Assignment 2

Report

Dominique Roduit

1 - How have you parallelized and optimized *array_process*.

The first optimization was already given in the code, it was to adresse the 2-dimensional arrays like a unidimensional one. That way, we can take advantage of the locality in keeping the frequently accessed data closer to the CPU, in the cache, avoiding to have a cache miss on every element when we get the values of the 8 cells around the cell to update.

For the rest, the optimizations come from the parallelization of the code, scheduling and work distribution only, and not from increasing the number of cache hit.

First, I started with the sequential code, with 3 simple nested loops. The first loop iterates *iterations* times over the two other nested loops. These 2 nested loops simply iterate over the whole grid, avoiding the boarders, and calculating the average of the neighborhood cells, with a simple condition to avoid updating the middle hot core.

Then, I parallelized this code. To do so, I wrapped the outermost for loop in a bloc with

```
#pragma omp parallel private (row) shared (input, output) if(length > 24)
```

inside this loop, I parallelize 2 for loops with a guided schedule for the load balancing, as follows :

```
#pragma omp for schedule(guided)
```

The first for loop is in charge of updating the grid, and the second one is charged to copy the output array in the input array to keep the values updated through the iterations.

List the optimizations you applied and the reason you chose to apply them.

1. **Loop optimisations** : As we saw on the Lecture 4, it's better to wrap the whole bloc with *#pragma omp parallel {}* and to use *#pragma omp for on every loop*, rather than a *#pragma omp parallel* for on every for loop. This maximize the parallel regions in avoiding the excess fork/joins.

2. **Loop optimisations** : I tried to use the 3 different schedulings for the division of work on the 2 nested for loops (the one updating the cells of the grid and the one copying the output array). Finally, we can notice a better execution time with static scheduling compared to dynamic and guided schedulings.

3. use of the *if* clause : I execute the loop in parallel only if the side length > 24.

2 – Report how much each optimization improves performance and explain the result obtained

1 -

- with optimisation :
  Running the algorithm with 16 threads on 10000 by 10000 array for 500 iteration takes 28.07s seconds

- without optimisation :
  Running the algorithm with 16 threads on 10000 by 10000 array for 500 iteration takes 31.51s seconds

  We avoid excess fork / joins and save about 3.5 seconds.

2 -

with static scheduling (no scheduling specified) :
Running the algorithm with 16 threads on 10000 by 10000 array for 500 iteration takes 28.07s seconds

with guided scheduling :
Running the algorithm with 16 threads on 10000 by 10000 array for 500 iteration takes 31.06s seconds

with dynamic scheduling :
Running the algorithm with 16 threads on 10000 by 10000 array for 500 iteration takes 39.14s seconds

We have a very better execution time with static scheduling. Indeed, we have low overhead with static unlike dynamic that has big overhead. We can save about 11 seconds between static and dynamic, which is not negligible.

3 – Use of the *if* clause

- with if clause :
  Running the algorithm with 8 threads on 24 by 24 array for 5000 iteration takes 0.02256s seconds

- without the if clause :
  Running the algorithm with 8 threads on 24 by 24 array for 5000 iteration takes 0.04239s seconds

  This improvement is only valid for a length of < 24, slight and not very significant, but we parallelize the loops only if it’s worth it, and we can see that the parallelization is worse for a small side length.

Overall, I was disappointed because the improvements discussed in class was interesting to try but with only the basic parallelization we have already used in the first assignment, we have a better execution time. We can easily see the big difference with the parallelization on many threads compared to a single thread. But every further attempt of optimisations took time and were often useless because worse.

3 – Present the execution time you measured for the following set of thread counts {1,2,4,8,16} running for 500 iterations and side length of 10’000.

Threads	Execution time
1	216.3 s
2	114.8 s
4	61.99 s
8	48.15 s
16	28.07 s