SPM progetto finale: Image Watermarking

Simone Schirinzi

7 giugno 2018

1 Introduzione

In questo progetto ho implementato un programma per l'applicazione di un watermark in bianco e nero a un insieme di immagini in formato jpg. L'implementazione è stata sviluppata utilizzando sia i meccanismi standard di c++ che la libreria Fastflow. Per la gestione del formato jpg è stata usata la libreria CImg che al suo interno utilizza libjpeg. Il programma è stato testato su un server con 256 core su 4 Intel xeon phi da 64 core l'uno.

2 Design dell'architettura parallela

Per applicare il filtro ad un immagine sono necessarie tre operazioni:

Open() Per la lettura del file da disco, la decompressione e il caricamento dell'immagine;

Filter() Per l'applicazione del watermark;

Close() Per la ricompressione dell'immagine, la scrittura del file su disco e la deallocazione della memoria occupata.

Open() e Close() sono funzioni prettamente di libreria e implementate in sequenziale, mentre l'implementazione di Filter() è discussa nel paragrafo 3.1. Le tre operazioni vanno effettuate nella sequenza sopra descritta.

Un problema così posto avrebbe la sua naturale evoluzione in una pipeline a tre stadi. Quindi ogni stadio andrebbe parallelizzato al suo interno. Tuttavia essendo Open() e Close() funzioni di libreria non possono essere parallelizzate. Quindi si sfrutta il fatto che ogni immagine può essere elaborata in maniera indipendente dalle altre. In questo modo il parallelismo puo essere ottenuto utilizzando una farm in cui ogni nodo opera su un immagine applicandone in sequenza le tre operazioni. La farm usa una politica ondemand in quanto nonostante le immagini abbiano tutte le stessa dimensione per definizione del problema, il tempo necessario per la decompressione e ricompressione del file in formato jpg può variare in base all'entropia propria dell'immagine. Da un modello così strutturato, considerando che la componente sequenziale del programma è minima, ci si aspetta che abbia un comportamento prossimo a quello ideale:

$$T_{seq} = T_{init} + n * (T(read) + T(filter) + T(write)) \tag{1}$$

$$T_{par}(k) = T_{init} + T_{initFarm} + n * \frac{T(read) + T(filter) + T(write)}{k} \tag{2}$$

2.1 Pipeline di farm

Profilando il codice emerge che la funzione Filter() richiede circa il 10% delle istruzioni globali del processo, al contrario il 57% delle istruzioni viene eseguito all'interno della libreria libjpeg, più un ulteriore 33% da CImg.h, esclusivamente per le operazioni di Open() e Close() facendole diventare dei colli di bottiglia.

Per venire incontro a questo problema è stato ipotizzato un ulteriore modello strutturato come una pipeline di farm asimmetriche in cui il primo e il terzo stadio hanno più lavoratori del secondo per compensare la disparità dei tempi computazionali. Per quanto riguarda le performance teoriche di un modello così strutturato di ci si aspetta un migliore utilizzo della banda di I/O che andrebbe a coprire i costi di comunicazione tra gli stadi del pipeline.

Incl.	Self	Called	Function	Location
100.05	61.83	21 877	<cycle 1>	libstdc++.so.6.0.21
17.17	17.17	1 303	cimg_library::CImg<>::_load_jpeg(_I...	spm_final: CImg.h
16.67	16.60	1 302	cimg_library::CImg<>::_save_jpeg(_I...	spm_final: CImg.h
14.66	14.65	5 330 822	0x0000000000007350 <cycle 1>	libjpeg.so.8.0.2
11.15	11.15	10 008 845	0x000000000001ce10 <cycle 1>	libjpeg.so.8.0.2
8.86	0.48	82 894	0x0000000000004b40 <cycle 1>	libjpeg.so.8.0.2
8.59	1.12	108 065	0x00000000000190f0 <cycle 1>	libjpeg.so.8.0.2
8.38	0.54	21 242 144	0x0000000000006560	libjpeg.so.8.0.2
7.48	7.48	40 658 315	0x00000000000343d0	libjpeg.so.8.0.2
6.30	0.05	1 303 000	0x0000000000026c60	libjpeg.so.8.0.2
6.25	0.04	1 383 592	0x0000000000015df0 <cycle 1>	libjpeg.so.8.0.2
6.13	6.13	1 288 000	0x0000000000033e60	libjpeg.so.8.0.2
5.08	0.00	1 303	std::vector<>::vector(std::vector<> ...	spm_final: stl_vector.h
5.08	0.00	1 303	std::pair<>* std::__uninitialized_cop...	spm_final: stl_uninitialized.h
5.08	0.00	1 303	std::pair<>* std::uninitialized_copy<...	spm_final: stl_uninitialized.h
5.08	0.98	1 303	std::pair<>* std::__uninitialized_cop...	spm_final: stl_uninitialized.h
5.01	5.01	31 224 144	0x00000000000342d0	libjpeg.so.8.0.2
4.79	0.02	880 000	0x000000000001ed40 <cycle 1>	libjpeg.so.8.0.2
4.56	4.56	1 289 000	0x0000000000033f80	libjpeg.so.8.0.2
3.43	0.00	4	concurrentExecution(std::vector<>, ...	spm_final: main.cpp
3.43	2.45	1 302	filter(cimg_library::CImg<>, std::vect...	spm_final: stuff.cpp
1.72	1.72	31 224 144	0x0000000000034320	libjpeg.so.8.0.2
1.60	1.02	101 736 937	void std::_Construct<>(std::pair<>*,...	spm_final: stl_construct.h
1.54	0.01	423 000	0x000000000001eca0 <cycle 1>	libjpeg.so.8.0.2
1.48	1.48	880 000	0x0000000000034100	libjpeg.so.8.0.2
1.42	0.89	101 738 240	bool __gnu_cxx::operator!=<>(__gnu...	spm_final: stl_iterator.h
1.11	1.11	31 224 144	0x0000000000034280	libjpeg.so.8.0.2
0.98	0.98	203 317 716	std::vector<>::operator[](unsigned l...	spm_final: stl_vector.h
0.70	0.02	658 000	0x00000000000165a0 <cycle 1>	libjpeg.so.8.0.2
0.68	0.68	1 288 000	0x0000000000034040	libjpeg.so.8.0.2
0.53	0.53	203 476 480	__gnu_cxx::__normal_iterator<>::bas...	spm_final: stl_iterator.h
0.49	0.49	101 736 937	__gnu_cxx::__normal_iterator<>::ope...	spm_final: stl_iterator.h

Figura 1: KCacheGrind callmap

$$T_{par}(k) = T_{init} + 3 * T_{initFarm} + T_{initPipe} + \frac{n * max\{\frac{T(read)}{asim} + T(filter) + \frac{T(write)}{asim}\}}{\frac{k}{2*asim+1}} \quad (3)$$

3 Panoramica del progetto

Il progetto si compone di 3 file:

stuff.cpp Funge da interfaccia per CImg.h e la gestione dei file e implementa le funzioni per preparare le immagini da leggere e la Filter();

BlockingQueue.cpp Implementa una coda ad accesso concorrente con push libera e pop bloccante;

main.cpp Definisce l'architettura (parallela o sequenziale) del programma.

3.1 Filter()

CImg memorizza le immagini come un vettore quadridimensionale di byte. La funzione di filter itera sui punti del watermark (il cui trattamento è descritto nel paragrafo 3.2) e riduce i calcoli per l'accesso ai byte corrispondenti ai punti da annerire. L'operazione poteva essere migliorata comprimendo con LRE il watermark per punti consecutivi. Questo non è stato fatto in quanto è emerso che la funzione di filter richiedeva già solo un piccola porzione del tempo totale.

3.2 Main.cpp

Fulcro del progetto, antepone all'esecuzione delle funzioni di elaborazione delle immagini una fase di preparazione durante la quale:

- Si ha la gestione degli argomenti del programma, descritti nel paragrafo 3.3;
- Avviene il parsing del watermark: quest'ultimo infatti viene letto una volta sola e memorizzato come un vettore di coordinate 2D di punti da annerire;
- Vengono preparati i file da trattare: legge la cartella di input e genera una lista dei nomi dei file presenti all'interno di essa che abbiano estensione jpg.

All'interno di main.cpp son presenti 5 diverse implementazioni della soluzione del problema:

sequentialExecution Implementa la versione sequenziale del programma tramite un loop sulle immagini applicando in sequenza Open() Filter() e Close();

concurrentExecution Implementa il modello farm tramite l'utilizzo di thread il cui compito è estrarre un task sequenziale da una BlockingQueue ed eseguirlo;

concurrentPipelineExecution Implementa il modello pipeline avviando tre farm le quali vengono messe in comunicazione tramite tre BlockingQueue. Ogni farm è composta da un diverso numero di workers ognuno dei quali estrae un task (una singola funzione) da una coda, lo esegue e lo invia alla coda successiva;

fastFlowExecution Implementa il modello farm utilizzando fastflow. Un emitter inizializza il task sequenziale che viene inviato ad un worker per l'esecuzione;

fastFlowPipelineExecution Implementa il modello pipeline utilizzando fastflow. Un emitter inizializza il task per la farm di Open() che è messa in comunicazione con la farm di Filter() e quella di Close() tramite una pipeline.

3.3 README

Come compilare ed eseguire un test:

```
1 unzip wm.zip -d wm
2 cd wm
3 cmake .
4 make
5 ./spm_final -d dataset_small -m watermark.jpg -w 4 -t
```

Utilizzo: ./spm_final -d dir -m mark -w nw [-f, -p par, -t, -o]

Argomenti dell'eseguibile:

- -d <dir>: directory delle immagini
- -m <mark>: watermark da applicare
- -w <nw>: workers da avviare
  - se nw $< 0 \Rightarrow$ esegue l'implementazione di fastflow
- -f : esegue l'implementazione di pipeline di farm
- -p <par>: grado di parallelismo asimmetrico nel pipeline di farm
- -t : test completo raddoppiando il numero di lavoratori ad ogni iterazione fino al nw
- -o : directory dove salvare le immagini modificate

4 Esperimenti

Gli esperimenti son stati eseguiti su un server da 256 core utilizzando un dataset da 1302 immagini, disponibile sul server nella cartella /home/spm18-schirinzi/openDataset/. Sono stati oggetto di test sia il modello farm che il modello pipeline farm con grado di asimmetria 4, in quanto è risultato il migliore nel test mostrato in figura 2.
Essendo la pipeline composta da tre stadi questo richiede che il grado di parallelismo dei test riguardanti la pipeline sia almeno 3.

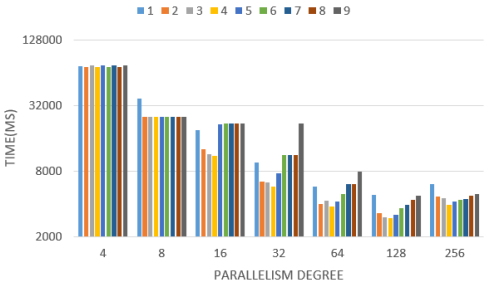
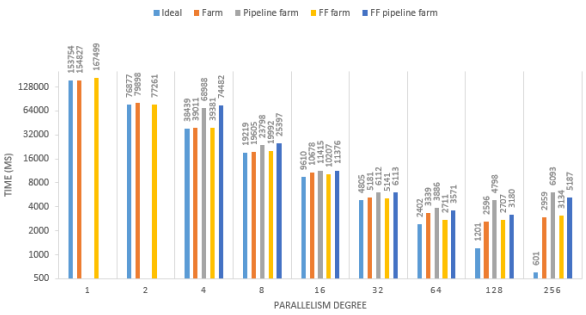


Figura 2: Pipeline farm

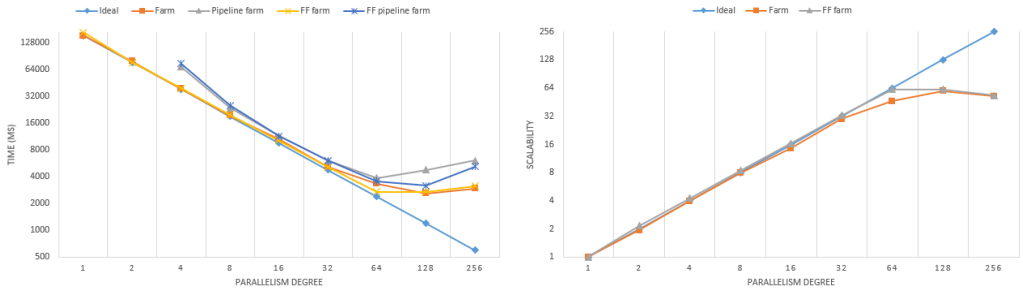
L'esperimento è stato eseguito con l'obbiettivo di verificare il comportamento delle diverse implementazioni del programma al variare del grado di parallelismo.

La componente sequenziale del programma (T_{init}) è trascurabile e viene ignorata, infatti dai test emerge avere un valore medio di 150ms. Il tempo ideale è ottenuto come il tempo sequenziale diviso il grado di parallelismo.

	Ideal	Farm	Pipe	FF farm	FF pipe
1	153754	154827	-	167499	-
2	76877	79898	-	77261	-
4	38439	39011	68988	39381	74482
8	19219	19605	23798	19992	25397
16	9610	10678	11415	10207	11376
32	4805	5181	6112	5141	6113
64	2402	3339	3886	2711	3571
128	1201	2596	4798	2707	3180
256	601	2959	6093	3134	5187

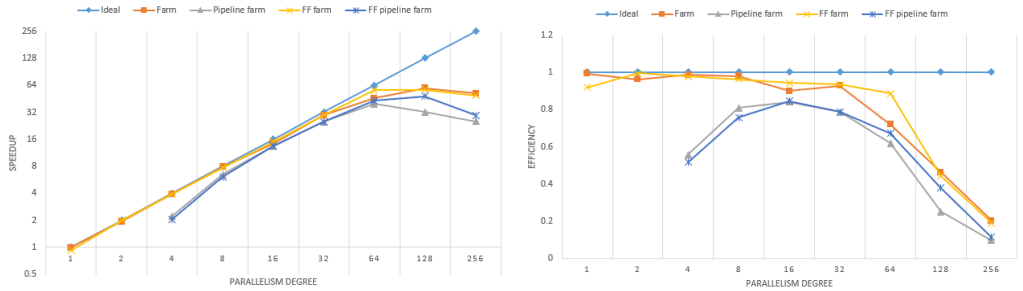


Si noti come i risultati migliori si ottengano utilizzando il modello farm la cui scalabilità è quasi ideale fino ai 64 workers. Oltre quel valore viene saturata la banda di I/O, emersa sui 250MB/s, portando a un peggioramento delle prestazioni generali. I tempi rispettano il modello teorico riportato in formula 2 rendendo trascurabile anche $T_{initFarm}$. Lo stesso non avviene per quanto previsto in formula 3 che in teoria dovrebbe diventare $\frac{2*T(read)}{k}$ ma che in pratica, in presenza di pochi nodi risente della ripartizione non eccellente dei worker; riesce ad ottenere risultati accettabili con un medio grado di parallelismo e in fine degrada le sue prestazioni prima del modello farm.



(a) Latenza (b) Scalabilità

Figura 3: Performance degli esperimenti



(a) Speedup (b) Efficienza

Figura 4: Performance degli esperimenti

Il grafico 3a evidenzia come l'implementazione di fastflow fornisca risultati paragonabili alla sua controparte non fastflow ma, come emerge nel grafico 3b, FF Farm ha un efficienza migliore oltre i 32 workers. Questo è dovuto all'overhead della BlockingQueue.

Osservando invece l'implementazione Pipeline nel grafo 4b emergono dei risultati poco soddisfacenti. Infatti tale implementazione, sia in versione fastflow che non, produce i tempi peggiori in assoluto. L'implementazione con fastflow ottiene risultati appena migliori. Tuttavia in entrambi i casi l'overhead di comunicazione dell'immagine tra gli stadi del pipeline risulta significativa e non bilanciata dalla presenza di più nodi atti alle operazioni di Open() e Close().

5 Conclusioni

In questo progetto ho implementato un programma per l'applicazione di un watermark a un insieme di immagini in formato jpg. L'implementazione parallela, utilizzando il modello farm, scala molto bene e rispetta i risultati teorici ma è sensibile al limite imposto dalla banda di I/O dovuto ai continui accessi su disco per la lettura e scrittura del file. E' stata anche considerata un implementazione tramite pipeline ma con scarsi risultati. In futuro potrebbe essere interessante migliorare il modo in cui avviene la lettura e la scrittura dei file su disco in modo da accedervi in maniera sequenziale. Oppure migliorare la fase di Filter comprimendo il watermark.