Algoritmo di ottimizzazione "Fish School Search" in linguaggio assembly x86-32+SSE, x86-64+AVX e openMP

Progetto di Architetture e Programmazione dei Sistemi di Elaborazione

Gruppo 1: Folino Filippo Andrea, Squillace Simone, Sullazzo Teodoro

DIMES (Dipartimento di Ingegneria Informatica, Modellistica, Elettronica e Sistemistica)
Università della Calabria, Via Pietro Bucci, 87036, Arcavacata di Rende, CS, Italia

23 gennaio 2022

Sommario: Il presente elaborato ha l'obiettivo di mettere a punto un'implementazione dell'algoritmo Fish School Search in linguaggio C e di migliorarne le prestazioni utilizzando le tecniche di ottimizzazione basate sull'organizzazione dell'hardware. L'ambiente sw/hw di riferimento è costituito dal linguaggio di programmazione C (gcc), dal linguaggio assembly x86-32+SSE e dalla sua estensione x86-32+AVX (nasm) e dal sistema operativo Linux (ubuntu).

1 Introduzione

Il **Fish School Search** (FSS) è un algoritmo di ottimizzazione ispirato al comportamento collettivo dei banchi di pesci. In quest'algoritmo il banco cerca cibo nelle posizioni in cui è più probabile trovarlo, facendo sì che i pesci si muovano verso il minimo della funzione al fine di mangiare, acquisendo così peso. Questa tecnica di ricerca presenta le seguenti caratteristiche:

1. è in grado di gestire le elevate dimensioni degli spazi di ricerca
2. rappresenta un approccio basato sulla popolazione, influenzato dal *comportamento collettivo emergente* per aumentare la probabilità di sopravvivenza del banco.

Il processo di ricerca in FSS si basa su una popolazione di individui con memoria limitata: i pesci mantengono solo la loro memoria innata (cioè i loro pesi), ciò contribuisce a mantenere un registro delle migliori posizioni visitate.

Lo sviluppo della tecnica FSS si basa sui seguenti comportamenti:

- **Alimentazione:** è ispirata dall'istinto naturale degli individui (pesci) a cercare cibo al fine di guadagnare peso (i.e. trovare zone del dominio dove la funzione è "migliore")
- **Nuoto**: imita il movimento coordinato e collettivo prodotto da tutti i pesci. Il nuoto è guidato dalle esigenze di alimentazione e, in definitiva, guiderà il processo di ricerca.
- **Allevamento**: ispirato dal meccanismo di selezione naturale, permette agli individui di successo di sopravvivere e di guidare il banco di pesci. In questo modo è possibile sfruttare il più possibile le soluzioni candidate migliori.

Le posizioni dei pesci sono rappresentati da vettori d-dimensionali. Ogni vettore rappresenta una possibile soluzione per il problema di ottimizzazione. In breve, l'algoritmo è composto dagli operatori di alimentazione e movimento, quest'ultimo diviso in tre componenti: in individuale, istintivo e volitivo.

La componente individuale del movimento consente a ogni pesce del banco di eseguire una ricerca locale (mediante un approccio casuale) di regioni promettenti nel dominio. La nuova posizione è accettata solo se la funzione migliora in questo nuovo punto. Altrimenti, il pesce rimane nella posizione precedente.

La componente istintiva è definita in modo tale che i pesci con una migliore variazione della funzione obiettivo attraggano gli altri nella loro posizione, questa è ottenuta calcolando la media ponderata degli spostamenti individuali, in relazione al valore della funzione. Invece, quella volitiva è utilizzata per regolare la capacità di esplorazione del banco di pesci durante il processo di ricerca, quest'ultima componente fa uso del baricentro del banco, anch'esso calcolato

con la media ponderata delle posizioni individuali ma, questa volta, rispetto al peso totale del banco di pesci.

2 Scelte progettuali

In questo paragrafo sono discusse le scelte alla base della progettazione dell'algoritmo FSS, tenendo conto dei suoi principi, punti critici da ottimizzare, discutendo la rappresentazione dei dati in memoria e il relativo accesso, e delle diverse tecniche di parallelizzazione implementate.

L'algoritmo FSS fa uso dei seguenti principi:

1. Il singolo pesce richiede operazioni semplici per essere gestito
2. Vari mezzi di memorizzazione delle informazioni (i.e. pesi del pesce e baricentro del banco)
3. Il nuoto è costituito da componenti distinte
4. Scarse comunicazioni tra individui vicini (i pesci devono pensare singolarmente, ma anche essere consapevoli di ciò che sta intorno)
5. Controllo centralizzato minimo (principalmente controllare autonomamente il raggio del banco)
6. Alcuni meccanismi per evitare l'affollamento[1]
7. Scalabilità (in termini di complessità delle attività di ottimizzazione/ricerca),
8. Autonomia (cioè capacità di autocontrollo del funzionamento).

Avendo seguito pedissequamente l'algoritmo nella realizzazione del programma, questi principi sono stati automaticamente raggiunti.

Per ottimizzare l'esecuzione del programma si è scelto di far uso delle seguenti tecniche di parallelizzazione:

1. **Parallelismo SIMD**. Si tratta di un'architettura di calcolo parallelo comprendente un certo numero di unità di elaborazione in grado di eseguire le stesse istruzioni (single instruction) su vettori di dati (multi data). In breve è presente un'unica unità di controllo che esegue un'istruzione alla volta comandando più ALU operanti in maniera sincrona tra loro. Ad ogni passo le ALU eseguono la stessa istruzione ma su dati differenti. Tale parallelismo è reso possibile grazie all'uso dei repertori SSE e AVX.
   - **SSE**
     Vengono introdotti registri a 128 bit, gli XMM. Da XMM0 fino a XMM7 per architetture a 32 bit, mentre, nel caso di architetture a 64 bit, si arriva fino a XMM15 (per un totale quindi di 16 registri).
   - **AVX**
     Vengono impiegati registri a 256 bit, gli YMM, i quali estendono gli XMM a 128 bit.
2. **Loop unrolling**. Consiste nell'aumentare la velocità di esecuzione del programma raggruppando più istruzioni in un singolo ciclo, ciò per ridurre il numero di controlli di fine loop che vengono effettuati al termine dell'iterazione. In tal modo è possibile minimizzare le penalità a cui si va incontro in caso di predizione errata da parte della BPU.

   In presenza di istruzioni indipendenti tra loro, è possibile che queste vengano eseguite in parallelo dal processore grazie alla presenza di più ALU duplicate al suo interno. La tecnica di ottimizzazione presenta tuttavia degli svantaggi. Si ricorda in particolare l'aumento del codice del programma ed il possibile incremento del numero di registri per iterazione per la memorizzazione delle variabili temporanee.

2.1 Rappresentazione dei dati

Un punto di fondamentale importanza per la realizzazione del progetto è stato la scelta di come rappresentare i dati, ed in particolar modo, la matrice delle posizioni dei pesci.

Quando ci si trova di fronte ad una matrice, infatti, sono tre le vie percorribili:

1. Rappresentarla come vettore di vettori
2. Rappresentarla per righe
3. Rappresentarla per colonne

[1] Dal punto di vista del matematico, "affollamento" è il movimento collettivo di un gruppo di entità semoventi, in natura è esibito da molti esseri viventi come uccelli, pesci, batteri e insetti. È considerato un comportamento emergente derivante da semplici regole che vengono seguite dagli individui e non comporta alcun coordinamento centrale.

La scelta non è univoca e dipende dal contesto. Nel nostro caso, dal momento che desideriamo in primo luogo aumentare la velocità del programma, la prima soluzione è da scartare. La forbice delle prestazioni tra CPU e RAM fa sì che, se il programma è costretto a recarsi di sovente in memoria centrale, le prestazioni diminuiranno notevolmente. Le cache sono state introdotte per ovviare a tale problema. Il funzionamento di tale memorie è legato al principio di località (spaziale e temporale). Proprio per "assecondare" questo principio è necessario realizzare algoritmi che, una volta acceduto alla locazione x, andranno a visitare le celle nell'intorno. È quindi per tale motivo che la prima scelta risulta difficilmente realizzabile.

Il dubbio rimane tra la seconda e la terza soluzione. Per comprendere quale sia la scelta migliore sarà necessario comprendere quali tipi di operazioni dovrà eseguire l'algoritmo.

L'operazione più pesante pare essere, apparentemente, quella per il calcolo del vettore I e del baricentro, in entrambi i casi infatti siamo costretti ad accedere per intero alla matrice delle posizioni (o dei vettori spostamento nel caso di B). Se decidessimo di rappresentare per colonne le matrici tale operazione risulterebbe molto probabilmente ottimizzata.
Vediamo perché. Supponendo di indicare la matrice con "x" e il vettore per cui moltiplicarla con C.

$$V = \frac{\begin{bmatrix} x_{1,1} \\ x_{1,2} \\ \dots \\ x_{1,d} \end{bmatrix} \cdot C_1 \begin{bmatrix} x_{2,1} \\ x_{2,2} \\ \dots \\ x_{2,d} \end{bmatrix} \cdot C_2 \dots \begin{bmatrix} x_{np,1} \\ x_{np,2} \\ \dots \\ x_{np,d} \end{bmatrix} \cdot Cnp}{\sum_{i=1}^{np} C_i}$$

Ciò vuol dire che il generico elemento di V può essere ottenuto come:

$$V_j = x_{1,j} \cdot C_1 + x_{2,j} \cdot C_2 + \dots + x_{np,j} \cdot C_{np}$$

Accedere quindi a gli i-esimi elementi di ogni vettore posizione permetterebbe di calcolare l'i-esimo del vettore risultato. Se la matrice è rappresentata per colonne, evidentemente, è possibile accedere a locazioni di memoria prossime, mentre, in una rappresentazione per righe, ogni elemento i-esimo è distante dall'altro di "d" elementi.

Vi sono però altri fattori da prendere in considerazione, difatti, il calcolo di I e di B non è l'unica operazione degna di nota effettuata durante l'esecuzione del programma. Una di esse è il calcolo della distanza euclidea.
La distanza euclidea tra l'i-esimo pesce e il vettore y si ottiene come:

$$dist_i = \sqrt{\sum_{j=1}^{d} (x_{i,j} - y_j)^2}, i \in [1, ..., np]$$

Evidentemente, è necessario, per ogni pesce, scorrere l'intero vettore. Questo tipo di operazione è facilmente implementabile se la matrice delle posizioni è rappresentata per righe, mentre presenta delle difficoltà nella rappresentazione per colonne.

Altra parte essenziale dell'algoritmo è quella del movimento individuale, ove ogni posizione viene singolarmente modificata per simulare la ricerca causale dei pesci di una zona favorevole. Questa operazione è fortemente legata al singolo pesce. La rappresentazione per righe sarebbe anche in questo caso più indicata.

Similmente al calcolo del movimento individuale, ottenere il valore della funzione in un preciso punto è un'operazione intrinsecamente legata al singolo vettore, una rappresentazione per colonne risulterebbe anche in questo caso poco consigliabile.

Tirando le somme, una rappresentazione per colonne gioverebbe a:

- calcolo del vettore I (eseguito una volta per iterazione);
- calcolo del vettore B (eseguito una volta per iterazione).

In entrambi i casi le operazioni riguardano tutta la matrice delle posizioni (delle posizioni e delle variazioni delle posizioni).
Una rappresentazione per righe gioverebbe invece a:

- calcolo delle distanze euclidee
- calcolo della funzione
- movimento individuale

Anche in questi casi, le operazioni coinvolgono pressapoco tutta la matrice. Del resto, è evidente che, poiché l'algoritmo si pone come obiettivo la modellazione del comportamento dei pesci, la gestione isolata dei singoli vettori (e quindi la rappresentazione per righe) è probabilmente quella più naturale da adottare.

Dopo tali riflessioni si è infine giunti alla decisione di adottare una rappresentazione delle matrici per righe.

2.2 Algoritmi a basso livello

La scelta di quale funzione ottimizzare a basso livello è stata guidata principalmente dai seguenti fattori :

1. Molteplicità con cui la funzione appare nel codice
2. Complessità temporale della funzione
3. Facilità implementativa della funzione

Sulla base di tale criteri di scelta, si elencano di le funzioni che sono state oggetto di interventi di ottimizzazione, assieme alle relative descrizioni dei compiti svolti da queste ultime più le motivazioni giustificanti gli interventi ottimizzativi :

- ***void min_vector_32/64(VECTOR x, int n, type* min)***
  Descrizione
  Dato un vettore x di float/double, di generica dimensione n, ne trova il valore minimo, memorizzandolo in min (n.b. min è passato per riferimento).

  $$min = \min_{j \in \{0,..,n-1\}} x[j]$$

  Motivazione
  Il calcolo del minimo in un vettore è un'operazione di banale implementazione che permetterebbe sottili miglioramenti del tempo di esecuzione dell'algoritmo.

- ***void vector_sum_32/64(MATRIX x, int offset, int n, VECTOR V)***
  Descrizione
  Data una matrice x memorizzata per righe, (dove ogni riga, di dimensione n, corrisponde al vettore posizione del pesce i-esimo) e un vettore V, effettua la somma membro a membro tra i corrispondenti elementi di x e v, memorizzando infine il risultato in x.

  $$x[j + \text{offset}] = x[j + \text{offset}] + V[j]$$

  $$j \in \{0, .., n-1\}$$

  Lo scostamento offset individua un determinato pesce, o meglio il vettore n-dimensionale che rappresenta la posizione di quest'ultimo. Offset è ottenuto dal prodotto tra l'indice rappresentante il pesce i-esimo e n.

  $$\text{offset} = i \cdot n$$

  $$i \in \{0, .., n_p - 1\}$$

  Motivazione
  La somma vettoriale e' una delle operazioni più ricorrenti all'interno dell'algoritmo. Risulta di facile implementazione a basso a livello, ed è facilmente predisposta alla parallelizzazione sia mediante un approccio implicito che esplicito utilizzando il paradigma SIMD.

- ***void eval_f_32/64(MATRIX x, int d, VECTOR c, int offset, type* quad, type* scalar)***
  Descrizione
  Individuato il vettore posizione d-dimensionale in x tramite lo scostamento offset, valuta il valore della funzione obiettivo. La formula di f è presentata di seguito:

  $$f(x) = e^{x^2} + x^2 - c \cdot x$$

  Alla fine del conseguimento di tale operazione bisogna dapprima calcolare il quadrato di x, e il prodotto scalare tra c ed x, dove c e' un vettore di coefficienti di dimensione d. Queste due ultime operazioni sono state svolte a basso livello, con i relativi risultati memorizzati rispettivamente in quad e in scalar. L'esponenziazione invece, per via della complessità implementativa, è stata svolta direttamente in linguaggio C mediante la funzione expf(). Ciò che tale funziona restituisce è quindi

  $$expf(quad) + quad - scalar$$

  Motivazione
  Tra le cinque funzioni elencate, eval_f detiene il maggior numero di chiamate per ogni iterazione di fss (almeno al tempo di scrittura). Non sono inoltre banali i calcoli che hanno luogo in tale funzione, in particolare l'esponenziazione che vede come base il numero di Nepero. Come accennato precedentemente, il calcolo di

  $$e^{x^2}$$

  non e' stato implementato in assembly (o meglio e' stato calcolato il quadrato di x a basso livello, mentre l'elevamento a potenza con base e in alto livello) in quanto non sono stati ottenuti risultati soddisfacenti in termini di accuratezza del risultato e di velocità rispetto alla già implementata expf. Nonostante ciò, sono visibili miglioramenti nel tempo di esecuzione grazie all'elevato uso di eval_f nel codice.

- ***void euclidian_distance_32/64(MATRIX x, int offset, VECTOR y, int d, type* dist)***

Descrizione
Individuato il vettore posizione d-dimensionale in x mediante lo scostamento offset, calcola la distanza euclidea tra il vettore x ed y come segue

$$d(x,y) = \sqrt{\sum_{j=0}^{n-1} (x[offset + j] - y[j])^2}$$

memorizzando il risultato in dist.

Motivazione
La distanza euclidea viene calcolata, per ciascuna iterazione dell'algoritmo fss, tra ogni vettore posizione relativo all'i-esimo pesce e il baricentro (anch'esso un vettore) del banco dei pesci. Sebbene la funzione venga richiamata solo in movimento volitivo, l'aritmetica presente ha un suo costo. E' stato deciso per tale motivo una riscrittura in basso livello della funzione in questione.

- ***void compute_avg_32/64(MATRIX x, int np, int d, VECTOR c, type den, VECTOR ris)***
  Descrizione
  Calcola la media ponderata prendendo in considerazione i vettori posizione d-dimensionali di tutti gli np pesci secondo la formula

$$ris = \frac{\sum_{i=0}^{n_p-1} x[i] \cdot c[i]}{\sum_{i=0}^{n_p-1} c[i]}$$

  memorizzando il risultato (un vettore) in ris. Ciascun elemento del vettore c, c[i], individua il peso relativo all'i-esimo pesce. Lo scalare den rappresenta la sommatoria dei pesi dei pesci, pre-computata gradualmente in C prima della chiamata della funzione compute_avg_32/64.

  Motivazione
  La computazione in questione risulta particolarmente influente dal punto di vista temporale (racchiude al suo interno diverse operazioni aritmetiche basilari rispetto alla "semplice" somma vettoriale). È inoltre richiamata in due punti distinti dell'algoritmo, ovverosia movimento volitivo e movimento istintivo. È quindi chiaro che un'appropriata implementazione a basso livello di tale funzione, sebbene non immediata, gioverebbe a ridurre i tempi di esecuzione dell'algoritmo.

3 Soluzioni intermedie

Durante lo sviluppo del progetto sono state inevitabilmente realizzate diverse versioni, alcune funzionanti ed altre meno. Il resoconto di questo percorso è stato affidato ad una repository Git Hub, la quale, oltre che rappresentare un utile strumento di sviluppo, condivisione di idee e coordinamento all'interno del gruppo, permette ora di delineare un disegno molto accurato del progetto nel suo divenire.

Si è quindi scelto di aggiungere, oltre alla presente relazione, la stessa repository, consultabile al seguente link:

https://github.com/thatsimo/progetto-21-22

Di seguito sono esposti in modo sintetico i principali passi nella realizzazione del progetto.

3.1 Alto livello

In questa sezione è trattata l'implementazione dell'algoritmo FSS esclusivamente per ciò che riguarda l'uso di meccanismi di alto livello (i.e. nel solo linguaggio C).

3.1.1 Versione solo C

Il primo passo nella realizzazione del progetto è stato l'implementazione dell'algoritmo FSS. È stata quindi creata una versione in C senza richiamare alcuna funzione Assembly. Il funzionamento di tale programma è molto basilare, ci si è limitati a seguire lo pseudocodice del Fish School Search.

Sono state prese delle decisioni per un'ottimizzazione preliminare, oltre all'uso delle strutture dati passate in input si è definita una struct secondaria, "Support". Al suo interno sono presenti:

- **type Stepind_curr, stepvol_curr**: che contengono il valore di stepind e stepvol corrente per ogni iterazione;
- **VECTOR W** : un vettore di np elementi che contiene i pesi correnti dei pesci;
- **type w_sum**: per evitare di scorrere nuovamente il vettore W per ottenere la somma (necessaria per il calcolo del baricentro) w_sum è ricavato nella fase di alimentazione dei pesci. Corrisponde a:

$$\sum_{i=1}^{np} W[i] \tag{1}$$

- **VECTOR f_curr**: un vettore di np elementi, contenente il valore della funzione corrispondente alla posizione corrente dei pesci. Si è optato per questa scelta al fine di evitare di calcolare f(x) troppo spesso;

- **type f_sum**: il cui significato è simile a quello di w_sum, corrisponde a:

$$\sum_{i=1}^{np} f\_curr[i] \tag{2}$$

- **VECTOR delta_f**: un vettore contenente le variazioni della funzione all'iterazione corrente;

- **MATRIX delta_x**: contiene invece i vettori spostamento dei pesci in seguito al movimento individuale;

- **VECTOR x_new**: un vettore utilizzato per mantenere momentaneamente la nuova posizione del pesce nel movimento individuale. Ottenuta come:

$$y_i[j] = x_i[j] + rand(-1, 1) \cdot stepind, j \in [i, ..., d]$$

- **int r_i**: l'indice usato per scorrere il vettore dei numeri random passati in input;

- **VECTOR I** : un vettore di d elementi che rappresenta il vettore del movimento istintivo;

- **VECTOR B** : il vettore del baricentro.

3.1.2 Versione C migliorata

Dopo aver realizzato la prima versione in C si è apportato un ulteriore miglioramento, partendo dall'intuizione che il calcolo dei vettori B e I viene effettuato in modalità molto simili, si tratta infatti di una "specie" di media pesata applicata su ogni elemento del vettore. Partendo da questo presupposto, invece di manentere in Support due vettori separati li si è sostituiti con un unico vettore V, e il calcolo è stato affidato ad una funzione compute_weighted_avg, che può essere richiamata sia per il calcolo di I, che di B, naturalmente, con gli appositi parametri.

3.2 Basso livello

In questa sezione è trattato l'inserimento di funzioni Assembly per il miglioramento delle prestazioni del programma precedentemente realizzato in C.

3.2.1 Versione con funzioni in Assembly

Come già anticipato nei paragrafi precedente si è deciso di realizzare alcune parti del programma, già implementato in C, in assembly, le ragioni dietro questa scelta sono state già in parte delineate, si vorrebbe infatti, in tal modo, aumentare la velocità di esecuzione, potendo anche "approfittare" del set istruzioni **SSE** e **AVX**, repertori di tipo SIMD. In tal modo, siamo in grado di lavorare non con un singolo operando, ma con vettori di operandi (il cui numero dipende dal tipo di precisione scelta, e.g. float o double) e dal tipo di repertorio utilizzato, SSE rende disponibili registri a 128 bit, mentre AVX a 256. Il programma risulta quindi più veloce, di un fattore all'incirca pari a

$$\frac{NormalExecutionTime}{OperandPerRegister}$$

Ovviamente, maggiore è il numero di operandi all'interno del vettore, maggiore sarà lo *speedup* ottenuto.

3.2.2 Versione con unroll

Uno dei modi per aumentare l'efficienza di un programma nel contesto Assembly è quello di realizzare il code unrolling. Dalla teoria è infatti noto che, ogni volta che il processore incontra un salto, interviene la *branch prediction unit* che tenta di predire se il salto verrà o meno eseguito, in caso di fallimento la pipeline della macchina si svuoterà, conducendo ad una notevole perdita di prestazioni.

L'unrolling permette di "comunicare" indirettamente al processore che le istruzioni all'interno del loop verranno eseguite, in tal modo diminuiamo la probabilità di predizioni errate e di svuotare la pipeline, con conseguente perdita di prestazioni.

Si è scelto di dimensionare il valore dell'unroll ad un massimo di 8.

4 Gestione dei vettori non multipli

Come ricordato più volte, il repertorio SSE e AVX ci consente di lavorare con vettori di operandi, gli XMM possono contenere quattro float o due double mentre gli YMM otto float o quattro double. Quando andremo ad estrarre i dati dalla memoria per inserirli all'interno di tali registri ne prenderemo n alla volta, tale situazione è ottimale e non conduce ad alcun problema di sorta nel caso in cui la struttura dati (e.g. un

vettore) dalla quale stiamo estraendo gli operandi ha una dimensione multipla di n. Nel caso in cui questo non si dovesse verificare si presenta, evidentemente, un problema da risolvere.

Se decidessimo di eseguire un normale for del tipo

```
for(int i=0;i<n;i+=4) {
    ...
}
```

Potrebbe verificarsi un errore. Prendiamo in esame il seguente caso.

```
Supponiamo n=7, quindi, non multiplo di 4
```

Iterazione	i
0	0
1	4

Il valore di i non aumenta ulteriormente, in quanto non può superare il vincolo ”iin”. Tuttavia, poiché ad ogni iterazione si accede a quattro elementi accadrà questo.
Dato il seguente vettore

$$X = \begin{bmatrix} x[0] & x[1] & x[2] & x[3] & x[4] & x[5] & x[6] \end{bmatrix}$$

Alla prima iterazione, essendo i=0 si accede agli elementi di X che vanno da X[0] a X[3].Alla seconda iterazione invece, poiché i=4 si accederà agli elementi di X che vanno da X[4] a X[7], è però chiaro che in questo modo usciremo da X, causando un errore di segmentazione.

Per evitare problemi di tale specie, si può riscrivere il for in questo modo:

```
for(int i=0;i<n-3;i+=4) {
    ...
}
```

Ed in generale, se ad ogni iterazione si accede a **q** elementi del vettore:

```
for(int i=0;i<n-(q-1);i+=q) {
    ...
}
```

Ponendo il vincolo su ”i” eviteremo di uscire dal vettore, in più, sarà possibile gestire separatamente gli elementi restanti, se presenti, come semplici scalari.

5 Versioni finali

Dopo aver vagliato i miglioramenti esposti nella sezione precedente si è infine giunti a una versione definitiva del progetto. La traccia prescriveva di realizzare 4 versioni in totale:

1. una versione con operandi a 32 bit (i.e. *float*) con funzioni nasm scritte avvalendosi del repertorio SSE;

2. una versione con operandi a 32 bit facente uso di **openMP**;

3. una versione con operandi a 64 bit (i.e. *double*) con funzioni nasm scritte avvalendosi del repertorio AVX;

4. una versione con operandi a 64 bit facente uso di openMP.

Lo scheletro di tutte e quattro le versioni rimane invariato e si basa sulla versione C migliorata esposta in precedenza. Difatti, per passare dalla versione a 32 bit a quella a 64 è stato necessario cambiare solo la dichiarazione di ”type” (float/double). Per quanto riguarda invece l’uso delle funzioni in nasm, visto il cambio di repertorio, è stato necessario ridefinirle nell’apposito file .nasm, tuttavia, per ciò che riguarda il solo file .c, è bastato modificare la terminazione dei metodi dichiarati **extern**, da _32 a _64.

Per ciò che concerne la versione OMP, è stato sufficiente aggiungere le direttive **”#pragma OMP parallel for”** nei punti più adatti del C. In alcuni casi è stato necessario modificare alcune parti del codice per evitare si verificassero race contition sulle variabili condivise.

5.1 x86-32+SSE

Come esposto in precedenza le funzioni implementate in Assembly sono:

1. **void min_vector_32(VECTOR x, int n, type* max);**

2. **void euclidian_distance_32(MATRIX x,int offset,VECTOR y,int d,type* dist);**

3. **void eval_f_32(MATRIX x, int d, VECTOR c, int offset,type* quad, type* scalar);**

4. **void compute_avg_32(MATRIX x, int np, int d, VECTOR c,type den, VECTOR ris);**

5. **void vector_sum_32(MATRIX x, int offset, int n,VECTOR V);**

Dettagliamo quindi il comportamento di ogni funzione.

compute_avg_32
Come descritto nella sezione relativa alle scelte progettuali, questa è l'unica funzione la cui implementazione è influenzata negativamente dalla rappresentazione per righe.
La funzione va quindi ad accedere progressivamente agli elementi di ogni vettore posizione e inserisce i risultati parziali nella locazione di memoria del risultato V.

$$V+ = XMM \leftarrow x[i] \cdot \frac{c[i]}{\sum_{j=1}^{np} c[j]}, i \in [1, ..., np]$$

Evidentemente, in questo modo ogni volta che viene calcolato il prodotto tra il vettore posizione e l'elemento i'esimo del vettore c, il risultato deve essere memorizzato in V (sommando il valore aggiunto con quello già presente). Ciò comporta un numero elevato di accessi in memoria.

L'algoritmo scelto opera nel seguente modo:

```
void compute_avg_32(MATRIX x, int np,
int d, VECTOR c,type den, VECTOR ris)
{

    XMM0 ← den;

    • Replica il denominatore
      su tutto XMM0

    for(int i=0;i<np;i++) {
        XMM7 ← C[i]

        • replica C[i]
          su tutto XMM7

        XMM7 ← XMM7/XMM0

        EDX ← i · d
        for(int j=0;j<d-3;j+=4) {
            XMM1 ← x[EDX+j]
            XMM1 ← XMM1 · XMM7
            x[EDX+j]+=XMM7
        }
        • Gestione vettore non
          multiplo di 4

    }
}
```

vector_sum_32
In questo caso, è necessario accedere ad ogni vettore posizione e sommargli il vettore I passato come parametro alla funzione. A differenza del caso precedente questa operazione risulta essere più efficiente in una rappresentazione per righe, è infatti necessario andare ad accedere agli elementi del vettore posizione i-esimo e sommargli quelli di I. Evidentemente, l'approccio per righe è più naturale in questo caso.

$$\bar{x}[i]+ = XMM \leftarrow \bar{x}[i] + I, i \in [1, ..., np]$$

In una rappresentazione per colonne, sarebbe necessario aggiornare il vettore posizione elemento per elemento. Nell'approccio per righe è possibile invece approfittare del parallelismo SIMD, riducendo così il numero di accessi alla memoria. all'interno del medesimo vettore XMM si mantiengono progressivamente i 4 valori minimi, ogni volta che si accede ai successivi elementi si confrontano con quelle nel vettore XMM ed eventualmente si aggiiornano.

min_vector_32
Questa operazione non è né avvantaggiata né svantaggiata dalla rappresentazione della matrice, in quanto la struttura dati a cui accediamo è un vettore. Grazie al set SSE è però possibile eseguire parallelamente su più operandi per volta.

```
void min_vector_32(VECTOR x, int n,
type* min) {

    XMM0 ← x[0]

    for(int i=4;i<np-3;i+=4) {
        XMM1 ← x[i]
        XMM0 ← MIN(XMM0,XMM1)
    }

    • riduzione di XMM0
```

```
    • Gestione vettore non
          multiplo di 4

    [min] ← XMM0

}
```

eval_f_32
Anche calcolare la funzione è un processo avvantaggiato dalla rappresentazione per righe, è infatti evidente che tale calcolo comprende il singolo vettore. Si mantengono quindi i due valori "quad" e "scalar" in due vettori XMM. Il funzionamento dell'algoritmo è riassumibile come di seguito.

```
void eval_f_32(MATRIX x, int d,
VECTOR c, int offset,type* quad,
type* scalar) {

    XMM0 = 0̄
    XMM1 = 0̄

    for(int i=0;i<d-3;i+=4) {
        XMM2 ← x[offset+i]
        XMM3 ← XMM2
        XMM2 ← XMM2 · XMM2 ; x[i]^2
        XMM0+=XMM2

        XMM4 ← c[i]
        XMM3 ← XMM3 · XMM4 ; x[i] · y[i]
        XMM1+=XMM3

    }

    • gestione vettore non
        multiplo di quattro

    [quad]   ← XMM0
    [scalar] ← XMM1


}
}
```

euc_dist_32

Come nei casi precedenti e come anticipato nella fase di progettazione, la distanza euclidea è un'altra di quelle operazioni avvantaggiate dalla rappresentazione per righe, dovendo, anche in questo caso, accedere progressivamente al vettore posizione. Se avessimo utilizzato quella per colonne avremmo dovuto aggiornare progressivamente i valori in memoria, e quindi un calo nelle prestazioni.
La struttura dell'algoritmo può essere così riassunta.

```
void euclidian_distance_32(MATRIX x,
int offset,VECTOR y,int d,type* dist)
{

    XMM0=0̄

    for(int i=0;i<d-3;i+=4) {
        XMM1 ← x[offset+i]
        XMM2 ← y[i]
        XMM1 ← XMM1-XMM2   ;(x[i] − y[i])
        XMM1 ← XMM1 · XMM1 ;(x[i] − y[i])^2
        XMM0+=XMM1
    }

    • Riduci XMM0

    • Gestione vettore
      lunghezza non multipla
      di quattro

    XMM0 ← √(XMM0)

    [dist] ← XMM0
}
```

Riduzione dei vettori
Sebbene il repertorio SSE permetta di operare con più operandi alla volta, molto spesso è necessario restituire un unico valore. Per questo motivo è necessario ridurre i vettori.

Nel caso di SSE ciò è realizzabile nel seguente modo:

```
MOVAPS XMM1,XMM0
SHUFPS XMM1,XMM0,00001110b
<oper> XMM0,XMM1
MOVAPS XMM1,XMM0
SHUFPS XMM1,XMM1,00000000b
<oper> XMM0,XMM0
```

Supponendo che il vettore da ridurre sia XMM0, e che su di esso debba essere applicata l'operazione <oper>, che nel caso del prodotto scalare è ADD e nel caso di min_vector MIN.

5.2 x86-32+AVX

La versione a 64 bit con repertorio AVX non si discosta molto dalla versione 32 bit con SSE, ovviamente è stato necessario modificare le istruzioni in modo tale che queste utilizzassero AVX, tuttavia, le differenze non sono sostanziali, la struttura degli algoritmi è rimasta praticamente la stessa.

In particolare, per evitare il context switch dovuto al passaggio da SSE ad AVX si è scelto di usare operazioni riguardanti solo l'ultimo repertorio, anche quando si sono usati i registri XMM di SSE sono state impiegate le istruzioni AVX, segnalate dalla "V" iniziale.

Un altro aspetto degno di nota riguarda la riduzione dei vettori, non è infatti possibile utilizzare l'approccio mostrato sopra. Si è quindi adottata la seguente strategia:

```
VHADDPD      YMM0,YMM0,YMM0
VPERM2F128   YMM1,YMM0,YMM0,00010001b
VHADDPD      XMM0,XMM1
```

Oppure, nel caso in cui è necessario trovare il minimo nel vettore

```
VEPERM2F128 YMM1,YMM1,YMM1,00010001b
VMINPD      YMM0,YMM1
VMOVAPD     XMM1,XMM0
VSHUFPD     XMM0,XMM0,01b
VMINPD      XMM0,XMM1
```

Avendo a disposizione i vettori estesi a 256 bit, e lavorando con operandi a 64 bit, il numero di elementi per vettore rimane lo stesso rispetto al caso sopra analizzato in SSE, quattro float nel caso precedente e quattro double nel caso attuale.

Come ulteriore nota, è qui necessario ricordare che a differenza di SSE in cui è possibile ottenere i parametri passati alla funzione tramite il registro **ESP** (si trovano quindi sullo stack), in AVX i parametri sono consegnati alla funzione tramite degli appositi registri, il cui uso e funzionamento è richiamato nelle *C-calling-conventions*.

5.3 openMP

L'utilizzo di openMP permette di impiegare nell'esecuzione dei nostri programmi dei thread hardware. È molto importante in questa fase, quindi: riuscire a ben bilanciare il carico di lavoro; evitare delle race condition quando è necessario accedere a delle variabili condivise; garantire un uso corretto del multithreading (i.e. far sì che il costo di creazione non superi il guadagno in termini di prestazioni. Dovremmo evitare che i thread eseguano poche operazioni).

A tal fine sono stati individuati nel codice i punti candidati all'introduzione del parallelismo. Ciò è stato fatto sia andando a trovare quelle zone che, proprio per la natura dell'algoritmo eseguito, erano naturalmente portate al multi-threading, sia eseguendo delle analisi sul codice, per identificare quelle parti sottoposte ad un maggior "stress". Il report ottenuto per una generica esecuzione del programma è mostrato di seguito.

% time	calls	name
40.02	16000	individual_movement
40.02	250	volitive_movement
10.00		fori_euc
10.00		vector_sum_32
0.00	16064	evaluate_f
0.00	250	alimentation_operator
0.00	250	instincitve_movement
0.00	250	update_parameters
0.00	10	_mm_malloc
0.00	10	get_block
0.00	6	alloc_vector
0.00	4	alloc_matrix
0.00	3	load_data
0.00	1	find_and_assign_minimum
0.00	1	fss
0.00	1	initialize_support_data_stuct
0.00	1	save_data

Si precisa che %time denota la percentuale del tempo totale di esecuzione del programma utilizzato dalla funzione.

È evidente che le funzioni più pesanti sono: il movimento individuale e quello volitivo. Sarà quindi su queste parti del codice che cercherà di concentrarsi la nostra opera di parallelizzazione.

Un altro punto che probabilmente verrà preso in considerazione è il movimento istintivo, difatti, oltre

che il calcolo del vettore I, è necessario eseguire la somma:

$$x_i + I, \quad \forall i \in [1, ..., np] \tag{3}$$

Ogni somma accede in scrittura a regioni diverse della matrice x, è quindi molto semplice parallelizzare.

Si è deciso quindi di parallelizzare: l'esecuzione del movimento individuale, assegnando ad ogni thread un certo numero di pesci; il movimento volitivo, andando a suddividere qui solo lo spostamento, descritto dalla formula:

$$x_i = x_i + \pm ste_vol \cdot rand(0,1) * \frac{x_i - \mathbf{B}}{dist(x_i, \mathbf{B})}$$

dopo aver calcolato il baricentro. Si è anche parallelizzata la parte del movimento istintivo mostrata sopra, sebbene, visti i dati del report, molto probabilmente questa non condurrà a evidenti migliorie.

L'utilizzo di openMP si è limitato all'inserimento delle direttive "#pragma omp parallel for" nei punti sopra discussi.

Per riuscire nel nostro intento, ed in particolar modo per la parallelizzazione del movimento individuale e volitivo è sorta la necessità di modificare opportunamente il codice a nostra disposizione, questo per rispettare i punti sopra elencati.

Per tale motivo, la parte del nostro codice parallelizzata è stata inserita in una funzione apposita. L'aspetto del codice è così riassumibile.

```
#pragma omp parallel for
for(...) {
    inner_function(...);
}
```

Inoltre, si è prestata particolare attenzione ad eventuali variabili condivise che avrebbero potuto far nascere delle race condition.

In entrambi i casi le funzioni accedono al vettore dei numeri random. Per permettere lo scrorrimento dell'indice r_i tale accesso è così gestito.

```
...input->r[sup->r_i++]...
```

In questo modo, però, r_i verrebbe acceduto in scrittura da più thread contemporaneamente. Per evitare questo scenario, si è deciso di accedere al vettore "r" utilizzando degli indici che di volta in volta vengono messi a disposizione dalla funzione, ad esempio:

```
...input->r[sup->r_i+i]...
```

e aggiornare il valore dell'indice solo alla fine, sulla base di quanti valori random sono stati infine consumati (e.g. il movimento volitivo ne consuma "np" per iterazione, l'individuale "np · d").

Altra potenziale causa di race condition relativa al movimento individuale è relatica all'uso di "f_sum", questa variabile viene calcolata progressivamente, ogni volta che un nuovo Δf_i viene ricavato. Utilizzando OMP questa ottimizzazione non può più essere realizzata in quanto genererebbe inconsistenze. Per tale motivo, il calcolo di "f_sum" è spostato a dopo la fine del movimento individuale, sul vettore Δf

6 Test delle prestazioni

All'interno di questa sezione sono presentati i risultati degli esperimenti portati avanti sulle varie versioni dell'implementazione dell'FSS. In particolar modo:

- La versione solo C a 32 bit
- La versione a 32 bit con funzioni NASM
- La versione a 32 bit con funzioni NASM e UNROLL
- La versione 32 bit con openMP
- La versione solo C a 64 bit
- La versione a 64 bit con funzioni NAMS
- La versione a 64 bit con funzioni NASM e UNROLL
- La bersione a 64 bit con openMP
- Ulteriori versioni intermedie che fanno uso solo di alcune funzioni NASM

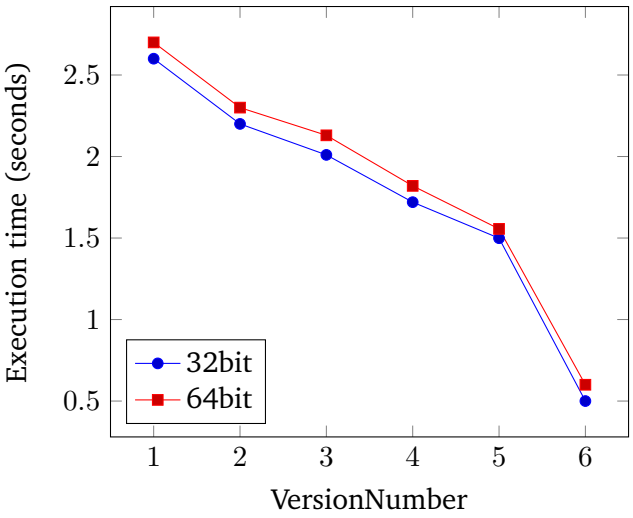
Il primo test è stato eseguito considerando un numero di pesci pari a 64, ognuno avente 8 dimensioni. L'algoritmo verrà eseguito per 250 iterazioni.

I	C-only	NASM	UNROLL
32 bit	0.009 s	0.0069 s	0.006 s
64 bit	0.007 s	0.0065 s	0.006

Come è evidente dai dati, i parametri scelti non sono adatti per apprezzare al meglio le diverse versioni e i

miglioramenti introdotti tra l'una e l'altra, soprattutto per quanto riguarda l'utilizzo di openMP, difatti, in quest'ultimo caso dobbiamo assicurarci che l'overhead dovuto alla creazione dei thread non sia superiore al tempo operativi di questi ultimi. Sebbene sia chiaro che i tempi tendano a diminuire, è necessario eseguire ulteriori test. É stato quindi eseguito un nuovo esperimento, questa volta scegliendo un dataset più grande. L'algoritmo è stato lanciato considerando 735 pesci con 125 dimensioni ciascuno. Il numero di iterazioni scelto è 750.
I dati ottenuti sono riportati nella tabella sottostante.

#	**Version name**	**32 Bit**	**64 Bit**
1	C-only	2.6 s	2.7 s
2	C + eval_f	2.2 s	2.3 s
3	C + eval_f + compute_avg	2.01 s	2.13 s
4	NASM	1.72 s	1.82 s
5	UNROLL	1.499 s	1.556 s
6	OMP	0.5 s	0.6 s



Per una maggiore chiarezza, le versioni che presentano nel nome **+ function_name** indicano che la funzione **function_name** è stata implementata a basso livello. La versione NASM comprende un'implementazione a basso livello di tutte le funzioni discusse in 2.2.

Il divario in termini di prestazioni è qui mostrato con maggiore nitidezza. L'introduzione delle funzioni nasm e del multi-thread rappresentano senza ombra di dubbio gli elementi più importanti.

I tempi ottenuti rappresentano naturalmente una stima, passare da un macchina all'altra potrebbe condurre ad ottenere valori completamente diversi (in teoria però, proporzionali).

Importante è stata anche l'introduzione dell'UNROLL, che permette di aumentare ulteriormente le prestazioni del programma.

7 Conclusione

In conclusione, la realizzazione di questo progetto ha dato a ogni membro di questo gruppo la possibilità di realizzare in prima persona quanto importante e determinante sia la conoscenza dei meccanismi a basso livello dei moderni calcolatori. Ancora oggi tali informazioni permettono di sfruttare al meglio le risorse hardware. I processori a nostra disposizione, infatti, rivelano il loro completo potenziale non più su dei semplici programmi sequenziali, ma su di un codice che implementa il parallelismo SIMD e MIMD. Tali tecniche non sono praticabili direttamente dalla CPU (come l'ILP) ma necessitano di un software scritto in modo adeguato. La comprensione di questi aspetti dell'informatica risulta quindi fondamentale e questa esperienza ci ha permesso di mettere in pratica tali conoscenze, che altrimenti sarebbero rimaste relegate al mondo della teoria.

In riferimento al lavoro presentato, la sola versione scritta in C presentava delle performance che, messe a confronto con la versione finale, ne mettono in evidenza l'inadeguatezza e la necessità di un'attenta progettazione volta alla ricerca delle sezioni del programma da riscrivere a basso livello. Durante la fase di analisi del problema è risultata determinante anche la scelta di come rappresentare i dati in memoria, al fine di ridurre il numero di accessi alla RAM, sfruttando al massimo le memorie cache.

Come già anticipato, il parallelismo MIMD è stato implementato tramite openMP, introducendo le direttive pragma. Questo strumento incredibilmente potente ha rappresentato un ulteriore passo avanti per il miglioramento delle prestazioni del programma, tuttavia, ha richiesto un ulteriore sforzo per evitare situazioni concorrenza tra i thread.