Assignment - 2

201401222 : Vaibhav Patel
201401409 : Tanmay Patel

Hardware details: CPU model, memory information details, no of cores, compiler, optimization flags if used, precision used:
CPU model- Intel® Core™ i5-4200U CPU @ 1.60GHz × 4
Memory information- 7.7Gb
Graphics - Intel® Haswell Mobile
No. of cores- 4
OS type: 64-bit
Compiler- gcc
L1 Cache size : 128 KBytes, 8 way associative, 64 byte line size.
L2 Cache size : 512 KBytes, 8 way associative, 64 byte line size.
L3 Cache size : 3 MBytes, 12 way associative, 64 byte line size.

Question 1:

Main Remarks: The problem basically has not very complex or time taking computations. So the time taken by the switching and communication between threads outnumbers the speed up. Albeit we are getting 3.37 but if it has some more computations we could get more.
1. Context:

- **Brief Description of the problem :** Calculation of pi using series

- **Complexity of the algorithm (serial) :**
  Serial Algorithm's is O(N) where N is number of terms in series to calculate pi.There are total N division operations.

- **Possible Speedup (theoretical) :** 4

→ We are getting **3.37 (maximum)**
Speedup = 1 / (p / N + s) where N is no of core.
For p = 1 and s = 0 (i.e. completely parallel code) Speedup = N.
For this problem therefore speedup possible is nearly N (N = 4 in our case).

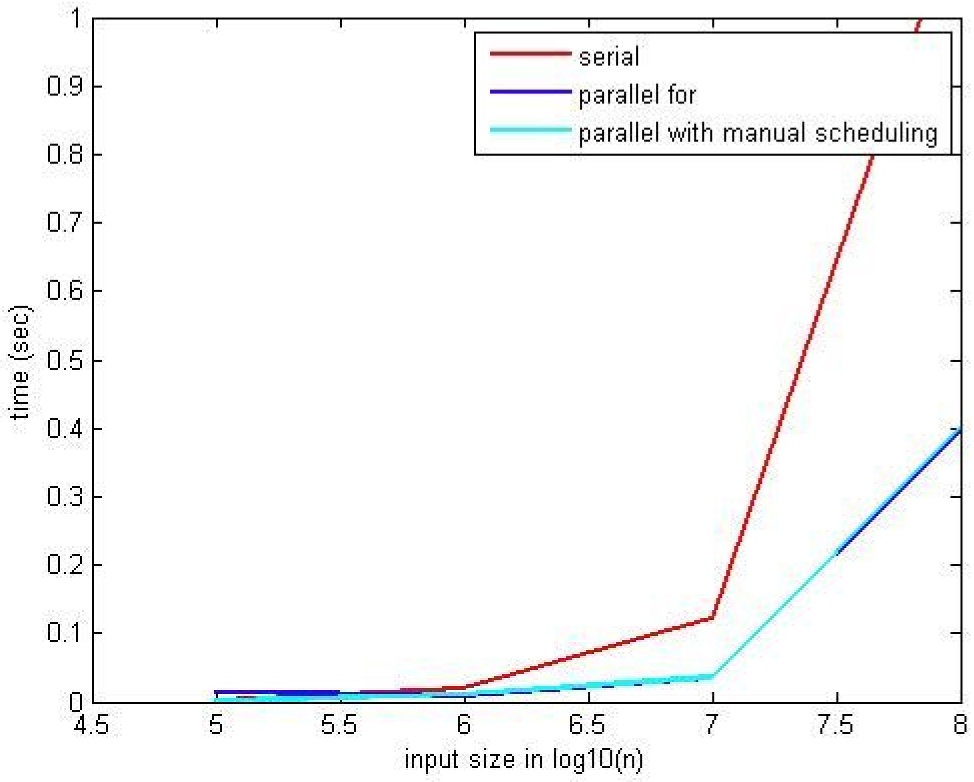
- **Optimization strategy :**
  The series has N / 2 additions and N / 2 subtractions operations. So we used two loops with (N / 2) additions / subtractions operations. We used #pragma omp parallel for reduction with + operator in one of the parallel implementation. For other implementation we used #pragma omp critical to add the sum of each threads to the global sum. The sum of each thread was calculated by finding start and end term of series by using id of thread.

- **Problems faced in parallelization and possible solutions :**
  The problem was straight forward. But at first there was better implementation of alternative plus and minus but we cannot do it in parallel code. So we split the loop in two parts.

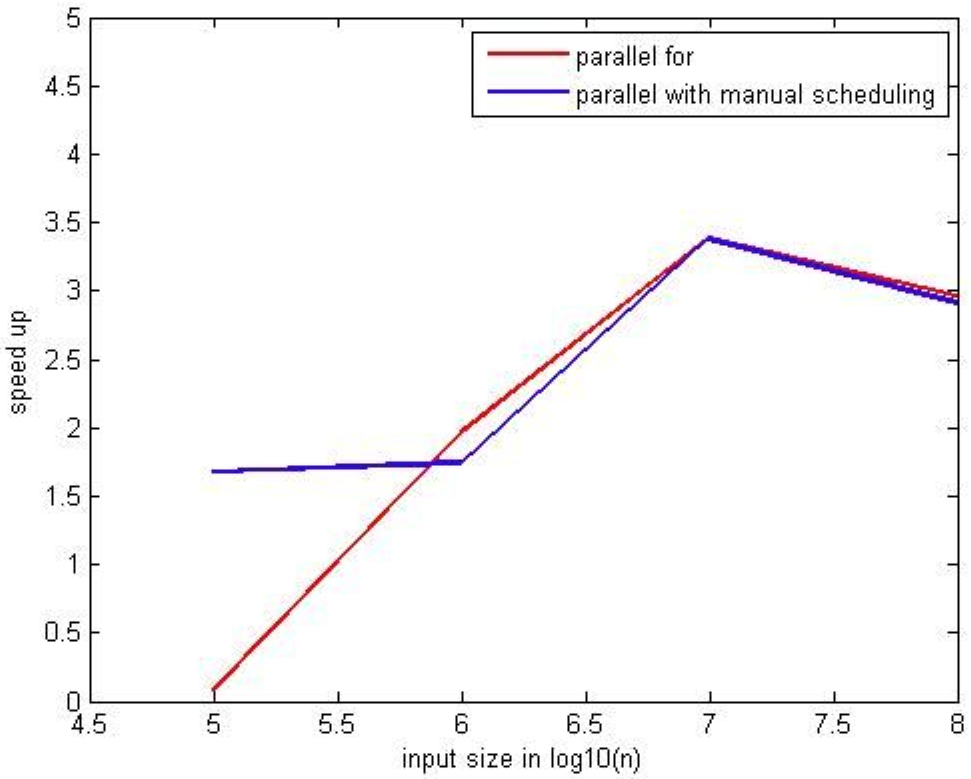
2. Input parameters and Output: Value of N can be considered as input parameter. With increase in N more accurate value of pi can be obtained as output.

3. Parallel overhead time: 0.04secs
Parallel overhead time = N * Tp - Ts where N is number of cores , Tp is time taken by parallel code and Ts is time taken by serial code for perfectly parallel code.

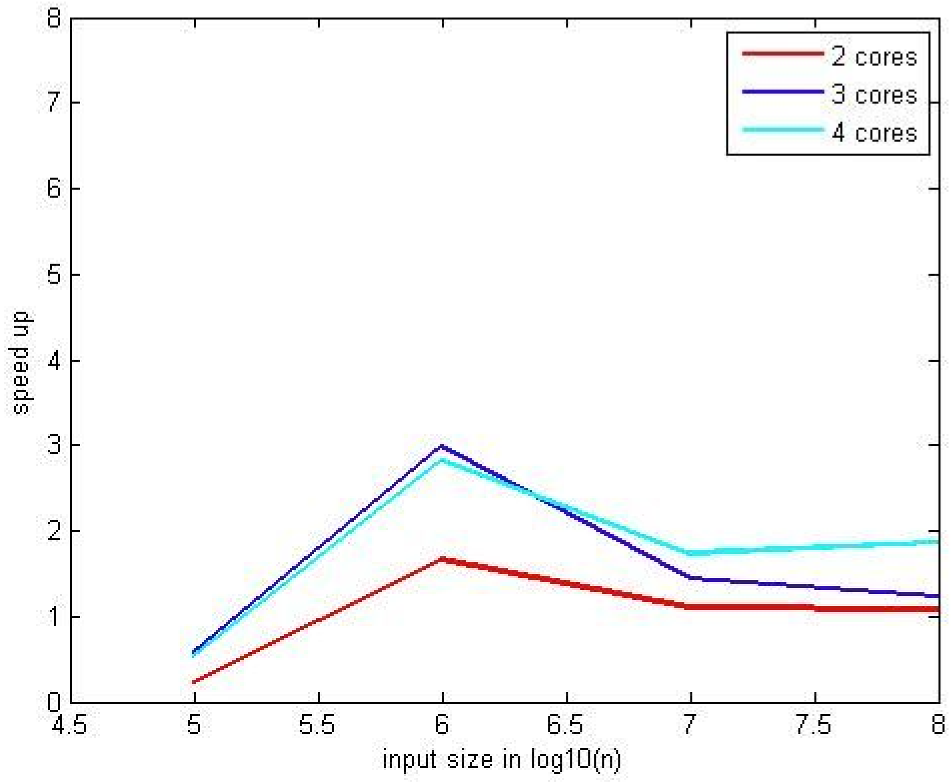
4. Problem Size vs Time:



5. Problem size vs. speedup curve.



6. No. of cores vs. speedup curve.



Question 2:

- **Main remarks: Granularity is very less in this algorithm. Here we are loading 64bytes/4bytes of data at a time using cache line. But the dirty bit problem is creating problem here. We are loading the data and merely multiplying it once.**

This problem also has not very complex or time taking computations. So the time taken by the switching and communication between threads outnumbers the speed up. We are getting only 2.4 because there is 1 data access (Read too much data access) for a single computation.

1. Context:

- **Brief Description of the problem :**

Multiplication of Two vectors followed by summation.

- **Complexity of the algorithm (serial) :**
  Complexity is O(N) where N is size of vectors. There are total N multiplication operations.

- **Possible Speedup (theoretical) : 4**
  We are getting 2.4
  Speedup = 1 / (p / N + s) where N is no of core.
  For p = 1 and s = 0 (i.e. completely parallel code) Speedup = N.
  For this problem therefore speedup possible is nearly N (N = 4 in our case).

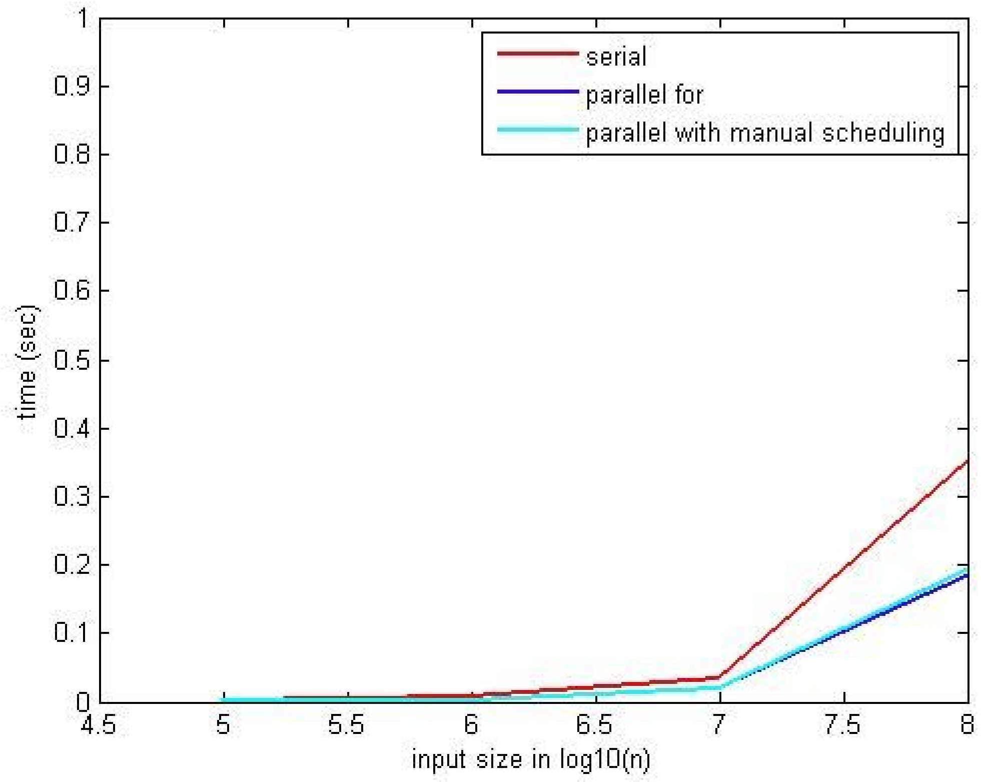
- **Optimization strategy :**
  For simple dot product of two vectors of size N , a for loop with N additions can be parallelized using #pragma omp parallel for reduction with + operator. In other parallel implementation we used #pragma omp critical to add the sum of each threads to global sum in similar way as we did in first problem.

- **Problems faced in parallelization and possible solutions :**
  The problem was straight forward.

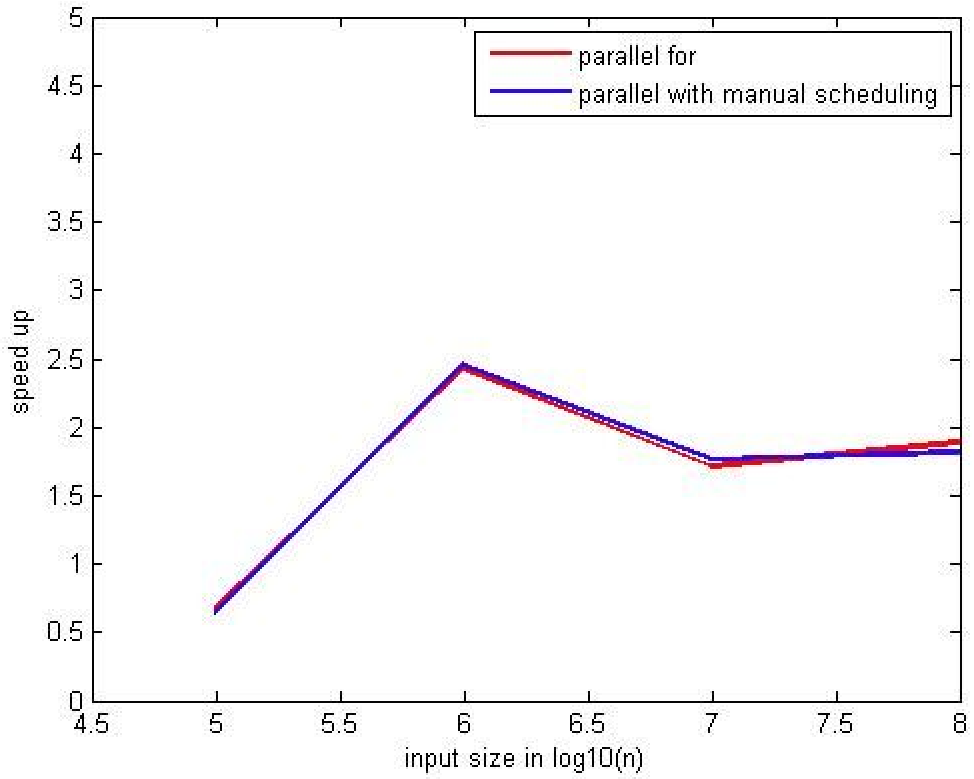
2. Input parameters and Output: Inputs are two vectors of same size (N). Output is the dot product of two vectors.

3. Parallel overhead time: 0.04secs
Parallel overhead time = N * Tp - Ts where N is number of cores , Tp is time taken by parallel code and Ts is time taken by serial code for perfectly parallel code.

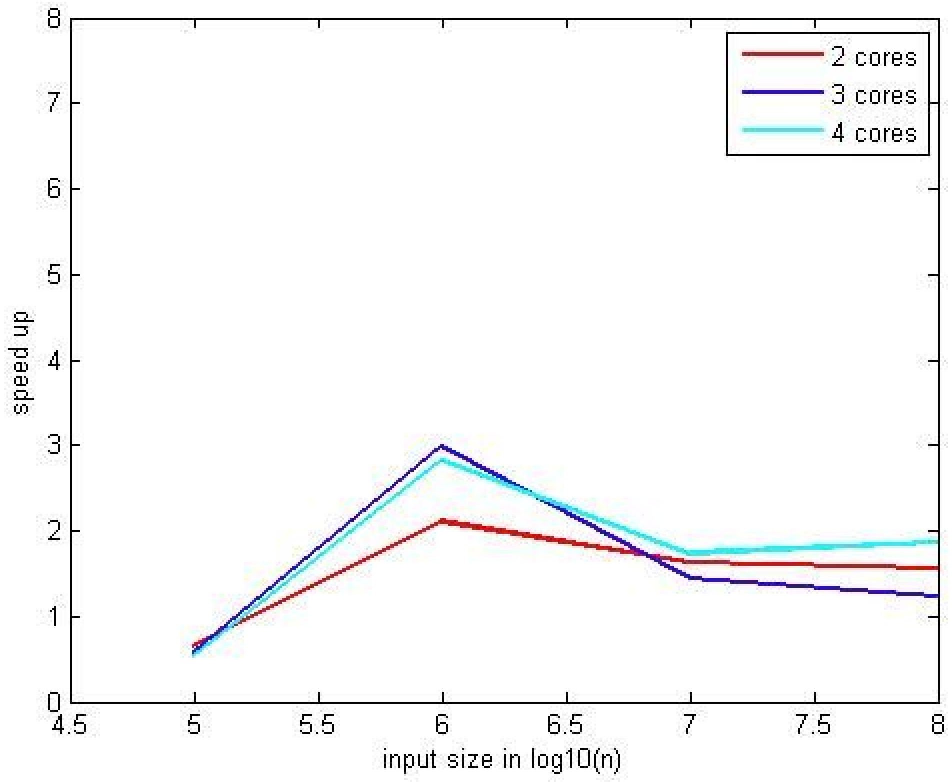
4. Problem Size vs Time:



5. Problem size vs. speedup curve.



6. No. of cores vs. speedup curve.



Question 3:

Main Remarks: Here we can see that each element of a matrix is multiplied with another n elements of the other matrix.So here we have possibility of getting more than theoretical speed up. We took a transpose of the matrix and then all the computation became like a unrolled array. And here we have more cache combine than the serial one. Because here we are actually using the cache more time.

1. Context:

- **Brief Description of the problem :** Multiplication of two matrices of size N x N.

- **Complexity of the algorithm (serial) : (N*N*N)**The algorithm consists of 3 loops all running n times.So the time complexity is n^3 because inside the loop there is constant computation.

- **Possible Speedup (theoretical) :** Is equal to number of cores because mainly the time taken by this computation only. So we can get nearly 4 speedup.

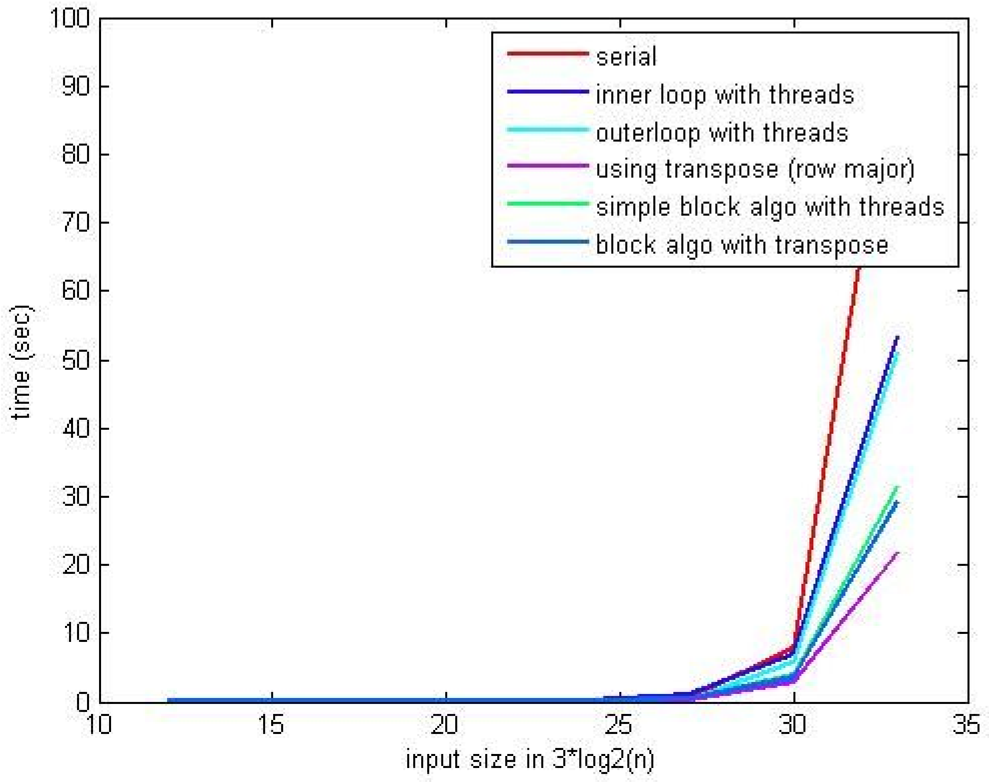
- **Optimization strategy :** C is a row major language. So we can get the transpose of the matrix and then we can access the elements quickly because now we are using elements which are situated in a contiguous row.

- **Problems faced in parallelization and possible solutions :** The main problem faced is that the best algorithm for matrix multiplication is recursive. And we do not know how to do the recursive part. So we did not do it.

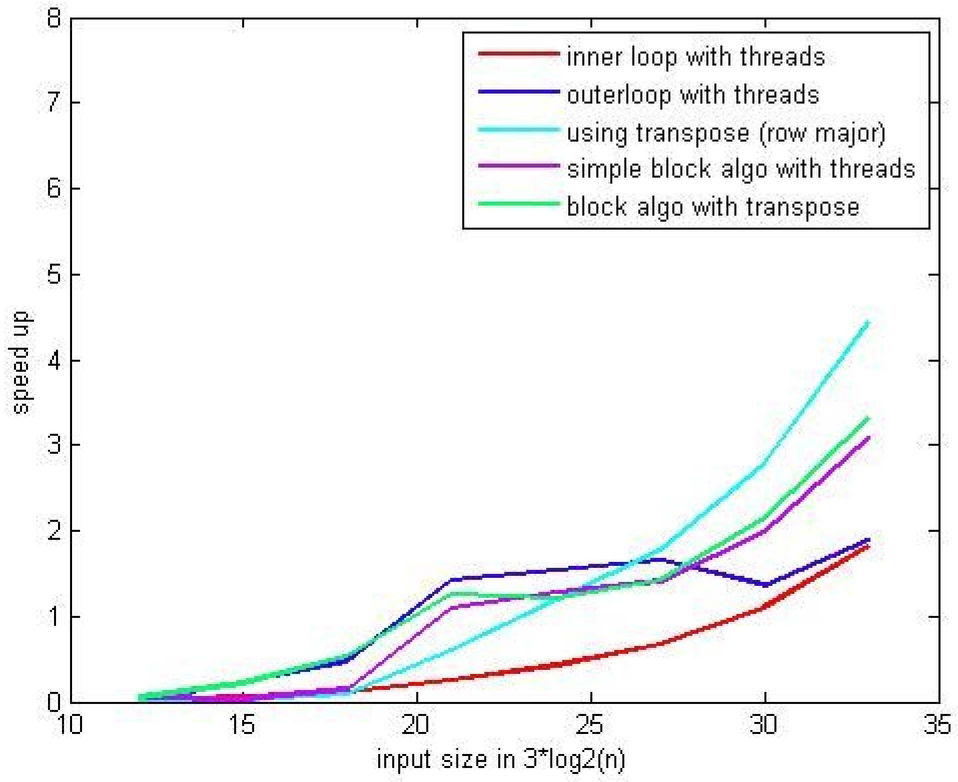
2. Input parameters and Output: Input is 2 n*n matrix and output is one n*n matrix. Multiplication of both of thme.

3. Parallel overhead time: 0.04secs

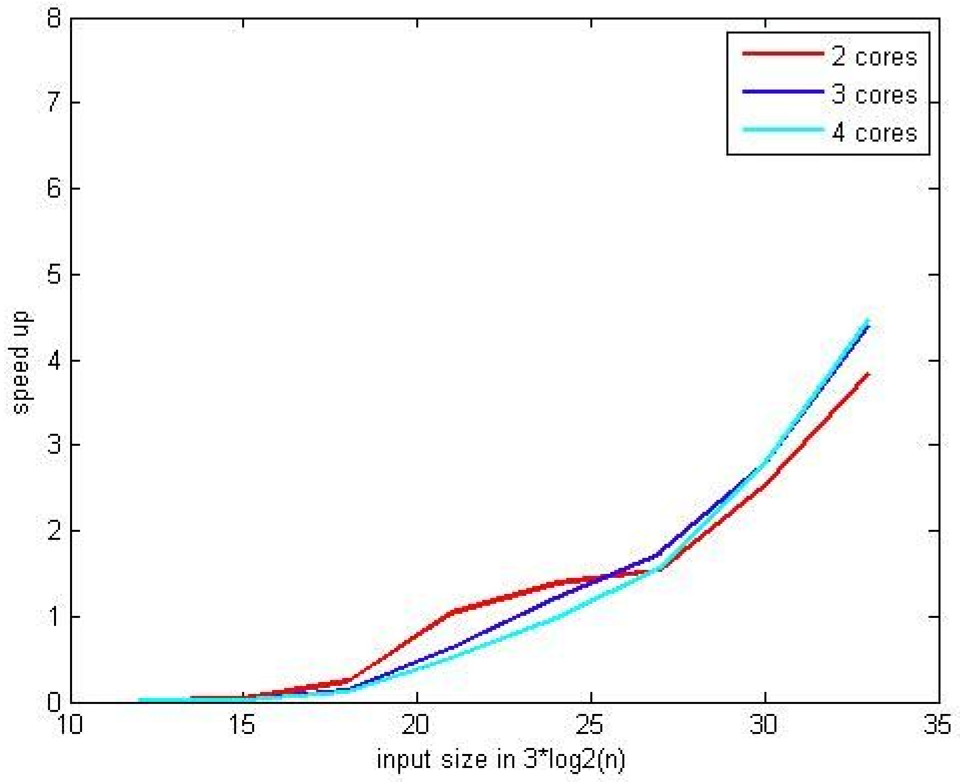
4. Problem Size vs Time:



5. Problem size vs. speedup curve.



6. No. of cores vs. speedup curve.


2 cores
3 cores
4 cores
speed up
input size in 3*log2(n)
0
1
2
3
4
5
6
7
8
10
15
20
25
30
35