INDIAN INSTITUTE OF TECHNOLOGY GANDHINAGAR

ES 611: Algorithms on Advanced Computer Architectures

Report on Computational Lab Project 2.2

by

Shashank Heda

March 8th, 2013

ABSTRACT

OpenMP is a programming model for shared memory parallel architectures which aims to reduce the human effort involved in parallelizing a code. To get the maximum performance benefit from a processor with Hyper-Threading Technology, an application needs to be executed in parallel. Parallel execution requires threads, and threading an application is not trivial. Tools like OpenMP can make the process a lot easier. The growing popularity of such systems, and the rapid availability of product-strength compilers for OpenMP, seems to guarantee a broad take-up of this paradigm if appropriate tools for application development can be provided.

This paper presents the ways OpenMP can be used to get the most out of Hyper-Threading Technology. It shows the way the loops can be parallelized, called work sharing, as well as the ways to exploit non-loop parallelism and some additional OpenMP features.

1. Parallel Matrix-Vector Multiplication

To develop a multithreaded program, using OpenMP, to carry out matrix-vector multiplication and making a comparison with the timings observed for the Pthreads version (for row-wise striped partitioning)

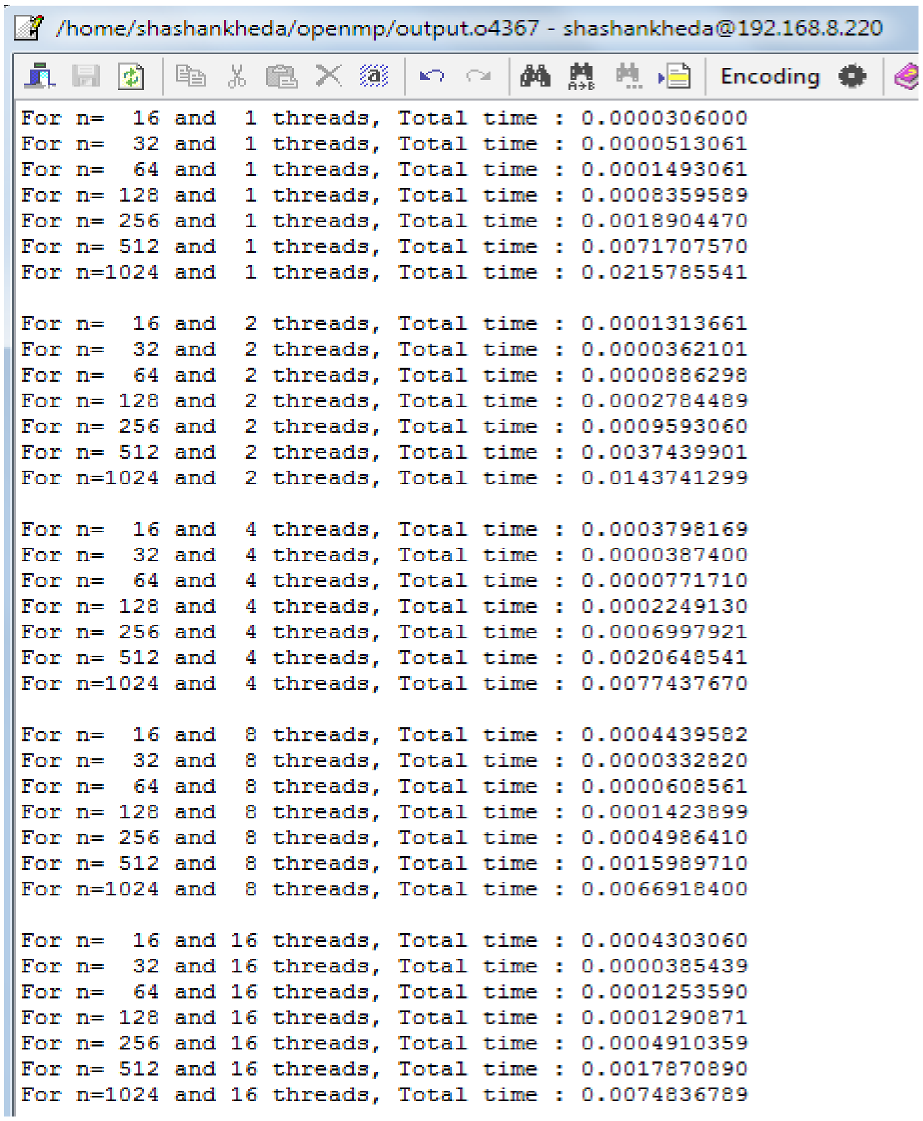
Method of approach

To carry out matrix-vector multiplication, y=Ax, where x is given vector and y is the vector that stores the result of the multiplication, we would perform the multiplication for the simplest case. After taking the order of the matrix as an input from the user, all the entries of the matrix as well as the vector are initialized to 1 (keeps the multiplication simple). This ensures that the result of the multiplication is an n*1 matrix with its each value = n (1+1+……+1 n times) and is a nice method to ensure whether our multiplication strategy is correct.

To perform the multiplication after receiving number of threads, matrix and vectors as inputs, we need to allocate memory for the matrix, vector and their product (during runtime). The external functions for Reading the matrix and vector as input from the user as well as function for getting the matrix-vector product can be parallelized easily using simple pragma (compiler) directives.

Results and Discussion

Output of the Program that multiplies the given matrix and the vector:


/home/shashankheda/openmp/output.o4367 - shashankheda@192.168.8.220
Encoding

```
For n=  16 and  1 threads, Total time : 0.0000306000
For n=  32 and  1 threads, Total time : 0.0000513061
For n=  64 and  1 threads, Total time : 0.0001493061
For n= 128 and  1 threads, Total time : 0.0008359589
For n= 256 and  1 threads, Total time : 0.0018904470
For n= 512 and  1 threads, Total time : 0.0071707570
For n=1024 and  1 threads, Total time : 0.0215785541

For n=  16 and  2 threads, Total time : 0.0001313661
For n=  32 and  2 threads, Total time : 0.0000362101
For n=  64 and  2 threads, Total time : 0.0000886298
For n= 128 and  2 threads, Total time : 0.0002784489
For n= 256 and  2 threads, Total time : 0.0009593060
For n= 512 and  2 threads, Total time : 0.0037439901
For n=1024 and  2 threads, Total time : 0.0143741299

For n=  16 and  4 threads, Total time : 0.0003798169
For n=  32 and  4 threads, Total time : 0.0000387400
For n=  64 and  4 threads, Total time : 0.0000771710
For n= 128 and  4 threads, Total time : 0.0002249130
For n= 256 and  4 threads, Total time : 0.0006997921
For n= 512 and  4 threads, Total time : 0.0020648541
For n=1024 and  4 threads, Total time : 0.0077437670

For n=  16 and  8 threads, Total time : 0.0004439582
For n=  32 and  8 threads, Total time : 0.0000332820
For n=  64 and  8 threads, Total time : 0.0000608561
For n= 128 and  8 threads, Total time : 0.0001423899
For n= 256 and  8 threads, Total time : 0.0004986410
For n= 512 and  8 threads, Total time : 0.0015989710
For n=1024 and  8 threads, Total time : 0.0066918400

For n=  16 and 16 threads, Total time : 0.0004303060
For n=  32 and 16 threads, Total time : 0.0000385439
For n=  64 and 16 threads, Total time : 0.0001253590
For n= 128 and 16 threads, Total time : 0.0001290871
For n= 256 and 16 threads, Total time : 0.0004910359
For n= 512 and 16 threads, Total time : 0.0017870890
For n=1024 and 16 threads, Total time : 0.0074836789
```

Figure 1.1 Output of the Program for matrix-vector multiplication

Table 1.1: Tabular Data for variation in timing vs n/p using OpenMP:

n/p	1	2	4	8	16
16	3.06E-05	1.31E-04	3.80E-04	4.44E-04	4.30E-04
32	5.13E-05	3.62E-05	3.87E-05	3.33E-05	3.85E-05
64	1.49E-04	8.86E-05	7.72E-05	6.09E-05	1.25E-04
128	8.36E-04	2.78E-04	2.25E-04	1.42E-04	1.29E-04
256	1.89E-03	9.59E-04	7.00E-04	4.99E-04	4.91E-04
512	7.17E-03	3.74E-03	2.06E-03	1.60E-03	1.79E-03
1024	2.16E-02	1.44E-02	7.74E-03	6.69E-03	7.48E-03

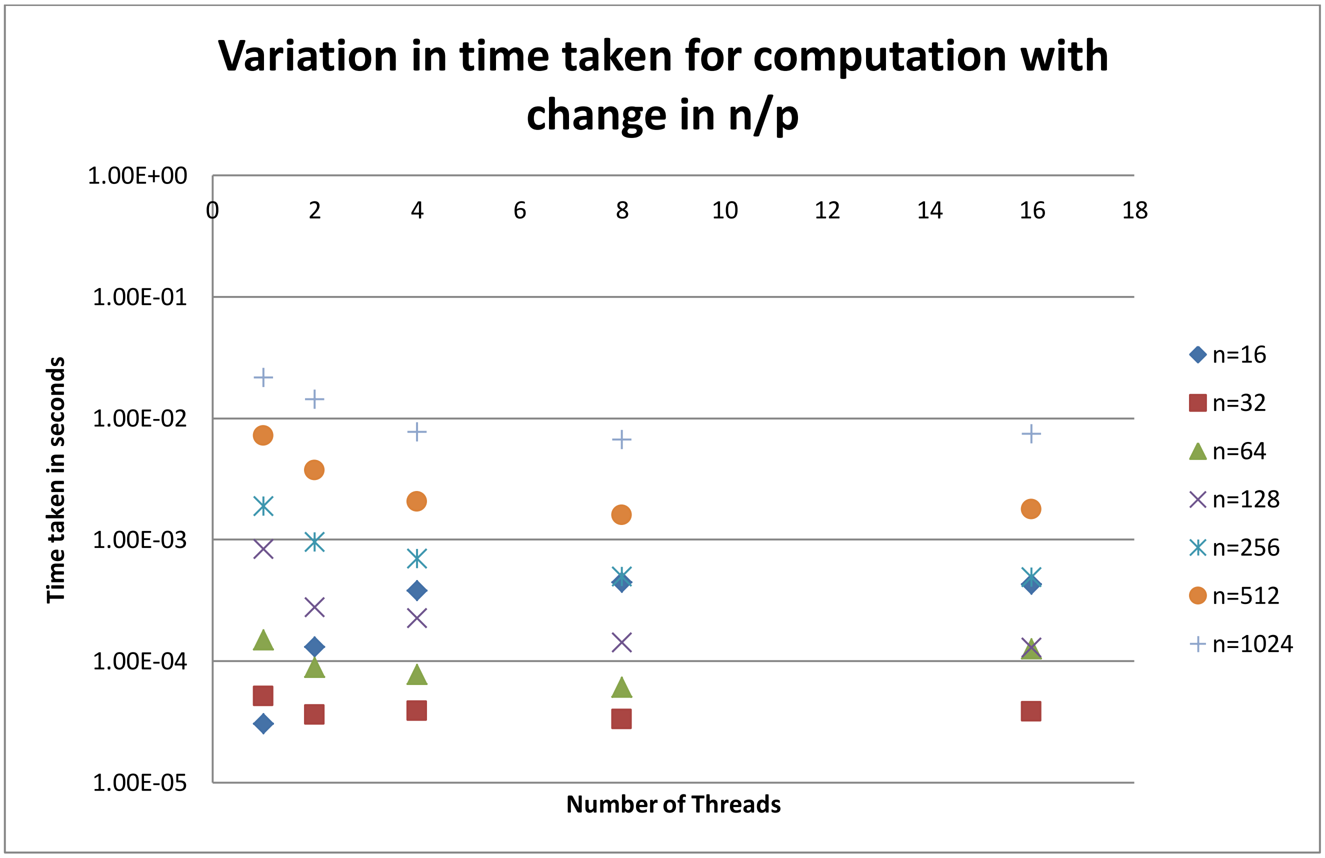


Figure 1.2 Variation in time with change in n/p

Now from the above results shown for n=16, 32, 64, 128, 256, 512, 1024 for varying number of threads, we may conclude following few aspects of matrix-vector parallel multiplication, when executed using OpenMP:

- It is verified that the matrix-vector multiplication parallel program written using OpenMPs works perfectly. When given code is executed for a matrix of order 16 multiplied to a 16*1 vector, with all their entries = 1, no matter what the number of threads is, the output is a 16*1 matrix with all entries = 16, which is true. Hence the practical results coincide with the theoretical results.

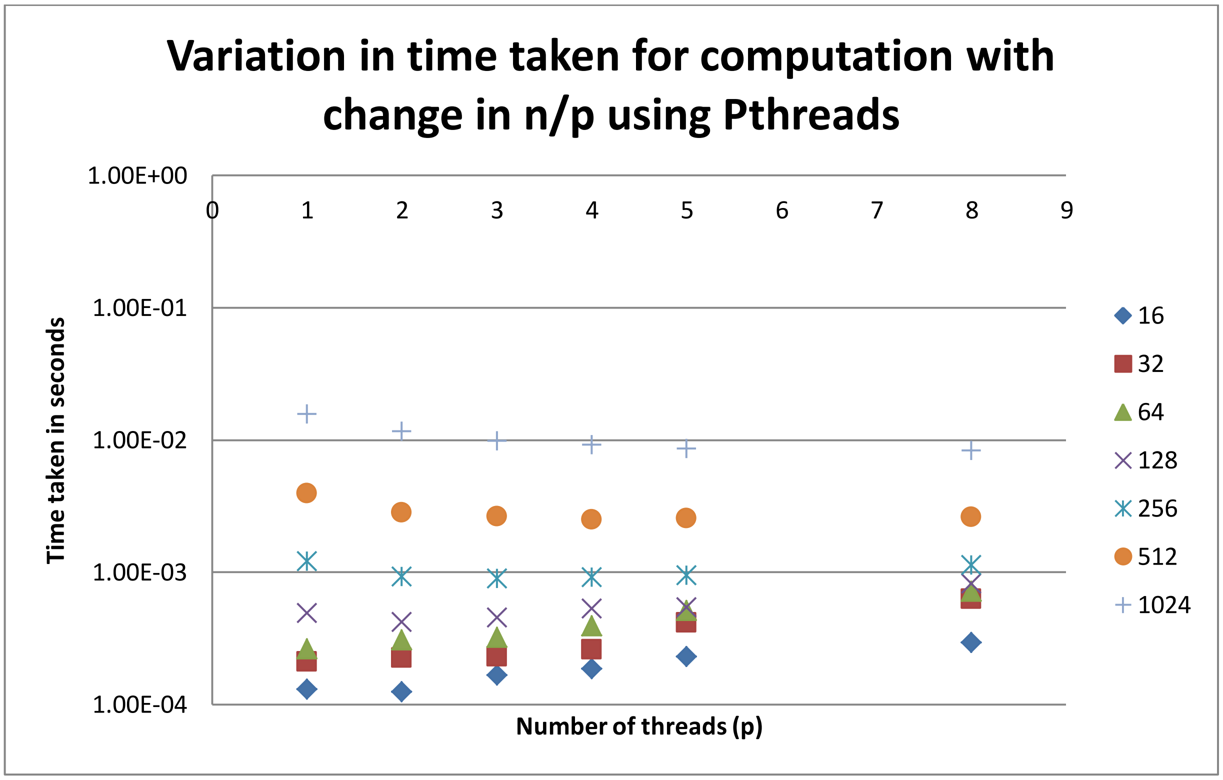
- Time taken for execution:
    - Theoretical Prediction: Time taken for calculation of matrix-vector product must decrease with increase in number of threads.
    - Practical Results:
    - It may be noticed that the time taken, in general, to compute the product increases with increase in number of threads for n=16 which does not match the theoretical prediction.
    - For all other values of n (=32, 64, 128, 256, 512, we notice a slight decline in time required for computation as the number of threads increase from 1 to 8 but then starts increasing (as number of threads increase to 16), which is somewhat consistent with the theoretical prediction [as long as the number of threads is less than 8].

- Reasons for Increase in time with increase in number of threads:
    - For smaller values of n (= 16), a possible reason for significant increase in time is the increase in number of overheads as new threads are formed (Thread formation needs some time [even after formation, information needs to be updated to the newly formed thread]). So it may be said that openmp, or for that matter, any other parallel programming technique must be used only when values involved in computation are very high, i.e., calculation time is more than communication time; and serial programming must be preferred for other cases.
    - For all other values of n, time taken in computation of the product decreases initially. But as number of threads become larger than 8, time starts increasing. The reason for this may be attributed to the fact that increasing the number of threads beyond an optimum level, increases the overheads involved in thread formation resulting in an overall increase in total time. So, we may conclude that number of threads to be formed must be optimal keeping track the order of the values involved.
- Also, we need not bother if the n/p ratio is an integer or not, which gives openmp architecture an edge over MPI or pthreads. OpenMP provides an easy method to create threads without requiring the programmer knowing about thread creation, synchronization and destruction. It incorporates a platform-independent set of compiler instructions resulting in letting user determine the best algorithm for maximum performance.

- Comparison with Pthreads:

Table 1.2: Tabular Data for variation in timing vs n/p using Pthreads:

n/p	1	2	3	4	5	8
16	0.00013	0.000125	0.000167	0.000186	0.000229	0.000293
32	0.00021	0.000226	0.000231	0.000261	0.000414	0.000629
64	0.000264	0.000308	0.000321	0.000391	0.000512	0.000721
128	0.000491	0.000419	0.000455	0.000533	0.000546	0.000818
256	0.001209	0.000924	0.000899	0.000912	0.000951	0.001133
512	0.003958	0.002834	0.002639	0.00251	0.002545	0.002624
1024	0.015771	0.011616	0.009821	0.009212	0.008591	0.008317



- From the two tables and graph plotted for changes in time taken vs n/p using Pthreads/OpenMP, our findings:

Table 1.3: Comparison between Pthreads and OpenMP

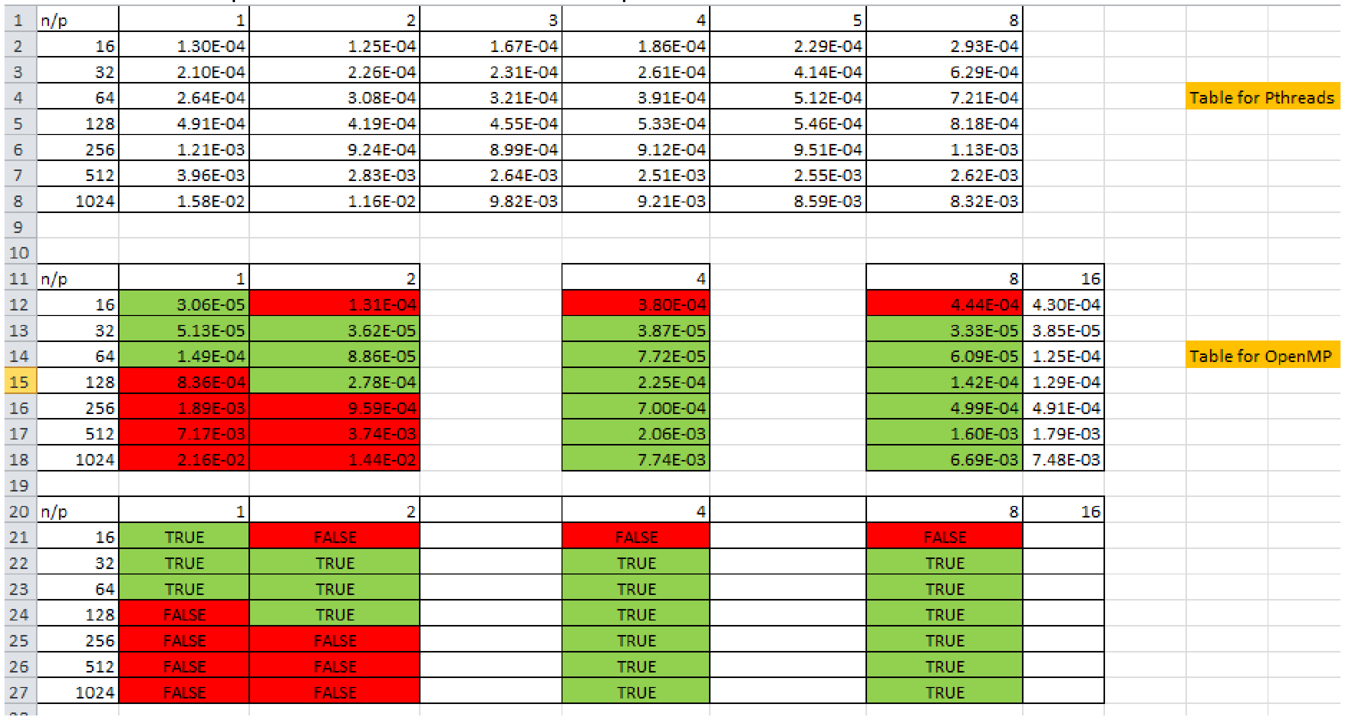


Table for Pthreads

n/p	1	2	3	4	5	8
16	1.30E-04	1.25E-04	1.67E-04	1.86E-04	2.29E-04	2.93E-04
32	2.10E-04	2.26E-04	2.31E-04	2.61E-04	4.14E-04	6.29E-04
64	2.64E-04	3.08E-04	3.21E-04	3.91E-04	5.12E-04	7.21E-04
128	4.91E-04	4.19E-04	4.55E-04	5.33E-04	5.46E-04	8.18E-04
256	1.21E-03	9.24E-04	8.99E-04	9.12E-04	9.51E-04	1.13E-03
512	3.96E-03	2.83E-03	2.64E-03	2.51E-03	2.55E-03	2.62E-03
1024	1.58E-02	1.16E-02	9.82E-03	9.21E-03	8.59E-03	8.32E-03

Table for OpenMP

n/p	1	2		4		8	16
16	3.06E-05	1.31E-04		3.80E-04		4.44E-04	4.30E-04
32	5.13E-05	3.62E-05		3.87E-05		3.33E-05	3.85E-05
64	1.49E-04	8.86E-05		7.72E-05		6.09E-05	1.25E-04
128	8.36E-04	2.78E-04		2.25E-04		1.42E-04	1.29E-04
256	1.89E-03	9.59E-04		7.00E-04		4.99E-04	4.91E-04
512	7.17E-03	3.74E-03		2.06E-03		1.60E-03	1.79E-03
1024	2.16E-02	1.44E-02		7.74E-03		6.69E-03	7.48E-03

n/p	1	2		4		8	16
16	TRUE	FALSE		FALSE		FALSE	
32	TRUE	TRUE		TRUE		TRUE	
64	TRUE	TRUE		TRUE		TRUE	
128	FALSE	TRUE		TRUE		TRUE	
256	FALSE	FALSE		TRUE		TRUE	
512	FALSE	FALSE		TRUE		TRUE	
1024	FALSE	FALSE		TRUE		TRUE	

The above comparison table consists of timing data obtained using Pthreads in the 1st table and timing data obtained using OpenMP in the 2nd table. The green colour (TRUE) indicates a decrease in timing when openMP is used while the red colour indicates an increased timing.

- Conclusions:
    - OpenMP performs better than pthreads for higher values of n (beyond 32 above) as the number of threads increase (more than 4).
    - Pthreads is more desirable for higher values of n (beyond 128 above) for smaller values of number of threads (less than 4).
- The real essence of parallelism is that *a large problem is divided into smaller ones* so that the smaller pieces can be solved concurrently. The pieces are mutually independent (to some degree at least), but they're still part of the larger problem, which is now being solved in parallel. OpenMP thus depicts true parallelism for higher number of threads (as desired in parallel programming) and higher order of calculations with timings far less than serial code, or pthreads, or MPI, giving it an edge over the contemporary parallel architectures.

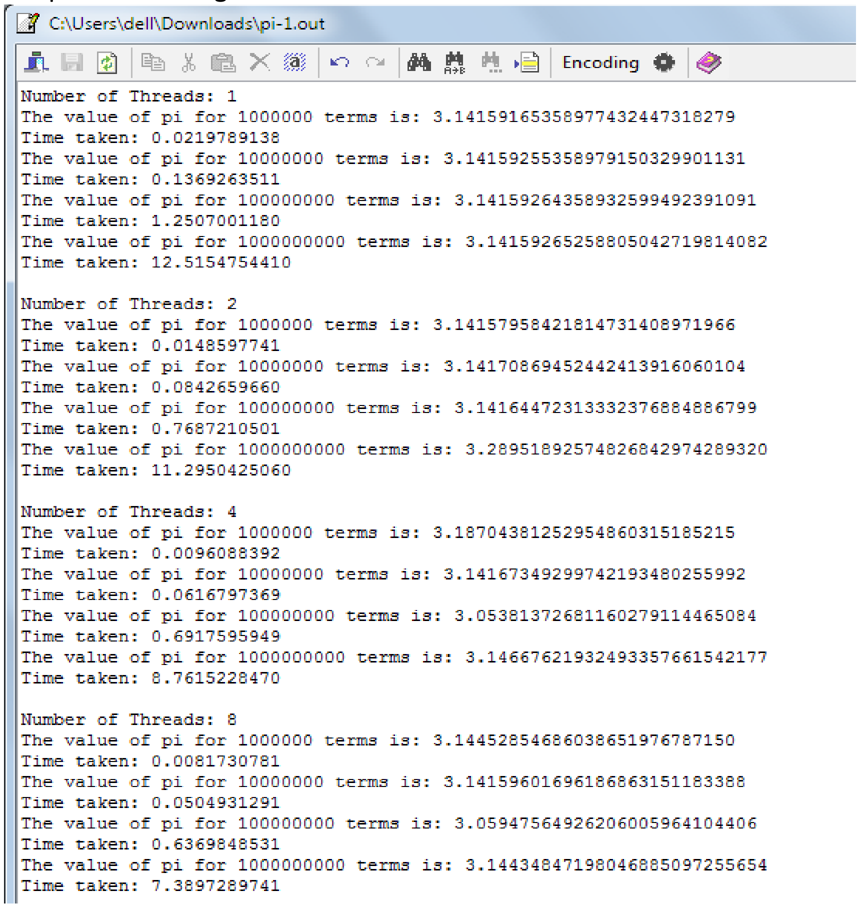
2. Calculation of pi using infinite series approximation

Method of Approach

This problem involving calculation of pi using infinite series approximation can be easily performed using openMP. After accepting the number of threads from the user, a loop is run in parallel on all the threads which multiply each fraction of the individual series by factor 1 or -1 depending on the position of the fraction in the infinite series and calculate local sums. Thus, the value of sum obtained from various threads is reduced to get the final value of sum which is then multiplied by 4 to get the value of pi.

Results and Discussion

Output of the Program:



```
Number of Threads: 1
The value of pi for 1000000 terms is: 3.1415916535897743244731827 9
Time taken: 0.0219789138
The value of pi for 10000000 terms is: 3.1415925535897915032990113 1
Time taken: 0.1369263511
The value of pi for 100000000 terms is: 3.1415926435893259949239109 1
Time taken: 1.2507001180
The value of pi for 1000000000 terms is: 3.1415926525880504271981408 2
Time taken: 12.5154754410

Number of Threads: 2
The value of pi for 1000000 terms is: 3.1415795842181473140897196 6
Time taken: 0.0148597741
The value of pi for 10000000 terms is: 3.1417086945244241391606010 4
Time taken: 0.0842659660
The value of pi for 100000000 terms is: 3.1416447231333237688488679 9
Time taken: 0.7687210501
The value of pi for 1000000000 terms is: 3.2895189257482684297428932 0
Time taken: 11.2950425060

Number of Threads: 4
The value of pi for 1000000 terms is: 3.1870438125295486031518521 5
Time taken: 0.0096088392
The value of pi for 10000000 terms is: 3.1416734929974219348025599 2
Time taken: 0.0616797369
The value of pi for 100000000 terms is: 3.0538137268116027911446508 4
Time taken: 0.6917595949
The value of pi for 1000000000 terms is: 3.1466762193249335766154217 7
Time taken: 8.7615228470

Number of Threads: 8
The value of pi for 1000000 terms is: 3.1445285468603865197678715 0
Time taken: 0.0081730781
The value of pi for 10000000 terms is: 3.1415960169618686315118338 8
Time taken: 0.0504931291
The value of pi for 100000000 terms is: 3.0594756492620600596410440 6
Time taken: 0.6369848531
The value of pi for 1000000000 terms is: 3.1443484719804688509725565 4
Time taken: 7.3897289741
```

Table showing variation in time with change in n/p

Powers of 10/p	1	2	4	8
6	0.021978914	0.01486	0.009608839	0.008173078
7	0.136926351	0.084266	0.061679737	0.050493129
8	1.250700118	0.768721	0.691759595	0.636984853
9	12.51547544	11.29504	8.761522847	7.389728974

Table showing variation in value of pi with change in n/p

Powers of 10/p	1	2	4	8
6	3.141591654	3.141579584	3.187043813	3.144528547
7	3.141592554	3.141708695	3.141673493	3.141596017
8	3.141592644	3.141644723	3.053813727	3.059475649
9	3.141592653	3.289518926	3.146676219	3.144348472

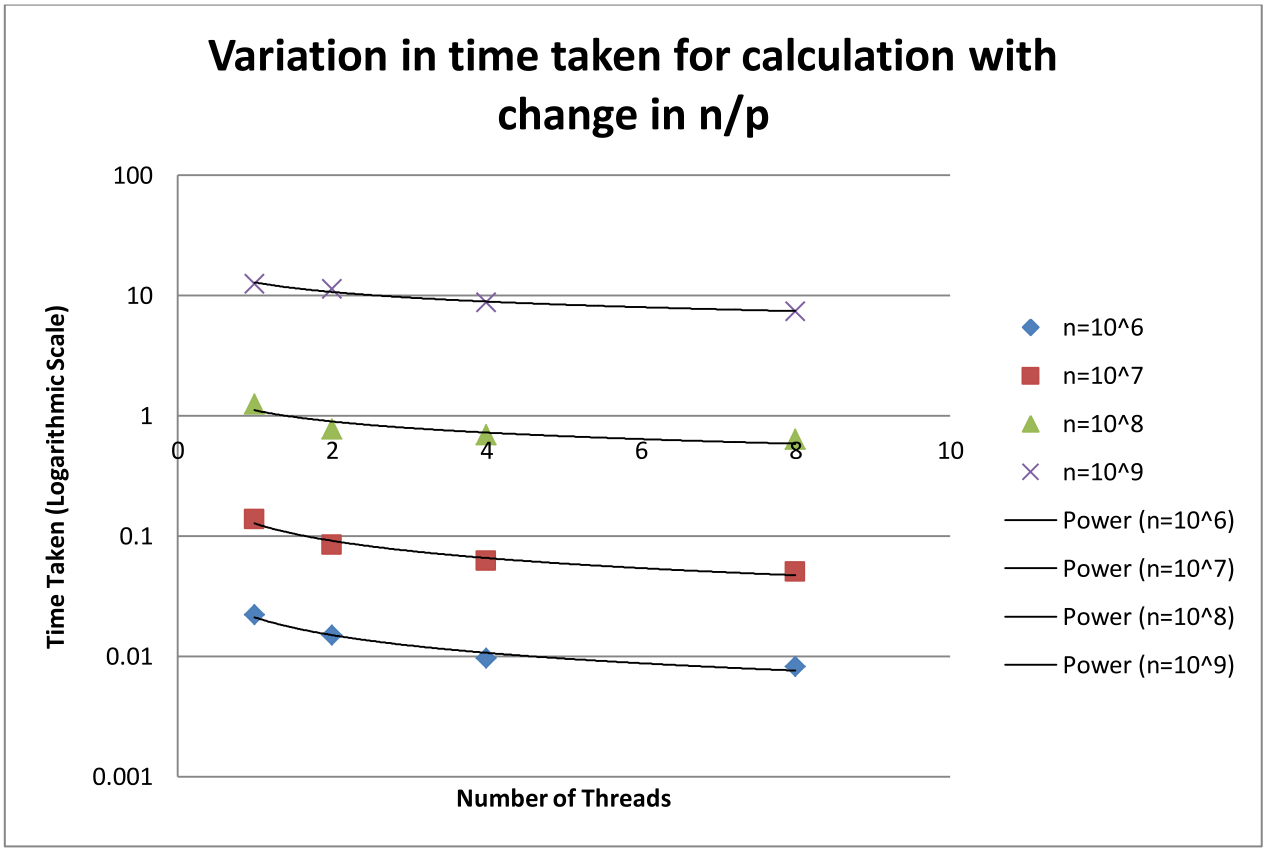


Figure 2.1 Variation in time taken with change in n/p

Table showing variation in % error with change in n/p

Powers of 10/p	1	2	4	8
6	3.1830989259E-05	4.1601102023E-04	1.4467553229E+00	9.3452385281E-02
7	3.1830988566E-06	3.6936976695E-03	2.5731982642E-03	1.0705945809E-04
8	3.1832483992E-07	1.6574250475E-03	2.7940900192E+00	2.6138654301E+00
9	3.1886375311E-08	4.7086394854E+00	1.6181492306E-01	8.7720423828E-02

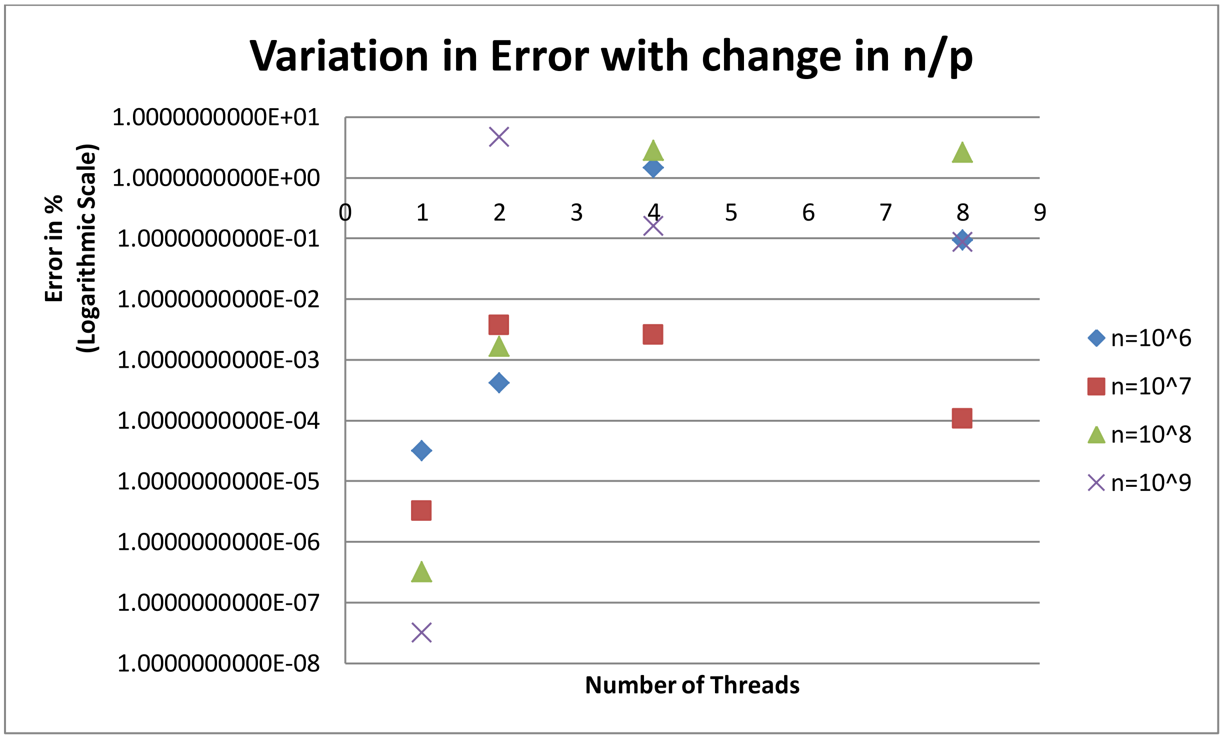


Figure 2.1 Variation in error (%) with change in n/p

- The time taken to compute the value of pi, keeping the order n constant, decreases with increase in number of threads, a result which coincides with theoretical predictions. It is because workload of each thread decreases as the total number of threads increase, hence decreasing the overall computation time.
- The time taken in computing the value of pi, keeping the number of threads constant, increases with increase in order of n, the reason being increased workload on each thread as they deal with higher orders values (as n increases).
- The value of pi is found to be accurate to 8 decimal places when order of n is 10^9 for p=1 thread.
- The percentage error in value of pi increases with increase in number of threads. This result totally out rules our theoretical prediction according to which, there must have been an exponential decrease in error. One possible reason for this absurd phenomenon might be False Sharing. False sharing occurs when two threads write to memory locations that are close to one another as to result on the same cache line. This results in the cache line constantly bouncing from core to core or CPU to CPU in multisocket systems and excess of cache coherency messages, which might result in wrong sum as we calculate the value of pi. A possible solution might be to add the values of sum obtained separately after the 'for' loop instead of reducing the value (when "# pragma omp parallel for reduction" is used)

- Comparison with pthreads:

Table showing variation in time with change in n/p

Powers of 10/p	1	2	3
6	0.013978	0.00709701	0.004951
7	0.0917051	0.048089	0.039263
8	0.803575	0.41325	0.275456

Table showing variation in value of pi with change in n/p

Powers of 10/p	1	2	3
6	3.141593654	3.141597654	3.141588654
7	3.14592754	3.141592754	3.141592254
8	3.141592664	3.141592704	3.141592614

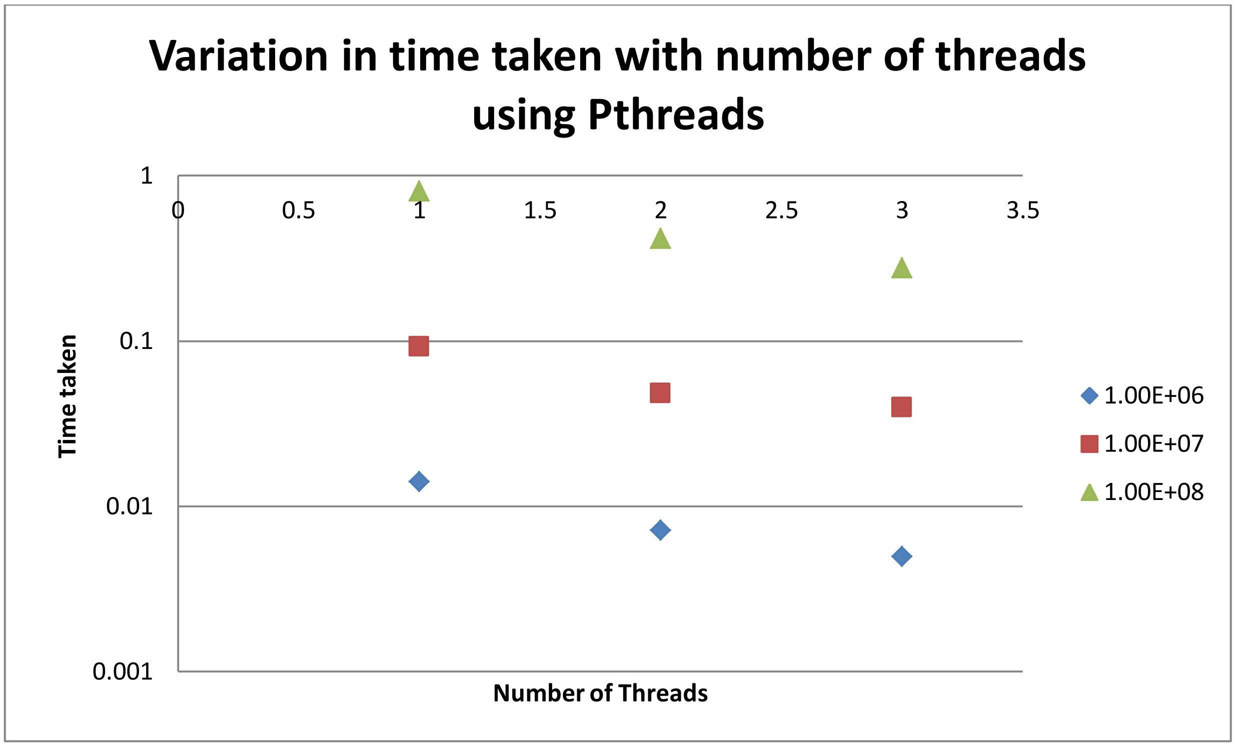


Figure 2.3 Variation in time taken with change in n/p

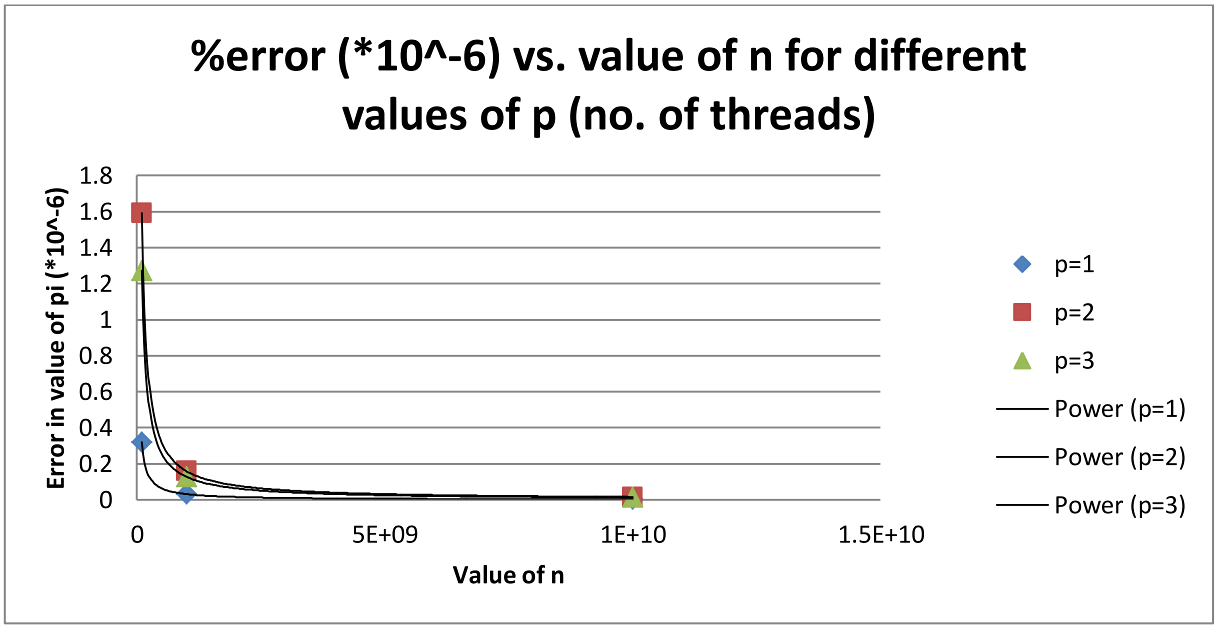


Figure 2.4 Error in value of pi (*10^-6) (Y-axis) vs Value of n (X-axis) for various no. of threads

- It may be concluded that the value of pi obtained using OpenMP is more accurate than when the value is calculated using pthreads when the number of threads is small. As the number of threads increase, pthreads is found to be more accurate. As for precision, both the methods offer the value of pi very precisely (where precision is seen as the number of places after the decimal while accuracy is the correctness of the value obtained using any method). [Error in value of pi decreases with increase in number of threads using pthreads while it is the reverse in case of OpenMP (due to False Sharing).
- Also, the time taken for calculation of value of pi using pthreads is less than the time needed to calculate using OpenMP. A possible reason might be the technique used in implementation of the code. Both Pthreads and OpenMP use shared memory architecture which saves them the valuable communication time (as in MPI). But the difference between them is that the thread formation and execution is controlled by user himself in case of pthreads while in case of OpenMP, threads are handled by compiler which forms and executes them by constantly interacting with the Operating System. Thus the values are first analyzed and then distributed to various threads in case of OpenMP, which involves computation. In pthreads, values to be executed by each thread are defined by user himself thus saving the precious computation time which goes into distributing the values in case of OpenMP.
- Essence: "Pthreads takes lesser time in execution but more time in coding; OpenMP takes more time in execution but lesser time in coding". Thus if overall time is considered (in making and executing the code, openMP is better than Pthreads.

- Monte-Carlo using OpenMP:

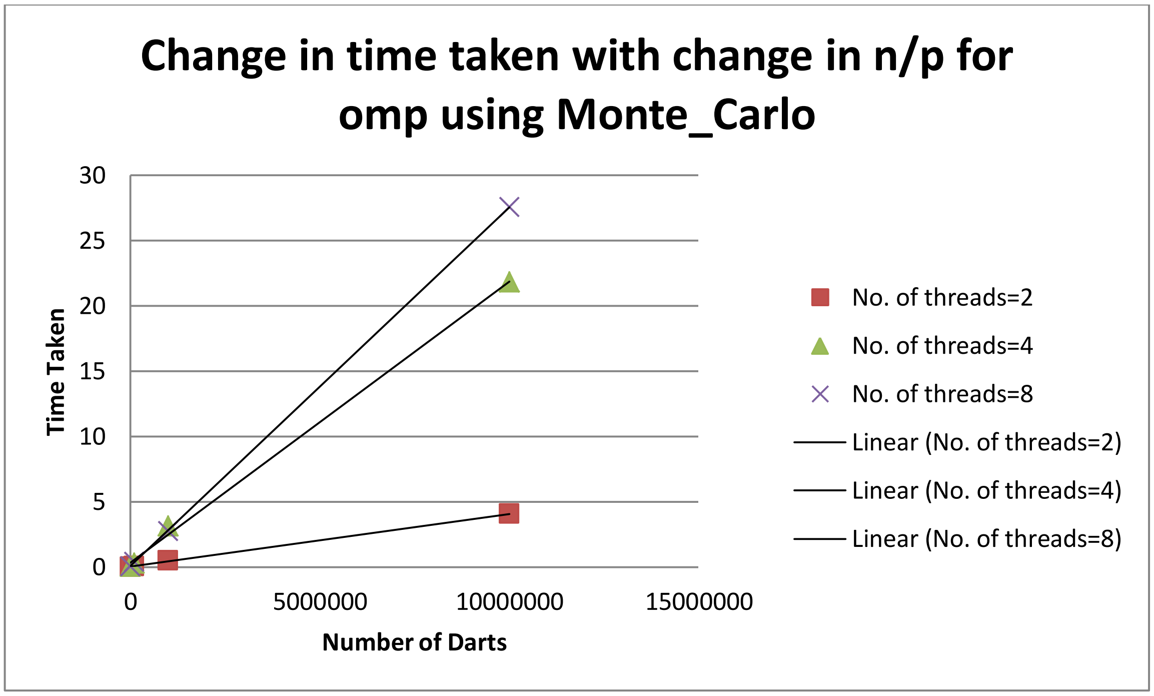


Figure 2.5 Variation in time taken with change in n/p

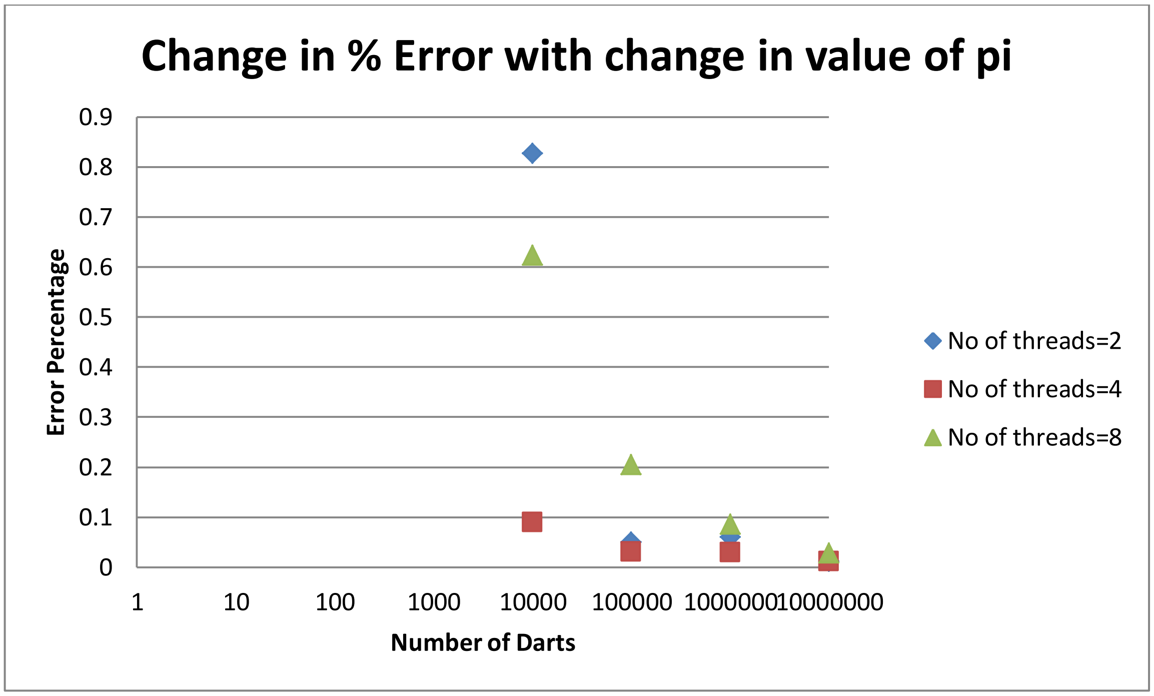


Figure 2.6 Variation in percentage error with change in n/p

- The time taken for execution for monte carlo (using omp) is quite higher than the time consumed when the same program is done using infinite series approximation.
- The percentage error is also found to be larger in case of monte-carlo method than the error percentage obtained using infinite series approximation.

3. Parallel Quadrature using Trapezoidal Rule for parallel numerical integration

Implementation of parallel numerical integration to estimate the volume bounded by two paraboloids using OpenMP.

Method of Approach

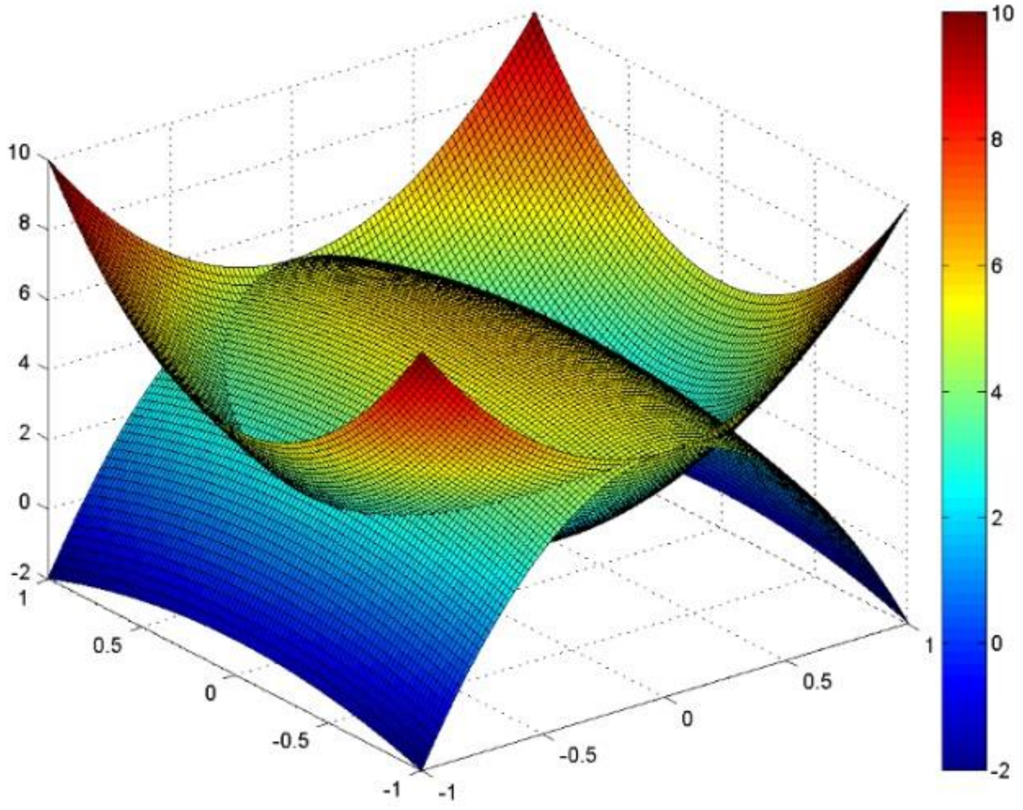


Figure 3.1 showing the region inscribed between the 2 curves: $z = 5(x^2 + y^2)$ and $z = 6 - 7x^2 - y^2$

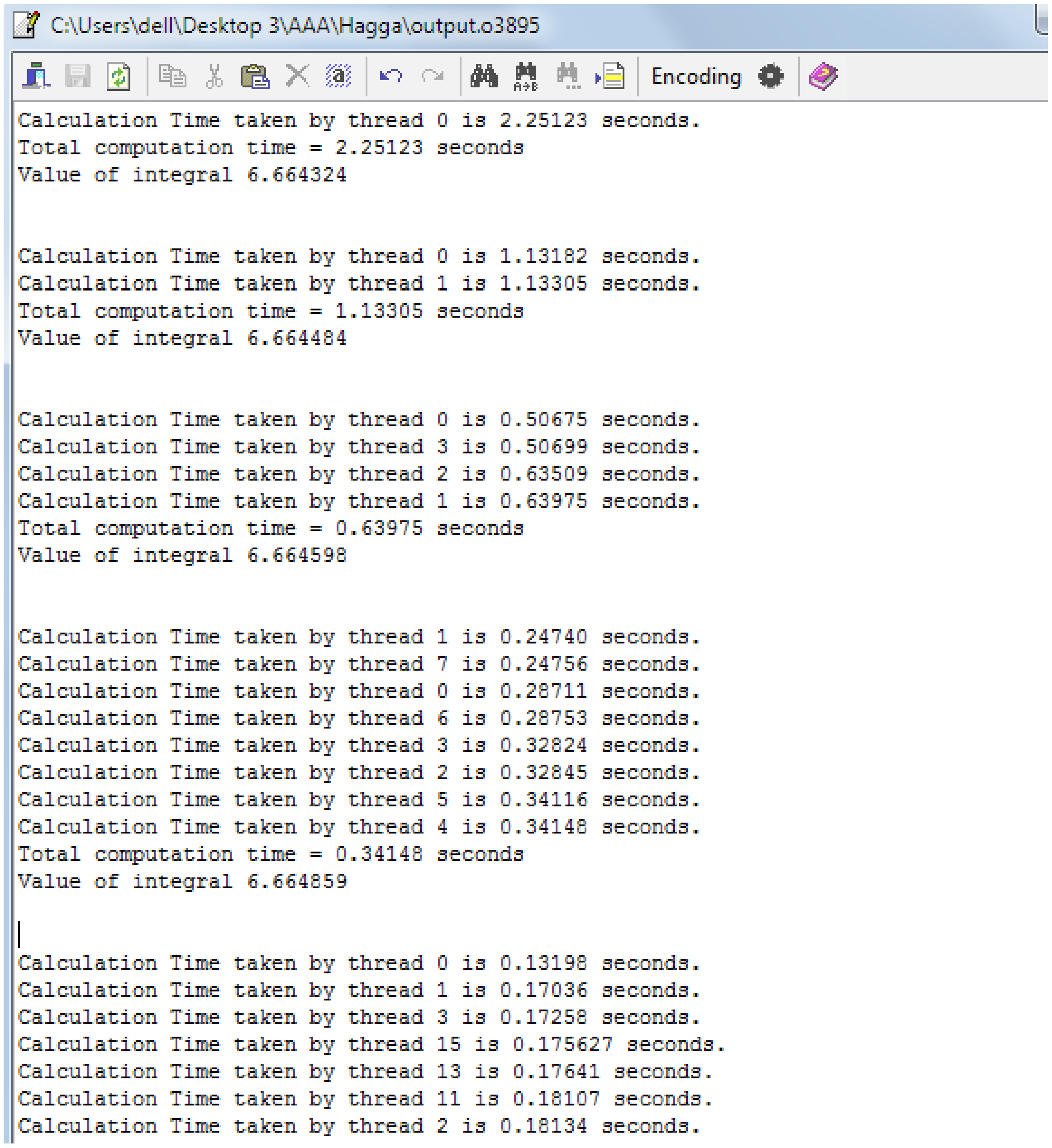
This problem about estimating the volume bounded by two paraboloids < and z_2 (shown in figure) had already been done in CP1.2 using MPI. Based on our experiments in assessing the number of strips, it was found that if n_x and n_y are 1000 each, the volume nearly comes as equal to the actual value. To calculate the above given inscribed volume using openMP, after accepting inputs about number of partitions and threads to be made from the user, the local integration regions are allocated to different threads, which after calculating their local volume of their region, get the total volume by reduction. The limits of integration for x and y variable can be known by putting $z_1 = z_2$, which gives us an equation of an ellipse in x- and y- plane. The equation thus obtained is actually the projection of the inscribed region (as shown in Figure above) in x-y plane.

After calculating the limits of integration, the x-axis is divided into equal parts (by dividing with the number of threads. Then the value of y is calculated for all values of x pertaining to each thread. If z_1 is found to be greater than z_2, no action is taken as that region lies outside the limits of integration, i.e., outside the inscribed volume. If $z_1 < z_2$, we find the value of the volume of the cuboid thus got. After calculating the local volume for all threads, we join all the threads in order to get total inscribed volume.

Results and Discussion

Output of the Program:

For nx=ny=1000;



```
Calculation Time taken by thread 0 is 2.25123 seconds.
Total computation time = 2.25123 seconds
Value of integral 6.664324


Calculation Time taken by thread 0 is 1.13182 seconds.
Calculation Time taken by thread 1 is 1.13305 seconds.
Total computation time = 1.13305 seconds
Value of integral 6.664484


Calculation Time taken by thread 0 is 0.50675 seconds.
Calculation Time taken by thread 3 is 0.50699 seconds.
Calculation Time taken by thread 2 is 0.63509 seconds.
Calculation Time taken by thread 1 is 0.63975 seconds.
Total computation time = 0.63975 seconds
Value of integral 6.664598


Calculation Time taken by thread 1 is 0.24740 seconds.
Calculation Time taken by thread 7 is 0.24756 seconds.
Calculation Time taken by thread 0 is 0.28711 seconds.
Calculation Time taken by thread 6 is 0.28753 seconds.
Calculation Time taken by thread 3 is 0.32824 seconds.
Calculation Time taken by thread 2 is 0.32845 seconds.
Calculation Time taken by thread 5 is 0.34116 seconds.
Calculation Time taken by thread 4 is 0.34148 seconds.
Total computation time = 0.34148 seconds
Value of integral 6.664859


Calculation Time taken by thread 0 is 0.13198 seconds.
Calculation Time taken by thread 1 is 0.17036 seconds.
Calculation Time taken by thread 3 is 0.17258 seconds.
Calculation Time taken by thread 15 is 0.175627 seconds.
Calculation Time taken by thread 13 is 0.17641 seconds.
Calculation Time taken by thread 11 is 0.18107 seconds.
Calculation Time taken by thread 2 is 0.18134 seconds.
```

Figure 3.2 shows the outputs as the number of threads is increased for p=1, 2, 4, 8, 16 threads

The value of the specified volume is 6.664859 as per the output of the program for nx=ny=1000 divisions when p=8 threads are used.

The value of the specified volume is 6.664324 as per the output of the program for nx=ny=1000 divisions when p=1 thread is used.

For nx=ny=10000;

p	Timings for volume integration using different number of threads															
1	2.25123															
2	1.13182	1.13305														
4	0.50675	0.50699	0.63509	0.63975												
8	0.24740	0.24756	0.28711	0.28753	0.32824	0.32845	0.34116	0.34148								
16	0.13198	0.17036	0.17258	0.17627	0.17641	0.18107	0.18134	0.18292	0.21380	0.21479	0.21670	0.22574	0.228053	0.242491	0.245039	0.260675

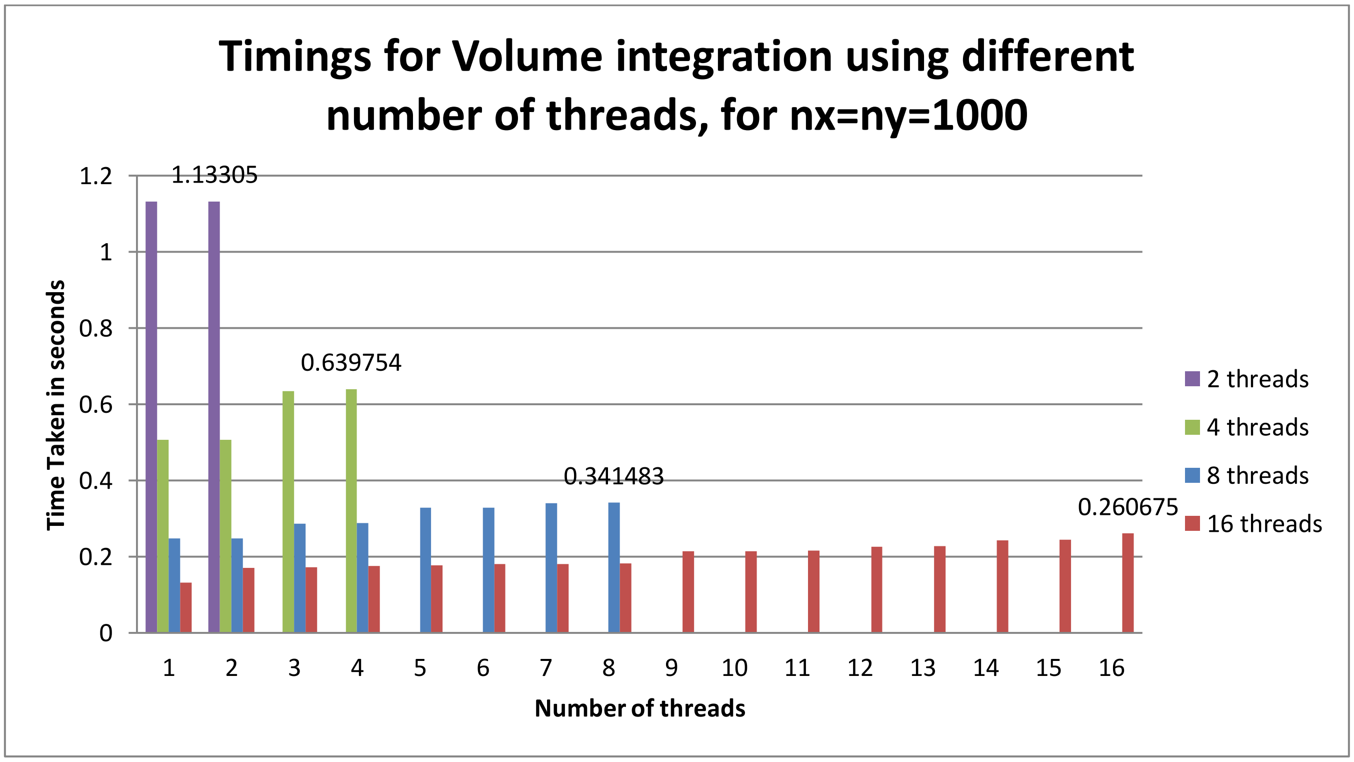


Figure 3.3 Histogram showing variation of time with change in number of threads

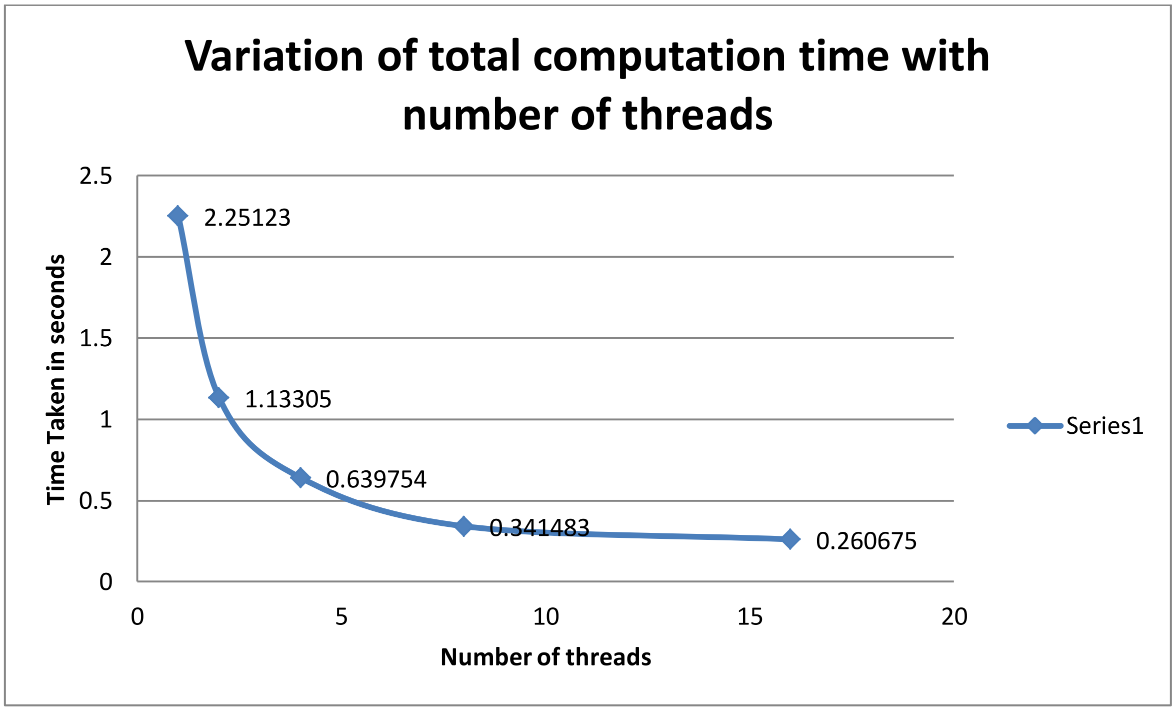


Figure 3.4 Variation of computation time with change in number of threads

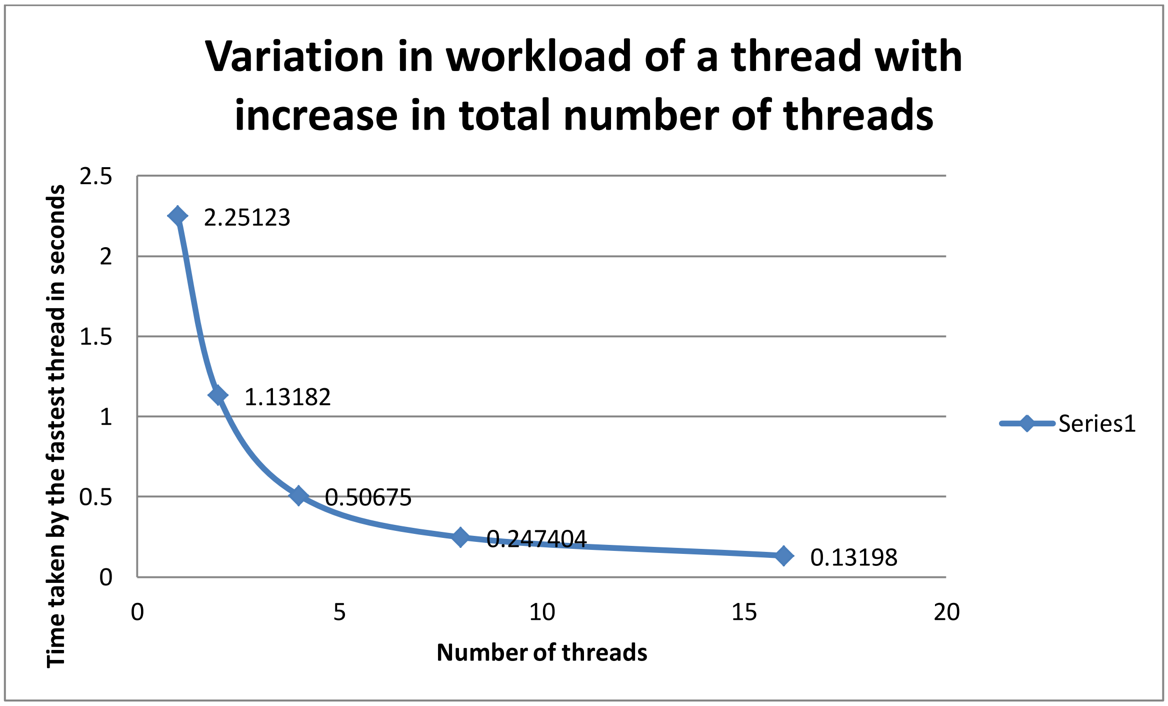


Figure 3.5 Variation in workload of a thread with change in number of threads

- The program for calculating the volume inscribed between two given paraboloids works correctly as the specified volume comes to be 6.665392 when 16 threads are used and 6.664324 when 1 thread is used. When calculated theoretically, the volume is found to be [3*pi/sqrt(2)] = 6.6643244. Hence the practical results coincide with the theoretical results.
- The above figure shows the time taken for finding the volume of the inscribed region between z_1 and z_2. It is clear from above histogram that as number of threads increase for

nx=ny= 1000, the total time taken in calculating the volume decreases. Figure 3.2 shows the computation time of the whole program.

- It may be noticed from Figure 3.3 that the workload on a thread decreases as the total number of threads in increased keeping the number of divisions nx and ny the same.
- From the histogram in Figure 3.1, it may be seen that time taken by each thread differs by a small amount for a particular value of nx and ny. For example, when 8 threads are used to execute a program, all of them take 8 different times to execute their share of the program. It is because when a program is run, the threads are allocated the values of local nx on a random basis. Now, the thread that gets its share of the program in the beginning(very small time difference between their creation) is the fastest as it has to process smaller values of x. The threads which get their share of values later, have to process larger values of x and hence take slightly more time.
- This exercise reasserts the fact that OpenMP takes lesser time in comparison to serial code for particular value of n. Moreover, as the threads increase, the overall computation time decreases as much of the work is then done in parallel.
- Attached is the code for the above question involving calculation of volume enclosed by two paraboloids using OpenMP.

4. Parallel Fast Fourier Transform

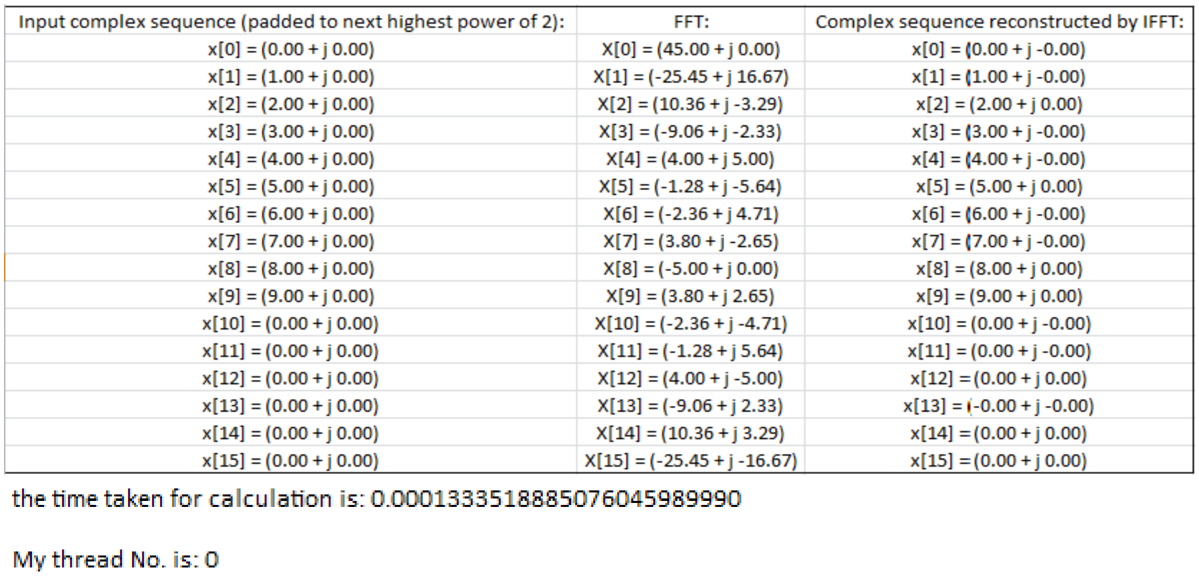
Parallelizing the serial fft (Fast Fourier Transform) code using shared memory implementation in OpenMP

Method of approach

To parallelize the Fast Fourier Transform code of a given complex number using the given serial code, we check each section of the program to see if it can be parallelized and make it parallel using different compiler directives if at all possible.
After parallelizing, we may profile the parallelized version of code using OmpP Profiler.\

Results and Discussion

Table 4.1: Output when number of threads = 1



Input complex sequence (padded to next highest power of 2):	FFT:	Complex sequence reconstructed by IFFT:
x[0] = (0.00 + j 0.00)	X[0] = (45.00 + j 0.00)	x[0] = (0.00 + j -0.00)
x[1] = (1.00 + j 0.00)	X[1] = (-25.45 + j 16.67)	x[1] = (1.00 + j -0.00)
x[2] = (2.00 + j 0.00)	X[2] = (10.36 + j -3.29)	x[2] = (2.00 + j 0.00)
x[3] = (3.00 + j 0.00)	X[3] = (-9.06 + j -2.33)	x[3] = (3.00 + j -0.00)
x[4] = (4.00 + j 0.00)	X[4] = (4.00 + j 5.00)	x[4] = (4.00 + j -0.00)
x[5] = (5.00 + j 0.00)	X[5] = (-1.28 + j -5.64)	x[5] = (5.00 + j 0.00)
x[6] = (6.00 + j 0.00)	X[6] = (-2.36 + j 4.71)	x[6] = (6.00 + j -0.00)
x[7] = (7.00 + j 0.00)	X[7] = (3.80 + j -2.65)	x[7] = (7.00 + j -0.00)
x[8] = (8.00 + j 0.00)	X[8] = (-5.00 + j 0.00)	x[8] = (8.00 + j 0.00)
x[9] = (9.00 + j 0.00)	X[9] = (3.80 + j 2.65)	x[9] = (9.00 + j 0.00)
x[10] = (0.00 + j 0.00)	X[10] = (-2.36 + j -4.71)	x[10] = (0.00 + j -0.00)
x[11] = (0.00 + j 0.00)	X[11] = (-1.28 + j 5.64)	x[11] = (0.00 + j -0.00)
x[12] = (0.00 + j 0.00)	X[12] = (4.00 + j -5.00)	x[12] = (0.00 + j 0.00)
x[13] = (0.00 + j 0.00)	X[13] = (-9.06 + j 2.33)	x[13] = (-0.00 + j -0.00)
x[14] = (0.00 + j 0.00)	X[14] = (10.36 + j 3.29)	x[14] = (0.00 + j 0.00)
x[15] = (0.00 + j 0.00)	X[15] = (-25.45 + j -16.67)	x[15] = (0.00 + j 0.00)

```
the time taken for calculation is: 0.00013335188850760459899990

My thread No. is: 0
```

Table 4.2: Output when number of threads = 2

Input complex sequence (padded to next highest power of 2):	FFT:	Complex sequence reconstructed by IFFT:
x[0] = (0.00 + j 0.00)	X[8] = (-5.00 + j 0.00)	x[8] = (8.00 + j 0.00)
x[1] = (1.00 + j 0.00)	X[9] = (3.80 + j 2.65)	x[9] = (9.00 + j 0.00)
x[2] = (2.00 + j 0.00)	X[10] = (-2.36 + j -4.71)	x[0] = (0.00 + j -0.00)
x[8] = (8.00 + j 0.00)	X[11] = (-1.28 + j 5.64)	x[1] = (1.00 + j -0.00)
x[9] = (9.00 + j 0.00)	X[12] = (4.00 + j -5.00)	x[2] = (2.00 + j 0.00)
x[10] = (0.00 + j 0.00)	X[0] = (45.00 + j 0.00)	x[3] = (3.00 + j -0.00)
x[11] = (0.00 + j 0.00)	X[1] = (-25.45 + j 16.67)	x[4] = (4.00 + j -0.00)
x[12] = (0.00 + j 0.00)	X[2] = (10.36 + j -3.29)	x[5] = (5.00 + j 0.00)
x[13] = (0.00 + j 0.00)	X[13] = (-9.06 + j 2.33)	x[6] = (6.00 + j -0.00)
x[14] = (0.00 + j 0.00)	X[14] = (10.36 + j 3.29)	x[7] = (7.00 + j -0.00)
x[15] = (0.00 + j 0.00)	X[15] = (-25.45 + j -16.67)	x[10] = (0.00 + j -0.00)
x[3] = (3.00 + j 0.00)	X[3] = (-9.06 + j -2.33)	x[11] = (0.00 + j -0.00)
x[4] = (4.00 + j 0.00)	X[4] = (4.00 + j 5.00)	x[12] = (0.00 + j 0.00)
x[5] = (5.00 + j 0.00)	X[5] = (-1.28 + j -5.64)	x[13] = (-0.00 + j -0.00)
x[6] = (6.00 + j 0.00)	X[6] = (-2.36 + j 4.71)	x[14] = (0.00 + j 0.00)
x[7] = (7.00 + j 0.00)	X[7] = (3.80 + j -2.65)	x[15] = (0.00 + j 0.00)

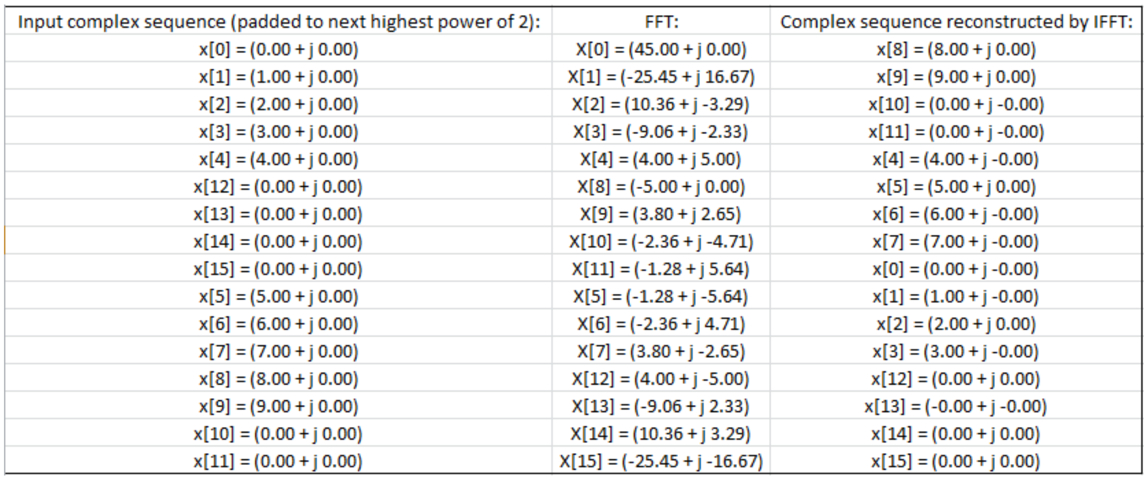
```
the time taken for calculation is: 0.0002880580723285675048828

My thread No. is: 1

My thread No. is: 0
```

Table 4.3: Output when number of threads = 3

Input complex sequence (padded to next highest power of 2):	FFT:	Complex sequence reconstructed by IFFT:
x[0] = (0.00 + j 0.00)	X[0] = (45.00 + j 0.00)	x[0] = (0.00 + j -0.00)
x[1] = (1.00 + j 0.00)	X[1] = (-25.45 + j 16.67)	x[1] = (1.00 + j -0.00)
x[2] = (2.00 + j 0.00)	X[2] = (10.36 + j -3.29)	x[2] = (2.00 + j 0.00)
x[3] = (3.00 + j 0.00)	X[3] = (-9.06 + j -2.33)	x[3] = (3.00 + j -0.00)
x[4] = (4.00 + j 0.00)	X[4] = (4.00 + j 5.00)	x[4] = (4.00 + j -0.00)
x[5] = (5.00 + j 0.00)	X[5] = (-1.28 + j -5.64)	x[5] = (5.00 + j 0.00)
x[12] = (0.00 + j 0.00)	X[12] = (4.00 + j -5.00)	x[12] = (0.00 + j 0.00)
x[13] = (0.00 + j 0.00)	X[13] = (-9.06 + j 2.33)	x[13] = (-0.00 + j -0.00)
x[14] = (0.00 + j 0.00)	X[14] = (10.36 + j 3.29)	x[14] = (0.00 + j 0.00)
x[15] = (0.00 + j 0.00)	X[15] = (-25.45 + j -16.67)	x[15] = (0.00 + j 0.00)
x[6] = (6.00 + j 0.00)	X[6] = (-2.36 + j 4.71)	x[6] = (6.00 + j -0.00)
x[7] = (7.00 + j 0.00)	X[7] = (3.80 + j -2.65)	x[7] = (7.00 + j -0.00)
x[8] = (8.00 + j 0.00)	X[8] = (-5.00 + j 0.00)	x[8] = (8.00 + j 0.00)
x[9] = (9.00 + j 0.00)	X[9] = (3.80 + j 2.65)	x[9] = (9.00 + j 0.00)
x[10] = (0.00 + j 0.00)	X[10] = (-2.36 + j -4.71)	x[10] = (0.00 + j -0.00)
x[11] = (0.00 + j 0.00)	X[11] = (-1.28 + j 5.64)	x[11] = (0.00 + j -0.00)

```
the time taken for calculation is: 0.0003334011416882276535034

My thread No. is: 0

My thread No. is: 1

My thread No. is: 2
```

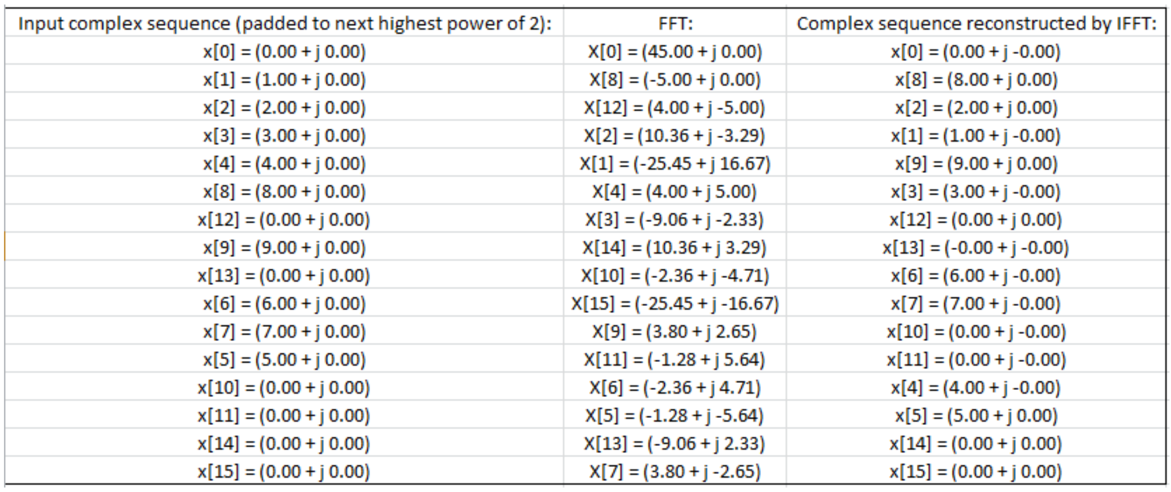
Table 4.4: Output when number of threads = 4



Input complex sequence (padded to next highest power of 2):	FFT:	Complex sequence reconstructed by IFFT:
x[0] = (0.00 + j 0.00)	X[0] = (45.00 + j 0.00)	x[8] = (8.00 + j 0.00)
x[1] = (1.00 + j 0.00)	X[1] = (-25.45 + j 16.67)	x[9] = (9.00 + j 0.00)
x[2] = (2.00 + j 0.00)	X[2] = (10.36 + j -3.29)	x[10] = (0.00 + j -0.00)
x[3] = (3.00 + j 0.00)	X[3] = (-9.06 + j -2.33)	x[11] = (0.00 + j -0.00)
x[4] = (4.00 + j 0.00)	X[4] = (4.00 + j 5.00)	x[4] = (4.00 + j -0.00)
x[12] = (0.00 + j 0.00)	X[8] = (-5.00 + j 0.00)	x[5] = (5.00 + j 0.00)
x[13] = (0.00 + j 0.00)	X[9] = (3.80 + j 2.65)	x[6] = (6.00 + j -0.00)
x[14] = (0.00 + j 0.00)	X[10] = (-2.36 + j -4.71)	x[7] = (7.00 + j -0.00)
x[15] = (0.00 + j 0.00)	X[11] = (-1.28 + j 5.64)	x[0] = (0.00 + j -0.00)
x[5] = (5.00 + j 0.00)	X[5] = (-1.28 + j -5.64)	x[1] = (1.00 + j -0.00)
x[6] = (6.00 + j 0.00)	X[6] = (-2.36 + j 4.71)	x[2] = (2.00 + j 0.00)
x[7] = (7.00 + j 0.00)	X[7] = (3.80 + j -2.65)	x[3] = (3.00 + j -0.00)
x[8] = (8.00 + j 0.00)	X[12] = (4.00 + j -5.00)	x[12] = (0.00 + j 0.00)
x[9] = (9.00 + j 0.00)	X[13] = (-9.06 + j 2.33)	x[13] = (-0.00 + j -0.00)
x[10] = (0.00 + j 0.00)	X[14] = (10.36 + j 3.29)	x[14] = (0.00 + j 0.00)
x[11] = (0.00 + j 0.00)	X[15] = (-25.45 + j -16.67)	x[15] = (0.00 + j 0.00)

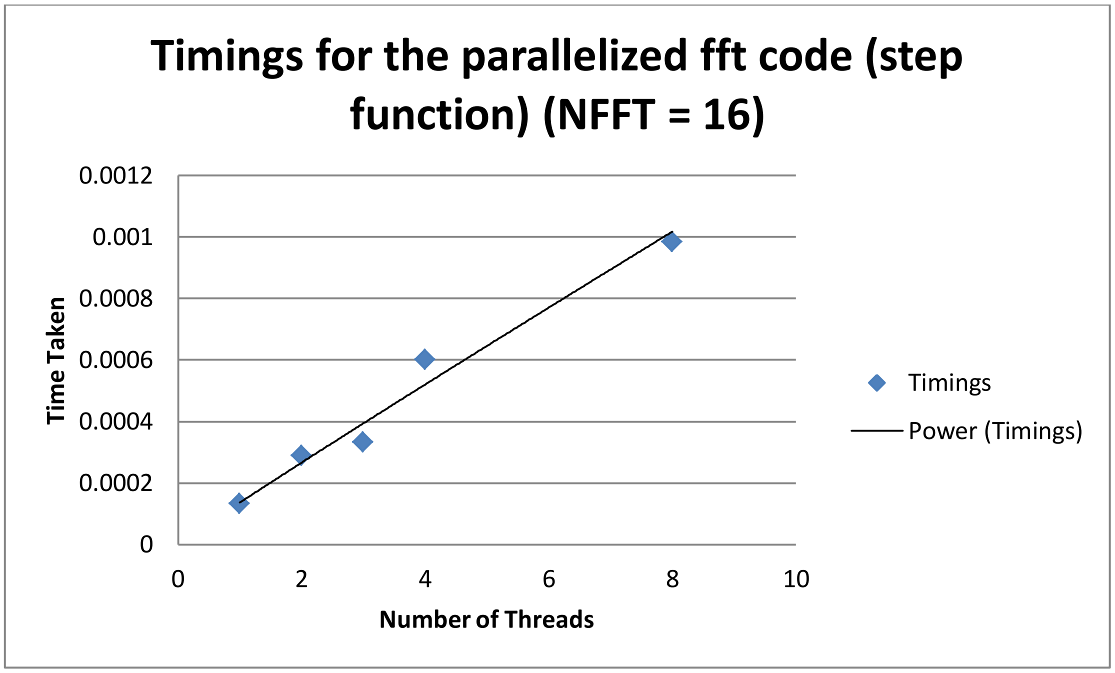
```
the time taken for calculation is: 0.0006016369443386793136597

My thread No. is: 0

My thread No. is: 1

My thread No. is: 3

My thread No. is: 2
```

Table 4.2: Output when number of threads = 8



Input complex sequence (padded to next highest power of 2):	FFT:	Complex sequence reconstructed by IFFT:
x[0] = (0.00 + j 0.00)	X[0] = (45.00 + j 0.00)	x[0] = (0.00 + j -0.00)
x[1] = (1.00 + j 0.00)	X[8] = (-5.00 + j 0.00)	x[8] = (8.00 + j 0.00)
x[2] = (2.00 + j 0.00)	X[12] = (4.00 + j -5.00)	x[2] = (2.00 + j 0.00)
x[3] = (3.00 + j 0.00)	X[2] = (10.36 + j -3.29)	x[1] = (1.00 + j -0.00)
x[4] = (4.00 + j 0.00)	X[1] = (-25.45 + j 16.67)	x[9] = (9.00 + j 0.00)
x[8] = (8.00 + j 0.00)	X[4] = (4.00 + j 5.00)	x[3] = (3.00 + j -0.00)
x[12] = (0.00 + j 0.00)	X[3] = (-9.06 + j -2.33)	x[12] = (0.00 + j 0.00)
x[9] = (9.00 + j 0.00)	X[14] = (10.36 + j 3.29)	x[13] = (-0.00 + j -0.00)
x[13] = (0.00 + j 0.00)	X[10] = (-2.36 + j -4.71)	x[6] = (6.00 + j -0.00)
x[6] = (6.00 + j 0.00)	X[15] = (-25.45 + j -16.67)	x[7] = (7.00 + j -0.00)
x[7] = (7.00 + j 0.00)	X[9] = (3.80 + j 2.65)	x[10] = (0.00 + j -0.00)
x[5] = (5.00 + j 0.00)	X[11] = (-1.28 + j 5.64)	x[11] = (0.00 + j -0.00)
x[10] = (0.00 + j 0.00)	X[6] = (-2.36 + j 4.71)	x[4] = (4.00 + j -0.00)
x[11] = (0.00 + j 0.00)	X[5] = (-1.28 + j -5.64)	x[5] = (5.00 + j 0.00)
x[14] = (0.00 + j 0.00)	X[13] = (-9.06 + j 2.33)	x[14] = (0.00 + j 0.00)
x[15] = (0.00 + j 0.00)	X[7] = (3.80 + j -2.65)	x[15] = (0.00 + j 0.00)

```
the time taken for calculation is: 0.0009842999279499053955078

My thread No. is: 0

My thread No. is: 6

My thread No. is: 2

My thread No. is: 4

My thread No. is: 7

My thread No. is: 5

My thread No. is: 1
```



- Attached is the OmpP Profiler usage statistics. It seems like the profiler displays time taken only upto two decimal places which may be the reason for it to show 0.00 for most of the readings.

- We may see that the time taken for calculation increases with increase in number of threads. It is because the values under consideration are very small (Nx = 10 and NFFT = 16) which makes the use of multiple threads for this case inefficient. With increase in number of threads, the overheads involved in their creation increase, resulting in an overall increase in total time taken.

Acknowledgement

I have taken efforts in this project. However, it would not have been possible without the kind support and help I received from Prof. Anand Sengupta and Prof. Murali Damodaran. I would like to extend my sincere thanks to him. I am highly indebted to his guidance and constant supervision as well as for providing necessary information regarding the project & also for his support in completing the project.
My thanks and appreciations also go to my friends Ravi, Parth and Nishant in verifying the correctness of project codes which we developed individually and people who have willingly helped me out with their abilities.