High Performance Computing
Homework 1 - OpenMP report

Federico Fontana
s4835118@studenti.unige.it

Filippo Siri
s4819642@studenti.unige.it

Marco Zucca
s4828628@studenti.unige.it

March 2024

Contents

1 Task

The goal of this homework is to parallelise/vectorise the following program corresponding to an implementation of the Discrete Fourier Transform algorithm.

We will focus on these aspects: hotspot identification, possible vectorization issues, scalability using a proper number of threads.

2 Setup

All the data present in this report was collected by running the program on the workstations in laboratory room 210.

2.1 CPU

The workstation CPU is a 12th Gen Intel(R) Core(TM) i9-12900K, and running `lscpu | grep -E '^Thread|^Core|^Socket|^CPU\('` we get:

```
CPU(s):                          24
Thread(s) per core:              1
Core(s) per socket:              16
Socket(s):                       1
```

It's worth mentionig that this CPU architecure has 2 tifferent kind of core: 8 'e-core' meant to be used for lighter tasks and for that reason they have hyperthreading factor equal to one, while the other 8 'p-core' are meant to handle heavier tasks and they have an hyperthreading factor equal to 2.

For more information: Intel documentation.

2.2 Compiler

To compile the project we used Intel compiler `icc` with the following flags:

```
    -fopenmp -std=c99 -O3 -march=alderlake
```

It's worth mentioning that we also compiled the program with `-fast` instead of `-O3 -march=alderlake` and found the binaries to have better performance, but that led to issues when using Intel Advisor or the '-qopt-report' compiler flag, so we decided to use the latter for the results in the report.

3 Hotspot

Using Intel Advisor we found the location of the hotspot:

We notice that the hotspot is the loop that computes the DFT:

```
for (int k=0; k<N; k++)
{
   for (int n=0; n<N; n++)  {
      // Real part of X[k]
      Xr_o[k] += xr[n] * cos(n * k * PI2 / N) + idft*xi[n]*sin(n * k * PI2 / N);
      // Imaginary part of X[k]
      Xi_o[k] += -idft*xr[n] * sin(n * k * PI2 / N) + xi[n] * cos(n * k * PI2 / N);
   }
}
```

As it is setup in the original code, the inner loop cannot be easily vectorized, as all the writes in the inner loop in a single outer cycle would write in the same memory location. This is why the compiler automatically swaps the inner and outer loop and vectorizes the inner one. This can be done because the operations inside the double loop only read from `xr` and `xi` and only write to `Xr_o` and `Xi_o`, meaning that we have no other dependencies that hamper the possible optmiziations.

4 Optimisation

4.1 Caching trigonometric functions

The first optimization that we decided to make was to try to cache the results of the computations of the `sin` and `cos` functions, since both results are used two times for each loop. This was done by creating two variables at the beginning of the inner loop as such:

```
for (int n=0 ; n<N ; n++) {
      FTYPE c = COS(n * k * PI2 / N);
      FTYPE s = SIN(n * k * PI2 / N);

      // Real part of X[k]
      Xr_o[k] += xr[n] * c + idft*xi[n]*s;
      // Imaginary part of X[k]
      Xi_o[k] += -idft*xr[n] * s + xi[n] * c;
}
```

This led to an experimental improvement in performance of approximately 33%.

4.2 Vectorization

Later we decide to tackle the vectorization of the code in the `DFT` function. The compiler automatically swaps and vectorizes the inner loop, as we can verify with this line in the optimization report produced by `icc`:

```
LOOP BEGIN at src/omp_homework_vect_final.c(89,3) inlined into src/omp_homework_vect_final.c(56,5)
   remark #15542: loop was not vectorized: inner loop was already vectorized

   LOOP BEGIN at src/omp_homework_vect_final.c(89,3) inlined into src/omp_homework_vect_final.c(56,5)
      remark #15542: loop was not vectorized: inner loop was already vectorized
      LOOP BEGIN at src/omp_homework_vect_final.c(91,5) inlined into src/omp_homework_vect_final.c(56,
         remark #15388: vectorization support: reference xr has aligned access   [ src/omp_homework_ve
         remark #15388: vectorization support: reference xi has aligned access   [ src/omp_homework_ve
         remark #15388: vectorization support: reference xr has aligned access   [ src/omp_homework_ve
         remark #15388: vectorization support: reference xi has aligned access   [ src/omp_homework_ve
         remark #15305: vectorization support: vector length 4
         remark #15309: vectorization support: normalized vectorization overhead 0.326
         remark #15355: vectorization support: at (96:7) is double type reduction   [ src/omp_homework
         remark #15355: vectorization support: at (98:7) is double type reduction   [ src/omp_homework
         remark #15300: LOOP WAS VECTORIZED
         remark #15448: unmasked aligned unit stride loads: 3
         remark #15475: --- begin vector cost summary ---
         remark #15476: scalar cost: 301
         remark #15477: vector cost: 34.500
         remark #15478: estimated potential speedup: 8.720
         remark #15482: vectorized math library calls: 1
         remark #15486: divides: 1
         remark #15487: type converts: 1
         remark #15488: --- end vector cost summary ---
      LOOP END
   LOOP END
LOOP END
```

From the report we can also notice that the elements inside `Xi_o` are not aligned, and could thus decrease performance. To solve this problem, we decided to call `_mm_malloc` and `_mm_free` instead of the regular versions of the functions. Furthermore, we decided to allow the compiler to assume that all the arrays used in the function are aligned by adding four calls to `_assume_aligned`. This showed no performance improvements in the specific case of this program, since the function is inlined in `main` (TODO: aggiungere ref da report), but could be used by the compiler in other scenarios to apply the same optimizations.

The same holds for the `[restrict]` we added on the function parameters. This has no effect in our specific scenario, but allows the compiler to assume that the pointers do not alias, and as such allows it to vectorize the inner loop, since it can assume that we're not reading and writing in the same memory position. This can be seen from the following optimization report lines:

```
LOOP BEGIN at src/omp_homework_vect_final.c(89,3)
   remark #15542: loop was not vectorized: inner loop was already vectorized

   LOOP BEGIN at src/omp_homework_vect_final.c(89,3)
      remark #15542: loop was not vectorized: inner loop was already vectorized

      LOOP BEGIN at src/omp_homework_vect_final.c(91,5)
         remark #15388: vectorization support: reference xr[n] has aligned access   [ src/omp_homework
         remark #15388: vectorization support: reference xi[n] has aligned access   [ src/omp_homework
         remark #15388: vectorization support: reference xr[n] has aligned access   [ src/omp_homework
         remark #15388: vectorization support: reference xi[n] has aligned access   [ src/omp_homework
         remark #15305: vectorization support: vector length 4
         remark #15309: vectorization support: normalized vectorization overhead 0.326
         remark #15355: vectorization support: *(Xr_o+k*8) is double type reduction   [ src/omp_homewo
         remark #15355: vectorization support: *(Xi_o+k*8) is double type reduction   [ src/omp_homewo
         remark #15300: LOOP WAS VECTORIZED
         remark #15448: unmasked aligned unit stride loads: 3
         remark #15475: --- begin vector cost summary ---
         remark #15476: scalar cost: 301
         remark #15477: vector cost: 34.500
         remark #15478: estimated potential speedup: 8.720
         remark #15482: vectorized math library calls: 1
         remark #15486: divides: 1
         remark #15487: type converts: 1
         remark #15488: --- end vector cost summary ---
      LOOP END
   LOOP END
LOOP END
```

From the same report we can also notice that the last loop that is executed only in the case of the inverse DFT has also been vectorized:

```
LOOP BEGIN at src/omp_homework_vect_final.c(104,5)
   remark #15388: vectorization support: reference Xr_o[n] has aligned access   [ src/omp_homework_vec
   remark #15388: vectorization support: reference Xr_o[n] has aligned access   [ src/omp_homework_vec
   remark #15388: vectorization support: reference Xi_o[n] has aligned access   [ src/omp_homework_vec
   remark #15388: vectorization support: reference Xi_o[n] has aligned access   [ src/omp_homework_vec
   remark #15305: vectorization support: vector length 4
   remark #15399: vectorization support: unroll factor set to 4
   remark #15300: LOOP WAS VECTORIZED
   remark #15448: unmasked aligned unit stride loads: 2
   remark #15449: unmasked aligned unit stride stores: 2
   remark #15475: --- begin vector cost summary ---
   remark #15476: scalar cost: 61
   remark #15477: vector cost: 17.500
   remark #15478: estimated potential speedup: 3.480
   remark #15486: divides: 2
   remark #15488: --- end vector cost summary ---
LOOP END
```

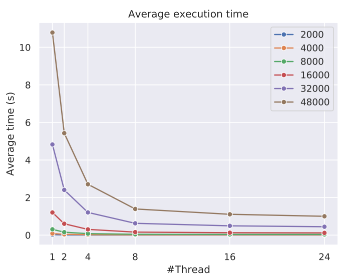
4.3 Parallelization

In the last optimisation step, we used OpenMP to split the calculations on multiple threads. First we parallelized the program by adding a simple `#pragma omp parallel for num_threads(THREAD_NO)` above the loop over the values for `k`. This yielded good results, but upon further analyzing the structure and the characteristics of the CPU we decided to use dynamic scheduling by adding the following directive `schedule(dynamic)` to the aforementioned line. This instructs the compiler not to assume that each thread will take the same amount of time to compute its chunk of data, and as such it will dynamically schedule the chunks of data to each CPU thread at runtime, instead of allocating static chunks of equal length at compile time. This improved the performance of our program, since the CPU we used has two different kinds of cores with different specs.

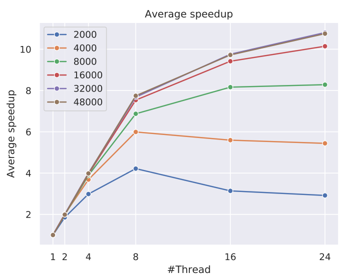
5 Results

To test the algorithm, we used matrices of dimensions 2000×2000, 4000×4000, 8000×8000, 16000×16000, 32000×32000, 48000×48000 and tested the program using 1, 2, 4, 8, 16 and 24 threads. To get better results we ran the algorithm with every possible combination 10 times and then averaged the times obtained.

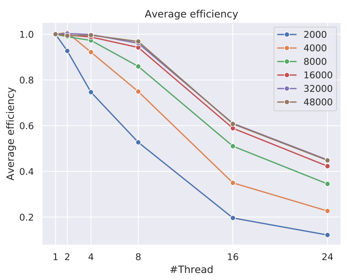
We compared the values obtained by plotting the average time, the speedup obtained and the efficiency. The raw data can be consulted in the table 1.



(a) Average time



(b) Average speedup



(c) Average efficiency

For small input matrix sizes such as 2000×2000 and 4000×4000, the speedup and efficiency values are not very high. In particular, we notice that 16 and 24 threads seem to slow the program down relative to the runs with 8 threads, meaning that the overhead brought by the creation of the threads and the switch between them is higher than the gain in performance. This is also visible in the plots, where the speedup values increase from 8 to 24 threads, and efficiency values thus decrease down to a minimum of 0.12154 for the 2000×2000 matrix.

As the size of the input matrix increases, the slowdown in execution time caused by the overhead of having more threads isn't felt, but the speedup and efficiency values signal that the program is not using the threads to their full potential.

The speedup and efficiency values increase considerably as the size of the matrix increases, with the highest recorded values for 24 threads being measured for the 32000×32000 and the 48000×48000 matrix.

The program is not perfectly parallelizable and shows sub-linear speedup for large thread numbers, but as the matrix size increases the speedup is closer to linear for larger thread numbers.

6 Conclusion

In conclusion, with this assignment, we had the chance to appreciate the change in performance of a fairly simple algorithm, by leveraging vectorization and multithreading. Other than that, we also noticed how small changes in the code could lead to significant change in performance.

Table 1: Measurements

Size	Thread Number	Average Time	Speedup	Efficiency
2000	1	0.02420	-	-
2000	2	0.01306	1.85351	0.92675
2000	4	0.00811	2.98537	0.74634
2000	8	0.00574	4.21594	0.52699
2000	16	0.00771	3.13799	0.19612
2000	24	0.00830	2.91701	0.12154
4000	1	0.08628	-	-
4000	2	0.04284	2.01417	1.00709
4000	4	0.02341	3.68531	0.92133
4000	8	0.01440	5.99219	0.74902
4000	16	0.01542	5.59497	0.34969
4000	24	0.01586	5.44042	0.22668
8000	1	0.30822	-	-
8000	2	0.15586	1.97750	0.98875
8000	4	0.07921	3.89117	0.97279
8000	8	0.04485	6.87218	0.85902
8000	16	0.03776	8.16226	0.51014
8000	24	0.03721	8.28230	0.34510
16000	1	1.20565	-	-
16000	2	0.60491	1.99313	0.99656
16000	4	0.30519	3.95053	0.98763
16000	8	0.16000	7.53547	0.94193
16000	16	0.12801	9.41810	0.58863
16000	24	0.11885	10.14405	0.42267
32000	1	4.82688	-	-
32000	2	2.40619	2.00602	1.00301
32000	4	1.20876	3.99325	0.99831
32000	8	0.62845	7.68056	0.96007
32000	16	0.49463	9.75862	0.60991
32000	24	0.44667	10.80627	0.45026
48000	1	10.78398	-	-
48000	2	5.42830	1.98662	0.99331
48000	4	2.70766	3.98276	0.99569
48000	8	1.39241	7.74484	0.96811
48000	16	1.10920	9.72234	0.60765
48000	24	1.00280	10.75384	0.44808