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 istanziazione 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 gli 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`. È necessario astrarli in opportuni function object, ad 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 detto l'implementazione parallela usa 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`. La lettura e scrittura delle immagini sono interamente delegate alla libreria ImageMagick tramite Magick++, la sua API per C++, versione `6.8.6-9`. Usiamo infine la libreria libpng versione `1.6.3` per la manipolazione di immagini in formato `png`.

4 Valutazione delle performance

Entrambe le implementazioni sono state eseguite sia su una macchina con più core sia su un cluster di più macchine. In questa sezione riportiamo i risultati e li confrontiamo con i modelli teorici.

Analizzando il codice è facile rendersi conto che il problema da risolvere è parallelo sui dati. Ci aspettiamo quindi che il tempo di servizio decresca linearmente con il numero di elementi di calcolo impiegati.

4.1 Multicore

I seguenti test sono stati svolti su `andromeda.di.unipi.it`, macchina con 2 processori Intel Xeon E5520 a 4 core con clock a 2.27GHz, ciascuno dotato di 2 contesti.

Ci aspettiamo quindi uno speedup quasi lineare fino a 8 elementi di calcolo, in seguito un plateau delle performance e un calo dell'efficienza.

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/service_time.awk` e `awk/multicore_service_time.awk` permette di stimare $T_{\text{seq}} \approx 1.12$ 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
$T_{\text{par}}(n)$	1.15	.55	.39	.28	.23	.20	.17	.15	.15	.15	.15	.15	.14	.16	.14	.15

L'esecuzione dello script `awk/multicore_speedup.awk` 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
$s(n)$	0.98	2.03	2.98	4.03	4.95	5.65	6.50	7.43	7.44	7.65	7.42	7.25	7.80	7.21	7.99	7.49

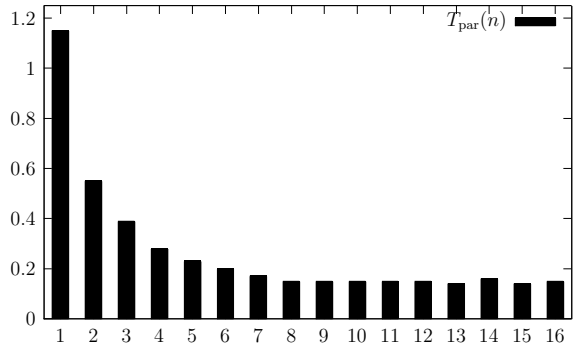
Dallo script `awk/multicore_scalability.awk` ricaviamo le seguenti stime per la scalabilità scalab(n):

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
scalab(n)	1.00	2.08	2.96	4.13	5.07	5.79	6.66	7.62	7.63	7.84	7.60	7.43	7.99	7.39	8.19	7.67

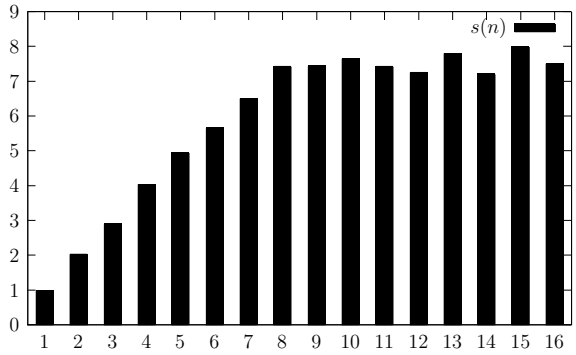
Inoltre dallo script `awk/multicore_efficiency.awk` possiamo stimare l'efficienza $\epsilon(n)$ come in tabella:

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$\epsilon(n)$	.98	1.02	.96	1.01	.99	.94	.93	.93	.83	.76	.67	.60	.60	.52	.53	.47

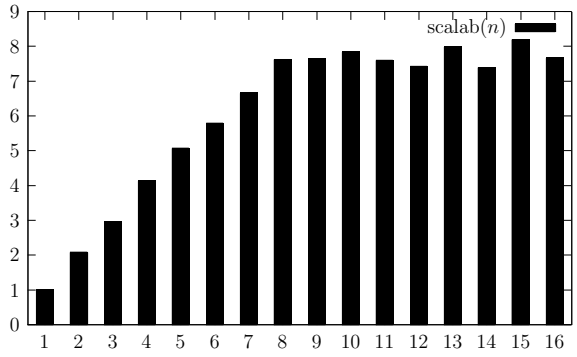
Rappresentiamo infine per immediatezza visiva le precedenti tabelle con istogrammi:



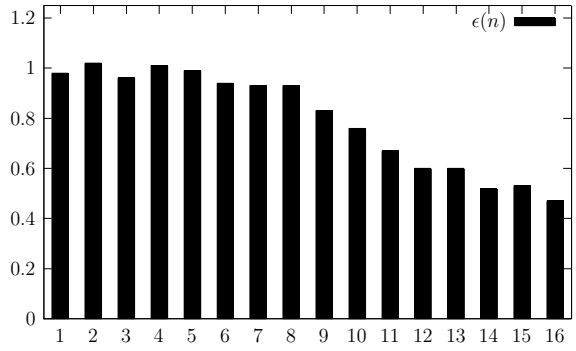
$T_{\text{par}}(n)$



$s(n)$



scalab(n)



$\epsilon(n)$

Osserviamo buona corrispondenza fra le previsioni teoriche e i dati: speedup quasi lineare fino a 8 core, poi un plateau delle performance.

4.2 Cluster

I seguenti test sono stati eseguiti sui computer dell'aula 4 del Dipartimento di Matematica dell'Università di Pisa. Ciascuno di questi possiede un processore AMD A8-3850 a 4 core

con clock 800MHz. Sono teoricamente disponibili 23 computer; in pratica alcuni di questi risultano inaccessibili perchè guasti o spenti. Lo script `bin/generateMachinefile` determina quali macchine sono accese e accessibili via `ssh`; tale informazione viene annotata nel file `Machinefile`. Tipicamente sono disponibili 20 computer.

I dati raccolti prevedono di processare 100 immagini. Avendo a disposizione 80 processori è evidente che non esista convenienza nell'usarne più di 50; infatti la computazione dovrà comunque essere effettuata in due stadi di distribuzione e raccolta di risultati. Per la stessa ragione prevediamo che quantità di processori tale da dividere la computazione in k stadi otterrano performance simili. Insomma prevediamo un andamento "a gradini" dello speedup.

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/service_time.awk` e `awk/cluster_service_time.awk` permette di stimare $T_{\text{seq}} \approx 1.49$ e $T_{\text{par}}(n)$ come dalla seguente tabella:

n	5	10	15	20	25	30	35	40	45	50	55	60	65	70	75	80
$T_{\text{par}}(n)$	.39	.19	.14	.09	.07	.07	.05	.05	.05	.04	.04	.04	.04	.04	.04	.04

L'esecuzione dello script `awk/cluster_speedup.awk` permette di stimare lo speedup $s(n)$ come segue:

n	5	10	15	20	25	30	35	40	45	50	55	60	65	70	75	80
$s(n)$	3.9	7.7	10.9	16.7	20.2	20.5	27.5	27.9	27.3	40.6	42.2	42.4	42.7	41.5	42.5	41.1

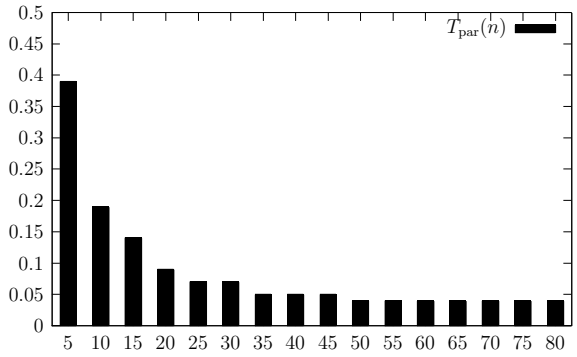
Dallo script `awk/cluster_scalability.awk` ricaviamo le seguenti stime per la scalabilità scalab(n):

n	5	10	15	20	25	30	35	40	45	50	55	60	65	70	75	80
scalab(n)	3.7	7.4	10.5	16.1	19.4	19.7	26.4	26.8	26.2	39.0	40.5	40.7	39.1	39.8	40.8	39.4

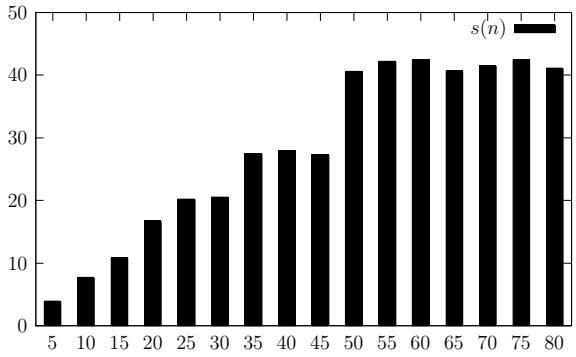
Inoltre dallo script `awk/cluster_efficiency.awk` possiamo stimare l'efficienza $\epsilon(n)$ come in tabella:

n	5	10	15	20	25	30	35	40	45	50	55	60	65	70	75	80
$\epsilon(n)$	.77	.77	.73	.84	.81	.78	.79	.70	.61	.81	.77	.71	.63	.59	.57	.51

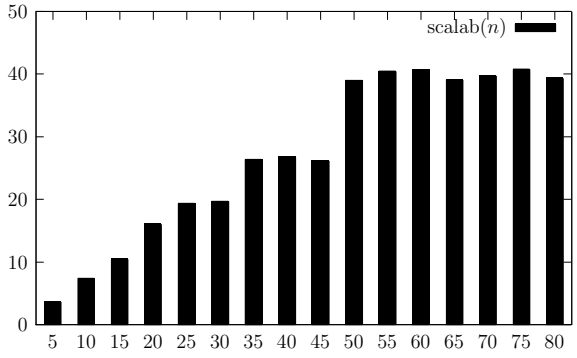
Rappresentiamo infine per immediatezza visiva le precedenti tabelle con istogrammi:



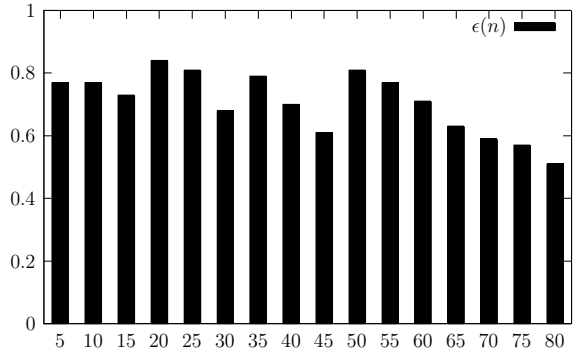
$T_{\text{par}}(n)$



$s(n)$



scalab(n)



$\epsilon(n)$

Osserviamo buona corrispondenza fra le previsioni teoriche e i dati: speedup "a gradini" fino a 50 core, poi un plateau delle performance.

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` eccetto `relazione.pdf`.

`gif`: Genera una gif animata i cui frame sono le immagini risultanti al variare della soglia `p` da 0 a 99.

`nproc`: Restituisce il numero di processori disponibili nel `Machinefile`.

`parallel`: Compila con `sketocxx` l'implementazione parallela.

`pdf`: Compila con `pdflatex` il presente documento.

`sequential`: Compila con `g++` l'implementazione sequenziale.

A.2 Eseguibili

`bin/generateMachinefile`: Genera tramite `ping` e connessioni ssh un `Machinefile` delle macchine funzionanti in aula 4.

`bin/sequential numberOfJobs`: Esegue l'implementazione sequenziale 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 `bin/sequential` a meno di operazioni di lettura e scrittura sul file system.

`bin/sequentialCluster`: Script usato nella generazione dei dati dell'implementazione sequenziale su una macchina dell'aula 4.

`bin/sequentialMulticore`: Script usato nella generazione dei dati dell'implementazione sequenziale su `andromeda.di.unipi.it`.

`bin/parallel`: Binario contenente l'implementazione parallela della soluzione al problema dell'Histogram Thresholding. Richiede di essere invocato tramite `sketorun`.

`bin/parallelCluster`: Script usato nella generazione dei dei dati dell'implementazione parallela sulle macchine disponibili in aula 4.

`bin/parallelMulticore`: Script usato nella generazione dei dati dell'implementazione parallela su `andromeda.di.unipi.it`.

A.3 Licenza

Il codice è rilasciato con licenza MIT ed è disponibile su Github.