Reducing Video Size
Implementazione per architetture multi-core mediante l'utilizzo delle librerie FastFlow e Threads C++11

Maurizio Idini
maurizio.idini [at] gmail.com
SPM Final Project (A.A. 2015-16)
6 luglio 2016

Abstract

In questa relazione si analizza la risoluzione e l'implementazione sequenziale e parallela del problema del **Reducing Video Size**, ossia dato un video, si riduce la grandezza di ogni frame secondo un dato parametro, lavorando su architetture a più cores. L'implementazione parallela è stata attuata mediante l'utilizzo della libreria **FastFlow** e della libreria standard **Thread C++11**. Si son confrontate poi le prestazioni fra loro e con i modelli teorici.

1 Introduzione

Un file video, generalmente di formato .avi, può essere descritto come una sequenza di frames, ossia matrici codificate in un determinato modo. Generamente, la codifica più comune definisce ogni pixel del frame come una tupla di t valori, corrispondente ai canali di codifica del frame, come RGB o HSV, e che dunque rappresentano il colore del pixel. Ad esempio, OpenCv codifica generalmente i frames mediante la macro `CV_8UC3`, dove 8 sono i bits che rappresentano il valore, U denota che il valore è Unsigned, e dunque il valore è espresso nel range [0,255], 3 denota il numero di canali. Ad esempio,

un elemento di un frame, ossia un'immagine definita da una matrice (che in OpenCV è rappresentata dall'oggetto Mat), è una terna [12,136,200].

Operare dunque su un file video significa operare su ogni singolo frame un'operazione (conforme), ossia

- accedere al video in input e creare un nuovo video in output
- accedere ad ogni singolo frame del video in input
- applicare un'operazione al frame corrente
- scrivere il frame risultante nel nuovo video

Dunque l'operazione da applicare al frame può generalmente essere un filtro `Blur`[1], un `Thresold`, una `Resize` della dimensione del video, o qualsiasi altra operazione, seppur ammissibile.
In questo progetto si è scelto di applicare la `cv::resize`[2] come operazione sui frames. Più avanti si formalizzerà il problema con più precisione.
Le implementazioni proposte sono realizzate in C++11, in particolare quelle parallele lavorano mediante la libreria **FastFlow**[3] e la libreria standard **Thread C++11**[4]. Ogni implementazione utilizza la libreria **OpenCV**[5] per operare su video e immagini.

FastFlow è un C++ parallel programming framework, supporta stream e data parallelism e la dichiarazione ed istanziazione degli skeleton è molto semplice ed immediata, il che permette di scrivere poco codice non funzionale e concentrare il lavoro su quello funzionale.

La **Standard-Library Thread** è il nuovo supporto alla programmazione Multi-Threading offerta da C++11; tale libreria si appoggia alla nota `Pthread`, ma risulta molto più chiara e user-friendly.

[1] `http://docs.opencv.org/3.1.0/da/d54/group__imgproc__transform.html#gsc.tab=0`

[2] `http://docs.opencv.org/3.0.0/da/d54/group__imgproc__transform.html#ga47a974309e9102f5f08231edc7e7529d`

[3] `http://calvados.di.unipi.it/`

[4] `http://www.cplusplus.com/reference/thread/thread/`

[5] `http://www.opencv.org`

L'implementazione parallela risulta molto differente utilizzando queste due librerie: da un lato, si ha la possibilità di concentrarsi sul codice funzionale, per assurdo quasi trascurando come avviene il parallelismo o come la libreria condivida, invia e processa i dati, e questo è il caso di FastFlow. Dall'altro lato, quasi non si distingue più fra codice funzionale e codice non funzionale, e questo è il caso di Thread C++11.

La libreria **OpenCV** ha una propria parallelizzazione interna (si può verificare mediante il metodo `getNumThreads()`), ma in tutte le implementazioni si è provveduto a settare il grado di parallelismo interno di OpenCV uguale a 1 mediante il metodo `setNumThreads(0)`.

Per prima cosa si presenta formalmente il problema del Reducing Video Size e si descrive l'algoritmo implementato. In seguito si descrivono le scelte implementative adottate durante lo sviluppo. Parte centrale è il confronto delle performance con le previsioni teoriche, seguito da una dimostrazione dei risultati ottenuti dal software. Si conclude col manuale d'uso per il software.

2 Design

La fase di design ha portato all'implementazione dei tre algoritmi: Sequential, ReduceVideo_FastFlow e ReduceVideo_Threads. Il primo si occupa di risolvere il problema in maniera sequenziale, mentre i restanti solo le implementazioni parallele, rispettivamente, utilizzando FastFlow e Threads.
Sia V un video e F l'insieme dei suoi frame. Supponiamo di avere accesso ad ogni singolo frame, e dunque ogni singola matrice $f \in F$. Denotiamo con $f_{i,k}^{ray \times ray}$ la k-esima sottomatrice del frame f_i di dimensioni $ray \times ray$ dove ray è il numero delle righe e delle colonne delle sottomatrici.
Il problema del Reducing Video Size consiste nel modificare ogni frame f_i nelle sue dimensioni sulla base di ray, quindi a partire dal frame corrente f_i di dimensioni $[m \times n]$, si crea un nuovo frame g_i di dimensioni $[m/ray \times n/ray]$.

2.1 Sequential

L'algoritmo di risoluzione sequenziale opera in questo modo:

- **input:** V Video.avi
- Tenta di aprire il video V in input e ne crea uno nuovo W
- se l'esito è positivo, allora
  - $\forall f_i$ crea un nuovo frame g_i di dimensioni ridotte proporzionalmente al ray
  - $\forall$ elemento i, j del nuovo frame, crea una sottomatrice del frame originale a partire dalla posizione i, j e grande $ray \times ray$, e applica la $Cv :: resize$
  - scrive il frame g_i nel video W
- **Output:** W NewVideo.avi

L'algoritmo è riportato in pseudo-code nel Codice 1.

Codice 1: Pseudo-code Sequential

```
input: Video V, int ray
output: Video W
foreach Frame  f^{m×n} ∈ V
    g = new Frame();
    g.create(f.size()/ray); //g^{m/ray × n/ray}
    for(i=0; i<g.rows; g++)
        for(j=0; j<g.cols; j++)
            f_i = submatrix of f //f_i^{ray×ray}
            cv::resize(f_i, g(i,j));
    W.write(g);
```

La `Cv::resize(input,output,output.Size)` applica una resize al frame in input e la scrive sul frame in output. Un esempio d'utilizzo è mostrato nella Figura **??**, in cui il frame di dimensioni $[283 \times 274]$ è ridotto alle dimensioni $[141 \times 137]$, con $ray = 2$.

Si noti come ad ogni iterazione, si ha bisogno di istanziare un nuovo frame, piuttosto che modificare il corrente. Ciò per evitare che sovrascritture creino errori nel frame.

(a) Originale

(b) Ridotta

Fig. 1: Esempio di utilizzo di Cv::resize con ray=2

2.2 ReduceVideo_FastFlow

L'algoritmo sequenziale può essere parallelizzato mediante l'utilizzo del **Modello Farm**: il Video viene aperto e ogni frame viene assegnato ad un worker; ogni worker provvede ad applicare la riduzione delle dimensioni del frame e restituisce un nuovo frame. I nuovi frame vengono scritti nel nuovo video in maniera ordinata mediante l'utilizzo del metodo `all_gather`, il quale si occupa di collezionare i risultati da ogni worker in maniera ordinata: il frame inviato al worker i è quello che viene salvato nel vettore di collezione in posizione i. Il calcolo termina quanto tutti i frame vengono processati. Il modello è rappresentato in figura **??**.

Le scelte implementative più importanti riguardano l'implementazione del Worker e del Collector: nel primo, piuttosto che suddividere la riduzione del frame in una ulteriore farm, o mediante metodi come il ParallelFor, si è preferito lasciare il calcolo sequenziale all'interno del Worker, poichè l'eventuale overhead introdotto non è compensato dai risultati del calcolo parallelo in termini di tempo. Inoltre l'utilizzo di una farm all'interno di ogni worker sarebbe stato vano, poichè, come si vedrà più avanti, sebbene la riduzione del tempo d'esecuzione dei Workers porti ad una miglioria nel

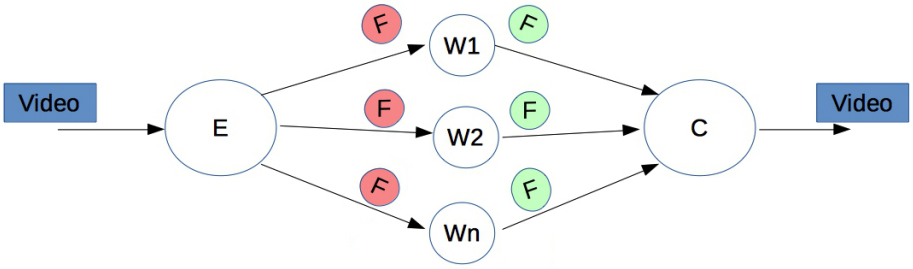


Fig. 2: Rappresentazione grafica del Modello Farm. I cerchi rossi rappresentano i frame da ridurre, quelli verdi i frame risultanti

tempo di esecuzione totale, il tempo totale d'esecuzione è limitato dal tempo di esecuzione del Collector, fortemente legato dalla scrittura su file di ogni singolo frame.
Un'alternativa sarebbe potuta essere rimuovere il collector e lasciare il compito di scrivere i frame risultanti ad ogni worker, ed il tempo di completamento, maggiormente dettato dal tempo di servizio del collector, sarebbe diminuito in proporzione al grado di parallelismo ma la scrittura su video vanificava i risultati per via della perdita di frame.
Per quanto concerne il Collector, il nodo è stato implementato come un normale nodo, ma con la differenza che colleziona i risultati dai Workers mediante il metodo `all_gather`: tale metodo implementa una **Map** tra i Workers e il vettore di collezione, mappando uno-a-uno ogni risultato dei Workers ed il vettore. Tale metodo risulta ottimo in termini di tempo ($\sim 0.05ms$) ma, come si vedrà nei prossimi paragrafi, il tempo di servizio è maggiorato a causa della scrittura dei frame su video.

2.3 ReduceVideo_Threads

L'implementazione parallela è per ovvi motivi differente utilizzando la libreria Thread. Il modello sviluppato è, come per la precedente implementazione, il Modello Farm.
Dapprima si è provveduto a implementare gli *Shared Object*, dei contenitori utilizzati per scambiare informazioni tra i nodi della Farm: tra Emitter e

Worker gli scambi avvengono mediante la *SharedQueue*, una Queue con metodi *push* e *pop* regolati da *mutex* e *condition_variable*; tra Worker e Collector gli scambi avvengono tra una *SharedMap*, costruita per preservare l'ordinamento.
La classe principale è *ReduceVideo*, la quale implementa al suo interno, oltre i classici Emitter, Worker, Collector, i metodi per l'esecuzione dei threads. La classe è rappresentata nell'algoritmo mostrato in Codice 2.

Codice 2: Pseudo-code ReduceVideo_Threads

```
input: Video V, int ray, int numthreads
output: Video W
SharedQueue em-to-wor
SharedMap   wor-to-col
emitter( Video V ){
    foreach Frame  f^{m×n} ∈ V
    em-to-wor.push(f)
}
worker( int ray ){
    while( frame = em-to-wor.pop() )
        //process frame
        wor-to-col.push(frame, pos)
}
collector( Video W ){
    while( frame = wor-to-col.pop() )
        W.write( frame )
}
create and run( int numthreads ){
    thread e( emitter )
    e.join()
    for( i = 0; i < numthreads; i++ )
        thread w( worker ))
        w.join()
    thread c( collector )
    c.join()
}
```

3 Valutazione delle performance

Al fine di valutare le performance di calcolo sono stati definiti modelli analitici, in seguito confrontati con le rilevazioni su una macchina target.
Le rilevazioni son state fatte mediante l'utilizzo del metodo `gettimeofday` incluso nella classe Unix `sys/tyme.h`.
La macchina su cui son stati fatti i test è *titanic.di.unipi.it* con due processori AMD Opteron 6176 a 12 cores. Il compilatore in uso è `GCC 4.9`.
Tutti i test sono realizzati utilizzando il file video `WarIsOver.avi`, contenente 250 frames e incluso nel software.

3.1 Performance

Nella valutazione delle performance si è tenuto conto di diverse misure: il Tempo di Completamento, lo SpeedUp, l'Efficiency e la Scalability.
Come è presumibile immaginare, questo è un problema parallelo sui dati, dunque ci si aspetta che lo SpeedUp cresca proporzionalmente ed il tempo di completamento decresca al crescere del grado di parallelismo. Ci si aspetta dunque uno SpeedUp quasi lineare nel numero di Cores disponibili.
Si mostrano i risultati per le due implementazioni

3.2 FastFlow

Siano T_{seq} e $T_{par}(n)$ rispettivamente il Completion Time dell'applicazione sequenziale e di quella parallela con grado di parallelismo n. L'esecuzione dello script `GetTimesAndPlots.sh` permette di stimare $T_{seq} \simeq 39.221sec$ e $T_{par}(n)$ come riportato nella tabella **??**.

Le performance per lo Speedup, indicato con $sp(n)$, sono riportate nella tabella **??**. Le performance per l'Efficency, indicato con $e(n)$, sono riportate nella tabella **??**. Le performance per la Scalability, indicato con $sc(n)$, sono riportate nella tabella **??**.

Tabella 1: $T_{par}(n)$ FastFlow

n	1	2	3	4	5	6	7	8	9	10
	36.134	18.383	12.425	9.967	9.679	9.518	9.556	9.622	9.537	9.317
11	12	13	14	15	1617	18	19	20		
9.246	9.424	9.255	9.149	9.193	9.144	9.247	9.253	9.240	9.268	
21	22	23	24							
9.137	9.121	11.207	10.955							

Tabella 2: $sp(n)$ FastFlow

n	1	2	3	4	5	6	7	8	9	10
	1.085	2.133	3.156	3.934	4.051	4.120	4.104	4.075	4.112	4.209
11	12	13	14	15	16	17	18	19	20	
4.241	4.161	4.237	4.286	4.266	4.289	4.241	4.238	4.244	4.231	
21	22	23	24							
4.292	4.299	3.499	3.579							

Tabella 3: $e(n)$ FastFlow

n	1	2	3	4	5	6	7	8	9	10
	1.085	1.067	1.052	0.984	0.810	0.687	0.586	0.510	0.457	0.421
11	12	13	14	15	16	17	18	19	20	
0.386	0.347	0.326	0.306	0.284	0.268	0.249	0.235	0.223	0.212	
21	22	23	24							
0.204	0.195	0.152	0.149							

Tabella 4: $sc(n)$ FastFlow

n	1	2	3	4	5	6	7	8	9	10
	1.000	1.965	2.908	3.625	3.733	3.796	3.781	3.755	3.788	3.878
11	12	13	14	15	16	17	18	19	20	
3.907	3.833	3.903	3.949	3.930	3.951	3.907	3.904	3.910	3.898	
21	22	23	24							
3.954	3.961	3.223	3.298							

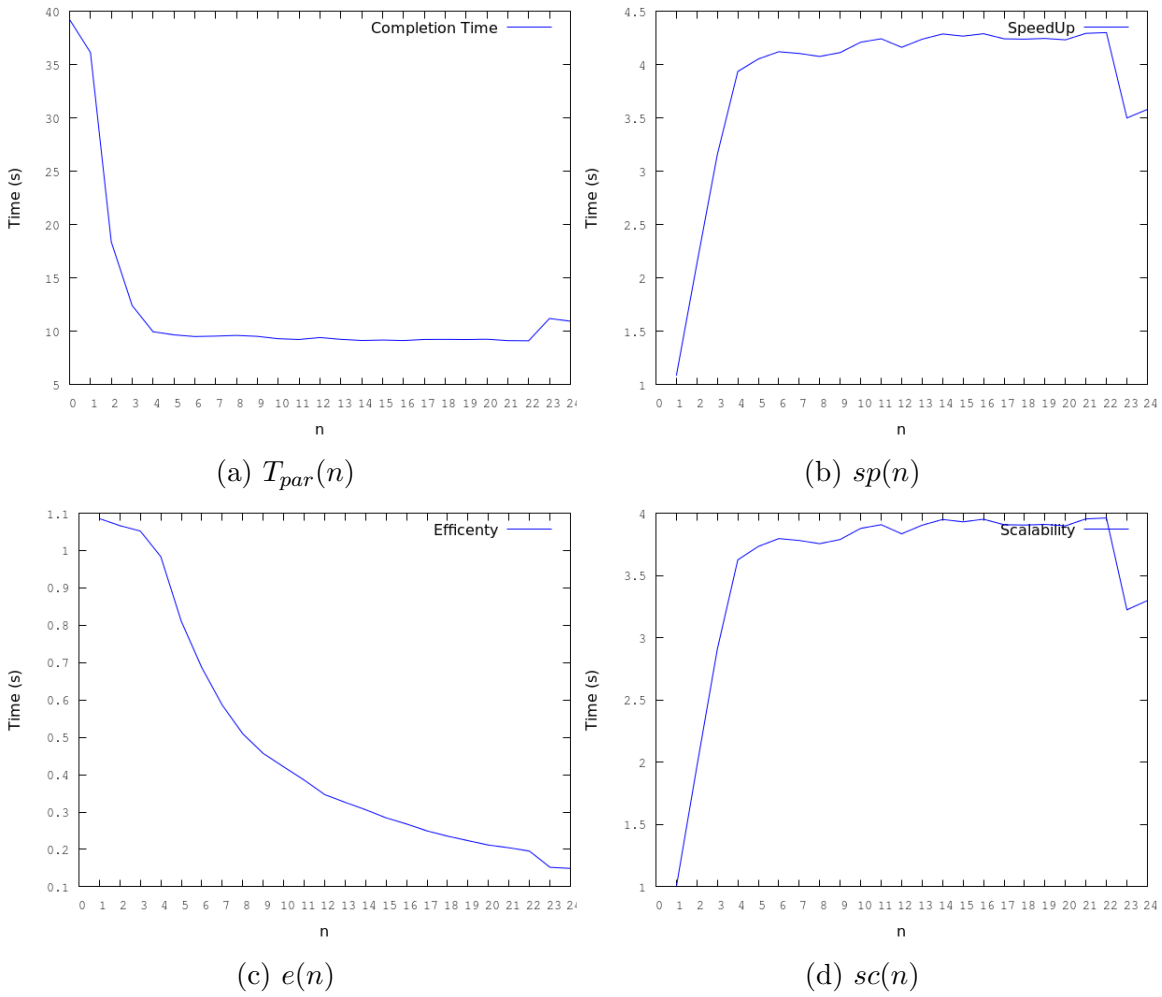
Si riportano, per immediatezza visiva, le precedenti tabelle su grafico nella figura **??**.

Come si può notare anche dai grafici, il Tempo di Completamento decresce vertiginosamente fino ad un grado di parallelismo uguale a 5, questo perchè il Tempo di Completamento è fortemente vincolato al Tempo dei Workers i quali processano i dati e poi inviano al Collector. Dunque bisogna attendere almeno che tutti i Workers abbiano processato tutti i dati, ma sopratutto bisogna attendere che il Collector scriva i risultati sul video, ed il Tempo del Collector, sebbene il metodo `all_gather` è super efficiente, risulta limitato dalla scrittura su video, la quale è molto lenta: 1 frame $\sim 44ms \times 250$, 2 frames $\sim 94ms \times 125$, 10 frames $\sim 432ms \times 25$, 24 frames $\sim 936ms \times 11$, che appunto, quest'ultima, supera i 10sec. Il Tempo dei Workers può dunque decrescere, ma sarà fortemente sovralimitato dal tempo del Collector. Sia lo Speedup, sia l'Efficiency e la Scalability, sono fortemente legati da questi fattori, e i loro grafici lo confermano. In fig.**??** si nota appunto il confronto tra Tempo dei Workers (mediato sul numero degli stessi) e tempo del Collector.

3.3 Threads C++11

L'esecuzione dello script `GetTimesAndPlots.sh` permette di stimare $T_{seq} \simeq 39.221$ e $T_{par}(n)$ come riportato nella tabella **??**.

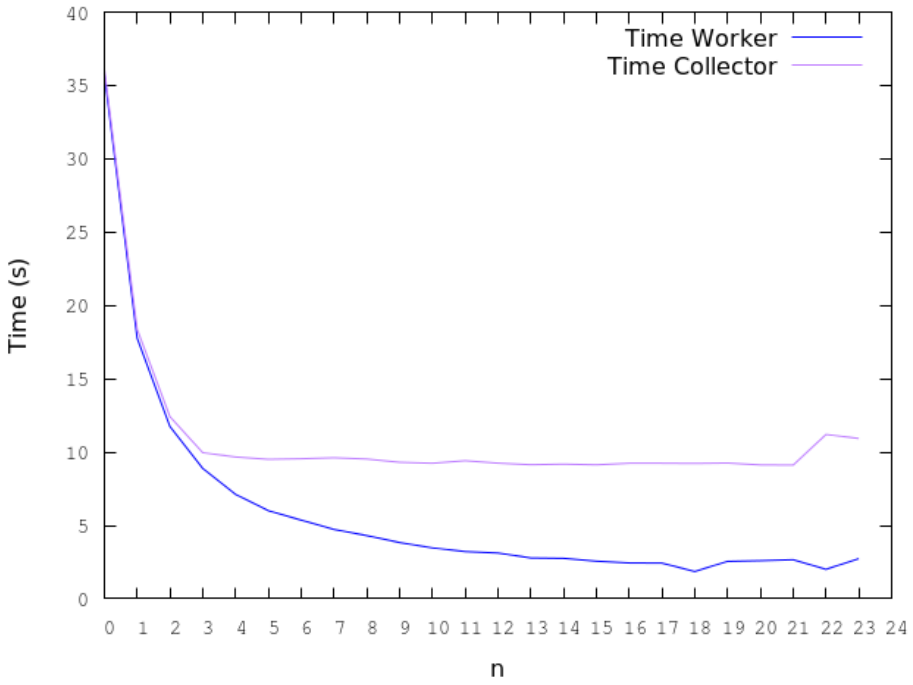
Le performance per lo Speedup, indicato con $sp(n)$, sono ri-



(a) $T_{par}(n)$ (b) $sp(n)$

(c) $e(n)$ (d) $sc(n)$

Fig. 3: Performance implementazione FastFlow



(a)

Fig. 4: Confronto fra Collector Time e Worker Time

Tabella 5: $T_{par}(n)$ Threads

n	1	2	3	4	5	6	7	8	9	10
	39.344	21.327	15.401	12.578	10.478	9.486	8.632	7.710	7.193	6.970
11	12	13	14	15	1617	18	19	20		
6.888	6.219	6.046	5.891	5.813	5.636	5.722	5.435	5.269	5.123	
21	22	23	24							
4.898	4.909	4.771	4.777							

portate nella tabella **??**.

Tabella 6: $sp(n)$ Threads

n	1	2	3	4	5	6	7	8	9	10
	.996	1.838	2.546	3.118	3.743	4.134	4.543	5.087	5.452	5.627
11	12	13	14	15	16	17	18	19	20	
5.693	6.306	6.486	6.657	6.746	6.959	6.8540	7.216	7.442	7.655	
21	22	23	24							
8.006	7.988	8.219	8.209							

Le performance per l'Efficency, indicato con $e(n)$, sono riportate nella tabella **??**.

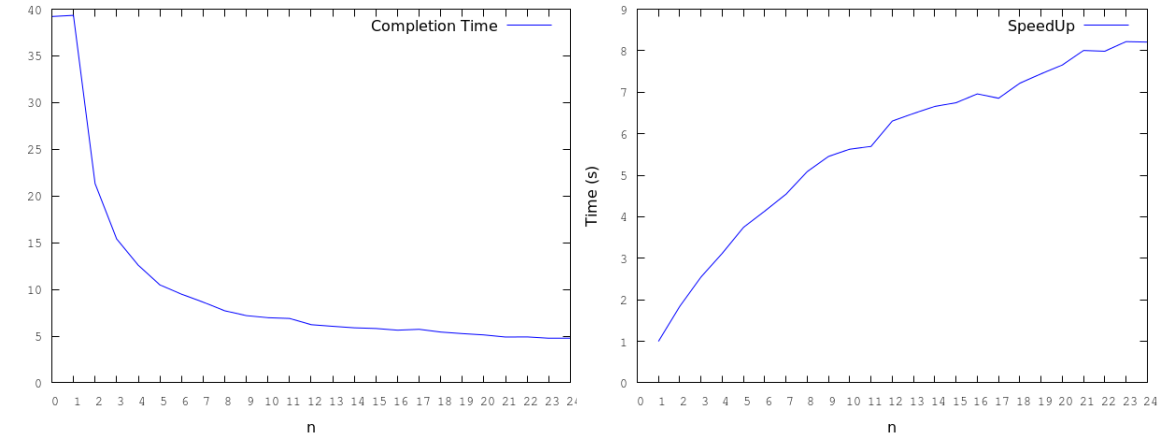
Tabella 7: $e(n)$ Threads

n	1	2	3	4	5	6	7	8	9	10
	.996	.919	.848	.779	.748	.689	.649	.635	.605	.562
11	12	13	14	15	16	17	18	19	20	
.517	.525	.498	.475	.449	.434	.403	.400	.391	.382	
21	22	23	24							
.381	.363	.357	.342							

Le performance per la Scalability, indicato con $sc(n)$, sono riportate nella tabella **??**.

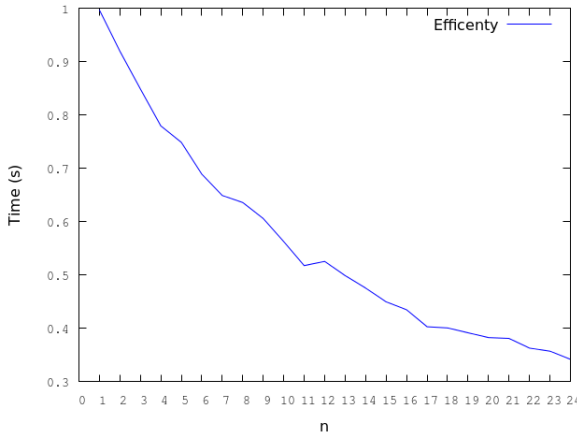
Si riportano, per immediatezza visiva, le precedenti tabelle su grafico nella figura **??**.

Come si può notare dai grafici, lo Speedup cresce in maniera lineare al grado di parallelismo, così come l'efficienza e la scalabilità, il che fa presumere che l'implementazione con Threads di C++11 raggiunge prestazioni migliori.

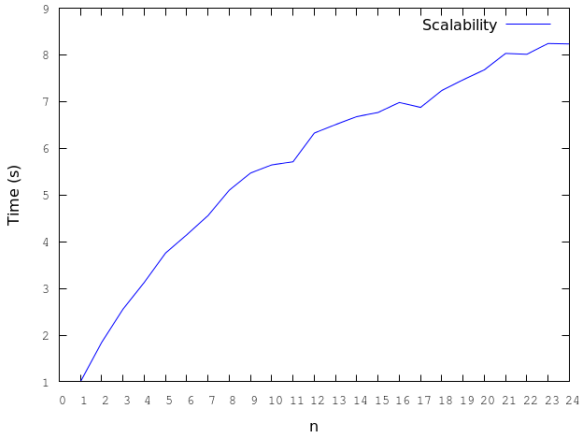


(a) $T_{par}(n)$

(b) $sp(n)$



(c) $e(n)$



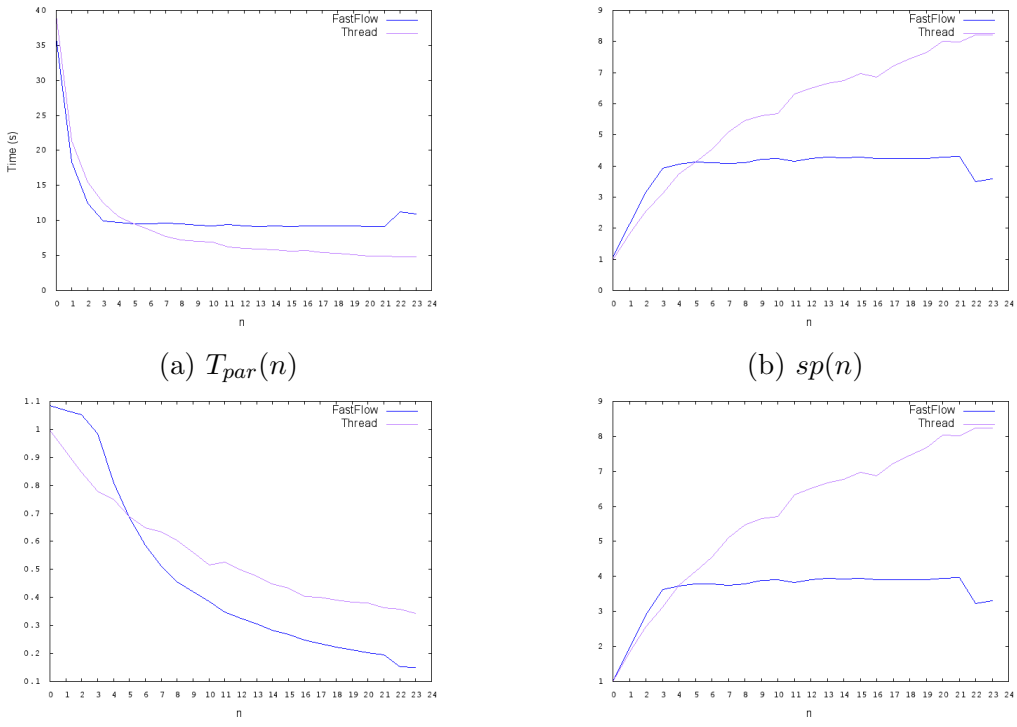
(d) $sc(n)$

Fig. 5: Performance implementazione Threads C++11

Tabella 8: $sc(n)$ Threads

n	1	2	3	4	5	6	7	8	9	10
	1.000	1.844	2.554	3.127	3.754	4.147	4.557	5.103	5.469	5.644
11	12	13	14	15	16	17	18	19	20	
5.711	6.326	6.507	6.678	6.767	6.980	6.875	7.238	7.466	7.679	
21	22	23	24							
8.031	8.013	8.244	8.235							

Si riportano, per competezza, i grafici di entrambe le implementazioni comparate in figura **??**.



(a) $T_{par}(n)$ (b) $sp(n)$

(c) $e(n)$ (d) $sc(n)$

Fig. 6: Performance delle due implementazioni

A Manuale d'uso

Una copia del Software è presente nella macchina `titanic.di.unipi.it`, nella directory `/home/idini/SPM/`

A.1 Makefile

Prima di utilizzare il software è necessario compilarlo, recandosi all'interno della directory dove esso è contenuto e da terminale eseguire il comando:

```
$ make
```

Esso compilerà i tre file contenuti all'interno della directory stessa:

Sequential.cpp è il file contenente l'implementazione sequenziale

ReduceVideo_FastFlow.cpp è il file contenente l'implementazione parallela utilizzando la libreria FastFLow

ReduceVideo_Threads.cpp è il file contenente l'implementazione parallela utilizzando la libreria standard Threads C++11

Al fine di compilare il software in maniera efficiente e compatibile è necessario aver sulla propria macchina una versione di GCC uguale o superiore a 4.8. Per visualizzare la versione del compilatore installata sulla propria macchina, eseguire

```
$ gcc --version
```

o

```
$ gcc -dumpversion
```

A.2 Eseguibili

Gli eseguibili risultanti dalla compilazione sono:

Sequential il quale accetta in input 3 parametri: **filevideo.avi ray outputvideo.avi**

ReduceVideo_FastFlow il quale accetta in input 4 parametri: **filevideo.avi ray outputvideo.avi filevideo.avi num_threads**

ReduceVideo_Threads il quale accetta in input 4 parametri: **filevideo.avi ray outputvideo.avi num_threads**

A.3 Script

Sono forniti inoltre degli script in `Bash`:

GetTimesAndPlots.sh compila, se non è stato ancora fatto, il codice elencato precedentemente, calcola il *Tempo di Completamento* per l'implementazione sequenziale e *Tempo di Completamento*, *Scalabilità*, *Efficienza* e *Speedup* per ognuna delle due implementazioni parallele

I plot, dopo l'esecuzione di quest'ultimo, si possono trovare all'interno della directory `plot/`, mentre i tempi si possono trovare all'interno della cartella `tmp/`.

A.4 Licenza

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 allindirizzo: `http://www.gnu.org/copyleft/gpl.html`

Per qualsiasi problema e/o quesito è possibile contattare lo sviluppatore all'indirizzo e-mail maurizio.idini [at] gmail.com