DEPARTMENT OF COMPUTER, MODELING, ELECTRONIC
AND SYSTEMS ENGINEERING

Master Degree Couse in Computer Engineering

"Architetture Avanzate dei Sistemi di Elaborazione e Programmazione" course project report

Authors
Emanuele Vita (ID 264140)
Nicolas Crescente (ID 269084)
Giuseppe Mattia Greco (ID. 264526)

Introduction

This project aimed to analyze and optimize the performance of the *Simulated Annealing* algorithm to predict the tertiary structure of proteins through three main phases:
an initial implementation in C language, low-level optimization using Assembly, and parallelization with OpenMP. The primary goal was to improve execution times by leveraging advanced optimization and parallelization techniques.

In the first phase, the code was written in C to ensure a solid and readable foundation. Subsequently, some of the critical parts of the algorithm were converted to Assembly to exploit the hardware's capabilities. Finally, OpenMP was used to introduce parallelism, aiming to scale performance across multiple cores.

The report outlines the implementation process, design choices, obtained results, and encountered challenges, highlighting the progressive improvement in performance.

C Implementation

The initial implementation of the algorithm for Tertiary Structure Prediction of Proteins was entirely developed in C language. This approach guaranteed modularity and ease of debugging, providing a solid foundation for introducing subsequent optimizations in Assembly and OpenMP.

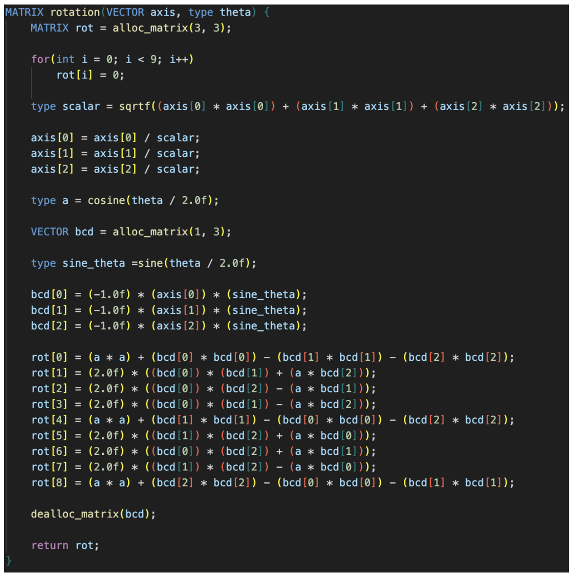
Various functions were designed to be completely independent, with each performing its specific calculations. For instance, energy calculation functions operate independently but contribute

to a central function that collects partial results and aggregates them with contributions from each function.
This modularity also extended to functions such as those for calculating *sine* and *cosine*, *distances*, and the *rotation matrix*.

Following an initial draft, we applied the first optimizations, such as maintaining data structures in ***row-major order***, including the `coords` matrix—designed to hold the three-dimensional coordinates of the protein chain atoms (N, Cα, C). This matrix was structured as a linear vector of size $(3 \cdot N) \times N$, allowing efficient traversal using an *offset*-based address management. Specifically, every triplet of elements in the matrix represents the three-dimensional coordinates (x, y, z) of an atom, enabling efficient calculations by skipping triplet by triplet.

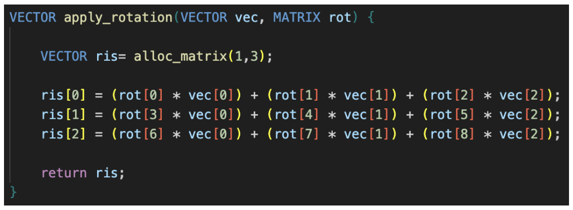
For energy calculation functions, we chose to implement `rotation` and `apply_rotation` separately, which are subsequently called within the `backbone` function responsible for calculating the new amino acid chain given the dihedral angles Φ and Ψ.
Specifically:

- **`rotation`:** computes the *three-dimensional rotation matrix* to rotate a vector around a specific axis by a given angle;
- **`apply_rotation`:** applies a rotation matrix to a three-dimensional vector, representing the new position of the atoms in the chain.



```c
MATRIX rotation(VECTOR axis, type theta) {
    MATRIX rot = alloc_matrix(3, 3);

    for(int i = 0; i < 9; i++)
        rot[i] = 0;

    type scalar = sqrtf((axis[0] * axis[0]) + (axis[1] * axis[1]) + (axis[2] * axis[2]));

    axis[0] = axis[0] / scalar;
    axis[1] = axis[1] / scalar;
    axis[2] = axis[2] / scalar;

    type a = cosine(theta / 2.0f);

    VECTOR bcd = alloc_matrix(1, 3);

    type sine_theta =sine(theta / 2.0f);

    bcd[0] = (-1.0f) * (axis[0]) * (sine_theta);
    bcd[1] = (-1.0f) * (axis[1]) * (sine_theta);
    bcd[2] = (-1.0f) * (axis[2]) * (sine_theta);

    rot[0] = (a * a) + (bcd[0] * bcd[0]) - (bcd[1] * bcd[1]) - (bcd[2] * bcd[2]);
    rot[1] = (2.0f) * ((bcd[0]) * (bcd[1]) + (a * bcd[2]));
    rot[2] = (2.0f) * ((bcd[0]) * (bcd[2]) - (a * bcd[1]));
    rot[3] = (2.0f) * ((bcd[0]) * (bcd[1]) - (a * bcd[2]));
    rot[4] = (a * a) + (bcd[1] * bcd[1]) - (bcd[0] * bcd[0]) - (bcd[2] * bcd[2]);
    rot[5] = (2.0f) * ((bcd[1]) * (bcd[2]) + (a * bcd[0]));
    rot[6] = (2.0f) * ((bcd[0]) * (bcd[2]) + (a * bcd[1]));
    rot[7] = (2.0f) * ((bcd[1]) * (bcd[2]) - (a * bcd[0]));
    rot[8] = (a * a) + (bcd[2] * bcd[2]) - (bcd[0] * bcd[0]) - (bcd[1] * bcd[1]);

    dealloc_matrix(bcd);

    return rot;
}
```

(a) `rotation` function



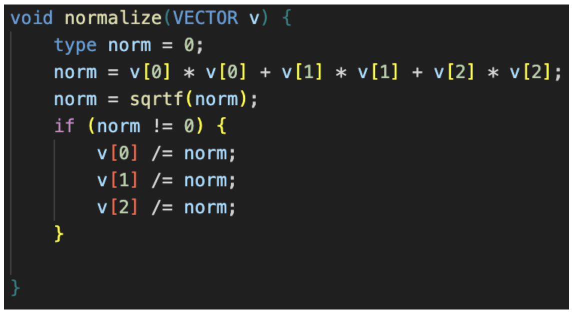
```c
VECTOR apply_rotation(VECTOR vec, MATRIX rot) {

    VECTOR ris= alloc_matrix(1,3);

    ris[0] = (rot[0] * vec[0]) + (rot[1] * vec[1]) + (rot[2] * vec[2]);
    ris[1] = (rot[3] * vec[0]) + (rot[4] * vec[1]) + (rot[5] * vec[2]);
    ris[2] = (rot[6] * vec[0]) + (rot[7] * vec[1]) + (rot[8] * vec[2]);

    return ris;
}
```

(b) `apply_rotation` function

As shown, in both functions, direct assignments to vectors and matrices are used. This implementation approach enables explicit result writing, avoiding loops or iterations that would add computational overhead. Additionally, it reduces the overhead of managing counters and control instructions in a hypothetical loop.

Since the `backbone` function is invoked multiple times, and the new directions of atoms are repeatedly calculated, using these direct assignments to frequently accessed elements helps *optimize cache usage*.

As a final implementation choice, we defined the `normalize` function to normalize a vector. Normalization is essential for many geometric operations, such as rotations and angle calculations, as it ensures vectors have a standard length. In our case, it was used within the `backbone` function.



```
void normalize(VECTOR v) {
    type norm = 0;
    norm = v[0] * v[0] + v[1] * v[1] + v[2] * v[2];
    norm = sqrtf(norm);
    if (norm != 0) {
        v[0] /= norm;
        v[1] /= norm;
        v[2] /= norm;
    }

}
```

`normalize` function

Distance Vector

Since calculating the energy of a configuration requires determining the distances between various atoms in the sequence, we opted to avoid recalculating distances at each iteration by using a vector to store the distances for the current configuration.
This is achieved through the function `all_distances`, which precomputes and stores all unique distances between pairs of Cα atoms in a linear vector called `distances`. This approach is more efficient than recalculating distances repeatedly during operations requiring them.

The logic is based on calculating the *number of unique atom pairs* in the sequence:

$$Number\ of\ pairs = \frac{N \cdot (N-1)}{2}$$

where N is the total number of amino acids in the sequence.

Instead of using a two-dimensional matrix of size $N \times N$ to store all distances, a linear vector was chosen for several reasons:

- **Memory efficiency:** An $N \times N$ matrix would require N^2 elements, including both symmetric distances $(d(i,j) = d(j,i))$ and the main diagonal (distances of a point to itself, always 0). These redundant elements would lead to inefficient memory usage;
- **Computational savings:** Since symmetric distances do not need to be computed twice, it is sufficient to store only unique pairs (i,j) where $i < j$. The number of such pairs is significantly smaller, amounting to $\frac{N \cdot (N-1)}{2}$;
- **Direct access to distances:** Using the mapping function `get_distance_index`, which maps each atom pair (i,j) to a unique position in the vector, distances can be accessed quickly without managing a two-dimensional structure.

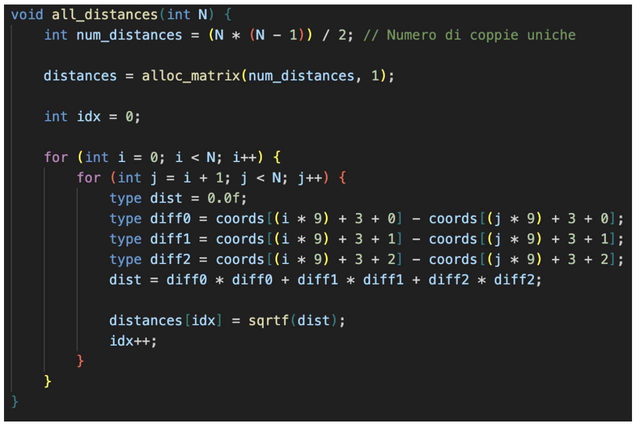
The distance between atoms i and j is computed using the Euclidean formula. The atom indices are derived from the `coords` matrix, which contains the three-dimensional coordinates of the atoms. Each calculated distance is stored in the `distances` vector at a position determined by the mapping function `get_distance_index`.

This function translates a pair (i,j) of atom indices into the corresponding index in the `distances` vector, as the distances are stored linearly. The logic is as follows:
Given i and j with $i < j$:

$$Linear\ index = i \cdot (N-1) - \frac{i \cdot (i+1)}{2} + (j-1)$$

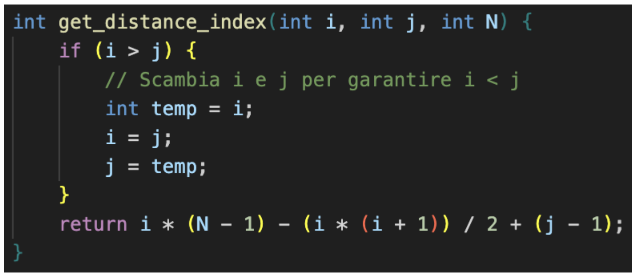
- $i \cdot (N-1)$: shifts the index to the correct row;
- $-\frac{i \cdot (i+1)}{2}$: removes the contribution of previous rows;
- $(j-1)$: selects the position in the current row.

This solution, since distances are frequently used and the `get_distance_index` function provides *immediate access* to the `distances` vector, has resulted in a further improvement in performance.
Below is a snippet of the relevant code:



```c
void all_distances(int N) {
    int num_distances = (N * (N - 1)) / 2; // Numero di coppie uniche

    distances = alloc_matrix(num_distances, 1);

    int idx = 0;

    for (int i = 0; i < N; i++) {
        for (int j = i + 1; j < N; j++) {
            type dist = 0.0f;
            type diff0 = coords[(i * 9) + 3 + 0] - coords[(j * 9) + 3 + 0];
            type diff1 = coords[(i * 9) + 3 + 1] - coords[(j * 9) + 3 + 1];
            type diff2 = coords[(i * 9) + 3 + 2] - coords[(j * 9) + 3 + 2];
            dist = diff0 * diff0 + diff1 * diff1 + diff2 * diff2;

            distances[idx] = sqrtf(dist);
            idx++;
        }
    }
}
```

(a) `all_distances` function

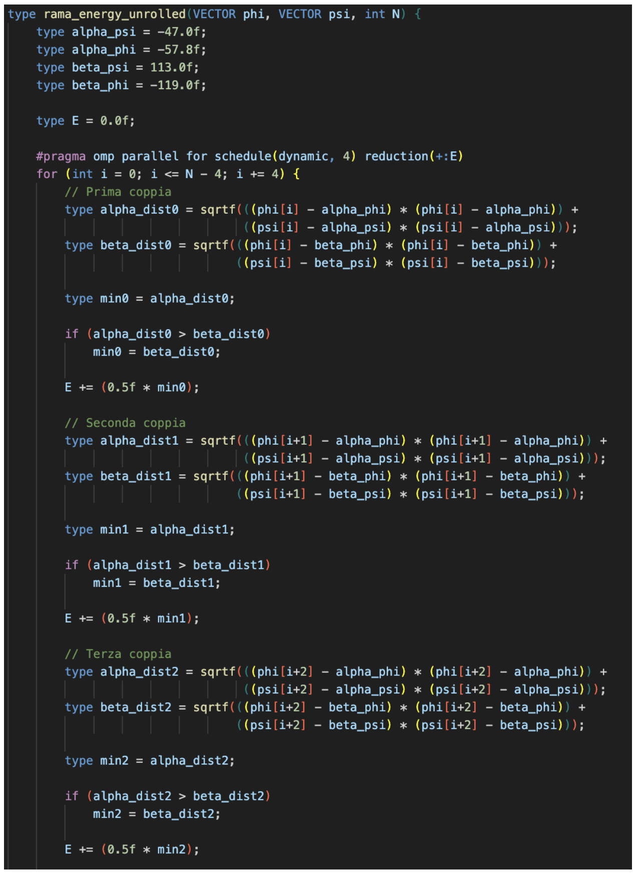


```c
int get_distance_index(int i, int j, int N) {
    if (i > j) {
        // Scambia i e j per garantire i < j
        int temp = i;
        i = j;
        j = temp;
    }
    return i * (N - 1) - (i * (i + 1)) / 2 + (j - 1);
}
```

(b) `get_distance_index` function

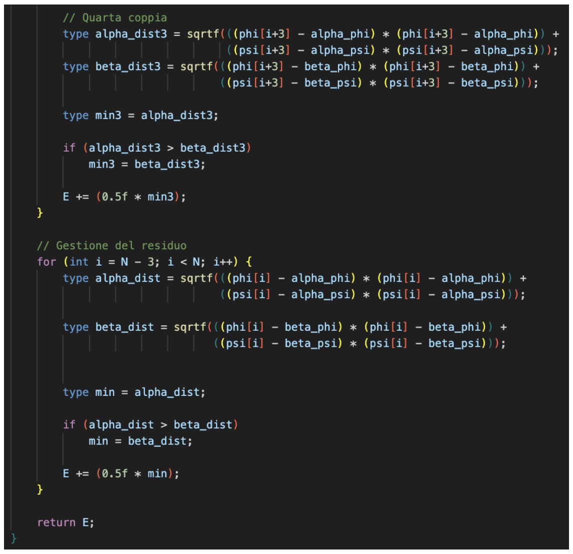
Unrolling of the `rama_energy` Function

During performance analysis, it was found that the `rama_energy` function, used to calculate the energy contribution based on the dihedral angles Φ and Ψ, was one of the critical points in terms of execution time. This function iterated over each amino acid in the sequence, repeatedly performing independent calculations for each element. To improve performance, the ***loop unrolling*** technique was applied to the inner loop, explicitly expanding multiple iterations of the loop into a single block of instructions.
Below is a snippet of the relevant code:



```c
type rama_energy_unrolled(VECTOR phi, VECTOR psi, int N) {
    type alpha_psi = -47.0f;
    type alpha_phi = -57.8f;
    type beta_psi = 113.0f;
    type beta_phi = -119.0f;

    type E = 0.0f;

    #pragma omp parallel for schedule(dynamic, 4) reduction(+:E)
    for (int i = 0; i <= N - 4; i += 4) {
        // Prima coppia
        type alpha_dist0 = sqrtf(((phi[i] - alpha_phi) * (phi[i] - alpha_phi)) +
                                 ((psi[i] - alpha_psi) * (psi[i] - alpha_psi)));
        type beta_dist0 = sqrtf(((phi[i] - beta_phi) * (phi[i] - beta_phi)) +
                                ((psi[i] - beta_psi) * (psi[i] - beta_psi)));

        type min0 = alpha_dist0;

        if (alpha_dist0 > beta_dist0)
            min0 = beta_dist0;

        E += (0.5f * min0);

        // Seconda coppia
        type alpha_dist1 = sqrtf(((phi[i+1] - alpha_phi) * (phi[i+1] - alpha_phi)) +
                                 ((psi[i+1] - alpha_psi) * (psi[i+1] - alpha_psi)));
        type beta_dist1 = sqrtf(((phi[i+1] - beta_phi) * (phi[i+1] - beta_phi)) +
                                ((psi[i+1] - beta_psi) * (psi[i+1] - beta_psi)));

        type min1 = alpha_dist1;

        if (alpha_dist1 > beta_dist1)
            min1 = beta_dist1;

        E += (0.5f * min1);

        // Terza coppia
        type alpha_dist2 = sqrtf(((phi[i+2] - alpha_phi) * (phi[i+2] - alpha_phi)) +
                                 ((psi[i+2] - alpha_psi) * (psi[i+2] - alpha_psi)));
        type beta_dist2 = sqrtf(((phi[i+2] - beta_phi) * (phi[i+2] - beta_phi)) +
                                ((psi[i+2] - beta_psi) * (psi[i+2] - beta_psi)));

        type min2 = alpha_dist2;

        if (alpha_dist2 > beta_dist2)
            min2 = beta_dist2;

        E += (0.5f * min2);
```

(a) First part



```c
        // Quarta coppia
        type alpha_dist3 = sqrtf(((phi[i+3] - alpha_phi) * (phi[i+3] - alpha_phi)) +
                                 ((psi[i+3] - alpha_psi) * (psi[i+3] - alpha_psi)));
        type beta_dist3 = sqrtf(((phi[i+3] - beta_phi) * (phi[i+3] - beta_phi)) +
                                ((psi[i+3] - beta_psi) * (psi[i+3] - beta_psi)));

        type min3 = alpha_dist3;

        if (alpha_dist3 > beta_dist3)
            min3 = beta_dist3;

        E += (0.5f * min3);
    }

    // Gestione del residuo
    for (int i = N - 3; i < N; i++) {
        type alpha_dist = sqrtf(((phi[i] - alpha_phi) * (phi[i] - alpha_phi)) +
                                ((psi[i] - alpha_psi) * (psi[i] - alpha_psi)));

        type beta_dist = sqrtf(((phi[i] - beta_phi) * (phi[i] - beta_phi)) +
                               ((psi[i] - beta_psi) * (psi[i] - beta_psi)));


        type min = alpha_dist;

        if (alpha_dist > beta_dist)
            min = beta_dist;

        E += (0.5f * min);
    }

    return E;
}
```

(b) Second part

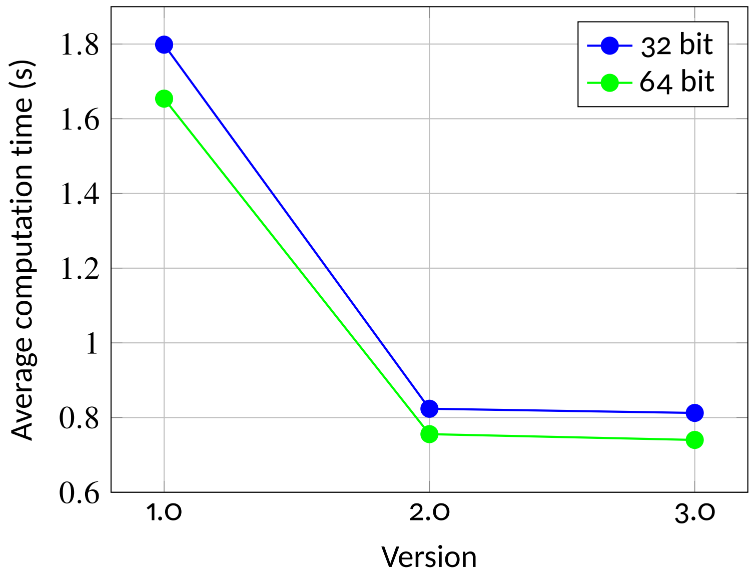
The benefits of this implementation choice led to a *significant* reduction in execution time and better scalability for longer sequences, with a direct impact on the overall time required for the *Ramachandran* energy calculations.

Results Achieved

Through the development of different versions, we obtained the following results:

VERSION	DESCRIPTION/MODIFICATIONS	AVERAGE COMPUTATION TIME
1.0	Base C Version	1.7983|1.6538
2.0	Introduction of `all_distances` and `get_distance_index` functions	0.8236|0.7401
3.0	Modification of the `rama_energy` function through loop unrolling	0.8124|0.7556

Average computation times for different versions over 10 executions (32|64 bit).



Line chart of average computation times for 32 and 64-bit versions.

After achieving significant improvements in execution time, we also analyzed the precision of the processed data. The use of *float* (32-bit numbers) introduces an *inevitable intrinsic error* due to their limited precision, especially when compared to the *double* type, which offers higher precision and was adopted in the 64-bit version.

To evaluate the impact of this difference, we developed a function to calculate the Mean Squared Error (MSE) and another function to extract data generated by our program and compare it with reference values provided as expected results.

The results obtained are as follows:

Architecture	**Time**	**MSE Float - Float**	**MSE Float - Double**	**MSE Double - Double**	**Angle Coincidences**
C - 32 bit	$0.793\ s$	$1.280022\ phi$ $1.144339\ psi$	$1.017244\ phi$ $0.882342\ psi$	X	200/256 phi 200/256 psi *Float - Double*
C - 64 bit	$0.749\ s$	X	X	$0.000000\ phi$ $0.000000\ psi$	256/256 phi 256/256 psi

For the 32-bit version, the Mean Squared Error was calculated by comparing the results first between *float* and *float* and then between *float* and *double*, as the latter are more precise. We wanted to assess the precision of the *float* implementation. Additionally, the angle coincidence count was performed between *float* and *double* to determine how closely our implementation approached the more precise version.

Assembly Optimizations

Optimization using Assembly language is a fundamental step in improving the performance of applications that perform intensive and repetitive calculations. While high-level languages like C are characterized by modularity and ease of development, Assembly allows direct access to hardware, fully leveraging its specific capabilities.

In the context of this project, the initial C implementation was optimized by converting the critical parts of the code into Assembly. This decision was driven by the need to optimize memory access, minimize latency times, and utilize SIMD instructions provided by SSE and AVX hardware extensions. These instructions allow parallel operations on multiple data elements, significantly enhancing computational efficiency.
Specifically, two versions were developed: the first for the `x86-32+SSE` architecture and the second for the `x86-64+AVX` architecture.

One of the initial strategies adopted involved converting the energy calculation functions into Assembly, aiming to utilize SIMD instructions to improve vector calculation performance.
Calculating the energy of a structure requires knowing the various distances between alpha-carbon atoms (Cα), which, as mentioned earlier, are calculated in C and stored in the `distances` vector. Transposing everything into Assembly is an extremely costly solution requiring multiple calls to the C function and, consequently, numerous memory accesses.

Considering this limitation, it was decided to avoid implementing all energy calculation functions in Assembly and instead focus on optimizing the `normalize` and `apply_rotation` functions. Finally, we considered combining the calculations for *hydrophobic energy* and *electrostatic energy* into a single function named `combined_energy`.
However, this idea was discarded as it generated too much ***overhead*** since calculating the atom distances involved calling the C function `get_distance_index`. Instead, a `combined_energy`

function was implemented directly in C to combine `packing_energy`, `electrostatic_energy`, and `hydrophobic_energy`.

Another strategy adopted was modifying the allocation of the direction vectors `v1`, `v2`, and `v3`, increasing the size from three to **four** elements. This decision was motivated by the fact that SSE `xmm` registers can hold four values. However, since the vectors consist of only three elements, the last value was set to 0 to ensure calculations were not compromised.

A similar approach was applied to the rotation matrix `rot`. Initially allocated as a 3×3 matrix in row-major order, it was reorganized into column-major order and converted into a 3×4 structure. This change included ***padding*** in the fourth column, ensuring all elements were contiguous in memory.

These modifications aimed to improve performance, increase opportunities for parallelizing calculations, and fully exploit SIMD instructions. Furthermore, optimizing memory *alignment* contributed to maximizing the overall efficiency of the implementation.

Implementation of the `normalize` Function

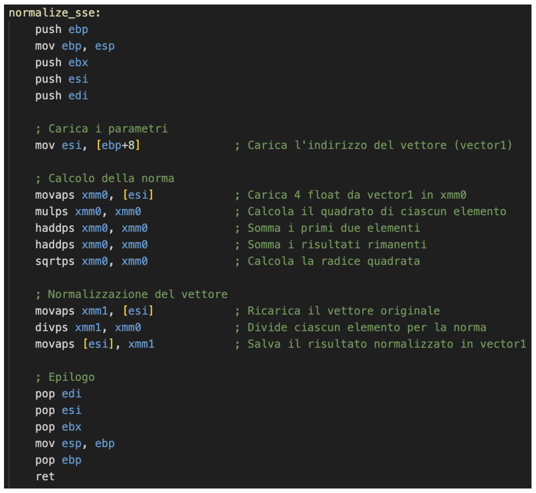
Regarding the `normalize` function, it takes a pointer to a vector of four numbers (`float` values stored in an `xmm` register or `double` values stored in a `ymm` register) as input and returns the normalized vector by overwriting it in memory.

`normalize_sse`

Specifically, the `x86-32+SSE` version loads four vector elements into the `xmm0` register, squares them using the `mulps` instruction, and then sums the squares via a *horizontal sum* using the `haddps` instruction. The result is square-rooted and stored in the `xmm1` register.
Finally, each element of the vector stored in `xmm0` is divided by the norm calculated and stored in the `xmm1` register using the `divps` instruction.

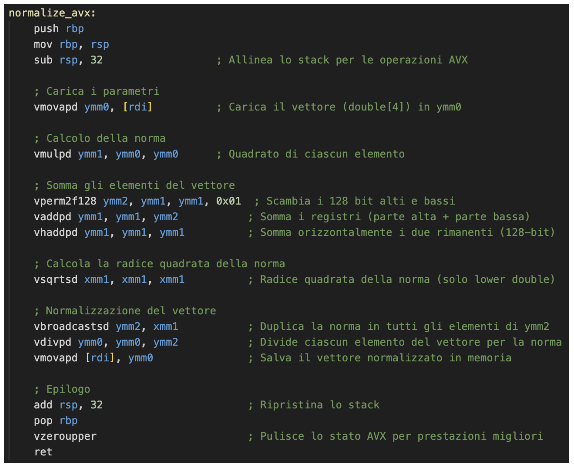
`normalize_avx`

The `x86-64+AVX` version is not significantly different from the previous one; the main difference lies in the fact that each AVX register extends the `xmm` register by adding another 128 bits (doubling its size). This allows data to be processed with greater precision.



```
normalize_sse:
    push ebp
    mov ebp, esp
    push ebx
    push esi
    push edi

    ; Carica i parametri
    mov esi, [ebp+8]              ; Carica l'indirizzo del vettore (vector1)

    ; Calcolo della norma
    movaps xmm0, [esi]            ; Carica 4 float da vector1 in xmm0
    mulps xmm0, xmm0              ; Calcola il quadrato di ciascun elemento
    haddps xmm0, xmm0             ; Somma i primi due elementi
    haddps xmm0, xmm0             ; Somma i risultati rimanenti
    sqrtps xmm0, xmm0             ; Calcola la radice quadrata

    ; Normalizzazione del vettore
    movaps xmm1, [esi]            ; Ricarica il vettore originale
    divps xmm1, xmm0              ; Divide ciascun elemento per la norma
    movaps [esi], xmm1            ; Salva il risultato normalizzato in vector1

    ; Epilogo
    pop edi
    pop esi
    pop ebx
    mov esp, ebp
    pop ebp
    ret
```

(a) `normalize_sse`



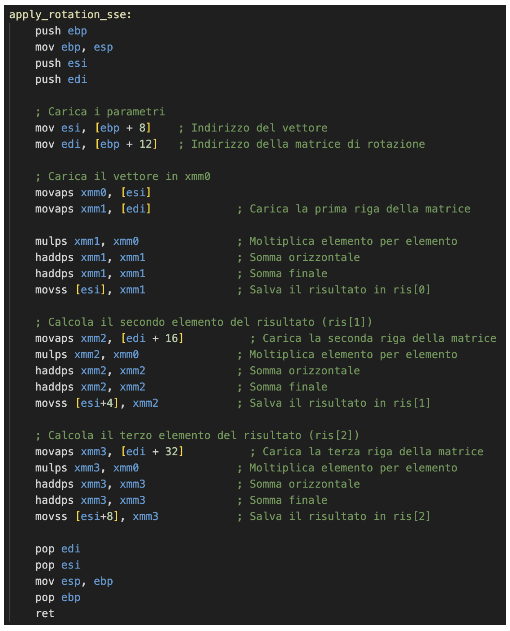
```
normalize_avx:
    push rbp
    mov rbp, rsp
    sub rsp, 32                 ; Allinea lo stack per le operazioni AVX

    ; Carica i parametri
    vmovapd ymm0, [rdi]         ; Carica il vettore (double[4]) in ymm0

    ; Calcolo della norma
    vmulpd ymm1, ymm0, ymm0     ; Quadrato di ciascun elemento

    ; Somma gli elementi del vettore
    vperm2f128 ymm2, ymm1, ymm1, 0x01  ; Scambia i 128 bit alti e bassi
    vaddpd ymm1, ymm1, ymm2            ; Somma i registri (parte alta + parte bassa)
    vhaddpd ymm1, ymm1, ymm1           ; Somma orizzontalmente i due rimanenti (128-bit)

    ; Calcola la radice quadrata della norma
    vsqrtsd xmm1, xmm1, xmm1           ; Radice quadrata della norma (solo lower double)

    ; Normalizzazione del vettore
    vbroadcastsd ymm2, xmm1            ; Duplica la norma in tutti gli elementi di ymm2
    vdivpd ymm0, ymm0, ymm2            ; Divide ciascun elemento del vettore per la norma
    vmovapd [rdi], ymm0                ; Salva il vettore normalizzato in memoria

    ; Epilogo
    add rsp, 32                        ; Ripristina lo stack
    pop rbp
    vzeroupper                         ; Pulisce lo stato AVX per prestazioni migliori
    ret
```

(b) `normalize_avx`

Implementation of the `apply_rotation` Function

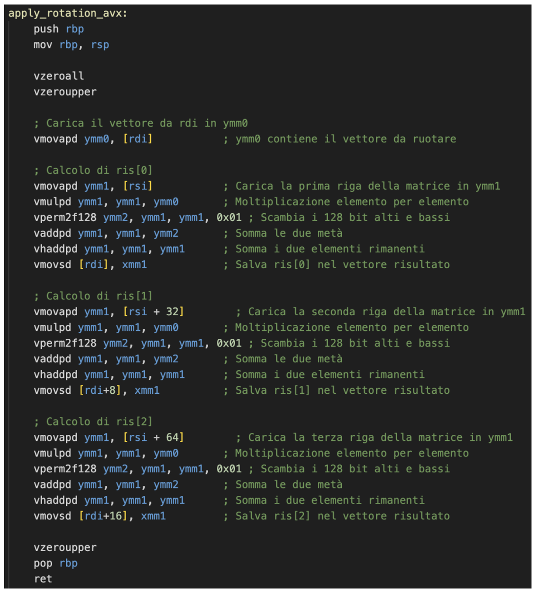
The `apply_rotation` function implements the multiplication between a three-dimensional vector and a 3×3 matrix using SSE and AVX instructions.
This operation is essential for calculating the new coordinates of a point in three-dimensional space after a rotation.

Specifically, the function performs the *dot product* between the matrix and the vector: it loads an entire column of the matrix into an `xmm/ymm` register, multiplies it by the vector previously loaded into another `xmm/ymm` register, and finally performs a horizontal sum along the matrix column. This process is repeated for all columns, and the result is returned.



```
apply_rotation_sse:
    push ebp
    mov ebp, esp
    push esi
    push edi

    ; Carica i parametri
    mov esi, [ebp + 8]    ; Indirizzo del vettore
    mov edi, [ebp + 12]   ; Indirizzo della matrice di rotazione

    ; Carica il vettore in xmm0
    movaps xmm0, [esi]
    movaps xmm1, [edi]            ; Carica la prima riga della matrice

    mulps xmm1, xmm0              ; Moltiplica elemento per elemento
    haddps xmm1, xmm1             ; Somma orizzontale
    haddps xmm1, xmm1             ; Somma finale
    movss [esi], xmm1             ; Salva il risultato in ris[0]

    ; Calcola il secondo elemento del risultato (ris[1])
    movaps xmm2, [edi + 16]         ; Carica la seconda riga della matrice
    mulps xmm2, xmm0              ; Moltiplica elemento per elemento
    haddps xmm2, xmm2             ; Somma orizzontale
    haddps xmm2, xmm2             ; Somma finale
    movss [esi+4], xmm2           ; Salva il risultato in ris[1]

    ; Calcola il terzo elemento del risultato (ris[2])
    movaps xmm3, [edi + 32]         ; Carica la terza riga della matrice
    mulps xmm3, xmm0              ; Moltiplica elemento per elemento
    haddps xmm3, xmm3             ; Somma orizzontale
    haddps xmm3, xmm3             ; Somma finale
    movss [esi+8], xmm3           ; Salva il risultato in ris[2]

    pop edi
    pop esi
    mov esp, ebp
    pop ebp
    ret
```

(a) `apply_rotation_sse`



```
apply_rotation_avx:
    push rbp
    mov rbp, rsp

    vzeroall
    vzeroupper

    ; Carica il vettore da rdi in ymm0
    vmovapd ymm0, [rdi]           ; ymm0 contiene il vettore da ruotare

    ; Calcolo di ris[0]
    vmovapd ymm1, [rsi]           ; Carica la prima riga della matrice in ymm1
    vmulpd ymm1, ymm1, ymm0       ; Moltiplicazione elemento per elemento
    vperm2f128 ymm2, ymm1, ymm1, 0x01 ; Scambia i 128 bit alti e bassi
    vaddpd ymm1, ymm1, ymm2       ; Somma le due metà
    vhaddpd ymm1, ymm1, ymm1      ; Somma i due elementi rimanenti
    vmovsd [rdi], xmm1            ; Salva ris[0] nel vettore risultato

    ; Calcolo di ris[1]
    vmovapd ymm1, [rsi + 32]        ; Carica la seconda riga della matrice in ymm1
    vmulpd ymm1, ymm1, ymm0       ; Moltiplicazione elemento per elemento
    vperm2f128 ymm2, ymm1, ymm1, 0x01 ; Scambia i 128 bit alti e bassi
    vaddpd ymm1, ymm1, ymm2       ; Somma le due metà
    vhaddpd ymm1, ymm1, ymm1      ; Somma i due elementi rimanenti
    vmovsd [rdi+8], xmm1          ; Salva ris[1] nel vettore risultato

    ; Calcolo di ris[2]
    vmovapd ymm1, [rsi + 64]        ; Carica la terza riga della matrice in ymm1
    vmulpd ymm1, ymm1, ymm0       ; Moltiplicazione elemento per elemento
    vperm2f128 ymm2, ymm1, ymm1, 0x01 ; Scambia i 128 bit alti e bassi
    vaddpd ymm1, ymm1, ymm2       ; Somma le due metà
    vhaddpd ymm1, ymm1, ymm1      ; Somma i due elementi rimanenti
    vmovsd [rdi+16], xmm1         ; Salva ris[2] nel vettore risultato

    vzeroupper
    pop rbp
    ret
```

(b) `apply_rotation_avx`

Results Achieved

With this implementation, we obtained the following results:

Architecture	Time	MSE Float - Float	MSE Float - Double	MSE Double - Double	Angle Coincidences
x86-32+SSE	$0.544\ s$	1.280022ϕ 1.144339ψ	1.017244ϕ 0.882342ψ	X	200/256 ϕ 200/256 ψ *Float - Double*
x86-64+AVX	$0.519\ s$	X	X	0.000000ϕ 0.000000ψ	256/256 ϕ 256/256 ψ

We can observe that neither the MSE nor the number of *float - double* coincidences changed compared to the implementation using only C. Additionally, there is an improvement in execution time.

OpenMP Optimizations

OpenMP is an API for developing cross-platform shared-memory applications in C, C++, and Fortran. The goal is to create a multithreaded implementation, enabling parallel code execution managed across multiple threads.

In practice, the parallel execution section is within loops, such as `for` statements, where a specific directive creates additional threads: the main thread (thread 0) branches into multiple sub-threads (with dynamic or defined multiplicity).

```
    #pragma omp parallel
```

Additionally, OpenMP provides extensions for controlling parallel structures, workload sharing, shared and private variable clauses, synchronization, runtime functions, and environment variables.

In our code, OpenMP was applied to only two functions. Since the code was distributed across numerous functions (both in C and Assembly), testing revealed that creating threads within the various loops did not provide time benefits. This was because the `for` statements were too short, meaning that instantiating and destroying threads quickly (relative to the computation) required more time than executing the code on a single thread. The functions where we implemented OpenMP are:

- `combined_energy`;
- `rama_energy_unrolled`.

The C code remained the same as the version without OpenMP, with only the addition of OpenMP directives.

Parallelization Structure of `combined_energy`

At the beginning of the function, we included the following OpenMP directive:

```
#pragma omp parallel for schedule(dynamic, 10) reduction(+:total_energy)
```

1. **Parallelization of the Outer Loop**
   The directive:

   ```
       #pragma omp parallel for
   ```

   distributes the iterations of the outer loop (ranging from $i = 0$ to $i = N - 1$) among the available threads, enabling them to calculate the energy contributions associated with each amino acid in parallel.

2. **Clause Type: `schedule(dynamic, 10)`**
   The clause type:

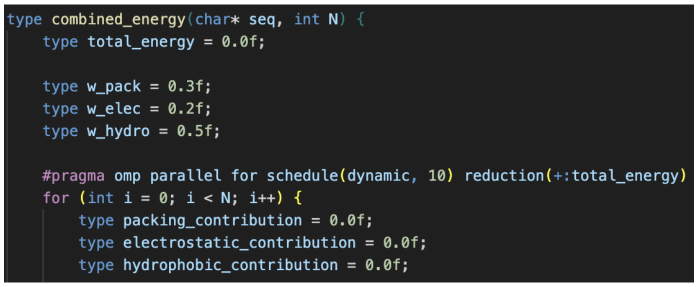
   ```
       schedule(dynamic, 10)
   ```

   divides the loop into blocks of ten iterations, dynamically assigning them to threads. This approach is particularly useful when the computation time for each iteration varies (e.g., due to the number of interactions between an amino acid and others). With `dynamic`, a thread that quickly completes its iterations can obtain a new block, improving overall efficiency and reducing thread idle time.

3. **Reduction of the `total_energy` Variable**
   The `total_energy` variable is shared among the threads. To avoid *data race* conditions, the clause:

   ```
           reduction(+:total_energy)
   ```

   is used, ensuring each thread has a private copy of the variable. At the end of the parallel execution, all private copies are safely combined (summed) into the global `total_energy` variable.



```
type combined_energy(char* seq, int N) {
    type total_energy = 0.0f;

    type w_pack = 0.3f;
    type w_elec = 0.2f;
    type w_hydro = 0.5f;

    #pragma omp parallel for schedule(dynamic, 10) reduction(+:total_energy)
    for (int i = 0; i < N; i++) {
        type packing_contribution = 0.0f;
        type electrostatic_contribution = 0.0f;
        type hydrophobic_contribution = 0.0f;
```

`combined_energy` function using OpenMP

Considerations on `combined_energy`

The use of OpenMP in this function is appropriate for handling the intensive computation of interactions in a protein sequence. However, the synchronization cost associated with the `reduction` clause and the `dynamic` scheduling type may introduce slight overhead for very small `N` values, where parallelization might not be advantageous. In such cases, a sequential version could be more efficient.

Parallelization Structure of `rama_energy_unrolled`

For this function, we used the following OpenMP directive:

```
#pragma omp parallel for schedule(dynamic, 4) reduction(+:E)
```

1. **Parallelization of the Main Loop**
   The function's main loop processes data in blocks of four (via *unrolling*) and iterates from $i = 0$ to $i = N - 4$. Thanks to the `#pragma omp parallel for` directive, the loop iterations are distributed among the available threads, enabling parallel computation of energy contributions associated with groups of amino acids.

2. **Dynamic Scheduling Scheme**
   The scheduling scheme `schedule(dynamic, 4)` divides the loop into blocks of four iterations, dynamically assigning them to threads. This approach is particularly useful in the case of computational imbalances among iterations, ensuring efficient thread utilization.

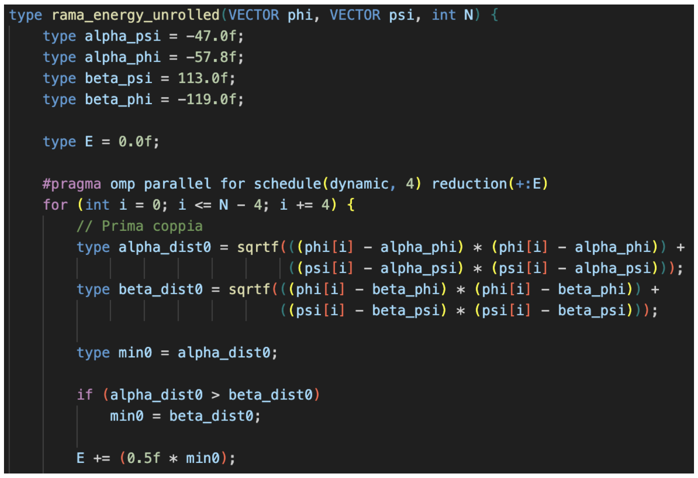
3. **Reduction of the `E` Variable**
   The variable `E` (total energy) is shared among threads. To avoid concurrency issues, the `reduction(+:E)` clause is used. Each thread maintains a private copy of `E` during computation, and at the end of the loop, the private copies are safely combined (summed) into the global `E` variable.

Residual Handling

The main loop processes blocks of four amino acids at a time. To handle any remaining elements ($N \mod 4$), a sequential loop calculates the energy contributions for the residuals independently; as this is a residue management loop, only a few elements are processed, so no OpenMP directive was included.

Considerations on `rama_energy_unrolled`

The use of OpenMP in the `rama_energy_unrolled` function is an example of *effective* optimization in a high-computation-intensity context. However, the effectiveness of parallelization depends on the sequence length (`N`) and the number of available threads. For very small `N` values, the overhead introduced by parallelization may outweigh the benefits, making sequential execution preferable.



```
type rama_energy_unrolled(VECTOR phi, VECTOR psi, int N) {
    type alpha_psi = -47.0f;
    type alpha_phi = -57.8f;
    type beta_psi = 113.0f;
    type beta_phi = -119.0f;

    type E = 0.0f;

    #pragma omp parallel for schedule(dynamic, 4) reduction(+:E)
    for (int i = 0; i <= N - 4; i += 4) {
        // Prima coppia
        type alpha_dist0 = sqrtf(((phi[i] - alpha_phi) * (phi[i] - alpha_phi)) +
                                 ((psi[i] - alpha_psi) * (psi[i] - alpha_psi)));
        type beta_dist0 = sqrtf(((phi[i] - beta_phi) * (phi[i] - beta_phi)) +
                                ((psi[i] - beta_psi) * (psi[i] - beta_psi)));

        type min0 = alpha_dist0;

        if (alpha_dist0 > beta_dist0)
            min0 = beta_dist0;

        E += (0.5f * min0);
```

`rama_energy` function using OpenMP

Benefits of Parallelization

- **Improved Performance:** Parallel execution enables distributing the computational load among threads, significantly reducing execution time compared to the sequential approach, particularly for high `N` values.
- **Optimal Utilization of Hardware Resources:** Dynamic distribution of iterations ensures a more uniform utilization of processor cores, reducing the risk of load imbalances among threads.
- **Reduced Computational Cost:** Since the primary energy calculations are independent for each iteration of the outer loop, they can be executed in parallel without conflicts, making the function particularly suitable for parallelization.
- **Efficiency Through *Loop Unrolling*:** Loop unrolling enhances computational efficiency by reducing the overhead associated with loop condition checks.

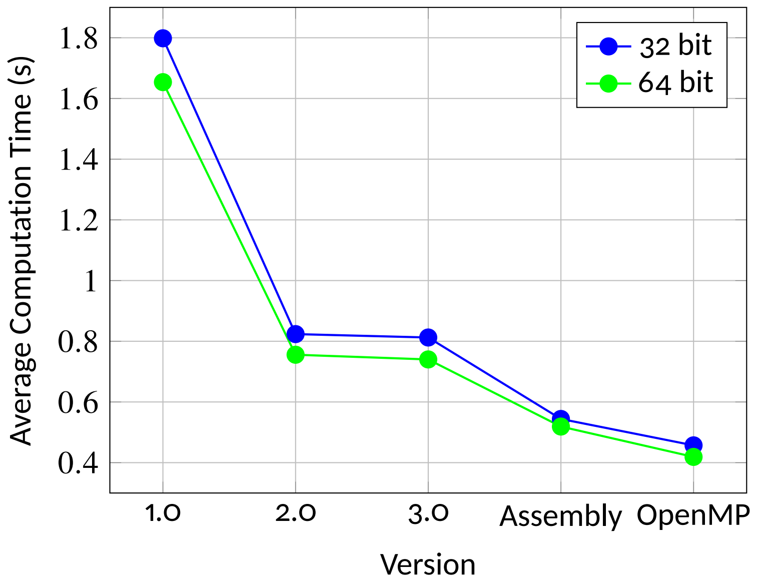
Results Achieved

With this implementation, we obtained the following results:

Architecture	**Time**	**MSE Float - Float**	**MSE Float - Double**	**MSE Double - Double**	**Angle Coincidences**
x86-32 + OpenMP	$0.457\ s$	1.280022ϕ 1.144339ψ	1.017244ϕ 0.882342ψ	X	200/256 ϕ 200/256 ψ *Float - Double*
x86-64 + OpenMP	$0.419\ s$	X	X	0.000000ϕ 0.000000ψ	256/256 ϕ 256/256 ψ

Conclusions

Below is a graph of the obtained times, providing a comprehensive view of the incremental improvement in execution time:



The average computation times were progressively reduced throughout all phases of the project. The use of Assembly demonstrated the value of hardware-specific optimizations, while the implementation of OpenMP highlighted the ability to leverage multicore architectures to further enhance performance.

To reduce computational bottlenecks, solutions such as loop unrolling and optimized memory allocation were adopted. However, some techniques—such as combining different energy functions—proved counterproductive, emphasizing the importance of balancing implementation complexity with tangible benefits.

The comparison between 32-bit and 64-bit versions underscored the importance of precision in energy calculations. While *float* values are more efficient, they introduce a larger margin of error compared to *double*. This aspect is crucial in scientific applications that demand high precision.