Histogram Thresholding - Implementazione per architetture multicore tramite il framework Skandium

Nicola Corti - 454413
Corso di Laurea Magistrale in Informatica - Università di pisa

20 Settembre 2013

Sommario

Questa relazione ha lo scopo di illustrare i dettagli relativi alla progettazione, alla realizzazione e all'uso di Histogram: un software per il calcolo dell'Histogram Thresholding di un'immagine implementato per piattaforme multicore grazie all'utilizzo del framework Java Skandium.

Indice

Introduzione

La realizzazione del software ha avuto come fare iniziale quella di design (sezione 1) in cui sono state valutate le varie alternative per la soluzione del problema e sono state prese le scelte che hanno portato alla fase di implementazione (sezione 2): il software è stato realizzato in Java utilizzando il framework Skandium per supportare l'esecuzione su piattaforme parallele.

Successivamente si è provveduto a valutare le performance (sezione 3) del software su piattaforme con configurazioni differenti al fine di valutare la bontà del modello descritto nella fase di design.

La fase di sviluppo del software è terminata con la realizzazione di un archivio contenente i sorgenti, la documentazione annessa e la guida utente (sezione 4) rivolta ad ogni utente che volesse utilizzare il software.

1 Design

La fase di design ha avuto come punto di partenza l'individuazione di un algoritmo sequenziale che effettuasse il calcolo dell'histogram thresholding su una singola matrice, in modo da poterlo utilizzare come punto di partenza per definire il modello parallelo.

1.1 Gli algoritmi sequenziali

Sono stati individuati due differenti algoritmi sequenziali: il primo, più inefficiente, ha un costo quadratico rispetto al numero degli elementi di una matrice, il secondo invece è più efficiente ed ha un costo lineare nel numero degli elementi della matrice.

I due algoritmi verranno presentati in dettaglio nei paragrafi seguenti:

1.1.1 Algoritmo Quadratico

L'algoritmo quadratico ricalca la definizione del problema: per ogni elemento della matrice considerata si vanno a contare quanti sono gli elementi della stessa matrice che sono maggiori (minori). Si va a settare a 0 o ad 1 il corrispettivo bit della matrice risultante in base al fatto che il conteggio sia superiore o inferiore alla percentuale di soglia.

L'algoritmo è presentato sinteticamente nel seguente pseudo codice:

Codice 1: Pseudo codice dell'algoritmo quadratico

```
soglia = threshold * (mat.righe * mat.colonne)

for(i = 0; i < mat.righe; i++){
    for(j = 0; j < mat.colonne; j++){

        // Considero un generico elemento
        conta = 0;

        // Conteggio i valori maggiori
        for(k = 0; k < mat.righe; k++){
            for(l = 0; l < mat.colonne; l++){
                if (mat[i][j] > mat[k][l]) conta++;
            }
        }

        // Aggiorno di conseguenza la matrice risultante
        if (conta > soglia)
            res[i][j] = 0;
        else
            res[i][j] = 1;
    }
}
```

Osservazioni Si noti come questo algoritmo abbia bisogno di utilizzare una matrice risultate e non sia possibile aggiornare direttamente i valori della matrice della partenza; ciò porterebbe infatti ad un'inconsistenza e dunque all'impossibilità di procedere al conteggio sequenziale di ogni singolo elemento.

Il seguente algoritmo ha un costo pari ad $O((n \times m)^2)$ per una matrice di dimensione $n \times m$

1.1.2 Algoritmo Lineare

L'algoritmo lineare procede invece in tre fasi:

1. Viene inizialmente costruito un istogramma dell'immagine: si utilizza un array dove si calcolano le frequenze di ogni singolo valore della matrice (vedi immagine 1),
2. Partendo dall'istogramma viene calcolato il valore numerico della soglia,
3. Nella seconda fase si scandisce linearmente la matrice di partenza e si impostano i bit a 0 o ad 1 in base al fatto che il singolo elemento sia superiore o minore rispetto al valore di soglia precedentemente calcolato.

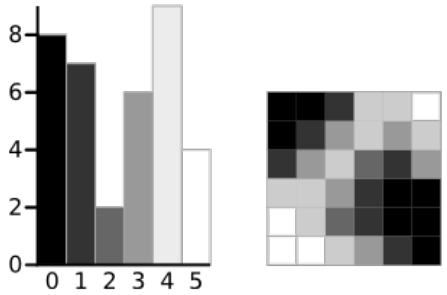

Figura 1: Esempio del calcolo dell'istogramma su una matrice 6×6 con valori da 0 a 5 (rappresentati con gradazioni di grigio)

L'algoritmo è presentato sinteticamente nel seguente pseudo codice:

Codice 2: Pseudo codice dell'algoritmo lineare

```
// Calcolo dell'istogramma
for(i = 0; i < mat.righe; i++){
    for(j = 0; j < mat.colonne; j++){
        isto[mat[i][j]]++;
    }
}

// Calcolo del valore di soglia
soglia = threshold * (mat.righe * mat.colonne)
risultato = 0; conta = 0;

while (risultato < isto.max AND conta < soglia){
    conta = conta + isto[risultato];
    risultato++;
}

// Aggiornamento della matrice
for(i = 0; i < mat.righe; i++){
    for(j = 0; j < mat.colonne; j++){
        if (mat[i][j] < risultato)
            mat[i][j] = 0;
        else
            mat[i][j] = 1;
    }
}
```

Osservazioni Si noti come questo algoritmo, a differenza del precedente, effettua semplicemente due visite lineari di tutta la matrice; inoltre non necessita di una matrice di appoggio, ma può aggiornare direttamente la matrice di partenza, una volta calcolato il valore numerico di soglia.

Il seguente algoritmo ha un costo invece pari ad $O(n \times m)$ per una matrice di dimensione $n \times m$

1.2 I modelli paralleli

Partendo dai due algoritmi sequenziali sono stati proposti differenti modelli che verranno presentati nelle sezioni seguenti.

1.2.1 Modello per l'algoritmo quadratico

L'algoritmo sequenziale quadratico può essere parallelizzato tramite l'utilizzo di una Map: si divide la matrice di partenza per righe (colonne) ed ogni porzione di matrice viene assegnata ad un worker differente (immagine 2).

Ogni worker provvederà a calcolare la propria porzione di matrice, utilizzando la matrice di partenza per effettuare i conteggi.

Il calcolo termina quando tutti gli worker hanno terminato di calcolare la propria porzione.

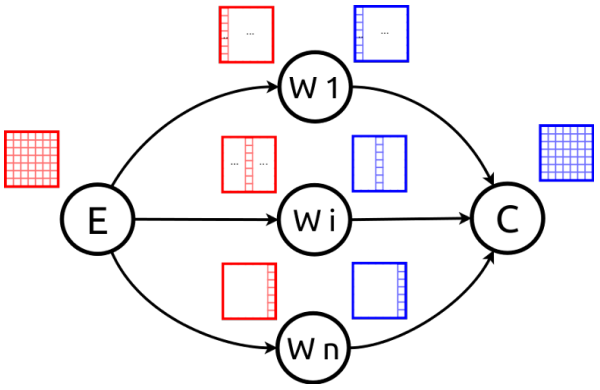

Figura 2: Rappresentazione grafica del modello Map, le matrici rosse rappresentano le matrici con i valori di partenza, le matrici blu rappresentano matrici di 0 o 1.

1.2.2 Modello per l'algoritmo lineare

Per l'algoritmo lineare sono stati individuati due modelli differenti:

Modello Pipeline Il primo modello individuato consiste nel realizzare un Pipeline a due stadi (immagine 3):

- Il primo stadio è una Map-Reduce che si occupa di calcolare l'istogramma in modo parallelo nella fase di Map, di ricomporlo nella fase di Reduce e successivamente di calcolare il valore di soglia,
- Il secondo stadio è una Map che si occupa di aggiornare in modo parallelo la matrice in base al valore di soglia calcolato nello stadio precedente.

L'utilizzo di un pipeline permette di gestire in modo ottimale uno stream di matrici in entrata.

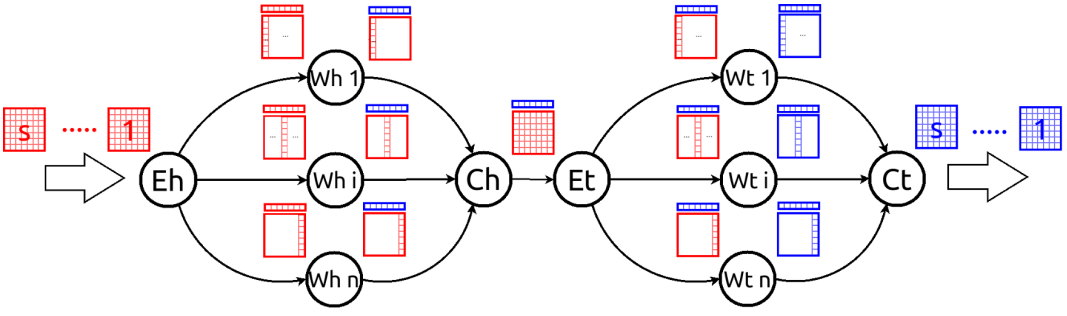

Figura 3: Rappresentazione grafica del modello Pipeline, le matrici rosse rappresentano le matrici con i valori di partenza, le matrici blu rappresentano matrici di 0 o 1. I piccoli vettori rappresentano gli istogrammi delle matrici, rossi se non sono stati calcolati, blu altrimenti

Modello Farm Il secondo modello considera una grana del calcolo più grossolana, realizzando un Farm: ogni worker si occupa di effettuare tutte le 3 fasi dell'algoritmo lineare, ognuno su una matrice distinta (immagine 4).

Anche con il modello Farm si riesce a gestire in modo ottimale uno stream di matrici in entrata.

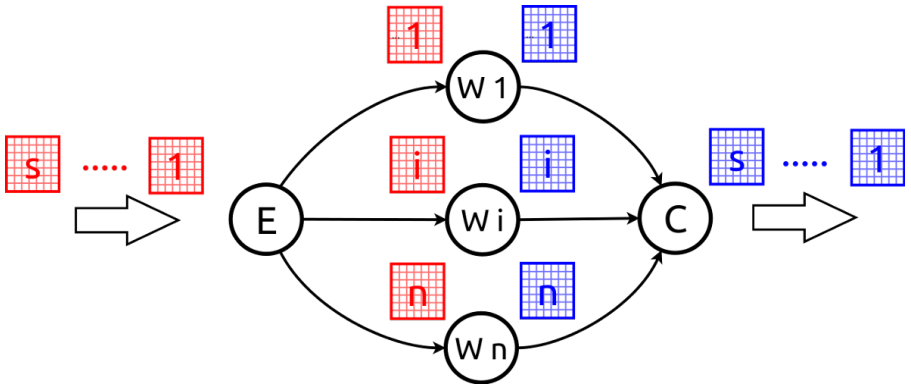

Figura 4: Rappresentazione grafica del modello Farm, le matrici rosse rappresentano le matrici con i valori di partenza, le matrici blu rappresentano matrici di 0 o 1.

2 Implementation

Il software è stato realizzato in Java, utilizzando il framework parallelo Skandium[1], che offre un ambiente per la realizzazione di applicazioni parallele utilizzando gli skeleton più comuni.

Nei paragrafi seguenti sono descritti tutti i dettagli implementativi e le scelte prese al fine di rendere più facile la comprensione del codice.

2.1 L'oggetto Matrix

La comprensione dell'oggetto Matrix risulta fondamentale per la comprensione del software nella sua completezza, esso infatti incorpora tutto il codice funzionale necessario ad effettuare il calcolo degli istogrammi.

Matrix contiene: il membro mat che consiste in una matrice di int dove vengono memorizzati i numeri della matrice, i membri sizeRow e sizeCol per memorizzare la dimensione della matrice ed i membri beginRow, beginCol, endRow ed endCol che permetto di memorizzare righe e colonne di inizio e di fine, in modo da poter considerare il calcolo solamente di una sottomatrice[2].

Compito delle istanze di Matrix è anche quello di memorizzare l'istogramma della matrice (o della sottomatrice) calcolato. Per la memorizzazione dell'istogramma si è deciso di usare una array di int (il membro histo), assumendo di conoscere a priori il valore massimo che può essere assunto dagli elementi della matrice.

Si noti che tale assunzione non è eccessivamente restrittiva, infatti se consideriamo le matrici come immagini RGB il valore massimo che può essere assunto da un pixel è 16581375 (255^4 ovvero 255 possibili valori per ogni singolo canale) mentre se consideriamo le matrici come immagini in gradazioni di grigio il valore massimo che possono assumere i pixel risulta essere invece 255. Il valore massimo risulta quindi settato nel membro statico PIXEL_MAX_VALUE.

Se non si facesse questa assunzione, si dovrebbe procedere con una struttura differente (ad esempio con una lista) che comporterebbe maggiori overhead, oppure si dovrebbe procedere ad una scansione preventiva della matrice al fine di determinare l'elemento massimo.

La classe Matrix, oltre ai costruttori classici, offre due costruttori per la generazione di matrici contenenti numeri casuali, in modo da facilitare la fase di test e di valutazione delle performance.

La classe Matrix contiene inoltre i seguenti metodi:

compute Metodo che implementa l'algoritmo quadratico, ritornando un nuovo oggetto Matrix aggiornato con i valori a 0 o ad 1,

computeHistogram Metodo che calcola l'istogramma della matrice, aggiornando l'array histo con i valori delle occorrenze,

computeThreshold Metodo che calcola il valore numerico di soglia, partendo da un istogramma calcolato (deve essere invocato dopo compute Histogram),

computeUpdateMatrix Metodo che aggiorna la matrice con 0 ed 1 in base al valore numerico di soglia calcolato da computeThreshold.

subMatrix Metodo che ritorna una nuova sottomatrice a partire da un oggetto Matrix.

print Metodo per la stampa della matrice.

[1] https://github.com/mleyton/Skandium

[2] Ciò risulta necessario ad esempio nel caso della Map

2.2 Classi Sequential e SequentialQuad

Le classi Sequential e SequentialQuad realizzano rispettivamente l'algoritmo lineare e l'algoritmo quadratico senza l'utilizzo del framework parallelo.

Queste classi sono state realizzate al fine di poter valutare il miglioramento delle performance fra l'implementazione puramente sequenziale e quella parallela.

Le classi hanno due metodi:

compute Per effettuare il calcolo di una singola Matrix,

testcompute Per simulare il calcolo di uno stream di generica lunghezza stream, di matrici casuali di sizeRow $\times$ sizeCol.

2.3 Classe ParallelQuad

La classe ParallelQuad implementa il modello parallelo Map utilizzando l'algoritmo quadratico: utilizza lo skeleton Map<P, R> definito da skandium.

Lo skeleton Map<P, R> prevede che siano forniti tre oggetti per la sua generazione: Split<P,X> split, Execute<X,Y> execute, Merge<Y,R> merge

- Split<P, X>, per la scomposizione del task in sottotask da inviare agli worker
- Execute<X, Y>, per l'esecuzione del codice funzionale del problema oppure uno Skeleton<X, Y> per permettere la composizionalità.
- Merge<Y, R>, per effettuare la ricomposizione dei risultati calcolati dagli worker.

Sono stati quindi definiti gli oggetti:

- SplitMatrix<Matrix, Matrix>, per la scomposizione delle matrici in sottomatrici divise per righe. Gli oggetti SplitMatrix vengono istanziati con un parametro necessario per definire il numero di sottomatrici da generare,
- ComputeQuad<Matrix, Matrix>, per l'esecuzione del calcolo tramite l'algoritmo quadratico sulla sottomatrice,
- MergeMatrix<Matrix, Matrix>, per effettuare la ricomposizione delle sottomatrici calcolate in un unica matrice risultante.

2.4 Classe Parallel

La classe Parallel implementa il modello parallelo Pipeline utilizzando l'algoritmo lineare: utilizza lo skeleton Pipeline<P, R> definito da skandium.

Lo skeleton Pipeline<P, R> prevede che siano forniti due oggetti di tipo Execute<X, Y> oppure due oggetti di tipo Skeleton<X, Y>. In particolare gli sono stati forniti due oggetti di tipo Map<Matrix, Matrix> andando a realizzare i due stadi del Pipeline.

Per la prima Map sono stati definiti gli oggetti:

- SplitMatrix<Matrix, Matrix>, già usato per la classe ParallelQuad.
- ComputeHistogram<Matrix, Matrix>, per l'esecuzione del calcolo dell'istogramma sulla sottomatrice,
- MergeHistoMatrix<Matrix, Matrix>, per effettuare l'operazione di Reduce partendo dagli istogrammi calcolati per arrivare all'istogramma risultante.

Mentre per la seconda Map sono stati usati gli oggetti:

- SplitMatrix<Matrix, Matrix>, già usato per la classe ParallelQuad,
- ComputeThreshold<Matrix, Matrix>, per l'esecuzione dell'aggiornamento della matrice partendo dal valore numerico di soglia,
- MergeMatrix<Matrix, Matrix>, già usato per la classe ParallelQuad.

2.5 Classe ParallelFarm

La classe ParallelQuad implementa il modello parallelo Farm utilizzando l'algoritmo lineare: utilizza lo skeleton Farm<P, R> definito da skandium.

Lo skeleton Farm<P, R> prevede che sia fornito un oggetto di tipo Execute<P, R> oppure un oggetto di tipo Skeleton<P, R>. In particolare è stato definito un oggetto ComputeFarm<P, X> che si occupa di effettuare tutto il calcolo dell'algoritmo lineare su una singola matrice.

2.6 Classe TestClass

La classe TestClass contiene tutte le operazioni per avviare le computazioni di test in base ai parametri dati in input.

3 Performance

Al fine di valutare le performance del calcolo sono stati dapprima definiti dei modelli analitici e sono state effettuati successivamente delle rilevazioni su alcune macchine.

Le rilevazioni sono state realizzate inserendo dei *probes* all'interno del codice che rilevavano il tempo impiegato nel calcolo tramite chiamate alla funzione System.currentTimeMillis()

Per i test è stata utilizzata una macchina multicore della classe *Nehalem* con 8 core, ognuno con due contesti, capace quindi di consentire l'esecuzione parallela di 16 thread.

Tutti i test vengono realizzati utilizzando matrici generate casualmente dal software e utilizzando come valore di soglia 0.5 (50%).

I grafici delle misurazioni sono raccolti nell'appendice A.

3.1 Performance dell'algoritmo quadratico

Il tempo di servizio del modello può essere stimato tramite la formula

$$T_{serv}(n) = T_{emit} + \frac{T_{work}}{n} + T_{coll}$$

dove: T_{emit}, T_{work} e T_{coll} sono rispettivamente i tempi per dividere, calcolare e ricomporre le matrici.

Possiamo notare che nel nostro caso il termine dominante è senza dubbio T_{work} e che l'overhead introdotto da T_{emit} e T_{coll} è trascurabile e decresce all'aumentare della dimensione della matrice, in quanto il termine T_{work} comporta un costo quadratico.

Questa implementazione, come si può facilmente immagine, raggiunge delle performance nettamente inferiori per quanto riguarda il tempo di servizio e di completamento se paragonate alle implementazioni lineari.

Ma invece ne consideriamo la scalabilità e l'efficenza, possiamo notare dai grafici che i risultati sono ottimi e che questa implementazione si comporta molto bene anche con un numero di thread elevato (immagine 7).

3.2 Performance dell'algoritmo lineare

L'algoritmo lineare riesce a realizzare dei tempi di servizio molto migliori rispetto all'algoritmo quadratico; risulta quindi impossibile paragonare i due tempi di servizio perché i due algoritmi lavorano su due ordini di grandezza differenti: la singola matrice che viene calcolata in decine minuti dall'algoritmo quadratico viene invece calcolata in qualche millisecondo tramite l'algoritmo lineare.

Aumentando la dimensione delle matrici dobbiamo però considerare un overhead che prima avevamo trascurato: il tempo di generazione della matrice casuale. Se infatti il tempo impiegato per generare una matrice casuale 100×100 risulta irrisorio, il tempo impiegato per generarne una 10.000×10.000 (o superiori) non lo è più.

Al fine di valutare correttamente il tempo di servizio e di completamento del calcolo è stato quindi necessario effettuare dapprima la generazione di tutto lo stream, in moda da ridurre il tempo di interarrivo.

Trascurare questo overhead ed effettuare la misurazione prima di generare lo stream avrebbe portato a risultati viziati, in quanto il tempo di completamento risultava sostanzialmente costante al variare del numero di thread (in quanto il tempo impiegato per la generazione di stream lunghi domina il tempo impiegato per il calcolo).

3.2.1 Performance di Parallel

Per il modello Pipeline consideriamo come tempo di servizio il massimo fra i tempi di servizio dei due stadi del pipeline:

$$T_{serv}(n) = \max\{T_{hist}(n), T_{upd}(n)\}$$

dove $T_{hist}(n)$ e $T_{upd}(n)$ sono rispettivamente i tempi di servizio delle due Map, per cui possiamo riscrivere la formula secondo lo stesso modello analitico usato per l'algoritmo quadratico (sezione 3.1)

$$T_{serv}(n) = \max\left\{\left(T_{hemit} + \frac{T_{hwork}}{n} + T_{hcoll}\right), \left(T_{uemit} + \frac{T_{uwork}}{n} + T_{ucoll}\right)\right\}$$

Purtroppo i risultati in termini di scalabilità e di efficienza non sono ottimi come quelli registrati con l'algoritmo quadratico, ma si riesce comunque a velocizzare il calcolo in parallelo rispetto al sequenziale (immagine 5).

Evidentemente gli overhead introdotti dal framework sono troppo elevati, oppure la grana utilizzata per il calcolo e' troppo fine.

3.2.2 Performance di ParallelFarm

Per il modello Farm consideriamo come tempo di servizio:

$$T_{serv}(n) = \max\left\{T_{emit}, \frac{T_{work}}{n}, T_{coll}\right\} = \frac{T_{work}}{n}$$

Vediamo che aumentare la grana del calcolo comporta un leggero miglioramento delle performance rispetto al modello Pipeline.

Purtroppo il modello Farm non comporta un miglioramento delle performance se si considera una sola matrice, dove invece il modello Pipeline permette di velocizzare il calcolo, anzi ottiene dei risultati peggiori persino all'esecuzione sequenziale.

Per stream di matrici che siano invece di dimensioni superiori si può notare che i risultati sono nettamente migliori (immagine 6).

4 User Guide

Per installare il software è necessario posizionarsi all'interno della directory dove è contenuto il software ed invocare da terminale il comando

```
ant
```

Nel caso si fosse interessati a conoscere i dettagli del processo di installazione, procedere con la lettura, altrimenti procedere dalla sezione relativa all'utilizzo del software.

4.1 Installation

Il software è corredato di un file **ant** (il file `build.xml`) che offre dei target per automatizzare il processo di compilazione e di configurazione del software.

In particolare è possibile inizializzare l'ambiente di lavoro utilizzando il comando

```
ant init
```

che provvede a creare le cartelle necessarie per completare il processo di compilazione.

Per compilare il progetto è necessario eseguire il target

```
ant build
```

che provvederà ad invocare il compilatore `javac` per compilare i sorgenti presenti all'interno della cartella `src/`, i file `.class` generati si troveranno all'interno della cartella `bin/`. Il target `build` provvede ad invocare automaticamente il target `init`, per cui non è necessario invocare direttamente il target `init` a meno che non si sia interessati a configurare l'ambiente senza effettuare la compilazione.

Per pulire la cartella `bin/` al fine di avere un ambiente pulito per poter effettuare una nuova compilazione è possibile utilizzare il target

```
ant clean
```

È infine possibile generare un file `jar` contenente tutti i file compilati e tutte le librerie necessarie all'esecuzione. Per farlo è sufficiente invocare il target

```
ant jar
```

Verrà generato un file chiamato `histogramthresholding.jar` all'interno della cartella principale del software. Per avviare il file `jar` è necessario invocare il comando

```
java -jar histogramthresholding.jar [parametri]
```

4.2 Documentation

Al fine di rendere il codice sorgente più comprensibile, il software è stato corredato di documentazione. In particolare tutte le parti del codice sorgente che potrebbero risultare di difficile comprensione sono state commentate in lingua inglese. Inoltre ogni funzione e classe del software è stata documentata con il formato javadoc, la documentazione generata può essere visionata all'interno della cartella doc/ e può essere rigenerata utilizzando il comando

```
ant javadoc
```

Per una comprensione organica del software si consiglia la lettura della seguente relazione nella sua interezza. La presente relazione viene rilasciata in Pdf ed in LaTeX e può essere visionata all'interno della cartella `doc/tex/`.

Per l'utente finale è anche disponibile una pagina di manuale contenente tutti le informazioni sui parametri di invocazione del software visionabile tramite il comando

```
./man.sh
```

4.3 Software Usage

Una volta compilato il software è possibile invocare il software utilizzando una delle differenti modalità. Il software permette inoltre di essere invocato indicando alcuni parametri al fine di poter valutare come varia il calcolo al variare dei dati.

Il software può essere invocato tramite il comando

```
ant <mode> [-Dthread=numt] [-Dstream=numstr] [-Drow=nrow] [-Dcol=ncol]
  [-Dthre=threshold] [-Dheap=heapsize]
```

I parametri sono i seguenti:

-Dthread= Permette di impostare il numero di thread con cui invocare il software. Questo numero rappresenta di fatto il grado di parallelismo con cui viene fatto eseguire il software. Tale parametro non viene preso in considerazione nel caso di esecuzioni sequenziali. Il valore di default di questo parametro è 4 per le computazioni parallele e 1 per le computazioni sequenziali.

-Dstream= Permette di impostare la dimensione dello stream di input ovvero quanti oggetti di tipo matrice devono essere generati e calcolati. Il valore di default per questo parametro è 1.

-Drow= Permette di impostare la dimensione per righe delle matrici date in input. Il valore di default per questo parametro è 5000.

-Dcol= Permette di impostare la dimensione per colonne delle matrici date in input. Il valore di default per questo parametro è 5000.

-Dthre= Permette di impostare la percentuale di soglia su cui effettuare il calcolo. Il valore deve essere un numero reale compreso fra 0 e 1 e può essere espresso con il punto (ad esempio 0.45 per indicare il 45%). Il valore di default per questo parametro è 0.50.

-Dheap= Permette di impostare il valore massimo della memoria utilizzabile dalla JVM, in modo da permettere il calcolo di matrici più grandi. Il valore di default di questo campo è `3072m` (3 Gb).

Le modalità sono le seguenti:

Sequential Il calcolo viene effettuato in sequenziale, utilizzando la classe Sequential (sezione 2.2) e quindi l'algoritmo lineare.

Parallel Il calcolo viene effettuato in parallelo, utilizzando la classe Parallel (sezione 2.4), quindi utilizzando il Pipeline a due stati costituito da due Map.

ParallelFarm Il calcolo viene effettuato in parallelo, utilizzando la classe ParallelFarm (sezione 2.5), avviando tanti worker quanti richiesti (tramite il parametro -Dthread), ed ogni worker esegue il calcolo di una singola matrice tramite l'algoritmo lineare.

SequentialQuad Il calcolo viene effettuato in sequenziale, utilizzando la classe SequentialQuad (sezione 2.2) quindi utilizzando l'algoritmo quadratico.

ATTENZIONE: l'algoritmo è particolarmente inefficiente e porta a computazioni molto lunghe anche su matrici di piccole dimensioni, si consiglia per cui di non impostare valori di -Drow e -Dcol troppo elevati.

ParallelQuad Il calcolo viene effettuato in parallelo, utilizzando la classe ParallelQuad (sezione 2.3) che realizza una Map ed applica l'algoritmo quadratico.

ATTENZIONE: l'algoritmo è particolarmente inefficiente e porta a computazioni molto lunghe anche su matrici di piccole dimensioni, si consiglia per cui di non impostare valori di -Drow e -Dcol troppo elevati.

4.4 Software Testing

Per effettuare il testing del software è possibile utilizzare gli script contenuti all'intero della cartella test, in particolare è possibile effettuare una serie di test utilizzando lo script bash

```
./test.sh
```

Per modificare i parametri del test è sufficiente aprire il file e cambiare le variabili presenti all'inizio del file (sono presenti dei commenti per aiutare l'utente nella scelta delle variabili).

Nel caso si volessero importare i dati in un foglio elettronico al fine di generare dei grafici è possibile utilizzare lo script python

```
./parser.py OUTPUT_FILE
```

che si occupa di analizzare l'output generato dal precedente script e ne realizza una stampa separando i valori tramite punti e virgola, in modo da renderne facile l'importazione in fogli elettronici come file di tipo csv.

5 Future Improvements

Il software permette alcuni miglioramenti che potrebbero essere effettuati in futuro al fine di migliorarne le performance:

- Modificare la classe SplitMatrix affinché realizzi la divisione delle matrici tramite righe o colonne, oppure a blocchi (l'oggetto Matrix non deve essere modificato per questo) e valutarne le performance in tutti e 3 i casi, al fine di valutare quale è il migliore.
- Effettuare test su matrici su macchine a performance maggiori, con più core, e più memoria, al fine di poter valutare come si comportano gli algoritmi lineare su matrici e su stream di dimensioni maggiori.
- Effettuare test su matrici che abbiano il valore massimo che può essere assunto dagli elementi della matrice (macro PIXEL_MAX_VALUE) non più a 256 ma a valori maggiori. In tali casi infatti l'istogramma assume maggiori dimensioni e la fase di Reduce, in cui si procede a calcolare l'istogramma risultante ha un costo maggiore che non può più essere trascurato.

6 Software Application

Tutti i test e le considerazioni sul software sono stati effettuati utilizzando matrici di interi generati a caso, ma possiamo utilizzare il software anche per effettuare il calcolo della soglia di immagini o di fotografie.

Nella tabella seguente è possibile vedere il calcolo della soglia di una fotografia 512×512 con differenti valori di soglia.

Originale	0.10	0.25	0.50	0.75	0.90

Ovviamente potremmo utilizzare i modelli Farm e Pipeline per effettuare computazioni di una serie di immagini (ad esempio l'insieme di foto contenute in una cartella).

7 Licence

Tutto il codice sorgente scritto viene rilasciato sotto licenza Gnu GPL - General Public Licence versione 3, ognuno è libero di modificare e di distribuire il codice sorgente entro i termini di tale licenza. Tale licenza può essere consultata all'indirizzo: http://www.gnu.org/copyleft/gpl.html

Per qualsiasi problema è possibile contattare lo sviluppatore all'indirizzo e-mail `cortin [at] cli.di.unipi.it`.

A Grafici delle performance

A.1 Grafici delle performance di **Parallel**

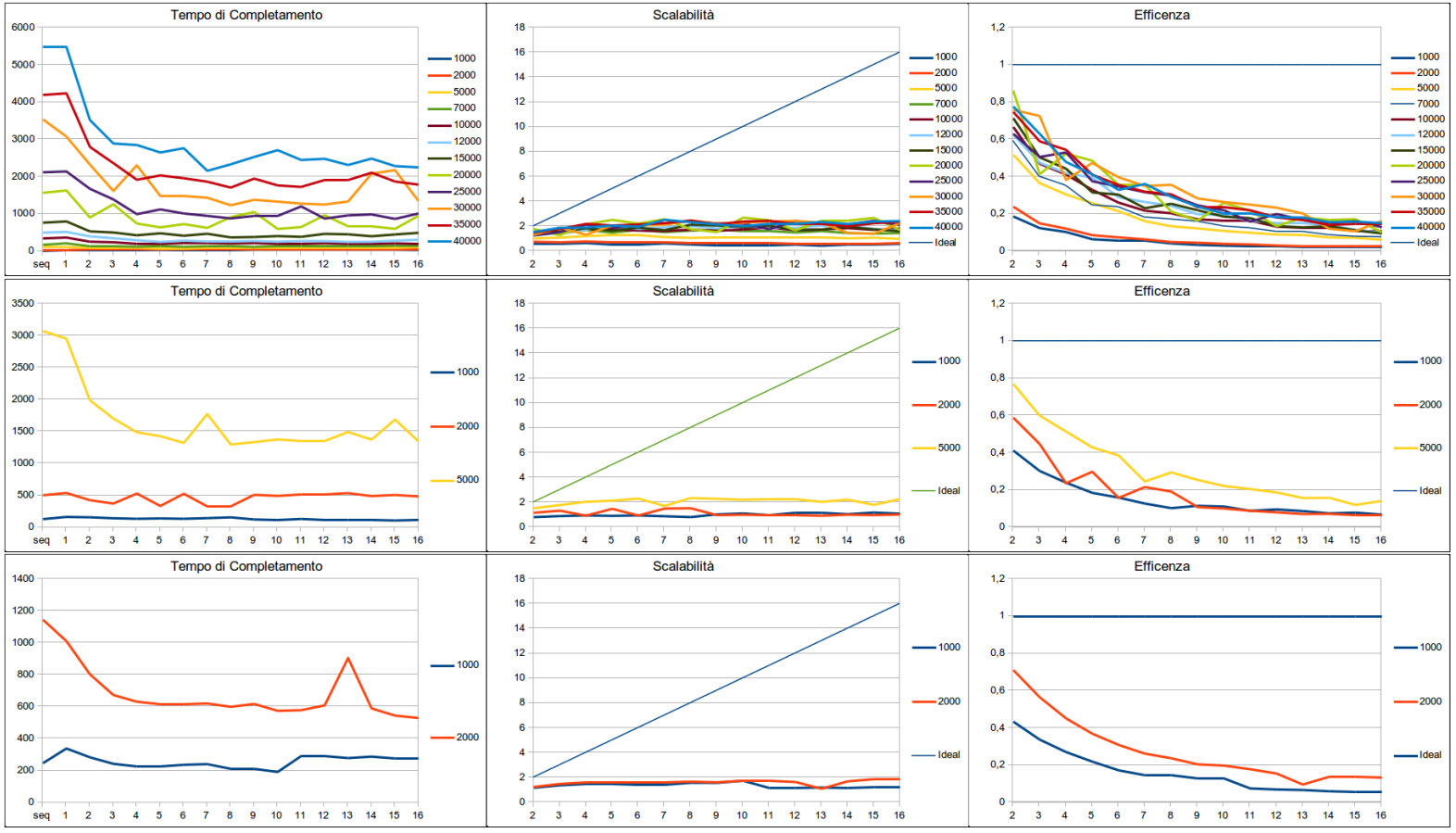

Figura 5: Grafici che mostrano tempo di completamento in millisecondi, scalabilità e efficienza del modello Parallel (Pipeline). Sulla prima riga si possono vedere test effettuati su stream di una singola matrice con dimensioni che variano da 1000×1000 a 40.000×40.000. Sulla seconda e sulla terza riga si possono notare test effettuati su stream di lunghezza 50 e 100; si noti come le dimensioni delle matrici sono ridotte, in quanto la macchina su cui sono stati effettuati test aveva memoria limitata

Dai grafici si evince che, malgrado il tempo di completamento sia stato ridotto tramite l'esecuzione parallela, l'applicazione scala poco, riuscendo a raggiungere un valore prossimo a 2. Anche l'efficenza risulta molto bassa, anche se notiamo che aumenta, all'aumentare della dimensione della matrice; sarebbe stato interessante provare ed eseguire test con stream di lunghezza 50 e 100 e matrici di dimensione 40.000×40.000, ma la memoria a disposizione non era sufficiente.

Eventuali oscillazioni dei grafici (le "punte" che appaiono nel grafico del tempo di completamento) possono indicare calcoli che sono stati effettuati in momenti in cui la macchina non era completamente scarica da altri task, ma aveva in esecuzione altri processi che potevano tenere impegnati i processori e rallentare l'esecuzione. Per attenuare questo fenomeno avremo dovuto eseguire più calcoli in momenti differenti e calcolarne la media.

A.2 Grafici delle performance di **ParallelFarm**

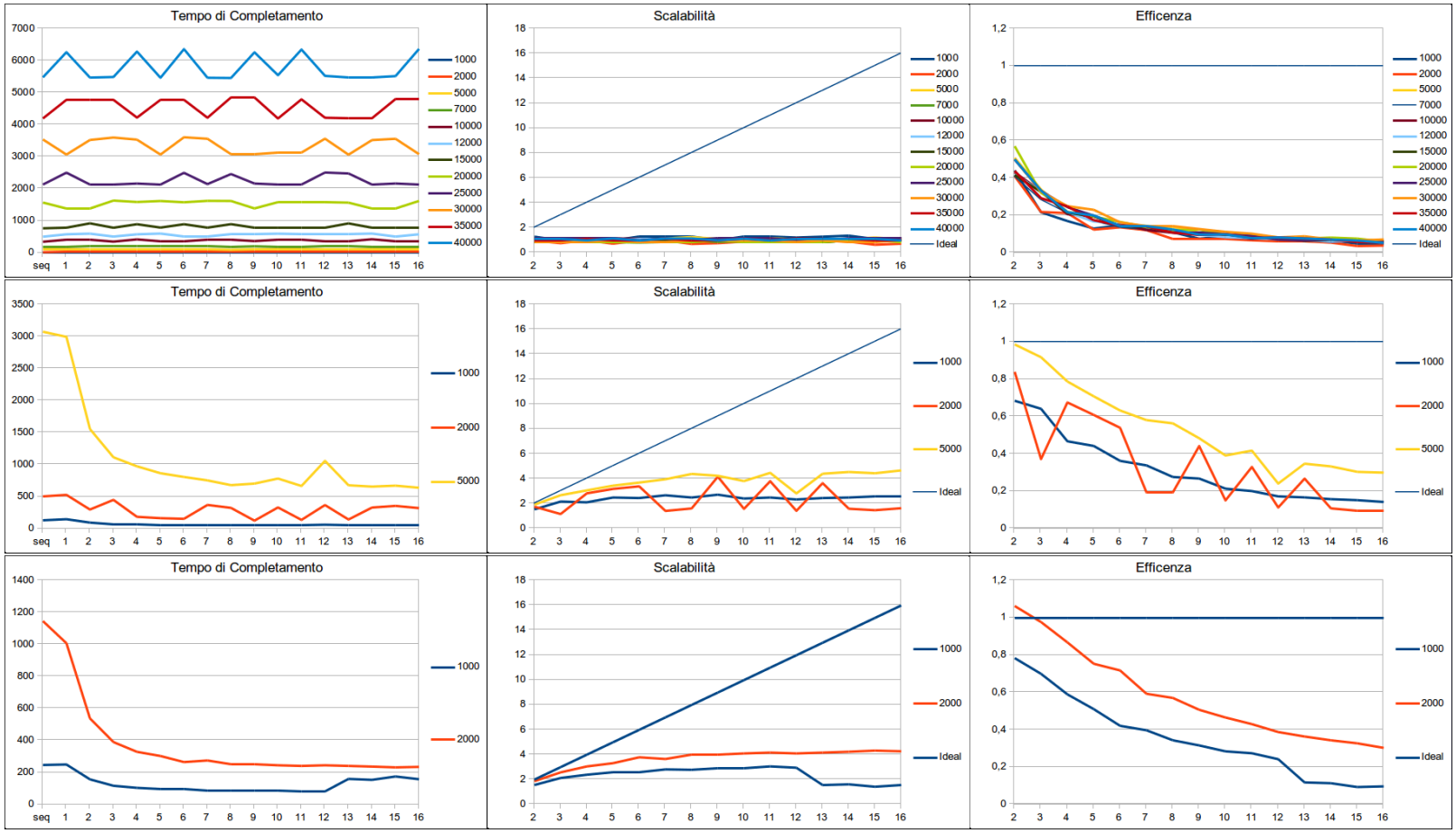

Figura 6: Grafici che mostrano tempo di completamento in millisecondi, scalabilità e efficienza del modello Farm. Sulla prima riga si possono vedere test effettuati su stream di una singola matrice con dimensioni che variano da 1000×1000 a 40.000×40.000. Sulla seconda e sulla terza riga si possono notare test effettuati su stream di lunghezza 50 e 100; si noti come le dimensioni delle matrici sono ridotte, in quanto la macchina su cui sono stati effettuati test aveva memoria limitata.

Dai grafici si evince che, come atteso, il modello Farm non è assolutamente adatto a calcoli con un singola matrice, il tempo di completamento risulta infatti costante per questi casi. Per quanto riguarda invece computazioni che coinvolgono stream di matrici si può notare che, malgrado il tempo di completamento sia stato ridotto tramite l'esecuzione parallela, l'applicazione scala poco, riuscendo a raggiungere un valore di scalabilità prossimo a 2. Anche l'efficienza risulta molto bassa, anche se notiamo che aumenta, all'aumentare della dimensione della matrice. Sarebbe stato interessante provare ed eseguire test con stream di lunghezza 50 e 100 e matrici di dimensione 40.000×40.000, ma la memoria a disposizione non era sufficiente.

Per stream di lunghezza maggiore invece notiamo che si raggiungono risultati migliori del modello Pipeline, ad esempio per il caso con stream di lunghezza 100 e con matrici 2.000×2.000 si nota che si raggiunge un scalabilità di 4 e che l'efficienza con grado di parallelismo 2 risulta essere superiore ad 1.

A.3 Grafici delle performance di **ParallelQuad**

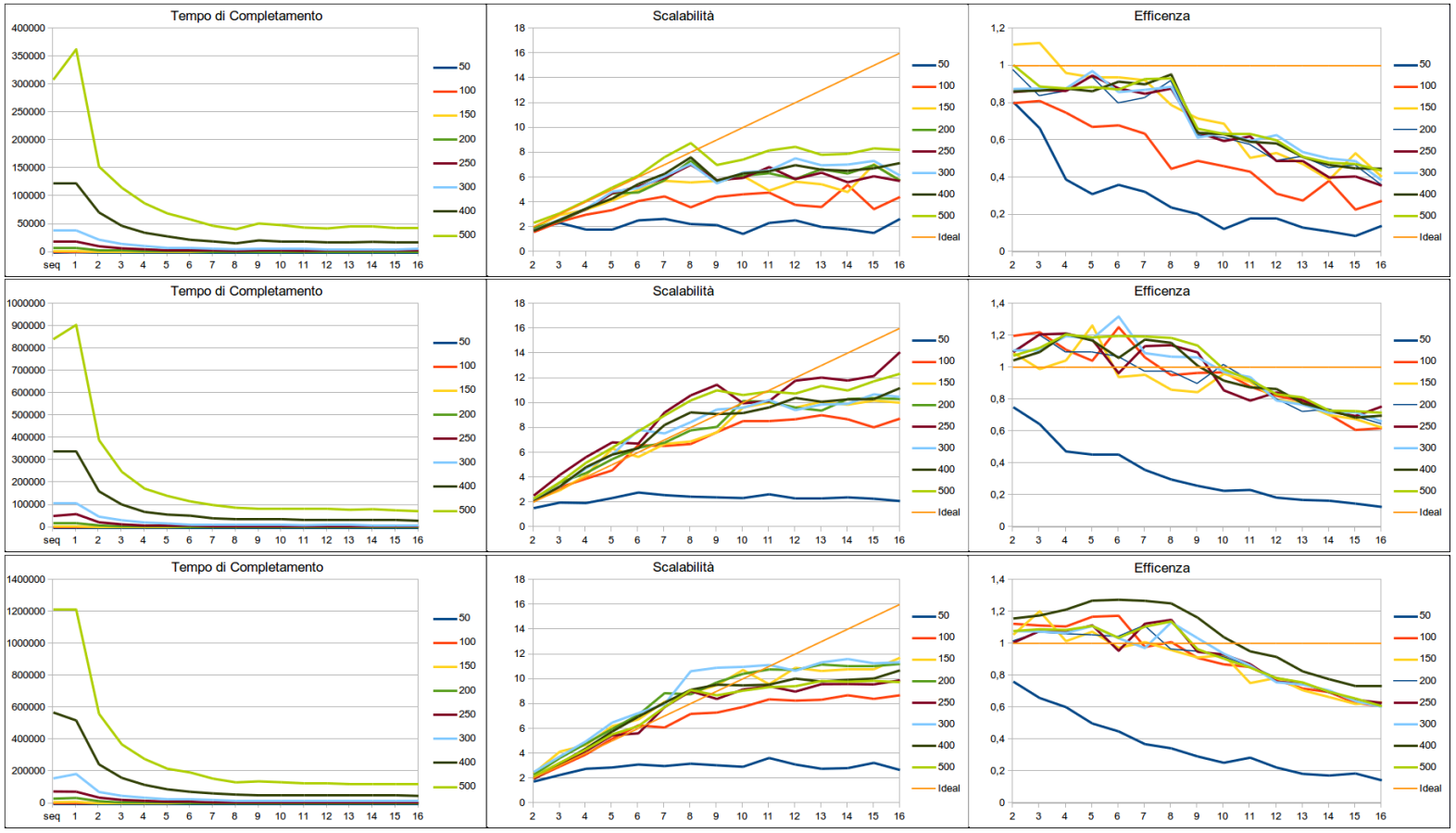

Figura 7: Grafici che mostrano tempo di completamento in millisecondi, scalabilità e efficienza del modello Map utilizzando l'algoritmo quadratico. Sulla prima riga si possono vedere test effettuati su stream di una singola matrice con dimensioni che variano da 50×50 a 500×500. Sulla seconda e sulla terza riga si possono notare test effettuati su stream di lunghezza 5 e 10; le dimensioni sono notevolmente ridotte perché il tempo impiegato per fare i test cresceva notevolmente.

Come possiamo notare in questi grafici, il tempo di completamento in valore assoluto è molto più alto di quello ottenuto con i modelli Pipeline e Farm (si pensi che per calcolare uno stream di 10 matrici 500×500 sono necessari circa 1.200.000 millisecondi, ovvero 20 minuti.

I risultati in termini di scalabilità e di efficienza sono invece più che ottimi, si riesce ad ottenere un risultato *super-lineare* infatti si nota che la scalabilità ottenuta è maggiore di quella indicata dalla linea ideale.

Ne consegue che si ottengono efficienze superiori ad 1, come dimostrato dai grafici dell'efficienza.

Si noti come l'efficienza rimanga pressoché stabile fino ad un grado di parallelismo pari ad 8 e poi tenda a decadere (un evento simile si può riscontrare anche nel grafico della scalabilità); ciò può essere dovuto al fatto che la macchina aveva 8 core con 2 contesti, per cui da grado di parallelismo 9 in su inizia ad eseguire thread su due contesti di uno stesso core, andando a perdere in performance