Assignment 1

Rajdeep Pinge 201401103
Aditya Joglekar 201401086

1. **Context:**

- **Brief description of the problem:**

We have to calculate the value of pi using the trapezoidal method for the function,

f(x) = 4.0 / (1+x^2)

The above function when integrated over the interval [0,1] gives pi i.e. the area under the curve for the above function in the interval [0,1] is pi.

The serial code written is to numerically calculate the integral of a function using the trapezoidal rule. The key idea is to recognize that integration is essentially the area under the function for a given interval. This can be approximated by taking small enough trapezoids underneath the curve, calculating their area and summing them up. Since the trapezoids sometimes leave out some area while in some cases overshoot it, the answer obtained is approximate. The approximation improves as we go on increasing the number of trapezoids.

Hence : Our problem size is a function of number of trapezoids(n)

For more details please see the code attached with the report.

- **Complexity of the algorithm (serial):**

The problem boils down to calculating and summing the area of all trapezoids and hence the problem has a complexity of the order of number of trapezoids (n) i.e. O(n)

- **Possible speedup (theoretical)** :

For problem size = 1e8

Total time taken to run serial code: 0.860216

Time taken to run the parallelizable code: 0.860161

Hence p(fraction of code which can be parallelized)= 0.9999

So, we can expect a very high speedup as p is very much close to 1. However as we will see below this is an overtly exaggerated figure. The actual speedup we can expect is bounded by the computational power available(eg. number of cores, latency etc). Thus Amdahl's law is strictly theoretical.

- **Profiling information (e.g. gprof). Serial time**

Note: gprof works only for serial code. It doesn't make any sense for the parallel code.

Flat profiling data:

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name

```
 80.46            3.57    3.57     1         3.57     4.48  trap_area
 20.46            4.48    0.91 1000000001          0.00     0.00  f
```

Flat profiling indicates that it will be beneficial to parallelise the trap_area function

Call graph profiling data:

```
index % time  self  children   called  name
                3.57    0.91    1/1             main [2]
[1]     100.0   3.57    0.91    1       trap_area [1]
                0.91    0.00 1000000001/1000000001      f [3]
-----------------------------------------------
                                <spontaneous>
[2]     100.0   0.00    4.48            main [2]
                3.57    0.91    1/1             trap_area [1]
-----------------------------------------------
                0.91    0.00 1000000001/1000000001      trap_area [1]
[3]     20.3    0.91    0.00 1000000001         f [3]
-----------------------------------------------
```

The above profiling data shows that the program spends about 80% time in the trap_area function which is the main function to calculate pi. Although this function calls function f which uses 20% time, it is a computational single command function hence cannot be parallelized. This suggests that the efforts of parallelizing the code should be focussed on trap_area function.

- **Optimization strategy:**

Thus we can parallelise the calculation of the trapezoidal areas and their additions. Simply dividing this task amongst threads ought to do the trick.

1. Implicit work distribution among threads using #pragma omp for
This is the openmp built-in pragma to implicitly divide the for loop work fairly equally among all the threads. An important consideration for this pragma is that it should not contain branching otherwise the compiler would not understand exactly how much time each thread takes. Also we need to define the number of threads that we are going to use by using export OMP_NUM_THREADS bash command

2. Critical:
Critical is the pragma used such that while updating a global variable each thread can lock onto it and no other thread can update it during that time. This is required because in parallel programmes, there is high probability that there might arise race condition where-in multiple threads might update at the same time and due to multiple atomic operations in a single update operation, an update might be lost. Hence critical is an essential component of a parallel programme.

3. **Reduction**: This can be used when a associative mathematical operation is to be repeatedly performed on a large chunk of similar natured data for e.g say data stored in an array. It takes advantage of the multicore architecture of the computer. It is best illustrated through an example.

Suppose we need to add n items. We know that addition is associative. It also has an identity element 0. We need to perform n-1 additions. However the naive way to do it would be to use n-1 clock cycles (the sequential way). Here is where parallelism helps us.
If we have say 4 cores we can perform 4 additions at once giving 4 numbers which we add up in 2 additions. Finally we get the sum of 8 numbers in 3 processor cycles or 3 real time units as against 7 real time units. Note that the computation required obviously stays the same. The trick is to do as many things as possible 'at once'

This can be generalized as follows.
We can add (2*num_cores) numbers in log_base2(2*number of cores), real time units.
The 2*num_cores is because addition is a binary operation and each core can do this operation. The count of numbers goes on reducing by a factor of 0.5 every step. Hence the log_base2 term.

Thus to summarize, the complexity of an associative binary operation can be reduced to O(logN) from O(N).
If this construct is used within a parallel pragma in OpenMP, it implicitly takes care of any race conditions by mutual exclusion.

4. Scheduling: Scheduling is the process by which the inter-dependent tasks are performed in the most optimal way. Based on the apriori information of the tasks at hand there are two way of scheduling:

A. Static Scheduling: This is done by the compiler. If there is a stall in the pipeline, no further instruction is issued by the compiler. In terms of parallel programming, The tasks shared among threads is fixed and cannot be changed.
B. Dynamic Scheduling: This is enforced by the hardware. This is somewhat different from the static in the sense that if the stall does not produce any structural hazard then the next instruction can be issued. In terms of parallel programming, based on the availability of the free threads at run time, the work is divided. Dynamic scheduling contains a lot of overhead therefore must be used carefully.

We can use both these techniques to speed up the code further if there are any inter-dependencies.

- **Problems faced in parallelization and possible solutions.**

1. We first had naively divided the work amongst the threads and were adding up their answers in a shared global variable. However this portion being critical is susceptible to race conditions. If a thread reads the variable value at the time when an earlier thread has not finished updating the value, it will be a lost update. Thus we were getting wrong answers.
   The solution is to use a critical pragma, so that when a thread is in the process of updating the shared variable, the other threads have to wait thereby avoiding a race.

2. Problem in code due to critical pragma.
   First we used the critical pragma inside the for loop by keeping only a global variable and no variable private to the thread. This meant that in every for loop iteration, the critical section was executed and a lock was put on the global variable. This meant that the code was effectively running as the serial code even with parallel constructs.

3. Another problem we faced was that the parallel code was taking much more time than the serial code.
   Problem in code:
   Code Snippet: for (i=1;i<=(num_steps-1);i++) }
   ```
                    X_curr = start + i*h;
                    approx += f(X_curr);
               }
   ```
   Here we have used two instructions while calculating the next function value. The parallel code is slower because in the first instruction the calculated value is stored in memory and then in second instruction, it is fetched back from memory. The memory access time is much slower than the computation time. Hence Total time required increases much more.

   Solution:
   Code Snippet: for (i=1;i<=(num_steps-1);i++)
   ```
                    approx += f(start + i*h);
   ```
   Here instead of memory access we are directly doing the computation which is much faster. Hence the time required is less.

2. Hardware details: CPU model, memory information, no of cores, compiler, optimization flags

We have tested the code on two different machines having different attributes

A. Intel Core i3 Dual Core Processor

CPU Model: Intel(R) Core(TM) i3-4010U CPU @ 1.70GHz (but with 4 threads ie 2 threads per processor)

CPU MHz : 900.000

Memory Information: L1d cache 32K
L1i cache 32K
L2 cache 256K
L3 cache 3072K

Number of cores- 2
Threads per core = 2
Compiler gcc version 4.9
Optimization flags: not used
Precision: Double precision in 64-bit processor (Data Type: double)

Hyperthreading:

A technique patented by Intel in 2003 which provides the illusion of having 4 cores even if the device actually has only 2 ‘physical’ cores. We had used two devices to test the codes on. The AMD powered device had 4 physical cores but the Intel device had only 2. Yet when we compared the /proc/cpuinfo files from the two we were surprised to see the Intel processor showing 4 cores.

The reason behind this is hyperthreading. It simply means that there are two logical cores on a physical core. The logical cores however run concurrently not parallel i.e. they have to share the resources of the physical core and must run sequentially.

Hence, the speedup achieved on the Intel device was just about 2x while the AMD device could reach almost twice the speedup.

B. AMD APU_A10 Quad Core Processor
CPU Model : AMD A10-5750M APU with Radeon(tm) HD Graphics
CPU MHz : 1400.000
Memory Information: L1d cache: 16K
L1i cache: 64K
L2 cache: 2048K

No. of Cores: 4
Compiler: gcc version 5.4.0 20160609
Optimization Flags: Not used
Precision: Double precision in 64-bit processor (Data Type: double)

3. Input parameters. Output. Make sure results from serial and parallel are same.

Input params: Interval_start, interval_end, num_steps

Output: value of function (approximate value of pi in this case)

The answers of serial and parallel are in most cases exactly the same. On occasions when they differ the difference is negligible

Eg-

Input size	Serial answer	Parallel answer
1e8	3.141593	3.141593
1e9	3.141592	3.141593

4. Parallel overhead time. (openmp version on 1 core vs serial without openmp)

A. Intel

CASE I:

Number of steps(h) = 1e9

Serial code:

Answer = 3.141593

Time taken = 8.373209 s

Parallel code (1 thread):

Answer = 3.141593

Time taken = 8.842286 s

Hence overhead = 8.842286 - 8.373209 = 0.469077 s (~5.60% of serial time)

CASE II:

Number of steps(h) = 1e8

Serial code:

Answer = 3.141593

Time taken = 0.846696 s

Parallel code (1 thread):

Answer = 3.141593

Time taken = 0.872428 s (averaged out)

Hence overhead = 0.872428 - 0.846696 = 0,025 s (~3.03 % of serial time)

B. AMD

CASE I :

Number of steps(h) = 1e9

Serial code:

Answer = 3.141593

Time taken = 9.817233 s

Parallel code (1 thread):

Answer = 3.141593

Time taken = 10.545736 s

Hence overhead = 10.545736 - 9.817233 = 0.728503 s (~7.42% of serial time)

CASE II:

Number of steps(h) = 1e8

Serial code:

Answer = 3.141593

Time taken = 0.996384 s

Parallel code (1 thread):

Answer = 3.141593

Time taken = 1.049708 s

Hence overhead= 1.049708 - 0.996384 = 0.053324 s (~5.35% of serial time)

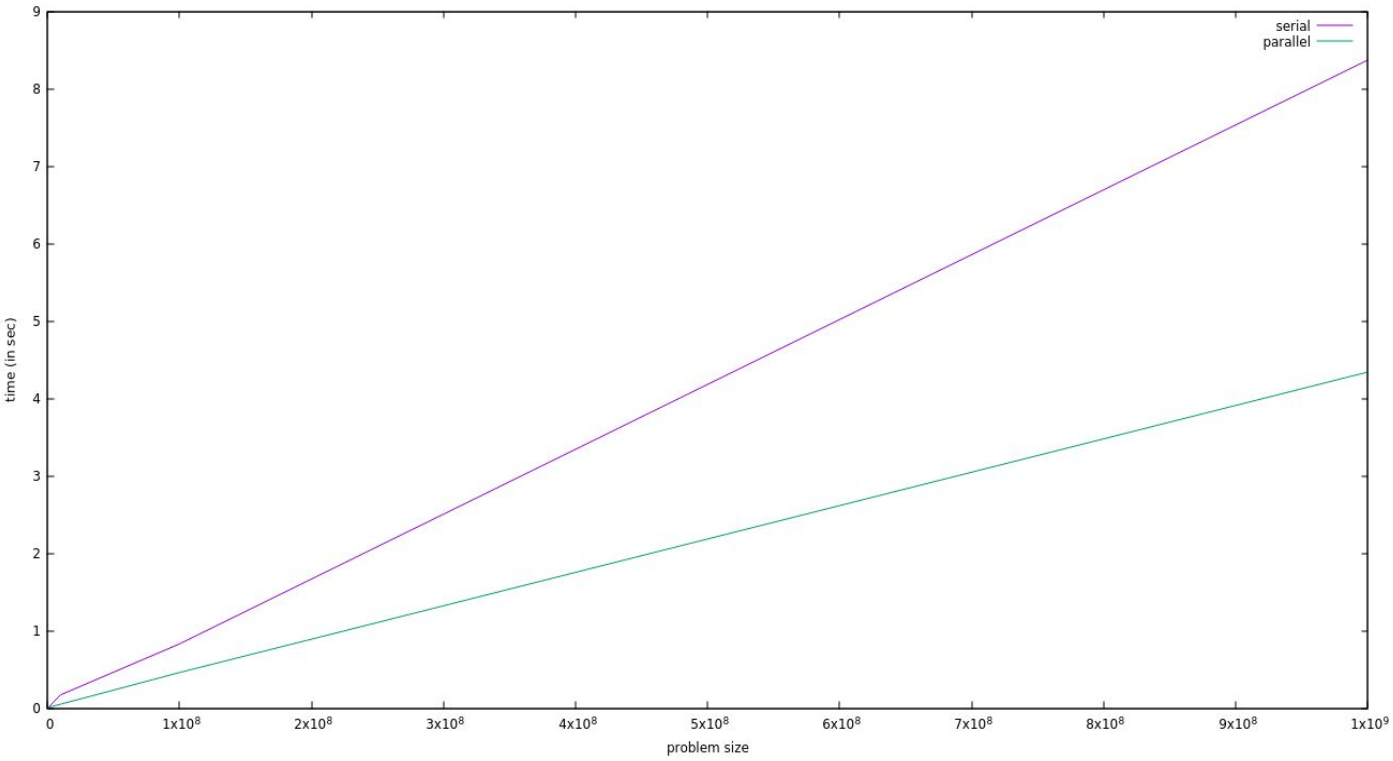
5. Problem Size vs Time (Serial, parallel) graph. Observations and comments about the results.

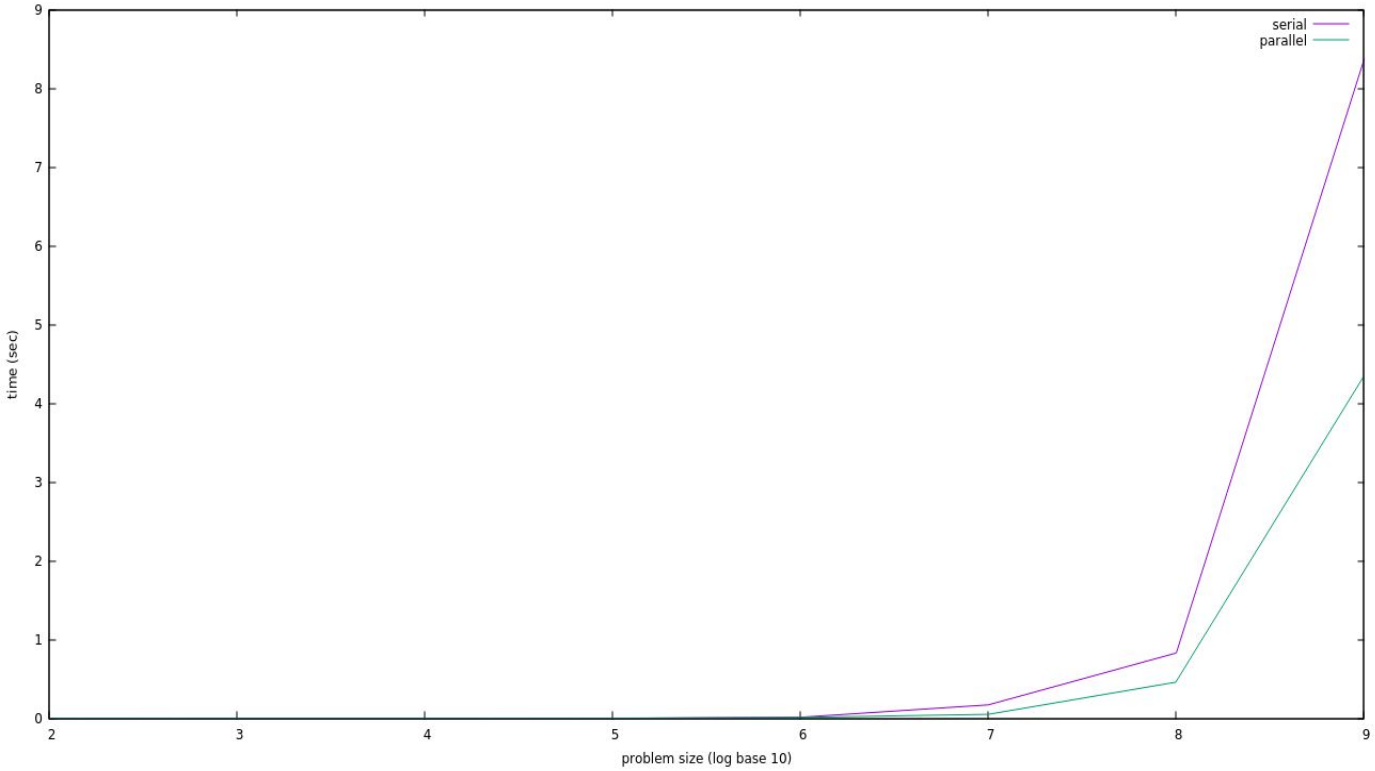
Observations:

Problem size (no. of iterations)	Time for serial code (in sec)	Time for parallel code (in sec)
100 = 1e2	0.000002	0.002528
1000 = 1e3	0.000028	0.000137
10000 = 1e4	0.000085	0.000230
100000 = 1e5	0.000128	0.003470
1000000 = 1e6	0.013806	0.013741
10000000 = 1e7	0.17286	0.052025
100000000 = 1e8	0.829832	0.461473
1000000000 = 1e9	8.373209	4.342792

Graphs:

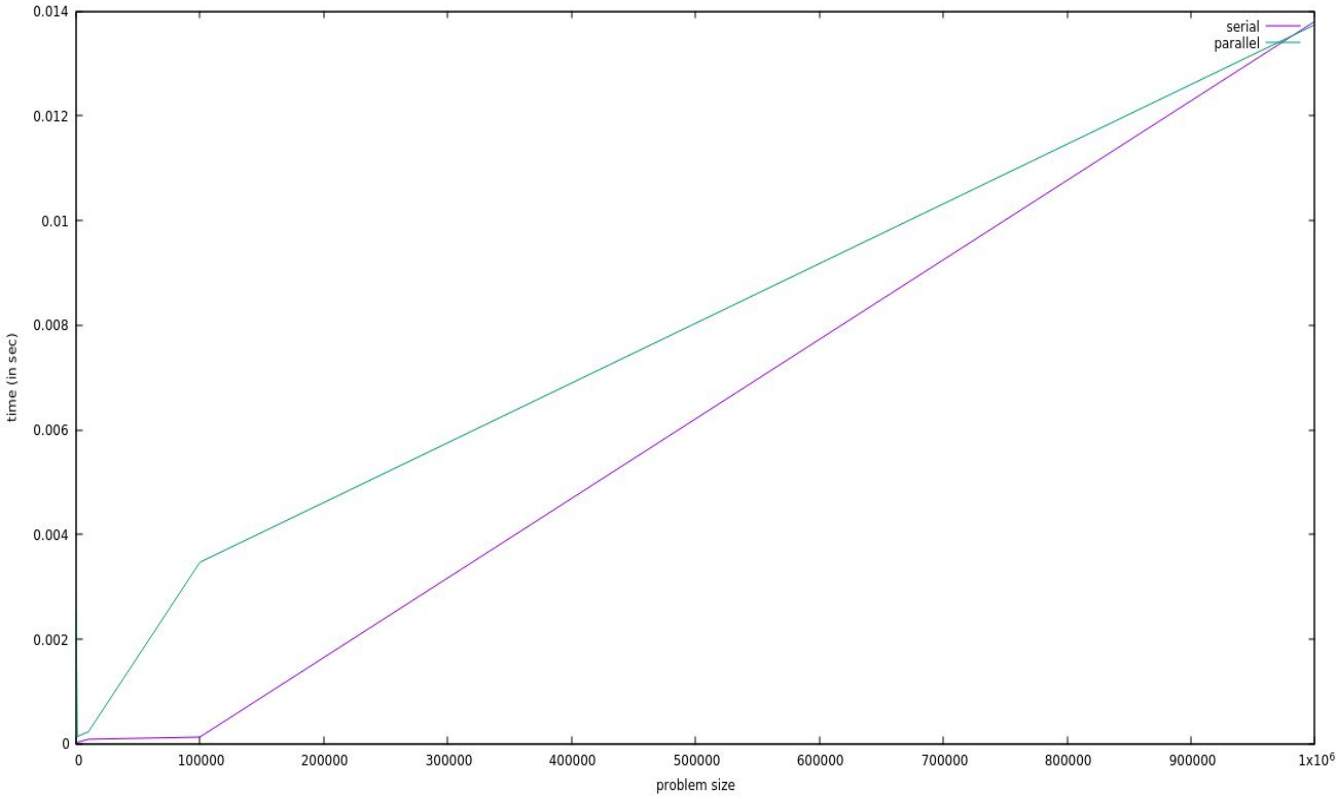
1. The following graph shows the problem size vs the time graph for both serial and parallel code. The graph gives a linear function i.e. as the problem size increases, the amount of time taken increases linearly.

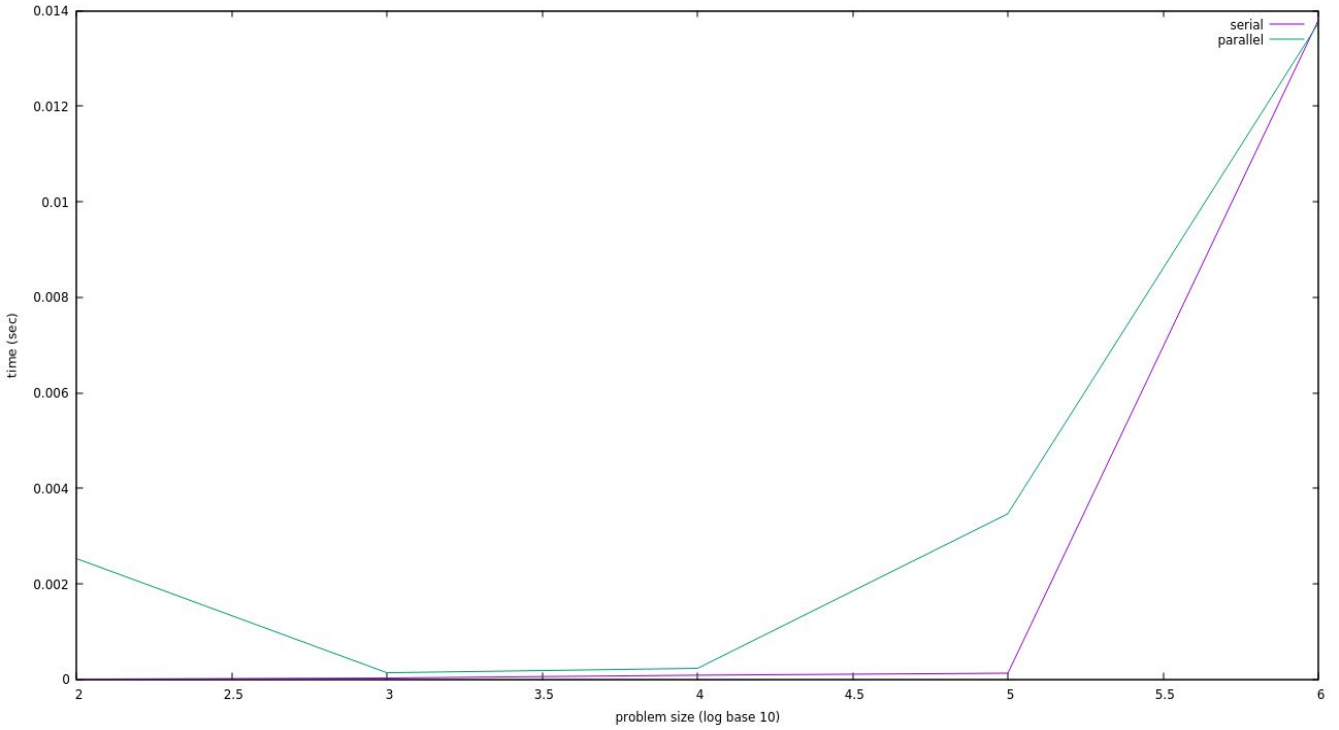




The above graph is the same graph of problem size vs time but with problem size taken in log scale. This is to conveniently understand the range of problem size in which time increases the most.

An interesting observation for Problem size < 1e6



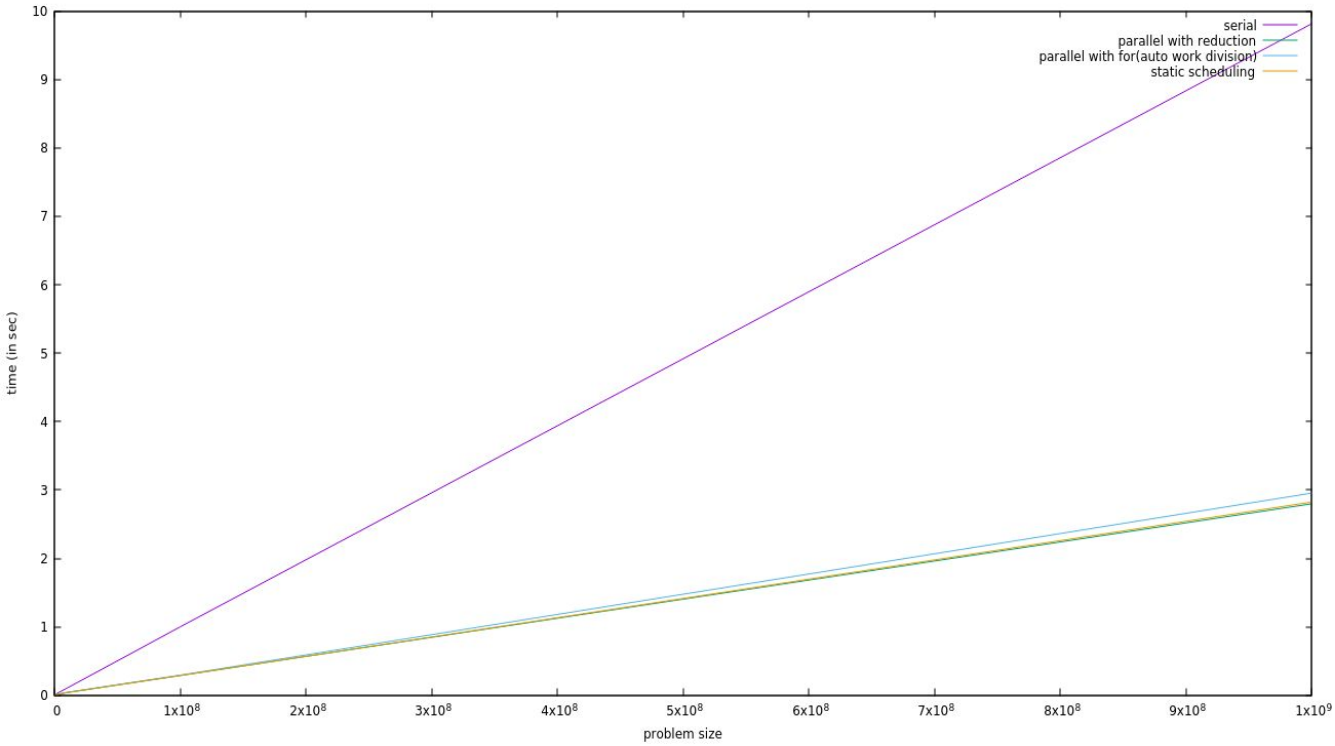


As per the observations, for problem size <= 1e6, serial code functions better than the parallel code as shown by both the above graphs. This is because the overhead time of the parallel code

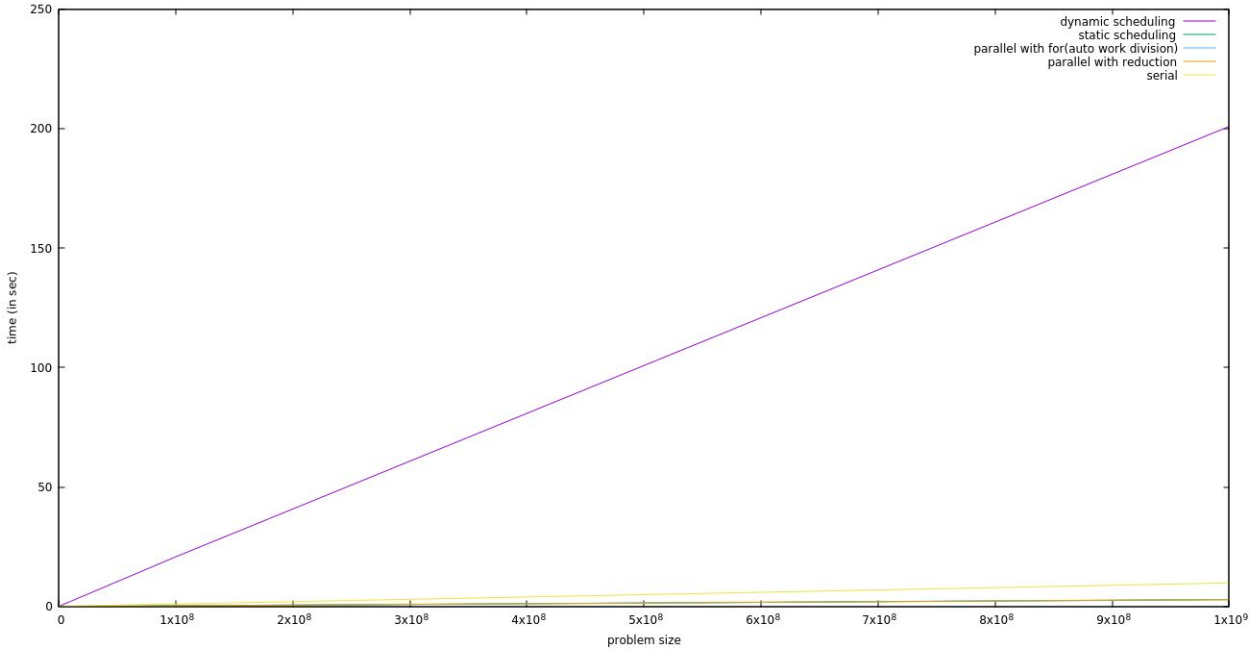
is much higher. But when the problem size is sufficiently high, the work done parallely in multiple threads saves a lot of time and hence the parallel code performs better of large number of iterations.

Comparing different methods of implementation for problem size vs time:

Problem Size	serial	parallel(reduction)	parallel(pragma for)	static_scheduling(default chunksize)	dynamic_scheduling(default chunksize)
1e2	0.000002	0.006369	0.000278	0.000251	0.001118
1e3	0.000018	0.001573	0.001438	0.000671	0.001451
1e4	0.000137	0.002560	0.002735	0.002453	0.003256
1e5	0.001755	0.006557	0.006090	0.005945	0.025454
1e6	0.013710	0.007014	0.006836	0.009256	0.210015
1e7	0.103584	0.038795	0.042559	0.037725	2.035680
1e8	0.997942	0.288748	0.292483	0.287172	20.739379
1e9	9.817233	2.797963	2.954334	2.824132	200.84245



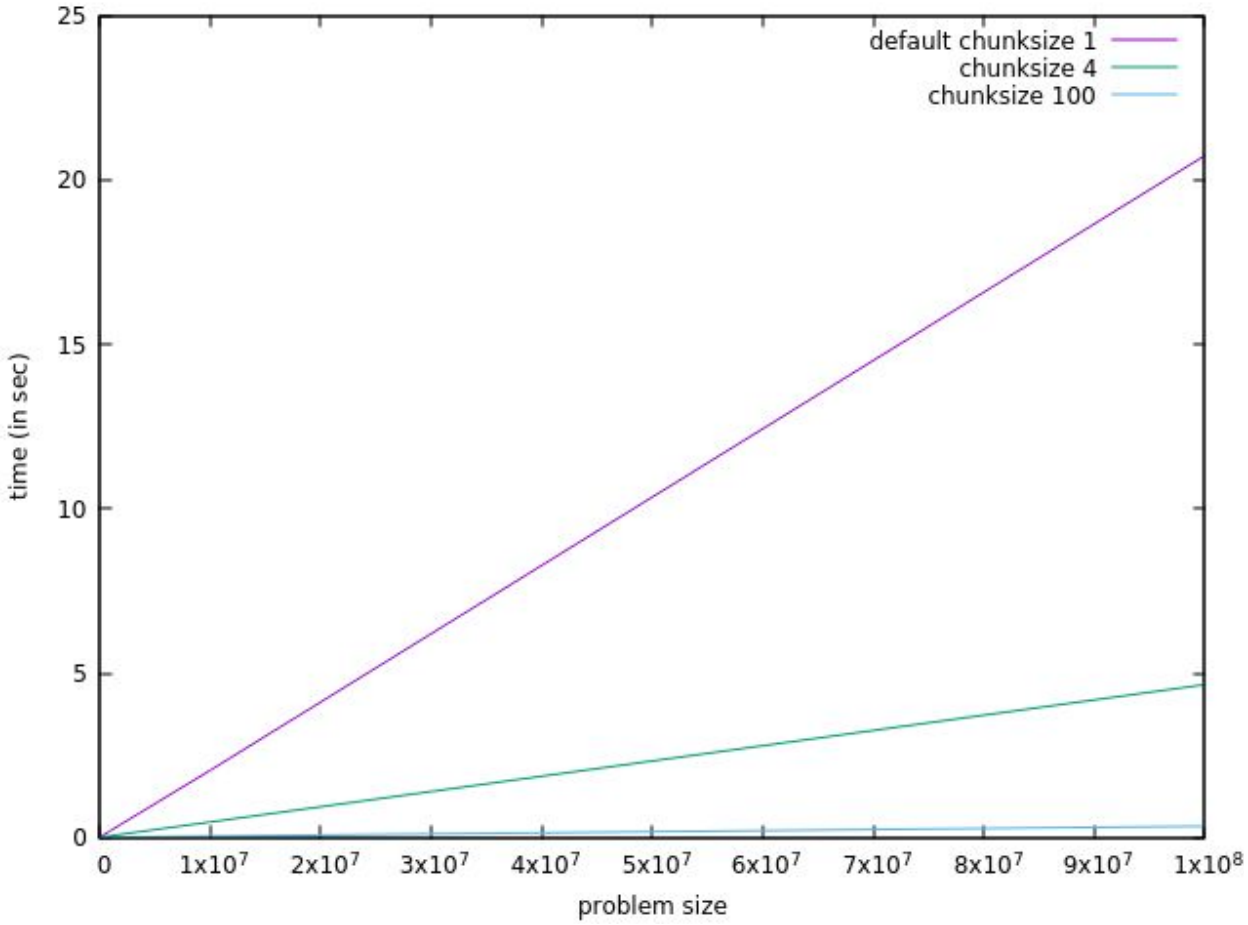
Here we have compared 4 different implementations of calculating PI. All the parallel methods, parallel(auto work division using 'for'), parallel with reduction, parallel with static scheduling(default chunksize = num_steps / number of threads) are very close in their complexities, while the serial method takes much more time.



Here we have compared all the methods with dynamic scheduling method. As can be seen from the graph, for large input size, the dynamic scheduling takes a lot of time due to its large overhead since its default chunksize is 1.

Since Dynamic scheduling is taking too much time here, we further compare dynamic scheduling method for different chunksizes.

Problem size	Chunksize = 1	Chunksize = 4	Chunksize = 100
1e2	0.001118	0.001692	0.002637
1e3	0.001451	0.006389	0.004852
1e4	0.003256	0.008033	0.005082
1e5	0.025454	0.013038	0.006318
1e6	0.210015	0.053649	0.007067
1e7	2.035680	0.467199	0.039392
1e8	20.739379	4.647082	0.344864



When the chunksize is near 1, the amount of overhead required is enormous as the problem size increases. As the chunksize increases, each thread does more work but the number of threads and the required overhead decreases and the time taken becomes comparable with the other methods.

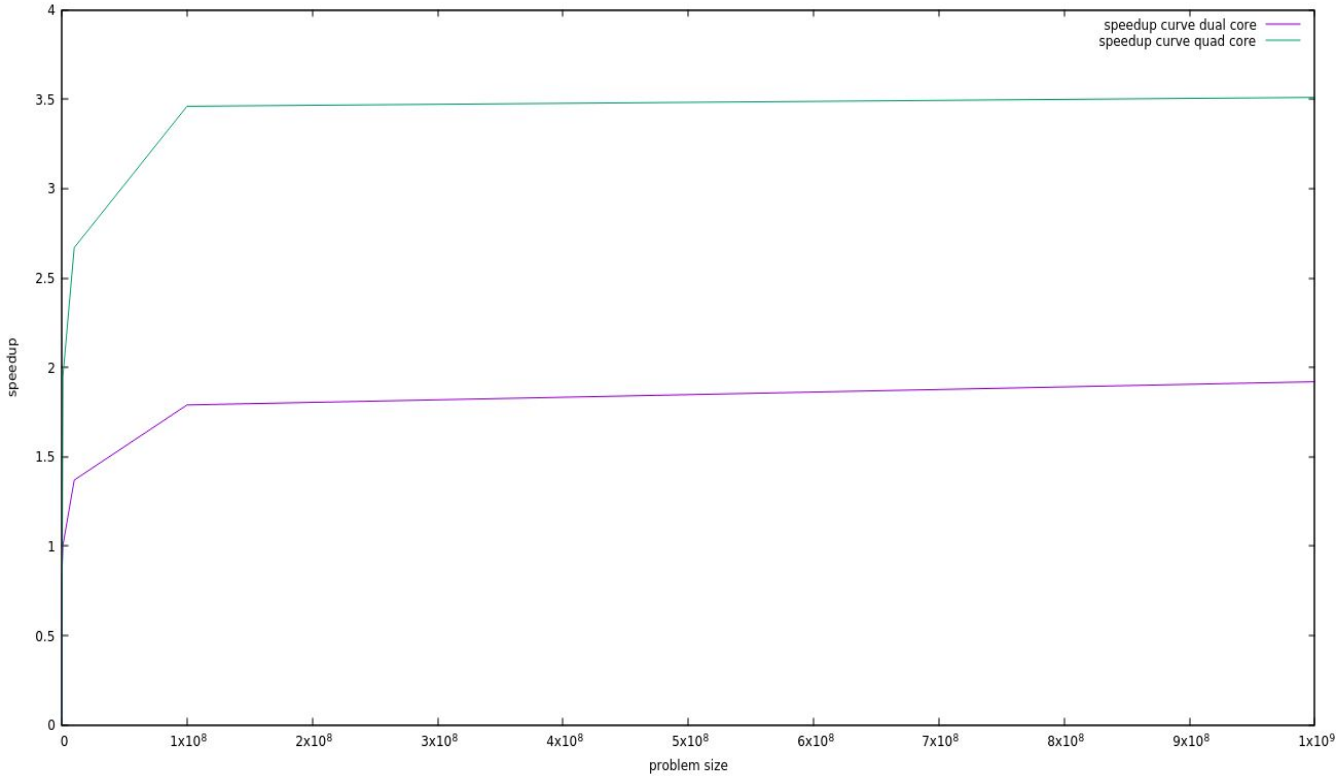
6. Problem size vs. speedup curve.

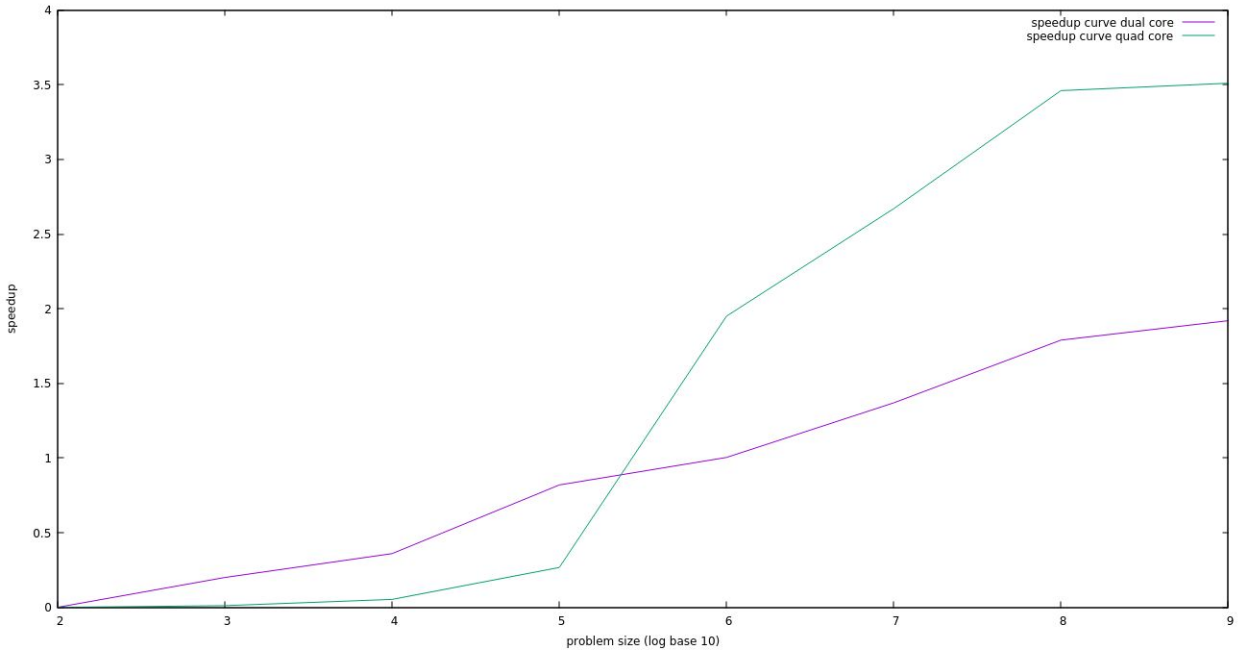
Speedup = T(serial) / T(parallel)

Observations:

Problem size (no. of iterations)	Speedup (Dual Core Intel)	Speedup (Quad Core AMD)
100 = 1e2	7.9e-4	3.14e-4
1000 = 1e3	0.2	1.14e-2
10000 = 1e4	0.36	5.35e-2
100000 = 1e5	0.82	0.267
1000000 = 1e6	1.004	1.95
10000000 = 1e7	1.37	2.67
100000000 = 1e8	1.79	3.46
1000000000 = 1e9	1.92	3.51

Graphs:





The above two graphs show the speedup obtained for the given problem for various problem sizes. As it is evident, the speedup obtained increases when problem size increases. But the rate of increase in speedup decreases. This means that even the parallel programme reaches a certain saturation point beyond which the speedup cannot be obtained.

The difference between the speedup curves for different number of processors is visible in the log scale graph. For Quad-core processor, when the problem size is small, the speed_up is less as compared to dual core. This is because, quad core processor has much more overhead than dual core. But when the problem size becomes sufficiently large, the quad core processor can give much more speedup because of its ability to do work simultaneously in 4 cores.

7. No. of cores vs. speedup curve for a couple of problem sizes.

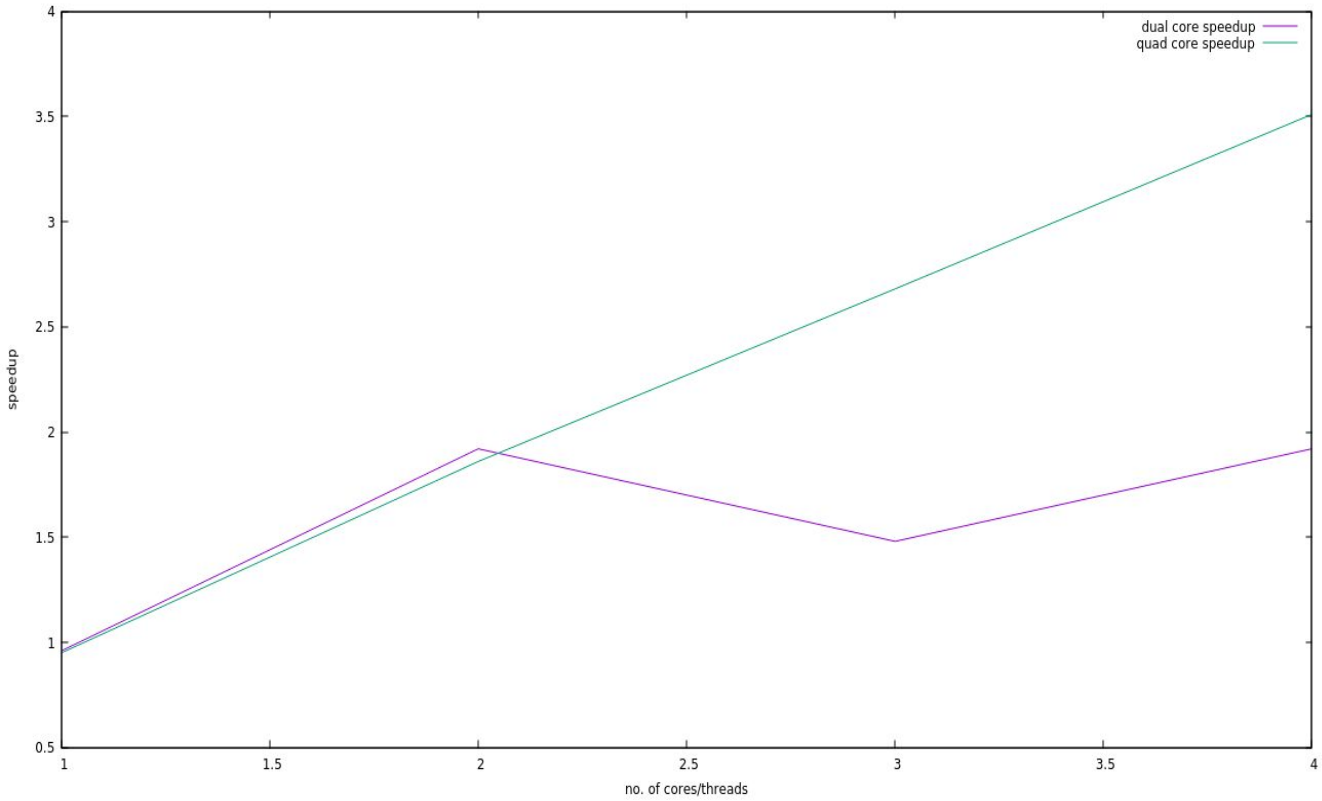
Observations:

No. of cores/threads	Speedup (Dual Core Intel)	Speedup (Quad Core AMD)
1	0.96	0.95
2	1.92	1.86
3	1.48	2.68
4	1.92	3.51

Dual Core: Since this is a dual-core processor with 4 possible threads, the code achieves almost the same speedup for both 2 and 4 threads. For 1 thread, there is some overhead time to create thread and parallelise the code for 1 thread but the code runs as if the serial code. Hence the speedup is less than one which shows that parallel code takes more time because of overhead. And interesting result is seen when we use 3 threads. Since this is a dual code architecture, therefore in 3 threads, 2 run on one core while the 3rd runs on the other core. This creates an imbalance and hence the achieved speedup is less than that with 2 cores.

Quad Core: This is quite straight-forward. The graph shows that as the no. of cores increase, the speedup increases.

Graph:

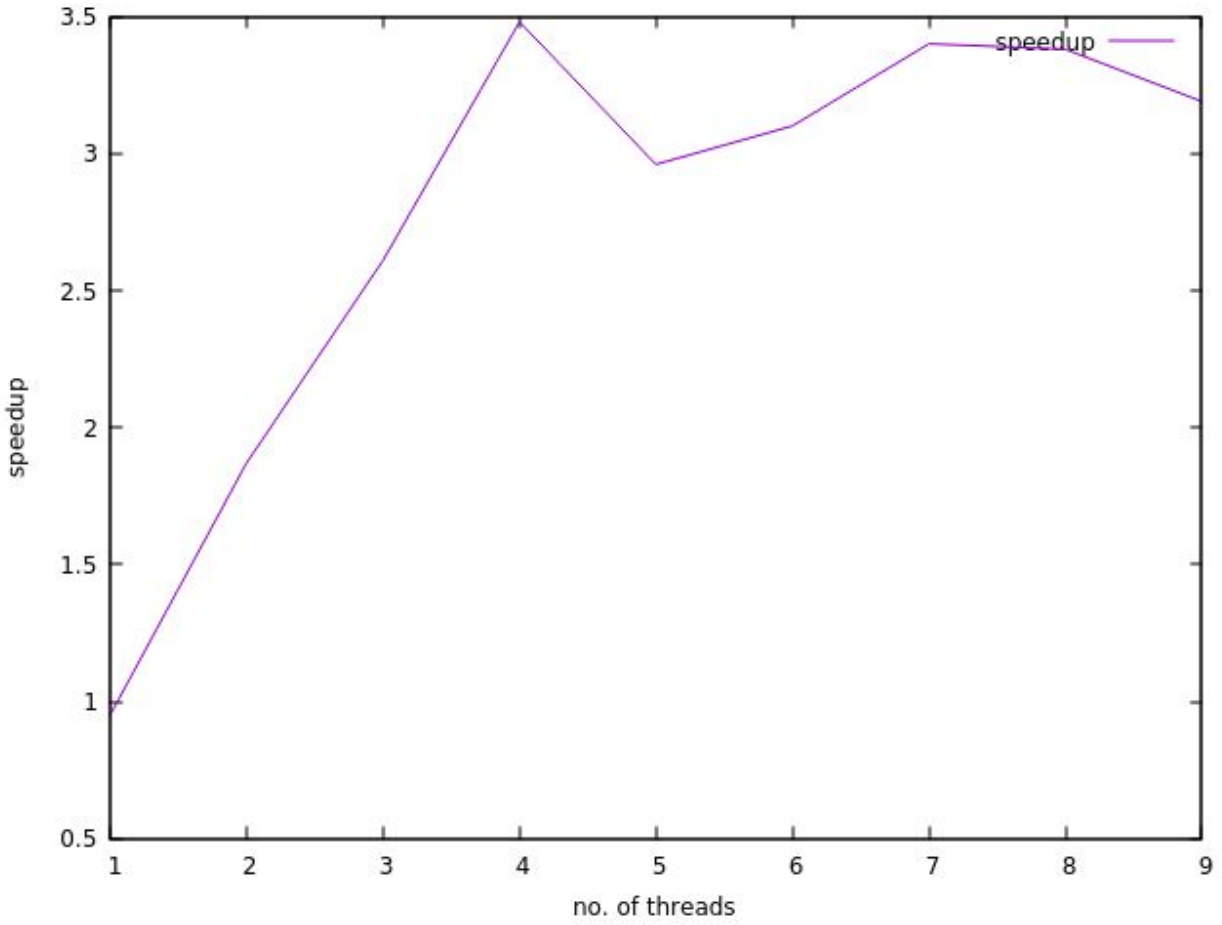


Experimenting further with more number of threads on the quad core processor, to get an idea of what happens when number of threads exceeds the available number of physical cores.

Problem size = 1e8

No. of cores/threads	Speed up
1	0.95
2	1.87
3	2.61
4	3.48
5	2.96
6	3.10

7	3.40
8	3.38
9	3.19
100	3.38
200	3.36

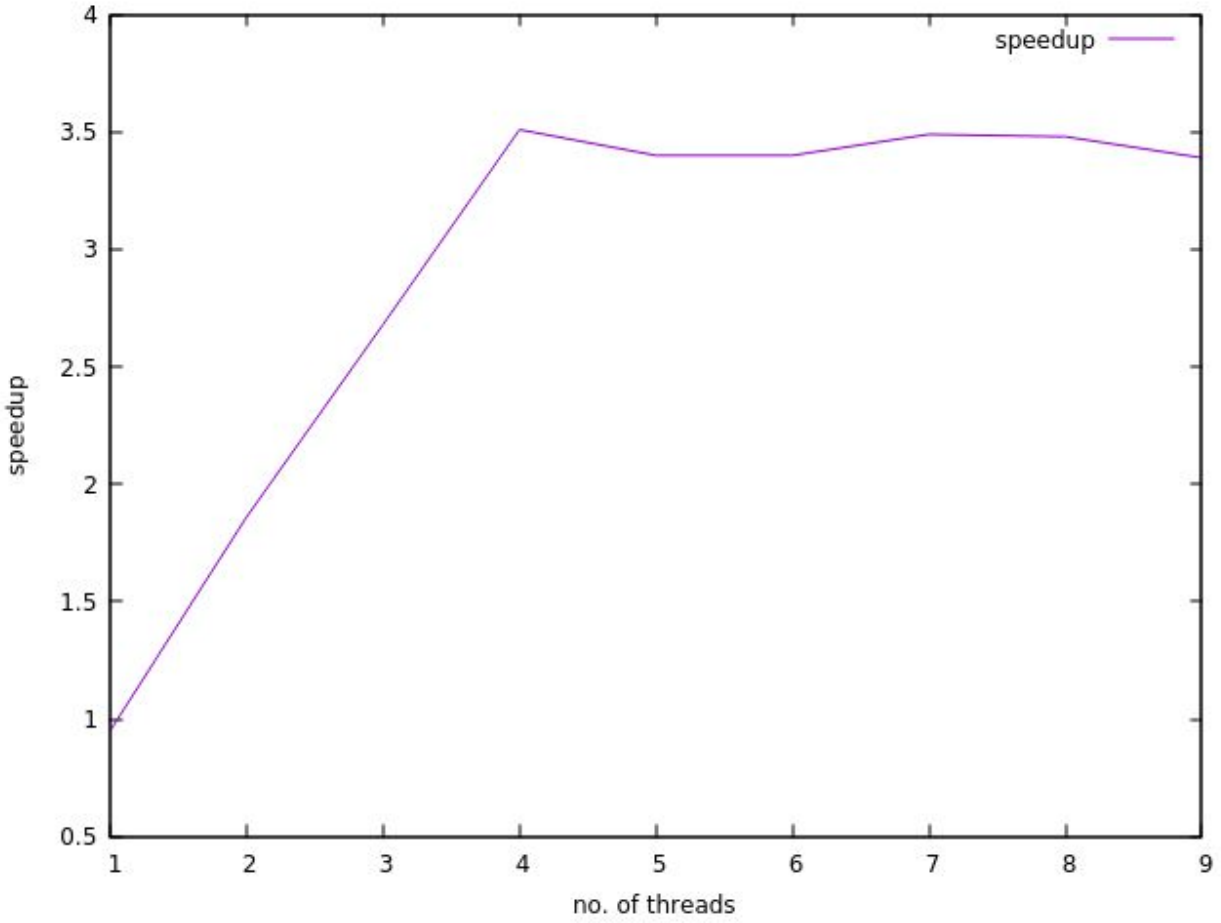


The above graph shows that even if the number of threads increase, the maximum speedup obtained remains equal to the available hardware cores. This means that even after increasing the number of threads, the speed obtained remains constant.

Problem size = 1e9

No. of cores/threads	Speed up
1	0.95

2	1.74
3	2.68
4	3.51
5	3.40
6	3.40
7	3.49
8	3.48
9	3.39
100	3.40
200	2.79



The above graph indicated that the speedup obtained by increasing the number of threads initially increases but later remains constant when the number of threads exceed number of available processors.
Another important observation from the table shows that as the number of threads increases too much, the speedup decreases. This might be because of the huge problem size and the large amount of overhead required to create those many number of threads. Also even after creating say 200 threads, since there are 4 processors, around 50 threads would run in a single core in serial manner hence no particular use of parallelism.

8. Measure performance in MFLOPS/sec.

Performance = Clock speed(MHz) * Operations per clock cycle * No. of Cores

Dual-Core Intel Processor Performance = 900 MHz * 4 * 2
= 7200 MFLOPS

Quad Core AMD Processor Performance = 1400MHz * 4 * 4
= 22400 MFLOPS