Università
di Genova

MSc Computer Science
Data Science and Engineering Curriculum

OpenMP assignment

High Performance Computing

Roberto Di Via

December, 2022

Contents

1 Introduction

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

1.1 How to compile the program

First of all, I have to connect to the INFN Cluster through the command:

```
ssh -CYJ rodivia@linuxge.ge.infn.it  rodivia@hpcocapie09.ge.infn.it
```

Then I use the Intel compiler ICC with the following flags:

-On, n=0,1,2,3,fast	Specify an optimization level, the default is -O2.
-qopenmp	Generate multi-threaded code based on OpenMP* directives.
-xHOST	Generate instructions for the highest instruction set available.
-qopt-report	Generate an optimization report. Levels goes from 0 to 5, default is 2.
-qopt-report-phase	Specifies optimizer phases for which optimization reports are generated.

I use the flags "**-qopt-report=5**" and "**-qopt-report-phase=vec**" in order to generate the vectorization report and so to understand more what the compiler does, and how I can improve the code to get better performance.

So, the shell line command that I use to compile my optimized version is:

```
icc -O2 -xHost -qopenmp omp_homework.c -o homework
```

and to execute it I simply do:

```
./homework
```

2 Analysis of the program

The program is composed of the following functions:

```
int DFT(int idft, double* xr, double* xi, double* Xr_o, double* Xi_o, int N);
int fillInput(double* xr, double* xi, int N);
int setOutputZero(double* Xr_o, double* Xi_o, int N);
int checkResults(double* xr, double* xi, double* xr_check,
                 double* xi_check, double* Xr_o, double* Xi_r, int N);
int printResults(double* xr, double* xi, int N);

```

and the computation time is calculated immediately before/after the DFT calls using the openMP function **omp_get_wtime()**.

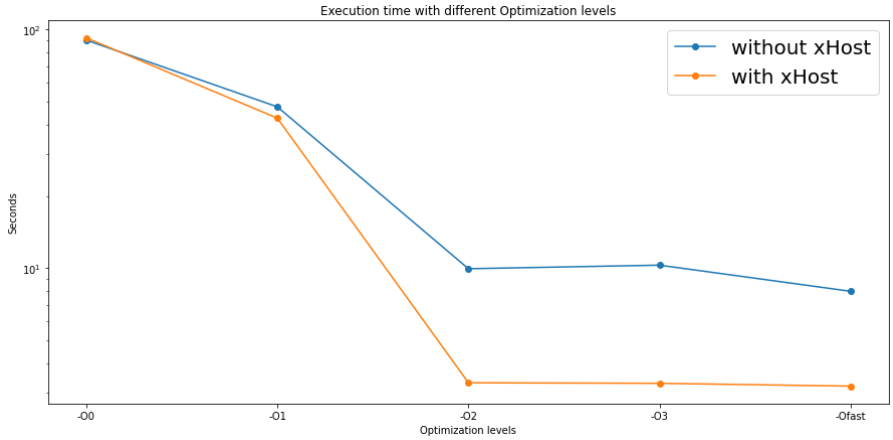
```
    // start timer
    double start_time = omp_get_wtime();

    // DFT
    int idft = 1;
    DFT(idft,xr,xi,Xr_o,Xi_o,N);
    // IDFT
    idft = -1;
    DFT(idft,Xr_o,Xi_o,xr_check,xi_check,N);

    // stop timer
    double run_time = omp_get_wtime() - start_time;
    printf("DFTW computation in %f seconds\n",run_time);
```

2.1 Execution of the Serial version

First I tried compiling and executing the serial version, according to the vectorization levels of the compiler, and the results (in seconds) that I got are the followings:

	-O0	-O1	-O2	-O3	-Ofast
	90.106722	47.384940	9.950080	10.297998	8.005422
-xHost	92.105992	42.465436	3.314895	3.296664	3.208941

Besides changing the optimization level, I also tried compiling with and without the **-xHost** flag which tells the compiler to generate instructions for the highest instruction set available on the compilation host processor. From this experiment, I could see that the results improved by using the flag, in particular by keeping the standard one (-O2) as the optimization level.



As we can see from this chart using the flag xHost the execution time decreases a lot faster. In order to highlight the differences I used a logarithmic scale for the y-axis.

2.2 Hotspot identification

Hotspot identification is an important step in the parallelism process. Within a program, it is the busy sections or hotspots, that should be made parallel. The more the hotspots contribute to the overall run time of the program, the better the performance improvement you will obtain by parallelizing them. A common approach is to identify the loops inside the program that consumes a significant amount of time. In this case, in the DFT function, there are two nested loops.

```
1  // idft: 1 direct DFT, -1 inverse IDFT (Inverse DFT)
2  int DFT(int idft, double* xr, double* xi, double* Xr_o, double* Xi_o, int N){
3    int k, n;
4    for (k=0 ; k<N ; k++)
5    {
6        for (n=0 ; n<N ; n++)  {
7            // Real part of X[k]
8            Xr_o[k] += xr[n] * cos(n * k * PI2 / N) + idft*xi[n]*sin(n * k * PI2 / N);
9            // Imaginary part of X[k]
10           Xi_o[k] += -idft*xr[n] * sin(n * k * PI2 / N) + xi[n] * cos(n * k * PI2 / N);
11       }
12   }
13
14   // normalize if you are doing IDFT
15   if (idft==-1){
16     for (n=0 ; n<N ; n++){
17       Xr_o[n] /=N;
18       Xi_o[n] /=N;
19     }
20   }
21   return 1;
22 }
```

In these two nested loops the DFT function computes real values and imaginary values separately and stores their values in two different arrays. So my attempt to parallelize the program started putting an 'omp parallel for' directive in line 6.

The **omp parallel for** directive effectively combines the omp parallel and omp for directives. This directive lets you define a parallel region containing a single directive in one step. I used **private** because variables can have either shared or private context in a parallel environment. Variables in a shared context are visible to all threads running in associated parallel regions. Variables in a private context are hidden from other threads.

In the private clause, I passed both the variables (k and n) because otherwise the value of 'n' is incremented every time and so other threads can't use the correct 'n' for their computation.

Later, in line 19, I used **omp parallel for** in order to parallelize also the last minor loop.

```
1
2  int DFT(int idft, double* xr, double* xi, double* Xr_o, double* Xi_o, int N){
3    int k, n;
4    int nthreads, tid;
5
6    #pragma omp parallel for private(k, n)
7    for (k=0 ; k<N ; k++)
8    {
9        for (n=0 ; n<N ; n++)  {
10           // Real part of X[k]
11           Xr_o[k] += xr[n] * cos(n * k * PI2 / N) + idft*xi[n]*sin(n * k * PI2 / N);
12           // Imaginary part of X[k]
13           Xi_o[k] += -idft*xr[n] * sin(n * k * PI2 / N) + xi[n] * cos(n * k * PI2 / N);
14       }
15   }
16
17   // normalize if you are doing IDFT
18   if (idft==-1){
19     #pragma omp parallel for
20     for (n=0 ; n<N ; n++){
21       Xr_o[n] /=N;
22       Xi_o[n] /=N;
23     }
24   }
25   return 1;
26 }
```

The result of these changes (visible in the code above) is a definitely lower time in seconds, as described later in section 2.4.

2.3 Possible vectorization issues

Let's start defining what is vectorization. Vectorization is the process of converting an algorithm from operating on a single value at a time to operating on a set of values (vector) at one time. In order to identify the possible vectorization issues I simply read the report generated by **-qopt-report**, in this case, generated with the optimization level "**-O2**".

```
LOOP BEGIN at omp_homework.c(79,7) inlined into omp_homework.c(42,5)
   remark #25444: Loopnest Interchanged: ( 1 2 ) --> ( 2 1 )
   remark #15542: loop was not vectorized: inner loop was already vectorized

   LOOP BEGIN at omp_homework.c(76,3) inlined into omp_homework.c(42,5)
   <Peeled loop for vectorization>
   LOOP END

   LOOP BEGIN at omp_homework.c(76,3) inlined into omp_homework.c(42,5)
      remark #25427: Loop Statements Reordered
      remark #15301: PERMUTED LOOP WAS VECTORIZED
   LOOP END

   LOOP BEGIN at omp_homework.c(76,3) inlined into omp_homework.c(42,5)
   <Remainder loop for vectorization>
   LOOP END
LOOP END
```

The compiler says that the first loop inside DFT is not vectorized, so it starts with a **Peel loop** that involves executing the first few iterations of the loop separately from the rest of the loop. This loop optimization technique can help to eliminate dependencies. Later the compiler does a **Loop permutation** that involves rearranging the loops in a way that allows the processor to make better use of its cache memory, or by rearranging the loops to better exploit data dependencies between iterations. These improvements are done for the first nested loop.

The latest loop (where there is normalization in the case of IDFT) is vectorized without any problems.

2.4 Scalability using a different number of threads

In order to achieve better performance, I carried out the experiments with a different number of threads, with different vectorization levels. The results are shown in the table below.

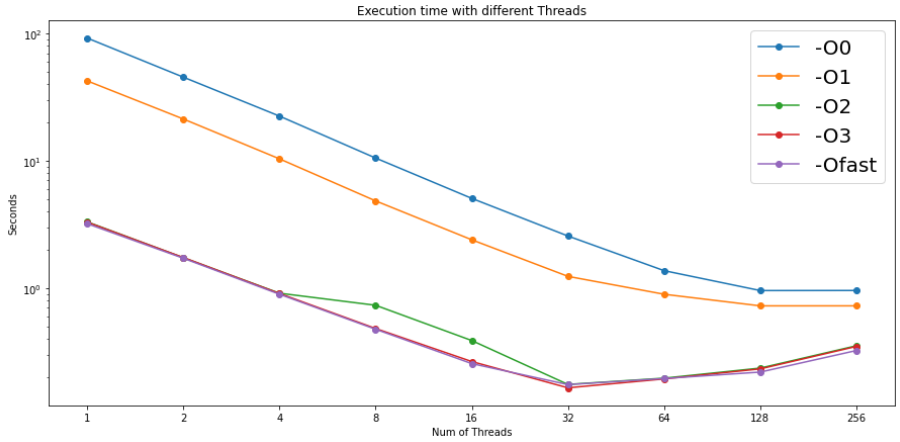
To set the number of threads I used:

```
export OMP_NUM_THREADS=<n>
```

where $< n >$ corresponds to the number of threads I want to use.

Num threads	-O0	-O1	-O2	-O3	-Ofast
2	45.412845	21.304359	1.733825	1.739835	1.716107
4	22.441435	10.362354	0.911925	0.907656	0.895164
8	10.512963	4.842752	0.733178	0.481678	0.475150
16	5.066985	2.396968	0.387434	0.265025	0.255276
32	2.566579	1.236884	**0.175175**	0.164723	0.174604
64	1.371186	0.894756	**0.196229**	0.193491	0.194986
128	0.956494	0.725453	0.235403	0.232100	0.218958
256	0.957998	0.725570	0.352540	0.348250	0.323433

As can be seen from both the table and the graph below, the best results are obtained with an optimization level of -O2 and with a number of threads equal to 32/64 (as well as the number of physical cores of the machine).



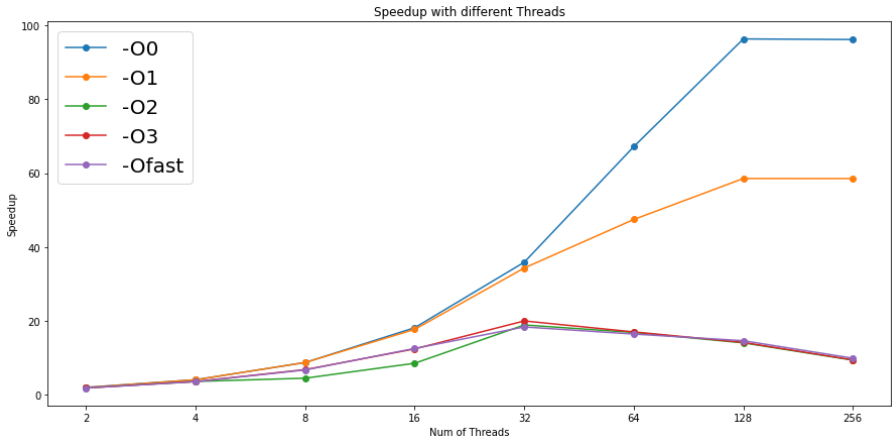
Using optimization levels higher than -O2 does not lead to great results in this specific case. On the other hand, using more threads than the number of cores can lead to reduced performance due to the overhead of context switching between threads.

2.5 The Speedup

Speedup is a measure of how much faster a parallelized program can run compared to the same program running on a single processor. There are several reasons why it is useful to calculate the speedup of a parallel computing system. By calculating it, we can compare the relative performance of different parallel programs. We can determine the optimal configuration under different configurations (e.g., using different numbers of threads), or we can determine how well the system is utilizing its resources and so identify inefficiencies.

The speedup in parallel computing can be defined as:

$$Speedup = \frac{Time\ on\ single\ processor}{Time\ on\ parallel\ system}$$



When the program is compiled with the default optimization level or a higher one, the speedup increases as the number of threads increases, up to 32 threads. Beyond this point, the speedup starts to decrease, this is due to the overhead of communication and synchronization between threads. This means that when using 32 threads or less, the program is able to efficiently utilize the available resources, but as the number of threads increases, the overhead becomes more significant and the program's performance starts to decrease.

On the other hand, when the program is compiled with no optimizations, the speedup also increases as the number of threads increases, but the point of saturation is reached at 128 threads. This means that in this case, the program is not able to efficiently utilize the available resources when using more than 128 threads.

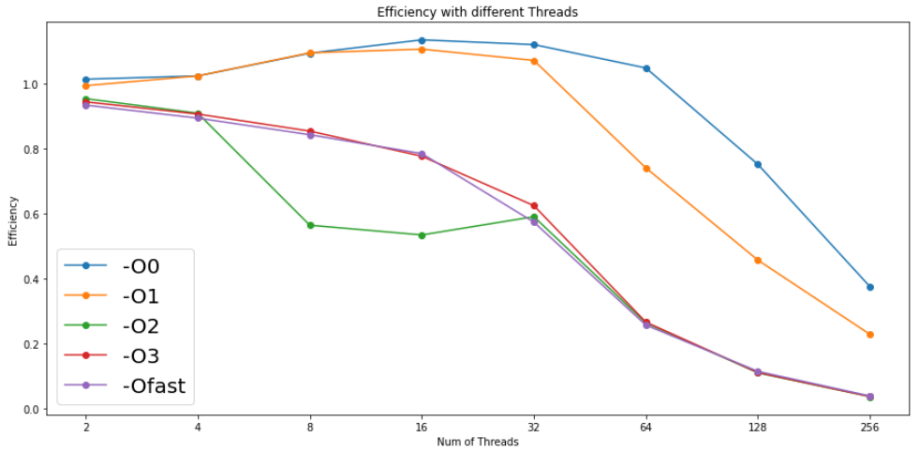
2.6 The Efficiency

The efficiency of a parallelized program is a measure of how well the program utilizes the available resources, it is calculated as the ratio of the speedup to the number of threads used.

$$E = \frac{Speedup}{Number\ of\ Threads}$$

Where *Speedup* is the ratio of the execution time of the program on a single processor to the execution time of the program on multiple processors.

In general, a parallelized program can be more efficient than a single-threaded program because it can take advantage of multiple processors or cores to perform tasks simultaneously. However, if the overhead of communication and synchronization between the processes or threads is high, it may decrease the overall efficiency of the program.



As the chart shows, the efficiency of the program decreases as the number of threads used increases, and this trend is consistent across all optimization levels. The efficiency starts to decrease significantly after 32 threads are used. This suggests that when using more than 32 threads, the program is not able to efficiently utilize the available resources and the overhead of communication and synchronization becomes more significant.

3 Conclusions

In this assignment, I aimed to optimize the performance of a program in C language through the use of vectorization and parallelization techniques, specifically using OMP and the ICC compiler. To evaluate the effectiveness of these techniques, I measured the execution time, speedup, and efficiency of the program using a variety of different numbers of threads and optimization levels.

From my experiments, I discovered that the best results were achieved when using 32/64 threads. This aligns with the physical cores in my machine, suggesting that the program is able to efficiently utilize the available resources.

Furthermore, I found that using a larger number of threads did not always lead to better performance, indicating the importance of finding the optimal balance between the number of threads and the workload. As I increased the number of threads, I observed that the execution time of the program decreased up to a certain point. Beyond that point, the program reaches a saturation point where the execution time starts to increase and the speedup value doesn't increase any further.

The results of my experiments also show that the use of a higher optimization level allows the program to efficiently utilize the available resources with a lower number of threads.