Prefix Sum
Report

Saket Dingliwal 2015CS10254

Algorithm

The algorithm to compute prefix sum involves doing a forward and a backward pass of adder tree. In the first pass, sum is computed of the sub tree in bottom up fashion. There are n/2 additions at the lowest level, n/4 , n/8 and so on as we move up the levels. These additions are done in parallel on different cores. There will be log n such levels. At the end of the first pass, the last element stores the sum of all the elements. The n/2th element contains the sum uptill n/2 elements and so on. Also there are n/2 + n/4…… = O(n) computations. The last element is saved and zero is put at that place for the next pass. In this pass, zero moves from nth to n/2th to n/4th element and prefix sum[i] gets stored in i+1 th position. At the end of the pass, first element contains zero.

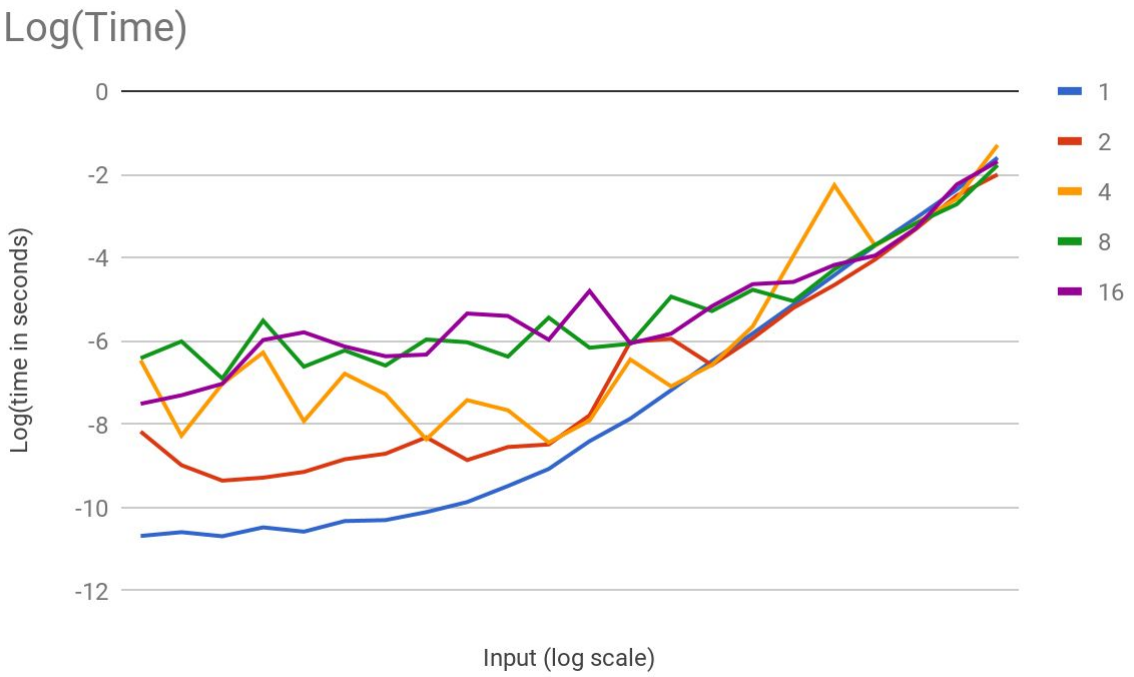
Implementation

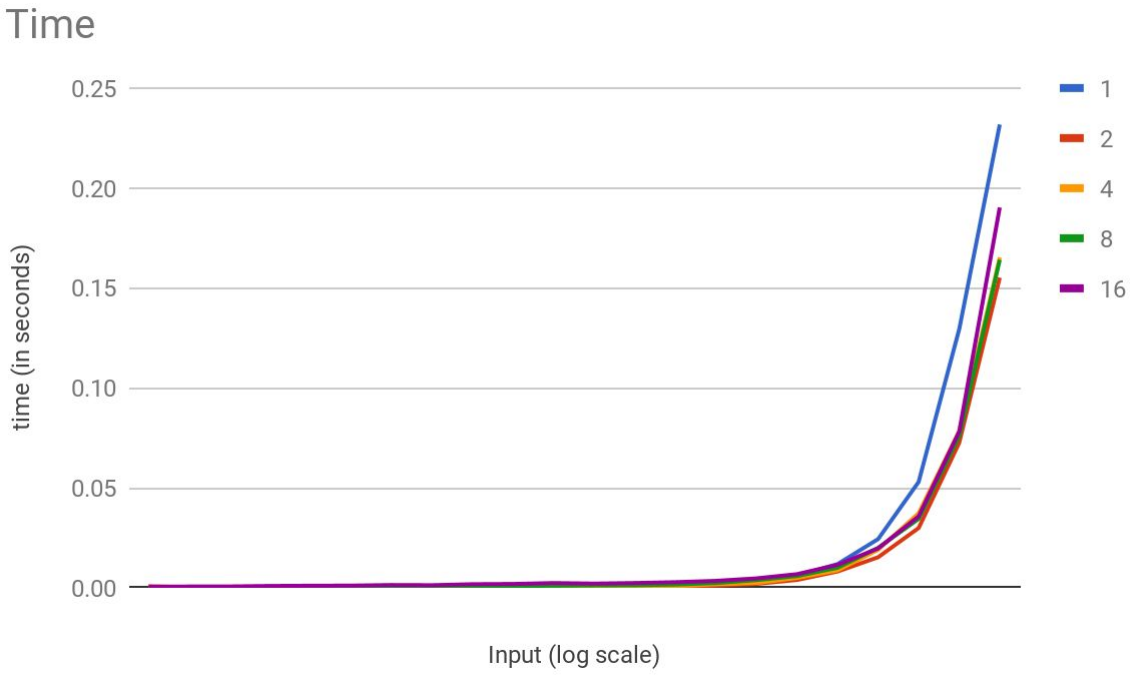
The parallel implementation is done using Open MP. For each level, a team of p threads are invoked that perform the required number of additions at that level. OMP parallel for construct is being used, which divide the iterations equally among the threads. Next level begins only when all the threads of that level have completed their tasks. No shared variable except the input array itself is used among the team. All threads write to different positions in the array which are separated by more than 4 words and hence eliminating the need of padding. The use of the parallel for loop nested inside a while loop can be seen through snap of the code below.

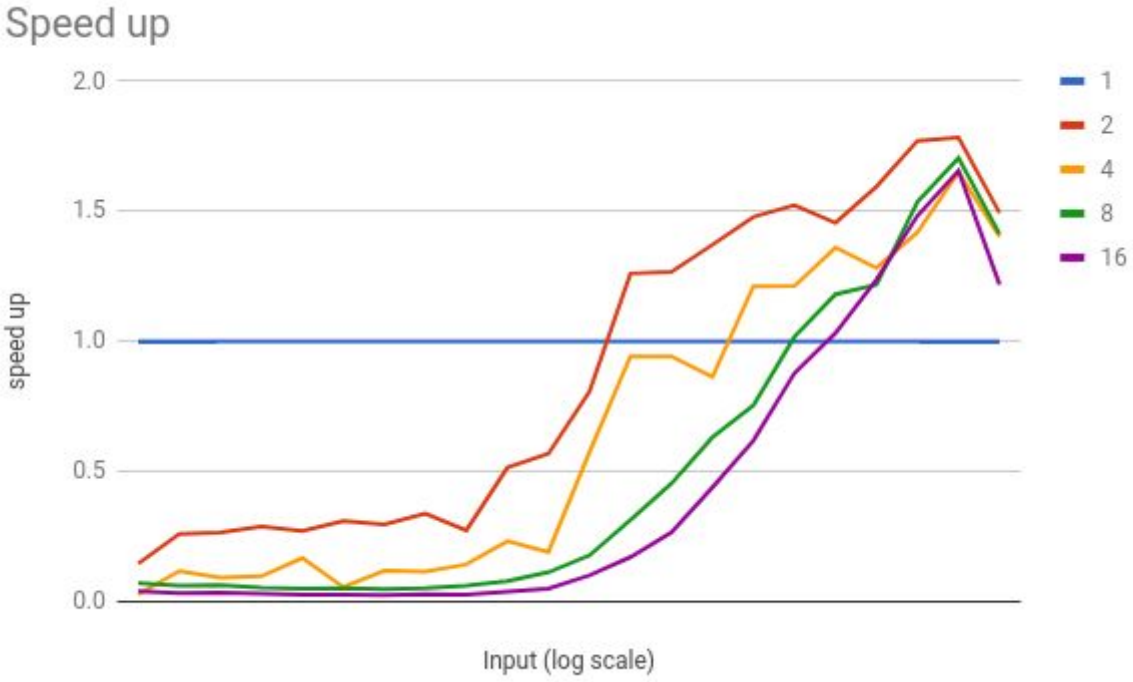
```
int level = 2;
while(level<=n)
{
    int untill = n/level;
    omp_set_num_threads(num_threads);
    #pragma omp parallel for
    for(int i=0;i<untill;i++)
    {
        int work_for = level*(i+1)-1;
        int get = work_for - (level/2);
        input[work_for] += input[get];
    }
    level *=2;
}
```

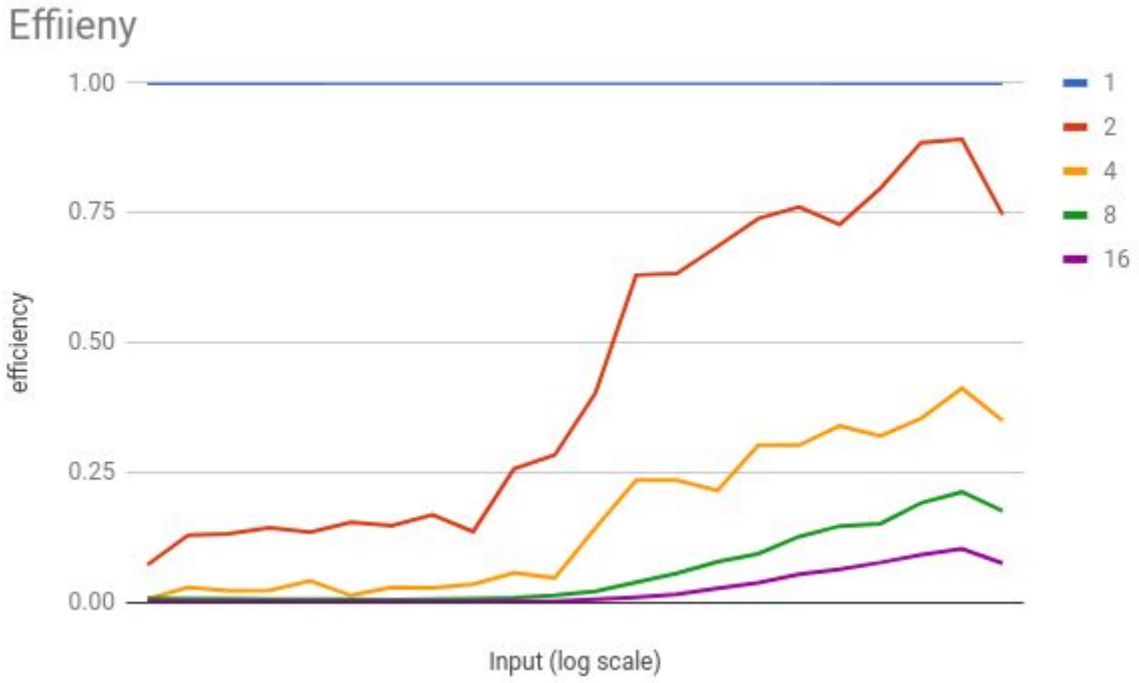
Plots

The implementation is run on my laptop with specifications as Intel® Core™ i3-5005U CPU @ 2.00GHz × 4 .
On this 4 core processor, the average of 20 runs with different values of n and p are plotted.
The speed up and efficiency curves are also plotted. The time taken with num_threads = 1 is used to calculate the serial time and hence speed up.




Time
0.25
0.20
0.15
0.10
0.05
0.00
time (in seconds)
Input (log scale)
1
2
4
8
16


Speed up
2.0
1.5
1.0
0.5
0.0
speed up
Input (log scale)
1
2
4
8
16



Analysis

Overall, the speed up obtained is low because of the overheads of the parallel implementation. For small input sizes, the speed up is smaller than even 1 ie the parallel implementation is slower than the serial one because addition is a cheap operation and the time spent in creation of the thread is larger. The time taken by 2 threads is smallest and efficiency is the highest since 4-core processor is used. The overheads increase and the parallel execution is limited by the number of cores and hence the efficiency of 16 threads is 0.25 only. Although, the increase in efficiency with serial workload is evident in the plots. The highest efficiency achieved is 0.8 and speed up being 1.75.