CS 301
High-Performance Computing

Lab 4 - B1

Problem B-1: SUM_VECTORS

Aditya Nawal (202001402)
Divya Patel (202001420)

March 15, 2023

Contents

1 Introduction

Vector addition is a basic operation in linear algebra that involves adding two vectors element-wise to produce a third vector. Parallel vector addition is a technique that exploits the independence of each element-wise addition to perform the operation concurrently on multiple processors or cores. The goal of this report is to evaluate the performance of parallel vector addition, with a focus on its run-time. In this report, we describe the hardware specifications used in the experiments, as well as the input parameters, output, and accuracy checks. We present the results of our performance evaluation, including plots that compare the run-time of the algorithm against the problem size.

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 B1

3.1 Description of the problem

The goal of this problem is to implement a program that performs the summation of two vectors followed by the addition of a constant. Given two input vectors A and B of size n, and a constant k, the program should compute a new vector C such that:

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

for all $i \in [0, n-1]$.

The program should be implemented in both serial and parallel versions. The serial version should perform the computation using a single thread while the parallel version should take advantage of multiple processors or cores on a computer to perform the computation concurrently.

3.2 Serial Complexity

Let n be the size of each vector. Assume that addition is constant-time operation.

The computation time for serial vector addition is given by:

$$T_{comp} = n \times O(1) = O(n) \tag{1}$$

Therefore, the total time complexity for serial vector addition is given by:

$$T_{total} = T_{comp} = O(n) \tag{2}$$

3.3 Parallel Complexity

To analyze the time complexity of parallel vector addition we need to consider two aspects: the computation time and the communication time. The computation time is the time spent by each processor or core to perform the element-wise addition of two vectors. The communication time is the time spent by each processor or core to exchange data with other processors or cores.

Let n be the size of each vector and p be the number of processors or cores. Assume that each processor or core has a local memory that can store n/p elements of each vector and that there is a shared memory that can store n elements of each vector. Also assume that multiplication and addition are constant-time operations.

For simplicity we will consider only one thread per processor i.e., $t = 1$. This means that each processor performs all its computations sequentially.

The computation time for parallel vector addition is given by:

$$T_{comp} = \frac{n}{p} \times O(1) = O(\frac{n}{p}) \tag{3}$$

This is because each processor performs n/p element-wise additions in constant time.

The communication time for parallel vector addition depends on how the data is distributed among the processors or cores. One possible way is to use a block distribution where each processor or core gets a contiguous block of n/p elements from each vector. In this case, the communication time is given by:

$$T_{comm} = O(\log p) \tag{4}$$

This is because each processor or core needs to send its partial sum to a shared memory using a binary tree reduction algorithm.

Therefore, the total time complexity for parallel vector addition is given by:

$$T_{total} = T_{comp} + T_{comm} = O(\frac{n}{p} + \log p) \tag{5}$$

3.4 Profiling Information

Following are the snapshots taken while profiling.

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
100.44      1.41     1.41                             main
  0.00      1.41     0.00        2     0.00     0.00  diff

 %         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

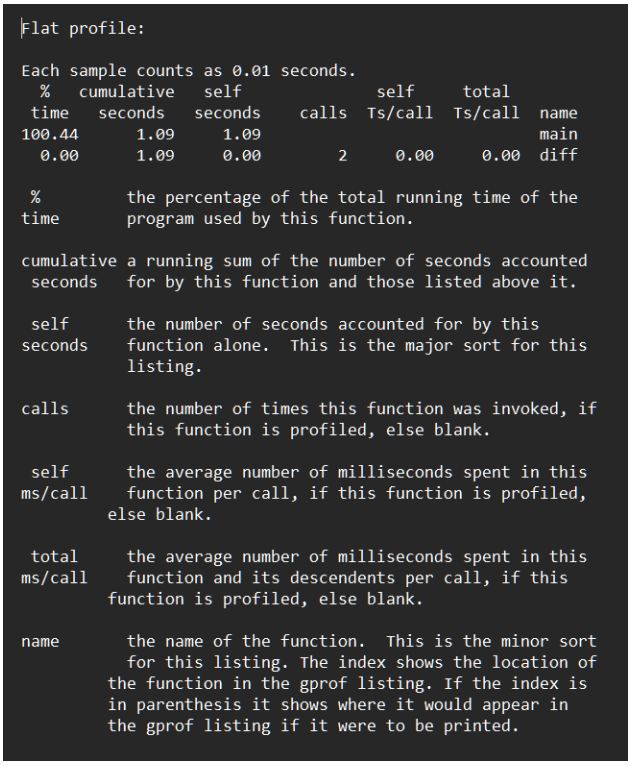

Figure 3: Profiling on HPC cluster using gprof

3.5 Optimization Strategy

The optimization strategy used in this code is **parallelization**. In the given code snippet, the `#pragma omp parallel for` directive is used to parallelize the for loop that calculates the sum of two vectors followed by the addition of a constant. This means that the iterations of the loop are distributed among multiple threads and executed concurrently.

```
#pragma omp parallel for private(i)
for (i = 0; i < N; i++) {
    sum[i] = a[i] + b[i] + c;
}
```

In OpenMP, `#pragma omp parallel for` is a combined directive that specifies that a `for` loop should be executed in parallel by multiple threads. The `private(i)` clause specifies that each thread should have its own private copy of the variable `i`, which is used as the loop index.

This can potentially speed up the execution time of the code by taking advantage of multiple processors or cores on a computer. However, it's important to note that not all problems can be effectively parallelized and there may be overhead associated with creating and managing threads.

3.6 Graph of Problem Size vs Algorithm Runtime

3.6.1 Graph of Problem Size vs Algorithm Runtime for LAB207 PCs

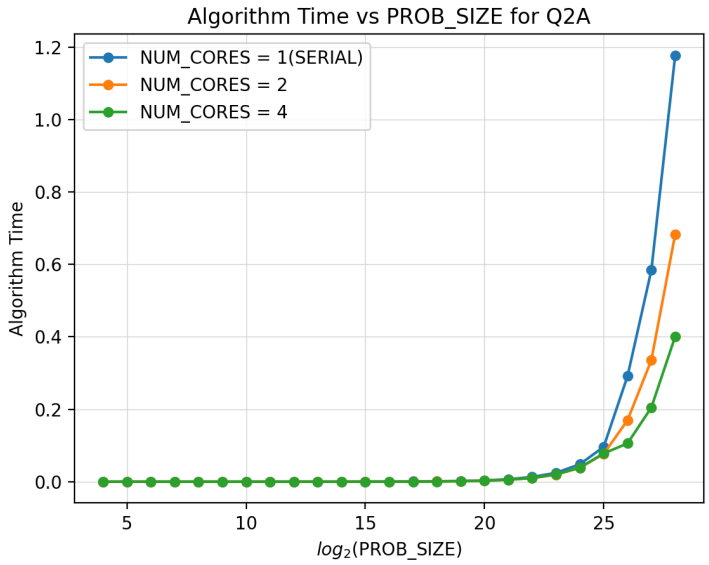

Figure 4: Graph of Problem Size vs Algorithm Runtime for Lab PC

3.6.2 Graph of Problem Size vs Algorithm Runtime for HPC Cluster

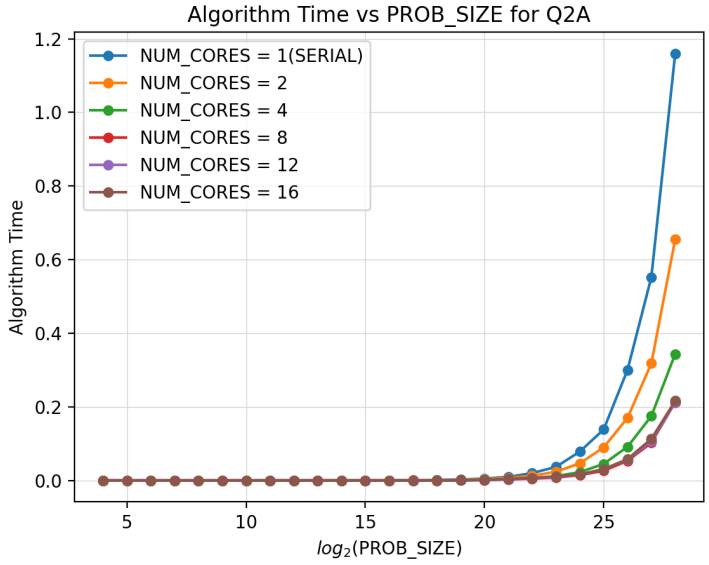

Figure 5: Graph of Problem Size vs Algorithm Runtime for HPC cluster

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

3.7.1 Graph of Problem Size vs End to End Runtime for LAB207 PCs

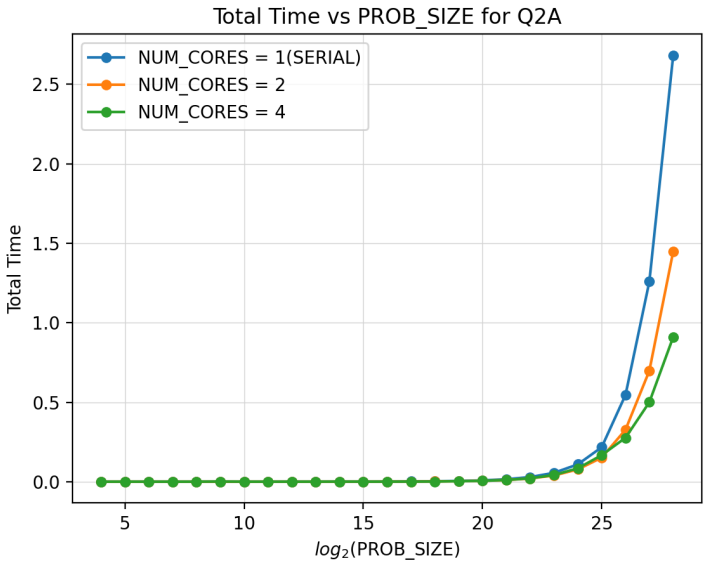

Figure 6: Graph of Problem Size vs End-to-End Runtime for Lab PC

3.7.2 Graph of Problem Size vs End to End Runtime for HPC Cluster

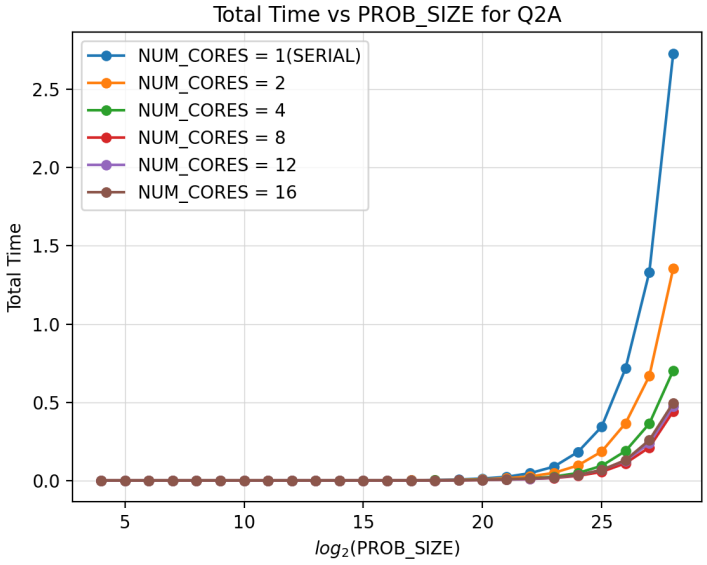

Figure 7: Graph of Problem Size vs End-to-End Runtime for HPC cluster

4 Discussion

The time complexity of the serial vector addition algorithm is $O(n)$, where n is the size of the input vectors. In an ideal scenario with perfect load balancing and negligible overhead, each thread would add $\frac{n}{P}$ elements in $O(\frac{n}{P})$ time. Since all threads work concurrently, this would result in a parallel time complexity of $O(\frac{n}{P})$, 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 vector addition 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 vector addition that must be executed serially (e.g., initializing variables) and let P be the number of processors used. Then according to Amdahl's Law, the maximum speedup achievable by parallelizing vector addition 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.