Assignment – 1

Tanmay Patel - 201401409
Vaibhav Patel - 201401222

(1) Context:

- **Brief description of the problem.**
  Write a parallel program to compute value of PI using trapezoid rule.And compare with serial program.
- **Complexity of the algorithm (serial).** For serial implementation the complexity is O(n).

- **Possible speedup (theoretical).**

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).

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

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
100.68      4.75     4.75        1     4.75     4.75  main
  0.00      4.75     0.00        1     0.00     0.00  integrate

 %         the percentage of the total running time of the
time       program used by this function.

                        Call graph (explanation follows)


granularity: each sample hit covers 2 byte(s) for 0.21% of 4.75 seconds

index % time    self  children    called     name
                                 1             integrate <cycle 1> [3]
[1]    100.0    4.75    0.00       1         main <cycle 1> [1]
                                 1             integrate <cycle 1> [3]
-----------------------------------------------
                                 1             main <cycle 1> [1]
[3]      0.0    0.00    0.00       1         integrate <cycle 1> [3]
                                 1             main <cycle 1> [1]
-----------------------------------------------
```

- **Optimization strategy.**

Using profiling information we know that the for loop which is doing actual integration is much more expensive. More than 95% of time is taken by this for loop.

So we will try to parallelize it.

At first we used only #pragma omp for and no security for the global variable and got wrong answer. And after printing each output we understood that **updates were lost.**

So we then tried using a global array for each thread and saving approx sum in that array. I.e. each index for each thread. And added all the result in final sum.

After then we tried open-mp directive function **(#pragma omp critical)** but we were not getting any speed up because of global shared variable is accessed by each thread.So the time was actually more than the serial time. So Critical is costly function.

We used a variable for sum for each thread and then added it to global answer sum using #pragma omp critical. As critical function is used only once per thread not much time is used.

Then we used reduction operation (reduction (+: var_name)) which took almost same time as above.

Other thing we noticed was that multiplication operation took much time in naive implementation of both parallel and serial code. Removing it by using previous sums in optimized code reduced time of both serial and parallel code.

- **Problems faced in parallelization and possible solutions.**

In the function which we are integrating we have to know the value of x at every point. In serial implementation we can get the value of x by just adding the value of previous x with the “h”. But in parallel implementation with (#pragma omp parallel for) directive we have to put the value of x as a function of i. And that involves big multiplication. Multiplication operation is very much expensive than addition.

(2). Hardware details: CPU model, memory information, 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

Optimization flag- no flags used but tried O3 in lab,(and got less time than the default one) but vulnerable

(3). Input parameters. Output. Make sure results from serial and parallel are same. Without critical or reduction or different global variables there is a possibility that the answer will contain race conditions.

(4). Parallel overhead time. (openmp version on 1 core vs serial without openmp)

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.

Serial without openmp	**Parallel with 1 core**
1.875455 secs	**1.916974 secs**
100000000 input size	**100000000 input size**
3.141593 answer	**3.141593 answer**

Parallel overhead = parallel - serial

= 0.041519 secs

Observation:

1. For very small input size parallel execution is taking more time than serial execution because of the parallel overhead that is fixed for every execution.
   But for small values of n less than 1e5 the time calculated is very different because of the omp_clock. It has so many ambiguity.
   But for very large input size the parallel is taking much less time.

   2: so from above analogy the speed-up is also first less than 1 then ~1 and then in between 3 and 4.

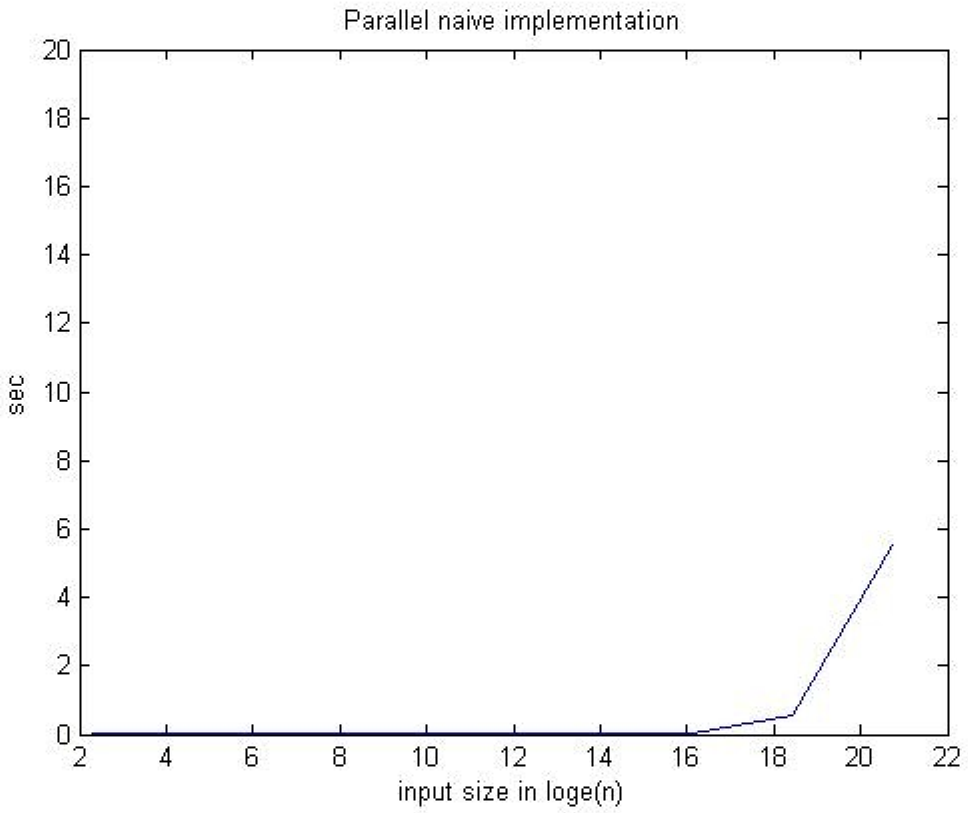
(5). Problem Size vs Time (Serial, parallel) curve. Speedup curve. Observations and comments about the results.

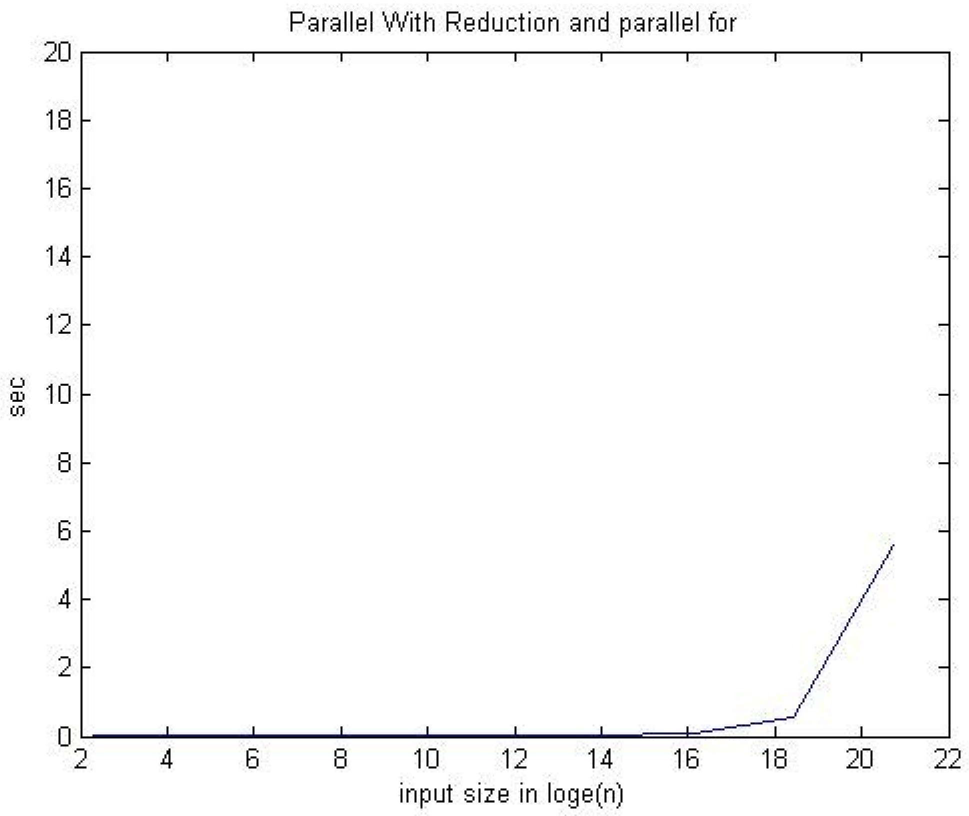
Code for Plotting in Matlab

```
close all;
n=[
     10   ;100;     1000;     10000    ;     100000   ;     1000000   ;
10000000  ;    100000000  ;    1000000000      ];
sn=[0.000001;0.000006;0.000060;0.000601;0.006084;0.024456;0.188777;
1.862289;18.628479];
```

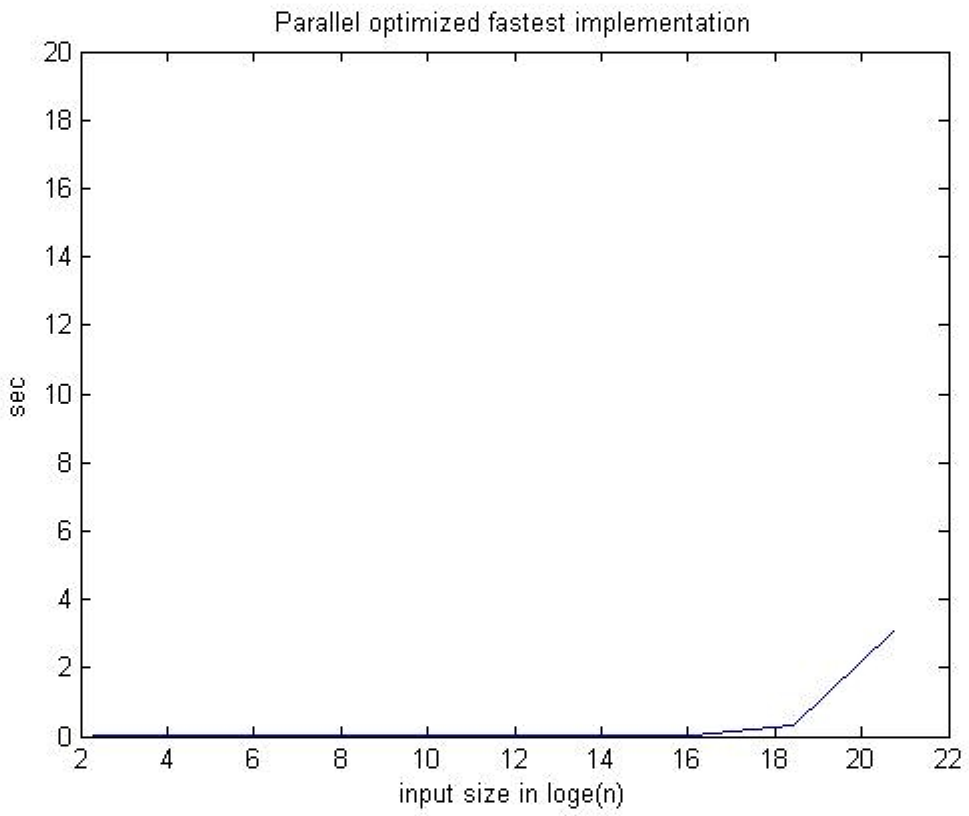
```
plot(log(n),sn)
title('serial naive');
ylabel('sec');
xlabel('input size');
ylim([0 20]);
figure

so=[0.000000;0.000001;0.000007;0.000074;0.000739;0.007570;
0.064303;0.580726;5.593472];
plot(log(n),so)
title('serial optimized algorithm');
ylabel('sec');
xlabel('input size');
ylim([0 20]);
figure


pn=[0.001853 ;0.000012 ;0.000023 ;0.000159 ;0.001558 ;0.012615 ;
0.056613 ;0.546845 ;5.524085
    ];

plot(log(n),pn)
title('Parallel naive implementation');
ylabel('sec');
xlabel('input size');
ylim([0 20]);
figure
po=[
    0.000263 ;0.000012 ;0.000016 ;0.000091 ;0.000928 ;0.006230 ;
0.031636 ;0.314115 ;3.069956
    ];

plot(log(n),po)
title('Parallel optimized fastest implementation');
ylabel('sec');
xlabel('input size');
ylim([0 20]);
figure

pr=[ 0.000037 ;0.000016 ;0.000023 ;0.000070 ;0.000550 ;
0.005786 ;0.058879 ;0.553408 ;5.603246 ];

plot(log(n),pr)
title('Parallel With Reduction and parallel for');
ylabel('sec');
xlabel('input size');
ylim([0 20]);
figure
col=hsv(10);
plot(log(n),sn,'color',col(1,:))
hold on
```

```
plot(log(n),so,'color',col(8,:))
hold on
plot(log(n),pn,'color',col(6,:))
hold on
plot(log(n),po,'color',col(9,:))
hold on
plot(log(n),pr,'color',col(5,:))
hold on
title('Red for serial naive,Green for parallel with reduction,Purple
for Fastest parallel implementation');
ylabel('sec');
xlabel('input size');
ylim([0                                                          20]);
```

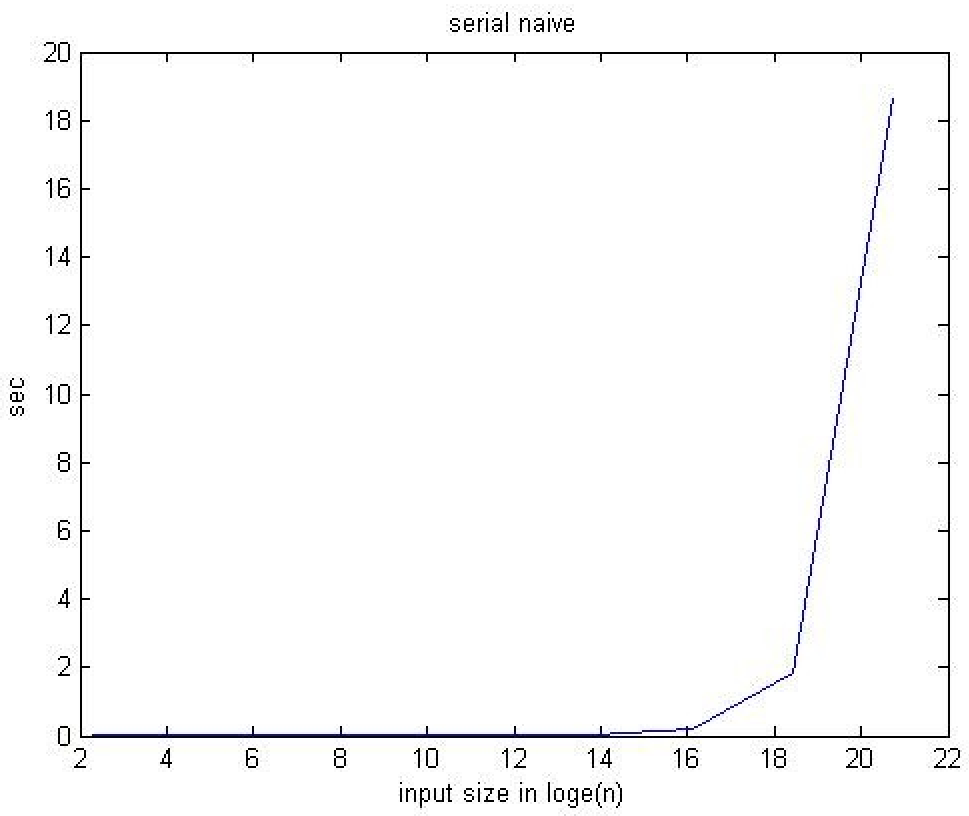


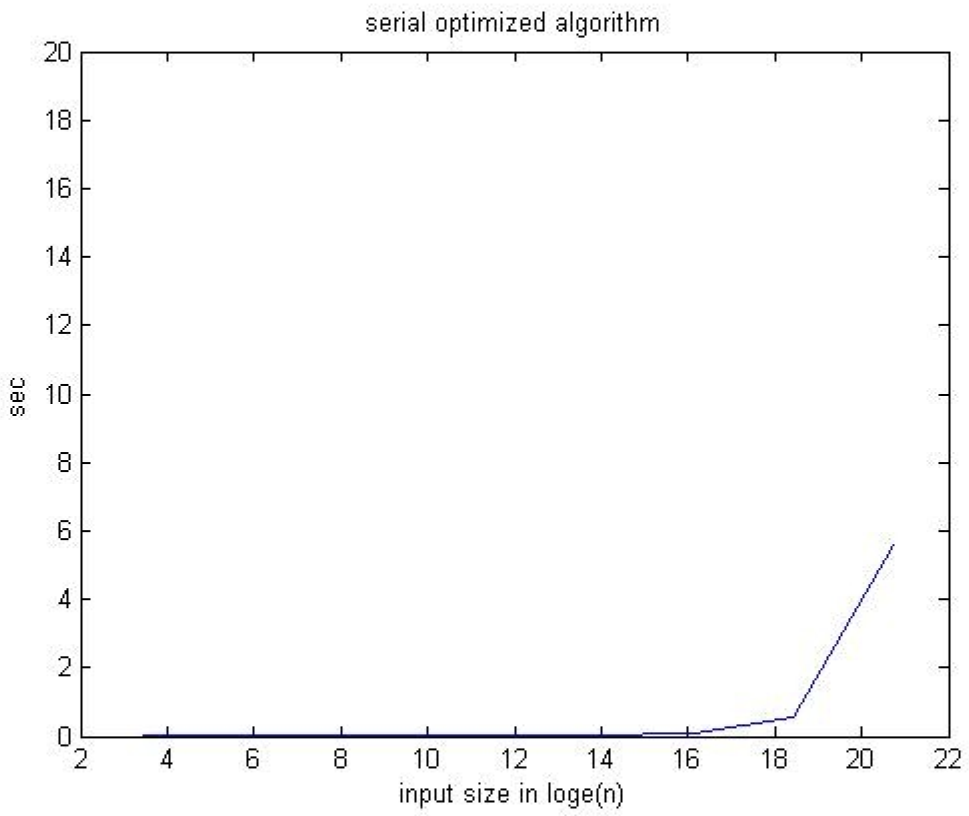


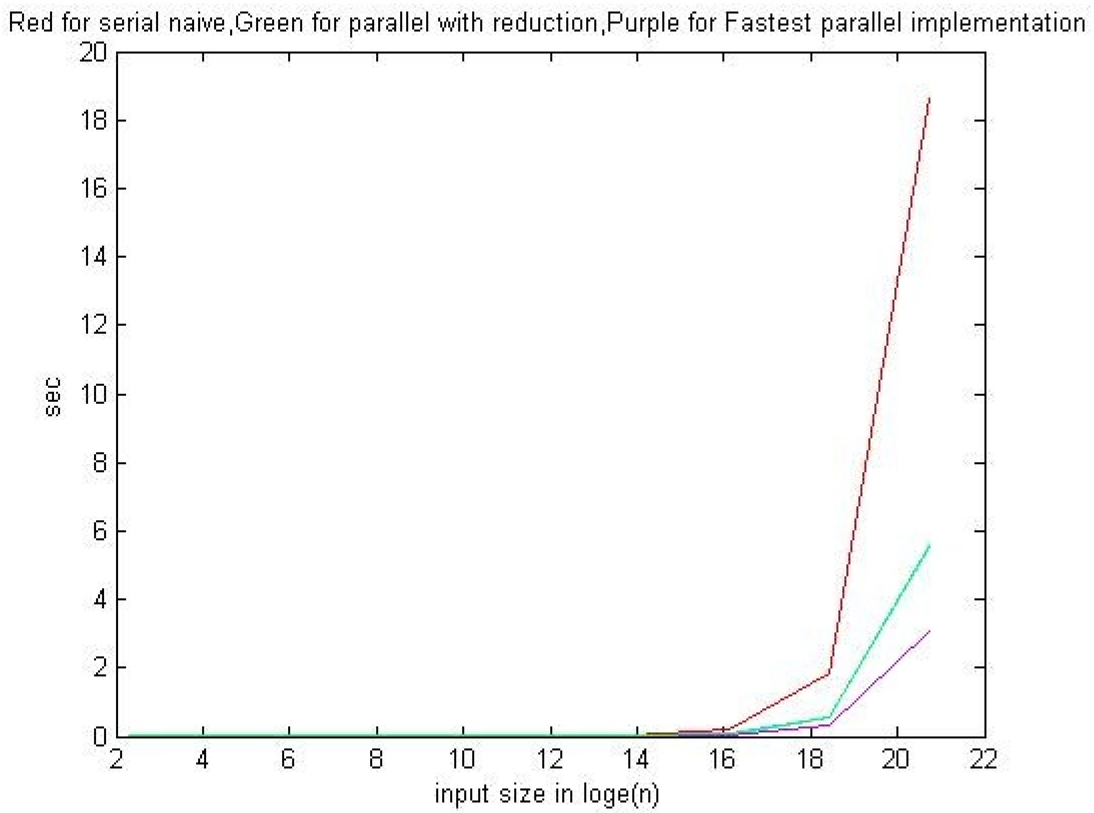
Parallel With Reduction and parallel for
sec
input size in loge(n)
0
2
4
6
8
10
12
14
16
18
20
22



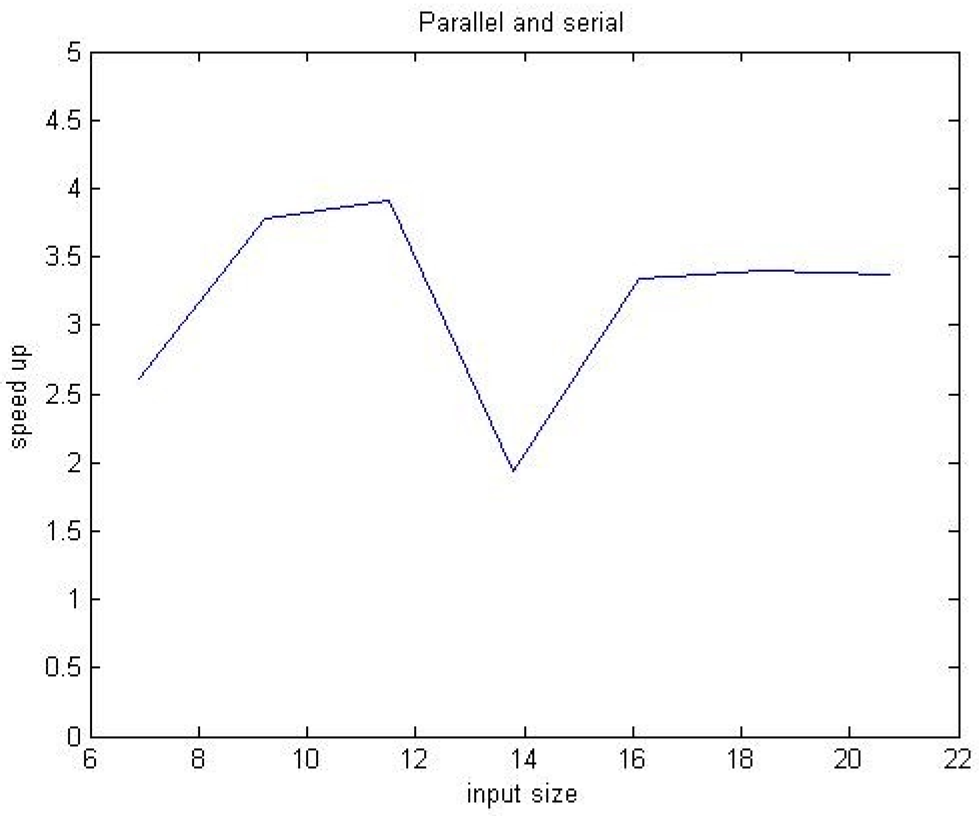
Parallel optimized fastest implementation
sec
input size in loge(n)
0
2
4
6
8
10
12
14
16
18
20
22


serial naive
sec
input size in loge(n)
0
2
4
6
8
10
12
14
16
18
20
22


serial optimized algorithm
sec
input size in loge(n)
0
2
4
6
8
10
12
14
16
18
20
22



Parallel naive vs serial naive



(size in logn)

Performance in MFLOPS/sec. = No of cores * clock rate * no of processor
= 4 * (1.67 / 4) * 4
= **6.68** G**FLOPS / sec**