

Relazione Progetto

Architetture e Programmazione

dei Sistemi di Elaborazione

**Gruppo 17**

Danny Di Cello 224527

Raffaele Miriello 227564

Giuseppe Seminara 227540

Davide Tomo 193055

**Sommario**

[1. Conversione Data Set 3](#_Toc62404679)

[1.1 La funzione “*estendi*” 5](#_Toc62404680)

[1.2 Valutazione delle prestazioni 7](#_Toc62404681)

[2. Discesa Stocastica del Gradiente 9](#_Toc62404682)

[2.2 La funzione “*prodottoScalare*” 12](#_Toc62404683)

[2.3 La funzione “*aggiornaSomma*” 13](#_Toc62404684)

[2.4 La funzione “*aggiornaTheta*” 13](#_Toc62404685)

[2.5 La funzione “*aggiornaThetaAdagrad*” 14](#_Toc62404686)

[3. Ottimizzazione OpenMP 15](#_Toc62404687)

# Conversione Data Set

L’algoritmo si divide in una funzione “*convert\_data*” e una funzione “*estendi*”, dove la prima si occupa di fornire gli input presenti nel dataset alla seconda, la quale svolge l’effettiva conversione.

Le scelte progettuali sono state discusse e concordate dall’intero team, di seguito verranno illustrati i punti più rilevanti del processo quali la scelta dell’algoritmo, la sua implementazione e ottimizzazione e i risultati in fase di testing.

Per effettuare la conversione dal vettore x al vettore x\* è necessario calcolare, dato un grado massimo preso da input, tutti i possibili polinomi di grado inferiore o uguale al massimo che hanno come variabili le componenti del vettore x. Risulta quindi utile prevedere quante possibili combinazioni di variabili esistano in funzione della dimensione del vettore e del grado scelto.

A tal proposito, viene proposta la seguente dimostrazione, basata sull’approccio “*stars and bars*”.

*Sia G il grado massimo scelto. Sia N la dimensione del vettore. Siano il grado di ogni variabile del vettore. Il nostro interesse è conoscere quante sono le possibili somme .*

È possibile rappresentare le N variabili dividendo un segmento con N-1 “bars” o separatori. È possibile distribuire il grado tra le porzioni di questo segmento rappresentandolo con G stelline.

Ad esempio, tutte le possibili distribuzioni di grado 2 su 3 variabili corrispondono alle seguenti espressioni, ognuna anagramma dell’altra:

||\*\*

|\*|\*

\*||\*

|\*\*|

\*|\*|

\*\*||

Chiaramente, i separatori evidenziano tre possibili “contenitori” distinguibili, mentre le stelline non sono distinguibili. Tuttavia, è più semplice vedere il problema come inserimento delle barre tra le stelline, in particolare se le stelline sono uguali a G, ciascuna barra può essere inserita in G+1 posizioni. Di conseguenza, per ogni anagramma devono essere scelti N-1 spazi tra le stelline, potendo scegliere tra G+1 posizioni.

Dunque, la risposta al quesito iniziale è:

dove **C** rappresenta il coefficiente binomiale.

Si noti che il nostro scopo non è conoscere le possibili somme uguali a G, ma minori o uguali a G. Di conseguenza, è necessario un separatore in più dove inserire stelline, dove è compreso tra . Così facendo, è possibile spostare dalla porzione di segmento al di là del separatore una stellina per volta, consentendole il movimento sul resto del segmento, rappresentando così tutte le possibili somme di grado come illustrato di seguito:

In conclusione, i parametri da passare ad una funzione di calcolo del coefficiente binomiale saranno , o equivalentemente

## 1.1 La funzione “*estendi*”

Per semplificare le operazioni di calcolo si è scelto di utilizzare un vettore di dimensione uguale al grado massimo come struttura di supporto. Successivamente, nel vettore convertito vengono inseriti gli elementi banali, ovvero il valore 1 corrispondente al grado 0 e gli elementi originali di x corrispondenti al grado 1. A questo punto, se il grado massimo scelto in input è maggiore di 1, si svolge un ciclo che va da 2 a grado massimo attraverso cui vengono manipolate un numero crescente di celle del vettore ausiliario.

Lo scopo del vettore ausiliario, infatti, è contenere gli indici delle componenti del vettore iniziale. Per rappresentare la componente elevata alla quarta, ad esempio, le prime 4 celle del vettore ausiliario conterranno l’indice 0 e le rimanenti celle non verranno prese in considerazione. Per rappresentare il prodotto della componente ed , le prime due celle conterranno invece gli indici 0 ed 1. Ad ogni variazione del vettore ausiliario, viene eseguito un ciclo su quest’ultimo, allo scopo di calcolare il prodotto delle corrispondenti in x ed inserire il risultato nel vettore convertito.

Per rendere più chiara l’idea, di seguito vengono mostrati gli stati del vettore ausiliario nel caso in cui da input venga assegnato un vettore di dimensione 3 e grado massimo 3:

**Prima iterazione:**

Grado corrente (i) = 2

Indice prossimo elemento vettore convertito (*ind*) = 1 + dimensione vettore (*lenX*)

Vettore ausiliario = \*

Valore del prossimo elemento del vettore convertito ;

;

\* il vettore ausiliario è inizializzato a 0, ma per il momento l’ultimo elemento non è soggetto a modifiche

**Successivi stati del vettore ausiliario:**

Una volta che l’elemento “corrente” del vettore non può essere più incrementato perché è uguale a lenX-1, viene incrementato il valore dell’elemento alla sua sinistra.

Per evitare casi di ridondanza, se , deve essere .

sono infatti lo stesso elemento del vettore convertito.

Una volta che , viene incrementato il grado corrente

La variabile *ind* è stata incrementata ad ogni modifica di aux ed il risultato corrispondende è stato salvato nel vettore convertito.

Con lo stesso meccanismo mostrato in precedenza, si hanno i seguenti stati di aux:

Mentre sarà di volta in volta uguale a

Di seguito il codice C della funzione “*estendi*”, dove viene implementato l’algoritmo scelto:



## 1.2 Valutazione delle prestazioni

In entrambe le versioni richieste 32 e 64 bit è stato implementato lo stesso algoritmo di conversione nel linguaggio C. Il tempo medio di esecuzione riscontrato per questa parte è di 3 ms per entrambe le versioni. Questo tempo è stato ottenuto eseguendo l’algoritmo sul dataset di test fornito che contava la presenza di 2000 osservazioni delle coppie (X, Y).

Vista l’elevata efficienza dell’algoritmo di conversione (ordine dei millisecondi), rispetto alla parte dedicata al SGD (ordine dei decimi di secondo), non sono state effettuate particolari modifiche per ottimizzare il codice, tuttavia è opportuno effettuare alcune considerazioni, rispetto alle possibili ottimizzazioni:

* **Ottimizzazioni SIMD-Based**: si basano sulla presenza di registri vettoriali, permettendo di eseguire la stessa istruzione su più dati differenti. Per quanto riguarda l’algoritmo di conversione, esso è pensato per lavorare su singoli elementi, piuttosto che su vettori, quindi non è stata implementata questo tipo di ottimizzazione, poiché non si avrebbe avuto nessun tipo di vantaggio.
* **Cache-Blocking**: sfrutta le gerarchie delle memorie per evitare il cache missing. La funzione estendi è stata pensata per sfruttare questo tipo di ottimizzazione, in particolare la singola chiamata alla funzione estendi, lavora su una singola osservazione di X, aumentando così la probabilità che esso sia presente in cache, riducendo quindi i cache missing, e di conseguenza i tempi di accesso alla memoria.
* **Loop unrolling**: è un meccanismo di ottimizzazione per spingere il processore a parallelizzare un gruppo di istruzioni sequenziali. Anche in questo caso la funzione estendi è stata pensata per sfruttare un eventuale loop unrolling, trasformando, ad esempio il ciclo:



nel seguente:



Tuttavia, le diverse prove effettuate, usando diversi fattori di unrolling, ad esempio 4 e 8, non hanno mostrato nessun miglioramento delle prestazioni, in alcuni casi abbiamo addirittura riscontrato un peggioramento delle prestazioni di 2ms, rispetto alla versione senza loop unrolling. Per questo motivo abbiamo deciso di non implementare il loop unrolling su questa porzione di codice.

# Discesa Stocastica del Gradiente

Per la discesa stocastica del gradiente era richiesta l’implementazione dell’algoritmo **SGDBATCH** con annessa accelerazione **Adagrad**. Come consigliato nelle specifiche del progetto è stato prima codificato l’algoritmo interamente in linguaggio C cercando di mantenere il codice il più modulare possibile tramite l’utilizzo delle chiamate a funzione. Questo ha consentito di poter andare a lavorare sulle singole funzioni, cercando di ottimizzare quelle che risultavano più pesanti per il calcolatore.

In una prima versione, è stato implementato SGD, senza tener conto dell’accelerazione Adagrad, seguendo lo pseudo codice fornito nelle specifiche.



Come si evince dalla porzione di codice, è stato sfruttato, utilizzando il metodo di allocazione di memoria già presente nello “scheletro” fornitoci, l’upper-bound *t*, calcolato nella fase di conversione del dataset, per creare il vettore *theta*.

I valori dei parametri theta vengono aggiornati tramite la chiamata al metodo "*calcoloNuovoTheta*" il quale lavora con il batch corrente, la cui dimensione è *v*.



Essendo una fase inziale, sono presenti chiamate a funzioni a cascata, poiché si è preferito isolare le operazioni da dover compire per il calcolo di theta, al fine di individuare i possibili errori di calcolo che ci sarebbero potuti essere.

Le funzioni presenti sono: “*mediaPesata*”, “*sottrazioneVettori*” e “*prodottoScalareVettore*”.

Già dal nome si denotano le operazioni che svolgeranno.

Particolare attenzione bisogna riporla nella funzione “*mediaPesata*”, questo perché racchiude la gran parte del calcolo effettivo.



Anche all’interno di questa funzione vengono utilizzate delle procedure ausiliarie, le quali, hanno come scopo il calcolo della sommatoria presente nella formula del metodo SGDBATCH qui sotto riportata.

I tempi ottenuti in seguito ai primi test effettuati sul codice non erano molto promettenti, per cui si è deciso di accorpare i metodi le cui operazioni svolte potevano essere eseguite nello stesso ciclo di lettura del data set.

In particolare, la struttura dei metodi “*sgd*” e “*mediaPesata*” è rimasta quasi identica alla prima versione, mentre la maggior parte del costo computazionale è stata concentrato nei metodi “*aggiornaSomma*” e “*prodottoScalare*”.



In seguito a queste modifiche si è riusciti ad ottenere dei tempi di esecuzione più bassi i quali però ancora non soddisfavano le aspettative.

Ancor prima di addentrarci nella fase di ottimizzazione è stata introdotta la gestione dell’accelerazione Adagrad all’interno del metodo “*sgd*”.



Eccezion fatta per la chiamata alla funzione “*mediaPesataAdagrad”*, “*calcoloNuovoThetaAdagrad”* presenta la stessa struttura della versione priva di accelerazione.

Di seguito vengono mostrate le differenze sostanziali presenti nel calcolo dei parametri theta, motivo per il quale è stato necessario scindere le due funzioni.

Dopo aver constatato il corretto funzionamento del codice c, è stato utilizzato il **profiler** al fine di analizzare la complessità spaziale e temporale del software e di supportare il processo di ottimizzazione.

Immagine che contiene testo, calcolatrice

Descrizione generata automaticamente

Dalla sua esecuzione sono emerse le funzioni i cui tempi di esecuzione aggravavano le prestazioni dell’intero programma. Partendo da questi risultati si è cercata una strategia che permettesse di abbattere tali tempistiche.

Nello specifico, sono state tradotte “*aggiornaSomma*” e “*prodottoScalare*” nel loro corrispettivo codice assembly, sfruttando la possibilità di invocare procedure scritte in linguaggio assembly all’interno del codice C stesso.

## 2.2 La funzione “*prodottoScalare*”

È stata la prima funzione ad essere ottimizzata data la sua semplicità e la sua importanza all’interno dell’intero progetto. Sia nella versione a 32 bit che nella versione a 64 bit è stata sfruttato il parallelismo offerto dai repertori di istruzioni SSE e AVX, rispettivamente.

La differenza che sussiste nella versione a 64 bit, rispetto quella a 32, è la necessità di utilizzare un registro intermediario per compiere l’ultima operazione di somma tra le due sotto parti del registro YMM0. Questo poiché la VPERM2F128 lavora con blocchi da 128 bit e quindi permuta 2 double per volta, cosa non utile se non per scambiare di posizione i double della prima parte del registro YMM1 con quelli della seconda parte così da poter effettuare la somma con YMM0.

A seguito di questa operazione si preleva il valore da XMM0 e lo si salva all’indirizzo a cui punta il registro R9 (indirizzo della variabile passata dal C).

## 2.3 La funzione “*aggiornaSomma*”

In seguito all’analisi della complessità temporale si è deciso di andare ad ottimizzare questa porzione di software, la quale ha il compito di calcolare i singoli termini della sommatoria utile all’aggiornamento dei parametri theta.

Poiché la funzione accede ai dati contenuti nel vettore “*x\_ast*” si è pensato di sfruttare tutte quelle operazioni che appartengono al repertorio di istruzioni “SSE”, le quali consentono di lavorare con quattro elementi per volta, nel caso di dati di tipo float.

Il tempo di esecuzione a seguito della riscrittura del codice in assembly è sceso a 0,5 s, questa miglioria è dovuta alla possibilità di eseguire più operazioni elementari in parallelo.

## 2.4 La funzione “*aggiornaTheta*”

Nell’ultima versione “*aggiornaSomma*” è stata sostituita dalla funzione “*aggiornaTheta*”, in seguito ad un’osservazione effettuata sulla formula di aggiornamento dei parametri theta.

In precedenza, veniva calcolato il vettore risultante dalla sommatoria e successivamente si effettuava la sottrazione con i valori di theta del passo precedente. Questo tipo di procedimento costava una lettura in più del vettore, la quale incideva sulle performance del software.

Si è preferito quindi sottrarre i valori di ogni somma parziale direttamente al vettore theta, questo ci ha permesso di mantenere il corretto funzionamento ma allo stesso tempo di guadagnare in termini di tempo di esecuzione.

Di seguito viene mostrata la versione finale del codice.



Per quanto riguarda la sua versione a 64 bit è stata sviluppata in modo del tutto similare, sfruttando però le istruzioni del repertorio **AVX**.

In seguito ad alcuni test di esecuzione effettuati sul dataset di prova è stato ottenuto un tempo medio di 0,230 s.

## 2.5 La funzione “*aggiornaThetaAdagrad*”

La scrittura di questa porzione di codice è avvenuta successivamente alla scrittura della versione senza accelerazione. Le differenze nel calcolo sono dovute all’utilizzo della matrice diagonale G, la quale di volta in volta deve essere aggiornata per ogni vettore x appartenente al data set.

La matrice è stata trattata come se fosse un vettore, salvando esclusivamente i valori contenuti nella diagonale, questo ha consentito un risparmio in termini di complessità spaziale.

Anche in questo caso il codice a 32 bit ed il codice a 64 bit sono del tutto speculari.

# 3. Ottimizzazione OpenMP

Tra le specifiche di progetto era richiesta una versione del software che utilizzasse le istruzioni OpenMP, le quali consentono al programmatore attraverso direttive al compilatore di introdurre parallelismo nel codice.

In particolare, nella versione apposita del software sono state inserite delle clausole, ove possibile, nei metodi principali.

Di seguito vengono riportate le parti di codice interessate.



La prima direttiva “*#pragma omp parallel for*” è stata inserita all’interno del metodo “*convert\_data*”, prima del for che va a richiamare al suo interno il metodo di conversione del data set, il quale lavora su righe indipendenti della matrice x di input. Questo garantisce che non ci siano fenomeni di Race Conditions e di conseguenza si avrà come risultato dei valori corretti.



Il secondo ciclo ottimizzato, si trova all’interno della funzione “*estendi*”, dove si effettua una semplice copia del valore da input->x in input->xast.