Name: Floriane Gabriel Sciper: 248112

Julien von Felten 234865

We started by parallelize the simulate method. To do so, we simply added a "omp_set_num_ threads (threads)" and "#pragma omp parallel for" after the n for-loop. Since the nth iteration depends on the n-1 th iteration, we cannot parallelize this for loop. But we can parallelize the i and j loop. This did not make a speed up for one thread, but we have a great first speed up with 2 and more threads.

"We decided to try to invert the 2 for-loop (i and j). We started with the j for loop, then i for loop. It was a really bad idea. We saw that on our machine, even with one thread, without any parallelization, it was considerably slower (it took twice as much time). So, we decided not to try it on SCITAS since it was trivial for us that this solution would never be better because of the data locality.

We looked at the code and we thought that the if clause was at a bad spot. Since it is just in 4 cases and we don't want to check this condition at every loop, we tried to remove this if condition. We just added the correct value at the end of the 2 for loops. This gave us a considerable speed up since the if condition was not checked anymore at every loop.

Since we saw tiling in class, we decided to try this method to optimize our memory usage. This optimization should not optimize the parallelization of our program, but the memory usage. If it is better, then we can see a speed up. We did this optimization in multiple step:

For all optimization, we have this 4 for-loops in the following way:

```
for(int k = 1; k < length; k+= block_size){

for(int l = 1; l < length; l += block_size){

for(int i=k; i<length-1 && i < k + block_size; i++)

for(int j=l; j<length-1 && l + block_size; j++)
```

1) Naïve way of tiling: we defined some block_size at 256. We removed the if-condition inside our for-loop since we know that it cannot be better. We saw that we didn't have any speed up and it was even worse!

2) We decided to change the block_size from 256 to the computation: $block_{size} = \frac{length}{threads}$. We removed it and change the block_size. This gave us finally a great speedup which was a bit better than our "simple" version.

Using the tiling method gives us a speed up since it helps with the data locality.

In this table, we wrote first the time need for the job (the task was as asked on the assignment which is 100 iterations, length 10000, and we wrote in second the speedup in comparison with the no parallelization with 1 thread.

Number of threads	No parallelization	Parallel for	Parallel for without if	Naïve Tiles (block size =256)	Optimize Tiles (block size = $\frac{length}{threads}$)
1	41.09	41.93 - 1.0	17.82 – 1.0	323.8 - 1.0	15.77 - 1.0
2	40.9	21.25 – 1.97	9.444 – 1.88	163.8 - 1.97	12.58 - 1.25
4	40.79	11.46 – 3.66	5.681 – 3.13	95.53 - 3.38	11.72 - 1.34
8	41.29	6.194 – 6.77	4.185 – 4.26	62.4 - 5.189	14.04 - 1.12
16	40.74	3.973 – 10.55	3.588 – 4.96	58.06 - 5.57	24.43 - 0.64

We evidently don't notice any speed-up in the "No parallelization" implementation. The "parallel for" implementation offers a good speed up but the original time of nearly 42 seconds s still too high. The time gets much better as soon as we take the if condition out of the loop and apply it at the end, seeing as it only apply to four squares. The first Tile implementation with a block size of 256 offers terrible results although the speed-up is good which led us to adjust the block size to $\frac{length}{threads}$, which allows a better work distribution among the threads. However the more threads we use, the smaller the blocks become which implies that more blocks will have to be fetched and each is fetched individually without prefetching. In the "parallel for no if" implementation the next line in the program gets prefetched when the program gets at the end of line, thus avoiding cache misses.

If we were to work with only one thread, the "optimized tiles" implementation offers the best result but as soon as we parallelize the program the "parallel for without if" gives us a better speed up which is why we have chose to submit this algorithm.

Result for the SCITAS run with the parameters: 100 iterations, length of 10000. We plotted the speed-up as asked in our comments of assignment1. We can see that the more threads, the less important the "if clause".