

Progetto del corso Sistemi Distribuiti: Paradigmi e Modelli, 2012/2013

Jacopo Notarstefano
jacopo.notarstefano [at] gmail.com

1 Introduzione

Scopo di questo progetto è dare un'implementazione sequenziale e una parallela di una soluzione al problema dell'Histogram Thresholding, per poi confrontarne le performance sia fra esse sia con i modelli teorici.

Entrambe le implementazioni sono realizzate in C++; in particolare quella parallela è data in termini di parallel skeleton, più nello specifico quelli forniti dal framework [SkeTo](#).

SkeTo è uno skeleton framework in cui dichiarazione e istanziamento degli skeleton avvengono allo stesso tempo. I metodi offerti dalle classi astratte del framework, implementate in termini di template, sono quelli tipici della programmazione funzionale, come `map`, `reduce` e `scan`. Queste caratteristiche rendono molto compatto il codice: in effetti l'intero codice non funzionale parallelo di `src/parallel.cpp` ammonta a 63 righe commenti inclusi, di cui appena 3 chiamate di libreria.

Il codice non funzionale sequenziale di `src/sequential.cpp` è un ordinario programma C++. La struttura delle due implementazioni è molto simile: questo perché il business code dell'applicazione è stato incapsulato nella classe `Job`, la cui interfaccia è definita in `src/job.h` e la cui implementazione in `src/job.cpp`.

Per prima cosa presentiamo il problema dell'Histogram Thresholding e descriviamo l'algoritmo risolutivo implementato. In seguito descriviamo in maggiore dettaglio le scelte implementative adottate durante lo sviluppo. Parte centrale del lavoro è il confronto delle performance con le previsioni teoriche, seguito da una dimostrazione dei risultati ottenuti dal programma. Concludiamo con un manuale d'uso che contiene le rule definite nel `Makefile` e altri script disponibili.

2 Il problema dell'Histogram Thresholding

Siano I un'immagine e p una percentuale. Supponiamo di avere accesso ai singoli pixel dell'immagine, e denotiamo con $I[i][j]$ il pixel alla riga i e colonna j . Il problema dell'Histogram Thresholding consiste nel restituire un'immagine in bianco e nero BW tale che

il pixel `BW[i][j]` sia bianco se il pixel `I[i][j]` è più luminoso di $p\%$ pixel dell'immagine originaria, nero altrimenti.¹

L'algoritmo di risoluzione implementato visita ogni pixel, ne calcola la luminosità e la scrive in un vettore. Successivamente lo ordina e ricava la luminosità soglia per la percentuale p . Infine visita nuovamente ogni pixel dell'immagine originaria e, confrontando con il valore soglia precedentemente ottenuto, assegna al corrispondente pixel dell'immagine risultante il colore bianco o nero. Più precisamente abbiamo il seguente:

Algoritmo 1 NAÏVE HISTOGRAM THRESHOLDING

Input: p intero, $0 \leq p < 100$, I immagine di larghezza `imageWidth` e altezza `imageHeight`

```
1: luminosities = []
2: for i ≤ imageWidth do
3:   for j ≤ imageHeight do
4:     luminosities.push(I[i][j].luminosity())
5: luminosities.sort()
6: threshold ← luminosities[luminosities.length() * p / 100]
7: for i ≤ imageWidth do
8:   for j ≤ imageHeight do
9:     if I[i][j].luminosity() > threshold then
10:      BW[i][j] ← white
11:     else
12:      BW[i][j] ← black
13: return BW
```

3 Descrizione delle implementazioni

Abbiamo posto particolare enfasi nella divisione fra codice funzionale e non funzionale; in particolare il business code dell'applicazione è interamente astratto nella classe `Job`.

Osserviamo che i metodi `execute` e `writeResult` hanno come tipo di ritorno `Job`; in effetti questi metodi ritornano l'oggetto stesso. Ciò consente di usarli in `sequential.cpp` come se avessero tipo di ritorno `void` (cioè ignorandone il valore di ritorno), e in `parallel.cpp`, come argomento di `generate` e `map`, dopo averli astratti in opportuni function object come nel seguente esempio:

```
struct generate_t : public sketo::functions::base<Job(size_t)> {
    Job operator()(size_t) const {
        int percentage = rand() % 100;
        return Job(percentage);
    }
};
```

¹ Osserviamo che in letteratura esistono più definizioni di luminosità di un pixel; le soluzioni proposte usano la coordinata L nello spazio colori HSL.

```

    }
} generate;

```

Come già ricordato l'implementazione parallela si avvale di SkeTo, in particolare la versione 1.1. Esiste una versione più aggiornata, la 1.50pre, scartata perché ad oggi priva di documentazione. Internamente SkeTo si avvale di un'implementazione disponibile di MPI; quella utilizzata da questo progetto è la distribuzione [MPICH](#) versione 3.0.4. Infine la lettura e scrittura delle immagini sono interamente delegate alla libreria [ImageMagick](#) tramite [Magick++](#), la sua API per C++, versione 6.8.6-9.

4 Valutazione delle performance

I seguenti test sono stati svolti su [andromeda.di.unipi.it](#), macchina dotata di 16 core. In particolare sono stati utilizzati per la raccolta dati gli script `test_sequential` e `test_parallel` descritti in appendice.

Siano T_{seq} e $T_{\text{par}}(n)$ rispettivamente il service time dell'applicazione sequenziale e di quella parallela con grado di parallelismo n . L'esecuzione degli script `awk -f sequential.awk sequential.dat` e `awk -f parallel.awk parallel.dat` porta a stimare $T_{\text{seq}} \approx 0.86$ e $T_{\text{par}}(n)$ come dalla seguente tabella:

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$T_{\text{par}}(n)$.89	.46	.38	.28	.18	.19	.19	.20	.19	.13	.14	.17	.16	.17	.18	.18

L'esecuzione dello script `awk -f speedup.awk parallel.dat` permette di stimare lo speedup $s(n)$ come segue:

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$s(n)$	0.96	1.85	2.29	3.03	4.67	4.49	4.53	4.39	4.50	6.72	6.03	5.17	5.49	5.18	4.72	4.84

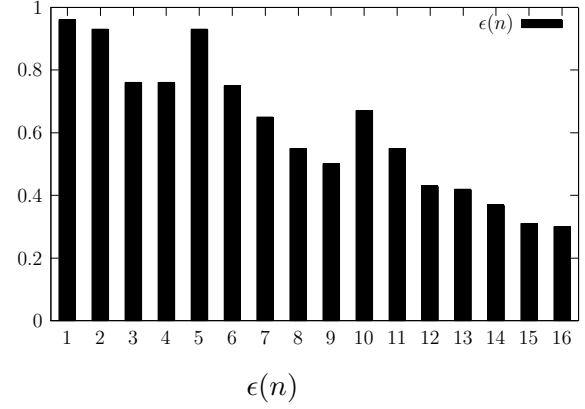
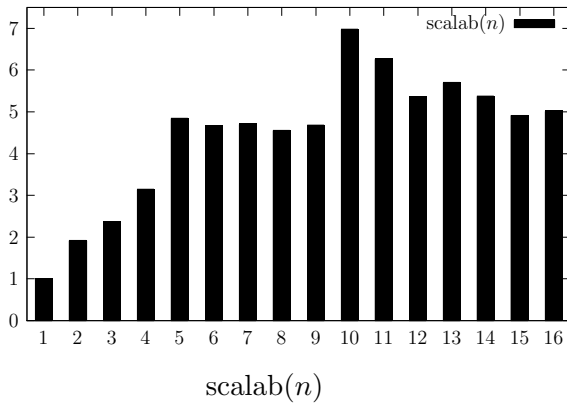
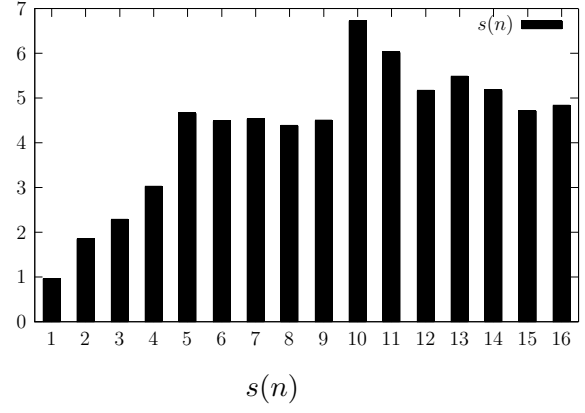
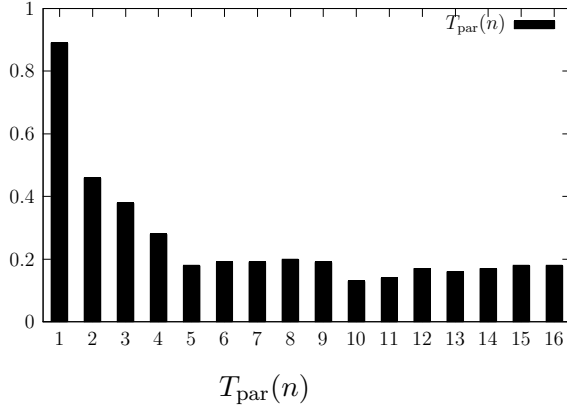
Dallo script `awk -f scalability.awk parallel.dat` ricaviamo le seguenti stime per la scalabilità $\text{scalab}(n)$:

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$\text{scalab}(n)$	1.00	1.92	2.37	3.15	4.85	4.66	4.71	4.56	4.68	6.98	6.26	5.37	5.70	5.38	4.91	5.02

Inoltre dallo script `awk -f efficiency.awk parallel.dat` possiamo stimare l'efficienza $\epsilon(n)$ come in tabella:

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$\epsilon(n)$.96	.93	.76	.76	.93	.75	.65	.55	.50	.67	.55	.43	.42	.37	.31	.30

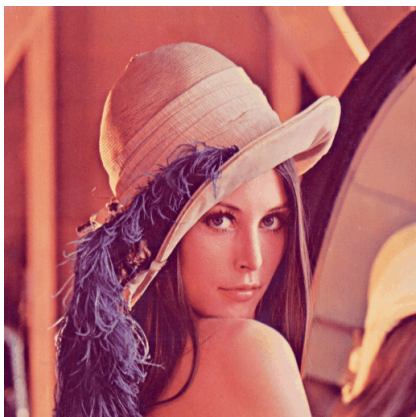
Rappresentiamo infine per immediatezza visiva le precedenti tabelle con istogrammi:



Osserviamo che il programma scala, ma meno di quanto teoricamente atteso, in particolare su più di 10 processori.

5 Valutazione dei risultati

L'algoritmo implementato nel file `src/job.cpp` produce i seguenti risultati al variare della soglia p :



Originale



$p = 20\%$



$p = 40\%$



$p = 60\%$



$p = 80\%$



$p = 99\%$

È inoltre possibile creare, usando l'utility `convert` di ImageMagick, una gif animata i cui frame siano le immagini risultanti al variare della soglia da 0 a 99. Per ottenerla è sufficiente invocare la rule `make gif` del `Makefile` oppure visitare il seguente indirizzo: <http://raw.github.com/jacquerie/SPM/master/lena.gif>.

A Manuale d'uso

A.1 Makefile

clean: Rimuove i file generati dalle rule **parallel**, **sequential** e **tex**.

gif: Genera una gif animata i cui frame sono le immagini risultanti al variare della soglia **p** da 0 a 99. Richiede l'installazione di ImageMagick.

parallel: Compila con **sketocxx** l'implementazione parallela. Richiede l'installazione di SkeTo, una distribuzione di MPI compatibile e ImageMagick.

pdf: Compila con **pdflatex** il presente documento. Richiede l'installazione di **pdflatex** e del pacchetto Python [pygments](#).

sequential: Compila con **g++** l'implementazione sequenziale. Richiede l'installazione di ImageMagick.

A.2 Eseguibili

bin/sequential numberOfJobs: Esegue **sequential** su **numberOfJobs** istanze del problema dell'Histogram Thresholding. Stampa su **stdout** una coppia separata da uno spazio in cui il primo numero è **numberOfJobs** e il secondo il tempo in secondi impiegato da **sequential** a meno di operazioni di lettura e scrittura sul file system.

A.3 Licenza

Il codice è rilasciato con [licenza MIT](#) ed è disponibile su [Github](#).