Università di Genova

Computer Engineerig

OpenMP assignment

High Performance Computing

Martin Martuccio , Lorenzo Mesi

April, 2024

Contents

1 Introduction

The purpose of this assignment is to parallelize and vectorize a program that implements the Discrete Fourier Transform (DFT) algorithm to enhance performance and minimize execution time. The focus will be on code analysis to identify computational bottlenecks, address vectorization issues, and evaluate the program's scalability with an increased number of execution threads.

1.1 Compiling the program

The compilation process played a crucial role in optimizing our program. This process was conducted in the **SW2** Lab, which is equipped with advanced computing systems.

To compile our program, we used the Intel C Compiler (ICC). The ICC offers a wide range of options for tuning applications for specific hardware architectures, thus maximizing performance.

We experimented with various compiler flags that allowed us to exploit different features provided by the compiler. Below is a list of the flags which icc offers:

–On, n=0,1,2,3,4,fast

Specify an optimization level, the default is -O2.

–openmp

Generate multi-threaded code based on OpenMP* directives.

–xHOST

Generate instructions for the highest instruction set available of a specific CPU.

–qopt–report =<n> n = 0,1,2,3,4

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.

–arch=

Specifies which kind of architecture you want to use for compiling the program.
So, an example of the shell line command that we used to compile the optimized version is:

```
icc -O2 -xHost -qopenmp project.c -o project
```

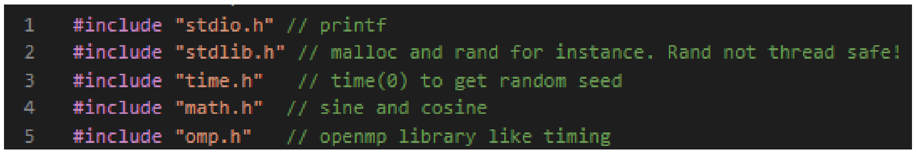
and to execute it we simply do:

```
./project
```

2 Analysis of the program

Libraries and Preprocessor Directives:

The code begins with the inclusion of standard libraries for input/output operations (stdio.h), dynamic memory allocation and random number generation (stdlib.h), time retrieval (time.h), and mathematical functions (math.h). The OpenMP library (omp.h) is included for parallel programming support.

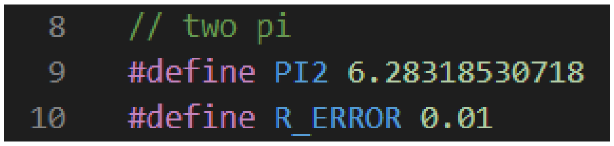


```c
#include "stdio.h" // printf
#include "stdlib.h" // malloc and rand for instance. Rand not thread safe!
#include "time.h"   // time(0) to get random seed
#include "math.h"  // sine and cosine
#include "omp.h"   // openmp library like timing
```

Figure 1: Standard and parallel programming libraries.

Constants Definitions:

PI2 is defined as the constant 2π, which is used in the computation of the DFT. R_{ERROR} is a threshold for checking the results' accuracy.

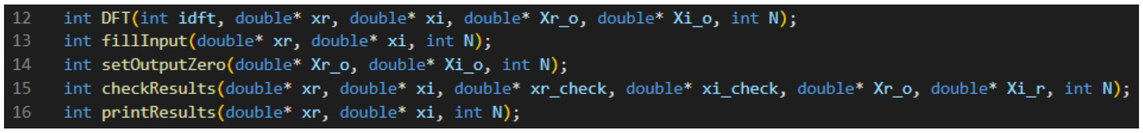


```c
// two pi
#define PI2 6.28318530718
#define R_ERROR 0.01
```

Figure 2: Constants.

Function Prototypes:

Five functions are declared which include *DFT* for computing the direct and inverse DFT, *fillInput* for initializing the signal arrays, *setOutputZero* for zeroing the output arrays, *checkResults* for validating the DFT results, and *printResults* for printing the transformed data.

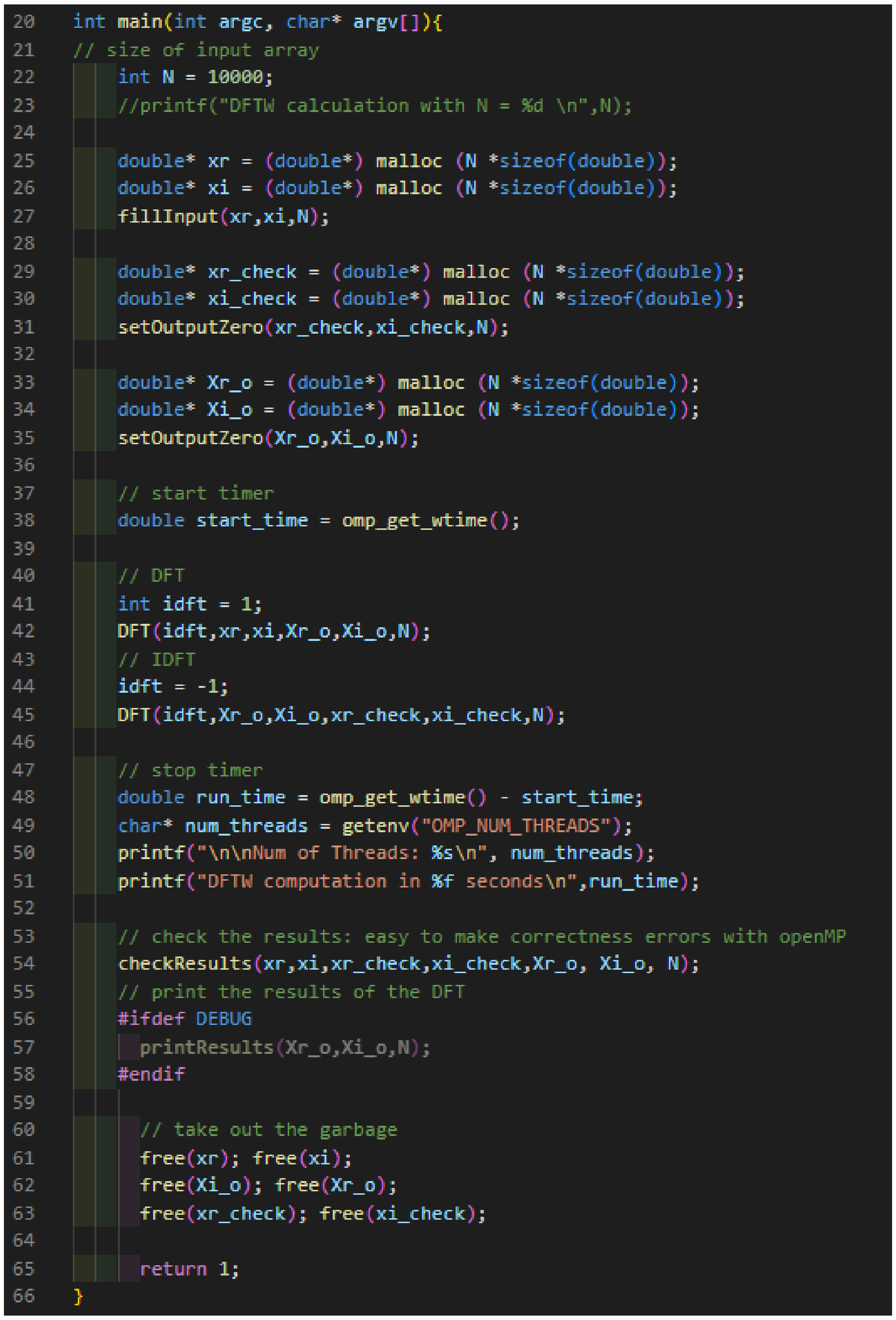


```
int DFT(int idft, double* xr, double* xi, double* Xr_o, double* Xi_o, int N);
int fillInput(double* xr, double* xi, int N);
int setOutputZero(double* Xr_o, double* Xi_o, int N);
int checkResults(double* xr, double* xi, double* xr_check, double* xi_check, double* Xr_o, double* Xi_r, int N);
int printResults(double*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

Figure 3: Declaration of the functions.

Main Function:

The main function of your program handles the calculation of the Discrete Fourier Transform (DFT). Initially, it allocates memory for the input signals and results, then initializes the data and clears the output arrays. Next, it executes both the DFT and its inverse (IDFT), measuring the execution time with OpenMP's timing functions. After the calculation, it checks the correctness of the results and, if the flag "DEBUG" is activated, prints the transformed data. Finally, release the allocated memory before finishing execution.

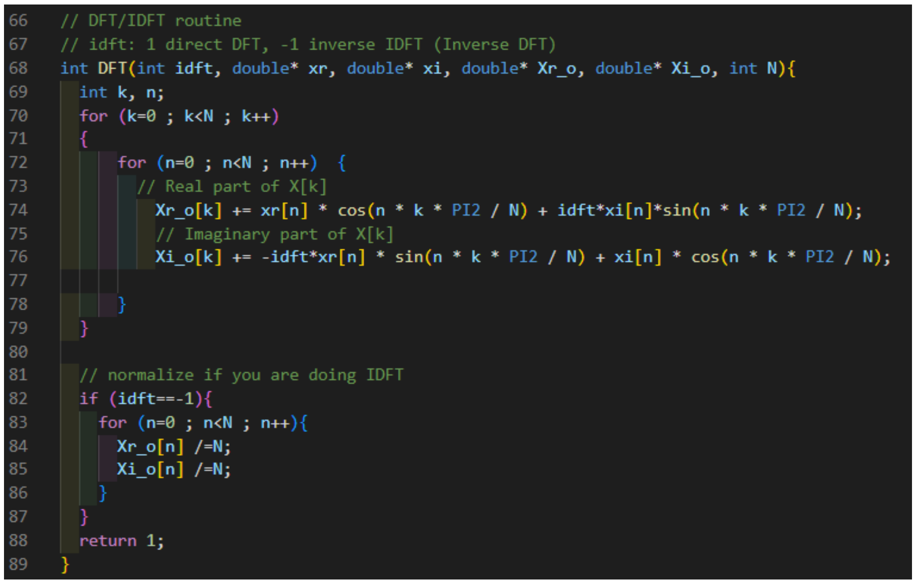


```c
int main(int argc, char* argv[]){
// size of input array
    int N = 10000;
    //printf("DFTW calculation with N = %d \n",N);

    double* xr = (double*) malloc (N *sizeof(double));
    double* xi = (double*) malloc (N *sizeof(double));
    fillInput(xr,xi,N);

    double* xr_check = (double*) malloc (N *sizeof(double));
    double* xi_check = (double*) malloc (N *sizeof(double));
    setOutputZero(xr_check,xi_check,N);

    double* Xr_o = (double*) malloc (N *sizeof(double));
    double* Xi_o = (double*) malloc (N *sizeof(double));
    setOutputZero(Xr_o,Xi_o,N);

    // 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;
    char* num_threads = getenv("OMP_NUM_THREADS");
    printf("\n\nNum of Threads: %s\n", num_threads);
    printf("DFTW computation in %f seconds\n",run_time);

    // check the results: easy to make correctness errors with openMP
    checkResults(xr,xi,xr_check,xi_check,Xr_o, Xi_o, N);
    // print the results of the DFT
    #ifdef DEBUG
     printResults(Xr_o,Xi_o,N);
    #endif

      // take out the garbage
      free(xr); free(xi);
      free(Xi_o); free(Xr_o);
      free(xr_check); free(xi_check);

      return 1;
}
```

Figure 4: Main function.

DFT/IDFT Computation:

The DFT/IDFT function is the core of the algorithm, calculating the DFT or its inverse depending on the value of the ‘idft‘ parameter. It uses two nested loops to process the real and imaginary parts of the signal separately.

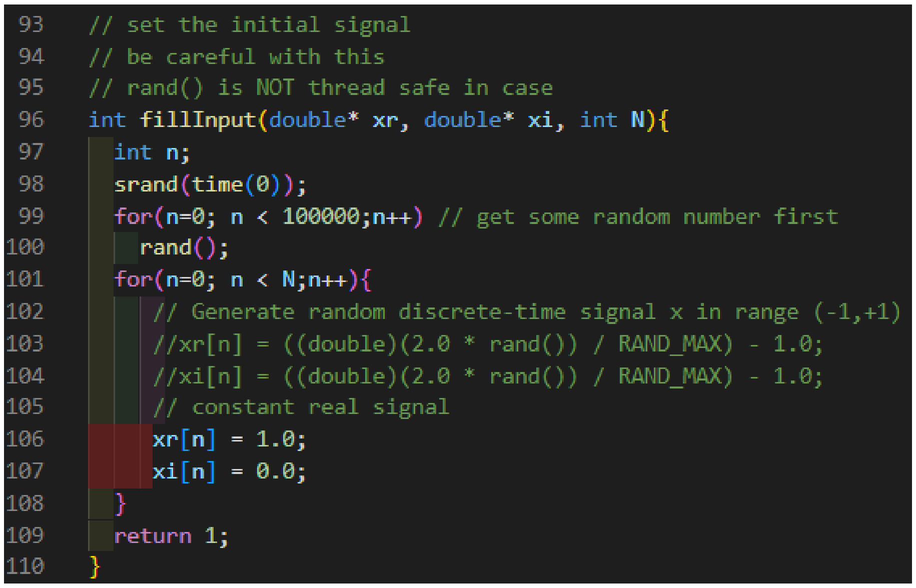


```
// DFT/IDFT routine
// idft: 1 direct DFT, -1 inverse IDFT (Inverse DFT)
int DFT(int idft, double* xr, double* xi, double* Xr_o, double* Xi_o, int N){
    int k, n;
    for (k=0 ; k<N ; k++)
    {
        for (n=0 ; n<N ; n++)  {
          // Real part of X[k]
            Xr_o[k] += xr[n] * cos(n * 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);

        }
    }

    // normalize if you are doing IDFT
    if (idft==-1){
      for (n=0 ; n<N ; n++){
        Xr_o[n] /=N;
        Xi_o[n] /=N;
      }
    }
    return 1;
}
```

Figure 5: Core of the algorithm.

Input Initialization:

The fillInput function initializes the input signal arrays with a constant real part and a zero imaginary part.



```
// set the initial signal
// be careful with this
// rand() is NOT thread safe in case
int fillInput(double* xr, double* xi, int N){
  int n;
  srand(time(0));
  for(n=0; n < 100000;n++) // get some random number first
    rand();
  for(n=0; n < N;n++){
     // Generate random discrete-time signal x in range (-1,+1)
     //xr[n] = ((double)(2.0 * rand()) / RAND_MAX) - 1.0;
     //xi[n] = ((double)(2.0 * rand()) / RAND_MAX) - 1.0;
     // constant real signal
     xr[n] = 1.0;
     xi[n] = 0.0;
  }
  return 1;
}
```

Figure 6: Input Initialization.

Output Initialization:

The setOutputZero function initializes with all zeros the output arrays in preparation for the DFT computation.

Results Checking:

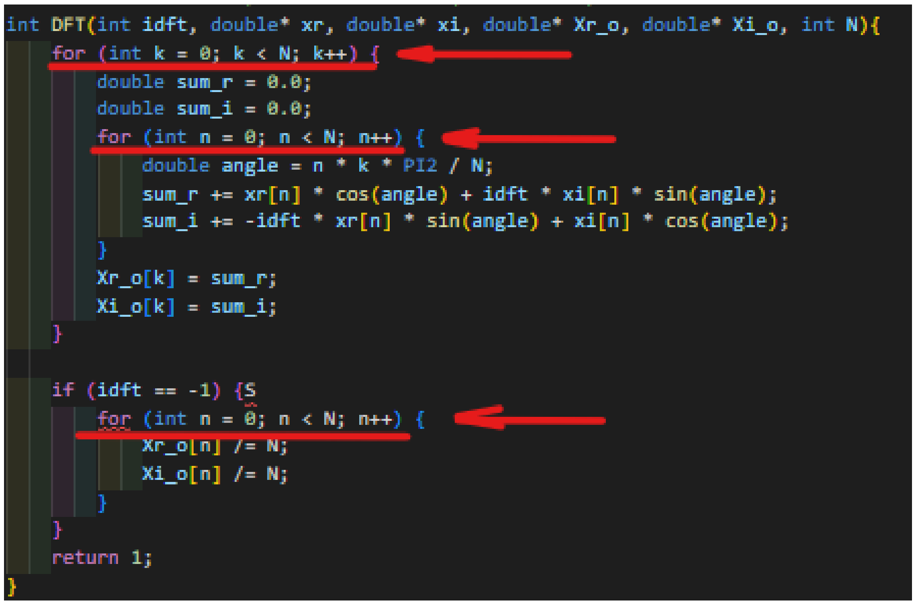
The checkResults function compares the original signal arrays with the transformed-then-inverted signal to ensure the DFT and IDFT are functioning correctly, taking into account a small rounding error defined by R_{ERROR}.

Results Printing:

Lastly, the printResults function outputs the real and imaginary parts of the transformed signal.

2.1 Hotspot of the program

Identifying hotspots is a critical phase in the parallelization workflow. In any given program, it's the sections with intensive activity, known as hotspots, that are prime candidates for parallelization. A typical strategy involves pinpointing the program's loops that account for a considerable portion of the execution time. Specifically, within the DFT function, there exist two nested loops.

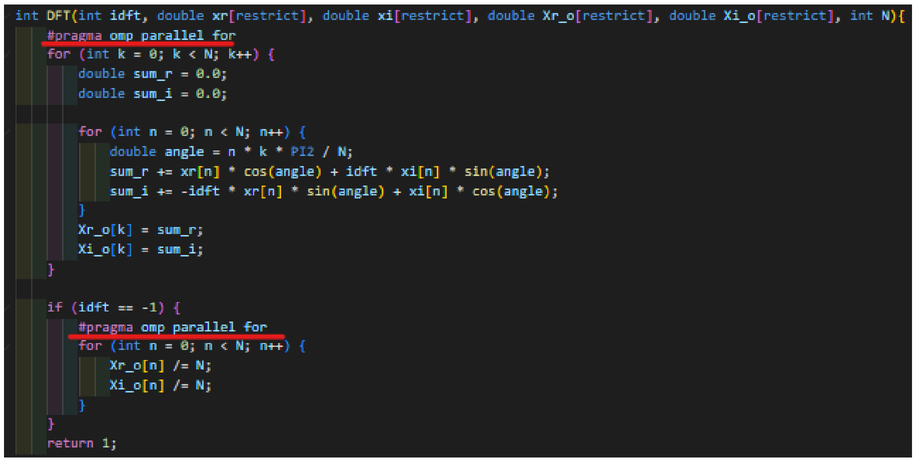


```
int DFT(int idft, double* xr, double* xi, double* Xr_o, double* Xi_o, int N){
    for (int k = 0; k < N; k++) {
        double sum_r = 0.0;
        double sum_i = 0.0;
        for (int n = 0; n < N; n++) {
            double angle = n * k * PI2 / N;
            sum_r += xr[n] * cos(angle) + idft * xi[n] * sin(angle);
            sum_i += -idft * xr[n] * sin(angle) + xi[n] * cos(angle);
        }
        Xr_o[k] = sum_r;
        Xi_o[k] = sum_i;
    }

    if (idft == -1) {S
        for (int n = 0; n < N; n++) {
            Xr_o[n] /= N;
            Xi_o[n] /= N;
        }
    }
    return 1;
}
```

Figure 7: Hotspot

Our approach to program parallelization began by placing an '*omp parallel for*' directive at the beginning of the outer loop. This directive allows you to define a parallel region containing a single directive in one step.

We used omp parallel for in order to parallelize also the last minor loop.



```c
int DFT(int idft, double xr[restrict], double xi[restrict], double Xr_o[restrict], double Xi_o[restrict], int N){
    #pragma omp parallel for
    for (int k = 0; k < N; k++) {
        double sum_r = 0.0;
        double sum_i = 0.0;

        for (int n = 0; n < N; n++) {
            double angle = n * k * PI2 / N;
            sum_r += xr[n] * cos(angle) + idft * xi[n] * sin(angle);
            sum_i += -idft * xr[n] * sin(angle) + xi[n] * cos(angle);
        }
        Xr_o[k] = sum_r;
        Xi_o[k] = sum_i;
    }

    if (idft == -1) {
        #pragma omp parallel for
        for (int n = 0; n < N; n++) {
            Xr_o[n] /= N;
            Xi_o[n] /= N;
        }
    }
    return 1;
```

Figure 8: Parallelization of loops

2.2 Possible vectorization issues

Vectorization is a method of increasing performance by allowing the processor to process more data in parallel. This is achieved by performing operations on sets of values, or vectors, at the same time, rather than on a single value at a time.

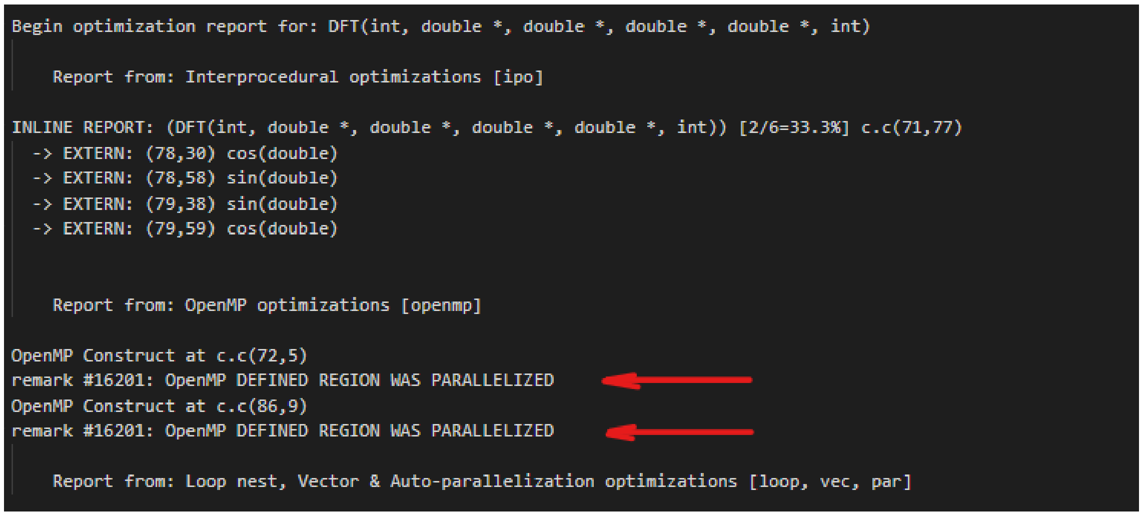
When using OpenMP to parallelize a program, complications with vectorization can arise. Common issues include managing data dependencies, properly aligning data in the cache, and using the architecture properly.

The use of specific register sets, such as AVX-512, which allow you to exploit the architecture more effectively, a permutation of operations are all useful strategies to overcome these obstacles.

To identify these problems, you can consult the compiler's vectorization report, generated by the *-qopt-report* option.

In some cases we need to specify data alignment, restructure the code and 'force' code parallelization because the compiler may avoid it to prevent erroneous results.

```
Compiler options: -O2 -fopenmp -qopt-report=5 -o Test
```



```
Begin optimization report for: DFT(int, double *, double *, double *, double *, int)

    Report from: Interprocedural optimizations [ipo]

INLINE REPORT: (DFT(int, double *, double *, double *, double *, int)) [2/6=33.3%] c.c(71,77)
  -> EXTERN: (78,30) cos(double)
  -> EXTERN: (78,58) sin(double)
  -> EXTERN: (79,38) sin(double)
  -> EXTERN: (79,59) cos(double)


    Report from: OpenMP optimizations [openmp]

OpenMP Construct at c.c(72,5)
remark #16201: OpenMP DEFINED REGION WAS PARALLELIZED
OpenMP Construct at c.c(86,9)
remark #16201: OpenMP DEFINED REGION WAS PARALLELIZED

    Report from: Loop nest, Vector & Auto-parallelization optimizations [loop, vec, par]
```

Figure 9: Report of the vectorized program

3 Measurements of performance with :

Now, we present a series of experiments conducted to evaluate the performance of the program using various optimization techniques.

3.1 Vectorization

Here we explore the effect of vectorization on performance. Vectorization allows the program to process multiple data simultaneously. We generated a graph having as abscissa the optimization level and on the ordered the execution time.

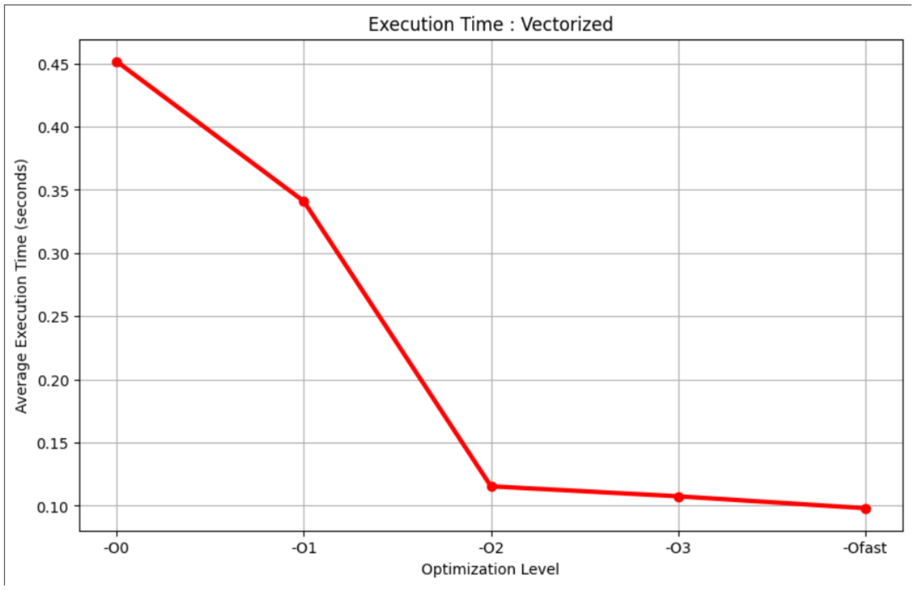


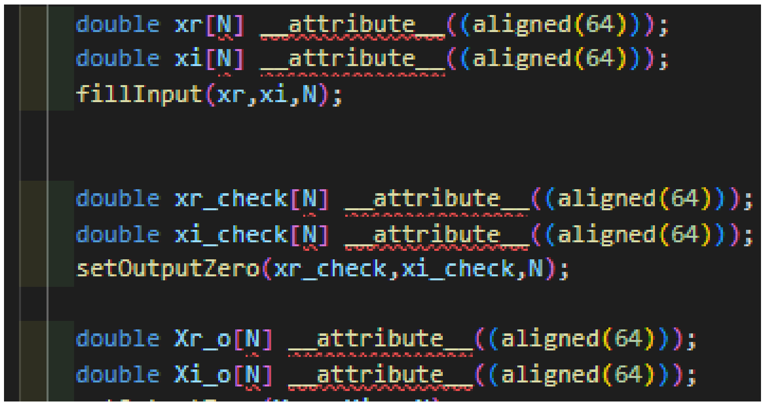
Figure 10: Vectorized code

We can conclude that starting from the -O2 optimization, we notice a decrease in the execution time of about 75%. Subsequent optimisations do not have a strong impact.

3.2 Vectorization and Alignment

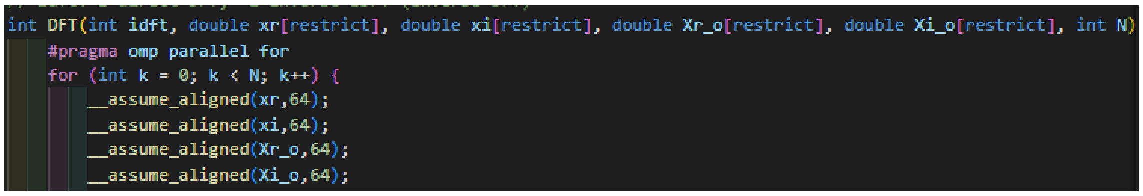
Alignment is the correct arrangement of data in memory (cache), so you can access it efficiently. Good alignment allows you to make the most of SIMD (Single Instruction, Multiple Data) instructions to perform operations on multiple data at the same time, this implies reduction of the number of clock cycles required to perform operations and, as a result, improve program performance.

We have made changes to the code shown in the following images.



```
double xr[N] __attribute__((aligned(64)));
double xi[N] __attribute__((aligned(64)));
fillInput(xr,xi,N);


double xr_check[N] __attribute__((aligned(64)));
double xi_check[N] __attribute__((aligned(64)));
setOutputZero(xr_check,xi_check,N);

double Xr_o[N] __attribute__((aligned(64)));
double Xi_o[N] __attribute__((aligned(64)));
```

Figure 11: Alignment Main



```
int DFT(int idft, double xr[restrict], double xi[restrict], double Xr_o[restrict], double Xi_o[restrict], int N)
    #pragma omp parallel for
    for (int k = 0; k < N; k++) {
        __assume_aligned(xr,64);
        __assume_aligned(xi,64);
        __assume_aligned(Xr_o,64);
        __assume_aligned(Xi_o,64);
```

Figure 12: Alignment DFT

We looked at performance through a series of benchmarks, the results of which were represented in the chart below.

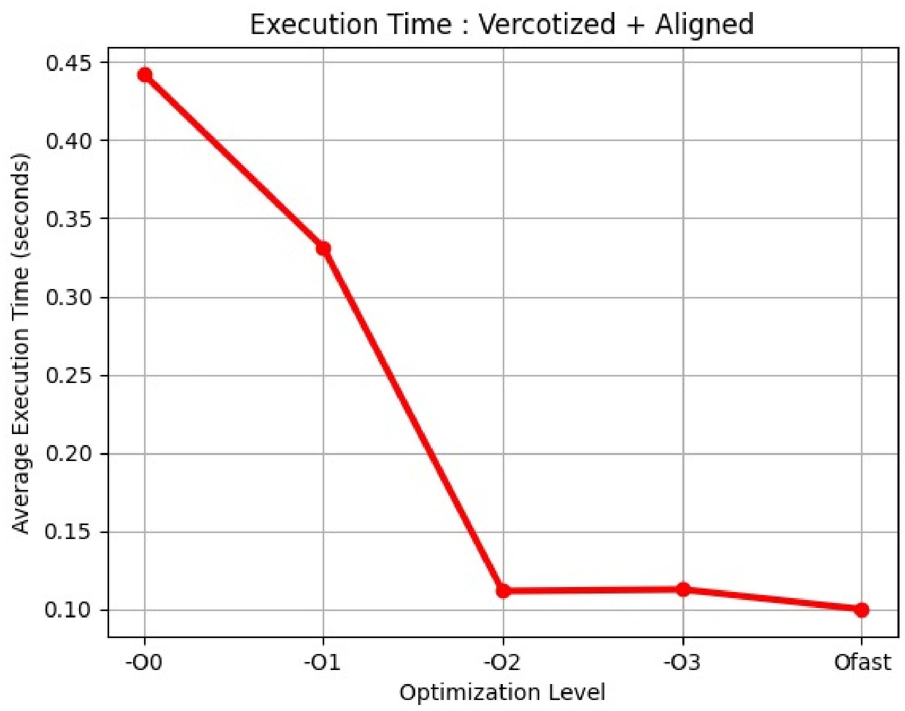


Figure 13: Vectorized Alignment code

In conclusion, the alignment of data has led to improvements in the execution time of our program. However, these advantages were not as substantial as those obtained in the previous chapter.

3.3 Vectorization, Alignment and Architecture

We specified the architecture that we want to use in order to improve the performances. To do this we simply added the flag ”-march=alderlake” in the command line.

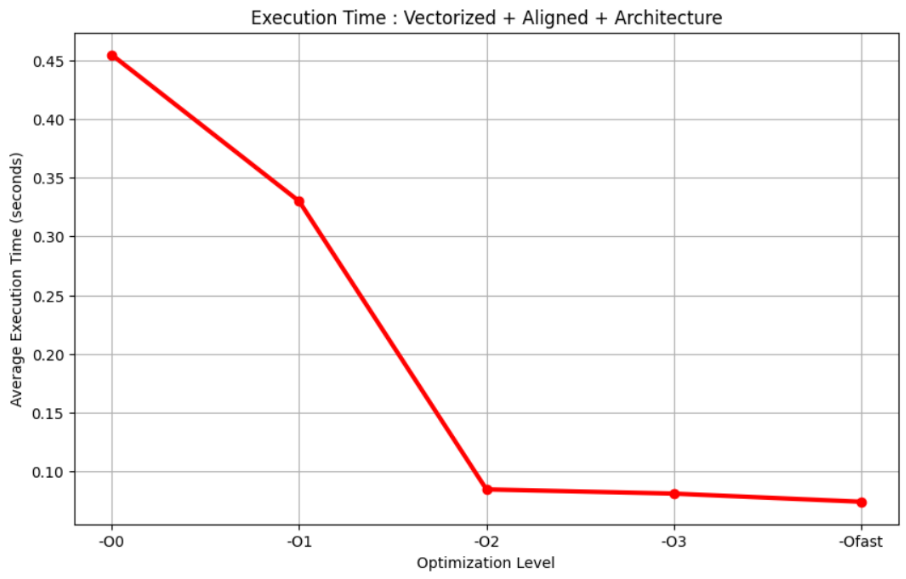


Figure 14: Vectorized Alignment Architecture code

In this case, specifying the architecture led to an increase in performance, especially with the optimization level -02 and later, where the execution time decreased under 0.10 seconds

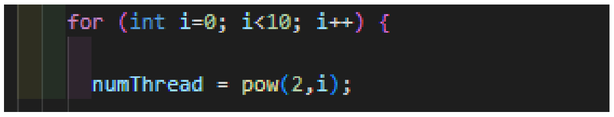
3.4 Vectorization, Alignment, Architecture and Threads

To further improve the performance of our program, we tried to run the program using threads.

We've made some changes to the code, which will be shown later, to allow effective distribution of the workload between the threads. Next, we run the program with different optimization levels, progressively increased the number of threads:

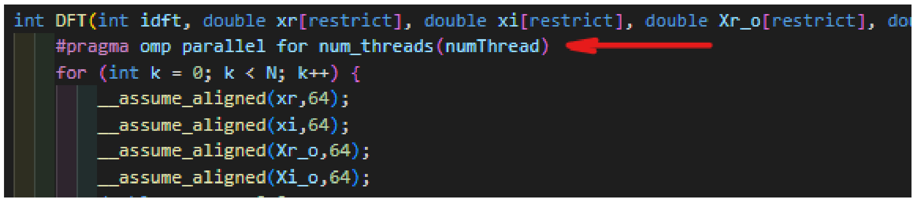
$$numThread = 2^i \text{ con } i = 1, \ldots, 10$$

and collected the performance data.



```
for (int i=0; i<10; i++) {

    numThread = pow(2,i);
```

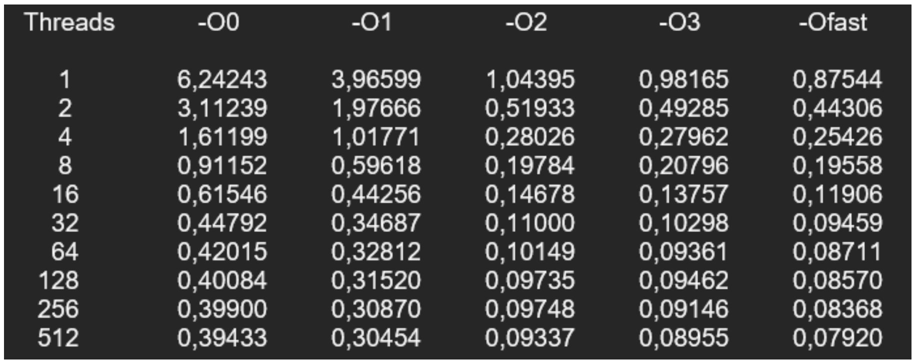
Figure 15: Different threads configuration



```
int DFT(int idft, double xr[restrict], double xi[restrict], double Xr_o[restrict], do
    #pragma omp parallel for num_threads(numThread)
    for (int k = 0; k < N; k++) {
        __assume_aligned(xr,64);
        __assume_aligned(xi,64);
        __assume_aligned(Xr_o,64);
        __assume_aligned(Xi_o,64);
```

Figure 16: Specify the number of threads

The results of these experiments have been summarized in the following table



Threads	-O0	-O1	-O2	-O3	-Ofast
1	6,24243	3,96599	1,04395	0,98165	0,87544
2	3,11239	1,97666	0,51933	0,49285	0,44306
4	1,61199	1,01771	0,28026	0,27962	0,25426
8	0,91152	0,59618	0,19784	0,20796	0,19558
16	0,61546	0,44256	0,14678	0,13757	0,11906
32	0,44792	0,34687	0,11000	0,10298	0,09459
64	0,42015	0,32812	0,10149	0,09361	0,08711
128	0,40084	0,31520	0,09735	0,09462	0,08570
256	0,39900	0,30870	0,09748	0,09146	0,08368
512	0,39433	0,30454	0,09337	0,08955	0,07920

Figure 17: Vectorized Alignment Architecture Threads tabled results

and displayed in a graph that highlights the relationship between the number of threads and the execution time.

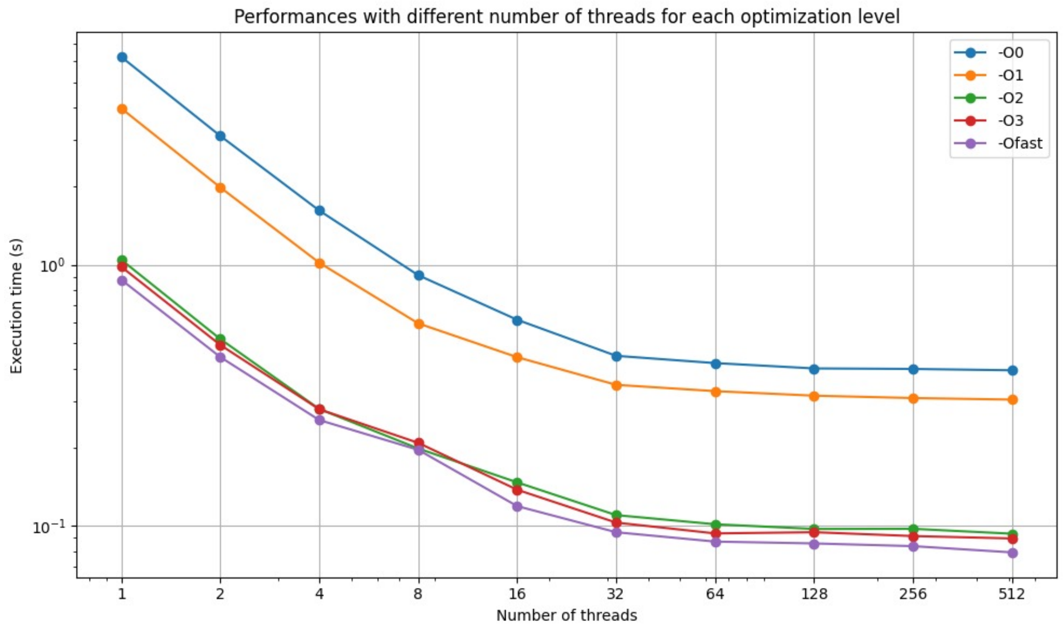
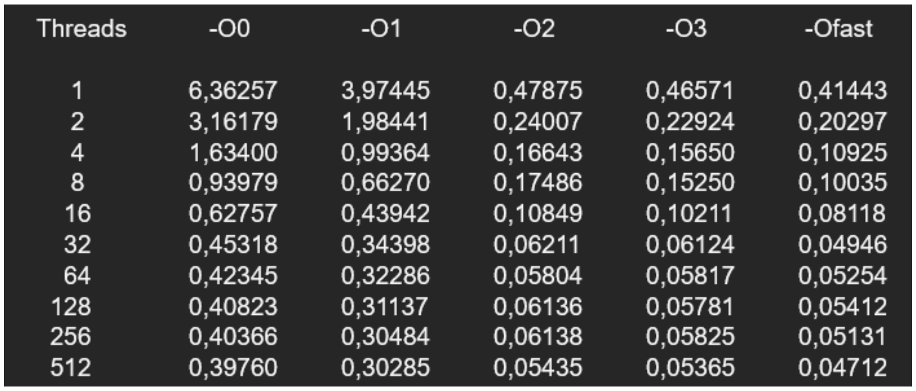


Figure 18: Vectorized Alignment Architecture Threads graphical results

3.5 Vectorization, Alignment, Threads and xHost

In this section we tried the compiler architecture using the *-xHost* flag rather than directly specifying the architecture as in section 3.3 and 3.4. Although this choice should not change the execution time, we opted for this choice to automatically optimize the code for the current machine architecture and compare it with the results obtained previously.
The results of these experiments have been summarized in the following table



Threads	-O0	-O1	-O2	-O3	-Ofast
1	6,36257	3,97445	0,47875	0,46571	0,41443
2	3,16179	1,98441	0,24007	0,22924	0,20297
4	1,63400	0,99364	0,16643	0,15650	0,10925
8	0,93979	0,66270	0,17486	0,15250	0,10035
16	0,62757	0,43942	0,10849	0,10211	0,08118
32	0,45318	0,34398	0,06211	0,06124	0,04946
64	0,42345	0,32286	0,05804	0,05817	0,05254
128	0,40823	0,31137	0,06136	0,05781	0,05412
256	0,40366	0,30484	0,06138	0,05825	0,05131
512	0,39760	0,30285	0,05435	0,05365	0,04712

Figure 19: Vectorization, Alignment, Threads and xHost tabled results

and displayed in a graph that highlights the relationship between the number of threads and the execution time.

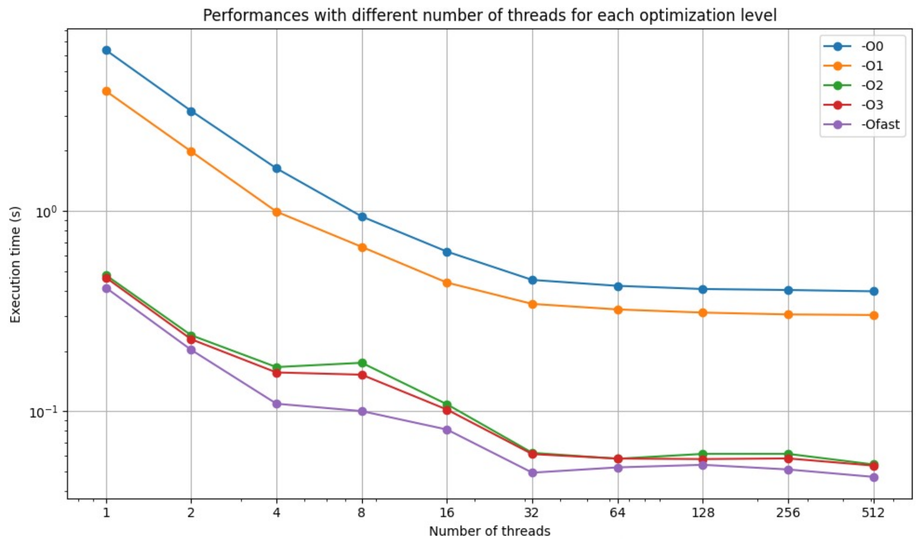


Figure 20: Vectorized Aligned Threads and xHost graphical results

The graph shows a decreasing trend respect to the execution time as threads numbers increase. It is evident that up to about 32 threads, the execution time is significantly reduced. This suggests that there is an optimal number of threads for which the program achieves maximum efficiency, and beyond that, the overhead of managing additional threads denies any increase in performance.

4 The Speedup

Speedup tells us how much a program runs faster, using many processors at once, instead of using just one. If the speedup number is bigger, it means the program is using all the processors well. By looking at the speedup, we can figure out the best way to run the program so that it's fast and doesn't waste resources.

The speedup can be computed in this way:

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

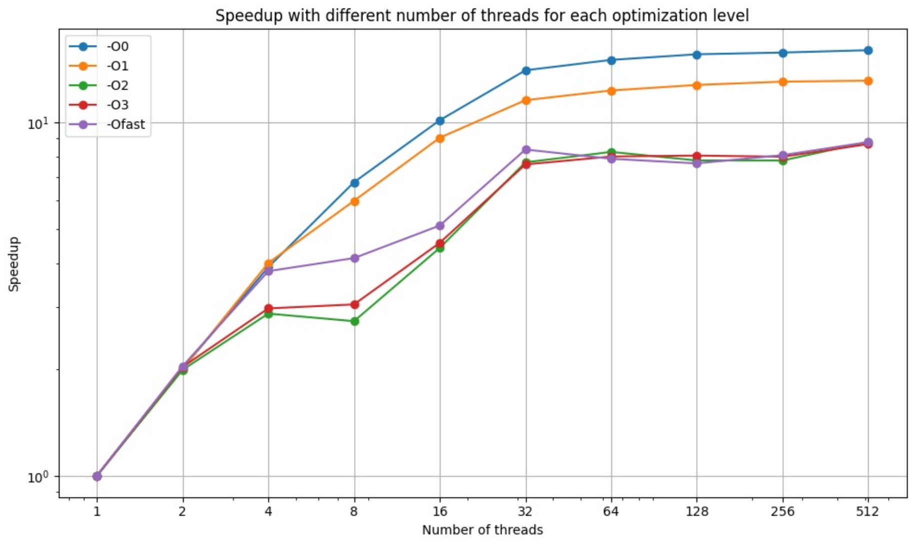


Figure 21: Speed Up

From the graph we can evaluate how the speedup increases in a semi-regular manner in all optimization levels up to 32 threads. After a certain number of threads, the speed-up stabilizes around a value and the increase in the number of threads does not lead to a substantial increase in speed-up. From the -O2 optimization level to rise, there is a decrease in speed-up when more threads are added due to communication and synchronization overload.

5 Efficiency respect to the number of threads

Efficiency is a metric that builds on top of speedup by adding awareness of the underlying hardware. [ref]

The Efficiency can be computed in this way:

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

We computed the efficiency in relation to the different levels of optimization and threads numbers using the data reported above.

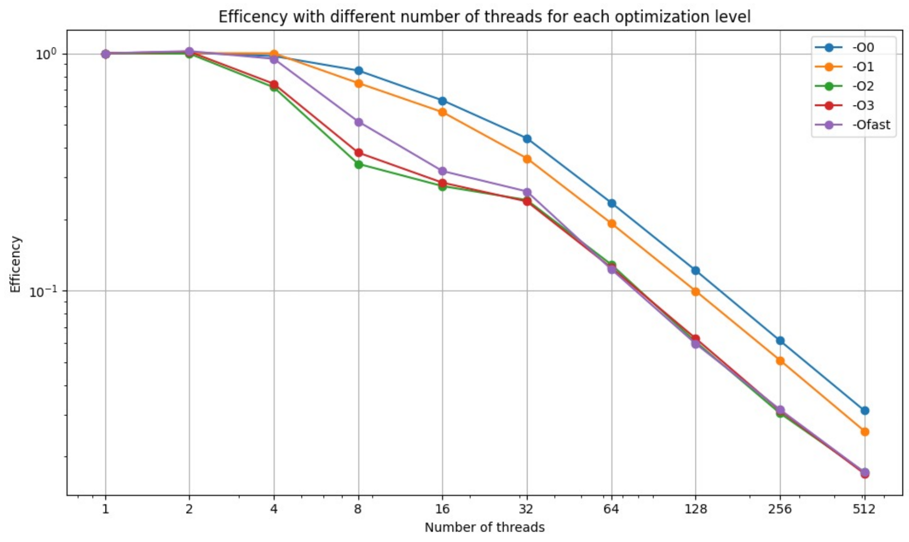


Figure 22: Efficiency

The efficiency of the program decreases significantly for each level of optimization when multiple threads are used. However, it should be noted that even with a suboptimal configuration (a number of threads below 16/32), the program effectively uses the available resources. Despite calculating efficiency based on the number of processors, we chose to calculate it with respect to the number of threads to understand how it decreases due to thread management. As observed in the graphs, as the number of threads increases, efficiency decreases.

6 Conclusions

In this assignment, our goal was to test different configurations and measure performance using vectorization and parallelization, in particular exploiting **OpenMP** and the Intel C compiler (**icc**). We evaluated: execution time, speed up and efficiency. All on different optimization levels.

The results showed that the best configuration used 32/64 threads and using higher optimization level allows the program to efficiently use the available resources with a lower number of threads. In particular, an higher number of threads does not proportionally improve performance. Additional threads no longer help to accelerate earnings, but our system has worse performance due to the workload in thread management, context changes, threads communication and synchronization.