Second Assignment FHPC course

Exercise 0

Touch-First vs Touch-by-All policy

1. The following bash script was implemented in order to automatize data collection:

```
export TIMEFORMAT='%3S %3U'
export KMP_AFFINITY=granularity=fine,scatter
module load intel/18.4
for i in {1..50}
icc  program -o program.x -DOUTPUT -std=c99 -lrt
time ./program.x 1000000000
do
icc  program -o program.x -DOUTPUT -std=c99 -lrt -fopenmp
for procs in {2..20}
do
export OMP_NUM_THREADS=${procs}
time ./program.x 1000000000
done
done
```

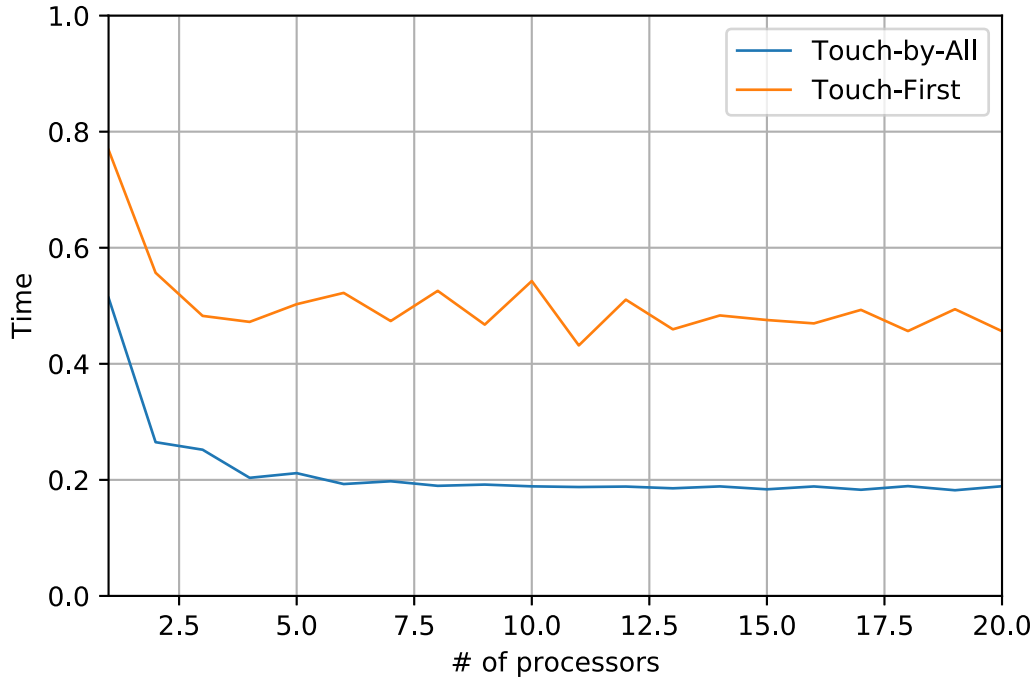
The first three line are respectively used to:

- Change the default output of command "time" in order to have more manageable data concerning time
- Fix one thread per each core
- Load the icc compiler module

The two programs were launched on the same computational node, with increasing number of threads, starting from just one (compiled in serial) up to 20. After a few runs, it became clear that the output times were heavily aleatory. In order to provide meaningful data, the final data used for the plot was obtained averaging the obtained results over 50 runs.

All data processing as well as all the following plots were obtained through a Python code (using Numpy and Matplotlib libraries).

The following plots shows the comparison between times taken inside the code of the touch-first and touch-by-all programs (not counting common initialization of variables):



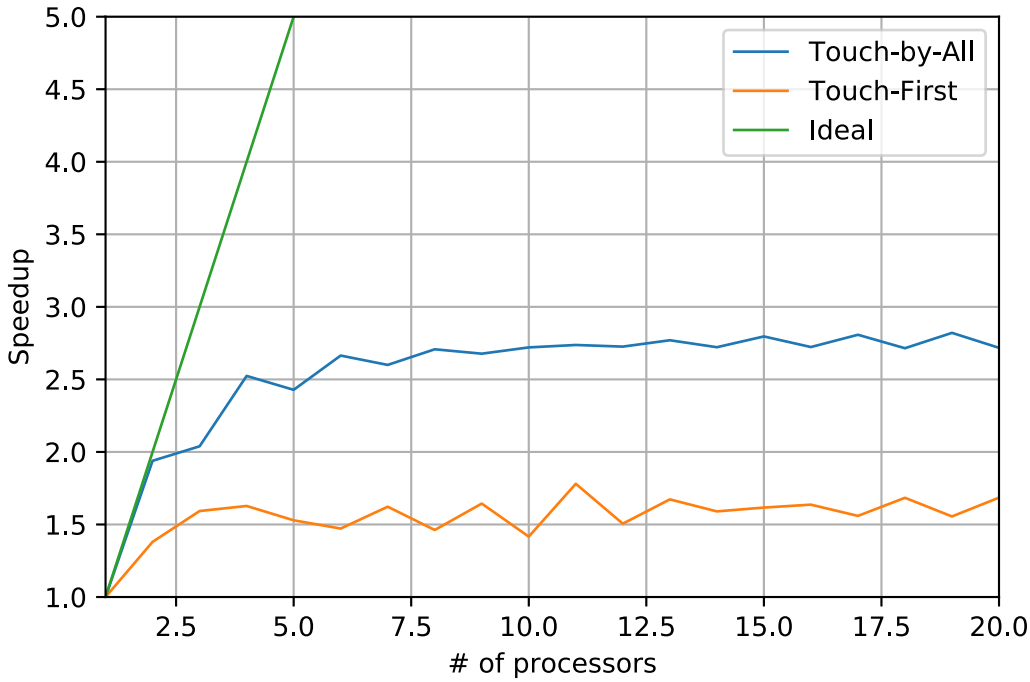
Unsurprisingly, the Touch-by-All algorithm has an altogether inferior runtime.

The speedup of a parallel program is defined as

$$S_p(P) = \frac{T_1}{T_P}$$

where T_1 denotes the execution time of the serial algorithm, while T_P the execution time of a parallel code solving the same problem on P processor.

Plotting a comparison of the speedup between the two code give as a result:



Which is ones again showing unsurprising better scalability in the Touch-by-all code. Even though the performance detected are heavily dependent on the computational node used, a sort of periodic behavior shown by the speedup functions is always present. It may be caused by the different time required for communication between sockets as soon as a new thread is assigned to a new core.

2. First of all, values of the serial fraction was estimated using the definition formula:

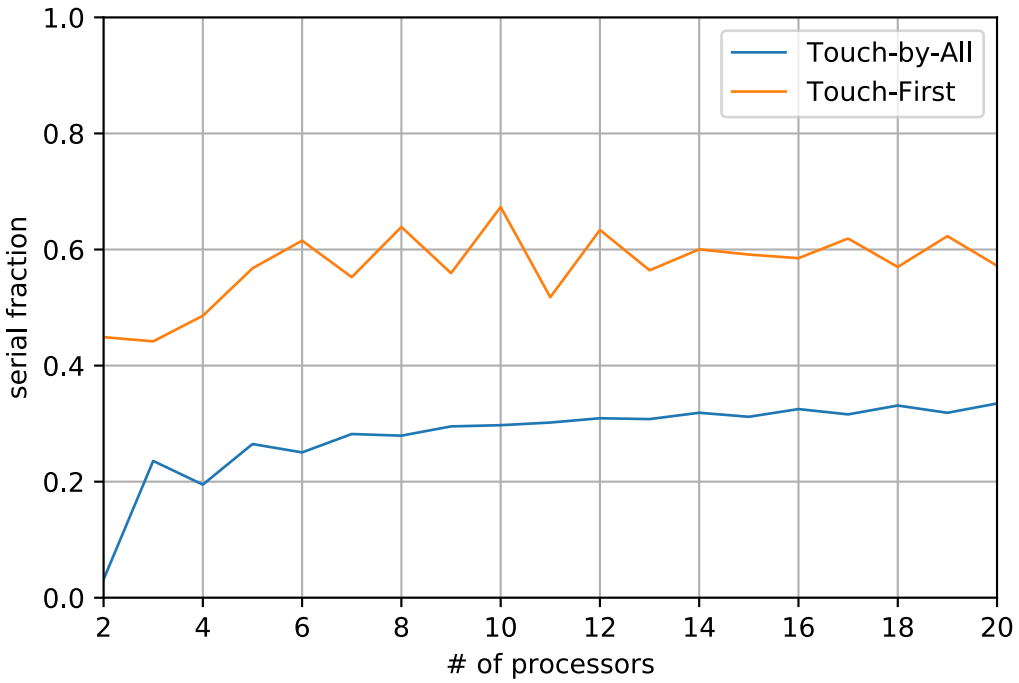
$$e(n,p) = \frac{\frac{1}{S_p(n,t)} - \frac{1}{p}}{1 - \frac{1}{p}}$$

Obtaining the following results:

p	2	4	8	10	20
Touch-first	0.449	0.486	0.639	0.673	0.623
Touch-by-all	0.031	0.264	0.279	0.297	0.334

For the Touch-by-all code the lack of scalability is probably due to the constancy of the serial fraction. At the opposite the increasing serial fraction of Touch-first probably imply, considering an even worst scalability, a bigger overhead.

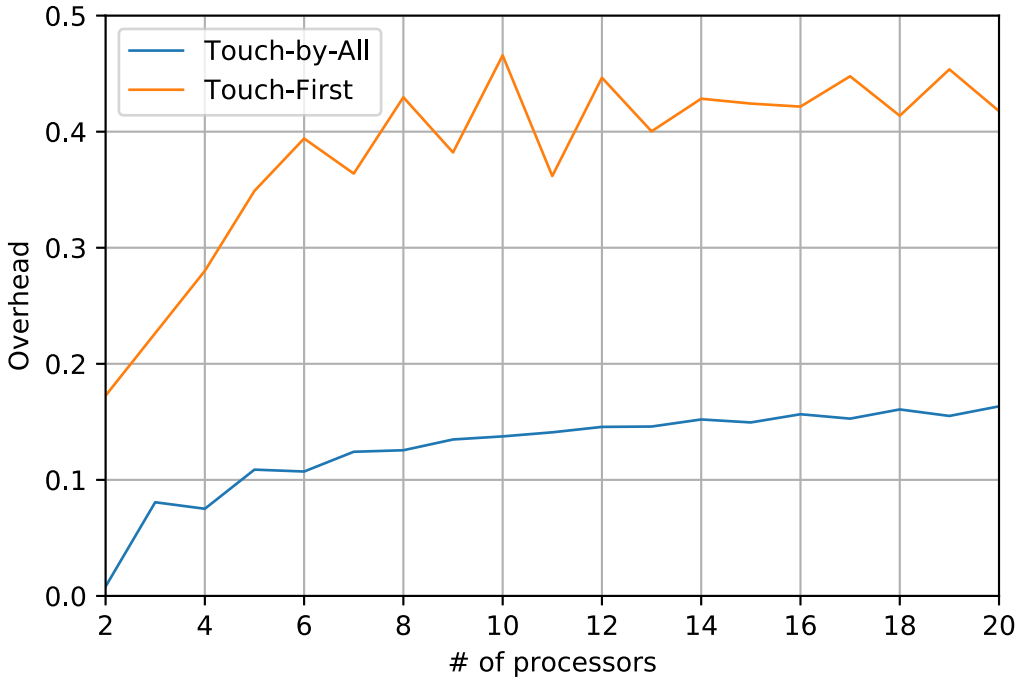
It follows the plot of the serial fraction for the two codes:



The overhead was estimated using the following formula:

$$O = T_p - \frac{T_s}{p}$$

Where T_p is the time taken by the parallel program, while T_s the time taken by the serial one.



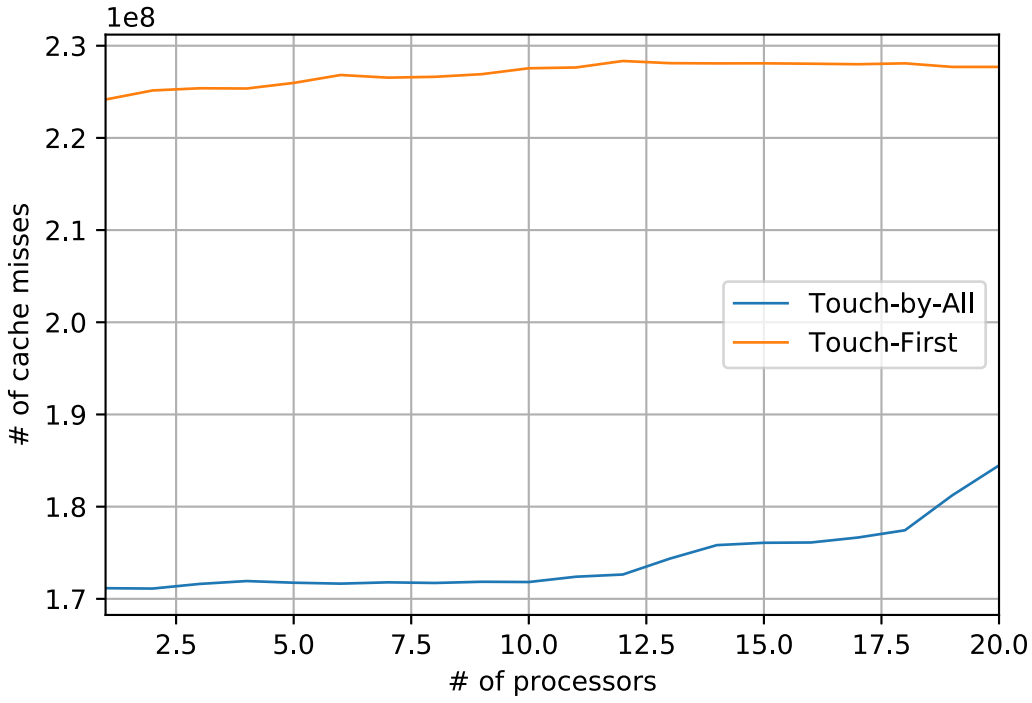
As expected Overhead is indeed far worst in the Touch-First program.

3. In order to better understand the already seen parameters, which show a clear distinctions in the runtime as well as in the overhead. Using command

```
perf stat -e cache-misses ./program.x 1000000000
```

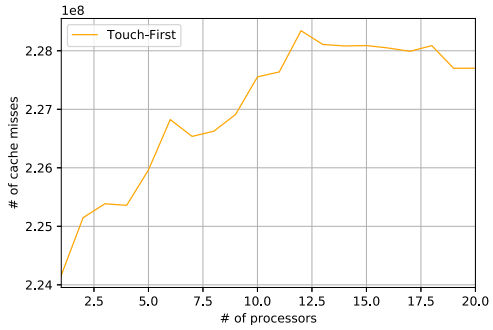
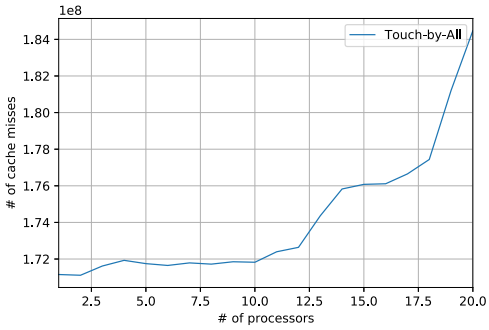
perf was used in order to retrieve data about cache misses in both programs.

Here is the plot concerning the comparison between cache misses for the two programs.



As shown, one of the main causes for the lack of performance in the Touch-by-First code, compared to Touch-by-All, could be the evidently superior number of cache misses. That is understandable considering that the initialization using #pragma omp gives directly to each thread his part of the array, placing it in the cache, while for the Touch-First every thread has to retrieve the data required for his calculation in the cache of the master, leading to more cache misses as well as more communications inside the node.

Looking one by one at the plots of the two cache misses:

Touch-First	Touch-by-All
	

It can be noticed that, for increasing number of threads, both code increase the number of cache misses. That can be pointed to as one of the main cause of the overhead, especially for the Touch-by-All code, for whom the serial part is actually increasing (so being less involve in the lack of scaling performance).

Optional

In order to allocate and correctly initialize the right amount of memory separately on each thread, the Touch-First program was taken as a basis. Using a similar strategy as the one implemented in the first segment of the prefix_sum.c program (look at exercise 3), every thread directly initialize its share of the array inside the parallel region.

Exercise 3

Prefix sum with OpenMP

The program developed to perform the prefix sum in parallel was written in C and called prefix_sum.c . It first of all initializes an array of doubles with value "1.0" of dimension N (given as input at runtime). That choice was taken in order to easily check if the code is giving the correct output, because in this way every entry of the final array will be his index plus one. Nether less, the algorithm make no use of the already known initialization, so the entries of the array could be chosen at will.

The serial version of the program was implemented with a simple straightforward:

```
a[ii]=b[ii]+a[ii-1];
```

Where a final output array is created using as entries the correspondent element from the first one and the one calculated from previous iteration. Doing the sum on two arrays, instead of doing it "in place", allow to perform memory aliasing.

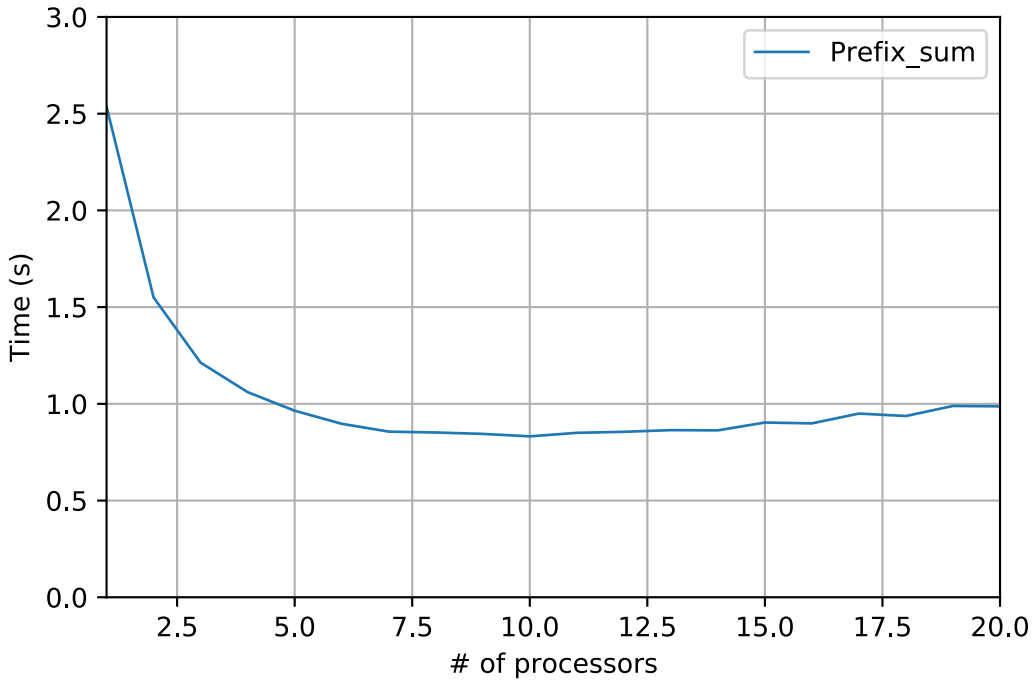
The parallel version instead is divided in two part:

1. First of all, the input array is divided in p parts (one per thread), and on every chunk the prefix sum is performed by each core as in the serial version of the algorithm.
2. Secondly, every chunk is sequentially corrected with the missing part of the sum (obtained as the last entry of the previous chunk) dividing the load among all the threads using a loop

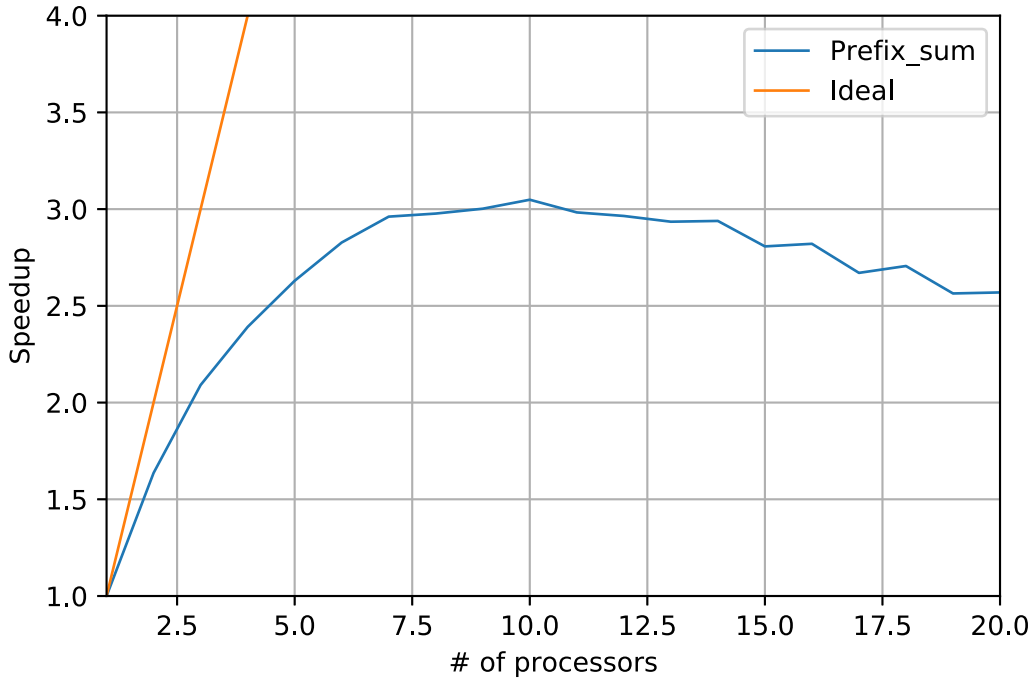
```
#pragma omp parallel for
```

Globally, the parallel version perform $N - \frac{N}{P}$ more operations than the serial on.

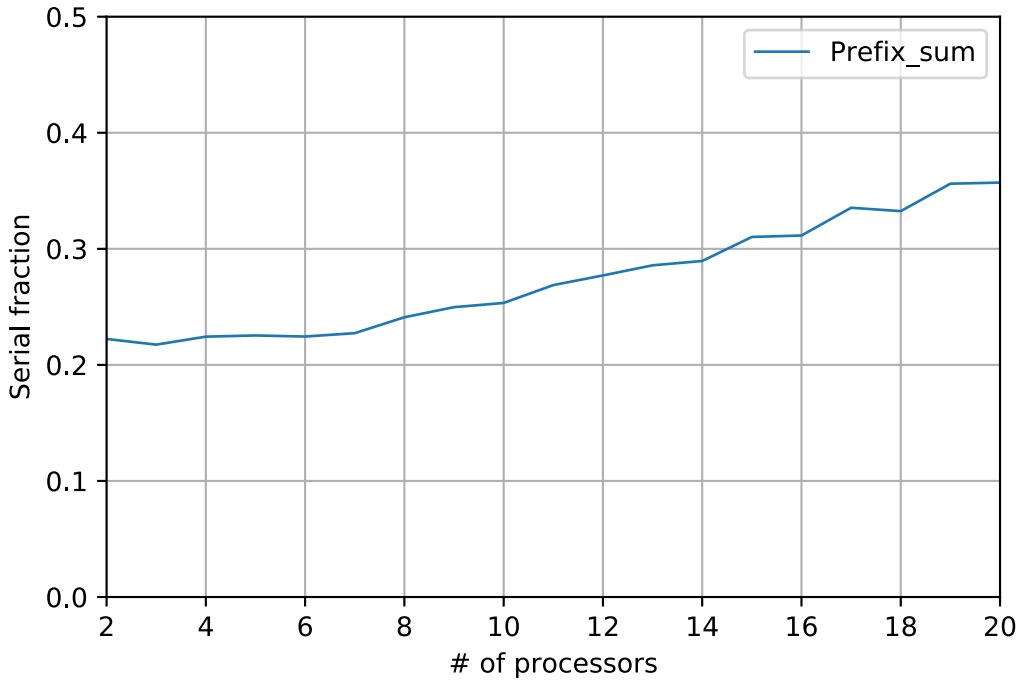
Times recorded per p,from 1 (serial) to 20, is plotted below:



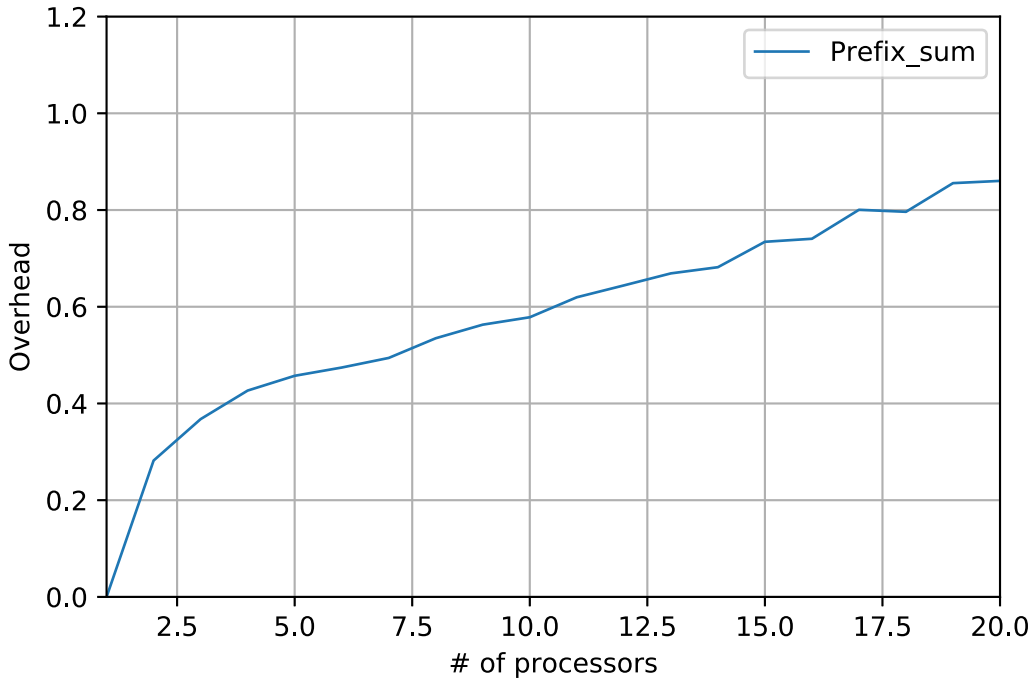
Speedup, calculated as in exercise 0, is:



From the last plot it is evident that the program doesn't scale well. In order to figure out what is causing such low performance, let's consider the serial fraction.



The serial fraction is increasing, which is reasonable, considering that the number of floating point operation performed by each thread is $\frac{N+(N-\frac{N}{P})}{P}$ which is exponentially decreasing. Given that, and considering also down-going trend of the speedup near the end of the chart, a fast growing overhead can be expected.



By fact its increase overwhelm the increase in the serial fraction.