CS 301
High-Performance Computing

Lab 4 - C1

Problem C-1: QUICK_SORT

Aditya Nawal (202001402)
Divya Patel (202001420)

March 15, 2023

Contents

1 Introduction

Quick Sort is a widely used sorting algorithm that can be implemented both serially and in parallel. In this report, we present a comparison between the serial and parallel implementations of Quick Sort. We analyze their performance and discuss their advantages and disadvantages.

Quick Sort is an efficient sorting algorithm that uses the divide-and-conquer approach. It works by selecting a pivot element from the array and partitioning the other elements into two sub-arrays according to whether they are less than or greater than the pivot. The pivot is then placed in its final position in the sorted array, and the two sub-arrays are recursively sorted.

2 Hardware Details

2.1 Lab 207 PC

- Architecture: x86_64
- CPU op-mode(s): 32-bit, 64-bit
- Byte Order: Little Endian
- CPU(s): 4
- On-line CPU(s) list: 0-3
- Thread(s) per core: 1
- Core(s) per socket: 4
- Socket(s): 1
- NUMA node(s): 1
- Vendor ID: GenuineIntel
- CPU family: 6
- Model: 60
- Model name: Intel(R) Core(TM) i5-4590 CPU @ 3.30GHz
- Stepping: 3
- CPU MHz: 3300.000
- CPU max MHz: 3700.0000
- CPU min MHz: 800.0000
- BogoMIPS: 6585.38
- Virtualization: VT-x
- L1d cache: 32K

- L1i cache: 32K
- L2 cache: 256K
- L3 cache: 6144K
- NUMA node0 CPU(s): 0-3
- Flags: fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc aperfmperf eagerfpu pni pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3 fma cx16 xtpr pdcm pcid sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm abm epb invpcid_single tpr_shadow vnmi flexpriority ept vpid fsgsbase tsc_adjust bmi1 avx2 smep bmi2 erms invpcid xsaveopt dtherm ida arat pln pts

```
[student@localhost ~]$ ifconfig
lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1  (Local Loopback)
        RX packets 60  bytes 5868 (5.7 KiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 60  bytes 5868 (5.7 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

p4p1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.100.64.86  netmask 255.255.255.0  broadcast 10.100.64.255
        inet6 fe80::b283:feff:fe97:d2f9  prefixlen 64  scopeid 0x20<link>
        ether b0:83:fe:97:d2:f9  txqueuelen 1000  (Ethernet)
        RX packets 32826  bytes 46075919 (43.9 MiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 8015  bytes 586362 (572.6 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

virbr0: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
        inet 192.168.122.1  netmask 255.255.255.0  broadcast 192.168.122.255
        ether 52:54:00:3a:16:71  txqueuelen 1000  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

Figure 1: IP address of Lab PC

2.2 HPC Cluster

- Architecture: x86_64
- CPU op-mode(s): 32-bit, 64-bit
- Byte Order: Little Endian
- CPU(s): 16
- On-line CPU(s) list: 0-15
- Thread(s) per core: 1
- Core(s) per socket: 8

- Socket(s): 2
- NUMA node(s): 2
- Vendor ID: GenuineIntel
- CPU family: 6
- Model: 63
- Model name: Intel(R) Xeon(R) CPU E5-2640 v3 @ 2.60GHz
- Stepping: 2
- CPU MHz: 1976.914
- BogoMIPS: 5205.04
- Virtualization: VT-x
- L1d cache: 32K
- L1i cache: 32K
- L2 cache: 256K
- L3 cache: 20480K
- NUMA node0 CPU(s): 0-7
- NUMA node1 CPU(s): 8-15

3 Problem C1

3.1 Description of the problem

The problem at hand involves parallelizing the quicksort algorithm using OpenMP. Quicksort is a well-known sorting algorithm that can be parallelized to improve its performance. By using OpenMP directives and library routines, we can parallelize the quicksort algorithm to take advantage of multiple processors and improve its run-time performance.

3.2 Serial Complexity

The time complexity of the serial quick-sort algorithm is $O(n \log n)$ on average and $O(n^2)$ in the worst case, where n is the size of the input array. In this algorithm, a pivot element is chosen and the array is partitioned such that elements less than the pivot are placed to its left and elements greater than the pivot are placed to its right. This partitioning step has a time complexity of $O(n)$. The algorithm then recursively sorts the sub-arrays to the left and right of the pivot.

3.3 Parallel Complexity

In a parallel version of quick-sort, multiple threads can be used to concurrently sort different sub-arrays. In the given code snippet, this is achieved using OpenMP task constructs. The partitioning step remains unchanged and has a time complexity of $O(n)$. However, after partitioning, two tasks are created using #pragma omp task directives to recursively sort the left and right sub-arrays concurrently.

The parallel time complexity of this implementation depends on several factors such as the number of threads used (p), load balancing between threads, and overhead associated with creating and managing tasks. In an ideal scenario with perfect load balancing and negligible overhead, each thread would sort $\frac{n}{p}$ elements in $O(\frac{n}{p} \log \frac{n}{p})$ time. Since all threads work concurrently, this would result in a parallel time complexity of $O(\frac{n \log n}{p})$. However, in practice it may be higher due to non-ideal conditions such as imbalanced loads between threads or overhead associated with creating and managing tasks.

3.4 Profiling Information

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ns/call  ns/call  name
 88.90      6.13     6.13 12283285   499.41   499.41  diff
 11.49      6.93     0.79                             main

 %         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.
```

Figure 2: Screenshot of text file generated from profiling on Lab 207 PC using gprof

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 88.30     19.27    19.27  2920292     6.60     6.60  diff
 12.03     21.89     2.63                             main

 %         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.
```

Figure 3: Profiling on HPC cluster using gprof

3.5 Optimization Strategy

We used median of the array as the pivot element, which results in a balanced partition and minimizes the number of comparisons.

In the given code snippet, OpenMP is used to parallelize the recursive calls to quick_sort_parallel that sort different sub-arrays. This is achieved using OpenMP task constructs.

After partitioning the input array into two sub-arrays using a pivot element, two tasks are created using #pragma omp task directives. These tasks recursively call quick_sort_parallel to sort the left and right sub-arrays concurrently. The first private clause is used to specify that each task should have its own private copy of certain variables.

This parallelization strategy can potentially reduce the run time of quick-sort by taking advantage of multiple processors or cores on a computer. By concurrently sorting different sub-arrays, the algorithm can potentially achieve a speedup over its serial counterpart.

However, it's important to note that not all problems can be effectively parallelized, and overhead may be associated with creating and managing tasks. Additionally, load balancing between threads can also impact performance.

3.6 Graph of Problem Size vs Algorithm Runtime

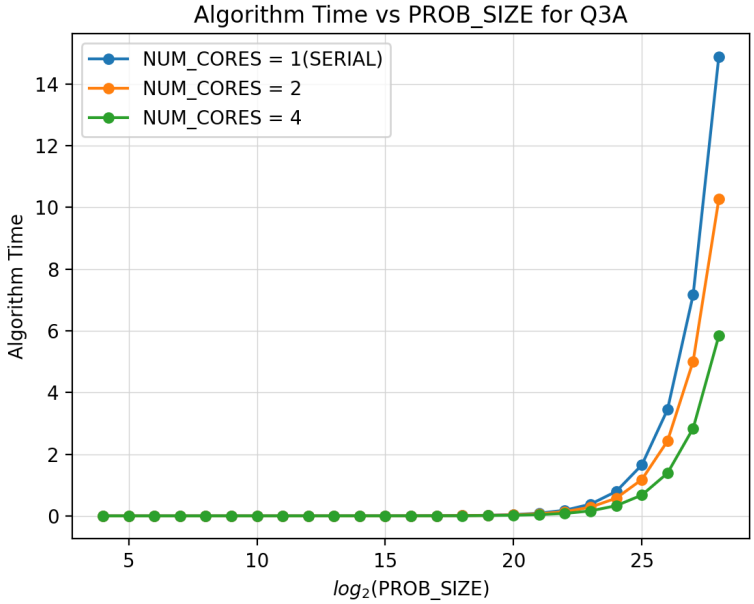

Figure 4: Graph of Problem Size vs Algorithm Runtime for Lab PC

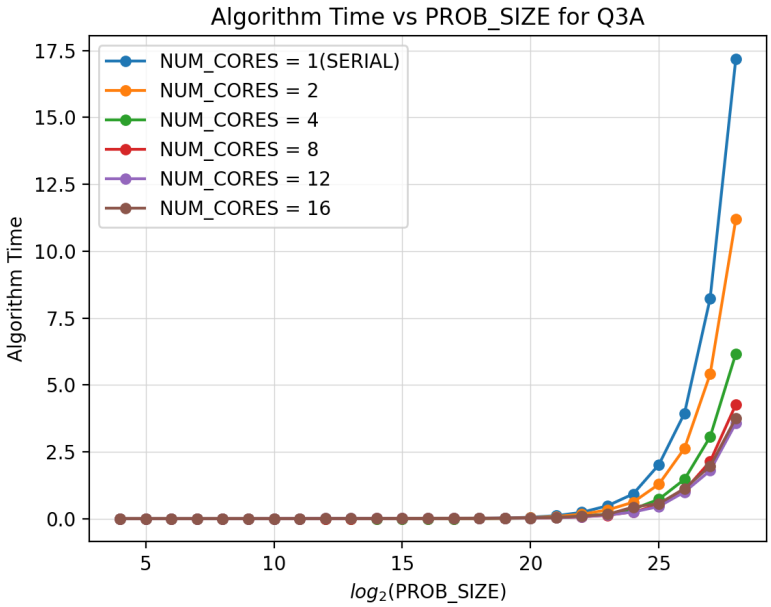

Figure 5: Graph of Problem Size vs Algorithm Runtime for HPC cluster

3.7 Graph of Problem Size vs End-to-End Runtime

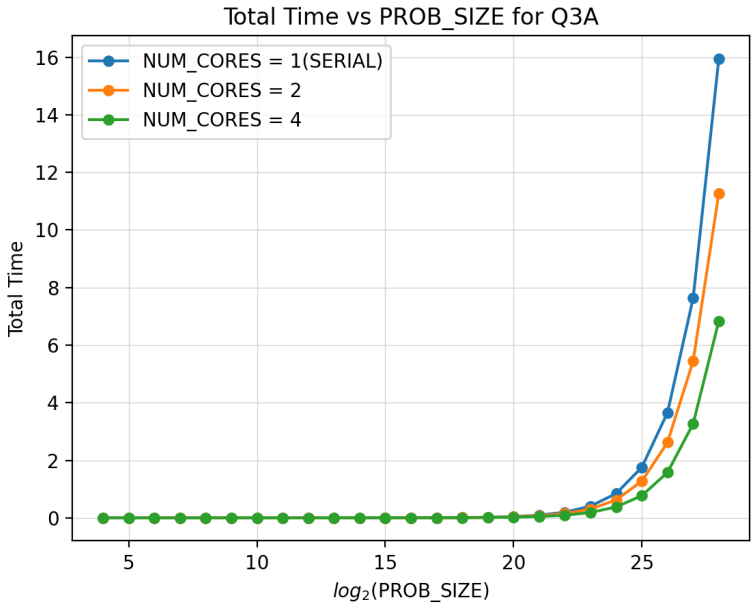

Figure 6: Graph of Problem Size vs End-to-End Runtime for Lab PC

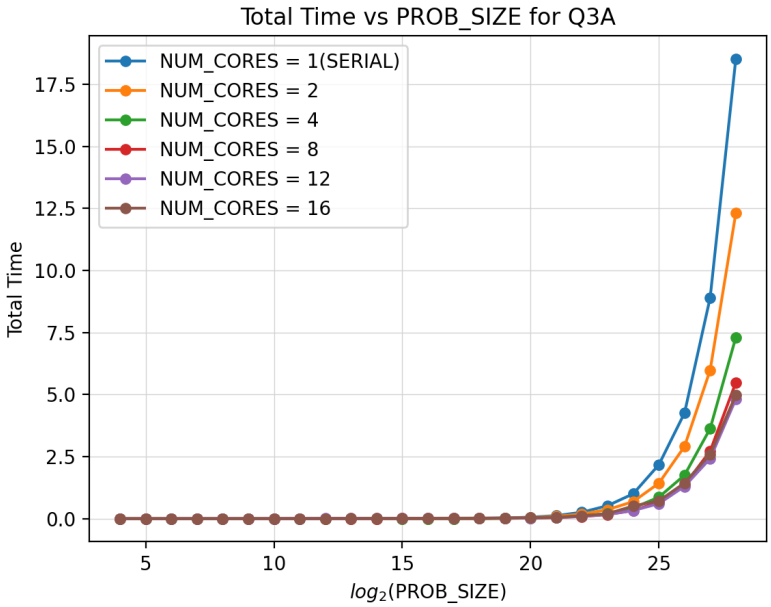

Figure 7: Graph of Problem Size vs End-to-End Runtime for HPC cluster

4 Discussion

The time complexity of the serial quick-sort algorithm is $O(n \log n)$ on average and $O(n^2)$ in the worst case, where n is the size of the input array. In an ideal scenario with perfect load balancing and negligible overhead, each thread would sort $\frac{n}{P}$ elements in $O(\frac{n}{P} \log \frac{n}{P})$ time. Since all threads work concurrently, this would result in a parallel time complexity of $O(\frac{n \log n}{P})$, where P is the number of processors used.

However, in practice it may be higher due to non-ideal conditions such as imbalanced loads between threads or overhead associated with creating and managing tasks. As the number of processors increases, the run time of quick-sort may decrease up to a certain point. Beyond this point, further increasing the number of processors may result in an increase in run time due to factors such as overhead or load imbalance.

This can be modeled mathematically using Amdahl's Law which states that the speedup achieved by parallelizing a program is limited by the fraction of the program that must be executed serially. Let S be the fraction of quick-sort that must be executed serially (e.g., partitioning) and let P be the number of processors used. Then according to Amdahl's Law, the maximum speedup achievable by parallelizing quick-sort is given by:

$$\text{Speedup} = \frac{1}{S + \frac{1-S}{P}}$$

As can be seen from this equation, as P increases, Speedup approaches $\frac{1}{S}$ asymptotically. This means that beyond a certain point, further increasing P will result in diminishing returns and may even increase run time due to factors such as overhead or load imbalance.