UNIVERSITÀ DEGLI STUDI DI TRIESTE

DIPARTIMENTO DI MATEMATICA E GEOSCIENZE
CORSO DI LAUREA MAGISTRALE IN DATA SCIENCE AND SCIENTIFIC COMPUTING

High Performance Computing: Assignment 2

STUDENT:
Pietro Morichetti - SM3500414

Anno Accademico 2019-2020

Exercise 0 (mandatory): Touch-First vs Touch-by-All policy

Keep in mind that all the information showed in this section are based on two new version of the Touch-First program and Touch-by-All program, which have been cleaned up in the codes to clearly identify the main parts. These new codes have been inserted into the codes folder.

Question 0 - Measure the time-to-solution of the two codes in a strong-scaling test (use some meaningful value for N, like 10^9), using from 1 (using the serial version) to N_c cores on a node.

It should be remembered that the strong scaling test requires that the size of the problem is fixed, while the number of processes grows and the result graph should be a positive straight line, even if this is only an ideal behaviour. Indeed, the figure 1 shows you different strong scaling test for the Touch-First program, but all of them don't scale very well. This happens because, just after the third thread, the slope of the curve is close to the zero value and the growing is very slow, almost like a constant function.

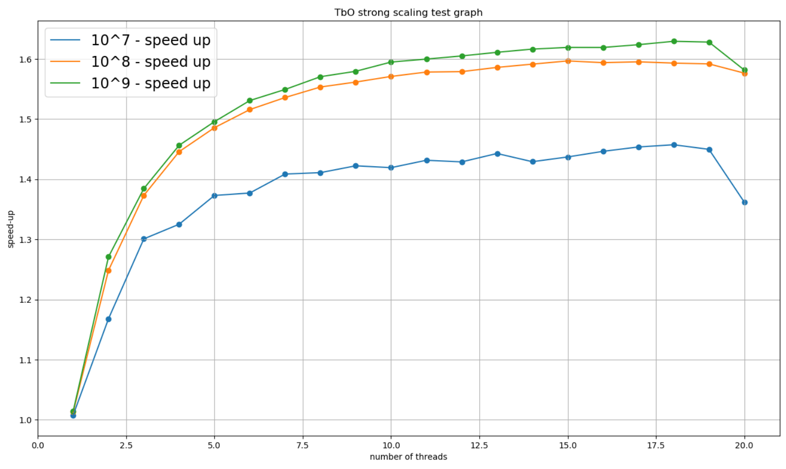

Figure 1: Strong scaling test for Touch by one program

On the other hand, figure 2 shows you the strong scaling test with the same fixed value for the problem size N. In this case, the regular trend of the different curves is closer to the ideal behaviour, in a more convincing way for value of N greater than 10^9.

In conclusion, this is proof that it is not advantageous to follow a Touch-first policy because if a single thread is initializing all the data, then all the

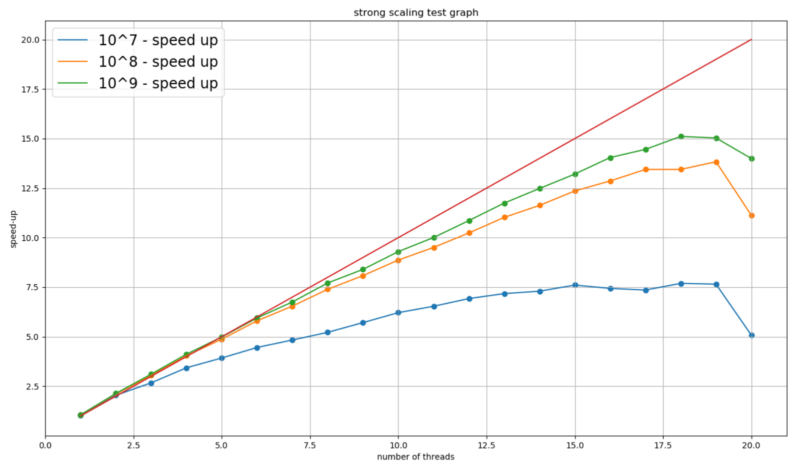

Figure 2: Strong scaling test for Touch by all program

data will reside in its memory and the number of remote accesses will be maximized. Indeed, the strong scale test proves that a significant gap exists between the two programs, because the first one is not efficient while the second one is with a greater order of magnitude.

It should be considered that the times have been taken by the following script.

```
typeset -i tmp=${1}
typeset -i procs=1

if [ -e ./file_${tmp}.txt ] ; then
 rm ./file_${tmp}.txt
fi

while (( procs<=20 ))
do
        perf stat -r 100 -e cycles ./a.out ${procs} 2> ./a.txt

        elapsedtime=$(cat ./a.txt | grep seconds |
                      tr --squeeze-repeats ' ' | cut -d" " -f2)
        percent_error=$(cat ./a.txt | grep seconds |
                      tr --squeeze-repeats ' ' | cut -d" " -f8 |
                      cut -d"%" -f1)

        row="${procs} ${elapsedtime} ${percent_error}"
        echo ${row} >> file_${tmp}.txt

        rm ./a.txt
 (( procs++ ))
```

```
done
```

Question 1 - Measure the parallel overhead of both codes, from 2 to N_c cores on a node.

There are several elements that contribute in the elapsed time, and in the case of parallel computing one of them is for sure the parallel overhead, so the TbO and TbA programs are included in this kind of case. Therefore, elapsed time could be considered as the linear combination of: the serial time of the program T_s, the "potentially" parallelizable time of the program T_p and the parallel overhead K; so, the elapsed time is explained by the following formula.

$$Elapsed\ Time = T_s + T_p + K \tag{1}$$

However, in this special case of TbO and TbA it is possible to define the specific regions of the programs in which the parallel overhead is present, i.e. the array initialization and the parallelized sum. So, it could be possible to overwrite the previous expression as:

$$Elapsed\ Time = T_s + T_{I_p} + T_{C_p} + K \tag{2}$$

where T_{I_p} is the time to initialize the array while T_{C_p} is the time to do the sums, each of them by the threads.

From the equation 2 results that the only unknown parameter is K inasmuch it could be possible to determine the others by using the chrono library. At the end, the parallel overhead is defined as shown below, but under the hypothesis that T_s is the serial time of the serial version of the program, which could be free from the overhead:

$$K = Elapsed\ Time - T_s - T_{I_p} - T_{C_p} \tag{3}$$

the following graph shows how the parallel overhead is changing due to different subset of threads and there is a growing for both TbO and TbA, but for the last one the overhead is more limited.

Note, the negative values in the first section of the graph are caused by measurement errors.

Question 2 - Provide any relevant metrics that explain any observed difference.

The scaling test and the survey of parallel overhead are two of the most important concepts to examine the behaviour of a program. Nonetheless, they provided only a superficial point of view, therefore if you want to understand more in detail, another kind of analysis is needed; at this point *perf* comes into play. It should be considered that perf allows a lot of kinds of analysis (or *events*), but on this paper it was preferred to focus on only few of them.

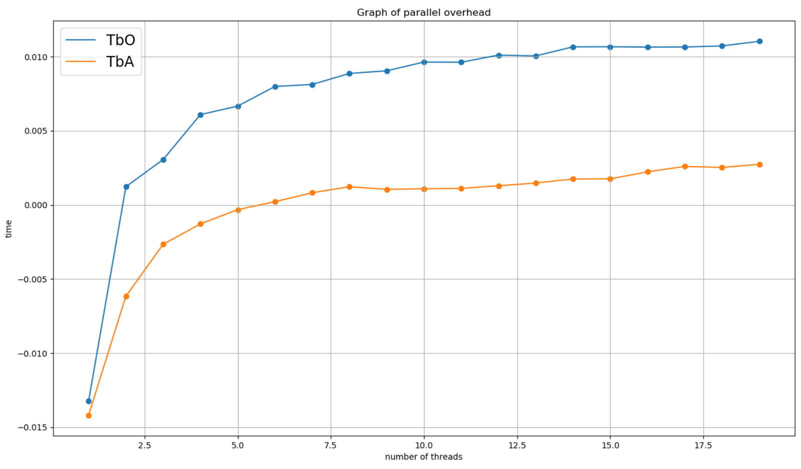

Figure 3: TbO and TbA parallel overhead

Note that all the following information is figured out through the exercise_00_dataset inserted into the dedicated folder; moreover this data has been obtained by the script_events.sh, which is available on the script folder.

cache misses and cache references The first focus is on the cache misses and cache references; in particular the first one rapresents the number of memory access that could not be served by any of the cache, while the second one explains the tendency to access the same set of memory locations for a particular period of time. As the figure 4 shows, the caches missed are more or less the same until five threads. After this point it records a linear growth for the Touch by one (or TbO) program, whereas the cache missed curve for the Touch by all (or TbA) remains constant. The same goes for the caches references.

All these behaviours are due to the cache policy, indeed in the TbO the thread 0 creates and fills the main array and then it pushes the main array into its own cache. When the thread 0 has to perform the sum operation it will hit the cache, whereas the other threads will miss the array. Therefore, the other threads will have to access the L3 cache level of the thread 0's socket.

Instead, in TbA each thread takes part of the main array initialization. It means that each thread will have a peace of main array into their own cache and when it will have to perform the summation operation, everyone will hit the cache. However, it does not mean there won't be any cache misses, because it depends on the size of the main problem and on the importance of the data.

Overall, the cache hit missed risk and references to the cache are much lower for the TbA than the TbO program.

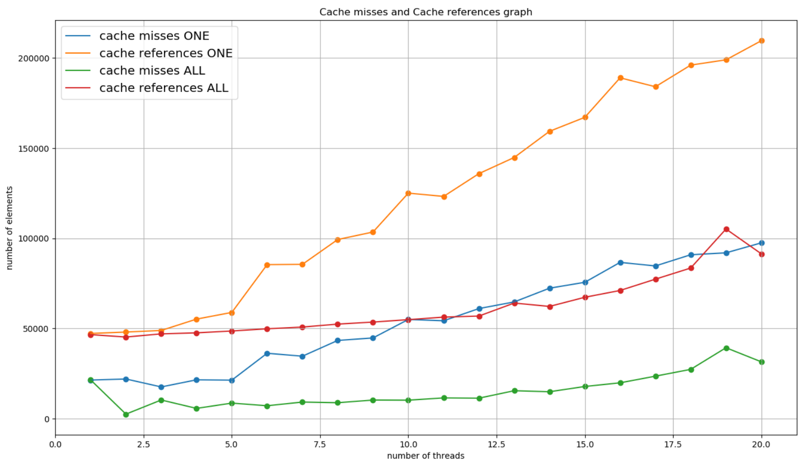

Figure 4: The cache misses and cache references for the Touch by one and Touch by all programs

LLC loads This event is specific to the Last Level Cache and shows the processor capacity to hit data loaded from the main memory to the LLC memory. In this particular case in which there is a hit missed of the data, the processor will have to take the data directly from the main memory and that will require a lot of time. The figure 5 shows that the LLC load curve for the TbO program is as much irregular as it is growing quickly, instead of the LLC curve for the TbA program that is more regular and nearly constant. These behaviours on the cache will propagate on the LLC load missed event, indeed for the TbO the figure shows how the hit missed curve tends to increase togheter the LLC load curve. Back to front, the hit missed curve for the TbA is: regular, constant and close to zero.

The reasons for the trend of these curves are mostly searched in the previous explanation about the TbO and TbA policy.

Overall, it can be observed that the LLC miss ratio for the TbO is equal to $0,4621$ (std $= 0,0846$), whereas the LLC miss ratio for the TbA is equal to $0,1064$ (std $= 0,0863$); more or less the same value.

LLC store This event is specific to the Last Level Cache and shows the processor capacity writes back the newly calculated data to the main memory, therefore the capacity to update data already into the cache. However,

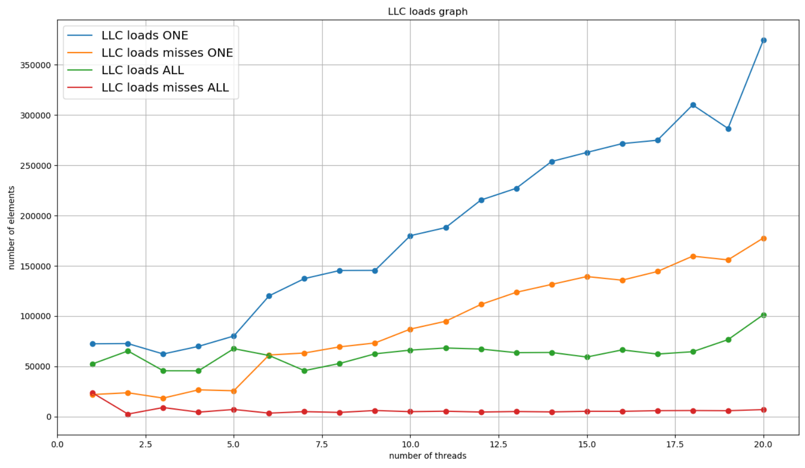

Figure 5: The LLC loading behaviours for the Touch by one and Touch by all programs

before all the updates, the processor will have to be sure that the LLC memory and the main memory are consistent, i.e. the data stores into the LLC memory and the main memory must be the same. Then, in the particular case in which this match is not found, the processor will update directly the main memory instead of the cache. The figure 6 shows how the LLc store curve for TbA is a kind of constant line for each of the subset of threads; therefore the LLc store TbA curve is more irregular, in detail it figures out that the amount of store instruction is different for different number of threads.

As regards the LLC store missed, the figure shows a particular behaviour for what concerns the TbO. Indeed, it results like a kind of converging "sinusoidal" function that could be caused by the Place and Binding Policy. On the other hand, the LLC store missed curve for TbA is a kind of converging "logarithmic" function, which has been more or less the same behaviour of the TbO curve using at least nine threads.

Overall, this graph is very interesting because is quite different from the other plots.

IPC and Gb/sec The IPC is very important to evaluate any program and for this reason it is done a kind of IPC analysis on specific portion of the codes, in particular on the initialization and calculus region. These analysis have been done by using the script_timer.sh that figure out few important parameters: cpu-cycle, instructions and the transfer data rate. In particular, for the last one it is needed to put two couple of timer into the codes in order

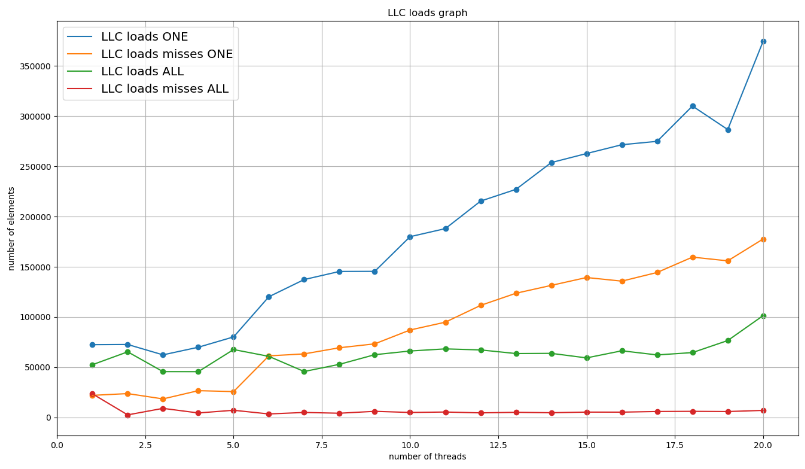

Figure 6: The LLC store events for Touch by one and Touch by all

to take the proccessor time (time spent only to complete the program and not other times); the "timer" version of the codes arrive for this region.

The figure 7 shows some interesting behaviours, i.e. that in all parallel portion of the codes, the trends of the curve are descending and for some of them it is very quickly. There could be several reasons for these features as for istance that if the number of threads increased, also the cpu-cycle increased (see the data_set). Indeed, this information about the cpu-cycle shows that the serial version of the portion is constant, while the parallel version starts close to it and then grows; it seems that the number of cpu-cycle could represent a different view of the parallel overhead and same behaviours are be detected for the transfer data rate.

other useful events In the present paragraph a series of events which have been detected during the analysis phase will be shown. In particular, these events result almost equal for both the programs as shown in the specific dataset.

Question 3 (optional) - figure out how you could allocate and correctly initialise the right amount of memory separately on each thread.

A lot of attempts have been made to answer this question, as for example by using prefetching tecnique, or pipeling or again implicite declaration to the compiler but they were not so satisfying. Therefore, The only successful attempts on the cleaned_t_b_all.c are shown below.

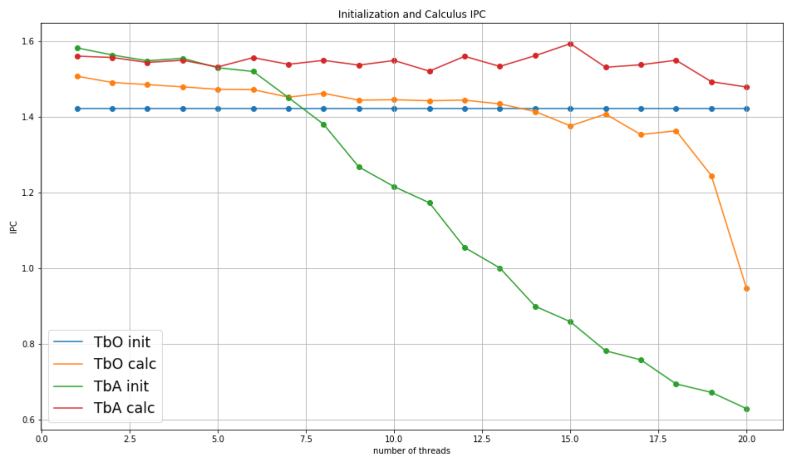

Figure 7: IPC analysis on the initialization and calculus regions of the TbO and TbA programs

option 0: allocation and initialization by manual settings The default behaviour of the Touch by All program is to entrust itself on the pragma library, i.e. the #pragma omp parallel for that will provide both allocation and initialization automatically. Therefore, the idea is to replace these features with some manual settings and it is done exactly as the Assignement 1, in the last exercise; a brief description will follow.

The dimension of the problem N is divided by the number of all threads that are involved in the code in order to determine dim_sub_array, size of one sub-problem. This subset of the main array has to collect a specific sequence of numbers which the particular thread have to sum up like a partial sum of the total problem. However, there is the possibility that some threads have to consider a portion of the array with dimension dim_sub_array + 1. These differences occur because the size of the main problem might not be a multiple of threads number; this means there could exist some remainder R. So, threads with rank minor than rest will have to manage it; more specifically, everyone will use its own rank to understand from what value to which value has to fill up the main array and then works on it.

In conclusion,you got the cleaned_t_b_all_option_1.cc program.

option 1: all in the same loop Another idea is to do the initialization and the sum in the same #pragma for loop, as cleaned_t_b_all_option_2 shows. This method is for sure one of the most efficient solutions and you can expect excellent results in terms of performance, as for instance about cache miss or LLC store features.

Exercise 3: Prefix sum with OpenMP

Question 0 - Develops an efficient version of a prefix sum with the + operator, and also to OpenMP-ize it.

This section shows a brief explanation of the Prefix Sum program that you will find in the codes folder; in particular, the task is completed in twice serial and parallel implementation into the same code by using the stantements #ifdef and #ifndef, but in any case both of them share a small part of the program i.e. the definition of the size N (size of the problem) and the monodimensional matrix *Array*.

The first step of the serial version requires that the Prefix_sum variable is equal to the first element in Array. Then, the algorithm proceeds adding up the actual position in Array with the actual Prefix_sum value and updates the actual Array position. Although, this algorithm is the most efficient because it doesn't require calculation from the beginning of the array but it is limited by serial logic.

On the other hand, thanks to the OpenMP library it is possible to use some parallel computing logic through several instance of the code i.e. the threads. It is precisely by using this tool that the implementation of the Prefix sum parallel algorithm is allowed. In the following section will be showed all the procedure in few steps.

step 0: setting shared variables The algorithm starts by defining some new parameters, as for example the number of threads which is defined directly in the code in case it doesn't receive the value as the argument of the program. Another parameter is a new array called *Partial_prefix_ sum* and it is inizialized by the calloc function. In particular, it will be used to collect all the final results for all the prefix sums performed by each thread.

step 1: setting private variables Until now all the parameters have been defined outside the parallel region which means they are shared among all the threads, while the variable defined inside this region are local as the ID of the thread. Other two important parameters are the *local_partial_prefix* and *local_ offset*, which are used to perform the prefix sum algorithm. The last operation of this section is the inizialization of the Array by all the threads, which is the same for all the threads and everyone has its own copy.

step 1: serial parallel prefix sum on the main array This section requires that all the threads play in the first paralell for loop that will result in a trasformation of the Array. In particular, each thread has to consider not all the array, but a "small" portion of it and to which it will apply the serial prefix sum algorithm.

This step is finished when all the threads have put the final element of the own prefix sum inside the Partial_prefix_sum array; in particular, they put it in the cell correspoding to their own ID increased by one, as figure 8 shows.

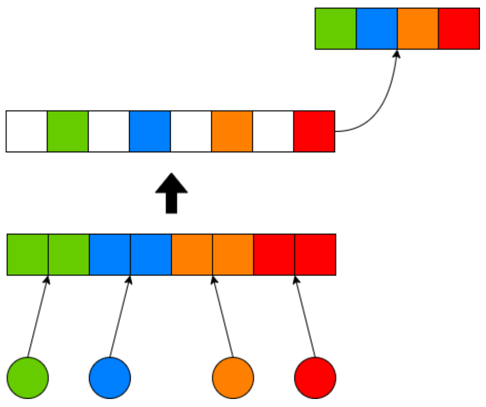

Figure 8: Prefix sum, first part of the procedure

step 2: serial prefix sum on the support vector This section is composed by a single serial for loop which in all the threads figure out the specific offset of their own portion of the main problem.

In this case it might be not so efficient if everybody defines their own offset instead of doing it in parallel way, but the dimension of the Partial_prefix_sum array is more or less equal to the number of threads, which in that case is very small. In conclusion it is meaningful to consider not the parallel procedure but the serial procedure to perform this specific operation.

step 3: serial parallel updates of the main array In this last procedure the threads play in the parallel for loop to perform the update for its own portion of the Array, adding up all the elements with its own local_offset parameter. At the end of the game you will obtain the prefix sum of the Array, as figure 9 shows.

Question 1 - Compares the serial and the parallel version, reporting at least about the time-to-solution and the parallel overhead.

The serial version of the Prefix Sum is one of the most efficient and it could be interesting try to compare this serial version and its parallel transformation; so, the first comparison has been done by the time-to-solution analysis shown by the figure 10.

As it is visible, the parallel elapsed time is under the serial elapsed time everytime, also for different sizes of the problem; rather, the gap serial and parallel is much larger as the size increases.

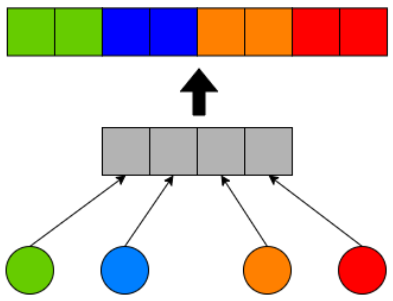

Figure 9: Prefix sum, second part of the procedure

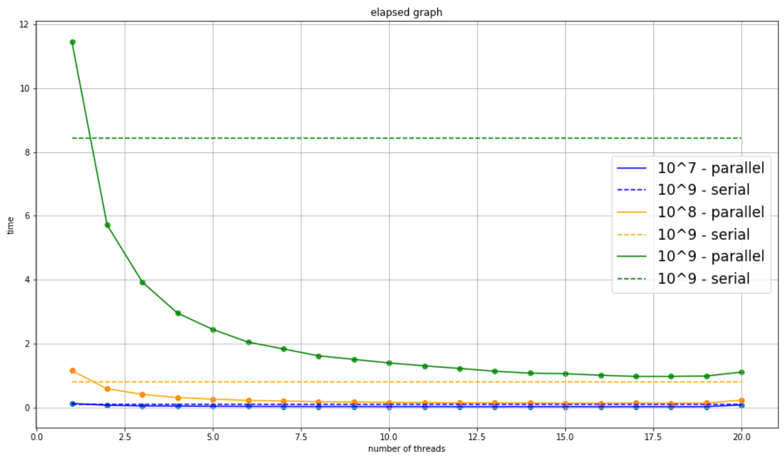

Figure 10: Time to solution analysis between serial version and parallel version

The second analysis is about the parallel overhead and the same considerations of the exercise 0 about the parallel could be done. But, in this particular case you don't split the T_p and it is all the time of the parallel region. Therefore, the overhead's formula remains:

$$K = Elapsed\ Time - T_s - T_p \tag{4}$$

where T_s and T_p have been taken by using the chrono library.

The following graph 11 is derived from the formula 4.

The figure shows that the trend's overhead is very irregular, but this behaviour could be associated to measurement errors and a significant standard error. Therefore, it could be considered the general trend that is a progressively increasing trend

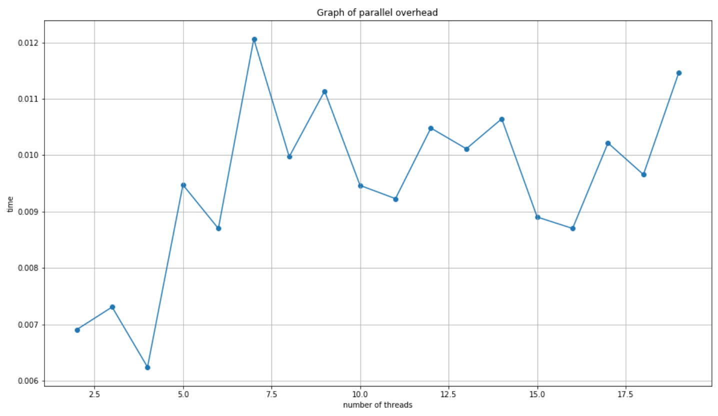

Figure 11: Prefix sum parallel overhead

IPC and Gb/sec Others useful information might be sow through perf analysis and some data has been stored in the dedicated folder; here it will be showed only the IPC analysis because it is one of the most important parameters.

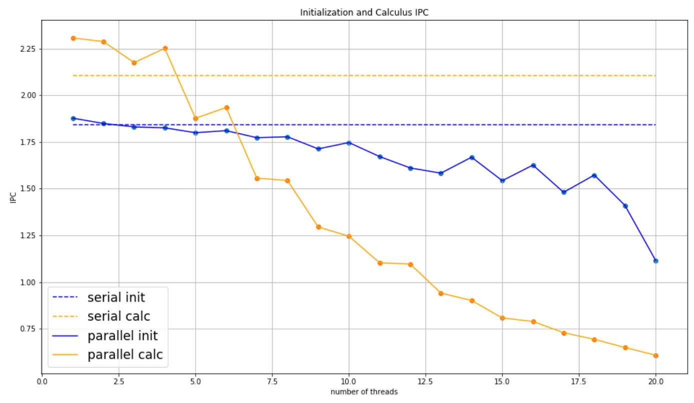

Figure 12: IPC analysis on the initialization and calculus regions of the TbO and TbA programs

Therefore, the figure 12 shows exatly the same behaviour of the previous IPC analysis, i.e. that in all parallel portion of the codes, the trends are descending and for the parallel calculus is very quickly. Also from the point of view of the transfer data rate you can see a significant reductions.