Report Assignment 02

Francesco Brand

December 17, 2019

Contents

1 Exercise 0

1.1 Strong Scaling

In order to test the strong scaling of our application we've considered the codes 01_array_sum.c as found in the assignment directory and 06_touch_by_all.c provided in directory L11. The files have been slightly modified in order to print the results in a cleaner way, so that the script responsible for saving the execution time into a csv file is less cumbersome than it would have been. Both programs have been tested on Ulysses cluster, after having being compiled with gcc. Further details on the compilation and on the run on the code can be found in the readme provided.

Both programs have been run 10 times in order to have more accuracy on their execution time and to provide some meaningful statistics regarding it. The times reported are all taken with the command /usr/bin/time of linux machines and they correspond to the elapsed time of the program.

By analyzing the implementation underling the two codes, we can observe that the only difference resides in the inizialization of the vector. Whereas in the 01_array_sum.c the array is initialized in the serial part of our code, in the 06_touch_by_all.c, the inizialization is performed by multiple threads by means of openmp library.

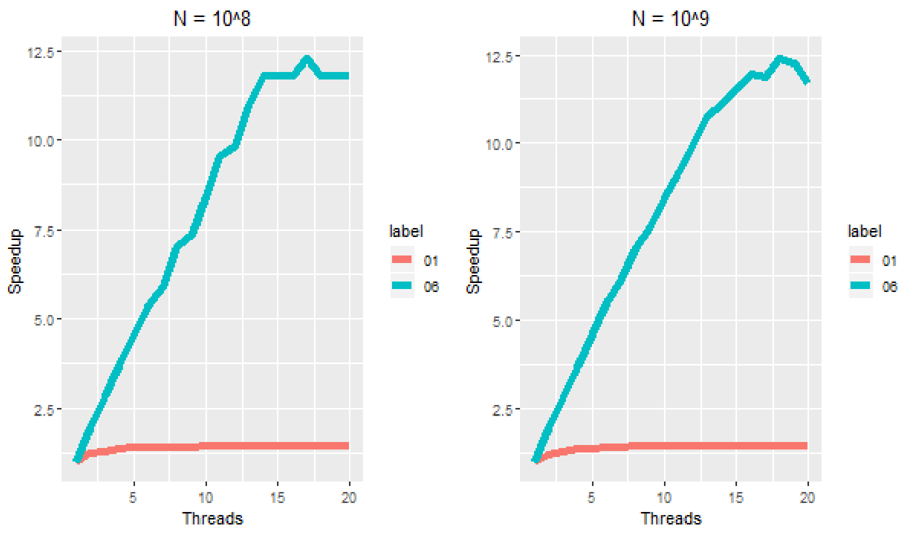

Figure 1: Speedup for touch-first (red) and touch-by-all (blue)

By this fact alone we can expect the execution time of 06_touch_by_all to be less than the one of 01_array_sum. Furthermore, since we are working on a multicore CPU in which the level 1 caches are private to each core, and every thread is assigned to a different core, by initializing the array with the touch by all policy, we are loading different parts of the array in all our caches, so that the threads don't have to load their respective parts from the RAM (which is a costly operation) but they have already them available to reuse for the reduction part of the code in their caches.

Nevertheless, it is hard to predict how much more effective this implementation is compared to the serial one. Since we are parallelising every aspect of our algorithm, i.e. both the inizialization and the reduction of our array, a naive expectation would be to observe $Speedup = p$ where p is the number of threads spawned. However, it will probably be really difficult to reach that results for two main reasons:

1. Ahmdal law guarantees that, even for a very large number of processors, the speedup will be bound by the unavoidable serial fraction of our code.
2. Even so, there are a number of sources of overheads taking place, such as the overhead associated to memory allocation functions and the overhead associated to the creation of the parallel regions.

As expected, Figure 1 shows that for both values of N considered, the 06_touch_by_all heavily outperforms the 01_array_sum for every thread considered. However, in absolute terms the speedup of our parallel codes is definitely distant from the perfect speedup: it rises up to just 1.5 for the 01_array_sum and it barely touches 12 even for the 06_touch_by_all. This might be caused to heavy overheads associated with our OpenMp calls and OpenMp synchronization. On top of this, the problem we are trying to solve, i.e. summing over the elements of an array, generally called reduction is known to be hard to parallelize. In particular, in both our codes we are experiencing the so called false sharing [1]: since S, the variable storing the array sum, is a shared variable between the threads, and it needs to be overwritten at every iteration of our loop, it needs to be refreshed in all the caches every time. This introduces a huge amount of overhead due to the necessary synchronization in between the threads.

1.2 Parallel Overhead

We will here try to estimate the parallel overhead utilizing the Karp-Flatt metric [2]:

$$e := \frac{t_s + k}{t_s + t_p} \tag{1}$$

where k is the parallel overhead, t_s is the execution time of the serial part of our algorithm and t_p is the execution time of our the part of the algorithm which we want to parallelize. This can be proved to be equal to

$$e = \frac{1/Sp - 1/p}{1 - 1/p} \tag{2}$$

By its definition, we can expect e to have the same behaviour of k, with respect to the variation of the number of threads. This is due to the hypothesis that the serial part of our program does not change in between the non-openmp program with respect to its parallel counterpart. In particular, if e is increasing we can affirm that k is increasing too.

As we can see from the Figure 2, both programs tend to have a stable overhead, especially for the highest value of N considered. Of course the difference in absolute value in between the two curves is really high (since the formula basically depends on the inverse of the speedup), and since both algorithms are derived from the same serial algorithm (which is what gives us t_s and t_p) we can conclude that the parallel overhead associated with 06_touch_by_all is much lower with respect to its touch-first counterpart. Again, I want to stress that this conclusion is possible if and only if we are certain to share the same serial algorithm. Furthermore, we can be confident that the overhead associated to OpenMP parallelism does not increase with respect to the number of threads spawned, as we can see from the average behaviour of the two curves in the plot.

1.3 Other metrics

We've already seen some motivations for which 06_touch_by_all outperforms in terms of execution time 01_array_sum. All of them derive from a direct analysis of the source code. However, there are ways to spot this difference even without

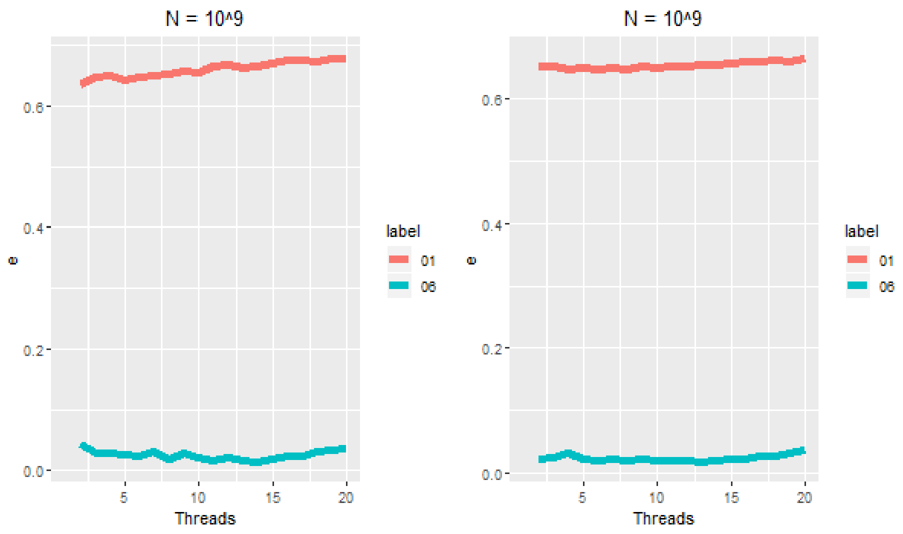

Figure 2: e metric for touch-first (Red) and touch-by-all (blue) programs

taking a direct look at the code. For example, we can use the performance counter of our CPUs to take a deeper look at cache misses, branch misses, instructions per cycle during the execution of our program.

There are a number of tools which can be used to perform this task: we chose to use Perf since it doesn't need to be integrated in the code (unlike PAPI, for example) and it does not create further overhead due to its usage. Its record feature can also be exploited to deeply understand some bottlenecks within our codes.

We've analyzed the performance of our codes using the command perf stat -d on our program, running it on the login node of ulysses with an input dimension of 10^9 and repeating the process 5 times (I know that the login node should not be used for computation, but on the computational nodes I could not obtain any measure for the L1 cache), which yields us both hardware counters and cache counters. In particular we will focus on the following metrics:

- instructions per cycle
- percentage of L1 cache misses
- percentage of Last-Level misses

As we can see from table 1, the only strong difference we have in between the two programs resides in the number of instructions per cycle which is higher in the touch-by-all program. This was to be expected: by parallelizing the initialization of our array we are actually having a better usage of the cpus of the cores in which the master thread is not running.

Program	IPC	L1 cache-miss-rate	LL cache-miss-rate
touch-by-all	1.02 +- 2%	0.05 +- 0.2%	0.62 +- 0.6%
touch-first	0.89 +- 0.5%	0.05 +- 0.1%	0.65 +- 0.4%

Table 1: Performance counters

A question that might arise naturally is why we don't experience a lower number of cache misses in our more performant code. Theoretically speaking, in fact, we should be lowering our remote cache access by initializing different part of the array in different threads, thus resulting in less cache misses once we recall the array in order to sum over its components. In practice though, data are showing us that there is no meaningful difference in between the two codes as regards the ratio of cache misses on the total cache access. Perhaps one reason is that the dimension of our problem far exceeds the number of the cores we have available thus causing the cache to be completely refreshed in between the array initialization and its reduction.

1.4 OpenMp and NUMA architecture

The 06_touch_by_all can be, in theory, furtherly tuned and optimized for the peculiar architecture of Ulysses' computational nodes. Each node is built with two sockets, each of these sockets containing 10 cores. Furthermore, every socket is directly connected to a RAM, making this a non-uniform memory access (NUMA) architecture which means that the two sockets can access both RAMs but with different times.

Our goal in this section is then to find a method to exploit this architecture by allocating our array in both RAMs and by assigning the threads to the cores close to the chunck of memory they are working on.

The method we have thought about works with these steps:

- We use OMP_PLACES=cores and OMP_PROC_BIND=close in order to have thread 1 on core 1, thread 2 on core 2, thread k on core k.
- We declare two arrays and let thread 0 and thread 10 (which should be placed in two different sockets) allocate memory for them. Note that here we are supposing that malloc function is by default allocating memory to the closer RAM available.
- We define two shared variables for the sum of the components and we execute the same reduction as we did before by controlling which thread access to which array in order to preserve data locality
- We sum up the two reductions obtained.

Unfortunately, our implementation did not achieve a good performance, as can seen in figure 3. I think there are two possible reasons for this result. First, it can be caused by a wrong ripartition of the threads, which might

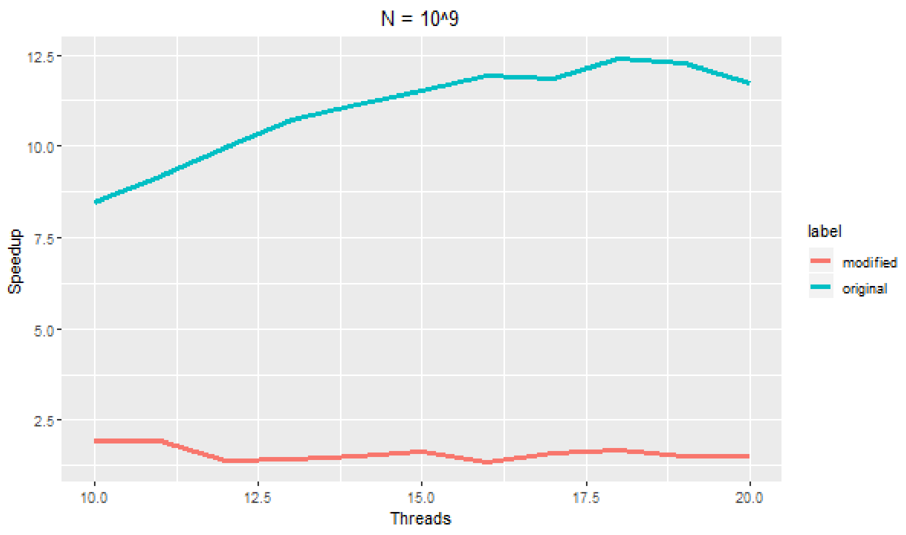

Figure 3: Speedup 10-20 threads for NUMA aware program

have even worsened data access. This is possible but it is difficult to obtain more information regarding this without the availability of counters related to remote data access, such as those provided by perf c2c utility. Another option is that, by rewriting part of the code, new sources of overhead have actually been introduced, especially due to synchronization in between the threads. Since we have split the work in between two groups of threads we actually need to wait for both of them to finish before making the last sum. This means that we are waiting for the slowest group of thread to finish its reduction on its array and this might represent e a huge bottleneck in our program.

2 Exercise 3

We are now going to tackle the PrefixSum problem in a parallel fashion, using OpenMp library to implement an algorithm that can solve the problem and possibly allow us to gain some efficiency with respect to its serial implementation. There are many example in literature of algorithms to efficiently parallelize the Prefix Sum, such as the one proposed by Guy Blelloch [3], which is known as one of the most work-efficient algorithm tackling the problem. However, since the cores that we have available are really few compared to the dimension of the problem we are tackling, we will focus on a more naive version of the algorithm.

2.1 Algorithm

We chose to compute the prefix sum following a SPMD (Single Program Multiple Data) paradigm. In particular we have implemented these steps:

1. First, we split the array into chunks (we actually use the OpenMP API to perform this task), and for every chunck we calculate the prefix sum while updating our array.

2. We then form an array of lenght equal to our number of threads with every i_{th} component corresponding to the prefix sum of the chunck assigned to the i_{th} thread.

3. We calculate the prefix sum of this array using a serial approach. Note that we could have done this in a parallel way, using an up-sweep, down-sweep approach, and obtain a theoretically better algorithm ($O(log(p))$ instead of $O(p)$ for this part). However, our algorithm has complexity $O(n/p + f(p))$, where $f(p)$ is either $log(p)$ or p based on the approaches just described. Since on Ulisse we can just exploit less than 20 threads, the difference in between the up-sweep down-speep and our approach is not significant, since n/p is anyway much bigger than the term $f(p)$.

4. We add to our chunks the offset stored in the corresponding position of the array of prefix sums just calculated in the previous point.

2.2 Three implementations and results

For the algorithm previously described we have implemented three different versions in order to try different techniques to improve the scalability of our code, as well as to see the impact of peculiar implementations to the performance. We have started with a naive version of our code, in which we keep a touch-by-first initialization policy, process the prefix sum on our original array and we don't try to catch some heavy cache-miss spots on our code. The second version features a touch-by-all inizialization, some better organization of the code with respect to loops and a slightly better usage of the pipelines of our CPUs. Lastly, in our third version we've tried to use an auxiliary array to perform our computation and, as we will see, this has drastically improved our performance, at least up until a certain amount of threads spawned.

From the plot in Figure 4 we can see how every implementation improves the performance with respect to the most naive implementation. In particular, we can see that the naive version was really inefficient, capping its speedup at less than 1.5, while the first optimization already doubles the result.

Eventually there is clear evidence from the plot that our code does not strongly scale, since we reach our peak performance after 8 threads for the optimized version and at 16 threads with the non-local one.

This might suggest us that our code relies for the most part on a serial fraction which caps our speedup quickly in our first two implementations, as theoretically explained by Ahmdal's law. Since the execution time is constant after a certain p_0 we can also conclude that the inherent parallel overhead associated with OpenMp is itself constant with respect to p. This, however, is not

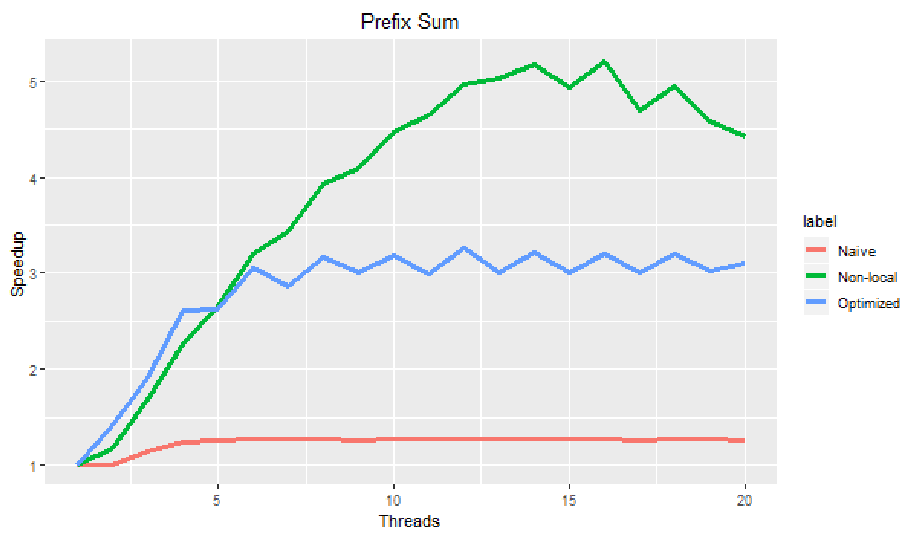

Figure 4: Speedup for 3 implementation of Prefix Sum

the case for the third implementation meaning that introducing a second array also introduces some overhead which scales with p.

References

[1] Eijkhout, Victor(2016): Introduction to High Performance Scientific Computing (pag 106)

[2] Quinn, Michael Jay(2003): Parallel programming in C with MPI and OpenMP (pag 170-171)

[3] Blelloch, Guy E. (2018): Prefix sums and their applications. figshare. Journal contribution. https://doi.org/10.1184/R1/6608579.v1