Hardware Details

CPU Model : INTEL(R) Core i5-4590
No. of cores: 16
Memory: 7.6 GiB
Compiler: gcc
Optimization flags if used: None
Precision: double

Possible Speedup (Theoretical):

Speedup = 1 / (S + (P/M)) = M(approx.)(Code can be fully parallelized. P = 1)

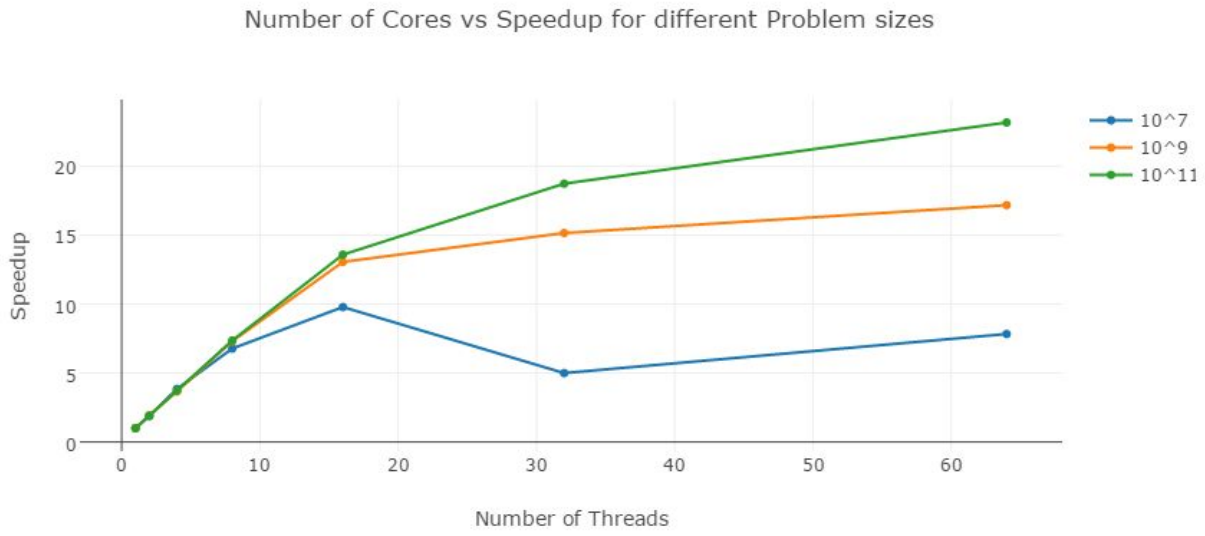
where M is no. of cores, P is fraction of code that can be parallelized and S is serial fraction of code. P+S=1.

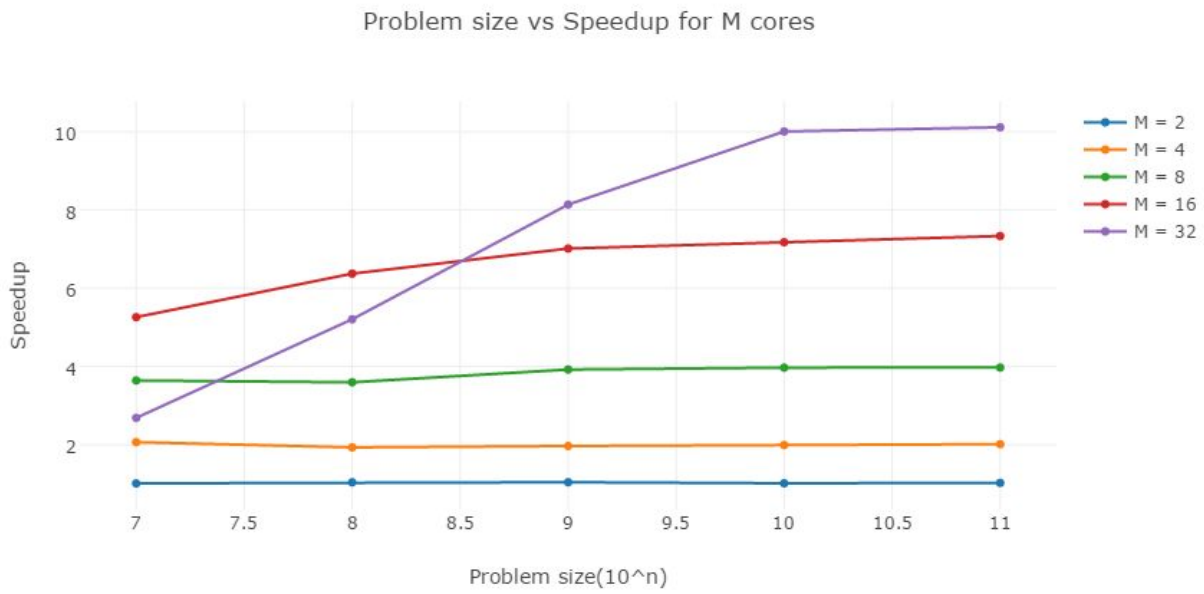
1.Calculation of PI using Series

$$\pi = \sum_{k=1}^{n} \frac{(-1)^k}{(2k-1)}$$

Complexity of the Problem (Serial):
The complexity of the problem is O (n), where n is number of terms which we take in the series for the calculation of pi.

Optimization:
We divide the n(number of terms of series mentioned above) among the p (number of threads) and each thread will calculate its partial sum and then add it to the global sum.By using the reduction operator we have avoide any possible race condition that can happen at sum variable due to each thread accessing it.





Problems in parallel code and possible solutions:

1. We could have a race condition if we didn't use the reduction operator and in case of not using a reduction operator we can have a critical section created where value of sum is found out by all the threads

2.Generating Fibonacci Series

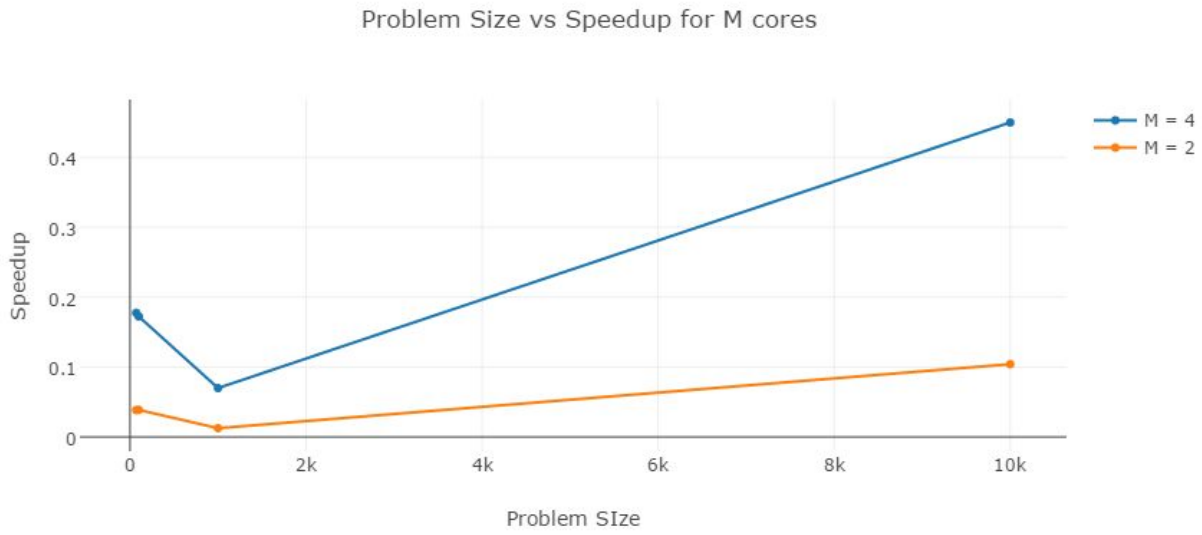
Here we try to generate fibonacci series for a large n using both serial as well as a parallel code(V1 , V2 and V3).
For V1 , we used OMP_CRITICAL section.
For V2 , we used the formula:
F(n+k)=F(n-1)*F(k)+F(k-1)*F(n) , and parallelize the code based on the above equation.
For V3 , we used task directive.

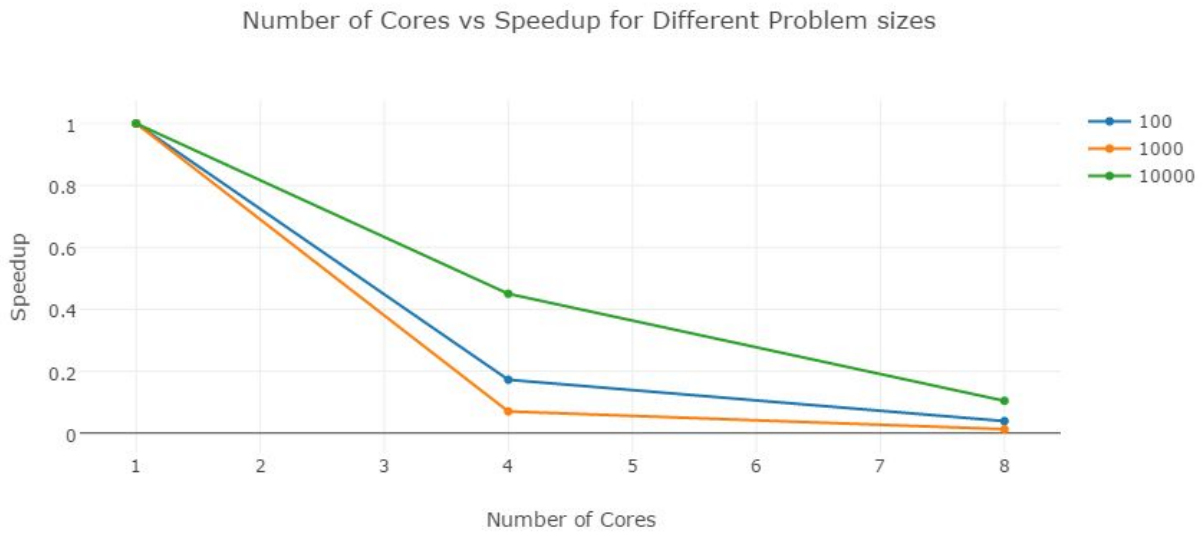
Complexity of Fibonacci Series:
Time Complexity: O(n)
Extra Space: O(n)

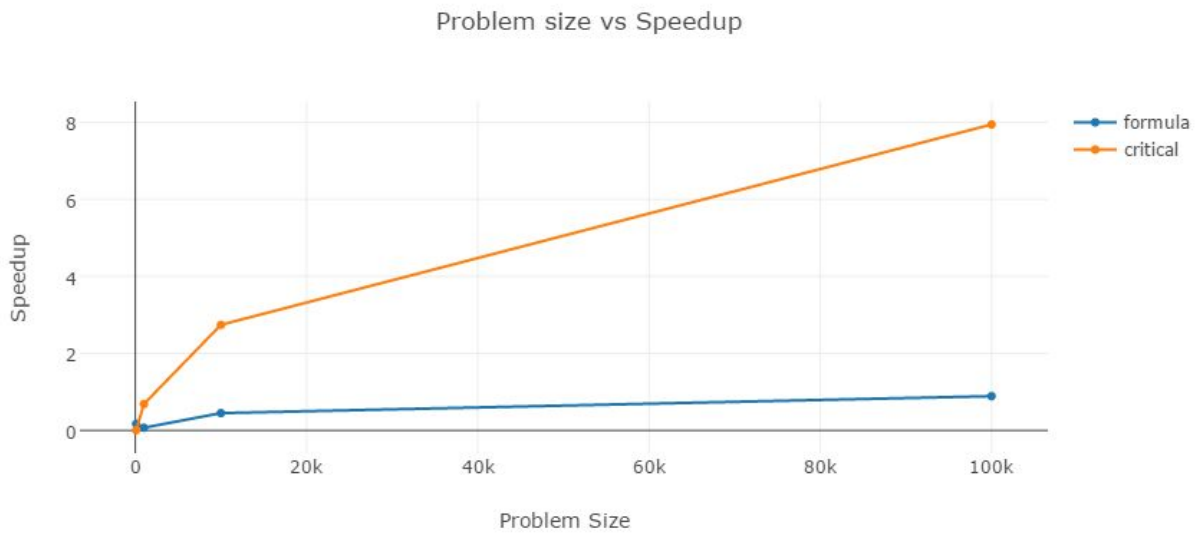
Optimization Strategy:
In serial code ,every next number(except 0 and 1) is found by adding up the two numbers before it.We used an iterative method to accomplish the task.
In V1, we created a critical section in fibonacci code where a fibonacci(n) depends on the values that number 'n-1' and 'n-2' take.By critical,only one thread could execute the code at a time
In V2,below formula is executed:
F(n+k)=F(n-1)*F(k)+F(k-1)*F(n)

and the number ‘n’ is divided into number of cores ‘p’.This formula helps to have more numbers in advance without actually reaching till that point .Ex.for n=10 and k=5,we can know the values of F(5),F(6),F(7),F(8),F(9) and so on [given that we have values for F(i),i is 1,2,3,4)]...Here , in each iteration thread execute different set of numbers(with different cardinality).
In V3,task directive is used.The call to fib(n) generates two tasks, indicated by the **task** directive. One of the tasks computes fib(n-1) and the other computes fib(n-2), and the return values are added together to produce the value returned by fib(n). Each of the calls to fib(n-1) and fib(n-2) will in turn generate two tasks. Tasks will be recursively generated until the argument passed to fib() is less than 2. The taskwait directive ensures that the two tasks generated in an invocation of fib() are completed (that is. the tasks compute i and j) before that invocation of fib() returns.

Input	Serial time	parallel time	speedup
10	0.000086	0.000589	0.1460109
20	0.000089	0.017992	0.049464
30	0.000112	2.178956	0.0000516
35	0.000289	30.17834	0.0000958
40	0.000130	58.12333	0.0000224


Problem Size vs Speedup for M cores
Speedup
0.4
0.3
0.2
0.1
0
0
2k
4k
6k
8k
10k
Problem SIze
M = 4
M = 2


Number of Cores vs Speedup for Different Problem sizes
Speedup
1
0.8
0.6
0.4
0.2
0
1
2
3
4
5
6
7
8
Number of Cores
100
1000
10000



Observations made from graphs:
For n>94 ,in case of parallel (V1) the code doesn't yield proper values(out of bound values).
F_{94}=: 19740274219868223167 =1.97*e^{19}
In V1 ,we used OMP_CRITICAL and find out that our results were similar to that of serial code for smaller values of n(up to 10^6) and then serial code took more time than our given parallel code.
In V2,as opposed to V1, more efficient parallelization has been done in this using the above mentioned formula.
In V3,we use task directive.Tasking facilitates the parallelization of applications where units of work are generated dynamically, as in recursive structures or **while** loops.The task construct is placed whenever a thread encounters a task construct, a new task is generated.
This parallelization didn't bore any results as the speedup kept on decreasing and moreover for n>35,the code took a very long time.

Problems in parallel:

1. As the stack size grew in number due to increasing value of the input parameter,the fibonacci number that were generated from the parallel code(as well as serial code) did not generate correct value i.e due to memory bound constraints , our fibonacci could be generated only upto F_{94}=: 19740274219868223167 =1.97*e^{19} even suing long long data type.
2. Task directive used , took more time from the original serial code ,
3. In V1,critical section is fib(n)=fib(n-1)+fib(n-2),this more or less makes the code less efficient when it comes to parallelization as we are allowing only a single thread to execute at a time hence no efficient parallelization.

4. In V1,there is dependencies in iterations that makes a code less efficient.

3.Bubble Sort

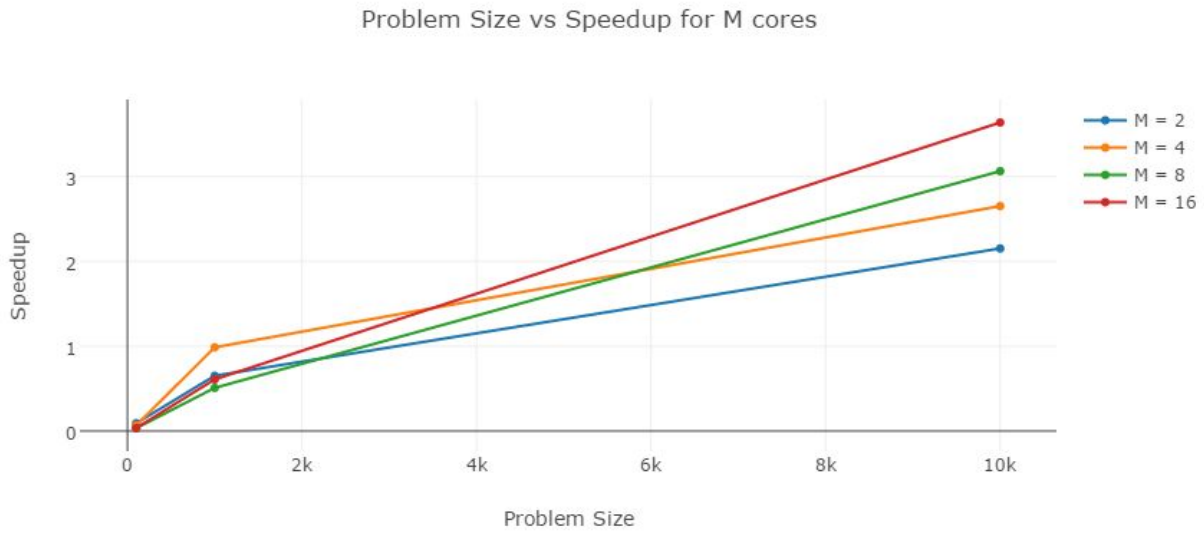
Sort a array of a very large size 'N' with parallel and serial code implementation.In serial code ,It works by repeatedly exchanging adjacent elements, if necessary. When no exchanges are required, the file is sorted.In parallel, it functions by comparing all odd/even indexed pairs of adjacent elements in the list and, if a pair is in the wrong order (the first is larger than the second) the elements are switched. This is repeated for even/odd indexed pairs (of adjacent elements). Then it alternates between odd/even and even/odd steps until the list gets sorted.

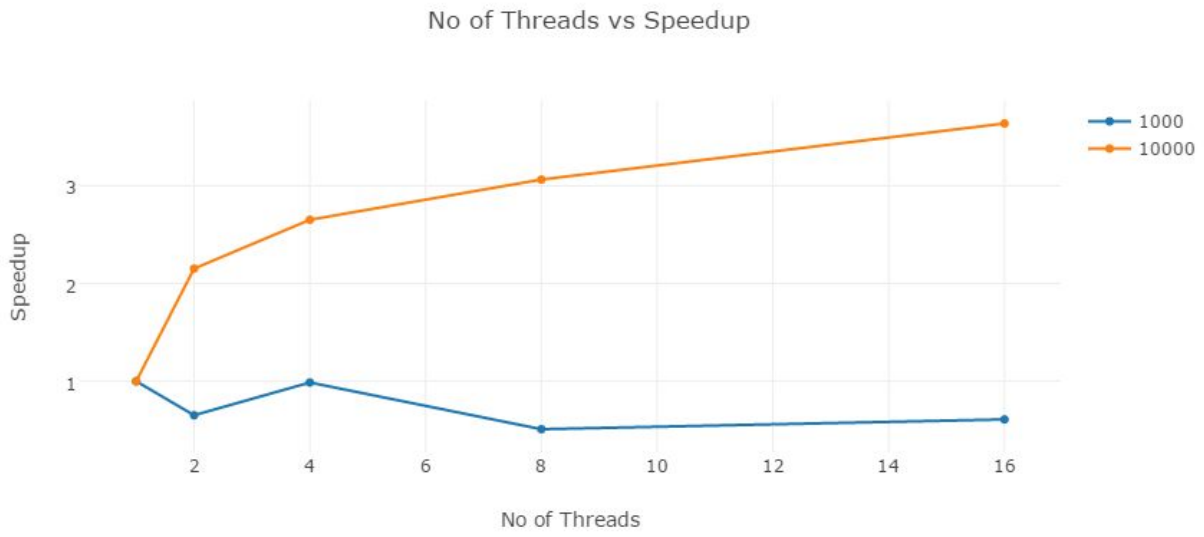
Complexity:

The sorting algorithm ends its execution after a number of maximum 'n' iterations of the main loop.
(In case of parallel , two phases do n/2 iterations each thus at the end completing n iterations)
The number of comparisons will be' n-1' and the number of swaps can be 2 approximated by (n-1)/2 . The minimum number of iterations is 1 , in the case when the elements of the array are already sorted. In this case the algorithm will perform n-1 comparisons and 0 swaps. Based on these results we can conclude that the complexity level of the algorithm for a common array is $O(n^2)$.Best case complexity will be O(n).

Optimization Strategy:

The parallel version can be obtained if we use the odd-even transposition method that implies the existence of 2 phases, each requiring n/2 compare and exchanges. In the first phase, called odd phase , the elements having odd indexes are compared with the neighbors from the right and the values are swapped when necessary. In the even phase , the elements having even indices are compared with the elements from the right and the exchanges are performed only if necessary.





Observation:

1.For the bubble sort the serial code runs little faster than the parallel code(dependencies in iterations) for $n<10^5$ but if we increase input parameter more then the speedup tends to increase for a larger value of n.

Problems in parallel code and possible solutions:-

1. The parallel code takes more time than the serial code i.e parallelizing the code with 4 threads didn't improve the speedup much,one of the possible reasons are : loop dependencies.Instead of using odd-even transportation for parallel implementation of bubble sort its better to use odd-even merge sort that has better efficiency.

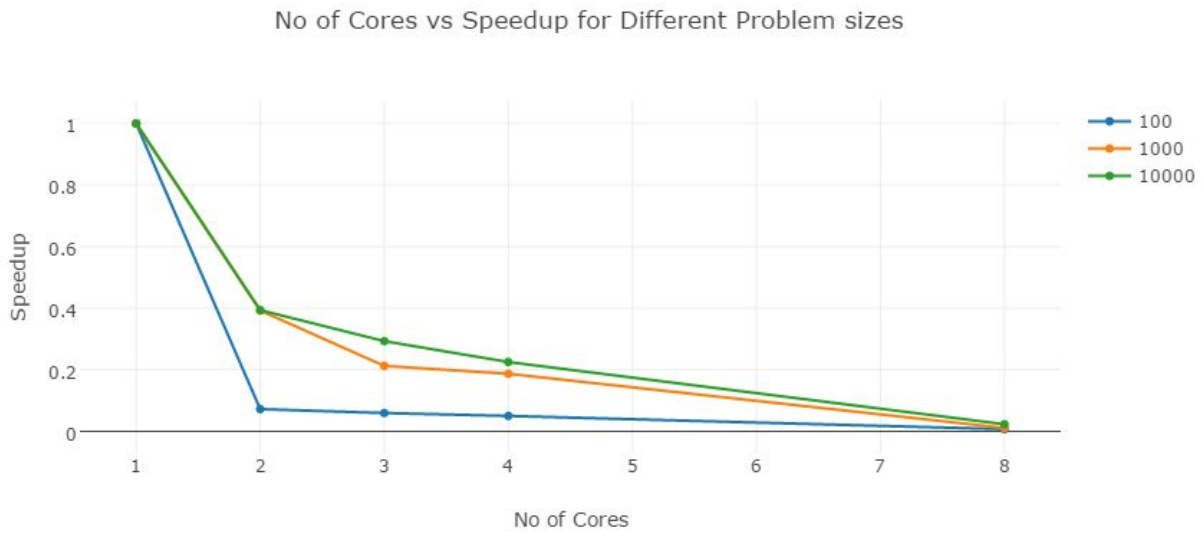
4.ODD - EVEN SORT:

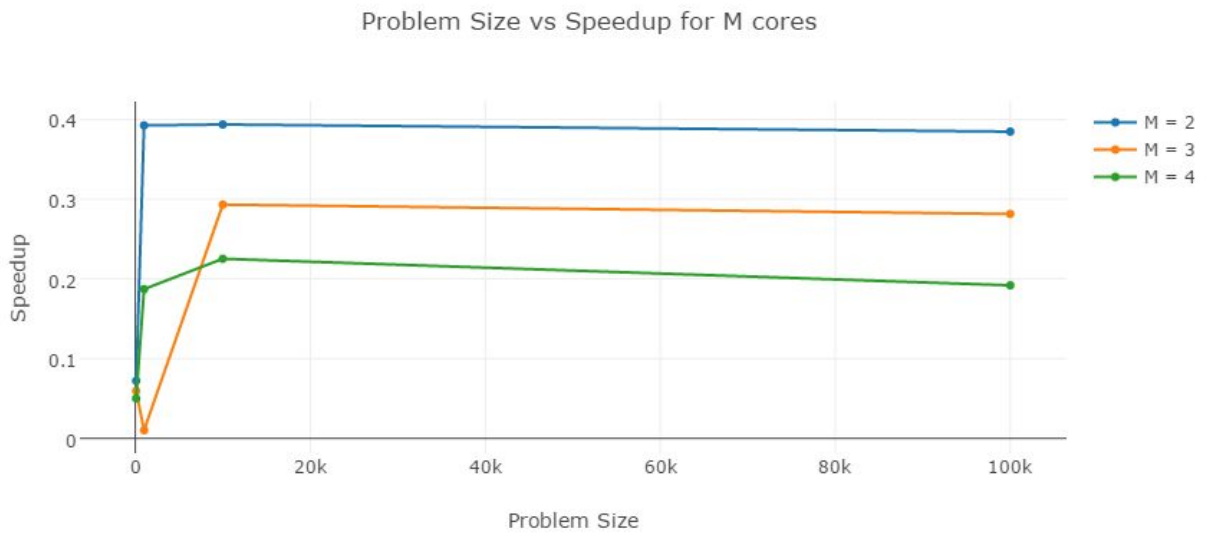
It is a comparison sort related to bubble sort, with which it shares many characteristics.
Serial:It functions by comparing all odd/even indexed pairs of adjacent elements in the list and, if a pair is in the wrong order (the first is larger than the second) the elements are switched. The next step repeats this for even/odd indexed pairs (of adjacent elements). Then it alternates between odd/even and even/odd steps until the list is sorted.Parallel:same procedure but threads are created to work on two different phases namely,ODD and EVEN phases.

Optimization Strategy:

1. It starts by distributing n/p sub-lists (p is the number of processors and n the size of the array to sort) to all the processors.
2. Each processor then sequentially sorts its sub-list.
3. The algorithm then operates by alternating between an odd and an even phase :
    1. In the even phase, even numbered processors(processor i) communicate with the next odd numbered processors (processor i+1). In this communication process, the two sub-lists for each 2 communicating processes are merged together. The upper half of the list is then kept in the higher number processor and the lower half is put in the lower number processor.
    2. In the odd phase, odd number processors (processor i) communicate with the previous even number processors (i-1) in exactly the same way as in the even phase.

Complexity:
This algorithm is not efficient, the average complexity is $0(n^2)$.



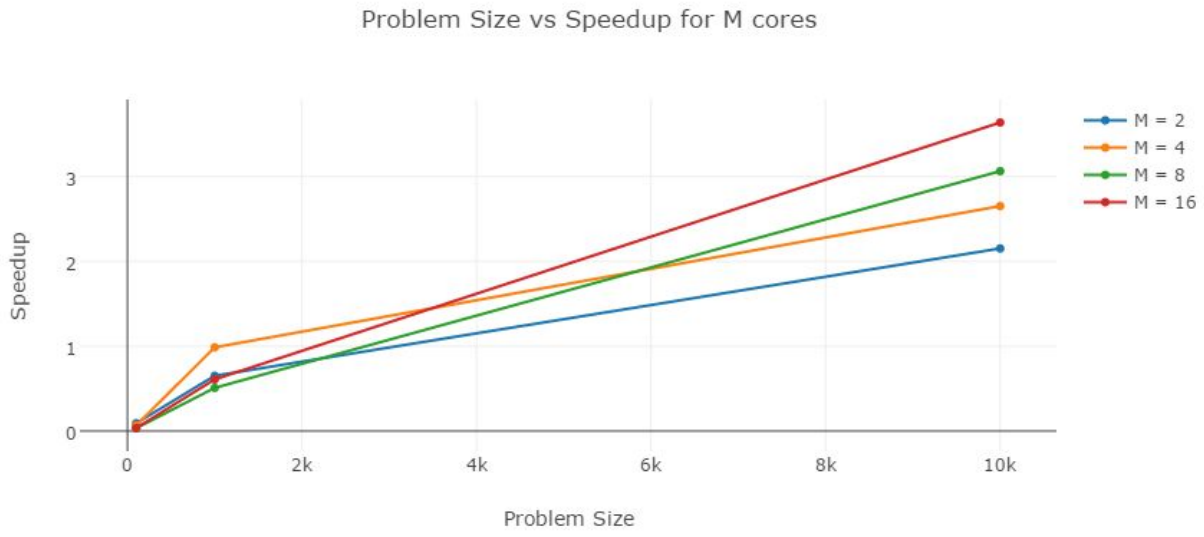


OBSERVATIONS:

For the odd-even sort the serial code runs little faster than the parallel code(dependencies in iterations) for $n<10^6$ but if we increase input parameter more the speedup tends to increase for a larger value of n.

Problems in parallel and possible solutions:-

1. The parallel code takes more time than the serial code i.e parallelizing the code with 4 threads didn't improve the speedup much,one of the possible reasons are : loop dependencies.To improve the parallel code we could follow the implementation of a bitonic code. In a bitonic the complexity is $O(\log^2 n)$ and thus is more efficient than $O(n^2)$,it is a sorting network , where the sequence of comparisons is not data-dependent. This makes sorting networks suitable for implementation in hardware or in parallel processor arrays. The sorting network bitonic sort consists of $\Theta(n \cdot \log(n)^2)$ comparators. It has the same asymptotic complexity as odd-even mergesort .
2. For values greater than 10^7,the speedup increases (>1)but for $n=10^7$,time taken is very large.


Problem Size vs Speedup for M cores
Speedup
0
1
2
3
0
2k
4k
6k
8k
10k
Problem Size
M = 2
M = 4
M = 8
M = 16