HPC Lab 2: Report

Rajdeep Pinge 201401103, Aditya Joglekar 201401086

August 27, 2016

Part I

Q1: Calculation of pi using series (take large value of "N" for summation)

1 Context:

- Brief description of the problem.

  This time we use a finite approximation of an infinite series to the value of pi.

- Complexity of the algorithm (serial).

  The algorithm runs over the whole summation range and hence takes the total number of steps equal to the range over which summation happens. Therefore, the complexity is O(n).

- Possible speedup (theoretical).

  We run the code to find the fraction of parallelizable code

  Fraction of parallelizable code (p) = 0.999929

  In theory also, since there is only one single loop and the instructions in it are independent of each other, The loop and hence, the whole program can be parallelized.

  Therefore by Amdahl's Law the Theoretical speedup that we should get = No. of processors = 4 in this case.

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

  granularity: each sample hit covers 2 byte(s) for 1.27

  index

  0.79 0.00 1/1 main [2]

  [1] 100.0 0.79 0.00 1 approx$_p i$[1]

¡spontaneous¿

[2] 100.0 0.00 0.79 main [2]

0.79 0.00 1/1 approx$_p i$[1]

The profiling information clearly shows that programme spends almost all the time in the function where pi calculation takes place and this calculation mainly contains for loop. Hence we need to parallelize the for loop.

- Optimization strategy.

  We will use the following opimization strategies to parallelize and optimize the processing time.

  1. critical with pragma for -
  2. reduction with pragma for
  3. storing answer in array and then adding (also explain why this is not a good idea)
  4. scheduling

- Problems faced in parallelization and possible solutions.

  Synchronization is important to avoid race conditions. The answers computed by the threads must be pooled in the global variable using a critical pragma.

  The series has alternating terms. We had used if else block to set the term as positive or negative. We improved this by instead constructing a simple function

$$coefficient = 1 - 2 * i$$

2 Hardware details:

- CPU model: AMD A10-5750M APU with Radeon(tm) HD Graphics
- memory information:
  1. L1d cache: 16K
  2. L1i cache: 64K
  3. L2 cache: 2048K

  Here the cache size does not matter much since there is no data access.
- No. of cores: The CPU has 4 physical cores with 1 thread per core. This means, there are 4 logical cores.
- Compiler: gcc version 5.4.0 20160609
- precision used: We have used double precision wherever possible so as to get more accuracy in the result.

3 Input parameters. Output.

The input parameter for both the serial and parallel codes is an integer specifying the number of steps or the range over which the summation works.

In the all the above mentioned parallelization techniques, the answer remains same indicating the correctness of optimization.

Answer for serial code: 3.141593

Answer for serial code: 3.141593

The answer may vary by the order of 1e-5 based on the number of executions and the method type.

4 Parallel overhead time:

(openmp version on 1 core vs serial without openmp)

- CASE I:

  Number of steps(h) = 1e9

  Serial code: Answer = 3.141593, Time taken = 6.882946 s

  Parallel code (1 thread): Answer = 3.141593, Time taken = 7.467788 s

  Overhead = 7.467788 - 6.882946 = 0.584842 s (8.50

- CASE II:

  Number of steps(h) = 1e8

  Serial code: Answer = 3.141593, Time taken = 0.690741 s

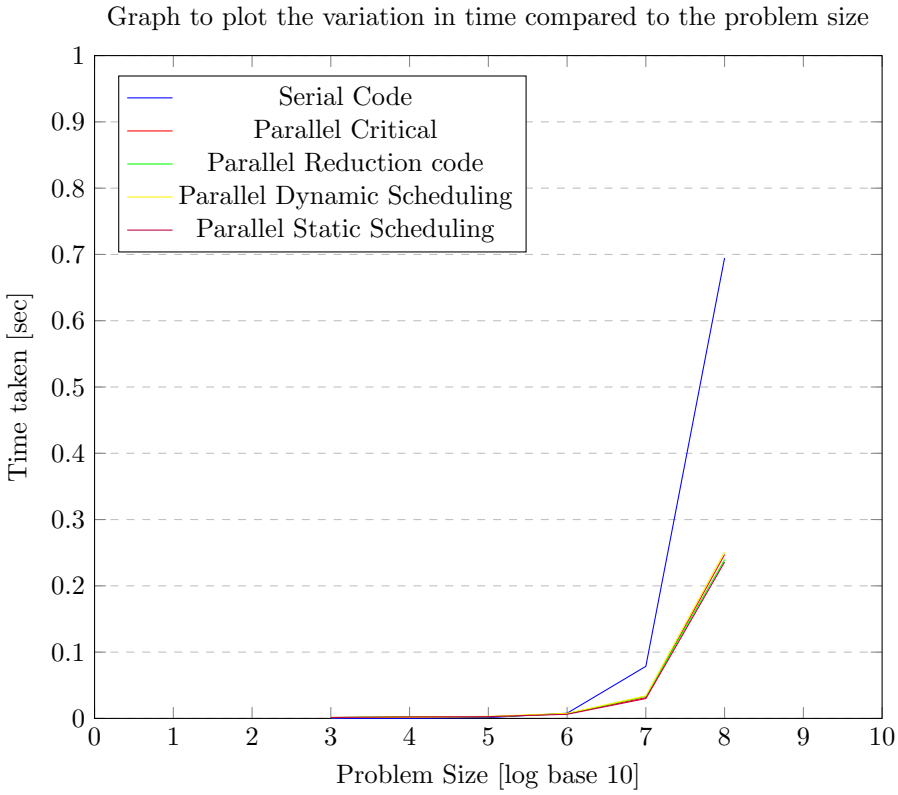
  Parallel code (1 thread): Answer = 3.141593, Time taken = 0.752298 s

  Overhead = 0.752298 - 0.690741 = 0.061557 s (8.91

5 Problem Size vs Time (Serial, parallel) curve:

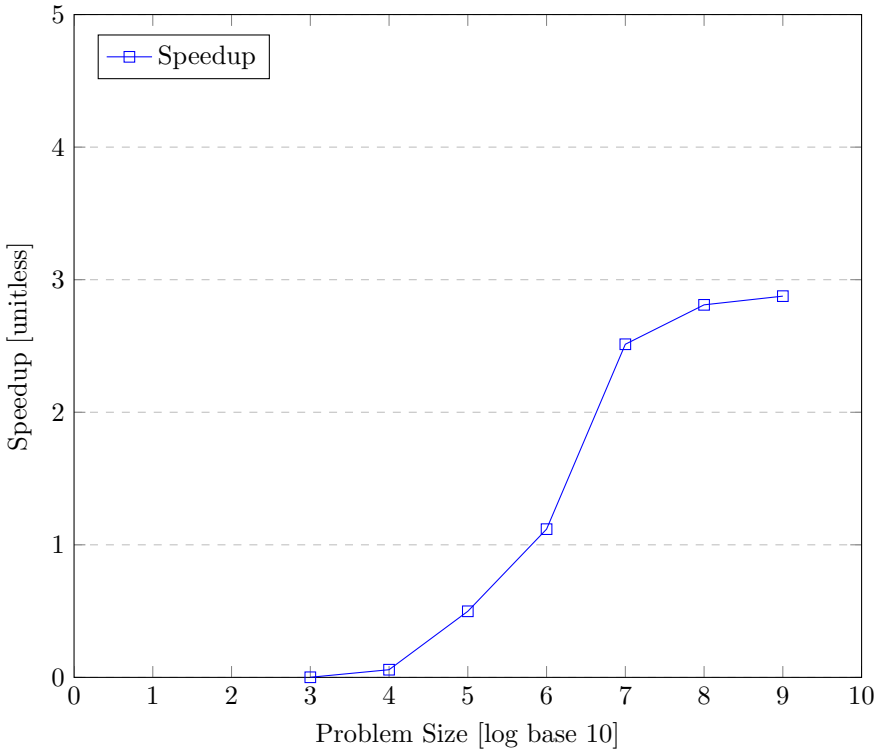
Speedup curve. Observations and comments about the results.

We have taken the number of cores equal to 4 throughout.


Graph to plot the variation in time compared to the problem size
Serial Code
Parallel Critical
Parallel Reduction code
Parallel Dynamic Scheduling
Parallel Static Scheduling
Time taken [sec]
Problem Size [log base 10]
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
0
1
2
3
4
5
6
7
8
9
10

6 Problem size vs. speedup curve.

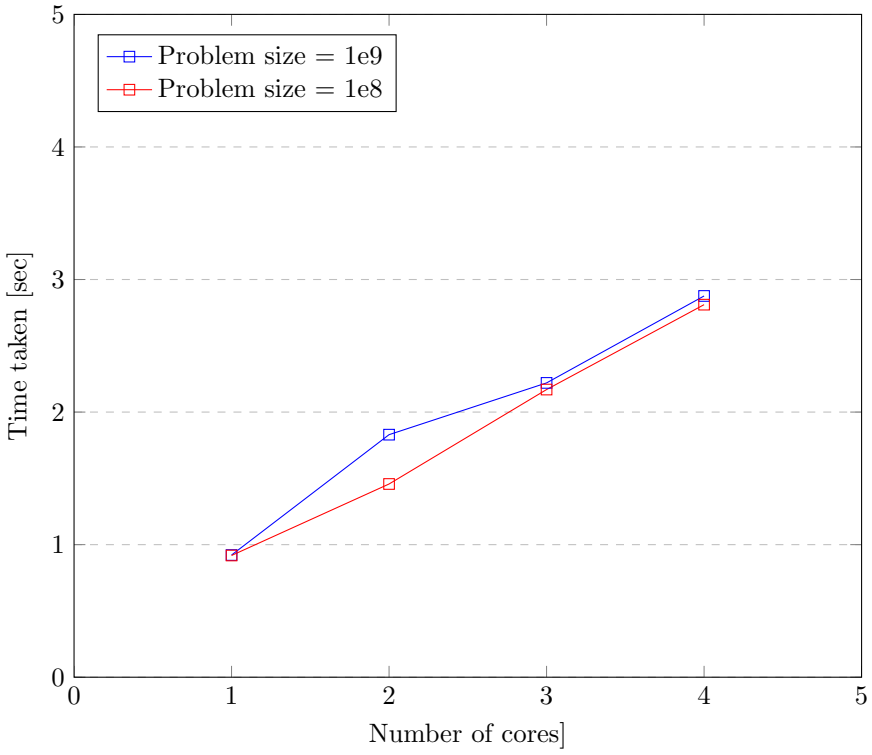
Graph to plot the variation in time compared to the problem size



A point to be noted. Here the speedup is increasing with increase in problem size however, the rate of change of speedup with problem size is decreasing towards the end. The indicates the change in speedup in powers of 10 in the end is decreasing. This means that the problem is not scalable as speedup is not able to keep pace with increase in problem size

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

Graph to plot the variation speedup compared to the number of cores



8 Measure performance in MFLOPS/sec.

According to the problem, we have total 6 Floaing Point Operations per cycle of for loop and we have k such cycles. Therefore we have total 6*k FLOP. We measure the time taken to execute those many operations.

$$performance = No.of MFLOP/time_taken(sec)$$

For parallel code, performance = 2251.044386 MFLOPS
For serial code, performance = 764.528139 MFLOPS

The performance of the parallel code is almost thrice the serial code. But theoretically it should have been 4 times! The reason that the perfomance is 3 times is because of the pipeline stalls. Since the operations contain * and % operations, each of them takes different amount of time in the CISC architecture and hence the length of the pipeline differes as mentioned in the lecture. Also

since only execute phase of the pipeline is used, there is no optimal use of pipeline and hence the performance optimization is limited.

Part II

Q2: Multiplication of Two vectors followed by summation. (serial and parallel comparison for accuracy of results)

1 Context:

- Brief description of the problem.

  This question essentially asks us to calculate the dot product of two vectors. As we know, the vectors can be represented as nx1 matrices. Hence the dot product is actually the sum of products of each of the row elements.

- Complexity of the algorithm (serial).

  The algorithm runs over the number of dimensions of the vector and hence takes the total number of steps equal to the number of rows in the matrix. Therefore, the complexity is O(n).

- Possible speedup (theoretical).

  We run the code to find the fraction of parallelizable code

  Fraction of parallelizable code (p) = 0.999920

  In theory also, since there is only one single loop performing the sum of product of two vector dimensions, The loop and hence, the whole program can be parallelized.

  Therefore by Amdahl's Law the Theoretical speedup that we should get = No. of processors = 4 in this case.

- Profiling information (e.g. gprof). Serial time. granularity: each sample hit covers 2 byte(s) for 0.56

  index

  ¡spontaneous¿

  [1] 100.0 1.08 0.70 main [1]

  0.70 0.00 1/1 dotpro [2]

  0.70 0.00 1/1 main [1]

  [2] 39.2 0.70 0.00 1 dotpro [2]

  Here it shows that the programme spends major time in main function. But we have included the vector array initialization part in the main

function which in theory should not be counted since we assume that we already have two stored vectors. Hence the time consumed is mainly in the other function. Therefore we need to analyze that function. Considering the problem, there is only one loop needed to parallelize the function and there is no dependency between the data accesses that are made in different loops. Hence, In theory almost all the code can be parallelized. This is the case as has been stated by the above practical observed time taken by the parallelizable code serially.

- Optimization strategy.

  We will use the following opimization strategies to parallelize and optimize the processing time.

  1. critical with pragma for
  2. reduction with pragma for
  3. scheduling

- Problems faced and possible solutions.

  The major problem that we faced was the availability of stack as well as heap memory. The vectors need to be stored in memory before using and when the dimensions of vector exceed 1e5, the stack memory becomes insufficient to store two such vectors. Therefore we came up with the solution of using heap memory by dynamic allocation using malloc. In this method also we were unable to get the size ¿ 1e8. This is because, we were taking data type int which have size 4 bytes. Therefore for size = 1e9, the total size required by one array is 4 * 1e9 = 4GB. Two such arrays require 8GB RAM. Our laptop contains 8GB RAM and hence unable to allocate this much memory which results in illegal memory access and hence core dumped error. Hence we went for the short data type which takes only 2 bytes of memory per element.

2 Hardware details:

- CPU model: AMD A10-5750M APU with Radeon(tm) HD Graphics
- memory information:

  1. L1d cache: 16K
  2. L1i cache: 64K
  3. L2 cache: 2048K

  Since there is data access from memory, it is important to get to know the cache line size of the system. It can be obtained by typing in the following commands: getconf LEVEL1_DCACHE_LINESIZE, getconf LEVEL2_CACHE_LINESIZE

  For level1 cache it is = 64 bytes, for level2 cache also it is = 64 bytes.

- No. of cores: The CPU has 4 physical cores with 1 thread per core. This means, there are 4 logical cores.
- Compiler: gcc version 5.4.0 20160609
- precision used: We have used double precision wherever possible so as to get more accuracy in the result.

3 Input parameters. Output.

Here for convenience purpose we have put all the dimensions of both the vectors to be equal to one. This is important to maintain similarity between serial and parallel code.

Input to the function: pointers to both the vector arrays, size of arrays.

Output: Dot product of two vectors. The answer is same for serial and all the parallel implementations indicating correctness of optimization.

4 Parallel overhead time:

(openmp version on 1 core vs serial without openmp)

- CASE I:

  Number of steps(h) = 1e9

  Serial code: Time taken = 6.866318 s

  Parallel code (1 thread): Time taken = 5.430741 s

  Overhead = 5.430741 - 6.866318 = -1.43 s

- CASE II:

  Number of steps(h) = 1e8

  Serial code: Time taken = 0.690802 s

  Parallel code (1 thread): Time taken = 0.532150 s

  Overhead = 0.532150 - 0.690802 = -0.16 s

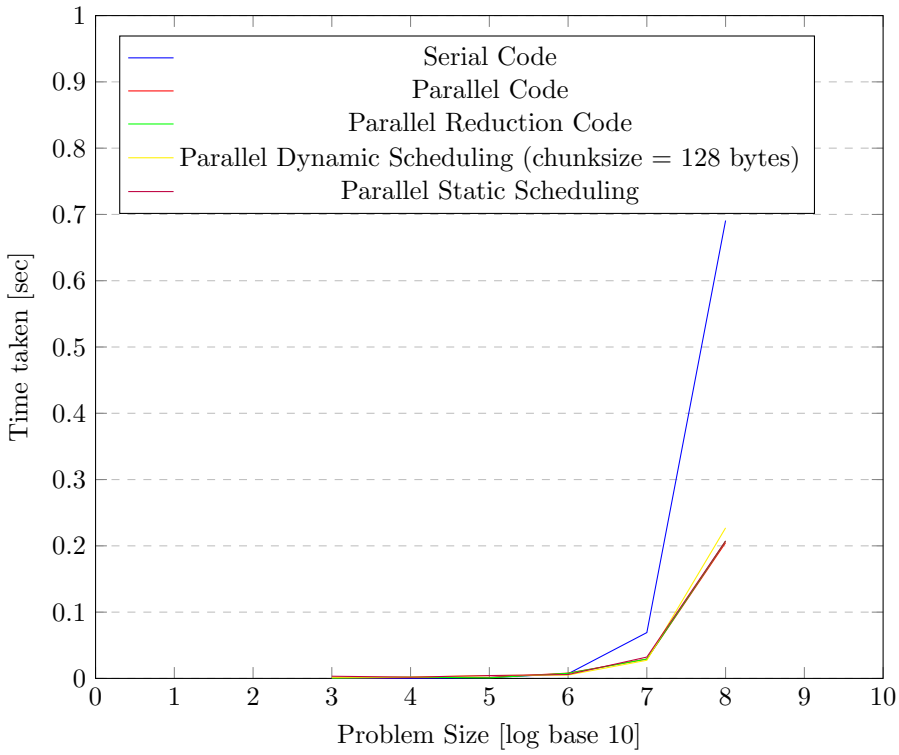
The time taken by single core is still less. This is very confusing but our assumption is that due to pragma for the work is automatically divided among other processors. Also the cache memory available increases and hence data access time decreases.

5 Problem Size vs Time (Serial, parallel) curve:

Speedup curve. Observations and comments about the results.

We have taken the number of cores equal to 4 throughout.

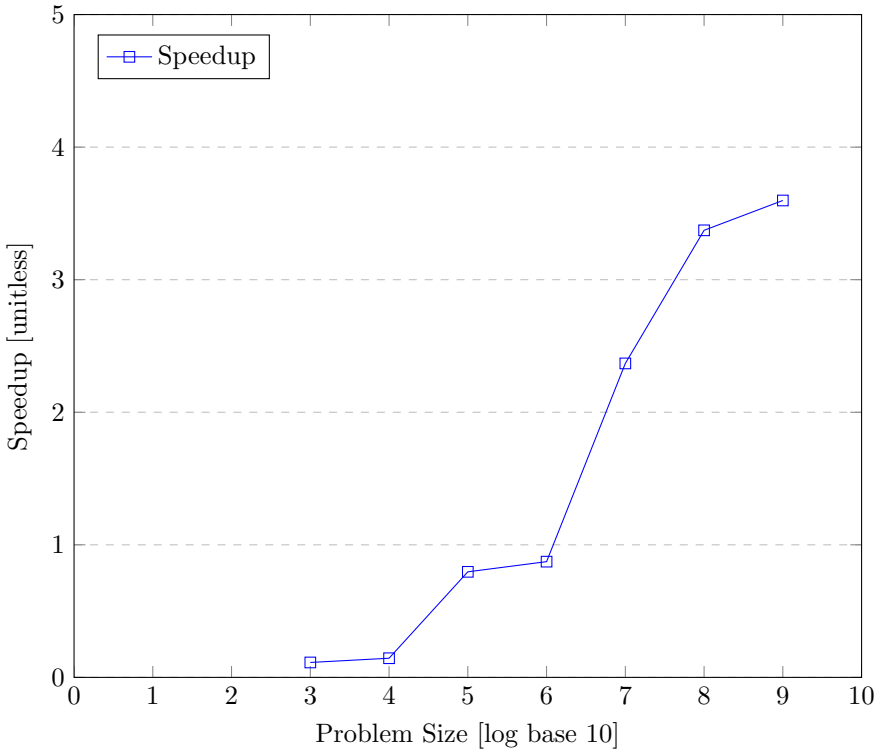
Graph to plot the variation in time compared to the problem size



Clearly, the parallel plots for all schemes scale better with the problem size. Initially, the overhead of creating threads gives the serial code an advantage. Notice that the 5 parallel schemes do not differ much in performance. Hence the plots overlap

6 Problem size vs. speedup curve.

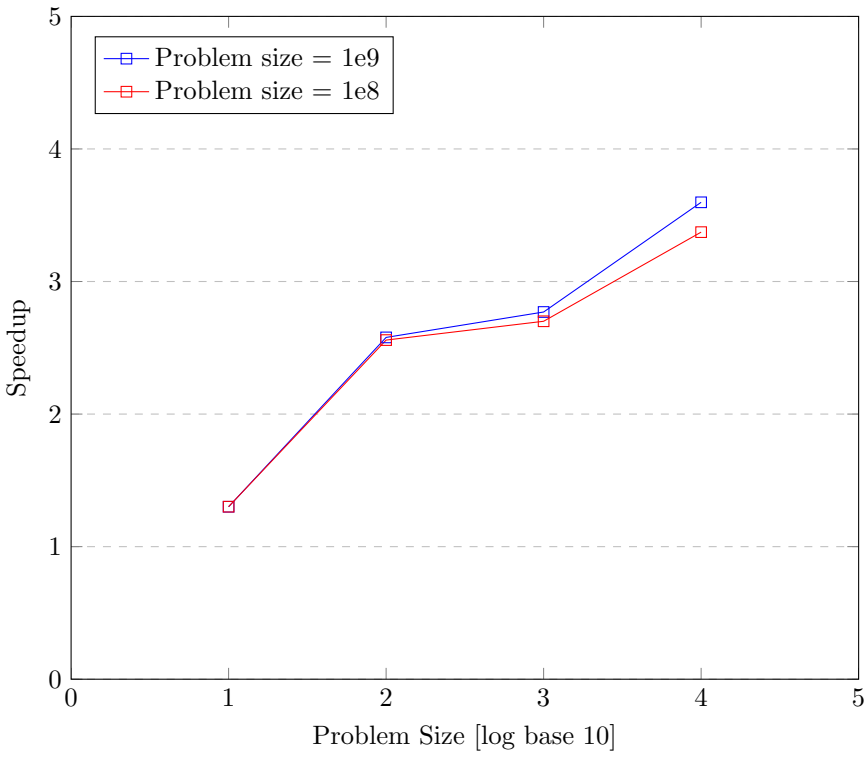
Graph to plot the variation in speedup compared to the problem size



The speedup increases with increasing problem size. Until the size is order of 10e3, serial code has an edge. After that for medium sized problems, there is a boost in speedup every alternet power of 10. While for large problems after , speed increases rapidly.

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

Graph to plot the variation in speedup compared to the problem size



The main bottleneck here is the data access. We have just parallelised the code in a computational sense, we have not included any data reuse or made use of spatial locality. This could be exploited to achieve greater speedup

8 Measure performance in MFLOPS/sec.

According to the problem, we have only 1 Floating Point Operation per cycle of for loop and we have N(dimension of vector) such cycles. Therefore we have total 1*N FLOP. We measure the time taken to execute those many operations. Note here that since each operation is dependent on the value stored in the memory, therefore we must also include the data access time required to get the data from the memory on which to operate.

$$performance = No.of MFLOP/time_taken(sec)$$

For parallel code, performance = 452.061514 MFLOPS
For serial code, performance = 144.776120 MFLOPS

Part III

Q3: Matrix Multiplication (starting with simple implementation – gradually implement Block algorithm as discussed during the lecture, create n x n matrix; take n in multiples of 2 while increasing the size of matrix (go up to 528 and 1024); change size of block to see the effect.)

1 Context:

- Brief description of the problem.

  In continuation with last question, here we have to perform the multiplication of two nxn matrices instead of two nx1 vectors. The amount of computations as well as the data access is much more as compared to the simple dot product and hence the problem is difficult to parallelize.

  The key observation is that the rows of the first matrix seem to be reused a lot for calculating the elements of the multiplication matrix. Hence the algorithm which is able to recycle the data and hence minimize data access will give a large boost in performance.

- Complexity of the algorithm (serial).

  During matrix multiplication, we multiply each row with each column. There are n row and n columns and each row and column has n elements. Hence total complexity is $O(n^3)$.

- Profiling information (e.g. gprof). Serial time. From the profiling info it is clear that the time in the triple nested for loop structure is the most and hence we need to find a way to parallelize the 3 for loops.

- Optimization strategy

  1. block method
  2. parallelizing the loop

- Problems faced and possible solutions.

  It was not possible to parallelize all the nested for loop because it always results in race conditions. Hence for normal parallelization of code, we have only parallelized the inner most for loop.

2 Hardware details:

- CPU model: AMD A10-5750M APU with Radeon(tm) HD Graphics
- memory information:
  1. L1d cache: 16K

2. L1i cache: 64K
3. L2 cache: 2048K

Since there is data access from memory, it is important to get to know the cache line size of the system. It can be obtained by typing in the following commands: getconf LEVEL1_DCACHE_LINESIZE, getconf LEVEL2_CACHE_LINESIZE

For level1 cache it is = 64 bytes, for level2 cache also it is = 64 bytes.

- No. of cores: The CPU has 4 physical cores with 1 thread per core. This means, there are 4 logical cores.
- Compiler: gcc version 5.4.0 20160609
- precision used: We have used double precision wherever possible so as to get more accuracy in the result.

3 Input parameters. Output.

Here for convenience purpose we have put all the dimensions of both the matrices to be equal to one. This is important to maintain similarity between serial and parallel code.

Input to the function: pointers to both the matrices, size of nxn matrices

Output: Multiplication of two matrices. The answer is same for serial and all the parallel implementations indicating correctness of optimization.

4 Parallel overhead time:

(openmp version on 1 core vs serial without openmp)

- CASE I:

  size of matrix(n) = 1024

  Serial code: Time taken = 26.292476 s

  Parallel code (1 thread): Time taken = 20.745241 s

  Overhead = 20.745241 - 26.292476 = -6 s

- CASE II:

  Size of matrix(n) = 512

  Serial code: Time taken = 2.020647 s

  Parallel code (1 thread): Time taken = 1.602209 s (averaged out)

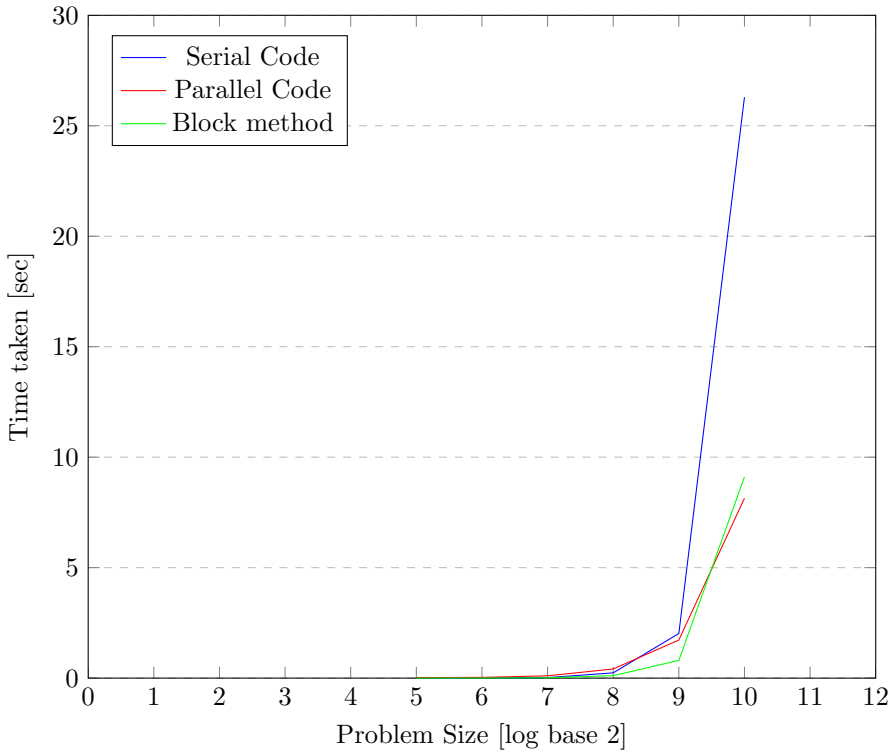
  Overhead = 1.602209 - 2.020647 = -0.42 s

The time taken by single core is still less. This is very confusing but our assumption is that due to pragma for the work is automatically divided among other processors. Also the cache memory available increases and hence data access time decreases.

5 Problem Size vs Time (Serial, parallel) curve:

Speedup curve. Observations and comments about the results.

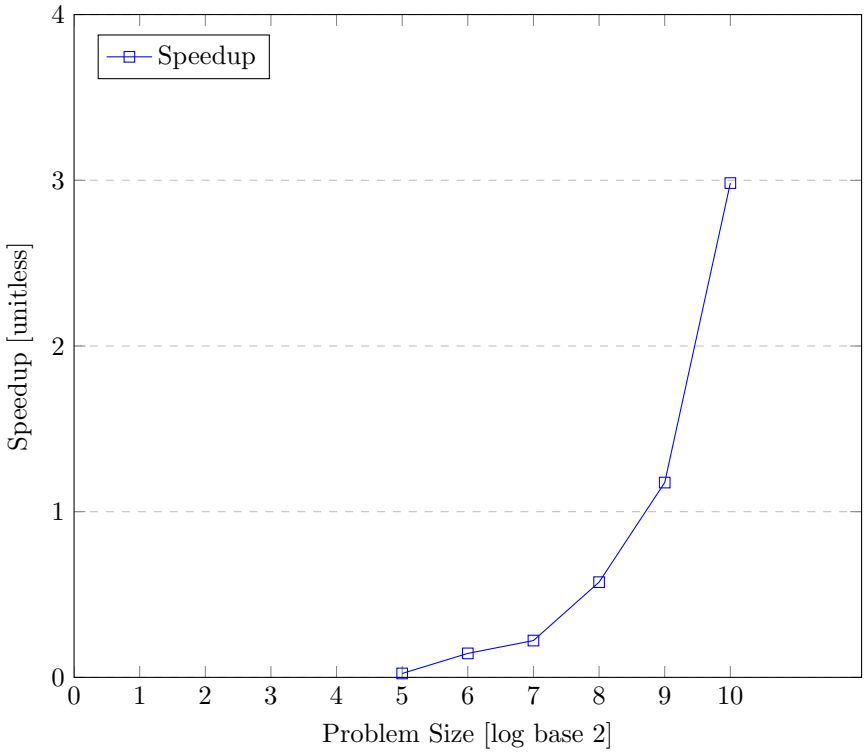
Graph to plot the variation in time compared to the problem size



Observing the plot it is clear that the block method is much faster that even the simple parallelization. This is because block method exploits the spatial locality property of the cache hence the memory access time, which is the main bottleneck of the program, reduces drastically.

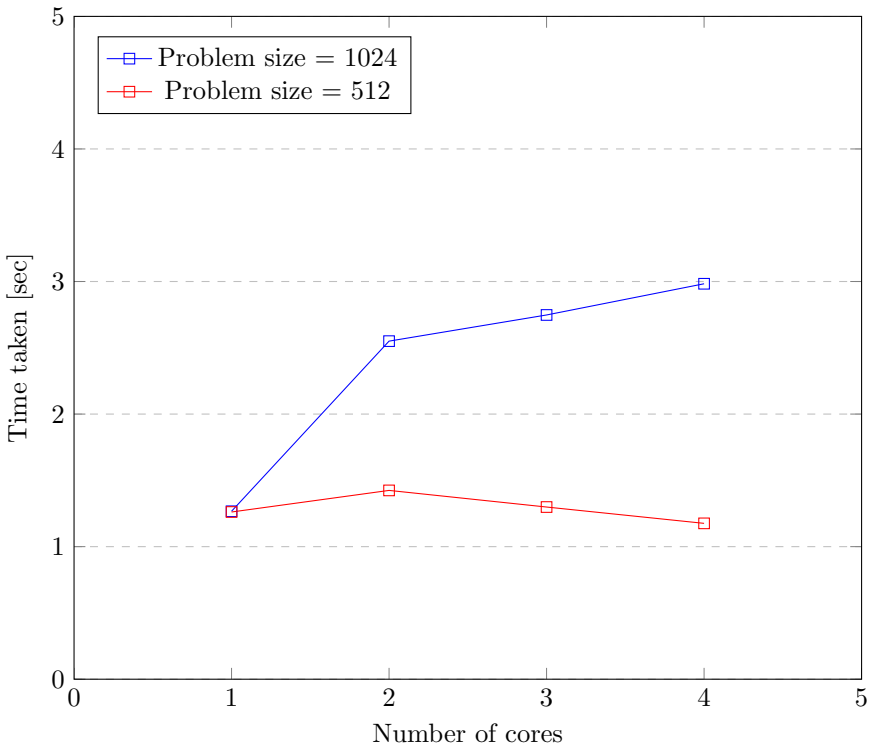
6 Problem size vs. speedup curve.

Graph to plot the variation in speedup compared to the problem size



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

Graph to plot the variation in speedup compared to the no of cores



Surprisingly the speedup decreases as the no. of increase for matrix of size 512. It is very difficult to interpret this result because logically as the amount of cache memory increases, the speedup should increase as is the case in the 1st case.

We have learnt that for a given problem size there is an optimal number of cores which give maximum speedup. This is because after a point the increase in performance does not justify the increase in overhead.

Thus for problem size 512, we can clearly see the optimum reached at 2 cores.

8 Measure performance in MFLOPS/sec.

According to the problem, we have only 2 Floaing Point Operation per cycle of inner for loop and 3 for loops. Also outside inner for loop we have 1 FLOP running for two for loops. Note here that since each operation is dependent on the value stored in the memory, therefore we must also include the data access

time required to get the data from the memory on which to operate.

$$performance = No.of MFLOP/time_taken(sec)$$

For parallel code, performance = 148.417792 MFLOPS

For serial code, performance = 123.194924 MFLOPS

Here we have not got much improvement since the data access time far exceeds the time of execution.

9 Observations and conclusions

The naive algorithm is of cubic order. This does not make any use of the spatial locality seen in this algorithm. The same chunk of data from a row is reused to calculate the C matrix elements. This though can be parallalized by merely dividing the computation. This however does not give significant speedup and this motivates the use of block based algorithms exploiting spatial locality.

Before going into the block algorithm, we would like to mention our attempts to parallalize the triple look of the serial code.

First we naively put the parallel pragma and expected it to divide the work amongst the threads. However this leads to race condiitons and is not the proper way. We couldn't wrap our heads round the way in which we could parallalize the outer loop so we instead settled with the inner one. As usual we used critical section to avoid race conditions. This simple method produced speedups of about 3.2x. If we manage to parallalize the outer loops too, the speed up would probably be higher.

We then used the block matrix algorithm which essentially divides the problem into smaller subsections. The elements in C matrix are now calculated partially from each block and are later summed up. Thus, each block is a quasi independent region. This algorithm improves data reuse and is thus more efficient

We were also looking at the possibility of parallalizing the work of these blocks. However the block code had 5 loops and it was difficult to imagine how pragma omp for would behave. We tried parallalizing the outermost loop but the 5 fold nesting was too complicated to analyze and the results were not proper. In future, we will take a look at how to parallalize nested loops. In this context we came across a keyword 'collapse' but are not sure of its usage.

One of the major inferences is the efficiency of the block algorithm and if we are able to parallelize the block method then it even might be possible to achieve super-linear speedup.