# CS 301
# High-Performance Computing

## Lab 4 - A1

Problem A-1: PI_TRAPEZOIDAL

Aditya Nawal (202001402)
Divya Patel (202001420)

March 15, 2023

# Contents

# 1 Introduction

In this problem, we will explore the integration of the function $\frac{4}{1+x^2}$ using the Trapezoidal rule in both serial and parallel implementations. The Trapezoidal rule is a numerical integration technique that approximates the definite integral of a function by dividing the area under its curve into trapezoids and summing their areas. The formula for this technique is given by:

$$\int_a^b f(x)dx \approx \frac{b-a}{2n}[f(a) + 2f(a+h) + 2f(a+2h) + ... + 2f(b-h) + f(b)]$$

where $h = \frac{b-a}{n}$ is the width of each trapezoid and $n$ is the number of trapezoids.

To implement this technique, we first divide the interval $[a, b]$ into $n$ equal subintervals with width $h$. Then, we evaluate the function at each endpoint of these subintervals and apply the above formula to approximate the definite integral.

We will implement this technique in both serial and parallel versions to compare their performance.

# 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

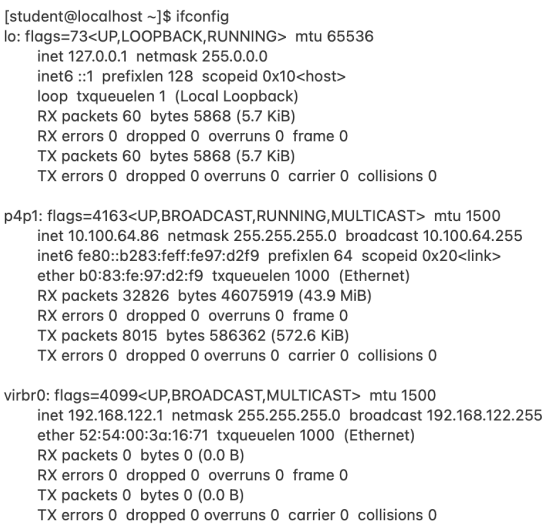


Figure 1: IP address of Lab PC

## 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 A1

## 3.1 Description of the problem

The goal of this problem is to implement a program that performs the numerical integration of the function $\frac{4}{1+x^2}$ using the Trapezoidal rule. Given an interval $[a, b]$ and a number of trapezoids $n$, the program should compute an approximation to the definite integral of the function over this interval.

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

The time complexity of the serial trapezoidal rule for estimating pi is $O(n)$, where $n$ is the problem size. This is because it requires $n$ evaluations of the function and $n$ additions to compute the total area.

## 3.3 Parallel Complexity

The parallel trapezoidal rule reduces this complexity by dividing the work among $t$ threads. Each thread computes a partial sum over a subset of the intervals. The time complexity of this parallel algorithm is $O(n/t)$, assuming that the work is evenly distributed among the threads and there are no overheads due to synchronization or communication.

## 3.4 Profiling Information

Following are the snapshops taken while profiling.

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 54.42      0.26     0.26 15543320     0.00     0.00  function
 41.86      0.46     0.20        1   200.95   462.18  main
  4.19      0.48     0.02        1    20.09    20.09  integrate
  0.00      0.48     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

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 56.87      0.90     0.90        1     0.90     1.60  main
 41.71      1.57     0.66  2887935     0.00     0.00  function
  1.90      1.60     0.03        2     0.02     0.02  diff
  0.00      1.60     0.00        1     0.00     0.00  integrate

 %         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

The code uses OpenMP to parallelize the for loop that computes the sum of function evaluations:

```
#pragma omp parallel for reduction(+:sum) private(i)
for (i = 1; i < n; i++)
{
    double x = a + i * h;
    sum += function(x);
}
```

The `#pragma omp parallel for` directive instructs the compiler to generate multi-threaded code for this loop. The `reduction(+:sum)` clause specifies that the variable `sum` should be subject to a reduction operation with the `+` operator. This means that each thread computes a partial sum and at the end of the loop, all partial sums are combined into a single value.

The `private(i)` clause specifies that the variable `i` should be private to each thread. This means that each thread has its own copy of this variable and can modify it independently without interfering with other threads. Additionally, To optimise, we added up all of the f $(x_0), f(x_1), ..., (x_{n-1})$ and then multiplied it by two. In this manner, we can lower the quantity of flop operations.

### 3.6 Graph of Problem Size vs Algorithm Runtime

#### 3.6.1 Graph of Problem Size vs Algorithm Runtime for LAB207 PC

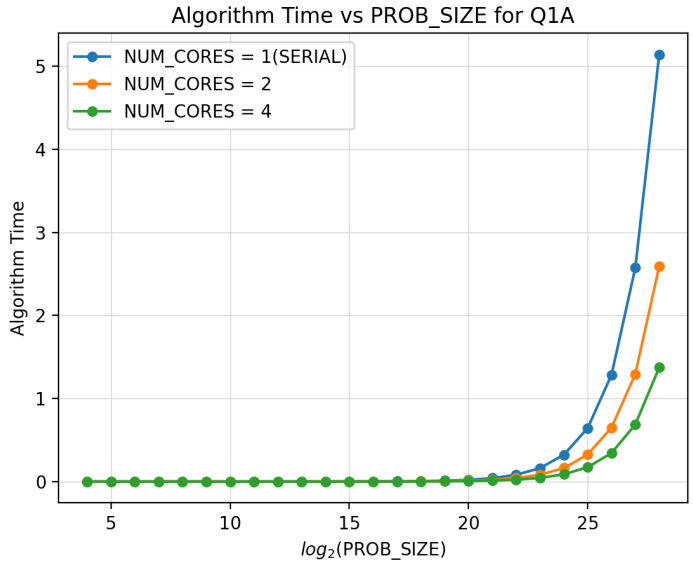


Figure 4: Mean Algorithm execution time vs Problem size plot for Lab 207 PC

#### 3.6.2 Graph of Problem Size vs Runtime for HPC Cluster

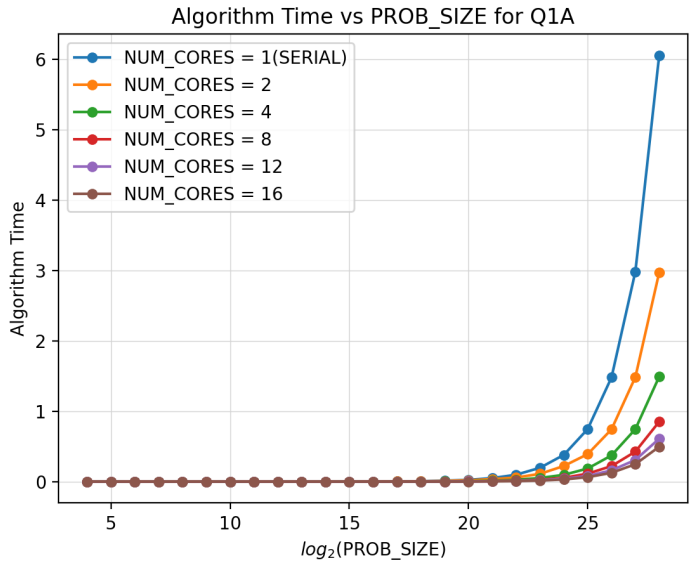


Figure 5: Mean Algorithm execution time vs Problem size plot for Lab 207 PC

## 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 PC

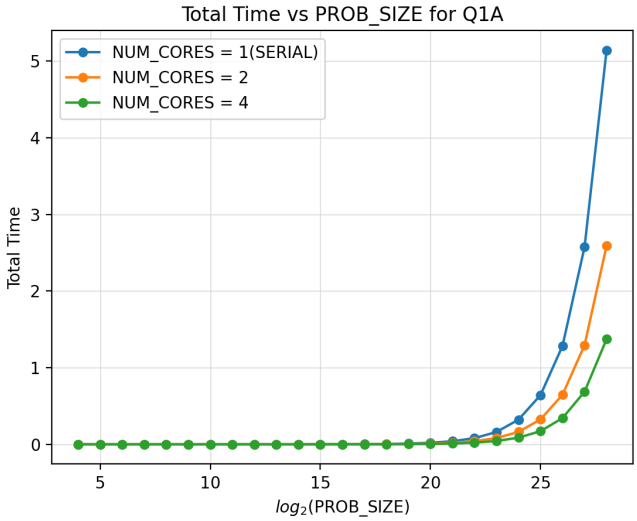


Figure 6: Mean End to End execution time vs Problem size plot for Lab 207 PC

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

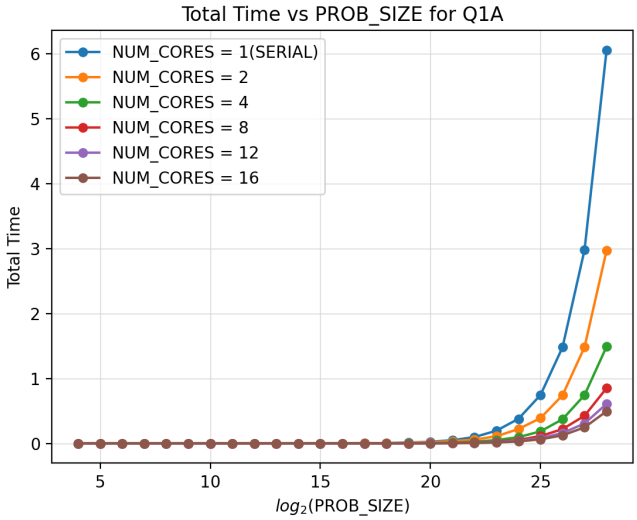


Figure 7: Mean End to End execution time vs Problem size plot for HPC Cluster

# 4 Discussion

The trapezoidal rule is a numerical integration technique used to approximate the definite integral of a function. In this problem, the trapezoidal rule is used to estimate the value of $\pi$ by integrating the function $f(x) = \frac{4.0}{1.0+x^2}$ over the interval $[0, 1]$.

By dividing the work among multiple threads, the parallel trapezoidal rule can reduce the time complexity from $O(n)$ for the serial version to $O(\frac{n}{t})$ for the parallel version, where $n$ is the problem size and $t$ is the number of threads.

According to Amdahl's law, which states that maximum speedup that can be achieved by parallelizing a program is limited by its sequential fraction. If a fraction $p$ of a program's execution time can be parallelized and run on $t$ threads then its maximum speedup is given by:

$$S = \frac{1}{(1-p) + \frac{p}{t}}$$

In this case since most of this code's execution time is spent in computing function evaluations within a for loop that has been parallelized using OpenMP directives we can expect significant speedup as we increase number of threads.

However it's important to note that there are overheads associated with creating and synchronizing threads. Therefore there will be diminishing returns as we keep increasing number of threads.

In conclusion using OpenMP to parallelize this code can significantly reduce its execution time and improve its performance on multi-core processors. However it's important to choose an appropriate number of threads based on factors such as problem size and hardware characteristics in order to achieve optimal performance.