INDIAN INSTITUTE OF TECHNOLOGY GANDHINAGAR

ES 611: Algorithms on Advanced Computer Architectures

Report on Computational Lab Project 2.1
by
Shashank Heda
March 8th, 2013

ABSTRACT

High performance applications on shared memory machines have typically been written in a coarse grained style, with one heavyweight thread per processor. In comparison, programming with a large number of lightweight, parallel threads has several advantages, including simpler coding for programs with irregular and dynamic parallelism, and better adaptability to a changing number of processors. The programmer can easily express a new thread to execute each individual parallel task; the implementation dynamically creates and schedules these threads onto the processors, and effectively balances the load.
In this project, we study the performance of a native, lightweight POSIX threads (Pthreads) library. To evaluate this Pthreads implementation, we use a set of parallel programs that dynamically create a large number of threads. The results indicate that the rich functionality and standard API of Pthreads can be combined with the advantages of dynamic, lightweight threads to result in high performance. Moreover, Pthreads is found to be more efficient than parallel programs based on MPI, all credits to its shared memory architecture.

1. Parallel Matrix-Vector Multiplication

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

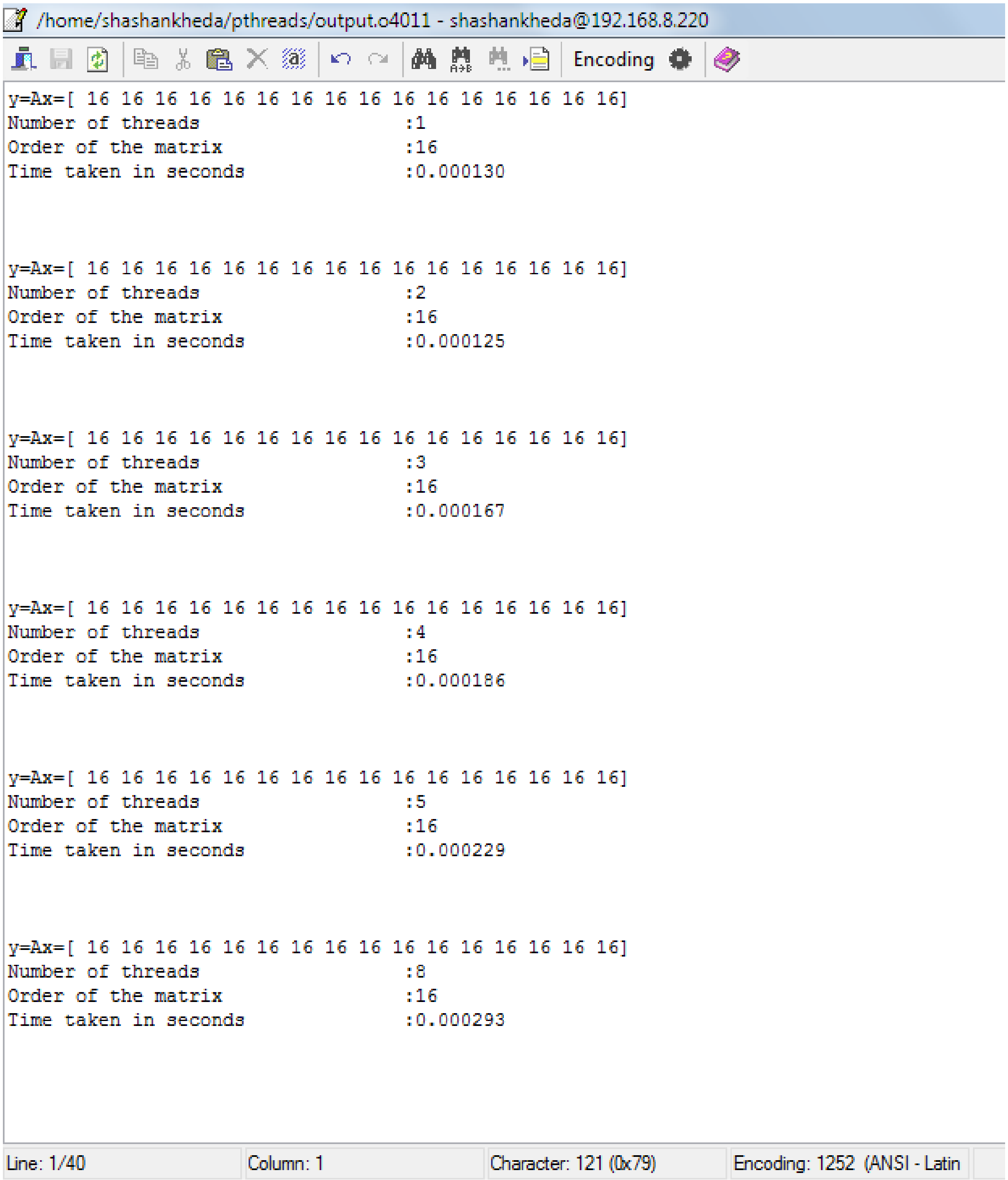
Method of approach

To carry out matrix-vector multiplication, y=Ax, where x is goven 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 the inputs are received and the matrix, vector are initialized, we need to allocate and memory and then create pthreads based on the number of threads we have to operate on. Then follows the usual sequence of joining the pthread handles and freeing up of memory at the end of the main() function. The external function for Matrix-Vector multiplication consists of statements which multiply each row of the matrix with column of vector x and store the result in vector y using two for loops for the same.

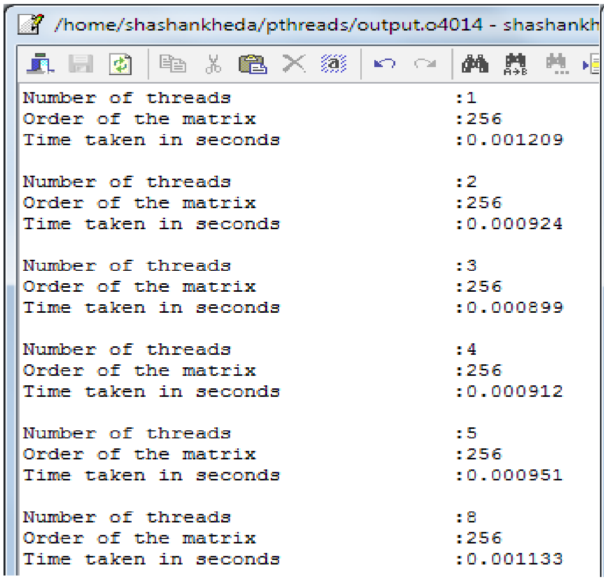
Results and Discussion

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

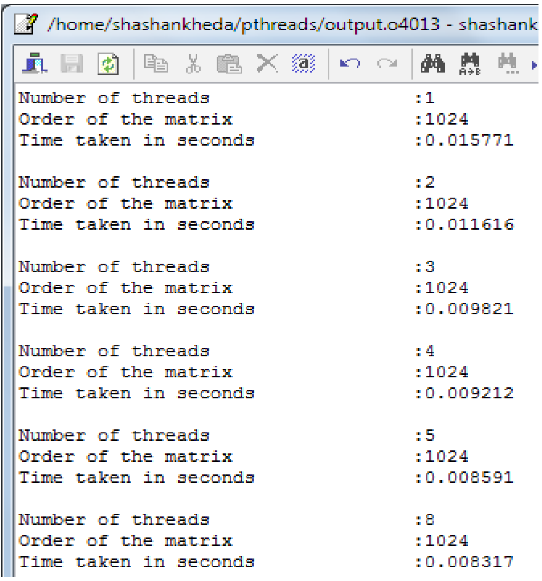

/home/shashankheda/pthreads/output.o4011 - shashankheda@192.168.8.220
Encoding
Line: 1/40
Column: 1
Character: 121 (0x79)
Encoding: 1252 (ANSI - Latin

```
y=Ax=[ 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16]
Number of threads                  :1
Order of the matrix                :16
Time taken in seconds              :0.000130



y=Ax=[ 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16]
Number of threads                  :2
Order of the matrix                :16
Time taken in seconds              :0.000125



y=Ax=[ 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16]
Number of threads                  :3
Order of the matrix                :16
Time taken in seconds              :0.000167



y=Ax=[ 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16]
Number of threads                  :4
Order of the matrix                :16
Time taken in seconds              :0.000186



y=Ax=[ 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16]
Number of threads                  :5
Order of the matrix                :16
Time taken in seconds              :0.000229



y=Ax=[ 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16]
Number of threads                  :8
Order of the matrix                :16
Time taken in seconds              :0.000293
```

Figure 1.1 Output of the Program for matrix-vector multiplication for n=16



```
Number of threads                   :1
Order of the matrix                 :256
Time taken in seconds               :0.001209

Number of threads                   :2
Order of the matrix                 :256
Time taken in seconds               :0.000924

Number of threads                   :3
Order of the matrix                 :256
Time taken in seconds               :0.000899

Number of threads                   :4
Order of the matrix                 :256
Time taken in seconds               :0.000912

Number of threads                   :5
Order of the matrix                 :256
Time taken in seconds               :0.000951

Number of threads                   :8
Order of the matrix                 :256
Time taken in seconds               :0.001133
```



```
Number of threads                   :1
Order of the matrix                 :1024
Time taken in seconds               :0.015771

Number of threads                   :2
Order of the matrix                 :1024
Time taken in seconds               :0.011616

Number of threads                   :3
Order of the matrix                 :1024
Time taken in seconds               :0.009821

Number of threads                   :4
Order of the matrix                 :1024
Time taken in seconds               :0.009212

Number of threads                   :5
Order of the matrix                 :1024
Time taken in seconds               :0.008591

Number of threads                   :8
Order of the matrix                 :1024
Time taken in seconds               :0.008317
```

n/p	1	2	3	4	5	8
16	0.000130	0.000125	0.000167	0.000186	0.000229	0.000293
32	0.000210	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

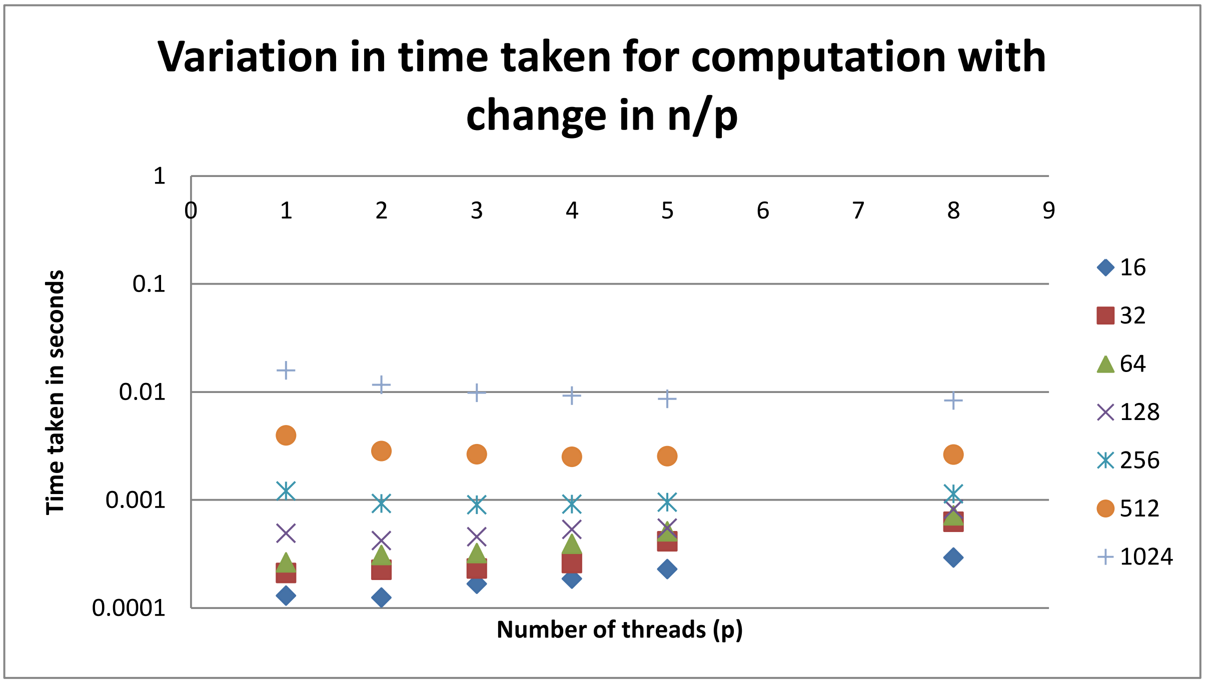
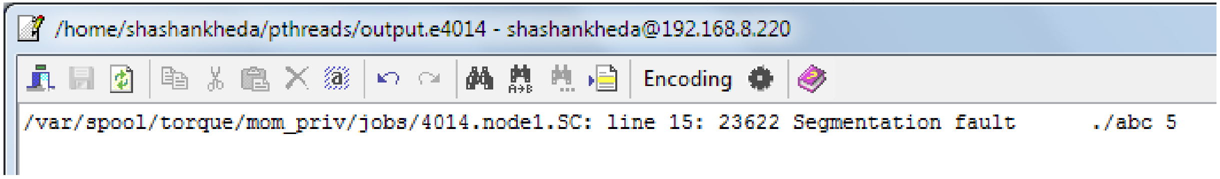


Figure 1.2 Variation in time with change in n/p

Now from the above results shown for n=16, 256, 1024 we may conclude following few aspects of matrix-vector parallel multiplication:

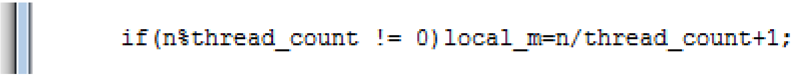
- The matrix-vector multiplication parallel program written using pthreads is executing correctly as shown in output of Figure 1. For a matrix of order 16, when multiplied to a 16*1 vector, all individual entries of matrix and the vector being 1, the output is a 16*1 matrix with all entries = 16, which 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, 32, 64, 128 which does not match the theoretical prediction.
  - Then for n=256, 512, we notice a slight decline in time required for computation which is not quite clear as it first decreases (as number of threads increase), but then starts increasing (as number of threads increases beyond 3/4 in the two cases), which is somewhat consistent with the theoretical prediction as long as the number of threads is less than 4.
  - For n=1024, a clear and significant decrease in time can be noticed from the results, which totally matches our theoretical prediction.
- Reasons for Increase in time with increase in number of threads:
  - For smaller values of n = 16, 32, 64, 128; a possible reason for significant increase in time is observed since the overheads also increase 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 pthreads or for that matter, any other parallel programming technique must be used only when values involved in computation are very high, and serial programming must be preferred for such cases.
  - For n=256, 512, time taken in computation of the product decreases initially. But as number of threads become larger than 3/4, time starts increasing. It is because of the same reason as mentioned above. 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 of the order of values involved.
  - For n=1024, the practical results follow the theoretical predictions as with such a big matrix (order = 1024*1024), the time lapsed in making new threads (overheads) is not as significant as the time taken in calculation.
- Also, the program does not work correctly if the n/p ratio is not an integer and we get segmentation fault errors as shown below. If not segmentation fault, the output vector contains many zero entries depending upon the order of the matrix. For example, if we provide a 5*5 matrix to be multiplied to a 5*1 vector using 3 threads, the output vector would be [5 5 5 0 0]. It is because the first three rows get allocated to the three different threads while the other two rows aren't allocated to any thread (equal load distribution).


/home/shashankheda/pthreads/output.e4014 - shashankheda@192.168.8.220
Encoding

```
/var/spool/torque/mom_priv/jobs/4014.node1.SC: line 15: 23622 Segmentation fault      ./abc 5
```

- To get rid of such errors caused because of unequal distribution of values, we would change the local values whenever n/p is not an integer as shown below.



```
if(n%thread_count != 0)local_m=n/thread_count+1;
```

- When the value of matrix-vector multiplication product is calculated using MPI (row-wise striped partitioning), the timings are found to be larger than when done using pthreads. The reason for more efficiency of pthreads is because of its dependency on shared memory computer architecture. This results in total absence of the invaluable communication time in case of Pthreads which makes it better than MPI based parallel programming architecture.
- The parallel codes Q1.c and Q1comparisonMPI.c are attached which have been used to analyse data for parallel matrix-vector multiplication.

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 pthreads. After creation of pthreads directed towards execution of the infinite series, we create a fucntion Thread_sum. This function decides the domain of values for each thread. After equally dividing the domain of values of x to all threads, we multiply each fraction by factor 1 or -1 depending on the position of the fraction in the infinite series.

Results and Discussion

Variation in Time taken for calculation with change in n/p

n/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

Variation in value of pi with change in n/p

n/p	1	2	3
6	3.141593654	3.141597654	3.141588654
7	3.145927540	3.141592754	3.141592254
8	3.141592664	3.141592704	3.141592614

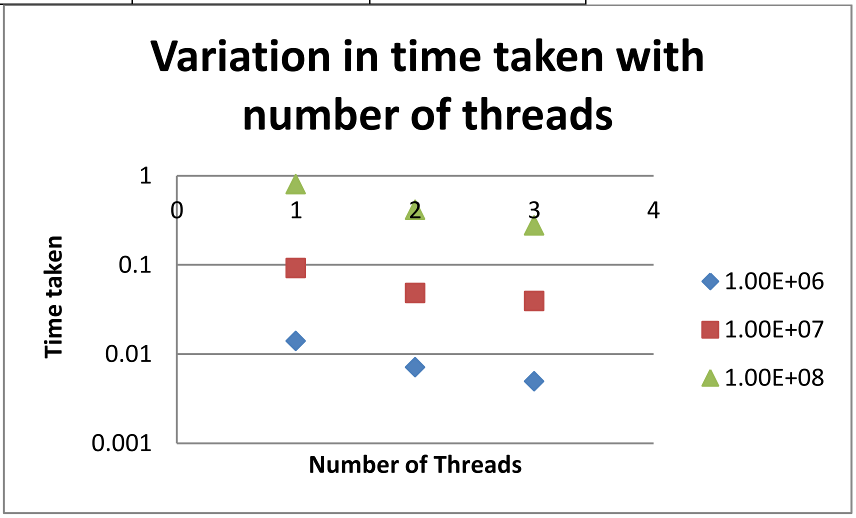


Figure 2.1 Variation in time taken with number of threads

- It can be concluded that the value of pi obtained using infinite series approximation is more accurate than when the value is calculated using MPI. 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)
- It can be seen that as the value of n increases, the value of pi becomes more accurate for a particular number of threads.
- As number of threads increase, the value of pi increases for a particular value of n.
- The practical predictions follow the theoretical predictions as shown in Figure 2.1 which show the variation of time taken in computation with change in n/p.

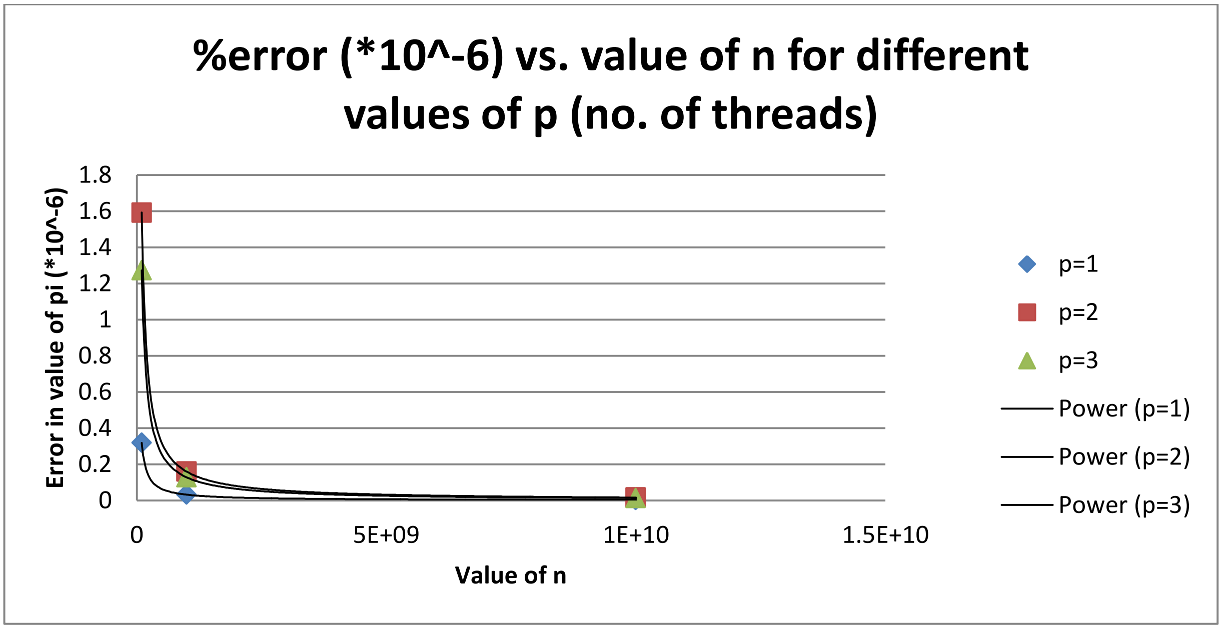


Figure 2.2 Error in value of pi (*10^-6) (Y-axis) vs Value of n (X-axis) for various no. of threads

- With increase in value of n, the error in the value of pi tends to zero for a particular value of p, i.e., as the value of n increases for p=1,2 or 3, the error in the value of pi tends to 0. Hence the value obtained is more precise.
- The timings obtained when the parallel program is run using pthreads are lesser than when the mode of implementation of parallelism is MPI. It is because pthread uses shared memory architecture. This gets us to the biggest advantage offered with pthreads which is the absence of communication time. The time consumed in pthreads apart from the natural computation time involves a very little time in making of new threads (overheads).
- Attached are the codes for infinite series approximation using pthreads and calculation using monte-carlo pthreads method.

3. Parallel Quadrature using Trapezoidal Rule for parallel numerical integration

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

Method of Approach

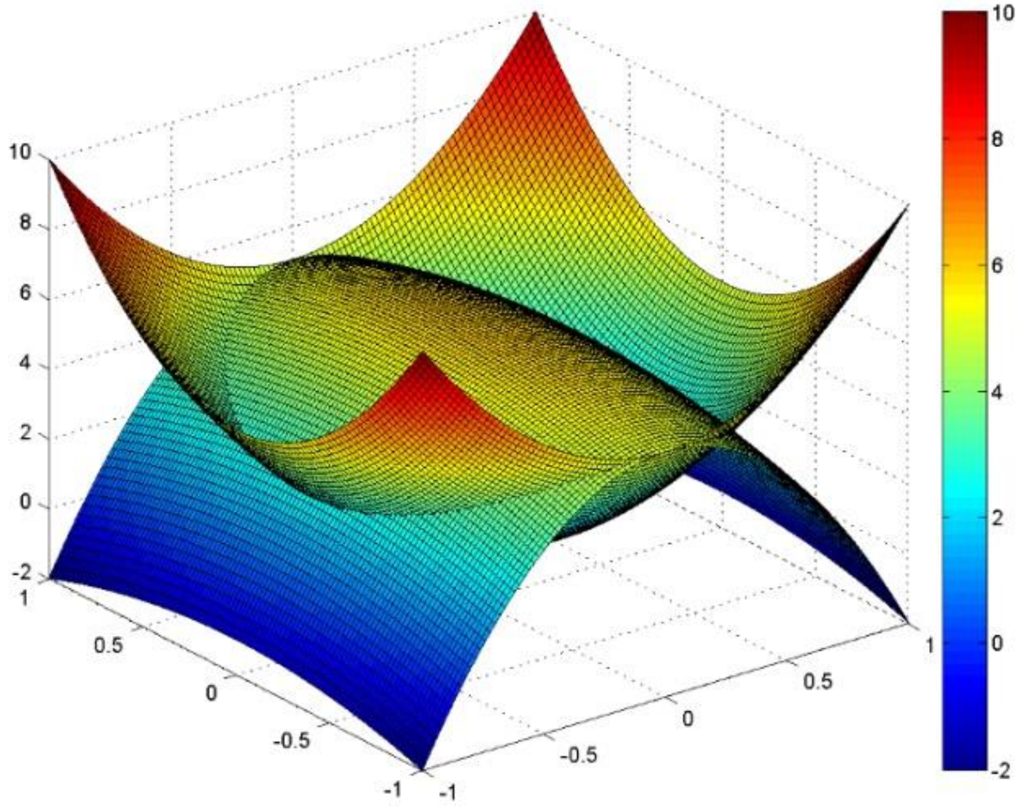
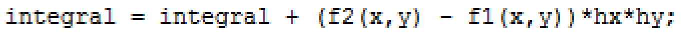


Figure 3.1 showing the region inscribed between the 2 curves: $z = 5(x^2 + y^2)$ and $z = 6 - 7x^2 - y^2$

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 pthreads, after allocating memory and creating threads, we define the region of volume integration in the function the thread would be executing. 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.



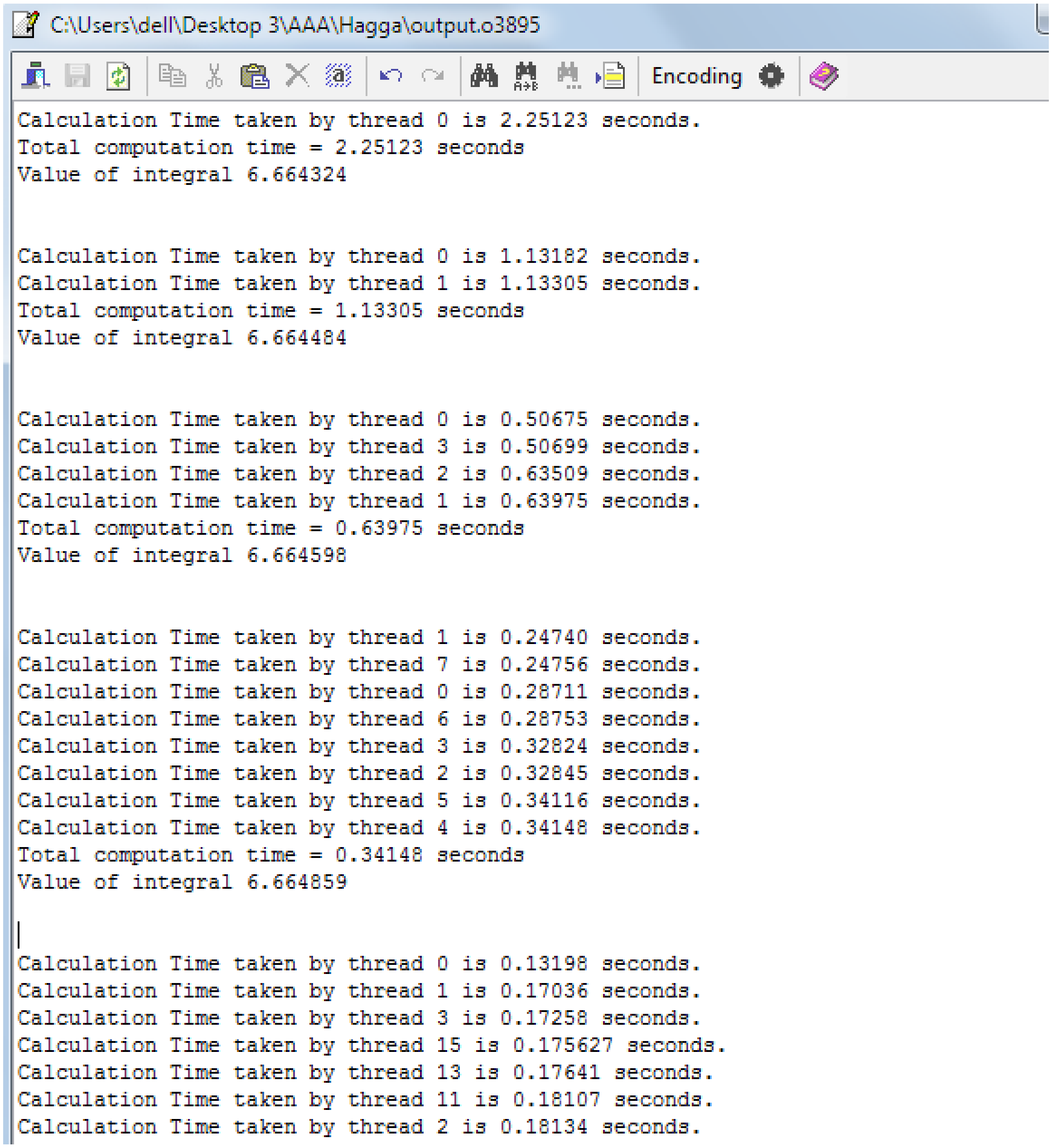
```
integral = integral + (f2(x,y) - f1(x,y))*hx*hy;
```

,

where hx and hy are (xmax-xmin)/nx and (ymax_ymin)/ny.

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

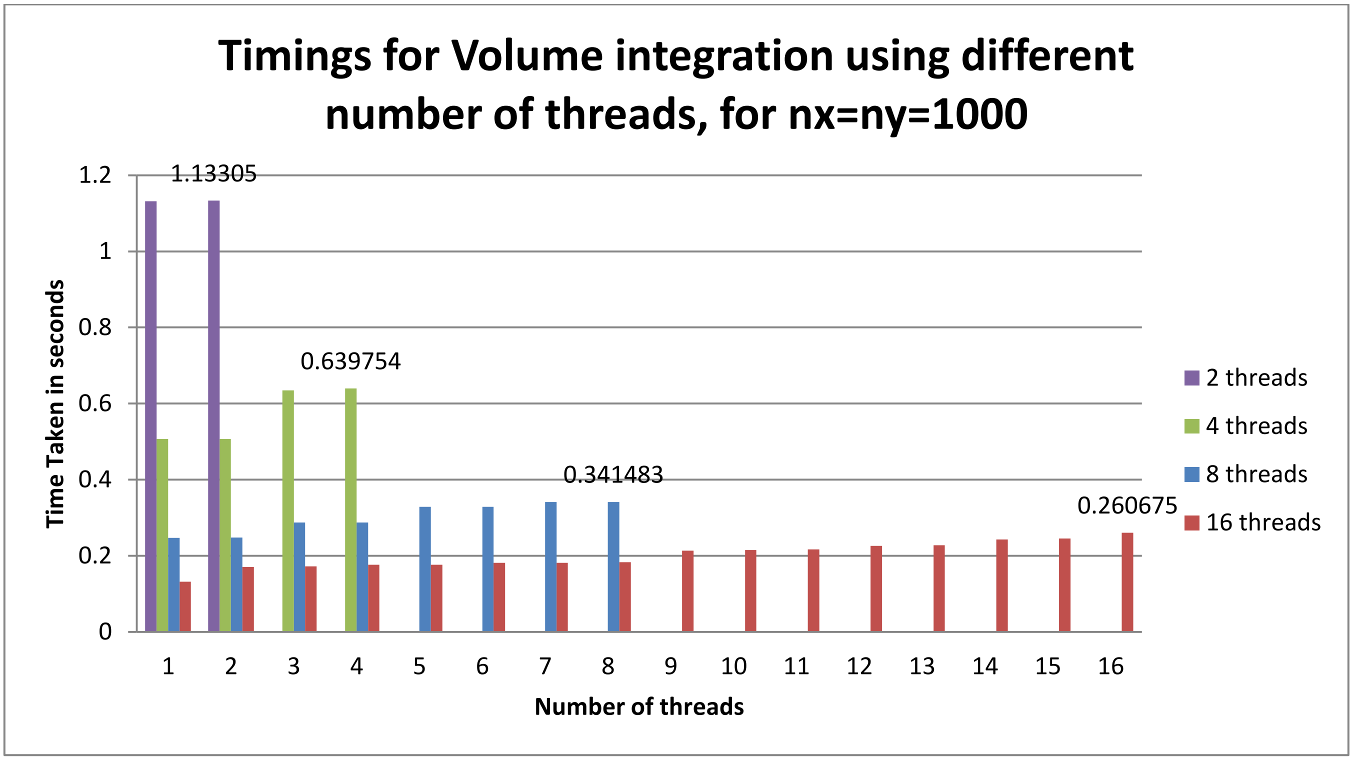


Figure 3.3 Histogram showing variation of time with change in number of threads

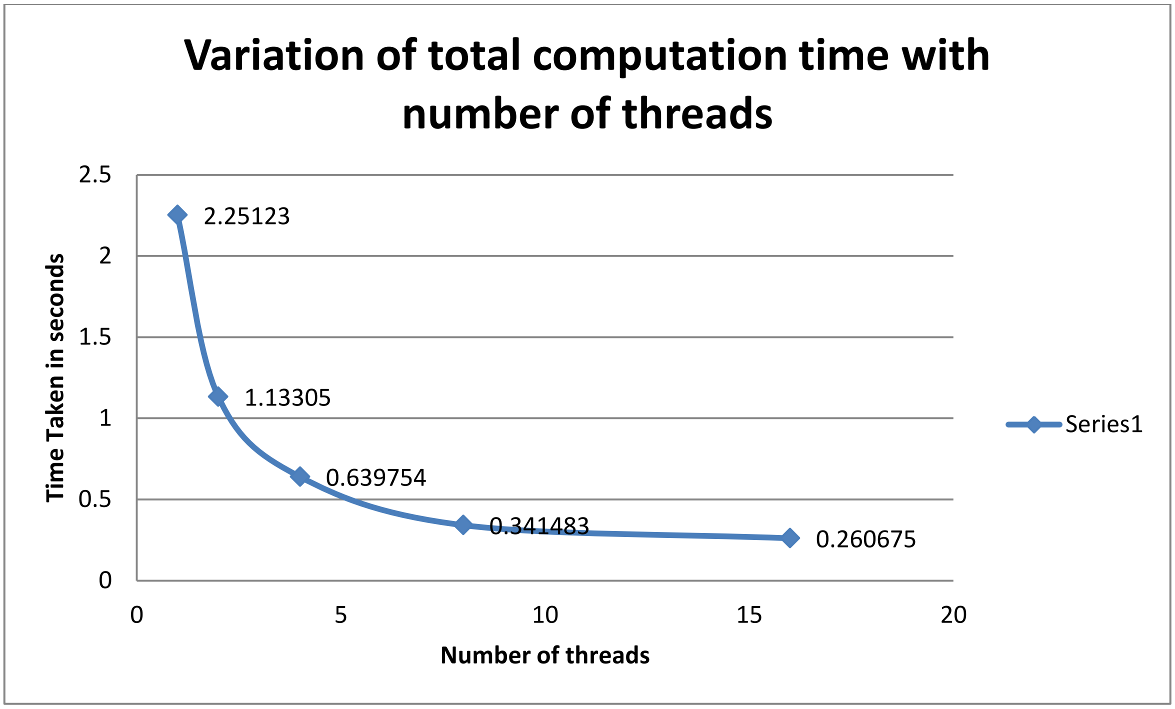


Figure 3.4 Variation of computation time with change in number of threads

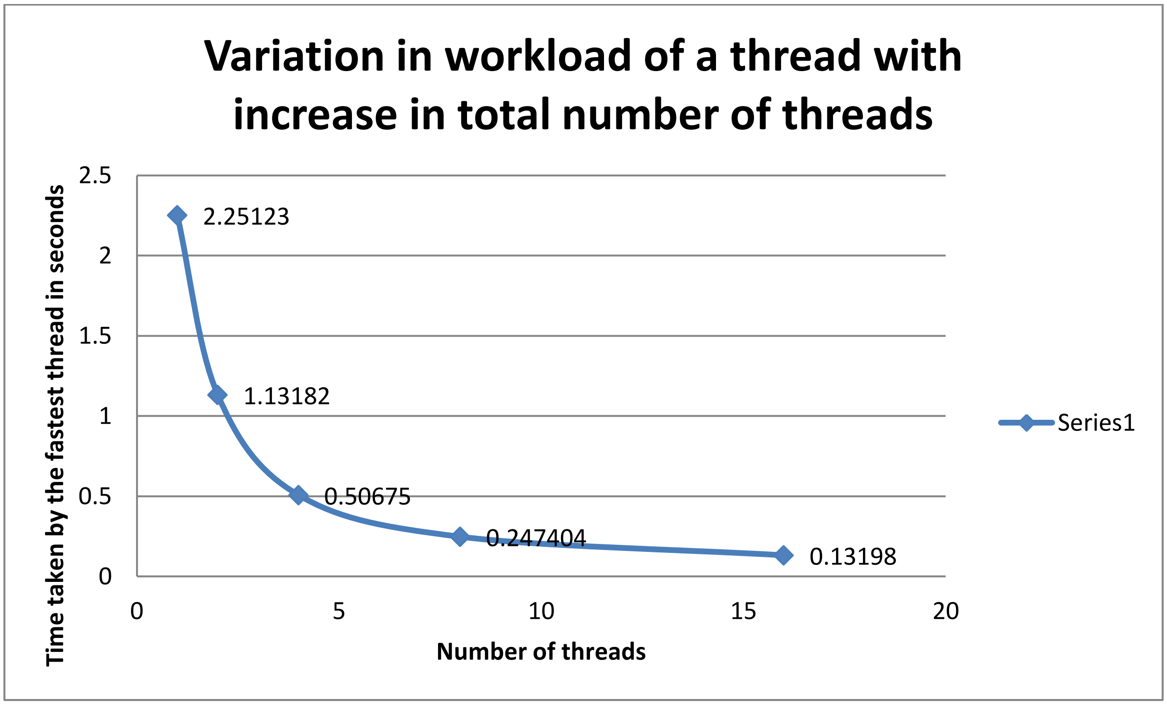


Figure 3.5 Variation in workload of a thread with change in number of threads

- 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 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 pthreads 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 pthreads.

4. Parallel Matrix-Matrix Multiplication Using Fox's Algorithm

Parallelizing the serial code for matrix-matrix multiplication using Pthreads and compare variation in execution time with change in values of n/p.

Method of approach

To carry out matrix-matrix multiplication, we need to accept two matrices form the user. This can be done by initially getting the order of the matrix as the input, declaring the matrices of required order, and then accepting individual entries of the two matrices. Then each row and column is assigned to a particular thread. It is done by first allocating memory to the thread and then executing the function containing the algorithm to multiply two matrices using that particular thread.

The function for matrix-matrix multiplication multiplies individual elements of rows to the corresponding element of the columns of another matrix and then sums them up. Thus we get the individual entries of the resultant matrix. Small changes can be made to the code to convert it into a rectangular matrix multiplier.

After conforming the results to be in sync with the serial matrix multiplication code sent to us, a time measurement code is entered into both the programs which take its value of time from the local system time of the machine the code is executed upon.

n/p	Serial Code	2	3	4	8
16	0.010545	0.009059	0.009080	0.009085	0.008808
32	0.037169	0.033123	0.031824	0.031722	0.031587
64	0.168914	0.149847	0.139290	0.110061	0.096345
128	0.571951	0.531643	0.524209	0.519292	0.517213
256	2.426658	2.355271	2.238314	1.694448	1.665635
512	11.154123	9.677383	9.493616	9.439062	9.411832

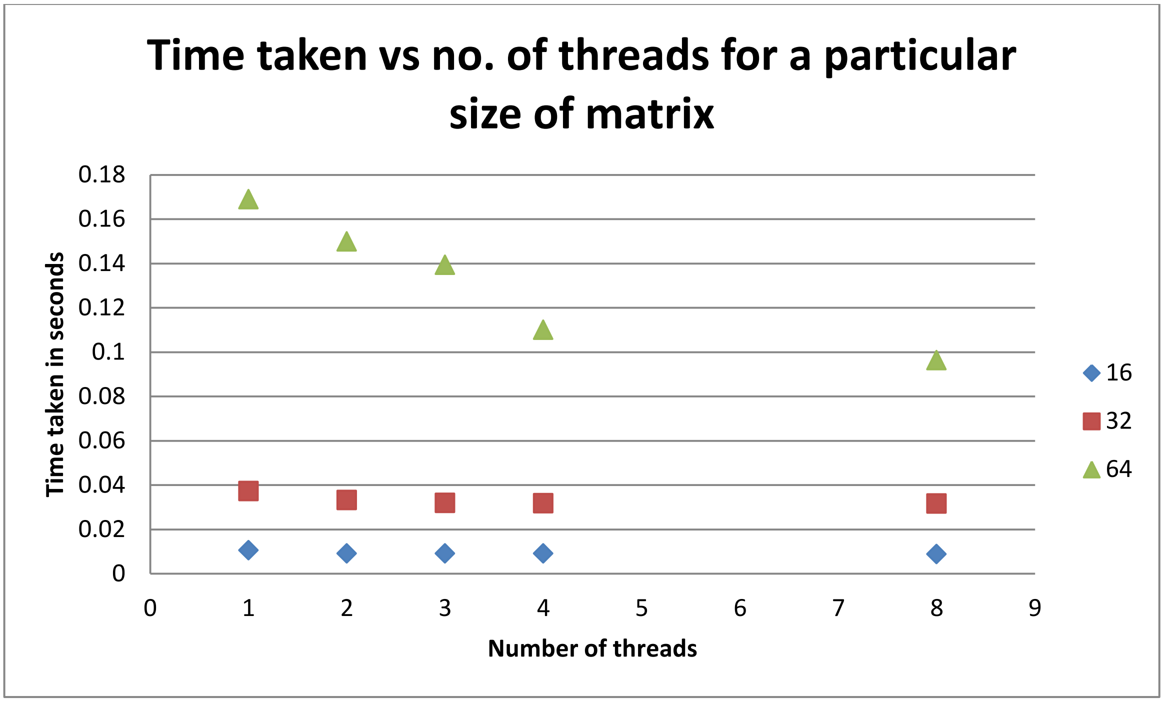


Figure 4.1 Variation in time taken with change in n/p for n=16,32,64

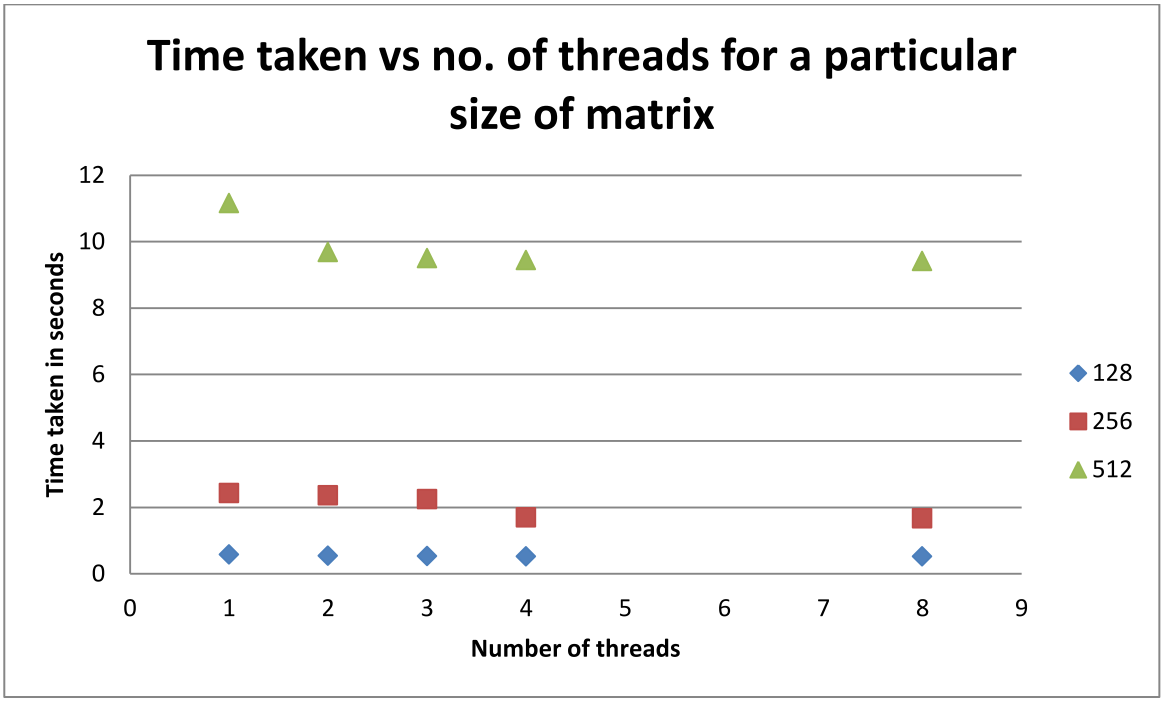


Figure 4.2 Variation in time taken with change in n/p for n=128,256,512

Now from the above results shown for n=16, 256, 1024 we may conclude following few aspects of matrix-vector parallel multiplication:

- The matrix-vector multiplication parallel program written using pthreads is executing correctly. The result of the multiplication is verified to be correct using the serial matrix multiplication code sent to us. 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.
    - The time taken for matrix multiplication using serial code is significantly higher than that when done using parallel code.
    - Practical Results: Time taken for multiplication of matrix of a particular order decreases with increase in number of threads. Moreover, time taken for multiplication increases with increase in order keeping the number of threads constant.
- The timings obtained using pthreads are significantly less than the timings got using MPI. It is because matrix-matrix multiplication involves maximum possible communication overheads when implemented using MPI while pthreads involves no communication time. Moreover, the time taken in creating new threads in case of pthreads is very very less than the communication overheads execution time in case of MPI.
- Attached is the code for fox algorithm using Pthreads, MPI and serial_multiplication case.

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. Pratyush Dayal and Prof. Murrali 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, Rachit, 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.