UNIVERSITÀ DI PISA

Department of Information Engineering
MSc in Artificial Intelligence and Data Engineering

Implementation of the NSGA-II Fast Non-Dominated Sort using SIMD instructions

Symbolic and Evolutionary Artificial Intelligence Project

Irene Catini
Ludovica Cocchella

Academic Year 2022/2023

Contents

1 Introduction

In this report, we present a detailed introduction to our project focused on implementing the NSGA-II Fast Non-Dominated Sort algorithm using SIMD (Single Instruction, Multiple Data) instructions. The main goal of our work is to significantly enhance the algorithm's performance through code vectorization, an approach that leverages SIMD instructions to execute parallel operations on multiple data.

The developmental path we will undertake includes several key steps. Firstly, we will begin by writing the initial algorithm in MATLAB, ensuring it is both efficient and clean. Subsequently, we will translate the MATLAB script into C++, adapting it to the context of the new programming language. Once this phase is complete, we will focus on vectorizing the code to fully utilize the available SIMD instructions, aiming to further optimize performance.

Finally, an essential part of our project will be dedicated to comparing the executions of the algorithm with and without vectorization. This will allow us to accurately assess the impact of our optimization on the overall execution speed of the algorithm. Through this in-depth analysis, we aim to demonstrate the effectiveness of our vectorization strategy in the specific context of the NSGA-II Fast Non-Dominated Sort algorithm.

2 NSGA-II

The NSGA-II, which stands for Non-dominated Sorting Genetic Algorithm II, represents a highly effective evolutionary algorithm for solving multi-objective optimization problems. This approach is grounded in principles inspired by natural selection and evolves through iterations, creating a population of candidate solutions that progressively move toward the optimal solution of the multi-objective optimization problem at hand.

2.1 NSGA-II Vs NSGA

NSGA is a second version of the based algorithm NSGA witch has the drawback of require additional parameters like the sigma value for the nitching mechanism. NSGA laid the foundation for multi-objective evolutionary algorithms, NSGA-II's improvements make it a popular and widely used choice in many applications.

Some considerations that might influence the decision to use NSGA-II over NSGA:

- Efficiency and Convergence Speed: NSGA-II generally exhibits better convergence and efficiency compared to NSGA. NSGA-II introduces improvements, such as fast non-dominated sorting and a diversity-preserving mechanism called crowding distance, which enhances its ability to converge towards the Pareto front more effectively.

- Handling Constraint Violations: NSGA-II tends to handle constraint violations more effectively. It employs a mechanism to penalize solutions violating constraints, which can be beneficial in problems where constraints play a significant role.

- Diversity Maintenance: NSGA-II incorporates mechanisms, like crowding distance, that contribute to maintaining diversity in the population. This is crucial for exploring a broader solution space and avoiding premature convergence to a specific region of the Pareto front.

- Scalability: NSGA-II is often preferred for large-scale optimization problems due to its efficiency and ability to handle a large number of candidate solutions in the search space.

- Adaptability and Robustness: NSGA-II's design enhancements make it more adaptable and robust across a variety of optimization problems. It often performs well without requiring extensive tuning for specific problem instances.

- Additional parameters: NSGA-II require only the parameters of a typycal genetic algorithm (pop_size, n_generations, prob_cross, prob_mut).

2.2 The concept of dominance

The key concept of NSGA-II is to identify the Pareto front, representing a set of non-dominated optimal solutions, or Pareto set. A solution is considered non-dominated when it achieves a suitable compromise across all objectives without degrading any of them. Pareto dominance involves comparing each solution x with every other solution in the population until it is dominated by one of them. If no solution dominates it, the solution x is considered non-dominated and is selected by NSGA-II to be part of the Pareto front. The definition of Pareto dominance is given by:
$x' \in$ Population is a Pareto minimum if there is no $x \in$ Population such that
$\forall i$, $f_i(x') \geq f_i(x)$ and
$\exists j$ such that $f_j(x') > f_j(x)$
where f_i represents the objectives to minimize.

2.3 The algorithm

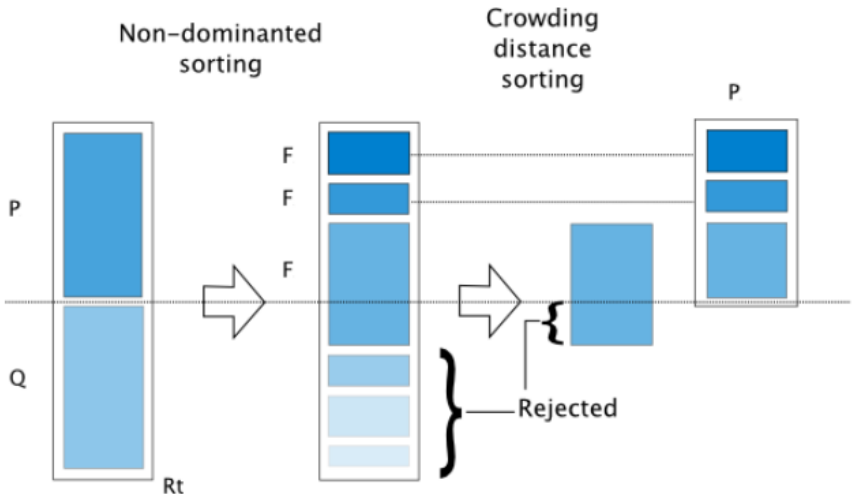
NSGA-II is distinguished by its implementation of an elitist strategy and its lack of reliance on an external archive. The elitist strategy ensures that the best-performing solutions, as determined by non-dominated sorting, directly carry over to the next generation without undergoing genetic operations. This mechanism safeguards the retention of superior individuals, contributing to the consistent improvement of the population over successive iterations. Moreover, NSGA-II's efficient non-dominated sorting and crowding distance mechanisms eliminate the need for an external archive. Unlike some other multi-objective optimization algorithms, NSGA-II adeptly manages the Pareto front, preserving diversity and exploring a wide range of non-dominated solutions without the necessity of an external storage mechanism. This streamlined approach enhances the algorithm's effectiveness in solving complex optimization problems by maintaining a balance between exploration and exploitation in the evolving population.

To be applied effectively, NSGA-II requires specifying key elements in its implementation:

- the representation of individuals used to create a population,
- a fitness function tailored to each objective for evaluating candidate solutions,

- the design of crossover and mutation operators aligned with the individual's representation.

The process begins with the random creation of the initial population P_0 of individuals encoded with a specific representation. Subsequently, a child population Q_0 is generated from the parent population P_0 using genetic operators such as crossover and mutation. In particular crossover allows to recombine two parents into two new offsprings, in fact, the idea is to use genetic material from the parents, and to mix them together or to perform a weighted combination of the two, and mutation probabilistically mutates some of the genes and changes their values using another value at random, provided that it is compliant with the universe of the specific gene. The two populations (P_0 and Q_0) merge creating R_t, and a subset of individuals is selected based on the dominance principle. The image below show the generation process of the next population:



The process begins by creating all the fronts in the population R_t. Then, it takes one front at a time and adds individuals to the next population (P_{t+1}). It starts with the top-ranked solutions (F1), which are the best ones and not dominated by others. After that, it includes the second-ranked solutions, and so on, until P_{t+1} reaches the chosen population size. Sometimes, there's a need to pick solutions within a specific group. To handle this, we use an extra method to select the best solutions within that group. NSGA-II suggests using the Crowding Distance procedure (explained in the section 2.3.2) for this selection. This approach is aimed at getting the most comprehensive set of efficient solutions.

2.3.1 Main loop pseudocode

```
Randomly generate the initial population P0
Q0= generate_offsprings(P0)
For t=1:max_gen
    R1 = Pt U Qt
    F = fast_non_dominated_sort(R)
    Set i= 1
    Set Pt+1 = 0
    While |Pt+1|+|Fi| ≤ pop_size
        Pt+1 = Pt+1 U Fi
        i = i + 1
    endWhile

    compute_crowding_distance (Fi)
    Set k = pop_size - |Pt+1|
    Pt+1 = Pt+1 U best_k_elements_in(Fi,k)
    Qt+1= generate_offsprings(Pt+1)
endFor
```

2.4 Crowding Distance

The crowding distance is a concept commonly used in the field of multi-objective optimization, particularly in the context of genetic algorithms and evolutionary algorithms. It is a measure designed to maintain diversity in a population of solutions when solving multi-objective optimization problems.

In a multi-objective optimization problem, there are typically multiple conflicting objectives that need to be optimized simultaneously. The goal is to find a set of solutions that represent a trade-off between these objectives, known as the Pareto front.

The crowding distance is used to quantify the distribution or "crowding" of solutions in the objective space. It is calculated for each solution within the population and is based on the distances between the solution and its neighboring solutions. The idea is to favor solutions that are in less crowded regions of the objective space, promoting diversity among the solutions.

Here's an explanation of how crowding distance is calculated:

1. For each objective, sort the solutions in the population based on their values for that objective.

2. Calculate the distance between adjacent solutions for each objective separately.

3. Sum the distances across all objectives to obtain the crowding distance for each solution.

Solutions with higher crowding distances are considered less crowded and are often given preference in the selection process to maintain diversity in the population. This helps prevent premature convergence to a subset of solutions in the Pareto front and encourages a more comprehensive exploration of the trade-off space.

2.4.1 Pseudocode

```
crowding_distance_assignment(I)
{
    l = |I|
    For each i
        I[i]distance = 0
    endFor

    For each objective m
        I = sort(I,m)
        I[i]distance = I[l]distance = inf

        For i = 2 to (l-1)
            I[i]distance = I[i]distance + (I[i-1].m - ...
               I[i+1].m)/(max_fm - min_fm)
        endFor
    endFor
}
```

2.5 Fast Non Dominated sorting

The fast non-dominated sort in NSGA-II is an efficient method for performing the non-dominated sorting of solutions in each generation. The primary purpose of this sorting is to organize the solutions into different Pareto fronts, where each front contains non-dominated solutions (those that are not worse than any other solution in the front in all objectives). In this part of the algorithm there is the use of dominance concept and Rank Assignment procedure where solutions are assigned to a rank based on their dominance relationships. The solutions in the first Pareto front (non-dominated solutions) are assigned a rank of 1.

2.5.1 Pseudocode

```
fast_non_dominated_sort(P)
{
    For each p in P
        Sp = 0
        np = 0

        For each q in P
            If (p < q)
                Sp = Sp U q
            elseIf (q < p)
                np = np + 1
            endIf
        endFor

        If np == 0
            prank = 1
            F1 = F1 U p
        endIf
    endFor

    i = 1
    While Fi not empty
        Q = 0
        For each p in Fi
            For each q in Sp
                nq = nq - 1
                If nq == 0
                    qrenk = i + 1
                    Q = Q U q
                endIf
            endFor
        endFor

        i = i + 1
        Fi = Q
    endWhile
}
```

3 SIMD instructions

3.1 Parallel Computing Approaches

Over the past century, processor performance was traditionally boosted by escalating clock speeds; however, despite its effectiveness for several decades, this method began to exhibit its limitations as early as the '90s.

The approach adopted to consistently produce higher-performing processors involves enhancing the parallelism of operations, given the following two observations

- Many data can be processed simultaneously without requiring reciprocal correspondence (Data level parallelism)
- Many operations, even when sequential in the code, can be executed in parallel, as they are independent of each other.

Michael J. Flynn created the first classification systems for parallel computers and programs, called as Flynn's taxonomy. Flynn grouped programs and computers on basis of operating using a single set or multiple sets of instructions, and whether those instructions were using a single set of data or multiple sets of data. The categories are the following:

- *Single instruction single data* (SISD): a single processing unit executes a single stream of operations, each of which operates on one piece of data at a time. Despite its apparent sequential nature, a degree of parallelism is achievable through the implementation of pipelining techniques.
- *Single instruction, multiple data* (SIMD): the same instruction is executed by multiple processors on different streams of data. In this scenario, each processor has its own memory, resulting in multiple memories. However, there is a single entity responsible for fetching instructions. Vectorized processors and multimedia extensions to standard ISAs fall into this category.
- *Multiple instruction, single data* (MISD): Multiple processors, each with its own memory (registers), will have their own instruction stream. These instructions will be executed on the same data stream. As of today, machines with this architecture have not yet been constructed for commercialization.
- *Multiple instruction, multiple data* (MIMD): these systems can simultaneously execute multiple different instructions on multiple sets of data. Typically, they consist of a collection of independent computing units, each executing its instructions independently of the others. Examples falling into this category include multi-core processors and computer clusters.

3.2 Vector processor

Vector processors fetch a set of data from memory, place them into vector registers ranging from 128 to 512 bits, perform operations on the data, and then store them back in memory. One advantage of this approach lies in the load operation latency: memory access is performed only once for the entire vector, rather than N times for each of the data it operates on.

The operations that can be performed among the elements of these registers range from simple additions and subtractions to comparisons, permutations, and bit shifts. Their efficiency is further enhanced by the ability to reinterpret the bits of the registers in different ways, depending on the chosen operation: a 128-bit register can hold 2 operands of 64 bits, 4 of 32 bits, 8 of 16 bits, and so forth.

3.3 Advanced Vector Extension

Advanced Vector Extensions are SIMD extensions to the x86 instruction set architecture for microprocessors from Intel and Advanced Micro Devices (AMD). Starting from 2011, both Intel and AMD began incorporating support for vectorization (AVX) within their processors, featuring 128-bit vectorized registers. In 2013, AVX2 was introduced, featuring 256-bit registers along with additional instructions that enhance the efficiency of loading non-contiguous data from memory. Currently, the latest extension is AVX-512, which supports 512-bit registers, allowing simultaneous operations on 8 sets of 64-bit data. AVX introduces 16 registers of 256 bits each, where the lower 128 bits can be referred to using symbolic names XMM0-XMM15. Many vectorized operations do not require specific registers. Alongside these, there is a status register, MXCSR, which facilitates error detection and allows for the modification of certain options during instruction execution. Almost all instructions introduced by AVX are in a three-operand format, with two source operands and one destination operand, of the following type:

```
InstrMnemonic DesOp, SrcOp1, SrcOp2
```

The same registers allow performing operations on real scalars, either in single or double precision.

AVX-512 increases the number of vectorized registers to 32, denoted as ZMM0-ZMM31, whose lower parts can still be referenced using YMMxx and XMMxx.

3.4 Fundamentals of AVX Programming

3.4.1 Data Types

A few intrinsics accept traditional data types like ints or floats, but most operate on data types that are specific to AVX and AVX2. There are six main vector types and Table 1 lists each of them.

Data Type	Description
__m128	128-bit vector containing 4 floats
__m128d	128-bit vector containing 2 doubles
__m128i	128-bit vector containing integers
__m256	256-bit vector containing 8 floats
__m256d	256-bit vector containing 4 doubles
__m256i	256-bit vector containing integers

Table 1: AVX/AVX2 Data Types

Each type starts with two underscores, an `m`, and the width of the vector in bits. AVX512 supports 512-bit vector types that start with `_m512`, but AVX/AVX2 vectors don't go beyond 256 bits.

If a vector type ends in `d`, it contains doubles, and if it doesn't have a suffix, it contains floats. It might look like `_m128i` and `_m256i` vectors must contain ints, but this isn't the case. An integer vector type can contain any type of integer, from chars to shorts to unsigned long longs. That is, an `_m256i` may contain 32 chars, 16 shorts, 8 ints, or 4 longs. These integers can be signed or unsigned.

3.4.2 Building AVX Applications

To build an application that uses AVX intrinsics, you don't need to link any libraries. However, you need to include the `immintrin.h` header file. This header includes other headers that map AVX/AVX2 functions to instructions.

The code in `hello_avx.c` illustrates what a basic AVX application looks like:

```
#include <immintrin.h>
#include <stdio.h>

int main() {
```

```
  /* Initialize the two argument vectors */
  __m256 evens = _mm256_set_ps(2.0, 4.0, 6.0, 8.0, 10.0, 12.0, 14.0, 16.0);
  __m256 odds = _mm256_set_ps(1.0, 3.0, 5.0, 7.0, 9.0, 11.0, 13.0, 15.0);

  /* Compute the difference between the two vectors */
  __m256 result = _mm256_sub_ps(evens, odds);

  /* Display the elements of the result vector */
  float* f = (float*)&result;
  printf("%f %f %f %f %f %f %f %f\n",
    f[0], f[1], f[2], f[3], f[4], f[5], f[6], f[7]);

  return 0;
}
```

To compile the application, you need to inform the compiler that the architecture supports AVX. The flag depends on the compiler, and gcc requires the `-mavx` flag. Therefore, the source file `hello_avx.c` can be compiled with the following command:

```
gcc -mavx -o hello_avx hello_avx.c
```

In this example, all the functions start with `_mm256` and end with `_ps`, so it's clear that all the operations involve 256-bit vectors containing floats. It's also clear that each element in the resulting vector equals 1.0. If you run the application, you'll see that this is the case.

4 MATLAB implementation

Within this chapter, the implementations carried out have been documented, following two different paths. This divergence arose as issues surfaced after the initial implementation, necessitating resolution. Section 4.2 will address the first (old) implementation, while Section 4.3 will cover the second (final) implementation.

4.1 Creation of Population

The initial part of the implemented algorithm is dedicated to creating an initial set of random points. Specifically, two parameters need to be passed to the main function for their definition: the population size (*pop_size*) and the number of objectives to be considered (*obj_n*). The function can also be called without explicitly defining the parameters, in which case default values will be used. The default parameters are set as follows: *pop_size = 1000* and *obj_n = 2*. In the case where the number of objectives is set to 2, the implemented code will be capable of visualizing the generated points on a plane, allowing for a visualization of their distribution. It is important to note that the initially generated points are rotated in such a way as to result in a final population that forms an inclined ellipsoid.

4.1.1 Code

Listing 1: Create population

```
% Population Generation
% Set default values for pop_size and obj_n if not provided
if nargin < 2
   obj_n = 2;
   if nargin < 1
      pop_size = 1000;
   end
end

% Determine whether to create plots based on the number of objectives
do_the_plot = true;
if obj_n > 2
    do_the_plot = false;
end

% Generate random points in a obj_n space
X = randn(pop_size, obj_n);

% Plot the points in the first subplot if plotting is enabled
if do_the_plot
```

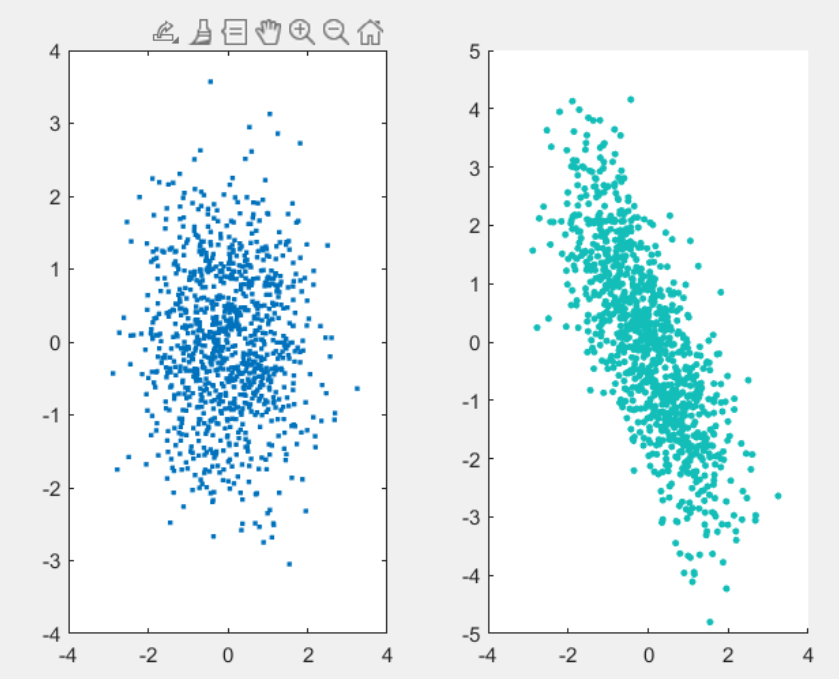
```
    subplot(1, 2, 1);
    plot(X(:, 1), X(:, 2), '.');
end

% Sort the points based on the values in the first column
[¬, is] = sort(X(:, 1)); % 'is' contains the indices of the ...
    elements once
                             % sorted based on the first column
X = X(is, :);

% Create a new set of points with a third dimension (Z) added
Z = linspace(2, -2, pop_size);
Y = X;
Y(:, 2) = Y(:, 2) + Z';

% Plot the new set of points in the second subplot if plotting is ...
    enabled
if do_the_plot
    subplot(1, 2, 2);
    plot(Y(:, 1), Y(:, 2), '.');
end
```

An example of the created population is shown on the image below:



4.2 Old version

Below the code written to implement the NSGA-II algorithm. It is only a firts experiment that it was updated in the second and final script described in the section 4.3.

4.2.1 Code

Listing 2: Old implementation

```
R = zeros(1, pop_size); % Vector containing the rank of the front ...
    for each point
F = [];
allF = []; % Keeps track of indices that have already been ...
    analyzed and saved
current_front_index = 1; % Index of the front to consider in the ...
    while loop

while size(allF) < pop_size
    for p = 1:size(P, 1) % p is an index relative to the matrix P
        if ismember(p, allF)
            continue;
        end
        n_p = 0; % Domination counter of p

        for q = 1:size(P, 1)
            if ismember(q, allF)
                continue;
            end
            if and(all(P(q, :) ≤ P(p, :)), any(P(q, :) < P(p, :)))
                n_p = n_p + 1; % p is dominated by q
            end
        end

        if n_p == 0
            R(p) = current_front_index;
            F = [F, p]; % Add p to the list of points in the ...
                current front
        end
    end
    allF = [allF, F]; % Add the list of points belonging to front F
                      % to the list of all fronts
    F = [];
    current_front_index = current_front_index + 1;
end
```

Here's a step-by-step description:

1. Initialization:

    - **R** is initialized as a vector of zeros with a length equal to pop_size. It will contain the rank of each point.
    - **F** is initialized as an empty list to keep track of points belonging to the current front being analyzed.
    - **allF** is initialized as an empty list to maintain the indices that have already been analyzed and saved.
    - **current_front_index** is initialized to 1, representing the index of the front to be considered in the while loop.

2. While Loop:

    - The while loop continues until the size of allF reaches pop_size because this means that all the points have been analyzed.
    - Inside the loop, a nested for loop iterates through each point (p) in the population P. The current iteration is skipped if the point p has already been inserted inside a front.

3. Domination Counter:

    - Another nested loop iterates through all other points (q) in P that haven't been analyzed.
    - For each q, it checks if p is dominated by q based on the objective values of the points.

4. Front Assignment:

    - If n_p remains 0 after the nested loop, it means that p is not dominated by any other point in the set. In this case, p is assigned the current front index, and its index is added to the list F representing the points in the current front.

5. Update and Next Front:

    - The list **F** is appended to the list **allF**.
    - **F** is reset to an empty list for the next iteration.
    - The **current_front_index** is incremented to move on to the next front.

4.2.2 Emerged problems

In the above-described code, there are potential issues related to the management of the vectors F and allF, as well as the use of the Matlab function ismember. The first observation made is about the length of the two vectors. They are initialized as empty vectors, and their size changes with each iteration of the algorithm. However, this dynamic management approach makes the execution costly. Consider what happens in the background (in terms of memory allocation) when x changes its size every iteration: At each iteration Matlab needs to find a free memory space to host the new size of x. If you are lucky, there is enough free space right after x so all that happens is a change to the amount of memory allocated to x and writing the new value at the right spot. However, if there is not enough free space just after x, Matlab has to find a new spot for all the elements already in x, allocate this new space for x, copy all values already in x to the new spot, and free the old spot x used. This allocate-copy-free operations happening in the background can be extremely time consuming, especially when x is large. In this type of project, F is a vector that could potentially become large since it may accumulate all the points, or the majority of them, if they fall into a single front. On the other hand, allF is certainly of large dimension as it will contain all the population's points at the end of the algorithm. Regarding the function ismember, after conducting various studies to understand whether it was the one causing the slowdown, as theory suggests, it should not be the bottleneck of the code. To investigate this, we used the "profile on" command in MATLAB, enabling code profiling, which provides timing information for each function and, in particular, for each individual line. From the results obtained, it appears that ismember slows down the code execution.

Below are the analyses conducted using Matlab and the results in terms of algorithm execution time, considering a problem with two objective functions and a population of 1000 individuals.

Function Name	Calls	Total Time (s)	Self Time* (s)	Total Time Plot (dark band = self time)
NSGAII_matlab_old_version	1	112.852	15.866	
ismember	14637015	96.737	32.323	
ismember>ismemberR2012a	14637014	64.413	34.570	
ismember>ismemberBuiltinTypes	14637000	29.841	29.841	

Line Number	Code	Calls	Total Time (s)	% Time
63	`if ismember(q, allF)`	14600000	108.563	96.2%
66	`if and(all(P(q,:) <= P(p,:)),any(P(q,:) < P(p,:)))`	9957076	2.620	2.3%

As we can observe, the total execution time is comprised of more than 96% due to the execution of the isMember function, which is performed a considerable number of times even with a small initial population. In contrast, the dominance-related part has a much lower impact in terms of time (just over 2%). Consequently, we wondered if it was possible to overcome the use of this function, considering that the bottleneck should be the repeated dominance test.

4.3 Final version

This implementation was developed as an alternative solution to the previous one in order to overcome all the problems shown in the previous section. Below the code used and its description are reported:

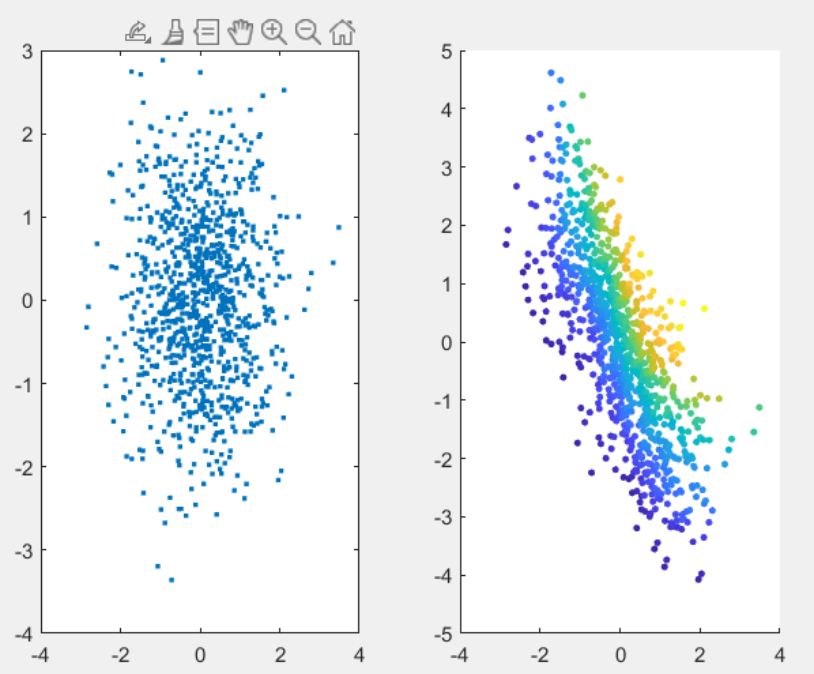
4.3.1 Code

```
R = zeros(1, pop_size); % Vector containing the rank of the front ...
    for each point
current_front_index = 1; % Index of the front to consider in the ...
    while loop
F = false(pop_size, 1);
already_assigned_to_a_front = false(pop_size, 1);
while sum(already_assigned_to_a_front) < pop_size - 1
    for p = 1:size(P, 1) % p is an index relative to the matrix
        if already_assigned_to_a_front(p)
            continue;
        end
        n_p = 0; % Domination counter of point p
        for q = 1:size(P, 1)
            if already_assigned_to_a_front(q)
                continue;
            end
            if and(all(P(q, :) ≤ P(p, :)), any(P(q, :) < P(p, :)))
                n_p = n_p + 1; % p is dominated by q
            end
        end
        if n_p == 0
            R(p) = current_front_index;
            F(p) = true;
        end
    end
    already_assigned_to_a_front(F) = true;
    current_front_index = current_front_index + 1;
end
```

The solution we implemented to replace the ismember function was to create a boolean vector called "already_assigned_to_a_front" with the same length as the number of solutions. It will contain True if the corresponding individual in the population has been previously assigned to a front, or False if it has not been assigned yet. For this reason, it should have a relative position for each individual and it is initially set to false. For the management of this new array, we have modified the declaration and usage of F. In the old version, it was comprised of an array of integers to which the index corresponding to the individual being inserted into the current front was added. In the final version, it is instead declared as a fixed-size boolean array (pop_size) initialized to false. Instead of adding integers at the end, we set the position corresponding to the individual we are analyzing to true if it belongs to the current front. The size is fixed at pop_size because in the worst case, all points in the population could belong to the same front. Once the current front is completed, we will use the F vector to update the "already_assigned_to_a_front" vector.

4.4 Results

In order to display the obtained results, we plotted a graph where individuals in the population were colored differently based on their belonging front. Below, we present the graph obtained by considering a problem with 1000 individuals and 2 objectives. The table shows the number of fronts found for different population sizes and numbers of objectives.



pop_size	n_obj	n_fronts
1000	2	35
1000	8	3
1000	12	2
5000	2	83
5000	8	4
5000	12	3
10.000	2	109
10.000	8	5
10.000	12	3

Below, we show the execution times of the old and final versions to highlight the significant time savings.

- Old Version: 208s
- Final version: 4s

5 C++ implementation

After writing the algorithm in MATLAB code, we translated it into C++ with the goal of exploring the possibility of vectorizing the operations contained within it. In the following paragraphs, we present the experiments conducted similarly to those in MATLAB. In the initial version, we can observe the use of the 'find' function (equivalent to MATLAB's 'ismember'), while in the final version, this function has been replaced as explained above.

In order to test this code we used a virtual machine with AVX-512. Below we reported the command used to check the relative resuts.

```
clang++ -march=skylake-avx512 -O3 -ftree-vectorize
-Rpass=loop-vectorize
-Rpass-missed=loop-vectorize-Rpass-missed=loop-vectorize
-Rpass-analysis=loop-vectorize
-o test algorithm_non_vectorized_new_version.cpp
```

When compiling C++ code using the Clang compiler, it's essential to leverage optimization and vectorization to enhance performance. Below are the flags used in the compilation command, each serving a specific purpose:

- `-march=skylake-avx512`: Specifies the target processor architecture as Skylake with AVX-512 extensions. This enables the compiler to generate code optimized for advanced vector operations, crucial for scientific computations.
- `-O3`: Activates the highest level of optimization offered by the compiler. The `-O3` flag applies various optimization transformations that can significantly boost code performance, although with potentially more complex generated code.
- `-ftree-vectorize`: Enables tree vectorization, a technique that enhances parallelization of operations on vectors of data. Vectorization is crucial for exploiting modern processors' capabilities.
- `-Rpass=loop-vectorize`: Turns on reporting during the loop vectorization phase. The compiler generates reports detailing the transformations performed and providing insights into loop vectorization.
- `-Rpass-missed=loop-vectorize`: Enables reports on missed loop vectorization opportunities. This flag is helpful for identifying portions of code that could benefit from vectorization but were not optimized for some reason.

- `-Rpass-analysis=loop-vectorize`: Activates analysis reports during loop vectorization. These reports offer additional details on the compiler's decision-making process regarding whether to vectorize a particular loop.
- `-o test`: Specifies the output executable file name as "test." The compiler will generate an executable based on the compiled code.

5.1 Old version

5.2 Code

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <fstream>
#include <sstream>
#include <string>

// Definition of a function to determine if one point dominates ...
    another point
// Returns true if point q dominates point p
bool dominates(const std::vector<double>& point_q, const ...
    std::vector<double>& point_p) {

    int count = 0;
    int strict_count = 0;

    for(int i = 0; i < point_p.size(); i++){
        if(point_q[i] ≤ point_p[i]){
            count++;
            if(point_q[i] < point_p[i]){
                strict_count++;
            }
        }
    }
    if(count == point_p.size() && strict_count > 0)
        return true;
    return false;
}

// Function to split a string based on a delimiter
std::vector<double> split(const std::string &s, char delimiter) {
    std::vector<double> tokens;
    std::stringstream ss(s);
    std::string token;
```

```
    while (std::getline(ss, token, delimiter)) {
        tokens.push_back(std::stod(token));
    }
    return tokens;
}

int main() {

    std::ifstream file("../output_8obj.csv");
    std::vector<std::vector<double>> matrix;

    if (file.is_open()) {
        std::string line;
        while (std::getline(file, line)) {
            std::vector<double> row = split(line, ',');
            matrix.push_back(row);
        }
        file.close();
    } else {
        std::cout << "Unable to open the file." << std::endl;
    }

    // Definition of variables and initialization
    const int pop_size = matrix.size();
    std::vector<std::vector<double>> P = matrix;
    std::vector<int> R(pop_size, 0); // vector containing the rank ...
       of the front for each p
    std::vector<int> F;              // F contains the indices of ...
       points belonging to the current front
    std::vector<int> allF;           // allF contains the indices ...
       of ALL points that have already been assigned a front
    int current_front_index = 1;     // index of the front to ...
       consider in the while loop

    while (allF.size() < pop_size) {
        for (int p = 0; p < P.size(); ++p) {
            /*std::find searches for a specific value within a ...
               specified range and returns an iterator
            pointing to the found element or the end() iterator of ...
               the container if the element is not present.*/
            if (std::find(allF.begin(), allF.end(), p) != ...
               allF.end()) {
                continue;
            }
            int n_p = 0; // domination counter of p
            for (int q = 0; q < P.size(); ++q) {
```

```
                if (std::find(allF.begin(), allF.end(), q) != ...
                    allF.end()) {
                    continue;
                }
                if (dominates(P[q], P[p])) {
                    n_p = n_p + 1; // p is dominated by q
                }
            }
            if (n_p == 0) {
                R[p] = current_front_index;
                F.push_back(p); // add p to the list of points in ...
                    the current front
            }
        }
        allF.insert(allF.end(), F.begin(), F.end()); // add the ...
            list of points belonging to front F to the list of all ...
            fronts
        F.clear();
        current_front_index = current_front_index + 1;
    }

    std::cout << "Fronts found: " << current_front_index - 1 << ...
        std::endl;
    return 0;
}
```

After compiling the above code with the command shown at the beginning of Chapter 5, the vectorization report is displayed. Analyzing this report, we were able to verify that the processor's auto-vectorization performs well concerning the "dominates" function, which includes the necessary loops for determining the dominance of one point over another. However, it does not provide information regarding the main function where the population scanning loops are present. Since we find the "find" function inside, we attempted to see if manual vectorization could yield better results compared to the case with auto-vectorization. The following image presents the results obtained without manual vectorization.

```
ubuntu@coco-tesista:~/SymbolicProject/cpp$ time ./non_vect
Fronti trovati: 10

real    0m0.007s
user    0m0.007s
sys     0m0.000s
```

5.3 Manual vectorization

Below the implemented code, to test whether manual vectorization of the 'find' function improves the obtained time results, is showed. The above-mentioned code has been modified by replacing the call to the "find" function with the new function provided below.

5.3.1 Code

```
#include <immintrin.h>
#include <vector>
// The function searches for the integer 'val' within the vector ...
   'vec' and returns a boolean
bool findValueInVector(int val, std::vector<int> vec, int vecSize) {
    // Load the integer 'val' into an AVX register
    __m256i target = _mm256_set1_epi32(val);

    // Iterate through the vector in blocks of 8 elements (AVX ...
       register size)
    for (int i = 0; i < vecSize; i += 8) {
        // Load 8 consecutive elements of the vector into the AVX ...
           register
        __m256i chunk = _mm256_loadu_si256(reinterpret_cast<const ...
           __m256i*>(&vec[i]));

        // Compare vector elements with 'val' and obtain a ...
           comparison mask
        __m256i compareResult = _mm256_cmpeq_epi32(chunk, target);

        // Check if there is at least one element equal to 'val' ...
           in the comparison mask
        int mask = ...
           _mm256_movemask_ps(_mm256_castsi256_ps(compareResult));
        if (mask != 0) {
            return true; // Found: 'val' is contained in the vector
        }
    }
    return false; // 'val' was not found in the vector
}
```

After compiling the above code with the command shown at the beginning of Chapter 5, the vectorization report is displayed. The following image presents the results obtained without manual vectorization.

```
ubuntu@coco-tesista:~/SymbolicProject/cpp$ time ./vect
Fronti trovati: 10

real    0m0.012s
user    0m0.012s
sys     0m0.000s
```

As we can see from the results, the execution times are higher compared to the auto-vectorized case. Therefore, we can conclude that manual vectorization is not as efficient as the implementation provided by the library. Additionally, when testing the "find" function within a file with a minimal usage example, the compiler does not return warnings. This indicates that auto-vectorization does indeed work for this function.

5.4 final version

5.4.1 code

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <fstream>
#include <sstream>
#include <string>
#include <numeric>

// Definition of a function to determine whether one point ...
    dominates another point
// Returns true if point_p is dominated by point_q
bool dominates(std::vector<double> point_q, std::vector<double> ...
    point_p, int n) {

    int count = 0;
    int strict_count = 0;

    for(int i = 0; i < n; i++) {
        if(point_q[i] ≤ point_p[i]) {
            count++;
            if(point_q[i] < point_p[i]) {
                strict_count++;
            }
        }
    }
    if(count == n && strict_count > 0)
        return true;
    return false;
```

```
}

// Function to split a string based on a delimiter
std::vector<double> split(const std::string &s, char delimiter) {
    std::vector<double> tokens;
    std::stringstream ss(s);
    std::string token;
    while (std::getline(ss, token, delimiter)) {
        tokens.push_back(std::stod(token));
    }
    return tokens;
}

int main() {

    std::ifstream file("../output.csv");
    std::vector<std::vector<double>> matrix;

    if (file.is_open()) {
        std::string line;
        while (std::getline(file, line)) {
            std::vector<double> row = split(line, ',');
            matrix.push_back(row);
        }
        file.close();

    } else {
        std::cout << "Unable to open the file." << std::endl;
    }

    // Definition of variables and initialization
    const int pop_size =  matrix.size();
    std::vector<std::vector<double>> P = matrix; // contains the ...
       population read from the file
    std::vector<int> R(pop_size, 0); // vector containing the rank ...
       of the front for each point
    std::vector<bool> F_bool(pop_size, false); // true if the ...
       corresponding point belongs to the current front
    std::vector<bool> already_assigned_to_a_front(pop_size, ...
       false); // true if the corresponding point has been ...
       assigned to a front previously
    int current_front_index = 1; // index of the front to be ...
       considered in the while loop

    while (std::accumulate(already_assigned_to_a_front.begin(), ...
       already_assigned_to_a_front.end(), 0) < pop_size) {
```

```
        for (int p = 0; p < P.size(); ++p) {
            if (already_assigned_to_a_front[p]){
                continue;
            }
            int n_p = 0; // domination counter of p
            for (int q = 0; q < P.size(); ++q) {
                if (already_assigned_to_a_front[q]) {
                    continue;
                }
                if (dominates(P[q], P[p], P[q].size())) {
                    n_p = n_p + 1; // p is dominated by q
                }
            }

            if (n_p == 0) {
                R[p] = current_front_index;
                F_bool[p] = true;
            }
        }
        #pragma clang loop distribute(enable)
        for(int i = 0; i < pop_size; i++){
            already_assigned_to_a_front[i] = F_bool[i];
        }
        current_front_index = current_front_index + 1;
    }
    std::cout << "Found fronts: " << current_front_index - 1 << ...
        std::endl;
    return 0;
}
```

After compiling the above code with the command shown at the beginning of Chapter 5, the vectorization report is displayed, and following that, we showed the most significant results:

```
algorithm_non_vectorized_new_version.cpp:16:4: remark: vectorized loop (vectorization width: 8, interleaved count: 4)
      [-Rpass=loop-vectorize]
   for(int  i=0; i<n; i++){
   ^
```

From the previous image, we can observe that the 'domina' function, which is the main function in our code, is correctly auto-vectorized.

```
algorithm_non_vectorized_new_version.cpp:51:9: remark: loop not vectorized: value that could not be identified as
      reduction is used outside the loop [-Rpass-analysis=loop-vectorize]
        while (std::getline(file, line)) {
        ^
```

Looking at the previous image, we can see that the loop handling data loading isn't auto-vectorized. This was expected because the loop involves instructions for reading from files, and the compiler can't predict how much data it has to fetch.

```
algorithm_non_vectorized_new_version.cpp:38:5: remark: loop not vectorized: loop control flow is not understood by
      vectorizer [-Rpass-analysis=loop-vectorize]
    while (std::getline(ss, token, delimiter)) {
    ^
```

From the previous image, we observe that the loop is not vectorized due to the 'getLine' instruction.

5.5 Execution

After running the above code with a test dataset containing 100 individuals, we obtained the following results along with their respective execution times.

```
ubuntu@coco-tesista:~/SymbolicProject/cpp$ time ./final
Fronti trovati: 10

real    0m0.010s
user    0m0.011s
sys     0m0.000s
```

We have executed the code also without auto-vectorization of the processor and below we reported the results obtained in term of timing:

```
ubuntu@coco-tesista:~/SymbolicProject/cpp$ clang++ -o test_non_vect final_version.cpp
ubuntu@coco-tesista:~/SymbolicProject/cpp$ time ./test_non_vect
Fronti trovati: 10

real    0m0.023s
user    0m0.020s
sys     0m0.004s
```

6 Conclusion

The objective of our project is to leverage vectorization for accelerating the execution of NSGAII. As evident from the results obtained in the final version, it can be concluded that vectorization has indeed enhanced the algorithm's execution time. Additionally, there was no need for manual vectorization implementation, as the processor autonomously detected and vectorized all the operations. Considering that the algorithm doesn't demand a high volume of operations, the only vectorized segment relates to the dominance calculation, which constitutes the core of the algorithm. Despite the limited vectorized components, we observe a significant efficiency in terms of saved time.