RELAZIONE PROGETTO

per l'esame di

Sistemi distribuiti: paradigmi e modelli

Framework: FastFlow
Architettura: Multicore

Stefano Stoduto
438422

Indice

Introduzione

Scopo di questo progetto è quello di fornire due implementazioni di una soluzione al problema dell'Histogram Thresholding. Per prima cosa viene presentato il problema e le specifiche dei test che verranno condotti. Si descriverà quindi un algoritmo risolutivo, che verrà implementato nella sua forma sequenziale. In seguito, analizzando il problema, si individuerà una soluzione in grado di applicare l'algoritmo sfruttando il parallelismo che una macchina multicore è in grado di offrire. Si passerà quindi ad implementare l'algoritmo nella sua versione parallela. Al termine si metteranno a confronto le prestazioni ottenute dall'implementazione parallela rispetto a quelle teoriche.

Problema: Histogram Thresholding

Data un'immagine rappresentata da una matrice di interi *I* e una percentuale *p*, costruire un'immagine binaria *B* tale che B_{ij} è uguale a 1 se non più del *p* percento dei pixel di *I* sono più chiari di I_{ij}. I dati di input sono quindi la matrice *I* e la percentuale *p*. Il thresholding deve essere applicato a uno stream di immagini in input.

Ulteriori specifiche

L'implementazione delle soluzioni sarà data in C++, quella parallela utilizzando i parallel skeleton messi a disposizione dal framework FastFlow.

I test verranno effettuati sulla macchina *andromeda.di.unipi.it*, che dispone di 8 core con hyperthreading su architettura NUMA. Ciascun socket contiene 4 core ed ha accesso a una memoria di circa 6GB, per un totale di 12GB. Una rappresentazione grafica dell'architettura di macchina è disponibile nell'appendice A.

Verrà generato uno pseudo-stream di immagini, ovvero uno stream composto da copie della stessa immagine memorizzato in un array. Sarà possibile specificare la lunghezza di questo stream al momento dell'esecuzione del programma.

Le immagini utilizzate saranno in scala di grigi, in cui il colore viene rappresentato da un intero. Un formato per questo genere di immagini semplice da trattare è il formato PGM. Verranno riportate le performance delle soluzioni proposte utilizzando, per generare lo stream, immagini di diverse dimensioni.

Il tempo impiegato per caricare le immagini, creare lo stream e scrivere i risultati su disco non verrà incluso nelle misurazioni: quello che interessa è infatti studiare le performance del calcolo del thresholding. Inoltre la sezione di codice che accede al disco è una parte inerentemente sequenziale del programma e non può essere parallelizzata.

Algoritmo

Chiamiamo “immagine histogram” l'immagine in bianco e nero prodotta al termine dell'algoritmo. Per poter generare una immagine histogram bisogna conoscere, per ciascuna gradazione di grigio, quanti pixel sono presenti nell'immagine con quella gradazione. Pertanto, una prima fase consiste nell'analisi dell'immagine di input e del calcolo delle frequenze di ciascuna gradazione di grigio.

In secondo luogo bisogna generare un'immagine che abbia pixel bianchi o neri, a seconda della percentuale di presenza di quel colore all'interno dell'immagine originaria. Per fare questo, è sufficiente identificare la gradazione di grigio che soddisfa la seguente condizione:

la somma delle frequenze (percentuali) delle gradazioni più chiare deve essere minore o

uguale alla percentuale di soglia specificata in ingresso.

Oppure, in formule, supponendo che le frequenze percentuali siano memorizzate in ordine dalla più scura (*black*) alla più chiara (*white*) all'interno di un array *freq,* e la percentuale di threshold in ingresso sia *p*, sarà la gradazione di grigio *i* tale che:

$$i \in [\text{black}, \ldots, \text{white}] \quad t.c. \quad \sum_{j=i}^{\text{white}} freq[j] \leqslant p$$

Questa gradazione di grigio viene identificata scorrendo la struttura *freq* che è stata riempita al passo precedente.

Una volta individuata questa particolare "gradazione di soglia", per generare l'immagine histogram è sufficiente una nuova scansione dell'immagine, in cui ogni pixel viene sostituito con uno bianco o nero, a seconda che il suo valore originale è maggiore o minore di *i*.

Riassumendo, l'algoritmo si compone di 3 fasi:

1. Conteggio delle frequenze di ciascuna gradazione di grigio
2. Calcolo della gradazione di soglia
3. Generazione dell'immagine histogram

I passi 2 e 3 dipendono entrambi dal risultato del passo precedente, per cui va atteso il completamento del passo 1 per poter iniziare il passo 2, e allo stesso modo bisogna attendere la terminazione del passo 2 prima di poter iniziare il passo 3.

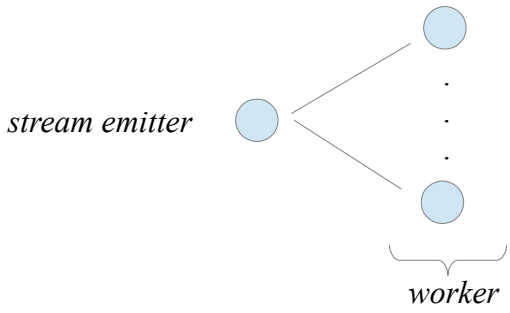
L'implementazione sequenziale sarà un semplice ciclo *for* sulle immagini dello stream (che, si ricorda, è memorizzato in un array), in cui vengono applicati a ciascuna immagine le 3 fasi appena descritte.

Soluzione parallela

Il problema è di tipo *stream parallel* e la soluzione deve tener conto che la macchina target ha un'architettura multicore a memoria condivisa.

L'algoritmo può essere implementato come una farm in cui ogni worker riceve in input il puntatore all'immagine su cui lavorare e effettua sequenzialmente i tre passi visti sopra su ogni immagine che riceve.

Il grafo dei task è il seguente



Non è necessario introdurre alcun collettore nella farm poiché il risultato dell'elaborazione viene lasciato in memoria. Al termine della computazione parallela tutte le immagini elaborate vengono scritte su disco nello stesso ordine in cui sono state caricate, perché le strutture dati che le descrivono sono memorizzate in un array.

Questa scelta rispetta il *two tier model,* traendo beneficio dal ridotto numero di comunicazioni fra

thread e dal consistente lavoro che ciascun worker deve compiere.

La “quantità di lavoro” che un worker deve eseguire è nota come *grana*. Essendo la grana di questa soluzione non troppo fine, il tempo impiegato dai worker per terminare l'elaborazione di una immagine è sufficiente per nascondere il tempo necessario all'emettitore per distribuire i task.

Da notare che il numero di comunicazioni va tenuto in considerazione perché, nonostante esse avvengano in memoria, il tempo impiegato dall'emettitore per effettuare le push nelle code di ingresso dei worker potrebbe risultare comunque non trascurabile se le immagini da elaborare hanno dimensioni ridotte.

Grazie a queste considerazioni si può supporre nel seguito che il tempo impiegato dall'emettitore per trasmettere i task ai worker sia trascurabile rispetto al tempo di elaborazione dei worker.

Prestazioni teoriche

Vengono ora illustrate le performance ideali dell'implementazione parallela. Indicando con:

- T_{seq} il tempo di completamento dell'algoritmo nella sua un'implementazione sequenziale
- nw il numero di worker presenti nella farm
- $T_{par}(nw)$ il tempo di completamento dell'implementazione parallela con *nw* workers

il tempo di completamento ideale per una farm, considerando trascurabile il tempo impiegato dall'emettitore per distribuire i task, è $T_{par}(nw)=\frac{T_{seq}}{nw}$ per cui lo speedup ideale risulta

$$sp(nw)=\frac{T_{seq}}{T_{par}(nw)}=nw$$

Si prevede che le prestazioni effettive saranno influenzate dall'architettura della macchina su cui verrà eseguito il programma e dall'ingresso che verrà fornito; in particolare se lo stream è troppo breve (ovvero composto da poche immagini) non tutti i worker verranno utilizzati, mentre se le immagini sono troppo piccole ciascun worker svuoterà le code per i dati in ingresso troppo velocemente e rimarrà in idle finché non riceverà l'immagine successiva. Viceversa, se le immagini sono molto grandi l'elaborazione richiederà molto tempo, le code si riempiranno e l'emettitore dovrà rimanere in attesa che uno dei worker termini l'esecuzione corrente per poter aggiungere una ulteriore immagine alla sua coda.

Per quanto riguarda il caso di questa applicazione, visto il ridotto numero di core disponibili il caso di stream di lunghezza insufficiente non si verificherà. Inoltre l'implementazione di FastFlow evita anche il caso di avere le code in ingresso dei worker piene, mentre la dimensione dell'immagine scelta per generare lo stream può variare sensibilmente. Ci si aspetta quindi che con immagini troppo piccole non si riuscirà a trarre vantaggio dall'implementazione parallela, mentre immagini grandi non dovrebbero portare particolari problemi.

È bene tenere presente che il caso preso in esame, ovvero uno pseudo-stream formato da immagini tutte uguali tra loro è di per sé già un caso limite, in quanto uno stream reale conterrebbe immagini di dimensioni diverse fra loro. I risultati che si otterranno dovrebbero quindi avvicinarsi a quelli ideali.

Soluzioni alternative

Continuando a rispettare il *two tier model* si potrebbe pensare di inserire un'elaborazione *data parallel* all'interno della farm, ad esempio organizzando ciascun worker come una map. Ci sono

diversi motivi per cui questo approccio viene scartato:

- Il numero di core disponibili dovrebbe essere esageratamente elevato per poter trarre vantaggio da un così alto grado di parallelismo. Infatti, supponendo di avere nw_{farm} worker nella farm e nw_{map} ulteriori worker nella map, il numero totale di core richiesti sarebbe di $1+nw_{farm}\cdot(1+nw_{map})$ (dove i "+1" sono dovuti agli emettitori). Anche con valori molto bassi per questi due parametri si eccede facilmente il numero totale di core disponibili sulla macchina di test.
- Le dimensioni delle immagini da inviare sullo stream dovrebbero essere anch'esse molto grandi; infatti suddividendo ciascuna immagine in parti più piccole il tempo che impiega un worker della map per terminare l'elaborazione si riduce e ci si ritrova nel caso descritto precedentemente in cui qualche worker rimane in idle.
- L'algoritmo presenta delle dipendenze tra dati che non possono essere ignorate, ovvero ciascuno step dipende dai dati prodotti dallo step precedente. Sarebbero dunque necessari almeno 2 stadi di elaborazione: nel primo si eseguono (in sequenza) i passi 1 e 2 dell'algoritmo e nel secondo il passo 3. Il secondo stadio può essere implementato usando una classica map. Se si vuole implementare come map anche il primo stadio si deve tenere presente il problema di gestire l'accesso concorrente all'array *freq* tra i worker. In entrambi i casi sarebbe inevitabile l'introduzione di overhead dovuti almeno alle ulteriori comunicazioni.

Da queste considerazioni si deduce che, data la macchina in uso (8 core fisici) e il tipo di dati in ingresso (immagini di dimensioni contenute), l'implementazione della farm è la migliore scelta per questa applicazione.

Descrizione dei sorgenti

La maggior parte del codice è suddiviso in file *.hpp e *.cpp contenenti rispettivamente la dichiarazione e l'implementazione di funzioni e/o classi. Nonstante in alcuni casi il file *.hpp sia molto semplice e di fatto non strettamente necessario, questi file sono comunque presenti perchè si è cercato di sfruttare alcune funzioni native e regole implicite di *make* all'interno del makefile.

PGM.hpp, PGM.cpp

Le immagini vengono caricate in memoria tramite la classe PGM definita nei file PGM.hpp e PGM.cpp. Questa classe contiene dei metodi per leggere e scrivere l'immagine da/su disco e dei membri che permettono di accedere alle informazioni dell'immagine caricata (come le dimensioni e i singoli pixel che la compongono). Per evitare inutili overhead questi membri hanno visibilità pubblica.

La classe PGM implementa inoltre un costruttore di copia, che viene utilizzato per velocizzare la fase iniziale del programma, in cui una immagine viene letta da disco e viene usata per generare lo pseudo-stream. Sfruttando il costruttore di copia è necessario un solo accesso al disco per leggere l'immagine, mentre le varie copie vengono effettuate direttamente in memoria. Ciò è particolarmente utile nella fase di test e raccolta dati, quando è necessario effettuare diverse esecuzioni del programma.

task.hpp, task.cpp

La classe *task* contiene le strutture dati necessarie a completare l'elaborazione di una immagine. In particolare contiene un membro di tipo PGM che corrisponde all'immagine su cui lavorare, il valore

del threshold da applicare e il (puntatore al) l'array *freq*. Tra i metodi ci sono quelli che corrispondono ai 3 passi dell'algoritmo descritti sopra. Il metodo *write_image()* delega la scrittura dell'immagine su disco al corrispondente metodo di PGM mentre il costruttore di copia è presente per la stessa ragione descritta sopra.

streamer.hpp, streamer.cpp, farm_worker.hpp, farm_worker.cpp

Questi file contengono la definizione delle componenti della farm.

Streamer è il nodo emettitore; viene inizializzato passandogli l'array dei task che costituiscono stream e provvede semplicemente ad assegnare ciascun task ai worker utilizzando lo scheduler predefinito di FastFlow, che applica una politica RoundRobin.

Farm_worker contiene il codice dei worker e provvede a chiamare in sequenza le 3 funzioni del task che riceve in ingresso. Come detto precedentemente, i worker lasciano il risultato della loro elaborazione in memoria, quindi non emettono dati in uscita e la farm non ha un collettore.

util.hpp, util.cpp

Contengono la definizione di due funzioni: *load()* per caricare un'immagine da disco e creare lo pseudo-stream in un array e *write()* per scrivere su disco l'immagine una volta terminata l'esecuzione del programma. La *write()* si occupa anche di liberare la memoria dinamica allocata dalla *load()*. Queste funzioni sono definite a parte perché vengono usate sia nell'implementazione sequenziale che in quella parallela.

Par.cpp e Seq.cpp

Infine questi due file contengono il main rispettivamente per l'implementazione parallela e sequenziale.

Test

I risultati sono stati ottenuti eseguendo i programmi sulla macchina *andromeda.di.unipi.it*.

Per i test è stato utilizzato lo script bash *test.sh*, che permette di eseguire questi programmi con diversi parametri in ingresso, calcolare la media e generare i grafici delle prestazioni in un'unica riga. Nel seguito verranno riportati i parametri passati allo script per generare i risultati mostrati. Una descrizione di questo script è riportata nell'appendice B.

Prima di passare alla presentazione dei dati ottenuti vengono esposte alcune considerazioni sull'architettura della macchina in uso.

Scelta della lunghezza dello stream

La macchina di test ha un'architettura NUMA e dispone di una memoria di 12 GB, suddivisa tra i 2 socket. Questo vuol dire che ciascun core ha accesso diretto solo a 6GB di memoria, mentre per accedere ai restanti 6GB bisogna utilizzare la memoria dei core disposti sull'altro socket. Per evitare questi accessi trasversali che andrebbero a interferire con l'elaborazione del programma è necessario scegliere con attenzione la lunghezza dello stream.

Nel caso tipico ad esempio (v. Dati ottenuti sotto), tenendo presente la dimensione dell'immagine (*flower.pgm*) che si sta usando, quello che si deve evitare è che al momento della generazione dello stream, la memoria richiesta non superi i 6GB. Questo è verificato per stream di lunghezza minore

di 4800 circa.

È tuttavia importante notare che questo accorgimento garantisce che i thread di elaborazione non interferiscono tra loro solo fintanto che questi vengono allocati sui core appartenenti allo stesso socket, ovvero fino a un numero di worker pari a 4 (supponendo che l'emettitore termini il suo compito in un tempo trascurabile, per cui tutti i core saranno impegnati nell'eseguire il codice di uno dei worker). Infatti, utilizzando un numero di worker superiore, i thread verranno inevitabilmente allocati sui core del secondo socket, e supponendo che le immagini vengono poste tutte nei 6GB di memoria accessibili ai primi 4 core ci sarà sempre un minimo di interferenza.

In ogni caso è ovviamente necessario fare in modo che la memoria occupata dalle immagini dello stream sia inferiore ai 12GB disponibili, altrimenti il sistema ricorrerà allo swap e i risultati saranno pesantemente influenzati dai tempi di accesso al disco.

Hyperthreading

Nonostante la macchina sia dotata di core che supportano l'hyperthreading, e quindi sia possibile allocare 2 worker sullo stesso core, questa particolare applicazione non è in grado di sfruttare tale tecnologia.

Questo è dovuto al fatto che, se i worker lavorano a regime (ovvero se le code dei task in ingresso sono sempre piene), ogni core sarà sempre impegnato a eseguire il codice di uno dei worker ad esso allocati. In altre parole non ci saranno dei momenti in cui uno dei due worker allocati sullo stesso core si troverà in idle, e quindi il parallelismo offerto dall'hyperthreading non potrà essere sfruttato appieno.

Ciò si tradurrà in uno speedup massimo pari al numero di core “fisici” (8 sulla macchina di test), anche se vengono utilizzati tutti i (16) core “logici”.

Dati ottenuti

Sono stati effettuati diversi test: il caso “tipico” utilizza l'immagine *flower.pgm*, avente dimensioni 1280x1024 pixel, mentre per i casi estremi sono state utilizzate le immagini *spider.pgm* e *bowtie.pgm*, aventi dimensioni rispettivamente 200x200 e 4000x3000 pixel.

I valori considerati sono il tempo di completamento del codice sequenziale e parallelo; in base a questi vengono calcolati anche la scalabilità e lo speedup, secondo le formule:

$$scalabilità = sc(nw) = \frac{T_{par}(1)}{T_{par}(nw)} \quad , \quad speedup = sp(nw) = \frac{T_{seq}}{T_{par}(nw)}$$

Tutti i test sono stati effettuati utilizzando una soglia del 50%. Inoltre i valori sono stati calcolati eseguendo il programma per 5 volte, scartando il miglior e il peggior risultato e eseguendo la media aritmetica sui 3 rimanenti.

Di seguito sono riportati i valori del tempo di completamento, scalabilità e speedup ottenuti su uno stream di 2000 copie dell'immagine *flower.pgm*.

Questi dati sono stati ricavati eseguendo lo script *test.sh* con i seguenti parametri:

```
./test.sh -f images/flower.pgm -s 2000 -e 5
```

dove l'opzione -f specifica il file da utilizzare per generare lo stream, -s la lunghezza dello stream e -e quante volte i programmi (sia sequenziale che parallelo) verranno eseguiti al fine di calcolare la media del tempo di completamento. Come detto, si rimanda all'appendice B per una spiegazione migliore delle opzioni.

Nota: tutti i tempi sono misurati in millisecondi.

$streamlen = 2000 \quad , \quad T_{seq} = 7757{,}01$

nw	T_{par}	*Scalabilità*	*Speedup*
1	7752,92	1	1
2	3872,85	2	2
4	1933,33	4,01	4,01
8	1042,69	7,44	7,44
16	995,58	7,79	7,79

La scelta della lunghezza dello stream rispetta i vincoli sulla memoria descritti sopra e infatti le prestazioni sono quelle attese fino al grado di parallelismo 4. Utilizzando 8 worker si riscontra un leggero rallentamento dovuto agli accessi trasversali alla memoria dei 4 core del secondo socket. Infine è evidente il limite delle prestazioni quando entra in gioco l'hyperthreading.

Di seguito sono riportati i grafici del tempo di completamento e dello speedup.

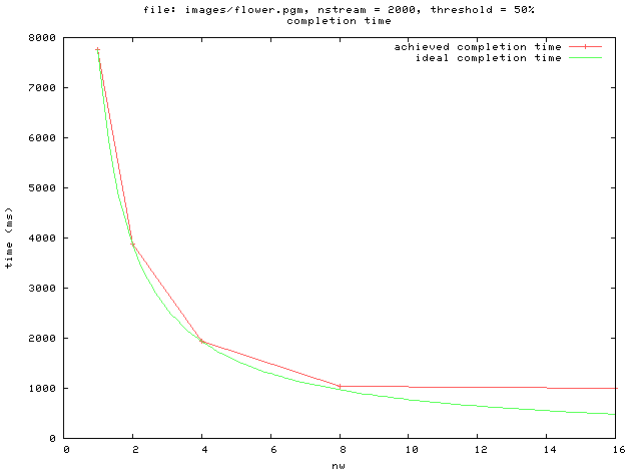


Figura 1: Tempo di completamento

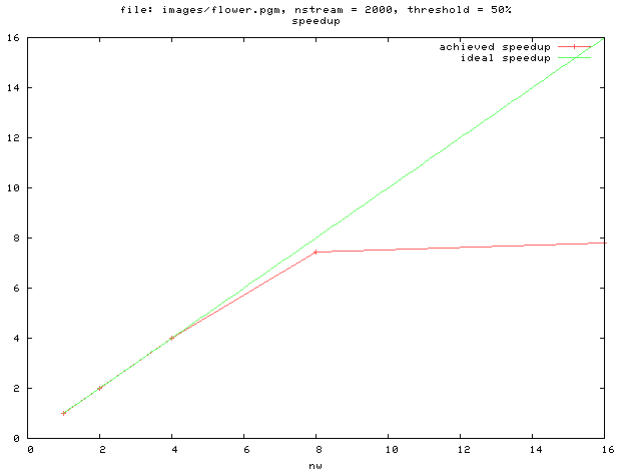


Figura 2: Speedup

Infine vengono mostrati i due casi estremi in cui per generare lo stream vengono utilizzate immagini molto piccole e molto grandi. Di entrambi si riporta a titolo di esempio solo il grafico dello speedup.

Come esempio di immagine “piccola” usiamo *spider.pgm* che ha dimensioni 200x200 pixel.

Questi dati sono stati ricavati eseguendo lo script *test.sh* con i seguenti parametri:

```
./test.sh -f images/spider.pgm -s 2000 -e 5
```

$streamlen = 2000$, $T_{seq} = 239{,}409$

nw	T_{par}	*Scalabilità*	*Speedup*
1	239,7	1	1
2	131,3	1,83	1,82
4	75,98	3,15	3,15
8	54,9	4,37	4,36
16	51,64	4,64	4,64

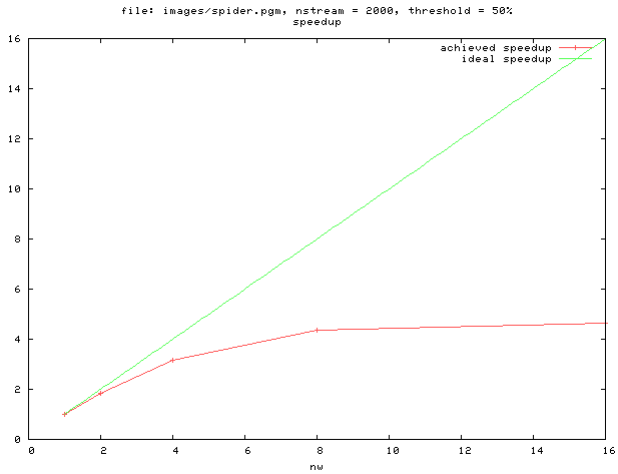


Figura 3: Speedup spider.pgm

Si vede subito che le prestazioni degradano molto velocemente. Questo è dovuto al fatto che, essendo l'immagine molto piccola, ciascun worker termina velocemente la sua elaborazione, prima ancora che lo schedulatore abbia la possibilità di assegnargli una nuova immagine. Il degrado peggiora con l'aumentare dei worker perché lo schedulatore predefinito di FastFlow applica una politica round robin, per cui più sono i worker più sarà il tempo necessario a effettuare le push in tutte le loro code di ingresso prima di poter ritornare al primo.

Ora consideriamo il caso di immagine "grande". Prendiamo a questo scopo l'immagine *bowtie.pgm* che ha dimensione 4000x3000 pixel.

Facendo i conti risulta che per non eccedere i 6GB di memoria bisogna usare al più 527 immagini circa, quindi per il test è stato usato un stream di 500 immagini.

I dati seguenti sono stati ricavati eseguendo lo script *test.sh* con i seguenti parametri:

```
./test.sh -f images/bowtie.pgm -s 500 -e 5
```

$streamlen = 500$, $T_{seq} = 17901{,}9$

nw	T_{par}	*Scalabilità*	*Speedup*
1	17998	1	0,99
2	8968,83	2,01	2
4	4519,76	3,98	3,96
8	2373,75	7,58	7,54
16	2301,38	7,82	7,78

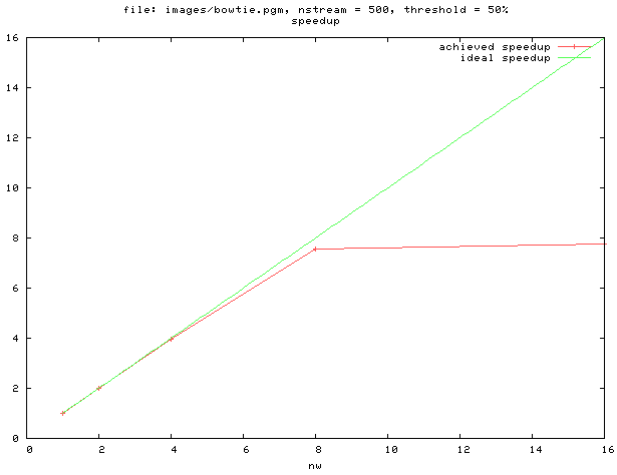
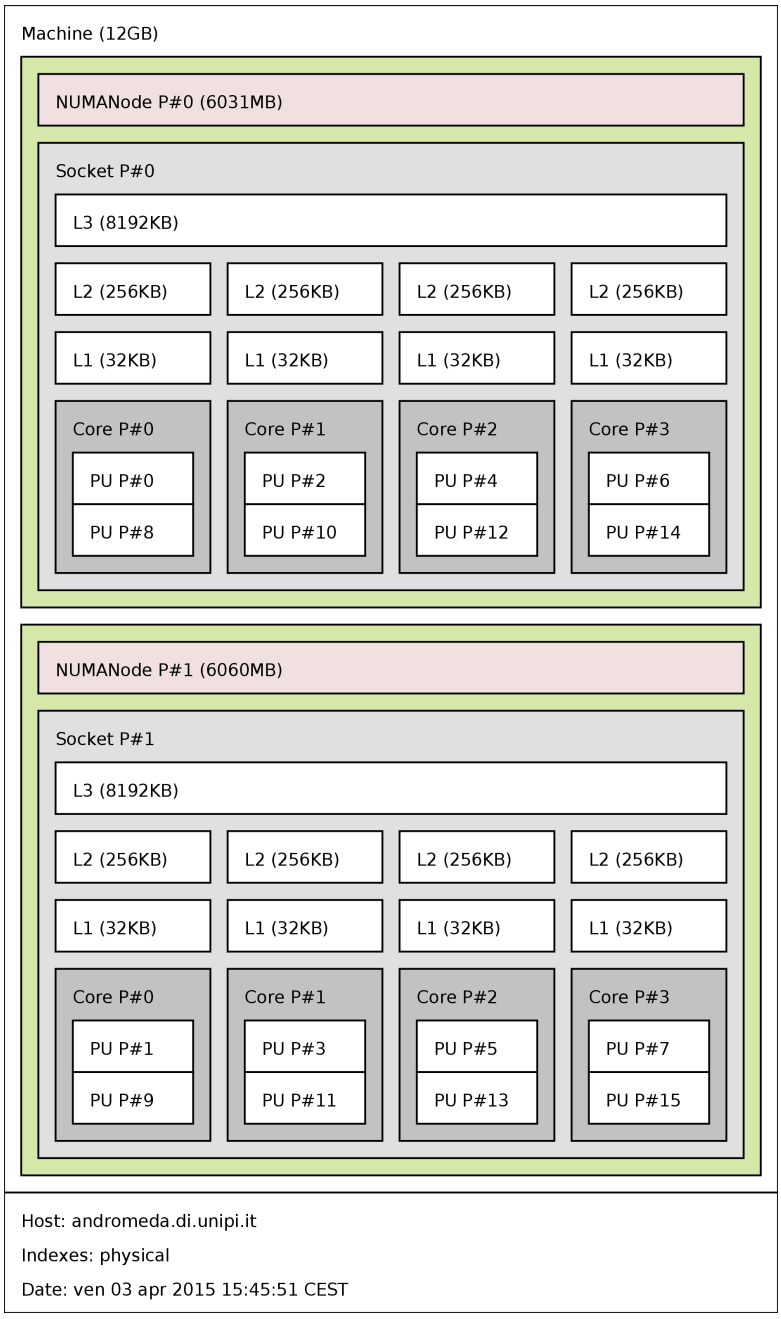


Figura 4: Speedup bowtie.pgm x 500

Come ci si aspettava le prestazioni risultano analoghe a quelle ottenute nel caso tipico. La conclusione è che utilizzare un'immagine molto grande per generare lo stream non comporta particolari svantaggi se non quello di riempire molto velocemente la memoria.

Appendice A – architettura di macchina

Architettura della macchina *andromeda.di.unipi.it* a 8 core con hyperthreading. Questo schema è stato ottenuto eseguendo il comando *hwloc-info* sulla macchina stessa.



Appendice B – manuale utente

Compilazione

Per compilare il codice basta eseguire il comando *make* nella cartella in cui sono stati estratti i file. È anche possibile compilare singolarmente il codice parallelo e sequenziale, rispettivamente usando i target *par* e *seq*. Il makefile presuppone la presenza di una cartella *bin/* che conterrà il codice oggetto prodotto, mentre gli eseguibili saranno posti nella cartella principale.

Sono anche disponibili i seguenti target per eliminare i file prodotti dalla compilazione e dai test:

clean: elimina i file oggetto in bin/ e i binari del codice sequenziale e parallelo

clean_graphs: elimina i file in graphs/ prodotti dallo script di test

clean_output: elimina le immagini in output/ prodotte dall'esecuzione del programma

Esecuzione

La compilazione creerà i file eseguibili *par* e *seq* nella cartella principale. Entrambi prevedono dei parametri in ingresso. Per il sequenziale si ha:

```
./seq <img_name> <threshold> <streamlen>
```

Mentre *par* aggiunge solo il parametro relativo al grado di parallelismo:

```
./par <img_name> <threshold> <streamlen> <nw>
```

dove:

- <img_name> è il nome dell'imagine da usare per generare lo pseudo-stream
- <threshold> è il valore (percentuale) della soglia da usare per generare le immagini di output
- <streamlen> è la lunghezza dello pseudo-stream
- <nw> è il numero dei worker da utilizzare nella farm

Gli eseguibili presuppongono la presenza di una cartella *output/* in cui verranno create le immagini histogram.

Test

Per generare i dati e i grafici dei test è stato usato lo script bash *test.sh*. Questo prevede delle opzioni a riga di comando. Di seguito vengono riportate le principali (La lista completa è visualizzabile chiamando lo script con l'opzione -?):

`-f` permette di specificare quale file o lista di file passare come parametro <img_name>

`-s` permette di specificare la lunghezza dello stream o una lista di lunghezze dello stream da usare come parametro <streamlen>

`-w` permette di specificare il numero massimo di worker da utilizzare per costruire i grafici. I programmi vengono eseguiti passando al programma parallelo un numero di worker <nw> da 1 fino a questo valore, avanzando per potenze di 2

`-e` permette di specificare quante volte ciascun programma (sia *par* che *seq*) deve essere eseguito per calcolare la media dei tempi di completamento. Se questo valore è minore o uguale a 3 lo script effettua semplicemente la media aritmetica, mentre per valori maggiori

di 3 calcola la media scartando il miglior e il peggior valore.

Le opzioni -f e -s accettano anche “liste” di valori. In questo caso i valori dei parametri vanno inseriti tra doppi apici e separati da spazi. Ad esempio è possibile eseguire un test sulle immagini *flower.pgm* e *spider.pgm* entrambe su uno stream di lunghezza 2000 in questo modo:

```
./test.sh -f “images/flower.pgm images/spider.pgm” -s 2000
```

Lo script produce 3 grafici e un file di testo contenente i dati numerici per ciascuna “esecuzione” (ovvero per ciascuna combinazione immagine-lunghezza dello stream) all'interno della cartella *graphs/*, che deve esistere al momento dell'esecuzione dello script. I nomi dei file contengono le informazioni relative all'esecuzione a cui si riferiscono. Ad esempio, il file *data.flower.pgm.2000.txt* contiene le misurazioni relative all'esecuzione con lunghezza dello stream di 2000 sul file *flower.pgm*, mentre *speedup_flower.pgm.2000.pbm* contiene il relativo grafico dello speedup.

Su terminale è possibile visualizzare il contenuto dei file data.* in maniera leggibile utilizzando il comando *column* con l'opzione -t, ad esempio:

```
column -t graphs/data.flower.pgm.2000.txt
```

Riferimenti

PGM - Portable Gray Map format specification

Multiprogramming Performance of the Pentium 4 with Hyper-Threading

Note del corso