FIBONACCI SERIES GENERATOR

By Deeksha Koul(201301435)

The Fibonacci Sequence is the series of numbers: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34,55, ...
The next number is found by adding up the two numbers before it.
We generated the given fibonacci series using serial code and a parallel code.

Hardware Details:
CPU Model : INTEL(R) Core i5-4590
No. of cores: 16
Memory: 7.6 GiB
Compiler: gcc
Byte Order: Little Endian
CPU(s): 32
On-line CPU(s) list: 0-31
Thread(s) per core: 2
Core(s) per socket: 8
Socket(s): 2
CPU MHz: 1200.000
L1d cache: 32K
L1i cache: 32K
L2 cache: 256K
L3 cache: 20480K

Brief introduction:
In serial code,we used dynamic programming to evaluate fibonacci numbers and stored them instantly.The formula used for the same was:

F[i] = F[i-1] + F[i-2] ;

Over 'n-1' iterations were carried out ,given that we knew that F[0]=0, F[1]=1.
Time complexity for this serial code was : O(n).

For parallel code:
In V1,we used omp_parallel over the calculation of fib(i) using the previous two values and see how openMP divides the iterations over the threads.

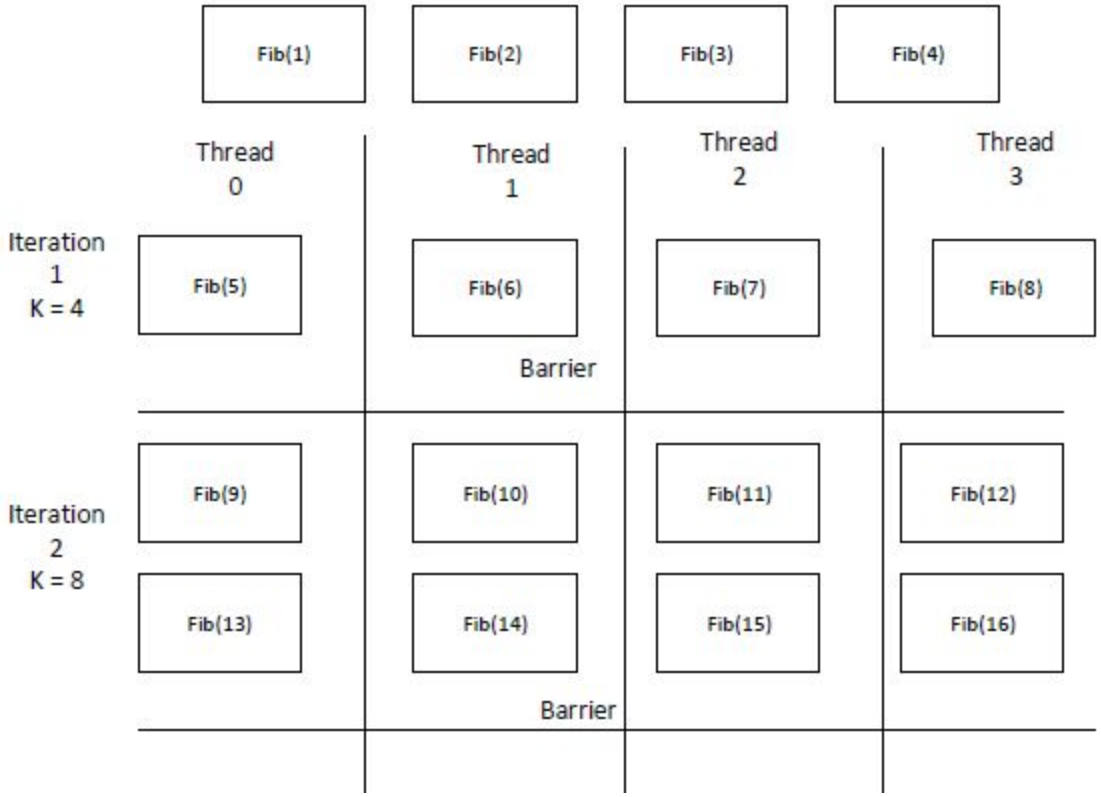
In V2,we used below defined formula for generating fibonacci series:

F(n+j) = F(n-1) * F(j) + F(j-1) * F(n),

In each iteration, each thread calculates value of F(n+j) independent of other threads using the values of fibonacci numbers already available. When it encounters a data

dependency, it waits for other threads to complete their work, due to conditions mentioned in the code.
e.g. We provide values of F(0) to F(4) at the beginning of the program. In the first iteration threads 0 - 3 calculate values of F(5) to F(8) independently(as j=4). Then they wait for all threads to complete.

$$Fib(n + k + 1) = Fib(n+1)*Fib(k) + Fib(n) * Fib(k+1)$$



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)

```
#pragma omp task shared(i) firstprivate(n)
        i=fib(n-1);
```

and the other computes fib(n-2),

```
#pragma omp task shared(j) firstprivate(n)
        j=fib(n-2);
```

and then the return values are added together to produce the value returned by fib(n).

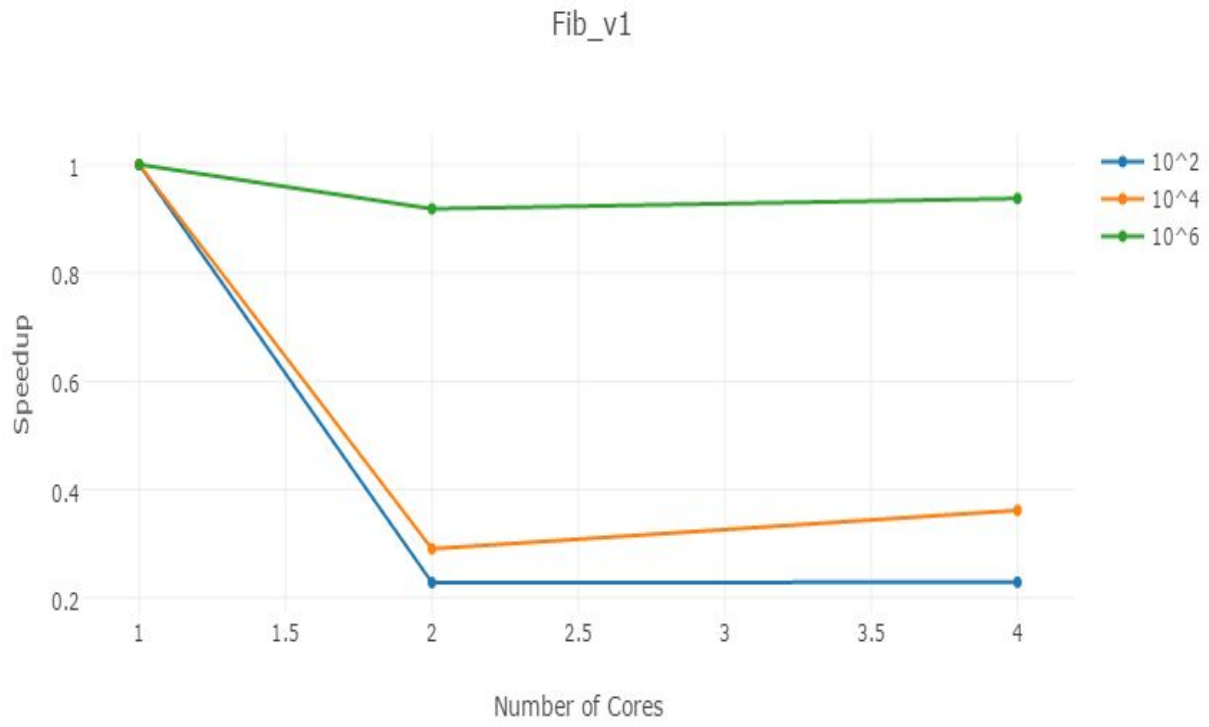
Each of the two tasks will be recursively generating fib() until the argument passed to fib() is less than 2.

```
#pragma omp taskwait
        return i+j;
```

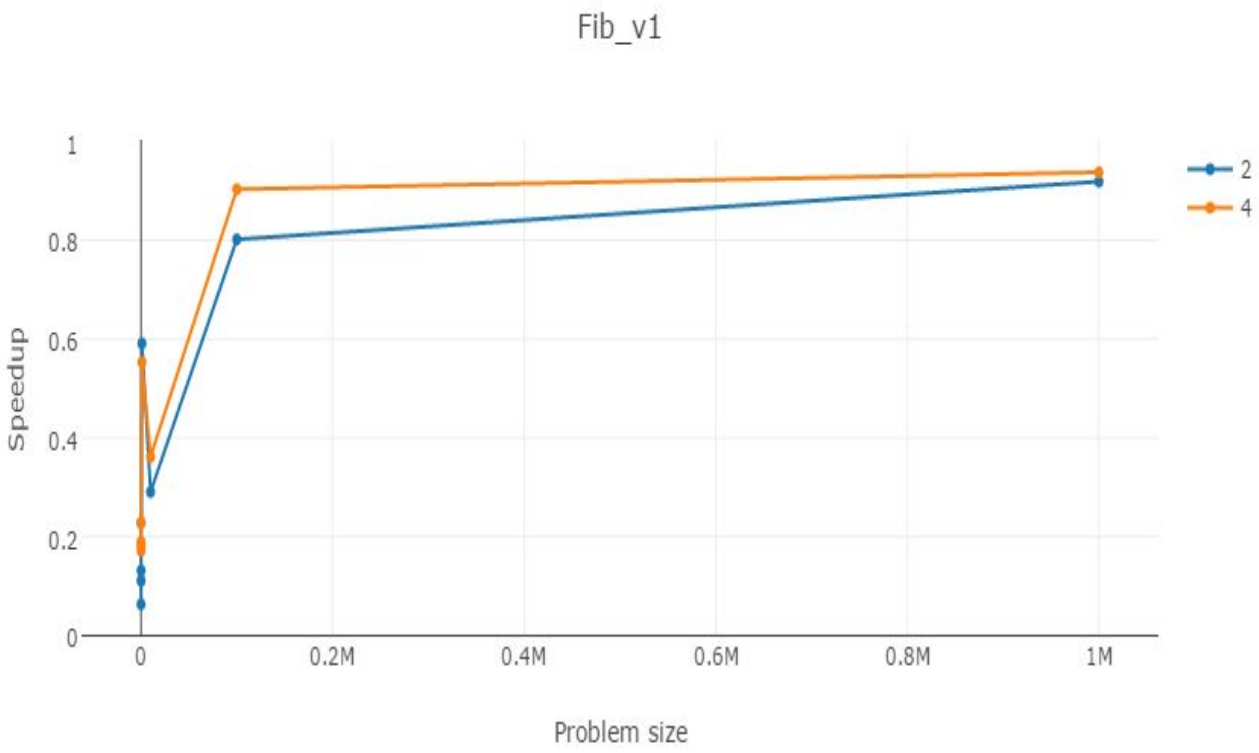
The taskwait directive ensures that the two tasks generated for fib(n-1) and for fib(n-2) are completed (that is. the tasks compute i and j) before that invocation of fib() returns.
In main method of V3,

```
#pragma omp single
 fib(n);
```

So,that only one available thread will be able to run this section of code i.e calling the fib(20) , for n =20 and then the other threads will do task of calculating fib(n-1) and fib(n-2)

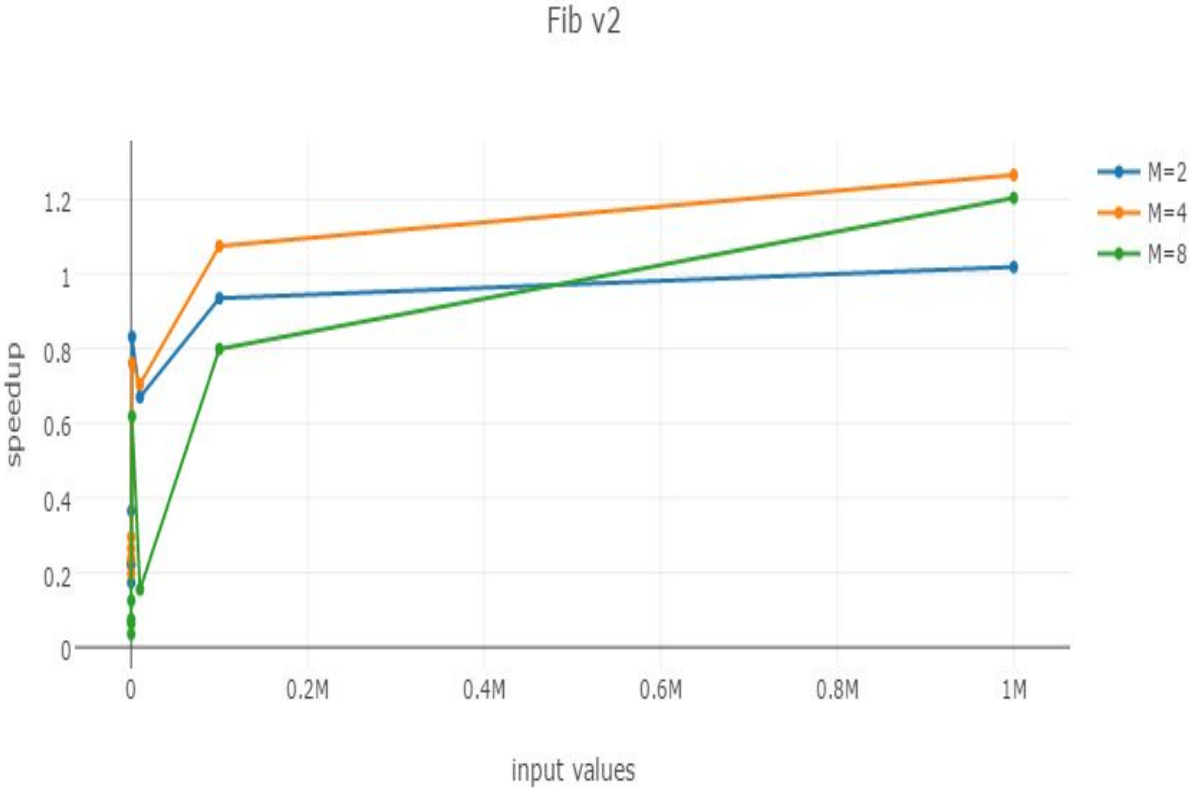


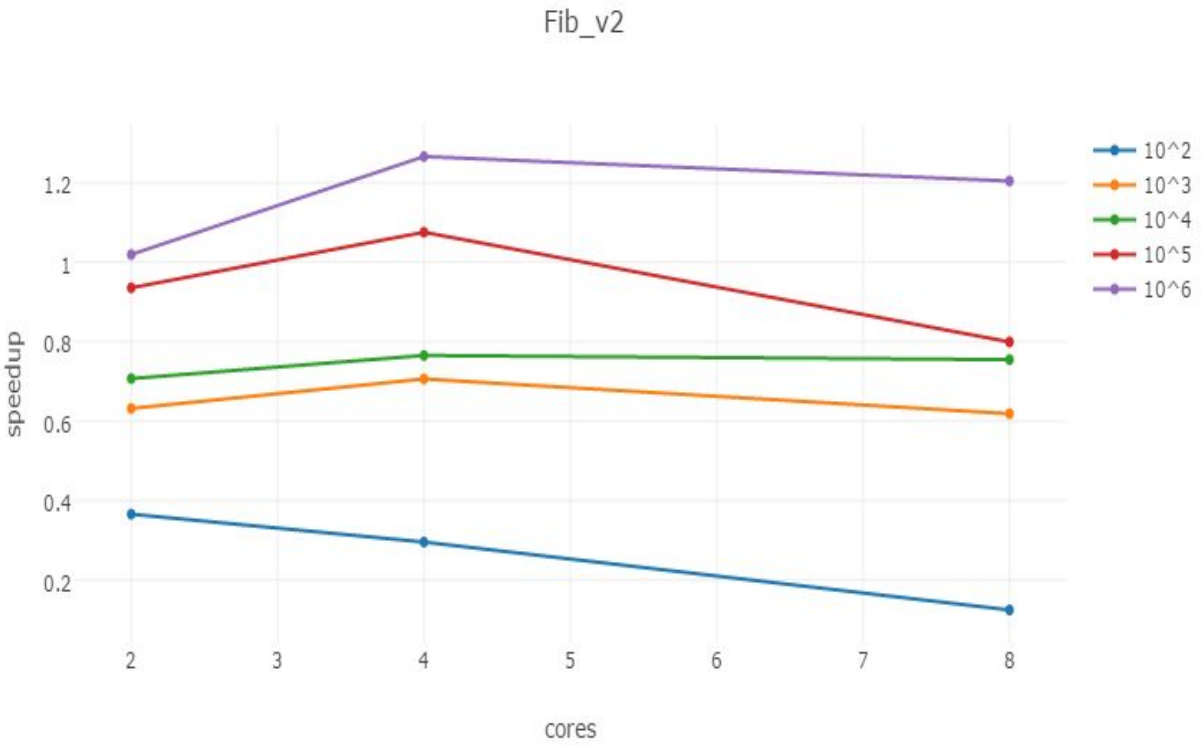
For Input as 10^2,10^4,10^6.



For cores as M=2 and M=4.

Observation for V1:
For a specific core say M=4,the speedup increases with input size and when the number of cores increases from 2 to 4 the speedup tends to increase for each different values of input.As the the number of cores increases to M=8 the pattern remains same i.e for larger input,the speedup for M=8 is incraesed and is better (wth less difference)than M=4.

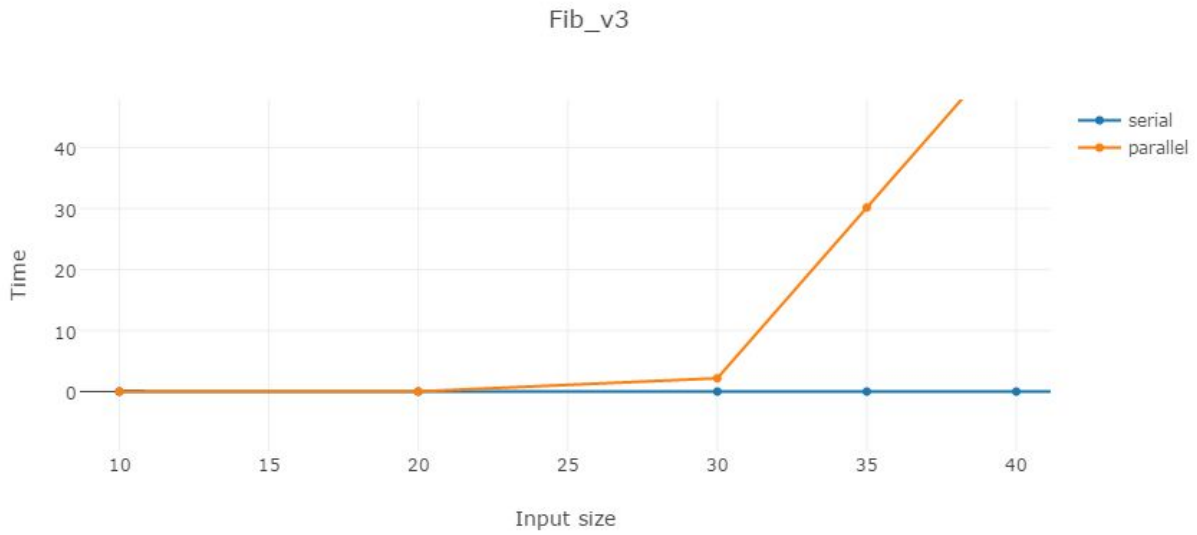

Fib v2
speedup
1.2
1
0.8
0.6
0.4
0.2
0
0
0.2M
0.4M
0.6M
0.8M
1M
input values
M=2
M=4
M=8



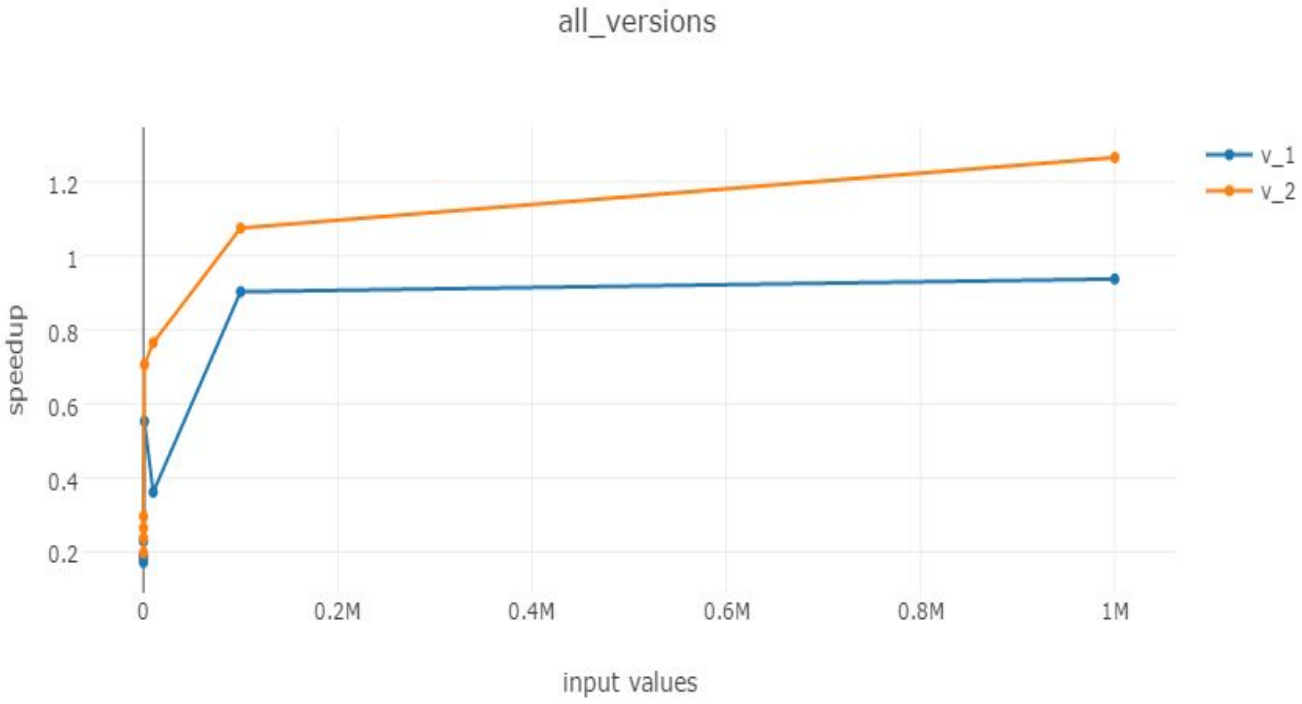
Observation for V2:

For inputs $N<10^5$ the serial code takes less time as compared to all the parallel versions of the code but as N grows larger in size the parallel code improves its efficiency (specially for M=4).i.e For each M = 2,4,8 the parallel time becomes less than the serial time and hence the speedup approaches to 1 and for $N>10^6$, speedup>1.

For M=8,speedup increases considerably only after $N=10^5$ and therefore for larger the speedup for M=8 will become more.



Observations for V3:
The serial code is highly efficient than the parallel code and as N grows larger is in size the parallel code takes more time to generate series thus lowering down the speedup to less than 0.05. This parallel code is using recursion whereas serial code the methodology is based on dynamic programming and hence serial code works better.But the same recursion code when run serially will take more time than the code in Version_3.



Observation:
The version_2 works better than version_1 as input increases for M=4,the structure of graph remains same for M=8,16 also.Therefore , Version_2 is better a algorithm as it efficiently uses and thus increases scope of parallelization , the time wasting in waiting for all threads to complete is still less in this algorithm as compared to the taskwait in version_3 and synchronization in version_1.

Efficiency of the best of the three proposed algorithms:

E=Speedup/Number of cores

here in all the versions,specially in version_2 the efficiency decreases as the number of cores increases from M=2,4,8 and so on..Parallelization overhead can be one of the important factor in decreasing efficiency.As calculating Fibonacci series includes lot of communication between the threads there is a possibility that the synchronization overhead creates a hindrance in efficiency.

Input size is 10^6			
Version_2	Cores=2	Cores=4	Cores=8
Efficiency	0.509554	0.316403	0.150507

Problems faced in parallelization of code:

1. In V1 & V2:

- case of data dependency,the program statement (Fib(i)=Fib(i-1)+Fib(i-2)) refers to the data of a preceding statement , that is why it some amount of time gets wasted in waiting for all the threads of a team.
- segmentation fault (core dumped):accessing memory which is out of scope or not having the permission to access.(After 10^6.)
- the code(V1) due to use of omp_parallel generates fibonacci series of any arbitrary size.Ex. if N=1000, the code will be executed and correct values will be generated till N=200.

2. In all the versions and serial code , the fibonacci series could only be generated upto N=92,F_{92}=7540113804746346429,for N>92 fibonacci numbers generated were given either negative values or zero as an answer due to memory limitations.
3. In V3,the speedup for even large number of n remains less than 0.5.The problem is that for every call(task) say fib(n-1) :it recursively generates its own two task and then each task generates its own two task thus forming a binary tree of very large size and thus making it difficult for execution.

SPEEDUP FOR V1 , V2 , V3

Input	Serial time	M=2	Speedup for M=2	Parallel time	Speedup for M=4
50	0.000017	0.000268	0.063432836	0.000099	0.171717172
70	0.000031	0.000279	0.111111111	0.000170	0.182352941
80	0.000038	0.000289	0.131487889	0.000200	0.19
10^2	0.000071	0.000311	0.22829582	0.000310	0.229032258
10^3	0.000198	0.000335	0.591044776	0.000358	0.553072626
10^4	0.000311	0.001070	0.290654206	0.000860	0.361627907
10^5	0.001259	0.001570	0.80159032	0.001389	0.903003033

10^6	0.012348	0.013446	0.918373649	0.013360	0.937376801

Input	M=2	M=4	M=8
50	0.173469388	0.197674419	0.035416667
70	0.221428571	0.264957265	0.064049587
80	0.224852071	0.2375	0.07495069
10^2	0.365979381	0.295833333	0.125
10^3	0.631932773	0.7061538462	0.61875
10^4	0.7070258621	0.76521542	0.754649428
10^5	0.935364042	1.075149445	0.799303765
10^6	1.019107901	1.26561158	1.204053587

Input	Serial time	parallel time	speedup
10	0.000006	0.000589	0.010186757
20	0.000006	0.007992	0.000750751
30	0.000006	2.178956	2.75361E-06
35	0.000006	30.17834	1.98818E-07
40	0.000009	58.12333	1.54843E-07
50	0.000017	130.1022	0.010186757