Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE

Corso di Laurea in Ingegneria e Scienze informatiche

Algoritmi paralleli per l'allineamento di immagini

Relatore:
Chiar.mo Prof.
Moreno Marzolla

Presentata da:
Alessandro Retico

Sessione II
Anno Accademico 2018/2019

Indice

Elenco delle figure

Sommario

In questa tesi si propone un algoritmo che, date una serie di immagini astronomiche raffiguranti uno stesso soggetto, cerca di ricavarne una rappresentazione migliore.
Scattare foto astronomiche non è affatto facile: bisogna seguire vari accorgimenti e regolare opportunamente le impostazioni della propria fotocamera; a questo scopo esistono vari siti web e libri[1] che cercano di spiegare queste tecniche agli amatori appassionati dell'astrofotografia. In genere gli astrofili, che non sempre dispongono di sofisticatissime attrezzature per fotografare lo spazio, scattano una serie di foto o registrano un video (da cui estrarranno i fotogrammi) del soggetto che vogliono immortalare, attraverso la lente di un telescopio: in questa maniera riescono a catturare dettagli invisibili all'occhio umano, ma le immagini ottenute presentano parecchio rumore; per questo motivo le astrofotografie sono quasi sempre sottoposte ad una fase di elaborazione. Una semplice procedura per ridurre il rumore consiste nel produrre una nuova immagine i cui pixel abbiano valore pari alla media dei corrispettivi pixel di tutte le immagini scattate. Dato che la tecnica di cattura precedentemente descritta è enormemente sensibile allo spostamento, prima di calcolare la media dei pixel, è necessario allineare le immagini (ovvero farle combaciare il più possibile), rimuovendo lo scostamento presente tra esse. Per effettuare questo procedimento (noto anche come *image registration*) esistono varie tecniche più o meno complicate: in questa tesi ne verranno presentate brevemente alcune, ma ci si limiterà ad implementarne una versione banale, basata su una tecnica di *brute force*.

La tematica affrontata si presta molto bene alla programmazione parallela, perciò l'implementazione dell'algoritmo che verrà proposto sfrutterà le clausole OpenMP e le istruzioni SIMD di un Raspberry Pi 2 B; è stata scelta tale architettura date le notevoli capacità di parallelizzazione che è in grado di offrire nonostante i ridotti costi, le contenute dimensioni e la minima energia richiesta, caratteristiche che rendono inoltre possibile la realizzazione di un progetto più ampio, in grado ad esempio di automatizzare le fasi di cattura ed elaborazione delle immagini.

I capitoli sono suddivisi come segue:

- Capitolo 1: viene affrontato più nel dettaglio il tema dell'*image registration*, elencando le varie tipologie, gli ambiti applicativi e alcune tecniche risolutive.
- Capitolo 2: si presentano le tipologie di architetture parallele esistenti e si analizza la macchina utilizzata in questa tesi (Raspberry Pi 2 B); si spiega inoltre il funzionamento alla base di OpenMP e SIMD.
- Capitolo 3: viene proposto un algoritmo risolutivo del problema, specificando le tecniche di parallelizzazione adottate.
- Capitolo 4: si introducono i concetti necessari ad effettuare un'analisi delle prestazioni; si valutano dunque i risultati ottenuti, comparando varie versioni del medesimo algoritmo.

Capitolo 1

Image registration

In questo capitolo si analizza la fase più complessa del problema da risolvere, chiarendo il risultato che si vuole conseguire, in che ambiti questa elaborazione risulta utile e menzionando diverse tecniche risolutive già note nello stato dell'arte. L'implementazione proposta da questa tesi verrà invece trattata nel capitolo 3.

1.1 Specifica del problema e ambiti applicativi

Spesso è necessario confrontare immagini di uno stesso soggetto catturate in situazioni diverse: a questo scopo è utile precedentemente allinearle, dato che potrebbero differire per scostamento, rotazione e/o scalatura; le cause di queste differenze possono essere attribuite ai diversi strumenti di cattura impiegati, alle condizioni in cui si trova il soggetto al momento dello scatto, al movimento, al punto di vista, ecc. Delle immagini disponibili ne viene selezionata una come immagine riferimento (*reference* o *source*), mentre le altre vengono catalogate come bersaglio (*target*).
L'allineamento delle immagini risulta fondamentale in vari ambiti, tra cui le immagini mediche, gli aggiornamenti della cartografia, l'astrofotografia e la realizzazione di panoramiche: in quest'ultimo caso l'elaborazione non serve

per effettuare un confronto, ma per aggiungere informazioni ad un'immagine (creandone una più ampia).

Nella figura 1.1 viene mostrato l'esito della procedura di allineamento implementata in questa tesi, su due immagini rappresentati Giove: l'immagine bersaglio [b] risulta essere spostata (più in alto e a destra) rispetto quella di riferimento [a]; come si può ben notare, in seguito all'allineamento (spostando [b] di 5 pixel a sinistra e di 96 verso il basso), [c] (a differenza di [b]) è confrontabile con [a].

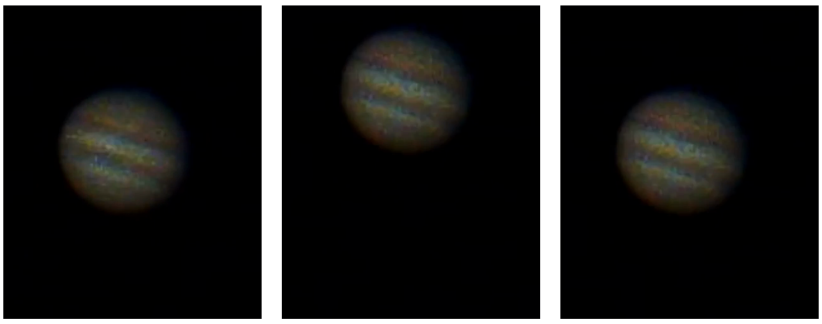

Figura 1.1: Esempio dell'esito della procedura di allineamento implementata in questa tesi.

1.2 Tipologie e tecniche risolutive

Gli algoritmi di allineamento delle immagini possono essere catalogati a seconda di varie caratteristiche[2, 3], tra cui:

- il tipo di corrispondenza: è possibile confrontare il valore dei pixel (*intensità*) oppure cercare delle somiglianze tra le *caratteristiche geometriche*.
- il modello di trasformazione: si può scegliere di modificare l'intera immagine target con *trasformazioni lineari* (tra cui rotazione, trasla-

zione e scalatura) oppure, alterarne solo alcune parti (*trasformazioni "elastiche"* o *"non rigide"*).

- il dominio: le immagini possono essere analizzate sia nel *dominio dello spazio* sia nel *dominio delle frequenze*.

- la modalità di acquisizione: si distinguono in a *singola modalità* (immagini ottenute dallo stesso mezzo) o *multi modali* (immagini ottenute da diversi strumenti di cattura).

- l'interattività: l'utente può essere chiamato ad intervenire o meno per portare a termine la procedura.

Esistono vari algoritmi per l'allineamento di immagini[4], tra i quali:

- quelli basati sulla trasformata di Fourier: sfruttando il *teorema dello spostamento di Fourier* (lavorando dunque nel dominio delle frequenze), è possibile trovare una funzione che assuma valore pari a 0 praticamente ovunque, ad eccezione del punto in cui si verifica l'allineamento migliore[5].

- quelli basati sulla corrispondenza dei bordi: questa tecnica funziona solo su immagini in cui i bordi sono ben definibili; un algoritmo in grado di estrarre tale informazione è il *Canny edge detector*[6].

Oltre a questi appena citati, esistono algoritmi specifici per determinati ambiti applicativi: ad esempio, nell'ambito astronomico è possibile allineare un cielo stellato, sfruttando i triangoli formati dalle stelle, come viene fatto da alcuni software sul mercato[7]. Per quanto riguarda l'algoritmo proposto in questa tesi invece, si tenta di allineare le immagini raccolte da una *stessa fonte*, sfruttando solamente il valore dell'*intensità* dei pixel; le foto vengono esclusivamente *traslate* (non ruotate o scalate) nel *dominio dello spazio*, in maniera praticamente *automatica*: l'unico parametro su cui può agire l'utente riguarda la traslazione massima consentita.

Capitolo 2

Programmazione parallela

In questo capitolo vengono presentate le differenti tipologie di architetture parallele esistenti, soffermandosi in particolare su quella impiegata in questa tesi (Raspberry Pi 2 Model B); infine si descrive il funzionamento delle due tecniche di parallelizzazione adoperate nel progetto (OpenMP e SIMD).

2.1 Architetture

In quasi tutti i calcolatori che utilizziamo quotidianamente, sono presenti più unità di esecuzione (*core*). Lo sviluppo di questo tipo di architetture è dovuto ai limiti fisici raggiunti: per aumentare la potenza di calcolo, nel corso degli anni, sono stati creati processori composti da *transistor* sempre più piccoli e veloci, con frequenze di lavoro sempre più alte; l'incremento eccessivo di queste caratteristiche porta a generare talmente tanto calore da raggiungere temperature troppo elevate, che rischiano di compromettere l'utilizzo dell'unità di esecuzione. Per questo motivo, l'aumento della potenza di calcolo, non si ottiene più dal miglioramento delle prestazioni della singola unità, ma dalla creazione di processori con più *core*: programmando opportunamente il calcolatore, è possibile distribuire il carico di lavoro tra di essi, eseguendo dunque più istruzioni in parallelo.

2.1.1 Tipologie

La figura 2.1, mostra la *tassonomia di Flynn*, un sistema di classificazione delle architetture dei calcolatori: in base alla possibilità di gestire uno o più flussi di dati e di istruzioni (posti rispettivamente sull'asse delle x e delle y), i calcolatori vengono catalogati in uno dei quattro gruppi. La parallelizzazione è ottenibile eseguendo una singola istruzione su più dati contemporaneamente (SIMD - *Single Instruction Multiple Data*) oppure computando più istruzioni nello stesso istante su dati diversi (MIMD - *Multiple Instruction Multiple Data*). Le architetture SISD (*Single Instruction Single Data*), come suggerisce il nome, eseguono un'istruzione alla volta su un singolo dato (la classica macchina di Von Neumann), perciò non effettuano alcun parallelismo, mentre le MISD (*Multiple Instruction Single Data*) non sono state ancora mai utilizzate in pratica.

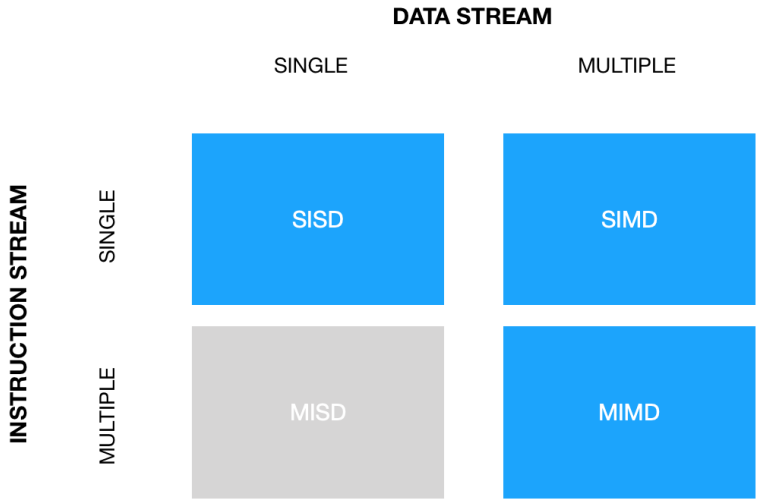

Figura 2.1: Tassonomia di Flynn.

Le architetture MIMD sono a loro volta suddivisibili in:

- A Memoria Condivisa: più unità di esecuzione condividono lo stesso spazio di memoria.

- A Memoria Distribuita: più macchine collaborano, comunicando attraverso una rete, per eseguire la computazione.
- Ibride: più macchine a memoria condivisa sono collegate tra di loro in una rete.

Le macchine a memoria condivisa risultano più semplici da programmare rispetto quelle a memoria distribuita, ma le loro potenzialità sono condizionate dalla limitata velocità di accesso alla memoria. Le architetture a memoria distribuita sono altamente scalabili (basta aggiungere un nodo alla rete per aumentare la potenza di calcolo), ma la latenza di comunicazione può incidere sulle prestazioni: si prestano bene a questo tipo di architettura i programmi che richiedono molta computazione e poca comunicazione.

2.1.2 Raspberry Pi

Il Raspberry Pi è un single-board computer dalle dimensioni ridotte (simili a quelle di una tessera), di costo contenuto (dai 20 ai 60 euro, a seconda della versione). È dotato di porte USB e di un uscita HDMI, perciò è possibile utilizzarlo collegando un mouse e uno schermo; è possibile accedervi anche da remoto, collegandolo alla rete tramite la porta Ethernet oppure via Wireless LAN (per le versioni precedenti alla 3 è necessario dotarsi di un apposito dispositivo USB, dato che il modulo wireless non è integrato nella scheda). L'accesso da remoto può avvenire sia in locale (tramite ssh all'indirizzo del computer) sia al di fuori della rete domestica, installando opportuni programmi come RealVNC[8]. Il sistema operativo è montato su una scheda Micro SD: è possibile installarne di vari tipi, ma quello raccomandato è Raspbian, un sistema ottimizzato per il Raspberry Pi, basato su Debian per l'architettura ARM. I consumi energetici sono ridotti e basta un semplice caricatore USB di uno smartphone per alimentare il dispositivo (per la nuova versione è necessario un alimentatore USB-C).
Questa architettura viene utilizzata in vari contesti, tra cui l'insegnamento scolastico, dato il costo limitato e la possibilità di lavorare "a basso livel-

lo", e il mondo dell'IoT (*Internet of Things*): lo sviluppo in quest'ultimo campo è dovuto alle dimensione ridotte, ai consumi energetici contenuti, al prezzo economico, alla possibilità di pilotare altre componenti (sensori e attuatori) e alla capacità di comunicare con altri dispositivi tramite Internet e Bluetooth (per le versioni meno recenti, dotandole eventualmente dei componenti mancanti). Per quanto riguarda l'applicazione nel mondo dell'HPC (*High Performance Computing*), è possibile realizzare programmi paralleli sfruttando:

- le istruzioni SIMD (come vedremo in 2.2.2).
- le direttive OpenMP: solo dalla versione 2 in poi, in seguito all'introduzione di un processore quad-core.
- MPI: è necessario costruire un cluster di Raspberry Pi[9].
- la GPU: Broadcom ha rilasciato la documentazione[10] della GPU del Raspberry Pi nel 2014; tuttavia programmarla risulta essere molto complicato.

La versione utilizzata in questa tesi è la 2 Model B (figura 2.2); è uscita nel 2015 e dispone di[11]:

- un processore quad-core ARM Cortex-A7 a 900MHz.
- 1GB di RAM.
- una porta Ethernet a 100 Mbit/s.
- 4 porte USB 2.0.
- una porta HDMI.
- jack di uscita audio/video composito da 3,5 mm.
- interfaccia camera (CSI).
- interfaccia display (DSI).

- uno slot per una scheda Micro SD.
- una GPU VideoCore IV 3D.

Figura 2.2: Raspberry Pi 2 Model B (immagine tratta dal sito ufficiale).

2.2 Tecniche di parallelizzazione

Una volta decisa l'architettura su cui lavorare, è necessario scegliere la tecnica di parallelizzazione da adottare. Come già anticipato, in questa tesi si utilizzeranno OpenMP e SIMD su un Raspberry Pi 2 Model B; analizziamone ora il funzionamento.

2.2.1 OpenMP

OpenMP[12, 13] è un modello portabile per la parallelizzazione su architetture a memoria condivisa che richiede il supporto del compilatore (C, C++ o Fortran).

Tramite l'applicazione di specifiche istruzioni al pre-processore in determinati punti del codice, è possibile parallelizzare alcune aree. La clausola principale è *#pragma omp parallel*: il thread master crea un gruppo di threads, i quali eseguiranno tutti le stesse istruzioni, ma su dati differenti; i processi creati verranno distrutti non appena tutti avranno completato il calcolo (viene dunque eseguita un'operazione di sincronizzazione al termine del blocco parallelo), e solo il thread master continuerà l'esecuzione (modello *fork-join*). Ogni thread dispone di un id (il master ha lo 0) e può accedere a variabili private (locali) e condivise: è importante fare attenzione alla visibilità delle variabili per non incorrere in errori dovuti alle *race condition*; se ad esempio due thread modificano la stessa variabile condivisa con un valore diverso, il risultato della computazione non sarà prevedibile, ma dipenderà dall'ordine di esecuzione dei thread (certamente non è il comportamento che si vuole ottenere).

Vediamo come è possibile parallelizzare una semplice somma tra vettori.

```
1    int a[10] = {0,3,−2,5,6,9,10,2,7,5};
2    int b[10] = {15,−20,3,7,6,9,10,15,3,10};
3    int c[10]; // Array con i risultati
4    int N = 10; // Lunghezza array
5
6   for(int i = 0; i < N; i++) c[i] = a[i] + b[i]; // Esegue la somma
```

Nell'esempio sottostante, viene utilizzata la clausola *#pragma omp parallel* per creare un gruppo di threads, i quali calcoleranno i propri indici di inizio e di fine, ed eseguiranno l'operazione di somma solo su una parte dei dati.

```
1    int a[10] = {0,3,−2,5,6,9,10,2,7,5};
2    int b[10] = {15,−20,3,7,6,9,10,15,3,10};
3    int c[10]; // Array con i risultati
4    int N = 10; // Lunghezza array
5    int num_threads = omp_get_max_threads(); // Numero di thread
```

```
6
7  #pragma omp parallel
8  {
9      int my_id = omp_get_thread_num(); // Id del thread
10     int my_start = my_id * N / num_threads; // Indice di inizio
11     int my_stop = (my_id + 1) * N / num_threads; // Indice di fine
12     // Esegue la somma
13     for(int i = my_start; i < my_stop; i++) c[i] = a[i] + b[i];
14 }
```

Il calcolo degli indici viene effettuato in modo tale da distribuire i dati da computare tra i threads: ogni processo calcola l'indice di inizio moltiplicando il proprio id per il numero di valori da sommare diviso il numero di thread disponibili, mentre quello di fine, corrisponde a quello di inizio del thread successivo (*my_id + 1*), garantendo così che tutti i valori vengano effettivamente sommati.

Un altro modo, più semplice, di parallelizzare lo stesso programma è mostrato nelle linee di codice che seguono: con la direttiva *#pragma omp parallel for* non sarà necessario preoccuparsi di calcolare gli indici.

```
1      int a[10] = {0,3,−2,5,6,9,10,2,7,5};
2      int b[10] = {15,−20,3,7,6,9,10,15,3,10};
3      int c[10]; // Array con i risultati
4      int N = 10; // Lunghezza array
5
6  #pragma omp parallel for
7      for(int i = 0; i < N; i++) c[i] = a[i] + b[i]; // Esegue la somma
```

Nonostante questa direttiva risulti essere più semplice, è importante saper calcolare anche manualmente gli indici, poiché non sempre è possibile o conviene usare questa clausola (come vedremo in 3.4).

Chiaramente questo era solo un facile esempio per comprendere il funzionamento di base, ma per parallelizzare programmi più complessi bisogna ricorrere ad altre tecniche. Esistono varie direttive OpenMP che permettono di gestire in modo differente i thread, come ad esempio eseguire un'azione atomicamente (un solo thread alla volta esegue una determinata operazione, in modo da evitare *race condition*), creare task indipendenti da assegnare (in maniera casuale) ai threads disponibili, sincronizzare i processi, ecc.

2.2.2 SIMD

Questo tipo di architettura[14], come accennato in 2.1.1, permette di eseguire un'operazione su più dati differenti contemporaneamente, impiegando circa lo stesso tempo che richiederebbe l'esecuzione della stessa operazione su un solo dato. In alcuni casi, questo genere di parallelizzazione, può incrementare notevolmente le prestazioni. Il punto debole di questo tipo di architettura è la portabilità, dato che ogni processore ha il proprio insieme di istruzioni SIMD.
Esistono vari modi di sfruttare queste estensioni, alcune delle quali sono:

- Compiler auto-vectorization: vengono aggiunti dei flag in fase di compilazione per dire al compilatore di inserire autonomamente istruzioni SIMD dove possibile; questa operazione è completamente trasparente al programmatore, ma non sempre comporta dei miglioramenti: in alcuni casi può capitare che la parallelizzazione prodotta non sia ottima, fallisca oppure non sia effettuata per non violare la sicurezza (talvolta il programmatore può rimediare a questi problemi, fornendo suggerimenti al compilatore).

- Vector data types: sono dei piccoli vettori di un certo tipo di dato, sui quali è possibile eseguire alcune operazioni aritmetiche. Questi vettori non sono portabili, in quanto sono estensioni specifiche del compilatore: sarà quest'ultimo a inserire le giuste istruzioni SIMD per l'architettura

utilizzata; nel caso non sia possibile inserirle, si occuperà di aggiungere l'equivalente codice scalare.

- SIMD intrinsics: sono delle funzioni in C che permettono di eseguire istruzioni a basso livello (assembly). A differenza dei *vector data types*, sono indipendenti dal compilatore, ma dipendenti dall'architettura. Nel progetto sviluppato in questa tesi, vengono sfruttate queste funzioni per parallelizzare parte del codice.

Il seguente esempio parallelizza, utilizzando le *SIMD intrinsics*, la somma tra due vettori; il primo cambiamento, da apportare al codice seriale, riguarda il costrutto *for*: dato che ad ogni iterazione vengono computati quattro valori alla volta, l'incremento dovrà diventare di 4, e la condizione di terminazione non sarà più $i < N$, ma $i < N$ - *3*, in modo tale da arrestare il ciclo non appena il numero di valori da computare rimasti sia minore di 4. Ad ogni iterazione vengono caricati nei registri i quattro valori dei due array da computare, dopodiché vengono sommati e salvati nell'array risultato. Una volta concluso questo primo ciclo *for*, è necessario gestire serialmente i restanti valori (che al massimo saranno 3).

```
1   int a[10] = {0,3,−2,5,6,9,10,2,7,5};
2   int b[10] = {15,−20,3,7,6,9,10,15,3,10};
3   int c[10]; // Array con i risultati
4   int N = 10; // Lunghezza array
5   int i = 0;
6
7   for(; i < N − 3; i+=4) {
8       int32x4_t a_4 = vld1q_s32(a + i); // Carica 4 interi dall'array a
9       int32x4_t b_4 = vld1q_s32(b + i); // Carica 4 interi dall'array b
10      // Somma i 4 interi dei due array
11      int32x4_t c_4 = vaddq_s32(a_4, b_4);
12      vst1q_s32(c + i, c_4); // Salva il risultato in c
13  }
```

```
14      // Gestisce scalarmente i valori rimanenti
15      for(; i < N; i++) c[i] = a[i] + b[i];
```

Come precedentemente detto, le *SIMD intrinsics* sono istruzioni specifiche del processore: è perciò necessario consultare la documentazione del produttore, per poter inserire le corrette chiamate alle funzioni. Il codice appena commentato è stato scritto per un Raspberry Pi 2 Model B (l'architettura descritta in 2.1.2), il quale dispone di 4 microprocessori ARM Cortex-A7: la documentazione[15] necessaria è stata trovata sul sito ufficiale di Arm.

Capitolo 3

Sviluppo dell'algoritmo

In questo capitolo si analizza l'algoritmo sviluppato in questo progetto: si inizia elencando i requisiti e gli input necessari all'esecuzione (chiarendone il significato), e si prosegue mostrando un'immagine prodotta dal programma; a seguire si descrive il processo di elaborazione, spiegando tutti i concetti utili a comprenderne il funzionamento, e sottolineando i miglioramenti apportati alla prima versione pensata. Infine si discutono l'implementazione delle parti principali e le parallelizzazioni applicate.

3.1 Un primo sguardo

Lo scopo dell'algoritmo sviluppato in questo lavoro di tesi (i cui dettagli implementativi verranno forniti più avanti) è quello di ottenere, da una serie di foto, una rappresentazione migliore del soggetto desiderato: per conseguire questo risultato si produce un'immagine in cui, ad ogni pixel, viene assegnata la media dei valori dei corrispondenti pixel delle foto scattate. Prima di poter eseguire questa semplice tecnica di miglioramento, è necessario allineare le immagini: una viene etichettata come riferimento (*reference* - quella su cui eseguire l'allineamento) e le rimanenti vengono classificate come bersaglio (*target* - quelle da allineare). Come discusso in 1.2, esistono varie tecniche di allineamento: in questo progetto ci limiteremo a confrontare il valore

dei pixel, in seguito a delle semplici traslazioni (non considerando quindi rotazioni e/o scalature).

Requisiti

Per il corretto funzionamento del programma, è indispensabile che le immagini:

- siano salvate nella stessa cartella;
- abbiano tutte la stessa risoluzione;
- siano in formato BMP, codificate a 24 bit, con lettura dal basso verso l'alto e da sinistra verso destra.

È necessario inoltre preparare un'apposita cartella, per salvare la versione allineata delle immagini bersaglio: il nome deve essere uguale a quello della cartella contenente le immagini, preceduto da "shifted_". Infine, entrambe le cartelle, devono essere contenute in quella da cui si lancia il programma.

Parametri richiesti

Per eseguire il programma bisogna fornire il nome della cartella contenente le immagini e il numero di file ".bmp" presenti al suo interno; opzionalmente è possibile specificare anche lo scostamento consentito (preceduto da "-o") e il nome dell'immagine riferimento (preceduto da "-i"): in caso questi dati non vengano impostati, il primo parametro sarà pari al 5% della dimensione maggiore delle foto, mentre per quanto riguarda l'immagine riferimento, verrà selezionata la prima trovata all'interno della cartella. Per scostamento si intende: il numero massimo di pixel per il quale, l'immagine bersaglio, può essere spostata in ogni direzione. Lo spostamento viene rappresentato con le coordinate x e y, dove x rappresenta la larghezza e y l'altezza. Un valore positivo per la x significa uno spostamento verso destra, mentre per quanto riguarda le y, un valore positivo equivale ad una traslazione verso il basso.
Nella figura 3.1 si mostrano gli spostamenti, con le relative coordinate, che

un'immagine subirebbe passando come argomento uno scostamento pari ad uno.

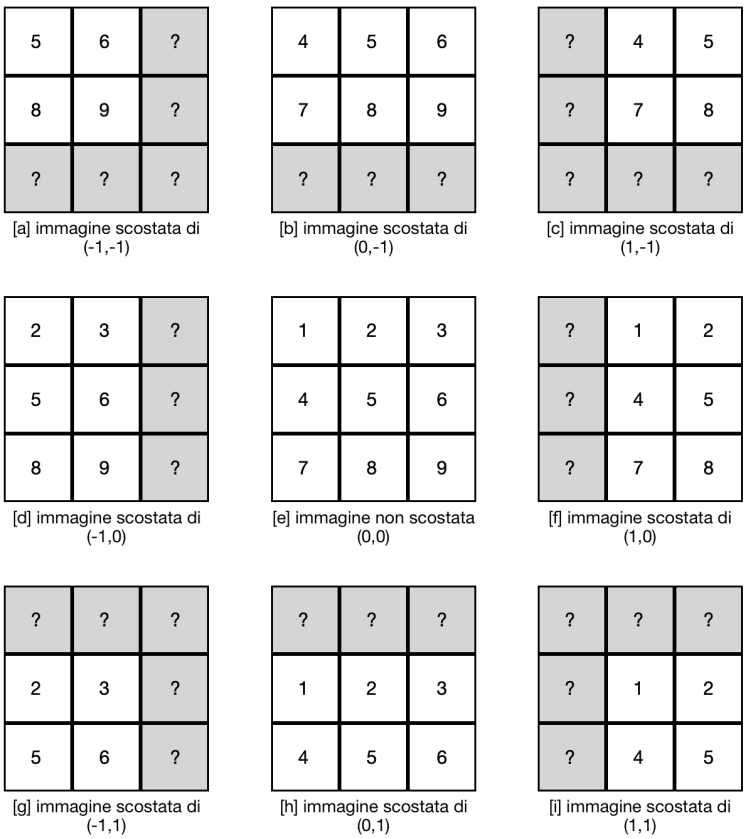

Figura 3.1: Spostamenti subiti da un'immagine con uno scostamento pari ad uno.

Esempio di risultato

Mostriamo subito un esempio di risultato ottenuto dall'applicazione di questo algoritmo: dopo aver estratto 56 fotogrammi da un video raffigurante Giove, è stato eseguito il programma, selezionando come riferimento l'immagine mostrata in figura 3.2 e specificando 100 come scostamento. Come si può notare in figura 3.3, l'immagine prodotta presenta meno rumore.

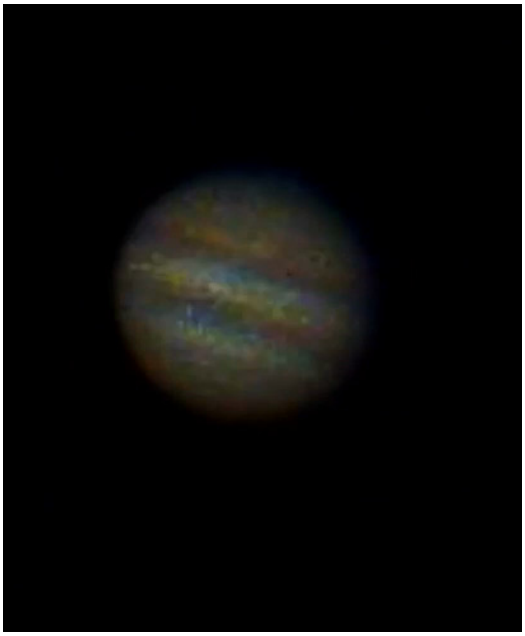

Figura 3.2: L'immagine selezionata come riferimento.

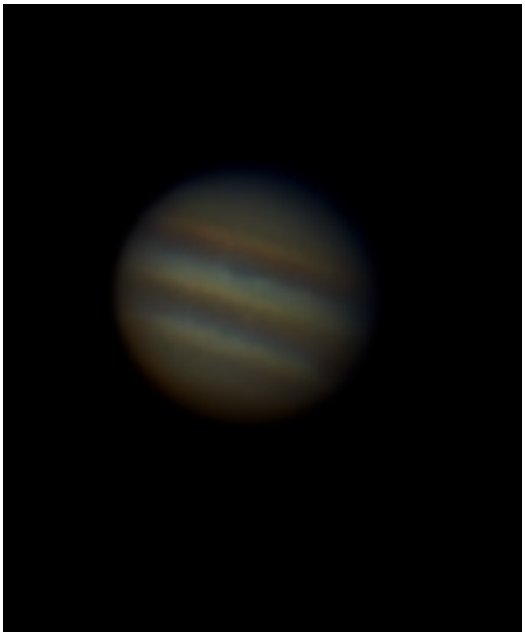

Figura 3.3: L'immagine risultato prodotta.

3.2 L'algoritmo

Una volta determinata l'immagine da utilizzare come riferimento (e di conseguenza quali come bersaglio), si inizia a cercare l'allineamento migliore di ogni foto *target* rispetto quella *reference*. La parte più complessa, sia da un punto di vista computazionale che di logica, sta proprio in questa elaborazione: per capire quale sia il miglior allineamento, si confrontano le due immagini pixel per pixel, traslando quella obiettivo in tutte le possibili direzioni (secondo lo scostamento massimo, come spiegato in 3.1); per ogni traslazione del *target*, viene calcolata la differenza con l'immagine *reference*: una volta eseguita questa operazione per tutti gli spostamenti possibili, vengono determinate le coordinate (x,y) della traslazione che ha reso minima questa differenza. Sorge però un problema: quale valore inserire nei pixel "nuovi" che entrano nell'immagine? Se ad esempio un'immagine viene traslata di uno verso sinistra, nei pixel che entrano a destra, che valore si inserisce? Dato che verrà fatta una media, per non alterare il valore finale di questi pixel per i quali non si ha un valore, si inserisce il valore corrispondente dell'immagine riferimento. Per comprendere meglio il concetto, nella figura 3.4, viene fornito un esempio di una parte del calcolo dell'allineamento migliore; le celle colorate in grigio rappresentano i "nuovi" pixel entrati: in linea con quanto spiegato prima, è stato inserito il valore di quelli situati nella stessa posizione nell'immagine riferimento; le celle gialle invece rappresentano, in fase di confronto, i pixel tra cui c'è una perfetta corrispondenza: chiaramente le celle grigie dell'immagine traslata, ovvero i "nuovi" pixel inseriti, saranno gialle al momento della sovrapposizione. Osservando le immagini sovrapposte, si deduce che l'allineamento migliore si ottiene con una traslazione di (1,0): così facendo infatti, le immagini combaciano perfettamente, senza alcuna differenza; se non si applica nessuna traslazione la differenza risulta essere di 40, mentre spostando l'immagine di (1,1) la differenza si riduce a 10. Chiaramente non bisogna eccedere con il valore massimo dello scostamento, altrimenti si rischia di ottenere una copia dell'immagine riferimento, e non un allineamento.

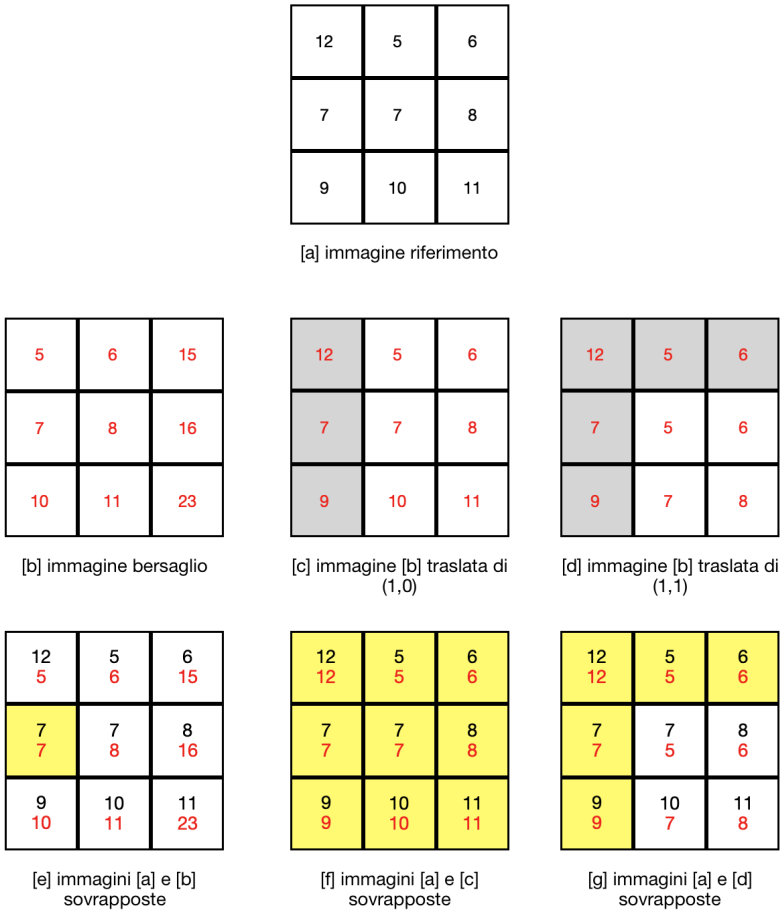

Figura 3.4: Esempio di una parte del calcolo dell'allineamento migliore.

Ricavate le coordinate, l'immagine viene traslata e salvata; dopo aver completato l'allineamento di tutte le foto, sarà finalmente possibile produrre la nuova immagine, calcolando la media di quelle traslate e di quella di riferimento.

Miglioramenti

In seguito ad un'analisi più approfondita di questa prima versione dell'algoritmo, ne è stata implementata un'altra, apportando i seguenti miglioramenti:

- Precalcolo degli offsets: come vedremo in 3.3, per effettuare il controllo, si agisce direttamente sull'immagine obiettivo, senza effettuare prima

la traslazione. Questa tecnica prevede vari calcoli in base allo scostamento dell'immagine, in quanto il pixel i dell'immagine riferimento, non dovrà essere confrontato con il pixel i di quella obiettivo, ma con quello $i + z$, dove z è un valore che dipende dalla traslazione; oltre a questo dato, è necessario calcolare quali pixel vanno confrontati e quali no: ad esempio i pixel "nuovi" dell'immagine obiettivo non devono essere confrontati con quelli dell'immagine riferimento, in quanto, per il metodo di traslazione scelto, hanno lo stesso valore (perciò, siccome la loro differenza è 0, possono essere ignorati). Dato che gli scostamenti e le dimensioni delle immagini sono uguali, questi dati possono essere calcolati una sola volta all'inizio, ed essere poi utilizzati direttamente durante la fase di allineamento, riducendo il costo computazionale.

- Interruzione preventiva del calcolo dell'allineamento: mentre si cerca l'allineamento migliore per un'immagine, è necessario salvarsi la differenza minima calcolata fino a quel momento (e chiaramente ricordarsi anche a quali coordinate corrisponde): man mano che si va avanti con la ricerca, questo valore tende a scendere sempre più; per ottimizzare i tempi, al termine del confronto di ogni riga dell'immagine, si verifica se la differenza minima trovata fino ad ora è già minore di quella che si sta calcolando attualmente: in tal caso sarebbe inutile continuare la computazione, perciò si interrompe il confronto e si passa all'offset successivo.

- Allineamento sulla versione in scala di grigi: ogni pixel dell'immagine è rappresentato da una terzina di valori: per ridurre il numero di confronti da effettuare, è possibile mappare questi tre valori in uno soltanto, rappresentando l'immagine in scala di grigi invece che a colori. Il risultato ottenuto dall'allineamento potrebbe risultare diverso da quello calcolato con le immagini a colori (probabilmente sarà un pò meno accurato), ma il guadagno in termini computazionali è notevole.

3.3 Implementazione

Viene mostrata l'implementazione delle parti principali dell'algoritmo.

Caricamento di un'immagine in memoria

La funzione *BMPimage readBMP(char* filename)* si occupa di caricare in memoria l'immagine contenuta nel file *filename*: i dati vengono restituiti nella seguente struttura.

```
1  struct BMPimage {
2      int width; // Larghezza (in pixel)
3      int height; // Altezza (in pixel)
4      int size; // Numero di dati (in byte)
5      int width_byte; // Larghezza (in byte)
6      int width_padding; // Byte per il padding
7      int y_inc; // Byte totali in ogni riga
8      short n_bit; // Numero di bit per pixel
9      int n_byte; // Numero di byte per pixel
10     int offset; // Offset dall'header all'inizio dei dati
11     int other_len; // Dimensione delle altre informazioni
12     unsigned char *header; // Header
13     unsigned char *other_info; // Altre informazioni
14     unsigned char *data; // Dati
15 };
```

Alcuni dati sono presi direttamente dal file dell'immagine (leggendo determinati byte dell'header[16]), mentre altri, utili per vari scopi all'interno del programma, sono derivati da essi.

```
1      image.other_len = image.offset - HEADER_LEN; // HEADER_LEN = 54
2      image.n_byte = image.n_bit / 8;
3      image.width_byte = image.width * image.n_byte;
4      image.width_padding = image.width_byte % 4;
```

```
5      image.y_inc = image.width_byte + image.width_padding;
6      image.size = image.y_inc * image.height;
```

Ogni riga di un'immagine BMP, deve essere multipla di 4 byte: in caso non lo fosse, vengono inseriti dei byte di *padding*.
L'immagine può essere riscritta su file, stampando in ordine *header*, *other_info* e *data* (*writeBMP(BMPimage image, char* filename)*): lo spazio di memoria per questi tre campi è allocato dinamicamente, perciò è consigliabile chiamare l'opportuna funzione *freeBMP(BMPimage image)* per deallocarla, non appena l'immagine non verrà più utilizzata.

Precalcolo degli offsets

La preparazione dei dati necessari alla fase di allineamento (come spiegato in 3.2) viene eseguita dalla funzione *offsetInfo* prepareOffsets(int width, int height, int off)*, dove i primi due argomenti sono rispettivamente, la larghezza e l'altezza delle immagini da elaborare, mentre *off*, indica lo scostamento massimo: i dati vengono restituiti in un'apposita struttura.

```
1  struct offsetInfo {
2      int y_off; // Scostamento y
3      int y_init; // Prima riga da computare
4      int y_limit; // Ultima riga da computare
5      int x_off; // Scostamento x
6      int x_init; // Prima colonna da computare
7      int x_limit; // Ultima colonna da computare
8      int reference_off; // Offset per reference
9      int target_off; // Offset per target
10 };
```

Il codice che segue, mostra come avviene il riempimento dei campi.

```
1   const int y_num_off = off * 2 + 1; // Numero di offsets per riga
2   // Numero di offsets (da [−off, −off] a [off, off])
3   const int num_off = y_num_off * y_num_off;
4   // Alloca spazio per salvare i dati di tutti gli offset
5   offsetInfo* offsets = (offsetInfo*) malloc (sizeof(offsetInfo) * (num_off));
6
7   for(int y_off = −off; y_off <= off; y_off++){
8       // Parte dell'indice a cui posizionare l'offset
9       const int curr_y_off = (y_off + off) * y_num_off;
10      const int y_init = y_off < 0 ? 0 : y_off;
11      const int y_limit = y_off < 0 ? height + y_off : height;
12      // Precalcola gli offset di y
13      const int reference_y_off = y_init * width;
14      const int target_y_off = (y_init − y_off) * width;
15
16      for(int x_off = −off; x_off <= off; x_off++){
17          // Indice a cui posizionare l'offset
18          const int curr_off = curr_y_off + (x_off + off);
19          // −y_off dato che un'immagine BMP si legge dal basso
20          // verso l'alto (bisogna ribaltare l'asse)
21          offsets[curr_off].y_off = −y_off;
22          offsets[curr_off].y_init = y_init;
23          offsets[curr_off].y_limit = y_limit;
24          offsets[curr_off].x_off = x_off;
25          offsets[curr_off].x_init = x_off < 0 ? 0 : x_off;
26          offsets[curr_off].x_limit = x_off < 0 ? width + x_off : width;
27          // Precalcola gli offset
28          offsets[curr_off].reference_off = reference_y_off;
29          offsets[curr_off].target_off = target_y_off − x_off;
30      }
31  }
```

Gli indici sono calcolati per immagini con pixel rappresentati da un unico valore (allineeremo infatti i dati in scala di grigio), e si riferiscono all'immagine riferimento: per calcolare quelli dell'immagine obiettivo è sufficiente sottrarre *x_off* o *y_off*. Verifichiamo ora l'esattezza dei calcoli, facendo degli esempi:

- *y_off = -1*: l'immagine va spostata verso l'alto di uno, perciò la prima riga ([0]) di *reference* deve essere confrontata con la seconda ([0 - (-1)] = [1]) di *target*, e la penultima riga di *reference* ([(*height* - 1) + (-1)] = [*height* - 2]) con l'ultima ([(*height* - 2) - (-1)] = [*height* - 1]) di *target*.

- *y_off = 1*: l'immagine va spostata verso il basso di uno, perciò la seconda riga ([1]) di *reference* deve essere confrontata con la prima ([1 - (1)] = [0]) di *target*, e l'ultima riga di *reference* ([*height* - 1]) con la penultima ([(*height* - 1) - (1)] = [*height* - 2]) di *target*.

- *x_off = -1*: l'immagine va spostata verso sinistra di uno, perciò la prima colonna ([0]) di *reference* deve essere confrontata con la seconda ([0 - (-1)] = [1]) di *target*, e la penultima colonna di *reference* ([(*width* - 1) + (-1)] = [*width* - 2]) con la l'ultima ([(*width* - 2) - (-1)] = [*width* - 1]) di *target*.

- *x_off = 1*: l'immagine va spostata verso destra di uno, perciò la seconda colonna ([1]) di *reference* deve essere confrontata con la prima ([1 - (1)] = [0]) di *target*, e l'ultima colonna di *reference* ([(*width* - 1)]) con la penultima ([(*width* - 1) - (1)] = [*width* - 2]) di *target*.

Per quanto riguarda *reference_off* e *target_off*, sono stati calcolati come la riga del primo pixel da computare (*y _init * width*): per *target* è stato necessario togliere (come precedentemente spiegato) *y_off* da *y_init* e *x_off* da *reference_off* (perciò *reference_off = (y_init - y_off) * width - x_off*). Vedremo in fase di allineamento come questi valori verranno utilizzati.

Conversione in scala di grigi

Per ottenere la conversione in scala di grigi di un'immagine si utilizza la funzione *unsigned char* getGrayscaleData(BMPimage image)*, implementata assegnando un peso ad ognuna delle tre componenti.

```
1   int size = image.width * image.height; // Un solo byte per pixel
2   unsigned char* g_image = (unsigned char *) malloc (sizeof
3                                 (unsigned char) * size);
4   int pos = 0;
5   for(int y = 0; y < image.height; y++){
6       const int y_val = image.y_inc * y;
7       // Mappa il pixel RGB in un singolo pixel in grayscale
8       for(int x = 0; x < image.width_byte; x+=image.n_byte){
9           unsigned char lum = image.data[y_val + x] * 0.11
10                              + image.data[y_val + x + 1] * 0.59
11                              + image.data[y_val + x + 2] * 0.3;
12          g_image[pos++] = lum;
13      }
14  }
```

Allineamento

La fase di allineamento viene eseguita su tutte le immagini obiettivo, rispetto quella riferimento (tutte in scala di grigi): il risultato di questa fase è riportato nella struttura *alignInfo*.

```
1 struct alignInfo {
2     int x_off; // Offset x
3     int y_off; // Offset y
4     long diff; // Differenze calcolata
5 };
```

L'elaborazione viene implementata da *alignInfo alignBMP(unsigned char* target, unsigned char* reference, int y_inc, offsetInfo* offsets, int num_offset)*, dove i parametri rappresentano rispettivamente i pixel in scala di grigi di *target*, quelli di *reference*, la lunghezza in byte di una riga, gli offsets precalcolati e il numero di offsets totali.

```
1   alignInfo best_align; // Miglior allineamento calcolato
2   best_align.x_off = 0;
3   best_align.y_off = 0;
4   best_align.diff = LONG_MAX;
5
6   for(int i = 0; i < num_offset; i++){
7       long diff = 0; // Somma delle differenze
8       // Salva le informazioni (precalcolate) dell'offset attuale
9       // per poter calcolare l'allineamento
10      const int y_init = offsets[i].y_init;
11      const int y_limit = offsets[i].y_limit;
12      const int x_init = offsets[i].x_init;
13      const int x_limit = offsets[i].x_limit;
14      int target_off = offsets[i].target_off;
15      int reference_off = offsets[i].reference_off;
16
17      for(int y = y_init; y < y_limit; y++) {
18          for(int x = x_init; x < x_limit; x++)
19              diff += abs(target[target_off + x] - reference[reference_off + x]);
20          // Aumenta gli offsets di una riga
21          target_off += y_inc;
22          reference_off += y_inc;
23          // Se la somma delle differenze attuale > della migliore
24          // trovata finora, interrompe la computazione per l'offset attuale
25          if(best_align.diff < diff) break;
26      }
27      // Se la somma delle differenze attuale < della migliore trovata
28      // finora, salva le informazioni relative all'offset attuale come il migliore
```

```
29          if(diff < best_align.diff) {
30            best_align.diff = diff;
31            best_align.x_off = offsets[i].x_off;
32            best_align.y_off = offsets[i].y_off;
33          }
34      }
```

Traslazione

Una volta ottenute le coordinate relative all'allineamento migliore, l'immagine obiettivo (quella a colori) viene traslata e salvata su file. La traslazione prevede il calcolo degli indici dello specifico offset (nella stessa maniera usata per il precalcolo), tenendo conto questa volta che l'immagine ha *n_byte* per ogni pixel, ed eventualmente *width_padding* byte di *padding*. La funzione da chiamare è *BMPimage shiftBMP(const BMPimage target_image, const BMPimage reference_image, const int x_off, int y_off)*.

```
1     // I pixel "nuovi" hanno il relativo valore di 'reference_image'
2     const BMPimage shifted_image = copyBMP(reference_image);
3     const int n_byte = shifted_image.n_byte;
4     const int height = shifted_image.height;
5     const int width = shifted_image.width;
6     // L'offset verticale viene invertito dato che l'immagine viene letta
7     // dal basso verso l'alto (bisogna ribaltare l'asse)
8     y_off = -y_off;
9     const int y_init = y_off < 0 ? 0 : y_off;
10    const int y_limit = y_off < 0 ? height + y_off : height;
11    const int y_inc = shifted_image.y_inc; // Lunghezza di una riga
12    const int x_init = x_off < 0 ? 0 : x_off * n_byte;
13    const int x_limit = x_off < 0 ? (width + x_off) * n_byte: width * n_byte;
14    // Precalcola gli offset
15    const int target_off = (-y_off * y_inc) + ((-x_off) * n_byte);
```

```
16
17      for(int y = y_init; y < y_limit; y++){
18          const int y_val = y * y_inc;
19          for(int x = x_init; x < x_limit; x++)
20              shifted_image.data[y_val + x] = target_image.data[target_off
21                                                                + y_val + x];
22      }
```

Media

L'ultima fase consiste nell'eseguire la media per ottenere l'immagine risultato: viene eseguita invocando *BMPimage reductBMP(const BMPimage reference_image, char** files_name, const int num_images)*, dove i parametri sono rispettivamente l'immagine riferimento, i percorsi dei file contenenti le immagini obiettivo traslate e il numero di quest'ultime.

```
1       // Istanzia le informazioni dell'immagini
2       // (il valore dei pixel viene poi modificato)
3       const BMPimage result_image = copyBMP(reference_image);
4       const int image_len = reference_image.size;
5       // Alloca un array di interi utilizzato per calcolare la somma dei valori
6       // dei pixel di tutte le immagini
7       int* image_sum = (int*)malloc(sizeof(int)*image_len);
8       // Inizializza il valore dall'array con il valore dei pixel di 'reference_image'
9       for(int i = 0; i < image_len; i++) image_sum[i] = reference_image.data[i];
10
11      for(int j = 0; j < num_images; j++){
12          const BMPimage target_image = readBMP(files_name[j]);
13          // Somma il valore di ogni pixel, all'array
14          for(int i = 0; i < image_len; i++) image_sum[i] += target_image.data[i];
15          freeBMP(target_image); // Libera la memoria
16      }
17      // Numero totale di immagini utilizzate per calcolare l'immagine risultato
```

```
18      // (+ 1 per includere la 'reference_image')
19      const int tot_n_images = num_images + 1;
20      // Assegna ad ogni pixel dell'immagine risultato, la media della somma
21      // del valore dell'immagine riferimento e di quella delle altre immagini.
22      for(int i = 0; i < image_len; i++) result_image.data[i] = image_sum[i]
23                                                              / tot_n_images;
24      free(image_sum); /* Libera la memoria. */
```

Main

Dopo aver spiegato le parti principali del programma, si mostra ora come utilizzarle per implementare l'algoritmo.

```
1       if (argc < 3) return 1; // Non sono stati passati tutti gli argomenti
2       const char* dir_name = argv[1];
3       int num_images = atoi(argv[2]);
4       // Parametri opzionali
5       int offset = 0;
6       char* reference_filename = NULL;
7       if (argc > 4) { // Sono stati passati parametri opzionali
8           // Offset (CL_OFF = "-o")
9           if (!strncmp(argv[3], CL_OFF, strlen(CL_OFF)))
10              offset = atoi(argv[4]);
11          else if (argc > 6 && !strncmp(argv[5], CL_OFF, strlen(CL_OFF)))
12              offset = atoi(argv[6]);
13          // Reference image name (CL_IMG = "-i")
14          if (!strncmp(argv[3], CL_IMG, strlen(CL_IMG)))
15              reference_filename = argv[4];
16          else if (argc > 6 && !strncmp(argv[5], CL_IMG, strlen(CL_IMG)))
17              reference_filename = argv[6];
18      }
19      // Nomi delle immagini da analizzare
20      char** files_name = (char**)malloc(sizeof(char*) * num_images);
```

```
21      // Salva il numero effettivo di immagini da analizzare
22      num_images = getBMPnames(dir_name, files_name, num_images,
23                                          reference_filename);
24      // Nomi delle immagini allineate (-1 dato che
25      // quella ''reference'' non viene traslata)
26      char** shifted_files_name = (char**)malloc(sizeof(char*)
27                                      * (num_images - 1));
28      // Controlla se ci sono meno di due immagini nella cartella
29      // (non ha senso eseguire la computazione) o se si sono
30      // verificati errori: in tal caso l'esecuzione viene interrotta.
31      if (num_images < 2) return 2;
32      // La prima immagine viene selezionata come immagine riferimento
33      const BMPimage reference_image = readBMP(files_name[0]);
34      // Si ottengono i pixel grayscale dell'immagine riferimento
35      unsigned char* reference_image_g = getGrayscaleData(reference_image);
36      // Parametro relativo all'offset non passato: ne viene calcolato
37      // uno di default in base alle dimensioni dell'immagine
38      if (offset == 0) {
39          const int max_dim = reference_image.width > reference_image.height ?
40                                      reference_image.width : reference_image.height;
41          // Per default si calcola come il 5% dei pixel lungo
42          // la dimensione maggiore dell'immagine
43          offset = max_dim * 5 / 100;
44      }
45      // Numero di confronti da effettuare per ogni immagine
46      const int num_offsets = (offset * 2 + 1) * (offset * 2 + 1);
47      // Precalcola gli offsets da applicare per allineare le immagini
48      offsetInfo* offsets = prepareOffsets(reference_image.width,
49                                              reference_image.height, offset);
50      // Per ogni immagine da allineare
51      for(int file_idx = 1; file_idx < num_images; file_idx++) {
52          char* filename = files_name[file_idx]; // Nome del file da computare
53          // Immagine obiettivo
```

```
54          const BMPimage target_image = readBMP(filename);
55          // Si ottengono i pixel grayscale dell'immagine obiettivo
56          unsigned char* target_image_g = getGrayscaleData(target_image);
57          // Cerca l'offset dell'allineamento migliore
58          // (sui pixel grayscale delle immagini)
59          alignInfo best_align = alignBMP(target_image_g, reference_image_g,
60                                          reference_image.width, offsets, num_offsets);
61          // Esegue la traslazione (sulle immagini in RGB non sui pixel grayscale)
62          const BMPimage curr_image = shiftBMP(target_image, reference_image,
63                                               best_align.x_off, best_align.y_off);
64          // Salva il nome del file (PREFIX_SHIFT_DIR = ``shifted_'')
65          const int len = strlen(PREFIX_SHIFT_DIR) + strlen(filename) + 1;
66          shifted_files_name[file_idx - 1] = (char*)malloc(sizeof(char)*len);
67          sprintf(shifted_files_name[file_idx - 1], "%s%s",
68                  PREFIX_SHIFT_DIR, filename);
69          // Salva l'immagine su file
70          writeBMP(curr_image, shifted_files_name[file_idx - 1]);
71          // Libera la memoria
72          freeBMP(target_image);
73          free(target_image_g);
74          freeBMP(curr_image);
75      }
76      // Libera la memoria. */
77      free(offsets);
78      for(int i = 0; i < num_images; i++) free(files_name[i]);
79      free(files_name);
80      // Esegue la riduzione sull'immagine riferimento, usando le ``num_images''
81      // - 1 ( - 1, dato che 'num_images' comprende anche quella riferimento)
82      // immagini allineate
83      BMPimage result_image = reductBMP(reference_image, shifted_files_name,
84                                        num_images - 1);
85      // Scrive l'immagine risultato su file
86      // (RESULT_IMAGE_NAME = ``result.bmp'')
```

```
87      writeBMP(result_image, RESULT_IMAGE_NAME);
88      // Libera la memoria
89      freeBMP(reference_image);
90      for(int i = 0; i < num_images - 1; i++) free(shifted_files_name[i]);
91      free(shifted_files_name);
92      freeBMP(result_image);
93      free(reference_image_g);
```

3.4 Parallelizzazioni

Vengono ora mostrate le parallelizzazioni applicate al codice descritto in 3.3.

OpenMP

È stata inserita una clausola *#pragma omp parallel for* nella maggior parte dei cicli: condizione necessaria per lo sfruttamento di questa parallelizzazione è l'assenza di *loop-carried dependencies* nei cicli interessati, ovvero la computazione di ogni iterazione non deve dipendere in nessuna maniera da quelle precedenti; nel caso in cui questo requisito non venisse rispettato, il risultato finale della computazione non sarebbe determinabile a priori, ma dipenderebbe dall'ordine di esecuzione delle iterazioni (ed analogamente a quanto visto per le *race condition* in 2.2.1, non è il comportamento che si vuole ottenere). A seguire vengono mostrati i cicli ai quali è stata applicata questa direttiva (chiaramente, per quanto appena spiegato, non presentano alcuna dipendenza).

```
1 prepareOffsets( . . . ) {
2       . . .
3 // I threads si dividono (per righe) il calcolo degli offsets
4 #pragma omp parallel for
5     for(int y_off = -off; y_off <= off; y_off++) {
6         . . .
```

```
7           for(int x_off = -off; x_off <= off; x_off++) {
8               ...
9           }
10      }
11      ...
12  }
13
14  shiftBMP( ... ) {
15      ...
16  // I threads si divisono (per righe) la traslazione
17  #pragma omp parallel for
18      for(int y = y_init; y < y_limit; y++){
19          const int y_val = y * y_inc;
20          for(int x = x_init; x < x_limit; x++)
21              shifted_image.data[y_val + x] = target_image.data[target_off
22                                                                + y_val + x];
23      }
24      ...
25  }
26
27  reductBMP( ... ) {
28      ...
29  // I threads si dividono l'inizializzazione dei valori
30  // dell'array ``somma'' con il valore dei pixel di ``reference_image''
31  #pragma omp parallel for
32      for(int i = 0; i < image_len; i++) image_sum[i] = reference_image.data[i];
33
34      for(int j = 0; j < num_images; j++){
35          ...
36      // I threads si dividono la somma di ogni pixel
37      // dell'immagine obiettivo all'array ``somma''
38      #pragma omp parallel for
39          for(int i = 0; i < image_len; i++) image_sum[i] += target_image.data[i];
```

```
40          . . .
41      }
42      . . .
43  // I threads si dividono il calcolo del valore
44  // dei singoli pixel dell'immagine risultato
45  #pragma omp parallel for
46      for(int i = 0; i < image_len; i++) result_image.data[i] = image_sum[i]
47                                                              / tot_n_images;
48      . . .
49  }
```

La fase di allineamento (che è quella con costo computazionale maggiore) è stata parallelizzata con una direttiva *#pragma omp parallel*.

```
1  alignInfo alignBMP( . . . ) {
2      const int num_threads = omp_get_max_threads(); // Numero di threads
3      // Ogni thread salva il miglior allineamento calcolato in questo array
4      alignInfo* best_align = (alignInfo *)malloc(sizeof(alignInfo) * num_threads);
5      // Ogni thread inizializza le informazioni sull'allineamento
6      for(int i = 0; i < num_threads; i++) {
7          best_align[i].x_off = 0;
8          best_align[i].y_off = 0;
9          best_align[i].diff = LONG_MAX;
10     }
11   // Crea i thrads
12 #pragma omp parallel
13 {
14     const int my_rank = omp_get_thread_num(); // Indice del thread
15     // Indice di inzio del thread
16     const int my_start = my_rank * num_offset / num_threads;
17     // Indice di fine del thread
18     const int my_stop = (my_rank + 1) * num_offset / num_threads;
19     // Ogni thread computa il miglior allineamento su una parte degli offsets
```

```
20      for(int i = my_start; i < my_stop; i++){
21          long diff = 0; // Somma delle differenze
22          // Salva le informazioni (precalcolate) dell'offset attuale
23          // per poter calcolare l'allineamento
24          const int y_init = offsets[i].y_init;
25          const int y_limit = offsets[i].y_limit;
26          const int x_init = offsets[i].x_init;
27          const int x_limit = offsets[i].x_limit;
28          int target_off = offsets[i].target_off;
29          int reference_off = offsets[i].reference_off;
30          for(int y = y_init; y < y_limit; y++) {
31              for(int x = x_init; x < x_limit; x++)
32                  diff += abs(target[target_off + x] - reference[reference_off + x]);
33              // Aumenta gli offsets di una riga
34              target_off += y_inc;
35              reference_off += y_inc;
36              // Se la somma delle differenze attuale > della migliore trovata
37              // finora, interrompe la computazione per l'offset attuale
38              if(best_align[my_rank].diff < diff) break;
39          }
40          // Se la somma delle differenze attuale < della migliore trovata
41          // finora, salva le informazioni relative all'offset attuale come il migliore
42          if(diff < best_align[my_rank].diff) {
43              best_align[my_rank].diff = diff;
44              best_align[my_rank].x_off = offsets[i].x_off;
45              best_align[my_rank].y_off = offsets[i].y_off;
46          }
47      }
48  }
49      alignInfo result = best_align[0];
50      // Determina l'offset migliore tra quelli calcolati da ogni thread
51      for(int i = 1; i < num_threads; i++)
52          if(result.diff > best_align[i].diff) result = best_align[i];
```

```
53      free(best_align); // Libera la memoria
54      return result;
55  }
```

Non è stata utilizzata una semplice clausola *#pragma omp parallel for* in quanto sarebbe stato necessario, all'interno del ciclo, ottenere ogni volta l'indice del thread, per salvare le informazioni relative al miglior allineamento calcolato fino a quel momento. Al posto di dividere tra i threads i vari offsets, si potevano dividere le righe dell'immagine: in tal caso bastava un unico dato relativo al miglior allineamento, ma sarebbe stata necessaria un'operazione di riduzione sulla variabile relativa alla differenza attuale, dato che operazioni di somma da parte di più threads su una variabile condivisa generano *race condition* (come spiegato in 2.2.1); inoltre, con questo approccio, non sarebbe stato possibile applicare il miglioramento relativo all'interruzione preventiva del calcolo dell'allineamento (descritto in 3.2), in quanto il valore della differenza sarebbe stato diviso tra i threads.

SIMD

Per quanto riguarda l'architettura SIMD, sono state sfruttate le *SIMD intrinsics* (in maniera analoga all'esempio mostrato in 2.2.2) solamente nel confronto dei pixel tra immagine riferimento e obiettivo, nella fase di allineamento: in seguito all'inserimento di tali istruzioni, ogni thread calcola 16 pixel alla volta.

```
1  alignBMP( . . . ) {
2      . . .
3  #pragma omp parallel
4  {
5      . . .
6      for(int i = my_start; i < my_stop; i++) {
7          . . .
8          int x = 0;
```

```
9           for(int y = y_init; y < y_limit; y++) {
10              // Computa 16 pixel alla volta
11              for(x = x_init; x < x_limit - 15; x+=16) {
12                  // Carica i dati dell'immagine target
13                  uint8x16_t target_a = vld1q_u8(target + target_off + x);
14                  // Carica i dati dell'immagine reference
15                  uint8x16_t reference_a = vld1q_u8(reference + reference_off + x);
16                   // Valore assoluto della differenza dei pixel caricati
17                  uint8x16_t result = vabdq_u8(target_a, reference_a);
18                  // Somma il valore assoluto delle 16 differenze eseguite
19                  for(int z = 0; z < 16; z++) diff += result[z];
20              }
21              // Esegue l'operazione sui pixel rimanenti
22              for(; x < x_limit; x++) diff += abs(target[target_off + x]
23                                                  - reference[reference_off + x]);
24              ...
25          }
26          ...
27      }
28  }
29      ...
30  }
```

Capitolo 4

Analisi delle prestazioni

Una fase fondamentale al termine della realizzazione di un programma parallelo è l'analisi delle prestazioni: in questo capitolo verrano forniti i concetti necessari a realizzarla. Una volta apprese le nozioni basilari, verrà effettuata tale analisi sul codice prodotto, e si metteranno a confronto varie implementazioni dell'algoritmo proposto.

4.1 Concetti teorici

Il primo fattore da considerare è lo *speedup*, ovvero il rapporto tra il tempo di esecuzione del programma seriale e quello parallelo:

$$S(p) = \frac{T_{seriale}}{T_{parallelo}(p)} \tag{4.1}$$

dove p è il numero di processori impiegati.
Una versione seriale dell'algoritmo è ottenibile eseguendo quello parallelo con un solo thread (sebbene risulti leggermente più lenta per la creazione di quest'ultimo), perciò normalmente lo speedup viene calcolato come $\frac{T_{parallelo}(1)}{T_{parallelo}(p)}$: in questo progetto non sarà possibile applicare quanto appena detto, dato che, come spiegato in 3.4, vengono utilizzate sia direttive OpenMP che istruzioni SIMD.

Normalmente non si verifica che $S(p) = p$ (*speedup lineare*) come ci si potrebbe aspettare, a causa di porzioni di codice che non possono essere parallelizzate (ad esempio per tempi di comunicazione, dipendenze nell'algoritmo, ...), perciò solitamente $S(p) < p$; in alcuni casi può accadere però che $S(p) > p$: tale fenomeno, che prende il nome di *speedup superlineare*, può essere dovuto alla disponibilità di core eterogenei (processori con maggiore potenza rispetto quello su cui si esegue il programma seriale) o ad una migliore località di acceso ai dati (grazie alle memorie *cache*).
Definito α come la porzione di codice non parallelizzabile, e di conseguenza $1 - \alpha$ come la parte che può esserlo, il tempo di esecuzione parallelo può essere definito come:

$$T_{parallelo}(p) = \alpha T_{seriale} + \frac{(1-\alpha)\, T_{seriale}}{p} \tag{4.2}$$

Sostituendo la (1.2) nella (1.1) e moltiplicando per $T_{seriale}$, si ottiene la legge di Amdhal:

$$S(p) = \frac{1}{\alpha + \frac{(1-\alpha)}{p}} \tag{4.3}$$

Da quest'ultima formula è possibile osservare che lo *speedup* massimo ottenibile è pari a $\frac{1}{\alpha}$ (facendo tendere il numero di processori all'infinito).

Il secondo criterio di analisi è l'efficienza. Questo fattore è valutabile sotto due aspetti: la Strong Scaling ($E(p)$) e la Weak Scaling ($W(p)$).
La $E(p)$ consiste nell'aumentare il numero di processori p, lasciando la dimensione totale del problema fissa (riducendo dunque l'ammontare di lavoro di ogni core): l'obiettivo è quello di ridurre il tempo di esecuzione totale, aumentando il numero di processori.

$$E(p) = \frac{S(p)}{p} \tag{4.4}$$

La $W(p)$ invece, aumenta il numero di processori, ma mantiene il carico di lavoro di ognuno di essi fisso (aumentando perciò quello totale): l'obiettivo,

in questo caso, è quello di risolvere problemi di dimensioni maggiori, nella stesa porzione di tempo.

$$W(p) = \frac{T_1}{T_p} \tag{4.5}$$

dove T_1 è il tempo necessario a completare un'unità di lavoro con un processore, e T_p quello per completarne p con p processori.
Chiaramente più $W(p)$ è alto, più il codice è efficiente: se ad esempio fosse uguale ad 1, significherebbe che eseguire 3 unità di lavoro con 3 core, richiederebbe lo stesso identico tempo di eseguirne una sola con un unico processore. Bisogna precisare che, per calcolare tale valore, è necessario conoscere il costo asintotico del programma, in modo tale da poter aumentare correttamente le dimensioni del problema, mantenendo costante il carico di lavoro per ogni processore.

4.2 Prestazioni dell'algoritmo proposto

Seguendo quanto presentato in 3.3, sono facilmente realizzabili quattro versioni di questo algoritmo: seriale, OpenMP, SIMD e OpenMP+SIMD;
OpenMP+SIMD è la versione implementata in questo progetto e comprende entrambe le parallelizzazioni spiegate in 3.4, mentre SIMD, sfrutta solo le *SIMD intrinsics* in fase di allineamento (con una solo unità di esecuzione), e OpenMP sfrutta le stesse direttive di OpenMP+SIMD, ma senza utilizzare istruzioni SIMD.
La figura 4.1 compara i tempi di esecuzione delle versioni appena descritte, eseguendo un allineamento su dieci immagini obiettivo con risoluzione 932 x 792 e offset pari a 30, mentre la 4.2 mostra i relativi speedup. La versione OpenMP+SIMD, che è stata lanciata con quattro threads (lo stesso vale anche per OpenMP), ovvero con il massimo numero di core dell'architettura, risulta chiaramente essere la più efficiente, dato che sfrutta entrambe le parallelizzazioni: si noti che la somma degli speedup di SIMD e OpenMP (rispettivamente 1.57 e 2.89) corrisponde all'incirca a quella di OpenMP+SIMD (4.38). Lo speedup ottenuto dalla versione più efficiente è *superlineare*: come

spiegato in 4.1, raramente si ottiene questo tipo di risultato, ma in questo caso il motivo è facilmente individuabile; ciò è dovuto allo sfruttamento delle istruzioni SIMD da parte di ogni thread: osservando infatti le prestazioni della versione OpenMP, si nota che lo speedup, senza tali istruzioni, non sarebbe stato nemmeno lineare.

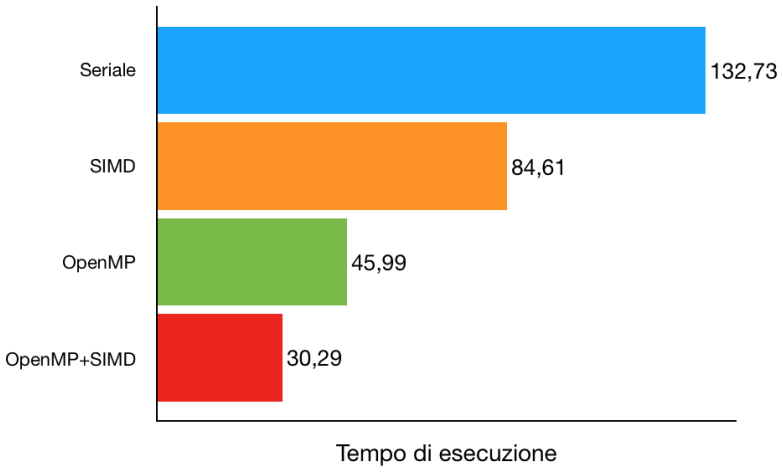

Figura 4.1: Confronto dei tempi di esecuzione tra quattro versioni dell'algoritmo proposto. Sono state allineate 10 immagini obiettivo di risoluzione 932 x 792 con offset pari a 30. Le versioni OpenMP e OpenMP+SIMD sono state lanciate con quattro threads.

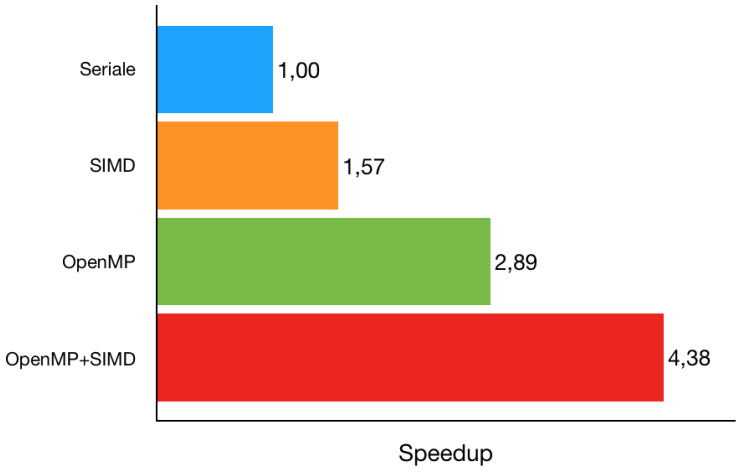

Figura 4.2: Confronto dello speedup tra quattro versioni dell'algoritmo proposto. Sono state allineate 10 immagini obiettivo di risoluzione 932 x 792 con offset pari a 30. Le versioni OpenMP e OpenMP+SIMD sono state lanciate con quattro threads.

Speedup

Il grafico in figura 4.3 rappresenta l'andamento dello speedup al variare del *numero di threads*, eseguendo la versione OpenMP+SIMD su 10 immagini obiettivo di risoluzione 932 x 792 e offset pari a 30. Lo speedup cresce in maniera più che lineare fino al raggiungimento dei quattro threads, soglia oltre il quale si mantiene abbastanza costante; ciò è dovuto all'hardware della macchina, che dispone di esattamente quattro microprocessori: infatti, quando il numero di threads supera quello dei processori disponibili, non è più possibile eseguirli tutti effettivamente in parallelo, perciò l'incremento dello speedup cala notevolmente, e spesso si verifica un peggioramento delle prestazioni (in questo caso avviene solo aumentando enormemente il numero dei threads).

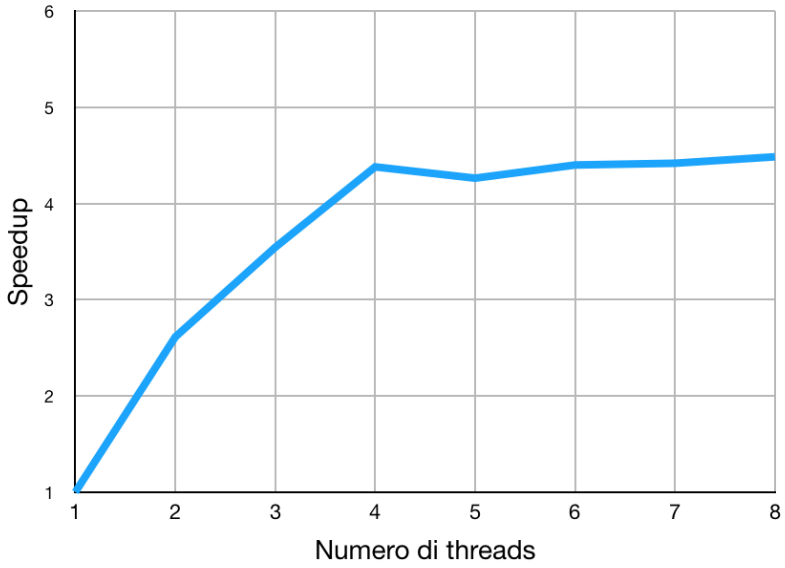

Figura 4.3: Andamento dello speedup al variare del numero di threads. Sono state allineate 10 immagini obiettivo di risoluzione 932 x 792 con offset pari a 30, lanciando la versione OpenMP+SIMD.

Risulta utile confrontare i tempi di esecuzione della versione seriale con quelli di OpenMP+SIMD (con 4 threads), al variare dei parametri che influenzano il carico di lavoro: *numero di immagini*, *offset* e *risoluzione*.

La figura 4.4 mostra il confronto tra i tempi di esecuzione della versione seriale e quelli di OpenMP+SIMD (con 4 threads), all'aumentare del *numero di immagini* obiettivo (con dimensioni 300 x 300 e offset pari a 30). L'andamento dei tempi di entrambe risulta essere abbastanza costante, probabilmente perché questo fattore influenza il numero di allineamenti da eseguire e il calcolo della media, ma non il carico di lavoro in fase di allineamento (che è quella con costo computazionale maggiore): dunque, dato che il tempo richiesto per allineare una singola foto non varia, un allineamento di 20 foto richiede all'incirca due volte il tempo richiesto per computarne 10 (mantenendo invariati gli altri parametri), sia per quanto riguarda la versione seriale,

sia per quella parallela.

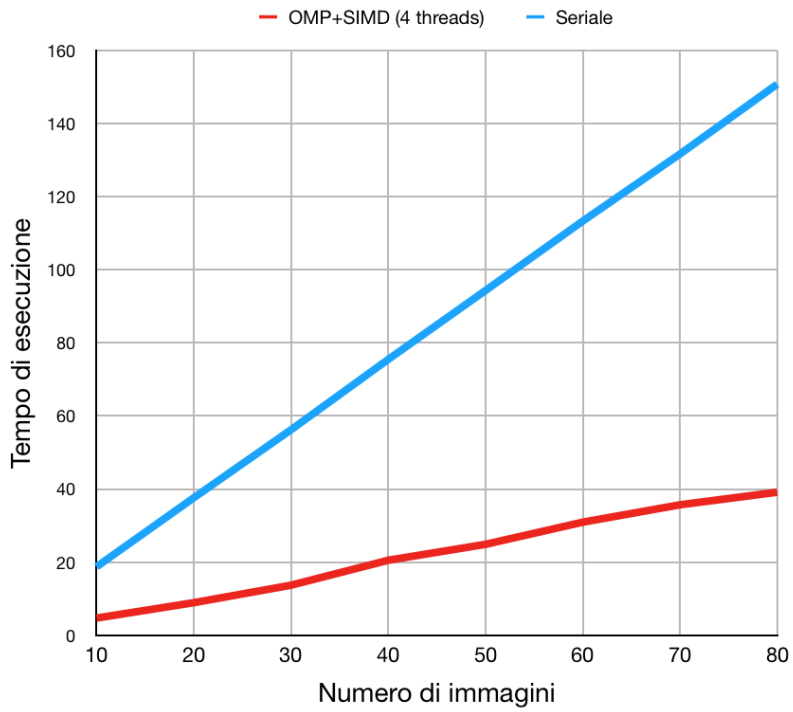

Figura 4.4: Confronto dei tempi di esecuzione della versione seriale con quelli di OpenMP+SIMD (con 4 threads), al variare del numero di immagini. Sono state allineate immagini di risoluzione 300 x 300 con offset pari a 30.

In figura 4.5 si può osservare l'andamento dei tempi di esecuzione, sia per la versione seriale che per OpenMP+SIMD (con 4 threads), al variare dell'*offset* (con 50 immagini obiettivo di risoluzione 300 x 300). Il divario tra le versioni tende a crescere in maniera sempre maggiore: questa volta il tempo di calcolo del singolo allineamento varia, dato che per ogni immagine sarà necessario eseguire un numero maggiore di confronti, perciò l'aumento del tempo di esecuzione non avviene in maniera costante (come nel caso precedente) tra le due versioni.

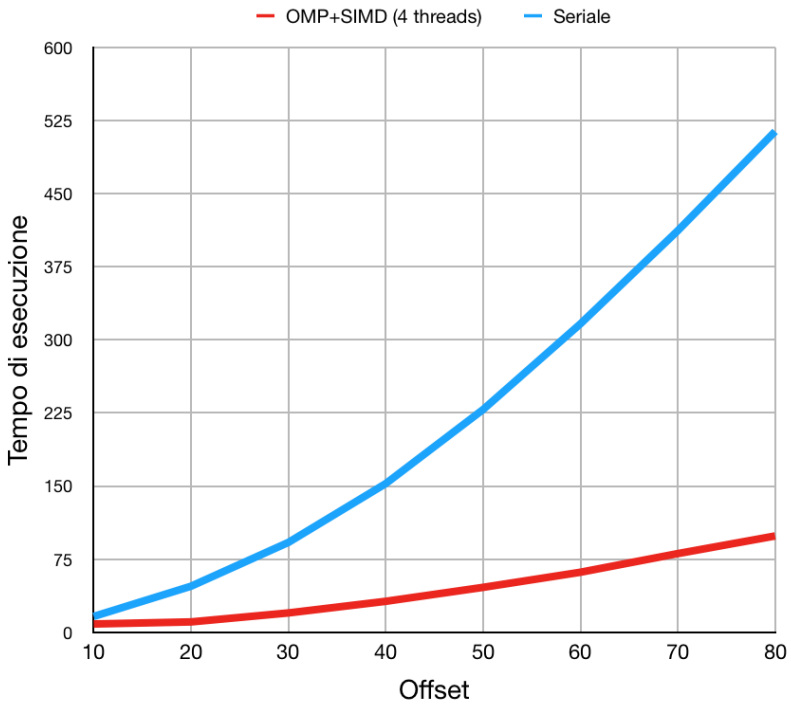

Figura 4.5: Confronto dei tempi di esecuzione della versione seriale con quelli di OpenMP+SIMD (con 4 threads), al variare dell'offset. Sono state allineate 50 immagini obiettivo di risoluzione 300 x 300.

Un comportamento simile si verifica incrementando la *risoluzione* dell'immagine: in figura 4.6 viene mostrato il confronto tra i tempi di esecuzione della versione seriale e quelli di OpenMP+SIMD (con 4 threads), allineando 50 immagini obiettivo (quadrate) con offset uguale a 30. Anche in questo caso la discrepanza tende a crescere sempre più: incrementando questo fattore non si influenza solo la fase di allineamento (nella quale le istruzioni SIMD possono essere sfruttate maggiormente), ma anche tutte le parti del codice parallelizzate con le clausole OpenMP che operano sui pixel dell'immagine.

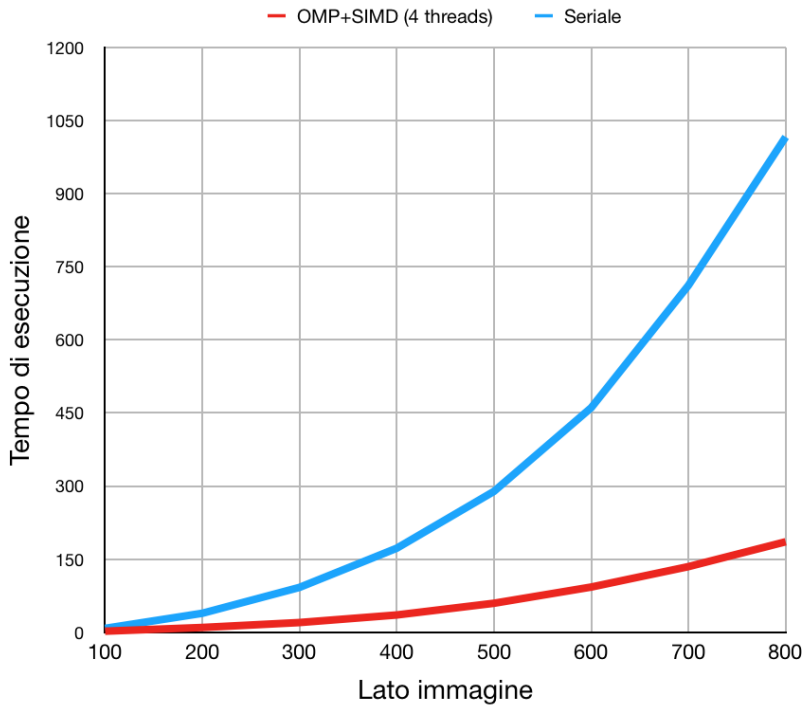

Figura 4.6: Confronto dei tempi di esecuzione della versione seriale con quelli di OpenMP+SIMD (con 4 threads), al variare della risoluzione. Sono state allineate 50 immagini obiettivo (quadrate) con offset pari a 30.

Strong Scaling Efficiency

Per quanto riguarda l'efficienza, nella figura 4.7, viene mostrata la *Strong Scaling Efficiency* relativa ai dati dello speedup calcolati precedentemente (in figura 4.3): la $E(p)$, fino a quattro threads, è ottima e assume addirittura valori superiori ad uno, poichè lo speedup ottenuto è superlineare; superata questa soglia, il valore tende a calare sempre più: lo speedup infatti è più o meno costante in corrispondenza di tali valori.

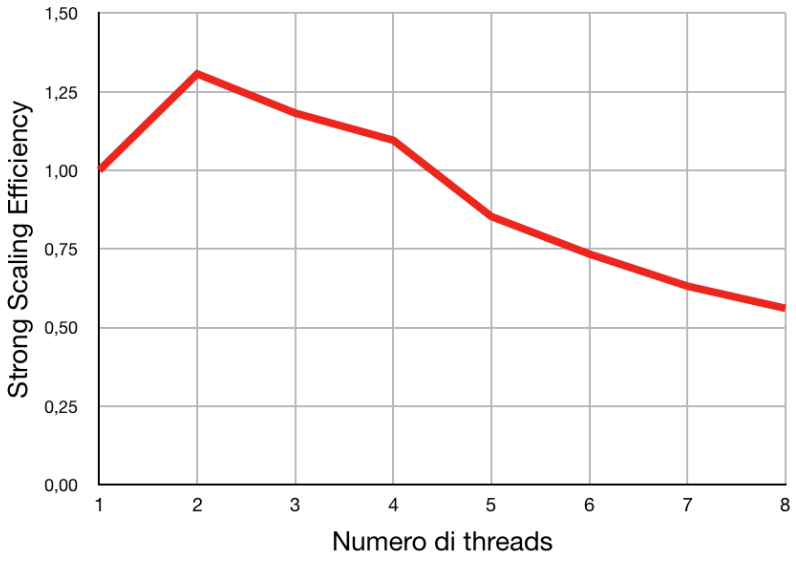

Figura 4.7: Andamento della Strong Scaling Efficiency. Sono state allineate 10 immagini obiettivo di risoluzione 932 x 792 con offset pari a 30, lanciando la versione OpenMP+SIMD.

Weak Scaling Efficiency

L'ultimo fattore rimasto da analizzare è la *Weak Scaling Efficiency*. In figura 4.8 viene mostrata la $W(p)$, calcolata (dalla versione OpenMP+SIMD) su 10 immagini obiettivo di dimensioni 932 x 792, con l'offset che varia secondo la formula:

$$\frac{\sqrt{441 * N_{threads}} - 1}{2}$$

dove 441, è il numero di confronti da effettuare per ogni immagine con offset pari a 10 (il valore associato all'esecuzione con un thread).
Dato che la fase di allineamento è quella dal costo computazionale maggiore, per mantenere il carico di lavoro di ogni thread costante, bisogna variare l'offset in modo tale da far eseguire ad ognuno circa 441 confronti per immagine; una volta determinato l'offset, il numero di confronti da effettuare viene calcolato come $(offset * 2 + 1)^2$: da quest'ultima è facile ricavare la

formula fornita prima.
Analogamente a quanto visto per la $E(p)$, il valore è buono fino al solito numero di threads, dopodiché tende a calare sempre più.

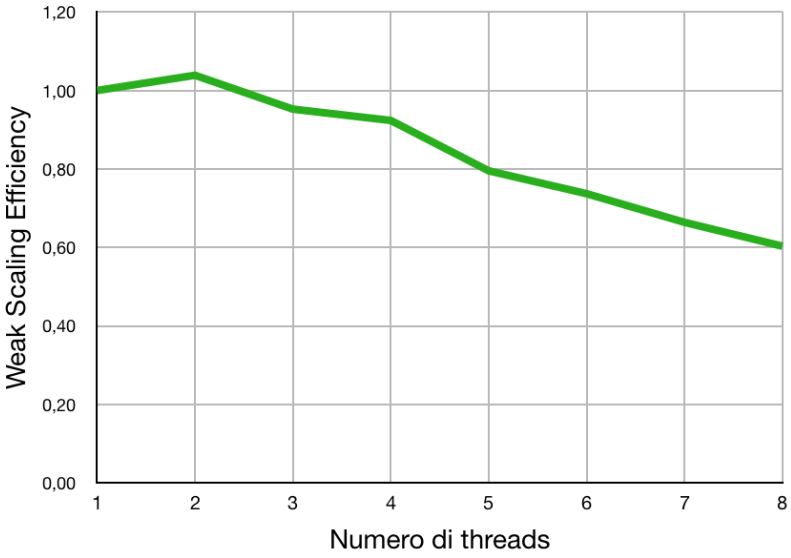

Figura 4.8: Andamento della Weak Scaling Efficiency. Sono state allineate 10 immagini obiettivo di risoluzione 932 x 792 con offset pari a 30, lanciando la versione OpenMP+SIMD.

Conclusioni

In questa tesi abbiamo affrontato il problema dell'*image registration*, ovvero l'allineamento di una o più immagini (dette obiettivo) rispetto ad un'altra (detta riferimento), considerandone in particolare l'applicazione nel campo dell'astrofotografia. Dopo aver introdotto tutti i concetti necessari allo sviluppo di un programma parallelo, e fornito alcune informazioni riguardanti l'architettura usata (Raspberry Pi), abbiamo visto l'implementazione di un semplice algoritmo parallelo basato su una tecnica di *brute force*; il programma realizzato (il cui codice è consultabile al seguente indirizzo `https://bitbucket.org/alessandroretico/astronomical-image-registration`) sfrutta le istruzioni SIMD e le direttive OpenMP di un Raspberry Pi, per allineare, considerando esclusivamente la traslazione, più immagini obiettivo e produrre una rappresentazione migliore dell'immagine riferimento, calcolando la media dei valori dei pixel delle immagini disponibili.
A seguito dei risultati ottenuti, si può affermare che questa architettura è adeguata per applicazioni parallele di dimensione contenuta come quella realizzata in questo progetto.

Per migliorare quanto proposto sarebbe opportuno permettere la gestione di vari formati di immagine (non solo BMP) e provare altre tecniche di allineamento, in grado di portare a termine correttamente l'operazione, anche in presenza di rotazione e scalatura tra le immagini.
Un'altra interessante modifica che si può apportare riguarda la tecnica di parallelizzazione adottata: si potrebbe provare a sfruttare la GPU del Rasp-

berry e comparare i risultati conseguiti in questa maniera, con quelli ottenuti da questa tesi.
Come ulteriore sviluppo futuro, si potrebbe realizzare un sistema che automatizza le fasi di cattura ed elaborazione delle immagini (includendo magari anche qualche altro algoritmo utile ad ottenere un'immagine migliore).

Bibliografia

[1] Michael A. Covington. *Digital SLR Astrophotography*. Cambridge University Press, 2 edition, 2018. doi: 10.1017/9781316996799.

[2] Sayan Nag. Image registration techniques: A survey. *CoRR*, abs/1712.07540, 2017.

[3] W R Crum, T Hartkens, and D L G Hill. Non-rigid image registration: theory and practice. *The British Journal of Radiology*, 77(suppl_2):S140–S153, 2004. PMID: 15677356.

[4] R. Ezzeldeen, H. Ramadan, T. Nazmy, M. Adel Yehia, and M. Abdel-Wahab. Comparative study for image registration techniques of remote sensing images. *The Egyptian Journal of Remote Sensing and Space Sciences*, 13(1):31–36, November 2010.

[5] B. Srinivasa Reddy and B. N. Chatterji. An fft-based technique for translation, rotation, and scale-invariant image registration. *IEEE Transactions on Image Processing*, 5(8):1266–1271, August 1996.

[6] Raffaele Cappelli. Fondamenti di elaborazione di immagini - estrazione dei bordi e segmentazione. Ingegneria e scienze informatiche - Università di Bologna.

[7] Siril: `https://www.siril.org`

[8] RealVNC: `https://www.realvnc.com/en/connect/download/viewer`

[9] Kevin Doucet and Jian Zhang. Learning cluster computing by creating a raspberry pi cluster. pages 191–194, 04 2017. doi: 10.1145/3077286.3077324.

[10] VideoCore IV 3D Architecture Reference Guide: `https://docs.broadcom.com/docs/12358545`

[11] Raspberry Pi 2 Model B specification: `https://www.raspberrypi.org/products/raspberry-pi-2-model-b`

[12] Barbara Chapman, Gabriele Jost, and Ruud van der Pas. *Using OpenMP Portable Shared Memory Parallel Programming.* Scientific and Engineering Computation. The MIT Press, October 2007. ISBN: 9780262533027.

[13] Ruud van der Pas, Eric Stotzer, and Christian Terboven. *Using OpenMP - The Next Step Affinity, Accelerators, Tasking, and SIMD.* Scientific and Engineering Computation. The MIT Press, October 2017. ISBN: 9780262534789.

[14] *Single-Instruction Multiple-Data Execution.* Synthesis Lectures on Computer Architecture. Morgan & Claypool, May 2015. ISBN: 9781627057639.

[15] Neon intrinsics reference: `https://developer.arm.com/architectures/instruction-sets/simd-isas/neon/intrinsics`

[16] Microsoft Documentation Bitmap Storage: `https://docs.microsoft.com/en-us/windows/win32/gdi/bitmap-storage`