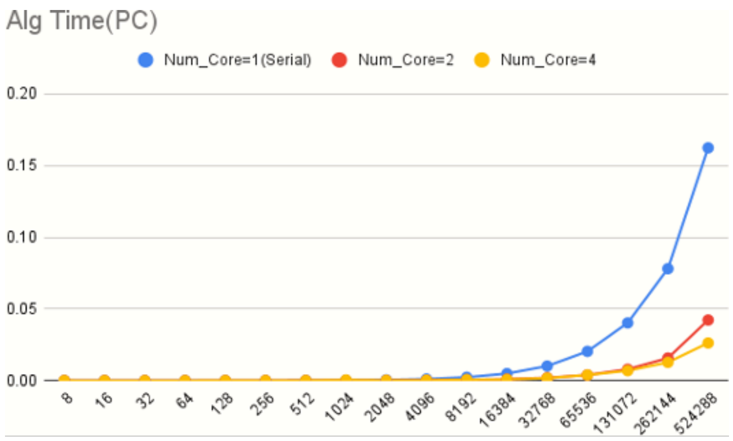


Figure 3: Total mean execution time (Algorithm time) vs Problem size plot for **problem size 10^6 (Hardware: LAB207 PC, Problem: MERGE_SORT)**.

3.6.3 Graph of Problem Size vs Total-time for HPC Cluster

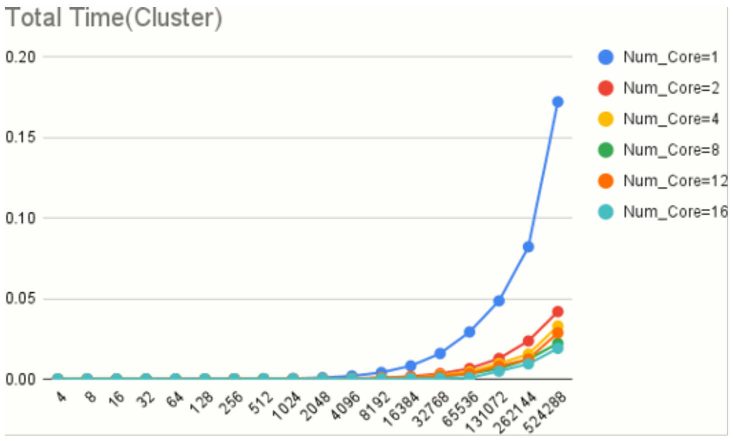


Figure 4: Total mean execution time (Total time) vs Problem size plot for **problem size 10^6 (Hardware: HPC Cluster, Problem: MERGE_SORT)**.

3.6.4 Graph of Problem Size vs Algorithm-time for HPC Cluster

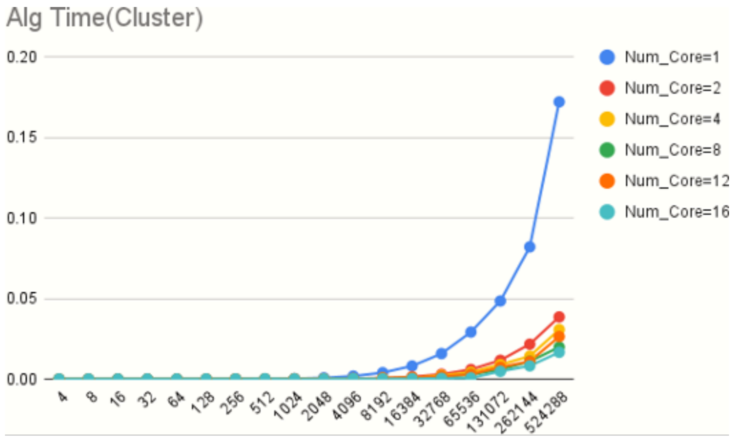


Figure 5: Total mean execution time (Algorithm time) vs Problem size plot for **problem size 10^6 (Hardware: HPC Cluster, Problem: MERGE_SORT)**.

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.

- Overall, the parallel implementation improves the performance of merge sort by leveraging multiple threads to sort the array concurrently.