Il calcolo parallelo nell'Image Processing

Antonella Galizia

27 maggio 2004

Indice

1 Introduzione all'Image Processing

L'elaborazione di immagini é una disciplina che negli ultimi decenni ha assunto una notevole importanza, avendo avuto un forte impulso dall'affermarsi di tecnologie digitali che permettono l'elaborazione di segnali a piú dimensioni; d'altro canto lo sviluppo di tale disciplina é supportato anche nell'importanza delle sue applicazioni, che abbracciano campi molto diversi, dalla medicina all'industria, dal militare alla meteorologia, o più in generale all'osservazione della terra. La diversità delle problematiche che si presentano nei diversi settori, ha portato allo sviluppo di molte tecniche di analisi e di elaborazione specifiche, dedicate a serie di immagini con particolari caratteristiche, o di cui si cerca di evidenziare solo qualche determinata peculiarità. Per citare solo alcuni dei possibili campi di applicazione possiamo trovare:

- *Problemi militari*: il tracciamento di obiettivi da dati video o il riconoscimento di bersagli da sequenze di immagini ottenute all'infrarosso;
- *Problemi industriali*: il monitoraggio di processi industriali in tempo reale o la visione di robot utilizzati nelle catene di produzione;
- *Problemi commerciali*: la compressione di segnali televisivi;
- *Problemi di meteorologia*: l'analisi dello spostamento delle nubi;
- *Problemi di oceanografia*: l'analisi dei mari e degli oceani sotto l'aspetto biologico e dinamico;
- *Problemi di trasporti*: il monitoraggio del traffico.
- *Problemi medici*: :)

Volendo formalizzare un pó il concetto, si parla di **elaborazione numerica di un'immagine** intendendo una serie di azioni compiute da appositi algoritmi su un'immagine numerica per modificarla in modo da raggiungere un prefissato obiettivo, producendo un risultato che é ancora un'immagine. Esempi possono essere il miglioramento di immagini, estrazione di contorni, riduzione del rumore, etc.
In realtá l'aver definito come dato di output un'immagine é un elemento che restringe un pó il campo d'azione, in quanto esistono altre discipline, (spesso semplicemente classificate come altri settori di image processing!) che elaborano immagini restituendo in output dati differenti. Per esempio vengono distinti altre due categorie:

1. *Analisi di immagini* (image analysis) in cui l'ouput é costituito soprattutto da misure, quali ad esempio: misura di gradienti, individuazione di punti angolosi, fotometria, etc!

2. *Comprensione di immagini* (image understanding) in cui l'ouput é costituito da descrizione ad alto livello, ad esempio il riconoscimento della presenza di un volto nell'immagine analizzata, etc!

Più in generale molti settori vengono considerati image processing e si differenziano a seconda dello scopo dell'elaborazione, oppure esitono ancora altre discipline e aree affini che sono considerate elementi un pò di frontiera, solo per citarne alcuni:

- Compressione di immagini (image coding);
- Riconoscimento di forme (pattern recognition);
- Trattamento di video (video processing).
- Analisi di scene (scene analysis);
- Visione artificiale (computer vision);
- Realtá virtuale (virtual reality);
- Grafica computerizzata (computer graphics).

Ad esempio quest'ulitma non é proprio un'elaborazione di immagini ma produce immagini da un insieme di dati che rappresentano punti nello spazio, colori, tessitura, etc etc.

2 Esigenza di High Performance

Tutte queste discipline, e numerose altre, processano una grossa quantitá di dati ed ovvimanente questo mette pone dei problemi alla velocitá di calcolo in un'elaborazione. Spesso questa é una componente fondamentale per la computazione; per esempio in aree quali la difesa del territorio, oppure motori di ricerca o ancora tutte quelle discipline basate sulla simulazione dei fenomeni mediante la descrizione di modelli computazionali, risposte che giungono "in ritardo" non sono neanche utilizzabili. Facciamo un pò di conti su esempi pratici quali possono essere la metereologia o il Car Crash Test. Infatti in metereologia l'elaborazione di previsioni per 2 giorni consecutivi, utilizzando un normale PC, richiede 23 giorni di computazione, oppure per il Car

Crash Test, sono richiesti circa 7 mesi di elaborazione sempre avvalendosi di un semplice PC! Abbiamo calcolato questi valori assumendo che un semplice PC con potenza di calcolo di un GFLOPS (10^9 operazioni floating-point al secondo) se invece passassimo ad una macchina un pó piú veloce, che per esempio abbia potenza di calcolo un TFLOPS (10^{12} operazioni floating-point al secondo) riusciremmo a prevedere il tempo in 33 minuti e a simulare un Crash in 6 ore! Ed abbiamo semplicemente utilizzato macchine più veloci.
Velocizzare le macchine però è possibile fino ad un certo punto in quanto esistono dei limiti tecnologici che non si possono aggirare! Per esempio, per poter avere una macchina con potenza di calcolo di un TFLOPS (ipotesi precedentemente utilizzata!) si deve avere un accesso ai dati in memoria per ogni ciclo di clock. Per potervi riuscire é necessario che le componenti hardware del calcolatore non distino per piú di 3 cm; ciò comporta notevoli problemi di packaging e di raffreddamento! Anche per le ripercussioni che avrebbe sulla memoria fisica. Dobbiamo migliore le tecnologie finché possiamo, ma a questo punto diventa necessario lavorare anche su altri fronti!

3 Il calcolo parallelo

La soluzione ai problemi di High Performance risulta essere sempre piú spesso il calcolo parallelo: il concetto é molto semplice, e detto in maniera molto spicciola, puó essere visto come il suddividere il lavoro attraverso piú unità di calcolo, dati i limiti tecnologici su cui abbiamo giá detto qualcosa! Spieghiamoci con un esempio molto molto semplice:
La costruzione di una casa. Se avessimo un solo operaio a costruire la nostra casa...forse non ci andremmo mai ad abitare! Se ci rivolgiamo un'impresa di operai, la costruzione sará piú veloce!
Il concetto quindi é quello di lavorare su diversi livelli, e gestirne le problematiche derivanti, il che equivale alla scomposizione del nostro problema di oridine N in una serie di P sottoproblemi di ordine N/P che sono risolti contemporaneamente eventualmente su piú calcolatori.
In primo luogo é doveroso distinguere le differenti forme di parallelismo, anche le piú note e citate sono sostianzialmente 2, che sono quelle che riportiamo, riprendendo l'esempio della costruzione di una casa e la ditta di operai.

Task parallelism

Nel parallelismo a livello di task ogni unità di calcolo, svolge differenti operazioni sugli stessi dati, eseguono contemporaneamente fasi successive dello stesso lavoro sugli stessi elementi, mettendo in piedi una **pipeline**. Ricordiamo che questo concetto si trova alla base della realizzazione delle architetture

RISC e dei calcolatori vettoriali. Nell'esempio questo si realizza mendiante una catena di montaggio: per costruire un muro uno degli operai stende il cemento, un altro poggia i mattoni ed un altro li livella, come nella bellissima figura 1!

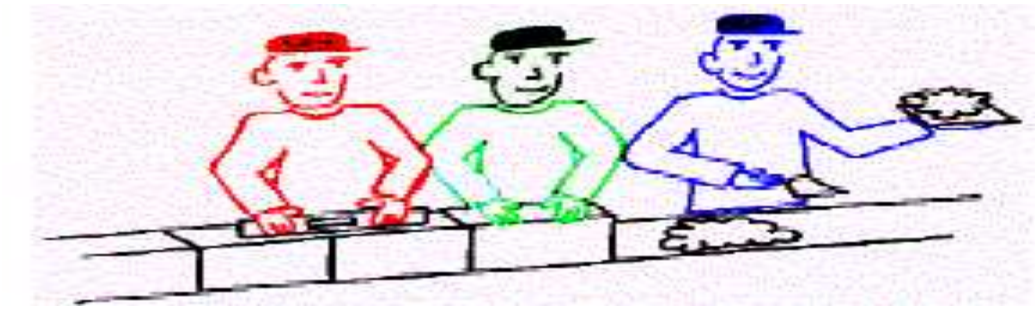

Figura 1: *L'esempio nel caso del Task Parallelism!!! :)*

Data parallelism

Il parallelismo a livello dati si esprime invece in maniera differente, infatti in questo caso ogni unità di calcolo elabora sempre la stessa operazione ma applicandola a dati diversi. Riportandoci al solito esempio, stiamo dicendo che i nostri operai svolgono tutti la stessa operazioni, per esempio la costruzione del muro, ma su mattoni diversi, sempre come la meravigliosa figura 2. Questo concetto invece é alla base dei calcolatori SIMD (evoluti poi in MIMD, banalizzandone un pò il percorso!).

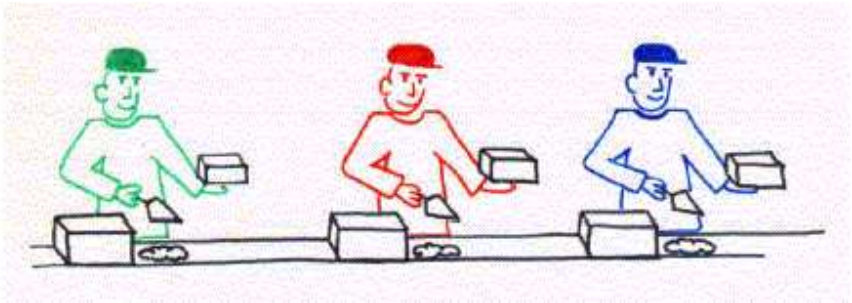

Figura 2: *L'esempio nel caso del Data Parallelism!!! :)*

Possiamo riportare esempi di discipline a cui si applica meglio un tipo di

parallelismo piuttosto che un altro, per esempio nel calcolo degli *ecosistemi* delle popolazioni tenuta in conto la presenza delle varie specie, oppure nella manipolazione di *segnali audio* il task parallelism puó risultare piú indicato ed efficiente, mentre *nell'image processing*, in settori quali la *valutazione dei danni da inquinamento*, oppure nel *gioco delgli scacchi* (!) si sfrutta meglio il parallelismo se operato a livello dati.
L'andamento generale dell'ultimo periodo é quello di preferire sempre piú spesso il data parallelism, probabilmente anche per una questione di tecnologie e di costi, essendo particolarmente adatta per lavorare su cluster beowulf o di workstation (esempi di calcolatori MIMD a memoria distribuita). Ad ogni modo non é questa la sede per parlare ancora di questo tipo di programmazione, delle politiche e delle problematiche che si pongono, ma da questa breve introduzione riusciamo comunque a cogliere la necessitá di costruire strumenti per permettere ai processi -unitá di calcolo- di comunicare tra loro, eventualmente sincronizzarsi o scambiarsi dati, e gestire il tutto in maniera opportuna, creando algoritmi paralleli che siano comunque consistenti e ci portino agli stessi risultati degli algoritmi sequenziali. In particolare nel data parallelism che computi su calcolatori MIMD la libreria MPI é diventata uno standard per la comunicazioni tra processi.
Per sottolineare ancora che il fine ultimo del calcolo parallelo é diminuire i tempi di calcolo, senza voler formalizzare una teoria sull'efficienza in ambiente parallelo, sembra comunque necessario dare la definizione di almeno un parametro che misuri intuitivamente la bontá della parallelizzazione effettuata! Tale paramentro viene chiamato *speed-up* e misura la riduzione del tempo di esecuzione, ovvero l'aumento della velocitá di esecuzione rispetto all'algoritmo sequenziale, per cui intuitivamente quanto ci é convenuto parallelizzare l'algoritmo.

Speed-up

Formalizzando supponiamo di avere T_1 il tempo di esecuzione dell'algoritmo ottimale nella risoluzione sequenziale, e indichiamo con T_p il tempo di esecuzione del problema su p processi, lo speed-up é dato dal rapporto:

$$S_p = \frac{T_1}{T_p}$$

Lo speed-up ideale é rappresentato da p stesso, basta pensarci un attimo: se impiego un'ora per fare la somma di due vettori in sequenziale, vorrei che considerando due calcolatori tale tempo si dimezzasse! Ribadiamo che questa é ovviamente la situazione ideale, in quanto è vero che si è recuperato del tempo nell'elaborare una quantitá di dati minore, ma se ne perde altro nella fase di comunicazione con gli altri processi! Per questo é bene avere un

parametro di riferimento per capire se vale la pena di utilizzare 16 processori per calcolare la somma di due vettori oppure per simulare un car crash! In piú tale parametro ci permette di vedere se il nostro algoritmo parallelo "scala" ripetto al numero di processori, e rispetto alle dimensioni del problema!

4 Parallel Image Processing

Come giá accennato, la forma di parallelismo che in maniera naturale si adatta all'image processing é il data parallelism. Vogliamo qui presentare il nostro modo di coniugare programmazione parallela ed image processing mediante lo studio di una libreria parallela di image processing che stiamo utilizzando. In particolare la libreria si basa sul concetto di *parallelismo trasparente*, cioé nascosto all'utente; infatti l'interfaccia utente della libreria é tale da sembrare assolutamente sequenziale ed il parallelismo é gestito in maniera autonoma dalla libreria stessa, che si avvale del supporto di politiche di ottimizzazione. La gestione autonoma del parallelismo da parte della libreria semplifica di molto il lavoro dell'utente e risulta quindi una caratteristica molto valida per ad esempio un ricercatore di elaborazione di immagini non necessariamente anche un esperto di calcolo parallelo!

5 La libreria ParHorus

La libreria ParHorus, nasce da un gruppo di ricerca dell'Università di Amsterdam, é un progetto che parallelizza la già esistente libreria Horus, sviluppata dallo stesso gruppo di ricerca. Implementata in C ed MPI, é reperibile sul sito:

http://carol.wins.uva.nl/~fjseins/ParHorusCode/

Vi si trovano operazioni comunemente utilizzate nell'image processing, definizioni di tipi di dati astratti e le relative operazioni per manipolarli. Per la classificazione delle operazioni di image processing si é considerata quella operata da G.X. Ritter e J.N. Wilson in *Computer Vision Algorithms in Image Algebra*, (CRS Press, Inc, 2000) dove si assume che esistano un numero di possibilità limitato per processare i pixel al fine di ottenere delle operazioni utili nell'ambito dell'image processing.
Le politiche di ottimizzazione automatica del codice parallelo si affiancano alla fase di calcolo; questa costruzione é ottenuta mediante la definizione di

un modello di performance che é progettato sulla base del concetto di APIPM (Abstract Parallel Image Processing Machine).

5.1 La rappresentazione di un'immagine digitale

Un'immagine digitale viene considerata come un set di pixel. Associato ad ogni pixel c'é una locazione (punto) ed un valore (pixel). La nostra notazione sarà di questo tipo: il valore di un pixel di un'immagine **a** nella locazione **x** é rappresentato da **a(x)**. L'insieme di tutte le locazioni é considerato il *dominio* dell'immagine, tipicamente questo risulta essere un sottoinsieme di Z^n con $n = 1, 2, 3$. Tale sottoinsieme é in ogni direzione contenuto in un rettangolo. L'insieme dei valori dei pixel é chiamato *range* dell'immagine e é denotato con F. Il valore dei pixel é costituito da un vettore di m scalari con $m = 1, 2, 3$, e tali scalari sono rappresentati da tipi comuni di dati (int, double) oppure da valori complessi. In questo modo é possibile lavorare con immagini con differenti rappresentazioni (livelli di grigio, RGB, etc).

5.2 Algoritmic patterns

Sempre basandosi sulla classificazione delle operazioni fatta in *Computer Vision Algorithms in Image Algebra*, la libreria si fonda sul concetto di *algoritmic patterns* che potremmo tradurre come *modelli algoritmici* e pensarli proprio come delle classi di operazioni, che ricoprono le piú comuni operazioni di image processing e sono raggruppate per similarità di comportamento. Ció significa che per ogni operazione di image processing esiste un modello algoritmico su cui essa si mappa, e di conseguenza ogni operazione viene implementata istanziando tale modello algoritmico con dei propri specifici parametri, compresa la funzione che si deve effettuare!
Per chiarire le idee consideriamo un esempio, abbiamo bisogno di calcolare il valore assoluto di una data immagine. Questa funzione dal punto di vista matematico é una funzione unaria, che data l'immagine di input, applica la funzione valore assoluto in modo puntuale, cioé applicando la ad ogni singolo pixel; cercando il modello algoritmico da associare, posso considerare la *funzione unaria* da applicare ai pixel dell'immagine data in input. Allora applico la funzione valore assoluto, istanziando il modello algoritmico "operazione unaria", in cui specifico che voglio calcolare la funzione valore assoluto, oltre ovviamente all'immagine di input!
Volendo mostrare il codice di tale operazione, é necessario premettere un pó di definizioni di tipi di dati che la libreria utilizza. Per chiarezza ho messo tutto in appendice! Vedi §A.1 per le definizioni delle strutture dati, per l'esempio del programma che calcola il valore assoluto vedi §A.2 e per chi é

curioso ho riportato in §A.3 la funzione valore assoluto!
Tornando ai modelli algoritmici, nella versione attuale della libreria ne sono stati implementati sei differenti livelli:

- *Unary pixel operation* Operazione in cui una funzione unaria é applicata ad ogni pixel dell'immagine. Esempi: negazione, valore assoluto, radice quadrata.
- *Binary pixel operation* Operazione in cui una funzione binaria é applicata ad ogni pixel dell'immagine. Esempi: addizioni, moltiplicazioni, soglia.
- *Reduce operation* Operazione in cui tutti i pixel di un'immagine sono combinati per ottenere un singolo valore risultante. Esempi: somma, prodotto, massimo.
- *Neighborhood operation* Operazione in cui diversi pixel nell'intorno di ogni pixel nell'imagine vengono combinati insieme. Esempi: percentile, mediana.
- *Generalized convolution* Casi speciali delle *Neighborhood operation*. La combinazione dei pixel nell'intorno di ogni pixel é espressa in termini di due funzione binarie. Esempi: convoluzioni, erosioni, dilatazioni.
- *Geometric (domain) operation* Operazioni in cui il dominio dell'immagine é trasformato. Esempi: traslazioni, rotazioni, scalatura.

Anche qui ho messo degli esempi di implementazione in appendice! In particolare ho riportato un esempio di Unary pixel operation (il valore assoluto!), un esempio di Binary pixel operation (somma di un valore) ed in esempio di Reduce operation (calcolo del massimo), il tutto in §A.3.

5.3 Parallelizable pattern

Per ogni modello algoritmico è stato ricavato un così detto *parallelizable pattern*, che potremmo tradure con *modello parallelo*. Ogni pattern é definito come la massima quantitá di lavoro che in un modello algoritmico può essere portata avanti senza imbattersi nella necessità di comunicare con gli altri processi. In altre parole in un parallelizable pattern tutti gli accessi ai dati si riferiscono a dati locali del nodo su cui si sta compiendo la computazione, risulta quindi essere un'operazione generica sequenziale che prende zero o piú strutture in input e produce una struttura di destizione come output. Parliamo in maniera generica di strutture dati in quanto nella libreria sono

previste principalmente 3 strutture dati differenti: IMAGE, KERNEL, MATRIX; della prima struttura abbiamo giá detto qualcosa, mentre le altre due strutture dati sono utilizzate rispettivamente per le convoluzioni e le operazioni geometriche. Per le definzioni vedi sempre in appendice §A.1!
A questo punto un algoritmic pattern risulta la concatenazione di operazioni di base della memoria (allocazione, copie di dati), di fasi di comunicazioni tra processi e di singoli parallelizable pattern, come riportato in figura 3.

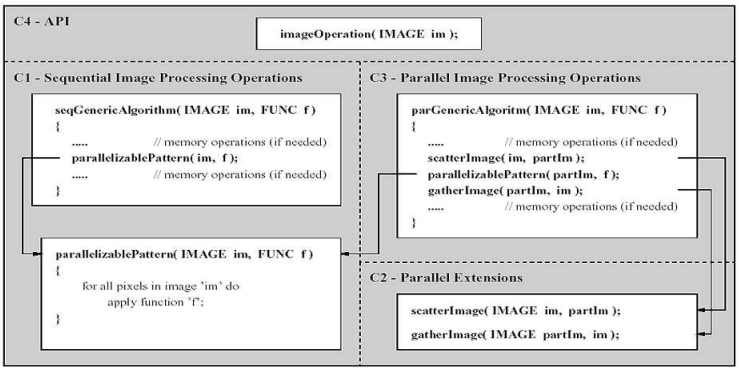

Figura 3: *Struttura logica di un generico programma parallelo!*

Ogni pattern é costituito da n task indipendenti, che definirei come delle semplici operazioni; per ognuna di queste si specificano quali sono dati di una struttura che devono essere acquisiti (read) al fine di modificare (write) il valore del singolo dato nella struttura di destinazione.
In un task é bene sottolineare che: per gli accessi in lettura alle strutture in input non si hanno restrizioni, ma non é possibile utilizzare strutture dati diverse da quelle specificate; per gli accessi in lettura sulle strutture in output sono limitati al solo punto da modificare e nessun altro.
Ogni task é legato ad una differente "task location" x_i con $i \in \{1, 2, \ldots, n\}$, che è determinato dalle posizioni accedute in tutte le strutture dati che l'operazione coinvolge. In base alle modalitá di questi accessi, si ha la definizione dei *Data access pattern types* che vengono classificati in:

1. *One-to-one*: Per una data struttura dati, in ogni task $T_i, i \in \{1, 2, \ldots, n\}$ si accede solo al punto x_i.

2. *One-to-one-unknown*: Per una data struttura dati, in ogni task $T_i, i \in \{1, 2, \ldots, n\}$ si accede solo al punto ma questa volta non uguale ad x_i.

3. *One-to-M*: Per una data struttura dati, in ogni task $T_i, i \in \{1, 2, \ldots, n\}$ si accede solo ai punti appartenenti ad un intorno di x_i.

4. *Other*: Per una data struttura dati, in ogni task $T_i, i \in \{1, 2, \ldots, n\}$ si accede a tutti i dati della struttura, oppure gli accessi sono irregolari o sconosciuti.

quindi questa classificazione ribadiamo serve solo per i dati delle strutture di input, in quanto per i dati delle strutture di output, con questa nuova terminologia sono permessi solo accessi del tipo *One-to-one* oppure il *One-to-one-unknown* ma non di piú!
A questo punto si capisce che per ogni parallelizable pattern é necesserio specificare qual'é il tipo di accesso per ogni struttura dati coinvolta nel calcolo! Nella figura 4 presentiamo due situazioni che possono essere individuate.

- *Type 1*
  - Il tipo di accesso ai dati é *One-to-one*;
  - nessun vincolo é imposto sulla concatenazione delle operazioni.
- *Type 2*
  - Il tipo di accesso ai dati é *One-to-one-unknown*
  - il risultato finale dell'operazione deve essere tale da potersi considerare come l'applicazione di una funzione *f()* associativa e commutativa.

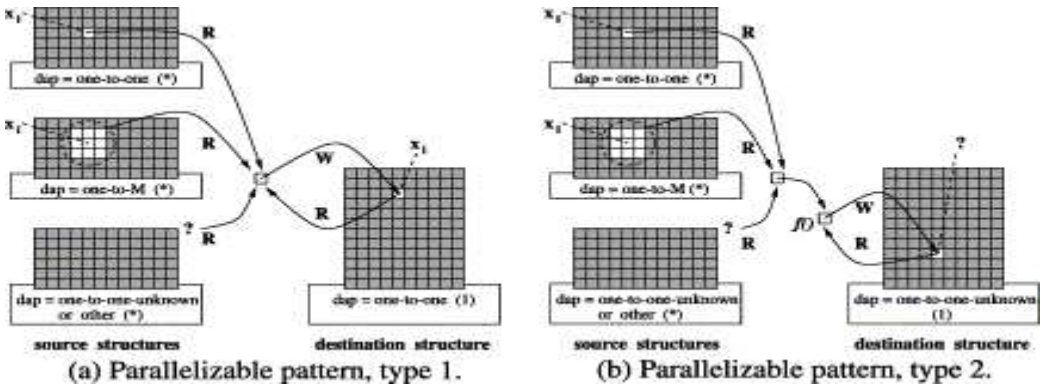

Figura 4: *Due tipi di pattern type!!!*

Ovviamente queste due tipologie non esauriscono l'insieme delle possibili operazioni! Passiamo ora a degli esempi pratici, proviamo a spiegare come si parallelizzano le operazioni, per fare questo saremo poco formali, avvalendoci soprattutto di figure e poi in appendice abbiamo riportato i codici.

5.4 Esempi di operazioni parallele

Riportiamo un esempio per entrambi i tipi di task citato, premettiamo che in una computazione parallela le strutture dati coinvolte nella computazione sono ottenute solamente in tre modi:

1. scatter senza regioni di bordo per ogni struttura dati con un accesso del tipo uno a uno;

2. scatter le regioni di bordo per ogni struttura dati con un accesso del tipo uno a M;

3. broadcast, in questo modo tutti i processi hanno tutta la struttura dati; in alternativa questa puó essere calcolata localmente.

Quanto detto si capirá meglio con gli esempi!

5.4.1 Operazione di riduzione

Una operazione di riduzione calcolata su un'immagine **a** produce come output uno scalare o un vettore di valori in F, range dell'immagine (insieme dei valori assunti dai pixel!); quindi in maniera piú esplicita tale vettore ha la stessa dimensione della variabile PIXEL. Esempi di questo tipo di operazioni possono essere il prodotto o la somma, oppure il calcolo del minimo o massimo, di tutti PIXEL presenti nell'immagine considerata. Ometto qualsiasi tipo di definizione matematica, che complicherebbe un pó il discorso!!!!! Risulterá piú semplice spiegare bastandosi sulla figura 5!

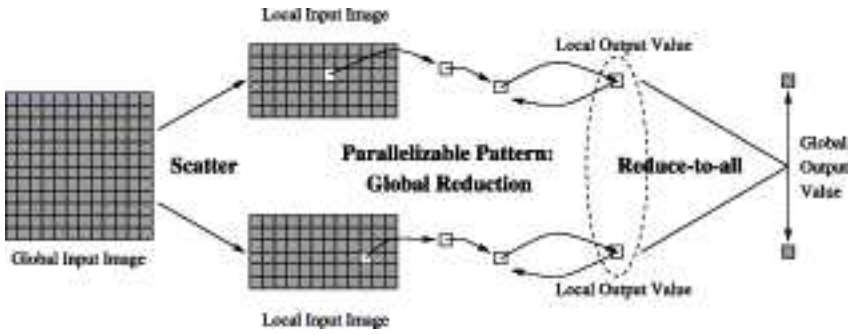

Figura 5: *Schema delle operazioni di riduzione!*

Come si puó vedere dalla figura 5, l'immagine viene scatterata tra i nodi di calcolo, nell'esempio questi sono 2. Il tipo di accesso all'immagine di input

é *one-to-one*, mentre al signolo valore del risultato é del tipo *one-to-one-unknown*. Ogni nodo calcola quanto deve (max, min, $\sum$, $\prod$) e poi i risultati parziali devono essere messi insieme. Per cui ci sará l'esigenza di un'ulteriore comunicazione e poi la computazione finale, in modo da avere che tutti i nodi convano il risultato!

5.4.2 Convoluzione generalizzata

Anche qui volendo dare la definizione di convoluzione generalizzata, forse non rimarreste svegli, per cui dico solo che risulta essere il prodotto convolutorio dell'immagine di input **a** e di un dato kernel t che in realtá caratterizza il tipo di convoluzione. L'output di tale operazione é un'immagine.
Piú in generale un prodotto di convoluzione di due funzioni si definisce come:

$$f * g = \int_0^t f(\tau)\, g(t-\tau) d\tau$$

Questo significa che ogni pixel dell'immagine di output dipende dai valori dei pixel dei suoi *vicini*, cioé dei pixel appartenenti ai suoi intorni, peró ovviamente nell'immagine in input, oltre che dal kernel t e dalla sua struttura! Si preferisce di solito tenere separata la parte di manipolazione dei "bordi" dalla parte di convoluzione, in quanto questo comporta robustezza e velocitá dell'algoritmo. In piú questo approccio computazionale comporta il vantaggio che l'algoritmo generico puó essere implementato proprio come il suo modello parallelo! Esistono diverse strategie di trattamento dei bordi, per esempio il mirroring o il tiling, ovviamente questo dipende dal tipo di convoluzione considerata.
Per poter operare sugli intorni é necessario dotare tutti i pixel dei loro vicini, questo diventa difficile quando le immagini sono spezzettate attraverso i processori! Una tecnica normalmente utilizzata é quella di allocare direttamente un'immagine "un pó piú grande", quel tanto da riuscire a memorizzare anche il bordo in piú di cui abbiamo bisogno, e poi aggiornarlo scambiandolo con gli altri processori nel momento opportuno (per esempio dopo un aggiornamento!). Le regioni di bordo vengono dette *ghost region*, perché non fanno parte delle vere e proprie immagini parziali!
Anche nei nostri parallizable patterns operiamo in questo modo, settando in modo opportuno anche le dimensioni dei bordi a seconda delle esigenze di calcolo. Ogni pixel dell'immagine locale (dotata di bordo!) é del tipo *one-to-M*, poi sono richieste queste comunicazione per l'aggiornamento (in realtá é una vera e propria sovrapposizione!) dei dati ed infine un gather é necessario per ricostruire l'immagine. Il processo semplificato di quanto succede é in figura 6.

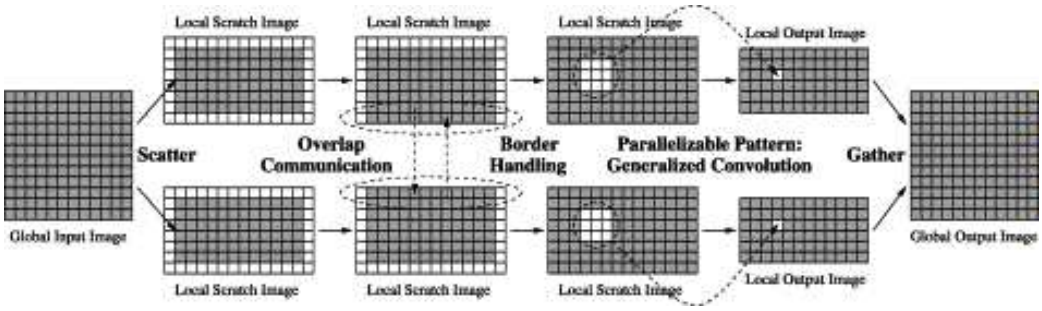

Figura 6: *Schema di convoluzione generalizzata*

5.5 Lo speed-up in una applicazione complessa

Per completezza ci sembra carino calcolare lo speed-up in un'applicazione più complessa degli esempi mostrati fino ad ora! Senza entrare nel merito del problema, e senza spiegare il metodo risolutivo adottato, riportiamo l'esempio di applicazione di image processing realizzata utilizzando la libreria. Il problema da risolvere é il rilevamento di linee (curve) in un'immagine! Ripeto non siamo interessati al metodo risolutivo ma ai tempi di esecuzione. L'immagine che si elabora è bidimensionale e rappresentata per livelli di grigio, l'ordine di grandezza é circa 1 MB. Lanciare in sequenziale quest'ap-

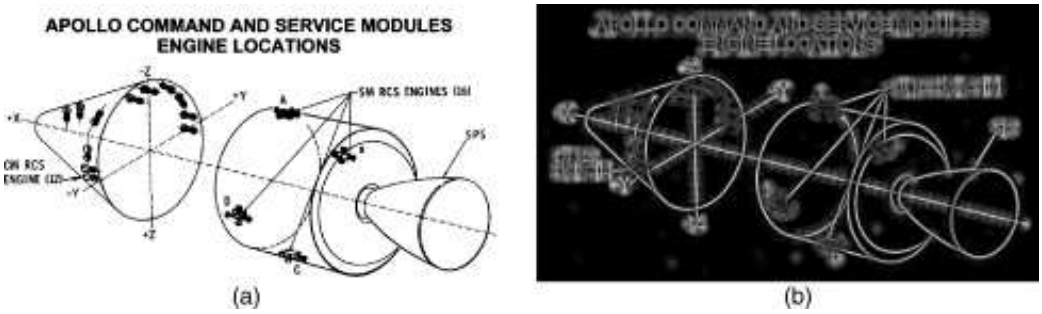

Figura 7: *Immagine in input ed immagine risultato*

plicazione richiede circa 30 minuti di computazione, i tempi risultano molto migliori passando al calcolo parallelo. Riportiamo un pò di conti:

Due processi

$$S = \frac{T_1}{T_2} = \frac{1758Sec}{943Sec} = 1,88$$

Tre processi

$$S = \frac{T_1}{T_3} = \frac{1758Sec}{710Sec} = 2,47$$

Quattro processi

$$S = \frac{T_1}{T_4} = \frac{1758Sec}{514Sec} = 3,42$$

6 Un esempio di ottimizzazione

Come già detto inizialmente, alla fase di calcolo della libreria si aggiunge una costruzione per una politica di ottimizzazione é sicuramente tedioso stare qui a spiegare la APIPM, peró trovo carino farvi vedere un esempio di ottimizzazione nel pratico, quindi vi presento la tecnica della **Lazy Parallelization**. Volendo rimanere coerente con il principio di trasparenza del parallelismo, la libreria applica una parallelizzazione di default; questa strategia é molto utile in quanto da questa si ereditano caratteristiche importanti quali la stabilità, l'efficienza e la correttezza dei risultati; ma spesso in una parallelizzazione di default esistono comunicazioni ridondanti e non si utilizzano strategie furbe, quali ad esempio il combinare messaggi multipli in una singola spedizione. La Lazy Parallelization apporta dei visibili miglioramenti, ottimizzando le chiamate alla libreria e riuscendo ad eliminare inconvenienti quali quelli citati prima. Tale costruzione si basa sulla definizione della **macchina a stati finiti, fsm**. Uno degli elemento essenziali della fsm é rappresentato da un *set di stati* che corrispondono ad una rappresentazione delle strutture dei dati distribuiti che risulti "valida", mentre l'altro elemento é la definizione delle *funzioni di transizione*, che specificano come da una rappresentazione "valida" di una struttura dati si passi ad un'altra rappresentazione "valida". Assumendo che ogni comunicazione o operazione di gestione della memoria inserita dalla parallelizzazione di default é ridondante fin quando non sia accertato il contrario, la lazy parallelization permette tali operazioni solo nel caso in cui una loro eliminazione porti ad una rappresentazione della struttura dati che sia giudicata invalida. In questo modo la lazy parallelization produce un codice parallelo che risulta corretto ma soprattutto ottimizzato. Analizziamo il comportamente di tale strategia con un semplice esempio e

notiamo la differenza tra il numero di comunicazioni e di operazioni di gestione della memoria operate delle due parallelizzazioni. Supponiamo di essere interessati all'immagine **c**, ottenuta da un'immagine **a** composta con un immagine **b**, a sua volta derivata da **a**. Le operazioni da applicare sono:

1. importare l'immagine **a**,
2. applicare all'immagine **a** l'operazione unaria per ottenere **b**,
3. applicare ad entrambe l'operazione binaria per ottenere **c**,
4. esportare l'immagine **c**,
5. cancellare tutte le immagini.

Le tabelle 1 e 2 che spiegano i comportamenti in entrambe le soluzioni:

Input(**a**); UnPixOp(**a**,**b**); BinPixOp(**a**,**b**,**c**); Export(**c**); Delete(**a**); Delete(**b**); Delete(**a**);	Input(**a**); Scatter(**a**,**loca**); UnPixOp(**loca**,**locb**); Gather(**locb**, **b**); Delete(**loca**); Delete(**locb**); Scatter(**a**, **loca**); Scatter(**b**, **locb**); BinPixOp(**loca**,**locb**,**locc**); Gather(**locc**, **c**); Delete(**loca**); Delete(**locb**); Delete(**locc**); Export(**c**); Delete(**a**); Delete(**b**); Delete(**c**);

Tabella 1: *Comportamento sequenziale e parallelo di default*

Input(**a**); UnPixOp(**a**,**b**); BinPixOp(**a**,**b**,**c**); Export(**c**); Delete(**a**); Delete(**b**); Delete(**c**);	Input(**a**); Scatter(**a**,**loca**); UnPixOp(**loca**,**locb**); BinPixOp(**loca**,**locb**,**locc**); Gather(**locc**, **c**); Delete(**loca**); Delete(**locb**); Delete(**locc**); Export(**c**); Delete(**a**); Delete(**b**); Delete(**c**);

Tabella 2: *Comportamento sequenziale e lazy parallelization*

7 Qualche riferimento

Ovviamente è possibile trovare in rete parecchie informazioni su quanto abbiamo detto. Ad ogni modo riportiamo qualche testo o indirizzo a cui poter far riferimento per poter approfondire gli argomenti esposti!
Per la libreria ricordiamo il sito da cui é possibile scaricare una versione free anche se parziale del lavoro.
http://carol.wins.uva.nl/~fjseins/ParHorusCode/
Da qui si raggiungono facilmente anche materiale descrittivo circa la libreria!
Per il calcolo parallelo riporto un manuale un pò vecchiotto, ma considerato un pilastro nel settore:
G. Fox, M. Johnson, et al., *Solving problems on concurrent processors*, Prentice - Hall International Edition, 1988
Per fornirvi un testo di image processing vi ricordo quello utilizzato all'interno del progetto ParHorus:
G.X. Ritter e J.N. Wilson, *Computer Vision Algorithms in Image Algebra*, CRS Press, Inc, 2000

A Qualche esempio di codice

A.1 Un pó di typedef per le strutture dati

Il primo passo é quello di definire il tipo di dato PIXEL, la precisione nella rappresentazione e poi la sua dimensione; questo perché sono previste differenti modalitá di rappresentazione di un'immagine!Come giá detto si puó utilizzare una rappresentazione per livelli di grigio, oppure l'RGB etc, per cui:

```
**********************

#define __PIX_TYPE_REAL
#define __PIX_PREC_32
#define __PIX_DIM_1

**********************

#if defined(__PIX_DIM_1)
#define PIX_DIM 1
#elif defined(__PIX_DIM_2)
#define PIX_DIM 2
#elif defined(__PIX_DIM_3)
#define PIX_DIM 3
#endif

**********************

#if defined(__PIX_TYPE_REAL) && defined(__PIX_PREC_32)
#define PIXVALTYPE float
#define M_PIXVALTYPE MPI_FLOAT
#define ARITHVALTYPE double
#define M_ARITHVALTYPE MPI_DOUBLE
#define PRPIXVAL %5.6f
#define PRARITHVAL %.2f,
#endif

**********************
```

Notiamo che PIXVALTYPE = float e di ARITHVALTYPE = double, ma queste sono conseguenze delle scelte fatte! Dopo questa serie di definizioni

possiamo passare alle strutture dati. Quelle che per ora ci interessano sono: PIXEL, IMAGE! In realtá aggiungiamo anche la definizione di ARITH, che ci servirá tra poco per dare un esempio di operazione binaria!

```
*********************************

typedef struct {
ARITHVALTYPE x;
#if !defined(__PIX_DIM_1)
ARITHVALTYPE y;
#endif
#if defined(__PIX_DIM_3)
ARITHVALTYPE z;
#endif
} ARITH;

*********************************

typedef struct {
PIXVALTYPE x;
#if !defined(__PIX_DIM_1) PIXVALTYPE y;
#endif #if defined(__PIX_DIM_3)
PIXVALTYPE z;
#endif } PIXEL;

*********************************

typedef struct img {
PIXEL *data;
int w;
int h;
int d;
short state;
short parAllowed;
struct img *partdata;
} IMAGE;

*********************************
```

Quindi ricapitolando per definire un'immagine abbiamo bisogno della definizione dei PIXEL, e di tutte le definizioni a questo associato, quindi tipo

di dato, tipo di rappresentazione ed il numero di variabili di cui si compone! Quando abbiamo definito tutto questo passiamo a definire la struttura PIXEL, che ci serve perché abbiamo fino a tre dimensioni nella rappresentazione dei PIXEL. Il passo successivo é l'immagine!
Altre strutture dati a cui siamo legati sono KERNEL e MATRIX. Queste strutture servono per computare alcuni algoritmic pattern, rispettivamente le operazioni di convoluzione e le operazioni geometriche. Ne riportiamo la definizione per completezza!

```
*********************************

   typedef struct { PIXEL *data;
int w;
int h;
int d;
double weight;
} KERNEL;

*********************************

typedef double HxMatrix;

*********************************
```

che poi viene usato come un puntatore a puntatore, per cui l'idea che ognuno di noi ha di matrice!

A.2 Finalmente un programma

Ed eccoci finalmente giunti al codice per il calcolo di un valore assoluto che avevamo proposto in §5.2 come esempio di algoritmic pattern!

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
#include "info.h"
#include "const.h"
#include "image.h"
#include "hximage.h"
#include "print.h"
```

```
#define IMAGE_IN ‘‘Images/celegori’’
#define IMAGE_OUTUV ‘‘Images/prova’’

int main(int argc, char *argv[])
{
int root = 0, xCPUs = 1, yCPUs = 1;
int operType = 0, lazyPar = 1;
int status;
double t0, t1;
IMAGE *origIm = (IMAGE *)NULL, *resltIm = (IMAGE *)NULL;

   /*** Initialize parallel system ***/

   initSystem(&argc, &argv);

   /*** Initialize data IO and scheduler ***/

   if (!initDataInOut(root)) {
abortSystem();
}

   if (!initScheduler(root, xCPUs, yCPUs, 1, lazyPar)) {
abortSystem();
}

   origIm = readImageFromFile(IMAGE_IN, &status);

   unaryPixOp(resltIm, HxIDabs);

   writeImageToFile(IMAGE_OUTUV, origIm);

   deleteImage(origIm);

   /*** Finish ***/

   return exitSystem();
}
```

A.3 Esempi di parallelizable pattern

Diamo un esempio di come siano implementate le vere e prorie operazioni all'interno della libreria! Riportiamo solo un esempio di Unary pixel operation, binary pixel operation e reduce operation, che hanno un codice piú semplice e compatto!

Unary pixel operation

```
void HxIDabs(IMAGE *imPtr)
{
int nPix = imPtr->w * imPtr->h * imPtr->d;
PIXEL *p = imPtr->data;

#if defined(__PIX_TYPE_INT)

while (-nPix >= 0) {
p->x = abs(p->x);
#if !defined(__PIX_DIM_1)
p->y = abs(p->y);
#endif
#if defined(__PIX_DIM_3)
p->z = abs(p->z);
#endif
p++;
}

#else /* if defined(__PIX_TYPE_REAL) */

while (-nPix >= 0)
p->x = fabs(p->x);
#if !defined(__PIX_DIM_1)
p->y = fabs(p->y);
#endif
#if defined(__PIX_DIM_3)
p->z = fabs(p->z);
#endif
p++;
}

#endif
```

```
}
```

Binary pixel operation

```
void HxIDadd_c(IMAGE *imPtr, ARITH val)
{
int nPix = imPtr->w * imPtr->h * imPtr->d;
PIXEL *p = imPtr->data;

   while (-nPix >= 0) {
p->x = (p->x + val.x);
#if !defined(__PIX_DIM_1)
p->y = (p->y + val.y);
#endif
#if defined(__PIX_DIM_3)
p->z = (p->z + val.z);
#endif
p++;
}
}
```

Reduce operation

```
ARITH HxIDmax(IMAGE *imPtr)
{ int nPix = imPtr->w * imPtr->h * imPtr->d;
PIXEL *p = imPtr->data;
ARITH result;

result.x = (ARITHVALTYPE)(p->x);
#if !defined(__PIX_DIM_1)
result.y = (ARITHVALTYPE)(p->y);
#endif
#if defined(__PIX_DIM_3)
result.z = (ARITHVALTYPE)(p->z);
#endif
p++;
nPix-;
while (-nPix >= 0) {
result.x = (result.x > (ARITHVALTYPE)(p->x) ?
```

```
result.x :  (ARITHVALTYPE)(p->x));
#if !defined(__PIX_DIM_1)
result.y = (result.y > (ARITHVALTYPE)(p->y) ?
result.y :  (ARITHVALTYPE)(p->y));
#endif
#if defined(__PIX_DIM_3)
result.z = (result.z > (ARITHVALTYPE)(p->z) ?
result.z :  (ARITHVALTYPE)(p->z));
#endif
p++;
}
return result;
}
```