Università degli Studi di Pisa

Facoltà di Scienze Matematiche Fisiche e Naturali
Corso di Laurea Specialistica in Tecnologie Informatiche

Tesi di Laurea

# Supporti alla programmazione Grid-aware - esperienze di allocazione dinamica di programmi ASSIST a componenti

Candidati

Marco Pasquali     Luigi Presti

Relatore

Chiar.mo Prof. Marco Vanneschi

Controrelatore

Prof. Gianluigi Ferrari

Anno Accademico 2003/2004

# Ringraziamenti

Ringraziamo particolarmente le nostre ragazze Concettina e Maria per questi anni gioiosi, le nostre famiglie che ci hanno permesso di arrivare fino a questo traguardo, il professor Vanneschi per la passione trasmessa, Massimo per la disponibilità dimostrata, Laura, Silvia, Pietro, Gianni e tutti gli altri ragazzi del laboratorio di architetture parallele per il loro prezioso aiuto. Ringraziamo gli amici con cui abbiamo condiviso questi anni: Matteo, Alessandro, Daniele, Carlo, Elena e tutti i gen, Totò, Ignazio, Andrea, Veronica, Luigi, Giovanni, Giancarlo e gli altri amici di facoltà. Ringraziamo inoltre i ragazzi di Via Garibaldi 89/A e di Via Fiorentina 47: Fabio, Franz, Massimo, Marco, Pino, Ivan e tutti gli altri ragazzi con cui abbiamo vissuto, per gli indimenticabili momenti trascorsi. Infine ci ringraziamo a vicenda per questi anni universitari vissuti insieme.

*Ai nostri genitori e ai nostri nonni.*

# Indice

**1 Introduzione** **13**

**2 Programmazione su griglia ed ASSIST** **19**

2.1 Lo sviluppo di applicazioni e le griglie computazionali . . . . . . . . . 19

2.1.1 Le applicazioni grid-aware viste come composizione di componenti ad alta performance . . . . . . . . . . . . . . . . . . . . . 21

2.1.2 Approccio uniforme alla programmazione parallela e distribuita 22

2.1.3 Le applicazioni dinamicamente adattive . . . . . . . . . . . . . 22

2.2 Il modello di programmazione di ASSIST . . . . . . . . . . . . . . . . 24

2.2.1 Programmi ad alta performance definiti come grafi . . . . . . 25

2.2.2 Uno skeleton generico: *parmod* . . . . . . . . . . . . . . . . . . 27

2.2.3 Supporto alle applicazioni grid-aware . . . . . . . . . . . . . . 30

2.2.4 Un esempio di applicazione adattiva . . . . . . . . . . . . . . . 30

**3 Il modello di programmazione a componenti** **33**

3.1 Perché si ha bisogno dei componenti? . . . . . . . . . . . . . . . . . . 34

3.2 Le categorie di *Componentware* . . . . . . . . . . . . . . . . . . . . . 35

3.2.1 Approcci Formali . . . . . . . . . . . . . . . . . . . . . . . . . 36

3.2.2 Approcci Descrittivi . . . . . . . . . . . . . . . . . . . . . . . 37

3.2.3 Approcci centrati sull'Architettura . . . . . . . . . . . . . . . . 38

3.2.4 Approcci centrati sul Processo . . . . . . . . . . . . . . . . . . 38

3.2.5 Component Frameworks . . . . . . . . . . . . . . . . . . . . . 39

3.3 Il CORBA Component Model (CCM) . . . . . . . . . . . . . . . . . . . 41

3.3.1 Common Object Request Broker Architecture (CORBA) . . . 41

3.3.2 Introduzione al modello a componenti di CORBA . . . . . . . 44

3.3.3 Il CCM in dettaglio . . . . . . . . . . . . . . . . . . . . . . . . 45

3.4 I component framework ad alta performance . . . . . . . . . . . . . 53

3.4.1 Lo standard CCA: XCat . . . . . . . . . . . . . . . . . . . . . 53

3.4.2 Lo standard CCA: CCAffeine . . . . . . . . . . . . . . . . . . 58

3.4.3 Grid-Web Service . . . . . . . . . . . . . . . . . . . . . . . . . 59

3.4.4 GridCCM . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 61

3.4.5 Conclusioni . . . . . . . . . . . . . . . . . . . . . . . . . . . . 63

**4 Integrazione ASSIST-CCM 65**

4.1 Scopo dell'integrazione . . . . . . . . . . . . . . . . . . . . . . . . . 65

4.2 Integrazione tra ASSIST e CORBA . . . . . . . . . . . . . . . . . . 67

4.2.1 Architettura del meccanismo di invocazione sincrona dei metodi di un server . . . . . . . . . . . . . . . . . . . . . . . . 68

4.2.2 Architettura del meccanismo ad eventi . . . . . . . . . . . . . 69

4.3 Integrazione tra ASSIST e CCM . . . . . . . . . . . . . . . . . . . . 71

4.3.1 La connessione lato CCM . . . . . . . . . . . . . . . . . . . . 73

4.3.2 La connessione lato ASSIST . . . . . . . . . . . . . . . . . . . 74

4.3.3 La comunicazione tra CCM ed ASSIST mediante CDR . . . . 76

4.3.4 Le interfacce non funzionali del componente CCM . . . . . . . 77

4.3.5 Chiamate RMI nel componente CCM . . . . . . . . . . . . . . 79

4.3.6 Integrazione di un'intera applicazione ASSIST . . . . . . . . . 80

**5 Metodologia proposta e sperimentazioni 83**

5.1 L'insieme di Mandelbrot . . . . . . . . . . . . . . . . . . . . . . . . 87

5.2 I compressori Mpeg . . . . . . . . . . . . . . . . . . . . . . . . . . . 88

5.3 L'applicazione in dettaglio . . . . . . . . . . . . . . . . . . . . . . . . . 90
5.3.1 Modulo Mandelbrot . . . . . . . . . . . . . . . . . . . . . . . . 93
5.3.2 Modulo PixConverter . . . . . . . . . . . . . . . . . . . . . . . 94
5.3.3 Modulo MpegEncoder . . . . . . . . . . . . . . . . . . . . . . . 95
5.3.4 Modulo StreamStore . . . . . . . . . . . . . . . . . . . . . . . 96
5.3.5 Modulo ApplicationManager . . . . . . . . . . . . . . . . . . . 97
5.4 Ambienti di prova . . . . . . . . . . . . . . . . . . . . . . . . . . . . 99
5.5 Esperimenti preliminari . . . . . . . . . . . . . . . . . . . . . . . . . 100
5.5.1 Studio dell'applicazione: prima versione . . . . . . . . . . . . 101
5.5.2 Studio dell'applicazione: seconda versione . . . . . . . . . . . 110
5.6 Esperimenti sulla versione finale dell'applicazione . . . . . . . . . . . 117
5.6.1 Prestazioni sul cluster omogeneo Pianosa . . . . . . . . . . . . 118
5.6.2 Prestazioni sui cluster eterogenei Pianosa e C1 . . . . . . . . . 128
5.6.3 Prestazioni sulla griglia . . . . . . . . . . . . . . . . . . . . . . 135
5.6.4 Conclusioni . . . . . . . . . . . . . . . . . . . . . . . . . . . . 142

**6 Conclusioni e proseguimento del lavoro 145**

**Bibliografia 149**

**A Codice sorgente 157**

**B Grafici delle sperimentazioni 183**
B.1 Prove sul cluster Pianosa . . . . . . . . . . . . . . . . . . . . . . . . 186
B.2 Prove sui cluster Pianosa e C1 . . . . . . . . . . . . . . . . . . . . . . 216
B.3 Prove sulla griglia . . . . . . . . . . . . . . . . . . . . . . . . . . . . 239

# Elenco delle figure

2.1 Un esempio di applicazione ASSIST definita come un grafo. . . . . . 26
2.2 Elementi di un modulo parallelo (parmod). . . . . . . . . . . . . . . . 28
2.3 Esempio di una applicazione adattiva costituita da componenti paralleli. 31

3.1 Categorie di *Componentware*. . . . . . . . . . . . . . . . . . . . . . . . 35
3.2 Trasmissione della richiesta di un client all'implementazione dell'oggetto CORBA (server). . . . . . . . . . . . . . . . . . . . . . . . . . . . 44
3.3 Descrizione di un componente CCM. . . . . . . . . . . . . . . . . . . . 46
3.4 *Abstract Component Model*: nuove interfacce introdotte. . . . . . . . 48
3.5 Descrizione dell'interfaccia *Home*. . . . . . . . . . . . . . . . . . . . . 49
3.6 Architettura definita dal *Container Programming Model*. . . . . . . . 51
3.7 Modello di programmazione XCat che usa un programma di controllo Java. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57
3.8 Modello di programmazione XCat che usa un programma di controllo Jython. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57
3.9 Componente parallelo introdotto nel GridCCM. . . . . . . . . . . . . 62
3.10 Il livello GridCCM intercetta e traduce l'invocazione remota di un metodo. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 63

4.1 Grafo dell'invocazione sincrona di un metodo ASSIST. . . . . . . . . 68
4.2 Grafo del meccanismo di comunicazione ad eventi. . . . . . . . . . . . 70
4.3 Architettura dell'integrazione tra ASSIST e CCM. . . . . . . . . . . . 73
4.4 Applicazione CCM *Client-Server* mediante chiamate RMI. . . . . . . 79

4.5 Rappresentazione di un modulo ASSIST incapsulato in un componente CCM. . . . 81

5.1 Immagine dell'insieme di Mandelbrot. . . . 87

5.2 Architettura dell'applicazione. . . . 92

5.3 Architettura del modulo Mandelbrot. . . . 94

5.4 Architettura del modulo MpegEncoder. . . . 96

5.5 Architettura della prima versione dell'applicazione. . . . 102

5.6 Banda del modulo Mandelbrot. . . . 104

5.7 Tempo di esecuzione della prima versione dell'applicazione. . . . 107

5.8 Banda di esecuzione del modulo Mandelbrot nella prima versione dell'applicazione. . . . 108

5.9 Tempo di esecuzione della seconda versione dell'applicazione. . . . 112

5.10 Banda di esecuzione del modulo Mandelbrot nella seconda versione dell'applicazione. . . . 114

5.11 Tempi relativi all'esecuzione dell'applicazione ASSIST su Pianosa: collo di bottiglia in Mandelbrot. . . . 119

5.12 Tempi relativi all'esecuzione dell'applicazione ASSIST su Pianosa: collo di bottiglia in MpegEncoder. . . . 120

5.13 Tempi relativi all'esecuzione dell'applicazione ASSIST su Pianosa: dinamicità applicazione. . . . 120

5.14 Tempi relativi all'esecuzione dell'applicazione ASSIST integrata con il CCM su Pianosa: collo di bottiglia in Mandelbrot. . . . 122

5.15 Tempi relativi all'esecuzione dell'applicazione ASSIST integrata con il CCM su Pianosa: collo di bottiglia in MpegEncoder (1). . . . 122

5.16 Tempi relativi all'esecuzione dell'applicazione ASSIST integrata con il CCM su Pianosa: collo di bottiglia in MpegEncoder (2). . . . 123

5.17 Bande dell'intera applicazione su Pianosa nelle sue diverse configurazioni. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 124

5.18 Bande dell'intera applicazione integrata con il CCM su Pianosa nelle sue diverse configurazioni. . . . . . . . . . . . . . . . . . . . . . . . . 124

5.19 Tempi relativi all'esecuzione dell'applicazione ASSIST su Pianosa: adozione della politica di adattività. . . . . . . . . . . . . . . . . . . . . 126

5.20 Tempi relativi all'esecuzione dell'applicazione integrata CCM su Pianosa: adozione della politica di adattività. . . . . . . . . . . . . . . . 127

5.21 Tempi relativi all'esecuzione dell'applicazione ASSIST su Pianosa e C1: Mandelbrot con grado di parallelismo 4. . . . . . . . . . . . . . . 129

5.22 Tempi relativi all'esecuzione dell'applicazione ASSIST integrata con il CCM su Pianosa e C1: Mandelbrot con grado di parallelismo 4. . . 129

5.23 Tempi relativi all'esecuzione dell'applicazione ASSIST su Pianosa e C1: Mandelbrot con grado di parallelismo 16. . . . . . . . . . . . . . 130

5.24 Tempi relativi all'esecuzione dell'applicazione ASSIST integrata con il CCM su Pianosa e C1: Mandelbrot con grado di parallelismo 16. . 130

5.25 Bande dell'intera applicazione su Pianosa e C1 nelle sue diverse configurazioni. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 131

5.26 Bande dell'intera applicazione integrata con il CCM su Pianosa e C1 nelle sue diverse configurazioni. . . . . . . . . . . . . . . . . . . . . . . 132

5.27 Tempi relativi all'esecuzione dell'applicazione ASSIST su Pianosa e C1: adozione della politica di adattività. . . . . . . . . . . . . . . . . 133

5.28 Tempi relativi all'esecuzione dell'applicazione integrata CCM su Pianosa e C1: adozione della politica di adattività. . . . . . . . . . . . 134

5.29 Allocazione dei moduli dell'applicazione su griglia. . . . . . . . . . . 136

5.30 Tempi relativi all'esecuzione dell'applicazione ASSIST su griglia: Mandelbrot con grado di parallelismo 4. . . . . . . . . . . . . . . . . . . . 137

5.31 Tempi relativi all'esecuzione dell'applicazione ASSIST integrata con il CCM su griglia: Mandelbrot con grado di parallelismo 4. . . . . . . 138

5.32 Tempi relativi all'esecuzione dell'applicazione ASSIST su griglia: Mandelbrot con grado di parallelismo 8. . . . . . . . . . . . . . . . . . . . . 138

5.33 Tempi relativi all'esecuzione dell'applicazione ASSIST integrata con il CCM su griglia: Mandelbrot con grado di parallelismo 8. . . . . . . 139

5.34 Tempi relativi all'esecuzione dell'applicazione ASSIST su griglia: Mandelbrot con grado di parallelismo 16. . . . . . . . . . . . . . . . . . . . 139

5.35 Tempi relativi all'esecuzione dell'applicazione ASSIST integrata con il CCM su griglia: Mandelbrot con grado di parallelismo 16. . . . . . 140

5.36 Bande dell'intera applicazione integrata con il CCM e non, su griglia, nelle sue diverse configurazioni. . . . . . . . . . . . . . . . . . . . . . . 141

# Elenco delle tabelle

3.1 Associazioni tra applicazione e tipo di *Container*. . . . . . . . . . . . 52

5.1 Banda di Mandelbrot - Intervallo di appartenenza dell'indice $n$ [0-1023]. 103

5.2 Banda di Mandelbrot - Intervallo di appartenenza dell'indice $n$ [0-511]. 104

5.3 Banda di Mandelbrot - Intervallo di appartenenza dell'indice $n$ [0-255]. 104

5.4 Banda di Mandelbrot - Intervallo di appartenenza dell'indice $n$ [0-127]. 104

5.5 Banda di MpegEncoder. . . . . . . . . . . . . . . . . . . . . . . . . . 105

5.6 Prestazioni della prima versione dell'applicazione. . . . . . . . . . . . 106

5.7 Prestazioni del modulo Mandelbrot nella prima versione dell'applicazione. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 108

5.8 Prestazioni della prima versione dell'applicazione integrata con il CCM. 109

5.9 Prestazioni del modulo Mandelbrot nella prima versione dell'applicazione integrata con il CCM. . . . . . . . . . . . . . . . . . . . . . . . 109

5.10 Prestazioni della seconda versione dell'applicazione. . . . . . . . . . . 111

5.11 Prestazioni del modulo Mandelbrot nella seconda versione dell'applicazione. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 113

5.12 Primo Test con 4 fade tra un'immagine e l'altra: scalabilità ed efficienza dell'applicazione. . . . . . . . . . . . . . . . . . . . . . . . . 113

5.13 Secondo Test con 10 fade tra un'immagine e l'altra: scalabilità ed efficienza dell'applicazione. . . . . . . . . . . . . . . . . . . . . . . . . 115

5.14 Prestazioni della seconda versione dell'applicazione integrata con il CCM. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 116

5.15 Prestazioni del modulo Mandelbrot nella seconda versione dell'applicazione integrata con il CCM. . . . . . . . . . . . . . . . . . . . . . . . . 116
5.16 Bande ottenute dall'applicazione su Pianosa con Mandelbrot a grado di parallelismo 8 con e senza politica adattiva. . . . . . . . . . . . . . 126
5.17 Bande ottenute dall'applicazione integrata CCM su Pianosa con Mandelbrot a grado di parallelismo 6 con e senza politica adattiva. . . . . 127
5.18 Bande ottenute dall'applicazione su Pianosa e C1 con Mandelbrot a grado di parallelismo 8 con e senza politica adattiva. . . . . . . . . . 134
5.19 Bande ottenute dall'applicazione integrata CCM su Pianosa e C1 con Mandelbrot a grado di parallelismo 8 con e senza politica adattiva. . 135

# Capitolo 1

# Introduzione

Il calcolo parallelo tradizionale è riuscito a dimostrare il suo valore in ogni campo dell'attività umana. Infatti i cluster omogenei sono ormai utilizzati nella modellistica e nella simulazione scientifica complessa, nella diagnosi di circostanze mediche, nelle previsioni del tempo e in molti altri campi. Tuttavia in diverse applicazioni scientifiche o in sistemi di supporto alle decisioni ad alta performance, l'ambiente di calcolo rimane inadeguato, dal punto di vista delle prestazioni, rendendo necessaria quindi una maggiore potenza di calcolo che vada oltre quelle di un cluster locale o di un *supercomputer*. Questo tipo di applicazioni, quindi, necessita di una collaborazione dinamica tra risorse appartenenti ad organizzazioni diverse e distribuite geograficamente, in modo da ottenere maggiori capacità di calcolo.

Le *griglie computazionali* [26, 29, 40] (*a hardware and software infrastructure that provides dependable, consistent, pervasive, and inexpensive access to high-end computational capabilities*, Foster e Kesselman 1998), da questo punto di vista hanno introdotto una reale inversione di tendenza rispetto ai sistemi paralleli tradizionali. Grazie, infatti, all'evoluzione della tecnologia di rete WAN e alla sempre più ricca disponibilità di risorse commodity, in linea di principio, permettono alle applicazioni ad alta performance di avere a disposizione non solo un semplice cluster parallelo, ma una piattaforma costituita da un insieme dinamico ed eterogeneo di risorse computazionali e di memorizzazione, geograficamente distribuito e con un ambiente di esecuzione altamente scalabile.

La griglia però, per le sue peculiarità, ha introdotto delle problematiche marginalmente presenti in un ambiente locale ed omogeneo (cluster di PC), che hanno stimolato diverse questioni dal punto di vista della ricerca. Le applicazioni su griglia (*grid-aware applications*), per esempio, devono poter gestire l'eterogeneità e la dinamicità della piattaforma sottostante, in modo da poter garantire uno specificato livello di performance, nonostante la varietà di eventi a run-time che possono modificare la disponibilità delle risorse (sbilanciamento del carico, fallimenti della rete o dei nodi, problemi amministrativi ed altro). Inoltre per questo tipo di applicazioni si ha la necessità di *tool* ed ambienti di sviluppo che garantiscano una programmabilità ad alto livello, fornendo al programmatore primitive capaci di strutturare il parallelismo nell'applicazione, che permettano di raggiungere una piena interoperabilità con il software esistente e che supportino il riutilizzo di codice già sviluppato in altre applicazioni. Da questo punto di vista si ha quindi bisogno, per le applicazioni grid-aware, di un ambiente di programmazione *component-based* [3], in cui si riescano a sviluppare le applicazioni come composizione di componenti ad alta performance, magari già esistenti, e che riesca ad adattare l'applicazione alla dinamicità della piattaforma, in modo da garantire uno specificato livello di performance. Infatti, i componenti ad alta performance sono caratterizzati da un modello di programmazione basato su diversi concetti: parallelismo strutturato, composizione e interazione tra componenti, adattività dell'applicazione.

Vi sono diversi progetti in letteratura orientati allo sviluppo di ambienti di programmazione component-based ad alta performance [13] come possibile approccio per lo sviluppo di applicazioni grid-aware. L'architettura a componenti *CCA* [38, 5] (*Common Component Architecture*), per esempio, è stata sviluppata come progetto attraverso la collaborazione di numerosi laboratori di ricerca ed università, con lo scopo di definire un'architettura comune e standard orientata allo sviluppo di applicazioni scientifiche a larga scala a partire da componenti software ben testati. A partire da questo progetto sono state realizzate delle implementazioni dello standard CCA (vedi sez. 3.4), tra le quali spiccano *XCat* [31] e *CCAffeine* [39], orientate al-

lo sviluppo di applicazioni su griglia. XCat, per esempio, utilizza il modello delle porte *uses/provides*, definito dallo standard CCA per comporre i componenti tra loro, che permette di modellare in modo naturale e semplificato applicazioni anche di una certa complessità. Fornisce anche al programmatore una serie di servizi ad alto livello per il controllo dell'applicazione. Vi sono però una serie di problemi, tra i quali la possibilità di comporre applicazioni utilizzando solamente componenti CCA scritti in Java, che non lo rendono adatto alla griglia. CCAffeine, invece, supera questo problema di interoperabilità, anche se affida la gestione dell'alta performance all'interno di un componente, e quindi le comunicazioni, alla responsabilità del programmatore, che può implementarle attraverso delle librerie di sua scelta (MPI, PVM, RPC, memoria condivisa). Altri progetti significativi in questo campo della ricerca sono *GridCCM* [21, 50], *Grid Service* [28, 27, 25], che vengono analizzati nella sezione 3.4. Dai risultati di tale analisi si evince però che ognuno di questi progetti è incentrato su un particolare aspetto delle problematiche della programmazione grid-aware, quindi non risulta adatto per lo sviluppo di applicazioni grid-aware in senso più generale.

In Italia è attivo il progetto di ricerca di base MIUR-FIRB *Grid.it*: *Enabling Platforms for High-performance Computational Grids orientated to Scalable Virtual Organizations*; in particolare questa tesi utilizza alcuni degli strumenti e dei risultati del *Workpackage WP8* (Ambienti di Programmazione ad Alta Performance basati su Componenti) e si collega alla sua problematica di ricerca. La strada intrapresa dal gruppo, per la definizione di un ambiente di sviluppo per applicazioni grid-aware, è diversa rispetto a quella seguita dai progetti esistenti [3], poiché prende in considerazione tutte le problematiche delle applicazioni grid-aware. Rispetto al progetto GridCCM per esempio, si focalizza maggiormente sull'adattività e sul *grid-awareness* dell'applicazione più che sull'ottimizzazione delle comunicazioni. Inoltre, rispetto ad altri progetti, come GrADs [18], si differenzia per il modello di programmazione proposto, la programmazione parallela strutturata [16]. Dal punto di vista della composizione dei componenti invece si avvicina a quella formulata dal progetto CCAffeine.

L'obiettivo del gruppo è di far evolvere *ASSIST* [37, 4, 57, 2, 49, 15], l'ambiente di programmazione orientato allo sviluppo di applicazioni parallele e distribuite ad alta performance sviluppato dal gruppo di Architetture Parallele del Dipartimento di Informatica dell'Università di Pisa, in un ambiente component-based, che supporti le peculiarità della griglia elencate precedentemente. Come primo passo si è deciso di sperimentare l'integrazione di ASSIST con un modello a componenti (non ad alta performance) esistente [3], attraverso l'incapsulamento di moduli paralleli ASSIST in componenti, valutarne l'impatto e comprendere quali siano i meccanismi necessari da includere nel modello a componenti per implementare applicazioni grid-aware ad alta performance. Ad esempio, potrebbe essere necessaria nel modello una composizione più esplicita dei componenti che enfatizzi l'adattività a run-time della struttura e lo sfruttamento di questa struttura gerarchica per gestire la riconfigurazione dell'applicazione. Alla fine, attraverso i risultati della sperimentazione, si potrebbe arrivare ad una integrazione del modello a componenti che possa far coesistere componenti ASSIST paralleli e componenti già esistenti, in una applicazione parallela e distribuita su griglia ad alta performance.

Proprio in questa direzione si è articolato il lavoro della tesi, in cui viene proposta e sperimentata una metodologia per il supporto di applicazioni grid-aware espresse come moduli paralleli ASSIST *puri* e incapsulati in componenti *CCM* [35, 54] (*CORBA Component Model*). L'utilizzo del CCM (vedi sez. 3.2.5), rispetto ad altri *Component Framework* non ad alta performance esistenti (e.g. *COM/DCOM/COM+* [41] della *Microsoft*, *EJB* [46] della *Sun*), viene giustificato dal fatto che è uno standard ben affermato e svincolato da uno specifico produttore, inoltre fornisce la nozione di componente più completa e sofisticata tra i framework considerati [6]. La metodologia oggetto della tesi prevede il monitoraggio del tempo di servizio di applicazioni, basate su *stream* di dati, e la riconfigurazione da parte di un *Application Manager*, che può autonomamente decidere quali moduli/componenti paralleli utilizzare per bilanciare il carico, rispettando quindi il contratto di performance o evitando uno spreco inutile di risorse. Per sperimentare quanto detto è stata svi-

luppata un'applicazione costituita da diversi moduli paralleli, che successivamente verranno integrati in componenti CCM, uno dei quali ha due implementazioni diverse, in esecuzione su nodi della griglia differenti. L'Application Manager sceglierà quale implementazione utilizzare di questo modulo in base ai tempi registrati e raccolti da altri moduli.

A conclusione della sperimentazione, verrà fatto un confronto tra i due approcci proposti (ASSIST puro ed integrato CCM), valutando l'impatto dei componenti CCM in una applicazione in esecuzione su una griglia, e valutando la bontà dell'Application Manager nei due casi.

La tesi è organizzata nel seguente modo:

**Capitolo 2** vengono introdotte le caratteristiche della programmazione su griglia, i problemi ad essi correlati e un primo accenno agli strumenti per risolverli. Viene presentato inoltre l'ambiente di programmazione parallela ASSIST e le sue peculiarità per lo sviluppo di applicazioni grid-aware.

**Capitolo 3** vengono studiate le caratteristiche della programmazione a componenti, in risposta ai problemi delle applicazioni grid-aware, i progetti esistenti ad alta performance e non, e nello specifico il modello a componenti di CORBA (CCM);

**Capitolo 4** vengono discussi gli scopi e i problemi dell'integrazione di moduli ASSIST in componenti CCM e gli strumenti utilizzati per risolvere tali problemi;

**Capitolo 5** viene descritta l'intera applicazione sviluppata e la metodologia seguita negli esperimenti. Vengono enunciati e commentati infine i risultati raggiunti, con particolare enfasi sull'eventuale *overhead* nel calcolo e nelle comunicazioni introdotto dai componenti CCM, e sulla bontà dell'Application Manager nel rimuovere i colli di bottiglia ed evitare lo spreco di risorse in una applicazione grid-aware;

**Capitolo 6** vengono riportate le conclusioni del lavoro e i problemi aperti da affrontare.

# Capitolo 2

# Programmazione su griglia ed ASSIST

Sommario

In questo capitolo verranno elencate le caratteristiche essenziali di un ambiente di programmazione orientato allo sviluppo di applicazioni grid-aware e ad alta performance, e un primo accenno ai progetti che vanno in questa direzione. Queste caratteristiche serviranno per introdurre l'utilizzo di ASSIST come punto di partenza per la costruzione di un ambiente di programmazione component-based per applicazioni grid-aware. Di ASSIST verranno anche descritte le funzionalità essenziali, che verranno contestualizzate al lavoro svolto nella tesi.

## 2.1 Lo sviluppo di applicazioni e le griglie computazionali

Le griglie computazionali [26, 29, 40] si possono vedere come delle architetture complesse e distribuite su scala geografica, i cui nodi sono macchine parallele di qualsiasi genere (e.g. cluster di PC/workstation). Queste piattaforme sono caratterizzate dall'eterogeneità dei nodi e dalla dinamicità nella gestione ed allocazione delle risorse. Le applicazioni *grid-aware*, quindi, si devono occupare effettivamente di questa eterogeneità e dinamicità (applicazioni *adattive*), in modo da garantire uno *specificato* livello di performance, nonostante la grande varietà di eventi che possono verificarsi

a run-time (sbilanciamento del carico, fallimenti nella rete e nei nodi, problemi amministrativi, emergenze). Diventa molto importante, quindi, avere degli ambienti e dei tool di sviluppo delle applicazioni ad alta performance che garantiscano una programmabilità ad alto livello, riutilizzo ed interoperabilità nel software, ed allo stesso tempo riescano a raggiungere un'alta performance, a prescindere dall'evoluzione della tecnologia sottostante (reti, nodi, cluster, sistemi operativi).

A tutt'oggi le applicazioni su griglia sono spesso progettate secondo un approccio a basso livello, relegando ai servizi *middleware* la gestione e lo scheduling delle risorse, e, in molti casi, esse consistono di singoli job o limitate forme di composizione di job (DAG). Il parallelismo, quando presente, è limitato dentro i singoli job, in modo tale che non abbia alcun effetto sulla struttura esterna del programma (un job potrebbe essere ad esempio un programma MPI). Raramente quindi le applicazioni eseguibili su griglia sono grid-aware ed ad alta performance seconda la nostra definizione.

Ciò di cui necessitano le applicazioni grid-aware è invece un modello di programmazione con le seguenti caratteristiche [4]:

1. le applicazioni sono espresse come composizione di componenti ad alta performance;

2. un approccio uniforme viene seguito per la programmazione parallela e distribuita: in generale i componenti possono avere un loro parallelismo interno e essere eseguiti in parallelo l'uno con l'altro;

3. la strategia che guida l'adattamento dinamico dell'applicazione grid-aware deve essere espressa nello stesso formalismo ad alto livello del modello di programmazione.

### 2.1.1 Le applicazioni grid-aware viste come composizione di componenti ad alta performance

I componenti sono il meccanismo base per raggiungere la composizionalità, garantendo allo stesso tempo interoperabilità e riutilizzo del software. La fusione della programmazione ad alta performance e della tecnologia a componenti, deve permettere al progettista di strutturare l'applicazione come composizione di componenti (ad alta performance), alcuni dei quali magari già esistenti, ed altri sviluppati da zero. Sfruttando questo tipo di composizione si possono generare nuovi *pattern* paralleli più adatti alle esigenze degli utenti. Come accennato nel capitolo 1, esistono diversi progetti in letteratura che affrontano il problema della composizione di componenti ad alta performance in applicazioni grid-aware (XCat [31], CCAffeine [39], GridCCM [21], Grid-Web Service [28]). Ognuno di essi però, per diversi motivi, non risulta adatto alla griglia computazionale, poiché focalizza la sua ricerca solo su particolari aspetti della griglia, lasciando invece scoperti altri problemi. Infatti, per esempio, GridCCM affronta il problema dell'ottimizzazione delle comunicazioni ma non quello dell'adattività dell'applicazione in un ambiente dinamico come quello della griglia. CCAffeine, invece, affronta il problema dell'interoperabilità dei componenti, che quindi possono essere scritti in diversi linguaggi di programmazione, ma lascia la responsabilità dell'alta performance al programmatore. Questi ed altri progetti verranno descritti nello specifico nel capitolo 3.

Come si vedrà nella sezione 2.2, l'ambiente di programmazione ASSIST [3], invece, fornisce l'astrazione necessaria per i componenti ad alta performance e per la loro composizione, pur continuando a mantenere le sue caratteristiche orientate allo sviluppo di applicazioni grid-aware. La strada seguita per trasformare moduli paralleli e programmi ASSIST in componenti ad alta performance segue diverse fasi. La prima fase prevede di sperimentare l'integrazione di ASSIST con un modello a componenti (*component framework*) non ad alta performance già esistente, attraverso l'incapsulamento di moduli paralleli ASSIST in componenti, valutarne l'impatto

e comprendere quali siano i meccanismi necessari da includere nel modello a componenti per applicazioni ad alta performance. In questa tesi il modello a componenti (base) scelto per l'integrazione è il CCM [35] (*Corba Component Model*) (vedi sez. 3.3), poiché rispetto ad altri modelli esistenti fornisce una più completa e sofisticata nozione di componente.

### 2.1.2 Approccio uniforme alla programmazione parallela e distribuita

Vi sono diversi motivi in supporto alla visione uniforme della programmazione parallela e distribuita:

- diverse applicazioni si possono notevolmente avvantaggiare dal parallelismo all'interno del nodo (*inter-nodo*), ma allo stesso tempo potrebbero guadagnare una maggiore performance ad un livello esterno al nodo (*intra-nodo*). Vista quindi la natura eterogenea e dinamica delle griglie computazionali, una distinzione a priori tra parallelismo inter ed intra nodo potrebbe essere difficile, e quindi, forzandola in qualche modo, potrebbe causare una sensibile degradazione della performance. Da ciò segue che la distinzione tra parallelismo inter ed intra nodo deve essere delegata agli strumenti di programmazione, sia a *compile-time* che a *run-time*;
- come conseguenza, un approccio che non limita le opportunità di parallelismo, è caratterizzato da una maggiore flessibilità e performance: per questo deve essere possibile adattare le applicazioni, senza sensibili modifiche, ai cambiamenti e alle evoluzioni della piattaforma sottostante.

### 2.1.3 Le applicazioni dinamicamente adattive

Le considerazioni fatte precedentemente possono essere generalizzate introducendo la possibilità, nel modello di programmazione grid-aware, di sviluppare applicazioni dinamicamente adattive, in cui, per esempio, l'allocazione delle risorse varia a run-

time per garantire un determinato livello di performance. La riallocazione e la rischedulazione delle risorse dovrebbero avvenire nel caso in cui un nodo non sia più disponibile o vi sia un certo sbilanciamento del carico assegnato ai nodi, oppure perché è richiesto un incremento di performance alle applicazioni in seguito ad una qualsiasi altra emergenza. Il problema, in questi casi, consiste principalmente nel prendere in esame delle azioni complesse che implicano una trasformazione nella versione eseguibile del programma, come:

- un differente grado di parallelismo;
- una differente distribuzione e partizionamento dei dati;
- delle versioni alternative dell'implementazione del programma con le stesse funzionalità.

Un approccio rigoroso al problema dell'adattività dei programmi si può basare su questi punti:

1. le diverse modalità per esprimere la strutturazione e la ristrutturazione di una computazione devono essere disponibili nello stesso formalismo di programmazione;
2. queste modalità devono essere caratterizzate da un modello di costo (*performance model*) che permette di guidare le fasi di strutturazione e ristrutturazione con complessità ed *overhead* accettabili.

Ciò può corrispondere all'uso di differenti forme di parallelismo, come normalmente accade nei modelli di programmazione parallela strutturata basati sul concetto di *skeleton* [16]. In questi modelli un insieme consistente di forme di parallelismo è fornito ai programmatori per strutturare le applicazioni: per esempio, *pipeline*, *farm* o *divide&conquer* sono tipici paradigmi *task-parallel* (*stream-parallel*), invece *map*, *reduce*, *prefix*, *scan*, *stencil* sono tipici paradigmi *data-parallel*. Nella programmazione parallela strutturata in cui è adottato un *linguaggio di coordinamento*, questo agisce

da meta-linguaggio ed è usato per comporre programmi scritti in qualsiasi linguaggio esistente (C, C++, Java, Fortran). Questi programmi, inoltre, potrebbero essere già esistenti sotto forma di librerie, componenti ed altro. Un'altra caratteristica degli skeleton è quella di possedere un modello semantico e un modello di costo, che rendono questo approccio molto promettente per le griglie computazionali: infatti, essendoci un modello di costo, l'implementazione dinamica e statica di ogni skeleton è parametrica rispetto a pochi parametri. Per esempio, il supporto a run-time, una volta che lo skeleton è stato istanziato per la particolare applicazione, può variare dinamicamente l'attuale grado di parallelismo, l'attuale numero di partizioni dei dati ed altro che potrebbe far migliorare la performance.

## 2.2 Il modello di programmazione di ASSIST

In questa sezione viene dimostrato come ASSIST possegga le caratteristiche essenziali degli ambienti di programmazione per applicazioni grid-aware. ASSIST [37, 4, 57, 2, 49, 15] (*A Software development System based upon Integrated Skeleton Technology*) è un ambiente di programmazione orientato allo sviluppo di applicazioni parallele e distribuite ad alta performance secondo un approccio unificato, sviluppato dal gruppo di ricerca di Architetture Parallele presso il Dipartimento di Informatica dell'Università degli Studi di Pisa. ASSIST, nato per macchine parallele e cluster omogenei, sta evolvendo verso ambienti di programmazione complessi ad alte prestazioni come le grid.

La programmazione parallela strutturata, basata sul modello a skeleton [16], è un approccio che va incontro alla complessità di progettazione delle applicazioni ad alta performance. La validità di questo approccio è stata provata per macchine parallele omogenee (MPP, Cluster, Cluster di SMP) ma risulta ancora più valida per piattaforme dinamiche ed eterogenee (griglie computazionali). La validità dei modelli a skeleton può essere così riassunta:

- i programmi paralleli sono scritti, attraverso un *linguaggio di coordinamento*,

come composizione di paradigmi paralleli predefiniti, chiamati skeleton;

- per ogni skeleton è conosciuto un insieme di *template* di implementazione e un modello di costi, entrambi parametrici, che possono essere usati per ottimizzare la compilazione e il supporto a run-time;
- dai precedenti punti segue che scrivere un programma parallelo è piuttosto veloce e facile poiché le decisioni più difficili riguardanti l'implementazione parallela vengono delegate alla compilazione e al supporto a run-time.

Pur con i suoi vantaggi, l'approccio classico a skeleton ha degli svantaggi in termini di potenza espressiva ed efficienza per applicazioni complesse, flessibilità nell'adattarsi a una varietà di combinazioni di paradigmi paralleli, così come in termini di interoperabilità ed idoneità alla programmazione basata a componenti per le applicazioni grid-aware. ASSIST da questo punto di vista si pone come un'evoluzione dei classici modelli a skeleton.

### 2.2.1 Programmi ad alta performance definiti come grafi

In diverse applicazioni le strutture di computazione esprimibili dai classici skeleton non sono adeguate. Infatti:

- molte applicazioni possono essere concepite in termini di composizione di componenti sviluppati indipendentemente, senza una struttura predefinita;
- la struttura della composizione potrebbe seguire alternativamente un modello *data-flow* o uno *non-deterministico* (*event-driven computation*);
- i componenti possono interagire attraverso diversi pattern di comunicazione, stream, eventi o invocazioni singole (e.g. RPC);
- molti algoritmi paralleli possono essere espressi in uno stile *Divide&Conquer* [17] (D&C) in cui, pur essendo definiti degli skeleton specifici, la performance potrebbe risultare bassa in generale, poiché vi è una grande varietà di configurazioni e situazioni in cui questi paradigmi occorrono.

In ASSIST, un programma parallelo può essere strutturato come un *grafo generico*, in cui i nodi corrispondono ai moduli paralleli o sequenziali, e gli archi, invece, ai canali di comunicazione sui quali sono trasmessi stream di valori 'tipati'. Gli stream, quindi, costituiscono il meccanismo di composizione generale dei moduli (vedi fig. 2.1). Altri pattern di comunicazione (invocazione singola, RPC, ecc...)

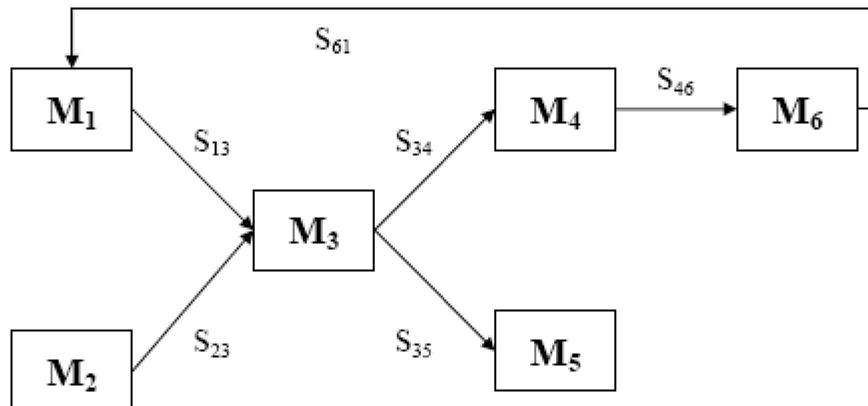



Figura 2.1: Un esempio di applicazione ASSIST definita come un grafo.

possono facilmente essere espressi come casi particolari di stream o implementati in un formalismo basato su stream. Una computazione, applicata ai valori di uno stream in ingresso, termina quando lo stream finisce. I moduli possono avere uno stato che è inizializzato all'inizio della computazione e che rimane significativo fino alla conclusione dell'applicazione. Ogni modulo può avere un qualsiasi numero di stream di ingresso e di uscita. In generale, gli stream di ingresso sono selezionati non-deterministicamente in una computazione *CSP-like* [34]. Al contrario, un nodo potrebbe avere un comportamento data-flow con tutti i suoi stream in ingresso, oppure, in un caso più generale, potrebbe avere un comportamento misto, sia non-deterministico sia data-flow.

La scelta di grafi e stream in ASSIST non ha solo una valenza generale, ma elimina anche le penalizzazioni di performance menzionate precedentemente. Infatti gli stream sono esplicitamente controllati dal programma, ed è più semplice raggiungere un'implementazione efficiente di semplici canali di comunicazione e co-

mandi con guardia nei complessi pattern di comunicazione che sono richiesti in una computazione D&C e più in generale in una computazione dinamica ed irregolare.

Nella versione a componenti di ASSIST [3], che verrà utilizzata nell'integrazione con il CCM, i singoli nodi di un grafo, o un intero sottografo, corrispondono ai componenti, le cui interfacce sono espresse in termini di stream che attivano le operazioni interne dei moduli. Strutturare quindi un programma ASSIST in termini di grafi e stream è sufficiente per catturare il significato essenziale dei componenti ad alta performance, in un modo che è completamente indipendente dalla specifica tecnologia a componenti.

### 2.2.2 Uno skeleton generico: *parmod*

Ogni skeleton classico (pipeline, farm, ecc...) corrisponde ad un paradigma specializzato per esprimere parallelismo. Sebbene in molti algoritmi c'è esattamente ciò di cui i programmatori necessitano, ci sono delle applicazioni che richiedono delle strutture più flessibili per ragioni di efficienza e/o potere espressivo. Degli esempi in riguardo possono essere:

- farm *stream-parallel* con una strategia di *scheduling* ad-hoc, *worker* che hanno delle forme di stato interno o delle comunicazioni aggiuntive, o con una strategia specifica di ordinamento dei *task*;
- algoritmi data-parallel con diversi tipi di *stencil* che possono essere riconosciuti staticamente o con stencil dinamici;
- strategie specifiche di distribuzione dati, che vanno dallo scheduling *on-demand* a *scatter*, *multicast*, *broadcast*;
- nessuna limitazione nel numero di stream in ingresso ed uscita, controllati secondo uno stile non-deterministico o data-flow;
- esistenza di uno stato interno considerato significativo per la durata della

computazione, che è di solito proibito dalla classica semantica degli skeleton data-flow;

- combinazioni miste di paradigmi stream e data-parallel (computazione *sistolica*);

- moduli capaci di comportarsi secondo differenti paradigmi in differenti fasi della loro esecuzione.

Tutte queste situazioni possono essere emulate attraverso skeleton specializzati, a scapito della semplicità e/o dell'efficienza del codice.

La soluzione di ASSIST è un skeleton generico, cioè un costrutto *general purpose* che può essere adattato alla specifica istanza di applicazione. Questo costrutto, chiamato modulo parallelo o *parmod*, ha come caratteristica essenziale, quando descrive la computazione equivalente ad uno skeleton specializzato, di non incorrere in sensibili *overhead*, dovuti alla sua generalità.

Un modulo parallelo è definito come segue (vedi fig. 2.2):

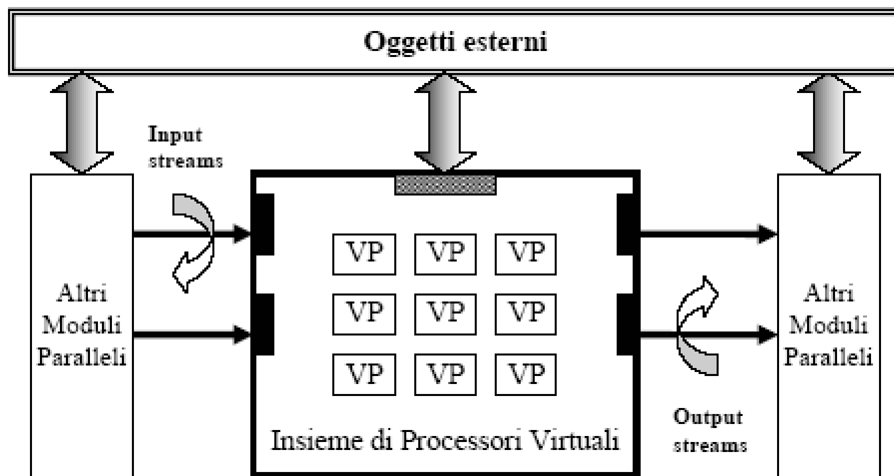



Figura 2.2: Elementi di un modulo parallelo (parmod).

- un insieme di *processori virtuali* (VP) con uno stato interno. L'insieme è dichiarato con una certa topologia che fornisce i nomi ai processori virtuali, quando richiesti. Le topologie possibili al momento sono:

**array multidimensionale** per indicare ogni processore virtuale mediante i valori di uno o più indici. Ad esempio nella topologia array[i:N][j:N] VP; il generico processore virtuale si chiama VP[i][j];

**none** nel caso in cui il nome dei processori virtuali non è significativo per la computazione. Utilizzato per esempio, in alcune strutture di tipo farm composte da worker completamente indipendenti;

**one** indica il parmod con un solo processore virtuale. Si differenzia da un modulo sequenziale perché permette il non-determinismo e il controllo degli stream in ingresso e in uscita.

- un insieme di stream in ingresso tipati, controllati in modo data-flow o non-deterministicamente. Un volta selezionato, il dato di uno stream in ingresso potrebbe essere *pre-processato*, e poi inviato ai processori virtuali secondo una strategia prescelta (scatter, multicast, broadcast, on-demand), espressa nel programma mediante una propria dichiarazione;

- un insieme di stream in uscita controllati indipendentemente attraverso operazioni *collettive* sui valori dello stato dei processori virtuali. Il *post-processing* sui dati da inviare agli altri moduli può essere eseguito in modo collettivo.

Il supporto a run-time provvederà a mappare l'insieme dei processori virtuali su un insieme di nodi di elaborazione (processori reali).

In questo modo, tutti gli skeleton specializzati stream e data parallel conosciuti possono essere espressi direttamente in ASSIST. Inoltre, molto importante è il fatto che nessuna penalità di performance viene pagata rispetto agli equivalenti skeleton specializzati. L'esecuzione di alcuni benchmark, scritti in ASSIST [2, 1, 48], ha dimostrato che la performance raggiunta da un parmod è paragonabile o migliore a quella raggiunta dagli equivalenti skeleton specializzati, da programmi sviluppati in un linguaggio data-parallel o direttamente sviluppati tramite la libreria MPI.

### 2.2.3 Supporto alle applicazioni grid-aware

In ASSIST vi è la possibilità di esprimere le strategie globali di adattamento dinamico nelle applicazioni grid-aware. Per fare ciò si ha bisogno di un *Application Manager* (AM), che, logicamente, è un'entità centralizzata (l'implementazione, invece, potrebbe essere decentralizzata) e che si occupa di:

- modello di performance;
- monitoring;
- resource discovery;
- strategie di scheduling sia locali ai singoli nodi che globali alla griglia;
- strategie di allocazione di codice e dati.

Per poter supportare questo, vi sono delle modifiche da effettuare sulla corrente implementazione di ASSIST [19, 20]. In particolare, tali modifiche riguardano il supporto a run-time dei parmod e degli oggetti condivisi, in modo che parti dello stesso componente parallelo possano essere riallocati dinamicamente in nodi differenti secondo le decisioni dell'AM. Per costruire, quindi, un'applicazione grid-aware, come discusso nella sezione 2.1.1, basterà che sottografi (o moduli) di programmi ASSIST vengano mappati in componenti standard, e, più in generale, vengano resi interoperabili con componenti non-ASSIST (e.g. componenti CCM). Inoltre, ogni componente dovrà possedere il proprio contratto di performance, affinchè l'AM possa decidere in merito alla sua esecuzione.

Questa versione di ASSIST, con l'AM, verrà inglobata nell'*assist-lib* dopo opportune sperimentazioni e valutazioni, incluse quelle effettuate con questa tesi.

### 2.2.4 Un esempio di applicazione adattiva

Per chiarire l'approccio di ASSIST alla progettazione di applicazioni dinamicamente adattive verrà mostrato un esempio di tale applicazione. La composizione dell'applicazione è quella mostrata in figura 2.3.

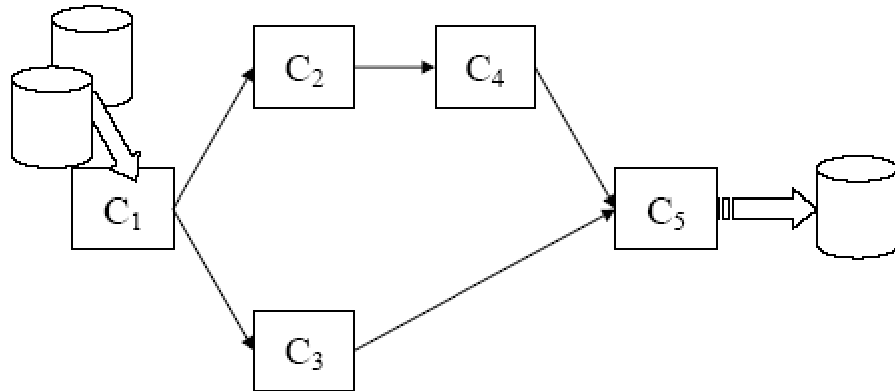



Figura 2.3: Esempio di una applicazione adattiva costituita da componenti paralleli.

**C1** è un componente d'interfaccia ad una gerarchia di memoria su griglia, che si occupa di trasformare l'insieme di dati disponibile sulla griglia in due stream di dati, in cui il primo è inviato a C2 e l'altro a C3. C1 potrebbe essere un componente già esistente e disponibile sulla griglia;

**C2** è un componente che incapsula un programma ASSIST. Il contratto di performance di C2 specifica che:

- C2, di default, è un modulo sequenziale (parmod con topologia one) che esegue una certa funzione F;
- quando i servizi di Monitoring e Performance Model generano l'evento che segnala la necessità/opportunità di aggiustare il corrente livello di performance, C2 viene trasformato in un farm (parmod con topologia none e distribuzione dei dati ai worker on-demand), i cui worker eseguono la stessa funzione F. È l'AM che determina il numero di worker necessari e la loro allocazione sulla griglia.

**C3** è un componente incapsulato in un programma ASSIST data-parallel (parmod con topologia array, opportuna distribuzione dei dati per lo specifico algoritmo data-parallel). Il contratto di performance di C3 specifica che, di default, il programma ASSIST deve essere eseguito su un singolo nodo della griglia con architettura interna di tipo cluster, mentre in fase di ristrutturazione può

modificare il grado di parallelismo. La riallocazione potrebbe utilizzare risorse appartenenti allo stesso nodo o ad altri nodi della griglia;

**C4** è un componente che incapsula un programma ASSIST, il quale, di default, è un modulo sequenziale, mentre in fase di ristrutturazione può essere trasformato in un parmod che opera sugli stream in ingresso secondo uno stile data-parallel o farm, in base ai valori dello suo stato interno e degli elementi dello stream. In questo caso il principio di adattamento è applicato sia a livello di programma sia a livello di allocazione;

**C5** è un componente che incapsula un programma ASSIST, il quale opera non-deterministicamente, sui valori di input ricevuti da C3 e C4, e trasforma i due stream in un insieme di dati. Il contratto di performance di C5 specifica che il componente può essere allocato ed eseguito solo su un determinato nodo della griglia e che non occorre nessuna riconfigurazione. Ciò potrebbe essere dovuto a ragioni di privacy e/o sicurezza.

Potrebbe accadere che ad un certo istante C2 diventi un collo di bottiglia, che causa una sostanziale degradazione nelle prestazioni dell'intera applicazione. AM, a questo punto, provvederà a trasformare C2 in una versione con l'apposito grado di parallelismo e la rischedulerà e riallocherà assumendo che le necessarie risorse possano essere trovate. Nel caso in cui si voglia ristrutturare un componente data-parallel, la strategia dell'AM deve essere anche applicata alla ridistribuzione dei dati che costituiscono lo stato interno del parmod.

# Capitolo 3

# Il modello di programmazione a componenti

Sommario

In questo capitolo verranno studiate le caratteristiche della programmazione a componenti in risposta ai problemi delle applicazioni grid-aware. Verranno elencate le varie categorie di tecnologie a componenti esistenti e i loro principali progetti. Verranno inoltre analizzati in dettaglio i progetti presenti nei *component framework* ad alta performance e non.

I componenti e le tecnologie basate sui componenti software (*componentware* [7]) sono oggi ben conosciuti e largamente utilizzati nello sviluppo di applicazioni. Vi è un grosso numero di lavori e ricerche nel componentware [6], i cui approcci sono in continua evoluzione ed è spesso difficile tenere traccia dei *trend* correnti in questo campo. I componenti potrebbero sembrare una tecnica ben sviluppata e pronta per essere utilizzata. Dopo trent'anni di ricerche e studi, però, non vi sono consensi su una definizione esatta di componenti.

La prima idea sui componenti può essere fatta risalire ad un articolo, *Mass-Produced Software Components*, pubblicato da M.D. Mellroy [45] alla conferenza NATO di Garmish nel 1968, in cui troviamo la seguente affermazione:

*...yet software production in the large scale would be enormously helped by the availability of spectra of high quality routines, quite as mechanical design is abated by the existence of families of structural shapes, screws or resistors...*

La definizione più citata di componente però è quella di Szyperski [55, 56]:

*Un componente software è una unità di composizione con solamente delle interfacce contrattualmente specificate e delle dipendenze esplicite. Un componente software può essere sviluppato indipendentemente e soggetto a composizione da terzi.*

## 3.1 Perché si ha bisogno dei componenti?

Lo sviluppo di software è una disciplina costosa e soggetta ad errori. Per questo, già nel 1968 alla conferenza NATO di Garmisch, furono coniati i termini *Software Crisis* e *Software Engineering*. Da quel momento, la disciplina informatica ha fatto passi da gigante: in un primo momento lo sviluppo del software è stato dominato dalla creazione di nuove tecniche di progettazione ed analisi strutturate, che sono state seguite dall'introduzione del paradigma *object-oriented*. Tale paradigma, anche se ha permesso di incrementare il riutilizzo, l'incapsulamento e lo sviluppo indipendente di software, ha diversi svantaggi:

- gli oggetti si compongono e cooperano solo se scritti con lo stesso linguaggio di programmazione;
- le interfacce sono unidirezionali (*import* solamente);
- gli oggetti non possono essere sviluppati indipendentemente;
- gli oggetti sono strettamente accoppiati: l'esecuzione avviene nello stesso processo e spazio di dati.

Oggi, i sistemi software sono distribuiti sulle reti e possono interagire l'uno con l'altro. Inoltre vanno in esecuzione in ambienti dinamici ed adattivi, possono essere parzialmente riutilizzati o sviluppati nuovamente, devono permettere alle applicazioni scritte in differenti linguaggi, su differenti tecnologie, e in differenti sistemi operativi di cooperare. I componenti, per le loro caratteristiche, rispondono egregiamente alle esigenze dei sistemi software odierni. Infatti, attraverso le lore in-

terfacce esplicite e la loro interoperabilità tra differenti linguaggi, permettono di incrementare:

**la riusabilità** tramite l'utilizzo di componenti esistenti che non necessitano, quindi, dello sviluppo della stessa funzionalità sempre più volte;

**la qualità del risultato** tramite il riutilizzo di componenti esistenti che sono già in uso da diverso tempo e che quindi possono essere considerati privi di errori;

**la possibilità di *outsourcing*** tramite il coinvolgimento di molti esperti, provenienti da diversi domini scientifici, nello sviluppo, e quindi nell'unione, di componenti specifici ad una determinata area scientifica.

Inoltre, attraverso l'utilizzo delle interfacce, si ha una migliore e più chiara definizione dell'architettura del sistema a componenti, che può essere facilmente estesa da nuovi componenti, mantenuta ed adattata a nuovi requisiti, e quindi persistere nel tempo.

## 3.2 Le categorie di *Componentware*

Il termine *componentware* [7] si riferisce alle tecnologie per lo sviluppo di sistemi software, attraverso l'utilizzo di componenti come blocco base. Le categorie di componentware esistenti [6] si possono così riassumere (vedi fig. 3.1):

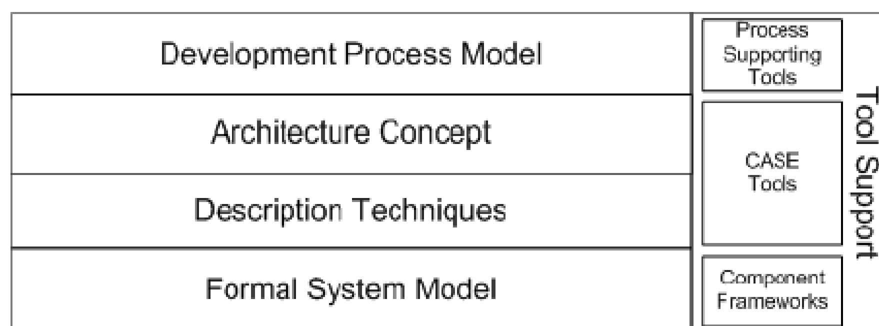



Figura 3.1: Categorie di *Componentware*.

**approcci formali** formalismi matematici usati per esprimere senza ambiguità i blocchi base del sistema, le loro relazioni e le loro proprietà;

**approcci descrittivi** notazioni testuali e grafiche per nascondere il formalismo sottostante all'utente. Un esempio grafico sono i diagrammi per rappresentare aspetti sia statici che dinamici;

**approcci centrati sull'architettura** approcci formali per la progettazione e la documentazione dell'architettura di un sistema a componenti;

**approcci centrati sul processo** forniscono dei modelli di processi orientati ai componenti, utilizzati, cioè, per strutturare il processo di sviluppo di un sistema a componenti;

**framework e tool** completano il processo di sviluppo, attraverso l'uso di tecniche specifiche di descrizione dei componenti.

Successivamente, per ogni categoria di componentware, verranno descritti brevemente alcuni progetti esistenti, con maggiore enfasi ai *component framework*.

### 3.2.1 Approcci Formali

I progetti orientati allo sviluppo di formalismi matematici, per esprimere senza ambiguità le unità base di un sistema, sono sviluppati generalmente in ambienti accademici o in laboratori di ricerca. Di seguito vengono presentati due tra i progetti più conosciuti: *FOCUS* e *ROOM*.

*FOCUS* [10]: è una teoria logico-matematica per la specifica e la progettazione modulare e per l'interpretazione astratta di sistemi a componenti. Fornisce una chiara nozione di componente, compresi i modi per specificarli, manipolarli e comporli. Definisce infine un operatore (*parallel composition with feedback*) che permette di modellare l'esecuzione e l'interazione simultanea di componenti in una rete.

*ROOM* [51]: *Real-Time Object-Oriented Modeling*, è un metodo per la specifica, la progettazione e la costruzione di sistemi real-time distribuiti. I blocchi base di un sistema, secondo ROOM, sono chiamati *Actors*. Essi incapsulano un thread attivo e delle informazioni di stato. Gli Actor comunicano attraverso delle porte (*Service Access Point*), le quali hanno associato un protocollo. Affinchè due Actor possano comunicare, vi è la necessità di stabilire un *binding* tra due porte degli attori che siano compatibili o *coniugate*.

### 3.2.2 Approcci Descrittivi

In questa sezione vengono presentati una classe di tecniche (*ADLs*) ed un progetto (*UML*) che forniscono delle notazioni testuali e grafiche rispettivamente per rappresentare aspetti sia statici che dinamici del sistema.

*UML* [9]: *Unified Modeling Language* è una notazione grafica orientata agli oggetti, utilizzata per la modellazione di sistemi secondo diverse prospettive. I modelli in UML si possono dividere in due categorie:

- *statici*, per descrivere gli elementi del sistema e le loro interazioni;
- *dinamici*, per descrivere il comportamento del sistema nel tempo.

Ogni modello, inoltre, ha diverse *viste*, che permettono la descrizione del sistema a vari livelli: casi d'uso, logico, operazionale, di sviluppo e fisico.

*ADLs* [42, 52]: *Architecture Description Languages* sono delle tecniche basate su componenti, per la descrizione di architetture software. Sono usati per definire e modellare un'architettura di un sistema prima della sua implementazione. Gli ADL, inoltre, sono stati sviluppati per la programmazione su larga scala.

### 3.2.3 Approcci centrati sull'Architettura

Di seguito viene presentato uno tra i progetti esistenti focalizzato sulla progettazione e documentazione di un sistema a componenti: *Quasar*.

*Quasar* [53]: definisce dei concetti architetturali su due livelli: il livello concettuale include i principi e le euristiche architetturali generali per lo sviluppo di sistemi a componenti, il livello di sistema, invece, un mapping esplicito tra i concetti generali ed uno specifico *technical component framework*.

### 3.2.4 Approcci centrati sul Processo

In questa sezione vengono presentati tre progetti centrati sulla specifica di un processo di sviluppo per un sistema a componenti: *Catalysis*, *KobrA* e *UML Components*.

*Catalysis* [22]: è stato uno dei primi *component-based software engineering processes* che ha combinato processi iterativi ed incrementali di sviluppo del software attraverso nozioni di componenti e di UML. I blocchi base in Catalysis sono gli *oggetti* e le *azioni*. Gli oggetti possono rappresentare qualsiasi cosa: linguaggi di programmazione, componenti software in esecuzione, macchine, e possono essere strutturati gerarchicamente. Le azioni rappresentano le interazioni che ci possono essere tra tutti i tipi di oggetti, e possono essere anch'esse strutturate gerarchicamente.

*KobrA* [23]: è un approccio per la progettazione su larga scala di sistemi component-based. Un sistema è visto come una semplice, generica ed astratta *scatola nera* (*black box*). In un processo iterativo, il livello di genericità e astrazione sono ridotti ed il sistema viene decomposto in componenti.

*UML Components* [14]: è un approccio con due obiettivi principali: la definizione di un semplice processo per identificare e specificare i componenti e la definizione

di un'applicazione della notazione UML per sostenere il processo. Il processo è un adattamento dell'*object-oriented Rational Unified Process* (RUP) per lo sviluppo basato su componenti. Le fasi del RUP *analysis* e *design* sono rimpiazzate dalle nuove fasi *specification*, *provisioning* e *assembly*. L'approccio definisce le attività e i metodi per identificare, far interagire e specificare i componenti.

### 3.2.5 Component Frameworks

Questo approccio si focalizza esplicitamente sullo sviluppo di componenti e sulla specifica di ambienti a run-time per componenti. Qui presenteremo tre dei maggiori component framework emersi in questi anni: *COM*, *EJB* e *CCM*. C'è da dire però che questi modelli non inglobano l'idea dell'alta performance dentro un componente, che invece è necessaria, come abbiamo visto nel capitolo 2, per le applicazioni grid-aware. I progetti in tal senso verranno presentati nella sezione 3.4.

*COM* [41]: *Common Object Model*, sviluppato dalla Microsoft, è uno dei component framework più datati in circolazione. COM, discendente dell'*Object Linking and Embedding Technology* (OLE), è stato sviluppato nel 1995. Inizialmente progettato come una semplice tecnologia locale a componenti, COM è stato esteso nel 1996 nel *Distributed Common Object Model* (DCOM). DCOM permette l'utilizzo di componenti COM in un ambiente distribuito. Attualmente, COM e DCOM sono uniti con il *Microsoft Transaction Service* (MTS) nel COM+, che fornisce il component framework per la piattaforma .NET. L'idea base dietro COM+, è che tutti i servizi ed applicazioni Microsoft possono essere incapsulati da componenti COM ed utilizzati facilmente in qualsiasi piattaforma Microsoft. Ciò porta ad una nozione di componente generale ma debole: COM+ vede un componente come un semplice binario con un identificatore unico e un'interfaccia non modificabile per assicurare compatibilità con le versioni precedenti. Inoltre, l'interoperabilità e la riusabilità sono ristrette alle piattaforme Microsoft.

*EJB* [46]: *Enterprise JavaBeans*, furono rilasciati dalla Sun Microsystems nel 1998. EJB rappresenta una parte essenziale della *Java 2 Platform Enterprise Edition* (J2EE). Originariamente definita dalla Sun in cooperazione con altre industrie, oggi le specifiche sono mantenute da una organizzazione *open*, la *Java Community Process* (JCP). Le specifiche EJB coprono diversi argomenti:

- tipi e struttura di un componente;
- sviluppo e composizione (*deployment*) di un componente;
- struttura di un ambiente a run-time per componenti (*container*);
- accordi tra differenti componenti;
- accordo tra componenti e container;
- regole nel processo di sviluppo.

Nell'EJB, i componenti sono chiamati *enterprise beans*. Un *enterprise bean* contiene una classe *bean* con una certa intelligenza, un'interfaccia locale (*home*) per la gestione del *bean* e un'interfaccia remota che permette l'accesso, ai clienti, all'intelligenza contenuta nel *bean*. Durante il processo di composizione sono aggiunte delle classi tecniche che permettono di stabilire degli accordi tra i *bean* e il loro ambiente a run-time. Inoltre lo standard EJB prevede un'implementazione di riferimento, cioè un modello di sviluppo, di supporto al processo di sviluppo, e una *test suite* per la certificazione di diverse implementazioni della J2EE. I componenti EJB, quindi forniscono uno standard *open*, in cui l'idea base sta nella riusabilità del software in ambienti eterogenei. Purtroppo, però, la nozione di componente nell'EJB è troppo restrittiva dal punto di vista concettuale; per esempio, non permette interfacce multiple, e, in generale, tende a rappresentare un oggetto più che un componente.

*CCM* [35, 54, 33, 32]: *CORBA Component Model* (per maggiori dettagli si faccia riferimento alla sez. 3.3), fu rilasciato nel 2000 dall'*Object Management Group*

(OMG), una delle più grandi organizzazioni per standard *open*, in risposta all'EJB, poiché l'uscita del prodotto della Sun aveva diminuito l'interesse verso la loro piattaforma *open* CORBA (*Common Object Request Broker Architecture*), un *middleware platform standard* per sistemi distribuiti che può essere comparato alla piattaforma J2EE, tranne per il modello a componenti (introdotto appunto con il CCM). Per ottenere un maggiore gradimento, il CCM fin dall'inizio ha supportato l'interoperabilità con le specifiche EJB. Per questo, infatti, definisce due tipi di standard per componenti:

- un livello base, più o meno una copia dello standard EJB;
- un livello esteso con una più sofisticata nozione di componente.

Un componente CCM può esportare ed importare diverse interfacce, chiamate nel primo caso *facets* e nel secondo *receptacles*. Inoltre, il CCM, come CORBA [47] d'altronde, si basa sull'IDL *Interface Definition Language*, che insieme all'ORB (*Object Request Broker*), costituiscono i punti forti per la piattaforma e per l'indipendenza dai linguaggi di programmazione da parte del component framework dell'OMG.

## 3.3 Il CORBA Component Model (CCM)

### 3.3.1 Common Object Request Broker Architecture (CORBA)

CORBA (*Common Object Request Broker Architecture*) [47, 54] è uno standard introdotto dall'*Object Management Group* (OMG). Introduce la possibilità di implementare applicazioni portabili basate su un sistema distribuito e interoperabile. Il meccanismo attraverso il quale le applicazioni inviano le proprie richieste e ricevono le adeguate risposte è l'*Object Request Broker* (ORB). Attraverso l'ORB, applicazioni diverse fra loro, eseguite eventualmente su macchine distribuite ed eterogenee, possono collaborare e scambiarsi informazioni.

La chiave per comprendere l'architettura di CORBA è il *Reference Model* che si compone di diverse parti:

- *Object Request Broker*, che, come già detto, implementa i meccanismi di comunicazione affinchè due applicazioni diverse possano collaborare fra loro, anche in ambienti distribuiti ed eterogenei;

- *Object Services*, un insieme di servizi (interfacce e funzioni) che consente di comporre applicazioni semplici per l'implementazione di applicazioni più complesse. Tali servizi sono indispensabili per l'implementazione di applicazioni distribuite, e sono indipendenti dal tipo di applicazione a cui si riferiscono. Ad esempio, il servizio di gestione del ciclo di vita di un'applicazione definisce i criteri per creare, rimuovere, spostare o copiare applicazioni da un punto all'altro del sistema, tutto questo senza che debba conoscere l'implementazione dell'applicazione che sta mantenendo;

- *Common Facilities*, un insieme di servizi condivisi da più applicazioni ma che non sono indispensabili come gli Object Service.

- *Application Object*, che sono implementati e venduti dai vari produttori di software e vanno ad arricchire le funzioni di CORBA.

L'Object Request Broker rappresenta l'aspetto fondamentale del Reference Model. In particolare l'ORB è responsabile di tutti i meccanismi necessari per trovare l'implementazione dell'oggetto per una richiesta, per preparare l'implementazione dell'oggetto a ricevere la richiesta e per comunicare i dati effettuando la richiesta stessa. Per invocare un metodo o una procedura in maniera remota, il client effettua la richiesta accedendo all'interfaccia[1] dell'oggetto (server), da invocare, in due modi differenti: accedendo ad una interfaccia indipendente da quella che l'oggetto richiesto possiede (*Dynamic Invocation Interface*) o accedendo ad uno *stub* specifico

[1] Un'interfaccia consiste in un insieme di operazioni e di parametri ad esse correlati.

per l'interfaccia dell'oggetto richiesto (*IDL stub*). Ciò significa che le interfacce degli oggetti possono essere generate in due modi differenti:

1. le interfacce posso essere definite staticamente in un *Interface Definition Language*, chiamato *OMG IDL*. Questo linguaggio definisce le interfacce e quindi le operazioni che espongono gli oggetti, ed i tipi dei parametri richiesti in tali operazioni. IDL è quindi il mezzo con il quale un'implementazione di un oggetto rende pubbliche quali sono le operazioni disponibili e come possono essere invocate dai suoi potenziali clienti;

2. alternativamente, o in aggiunta, le interfacce possono essere aggiunte ad un *Interface Repository Service*; questo servizio rappresenta i componenti di un'interfaccia come oggetti, permettendone l'accesso a run-time.

Le definizioni delle interfacce in OMG IDL e Interface Repository, sono usate per generare i *client stub* e gli *object implementation skeleton*, tramite un processo di compilazione. Questi ultimi sono utilizzati per gestire, in modo trasparente alle applicazioni, l'interoperabilità tra client e server (oggetto CORBA) che possono essere implementati in linguaggi di programmazione diversi ed essere in esecuzione su piattaforme diverse. Infatti il client non si interfaccia direttamente con il server ma con il client stub, che ne espone le interfacce e si incarica di effettuare tutte le operazioni necessarie (e.g. serializzazione/deserializzazione dei dati) per effettuare la chiamata al server e consegnare l'eventuale risultato di quest'ultimo al client. Allo stesso modo il server non si interfaccia direttamente al client ma al suo object implementation skeleton, che effettua tutte le operazioni necessarie per consegnare la richiesta al server e spedire il risultato al client. Per fare in modo che tutto ciò sia possibile, lo stub deve essere generato nello stesso linguaggio di programmazione del client, e lo skeleton nello stesso linguaggio di programmazione del server. CORBA supporta linguaggi di programmazione differenti, in quanto vi sono compilatori che generano, a partire dall'IDL, stub e skeleton in diversi linguaggi di programmazione: C/C++, COBOL, Java, ADA, SmallTalk.

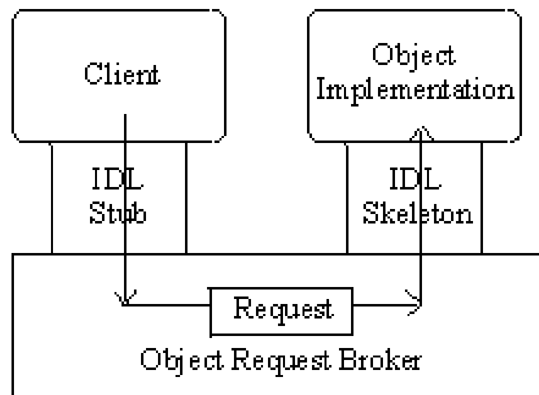



Figura 3.2: Trasmissione della richiesta di un client all'implementazione dell'oggetto CORBA (server).

### 3.3.2 Introduzione al modello a componenti di CORBA

Il *CORBA Component Model* [35, 33, 32], o meglio CCM, è un'estensione degli oggetti CORBA (vers. 3) [54] definita dall'OMG. Questa estensione avviene attraverso la definizione di caratteristiche e servizi aggiuntivi che rimangono consistenti con l'ambiente standard di CORBA. CCM permette la definizione di componenti che incapsulano applicazioni già definite ed implementate in modo da poter garantire l'interoperabilità e la portabilità anche di applicazioni completamente diverse fra loro. CCM si occupa della generazione del codice necessario al corretto funzionamento delle applicazioni incapsulate (interfacce), della gestione di tali applicazioni, e dell'integrazione con i normali servizi messi a disposizione da CORBA quali, ad esempio, gestione delle transazioni, sicurezza, persistenza e notifica di eventi. Nello specifico esiste un contenitore che si interpone fra il mondo CCM (e quindi CORBA), e l'applicazione vera e propria. Tale contenitore si compone di due parti che vedremo nel dettaglio in seguito: l'*Application Server* ed il *Container*. Inoltre, CCM, permette la configurazione dinamica del componente.

I tre principali concetti introdotti dal CCM sono:

- *Abstract Component Model*, permette di definire quali sono le funzioni esportate dall'applicazione che si sta incapsulando, quali sono i servizi esterni che

tale componente richiede (offerti da altri componenti), quale tipo di comunicazione è necessario stabilire con gli altri componenti (sincrona, per l'invocazione di operazioni, asincrona per la notifica di eventi), quali sono gli attributi riconfigurabili, qual è il ciclo di vita del componente;

- *Component Implementation Framework* (CIF), definisce un modello per la costruzione di un componente CCM. In altre parole definisce il modo in cui il componente deve essere implementato. Questo strumento gestisce alcune funzioni, ad esempio introspezione delle interfacce del componente, persistenza, attivazione, ed altro, in maniera automatica;

- *Component Container Programming Model* è uno strumento per la definizione dell'Application Server del componente stesso. Generalmente, l'Application Server è inserito all'interno del CCM in modo da renderlo portabile e semplificare le operazioni di gestione dell'applicazione incapsulata (attivazione, disattivazione, ed altro). La funzione principale dell'Application Server è quella di interfaccia verso i servizi CORBA (persistenza, sicurezza, transazioni, eventi). L'Application Server non accede direttamente alle applicazioni incapsulate, ma si interfaccia con i Container (specializzati per ogni categoria di applicazione) che accedono all'applicazione specifica.

### 3.3.3 Il CCM in dettaglio

I componenti sono il blocco base da cui si costituisce un'applicazione basata sulla tecnologia CCM. Il programmatore, che costruisce un'applicazione CCM, deve definire univocamente l'interfaccia del componente attraverso l'IDL (*Interface Definition Language*), in modo che sia chiaro il tipo di servizio messo a disposizione dal componente stesso. In questo modo il client è in grado di accedere a tali servizi senza dover conoscere l'implementazione del servizio stesso. Questa caratteristica permette la costruzione di applicazioni complesse attraverso la composizione di componenti separati. Tali componenti possono essere distribuiti su macchine diverse e

interconnessi dinamicamente al resto dell'applicazione solo nel momento in cui vi è effettiva necessità di uno specifico servizio.

In un primo momento, il componente CCM può essere immaginato come mostrato in figura 3.3.

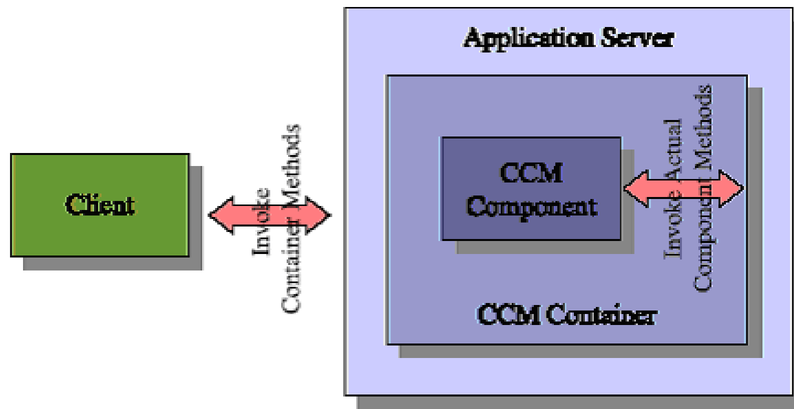



Figura 3.3: Descrizione di un componente CCM.

La principale funzione dell'Application Server è quella di rendere accessibile al mondo esterno l'applicazione racchiusa all'interno del componente CCM. L'Application Server gestisce gli accessi da parte dei client alle varie applicazioni incapsulate, indirizzando le richieste, ed eventuali dati, al giusto destinatario. Inoltre è in grado di espletare alcune funzioni di gestione quali ad esempio istanziare nuovi componenti al fine di migliorare le prestazioni dell'intera applicazione o di esportare ulteriori servizi.

Il Container rappresenta l'interfaccia fra l'applicazione interna ed il mondo CCM. Di fatto, nel mondo CCM, un'Application Server non è in grado di accedere direttamente ad un'applicazione racchiusa all'interno di un componente, in quanto tutti gli accessi sono mediati dal Container che invoca i metodi dell'applicazione per l'espletamento del servizio esportato.

### Abstract Component Model

L'*Abstract Component Model* fornisce la specifica del componente CCM, in quanto individua le caratteristiche dell'applicazione che viene integrata dal CCM stesso. Di fatto il componente incapsula un'applicazione, ponendo però dei vincoli per quel che riguarda le interfacce e il tipo di accesso ai metodi dell'applicazione stessa. In figura 3.4 è mostrata la struttura di un'applicazione integrata in un componente CCM, con particolare interesse per i meccanismi di interazione adottati. CCM estende lo standard CORBA mettendo a disposizione delle nuove interfacce che vengono classificate in sei diverse categorie:

**Facets** queste interfacce, che adottano meccanismi di comunicazione sincroni, sono messe a disposizione verso i componenti esterni per l'invocazione delle funzioni che l'applicazione, incapsulata dal CCM, fornisce. Le Facets, che sono identificate da nomi univoci in quanto un componente ne può esporre più di una, rappresentano una vista del componente stesso, e forniscono un insieme di regole, che il client deve rispettare, per accedere ai servizi offerti. Le diverse Facets dello stesso componente sono indipendenti l'una dall'altra.

**Receptacles** queste interfacce servono per connettere il componente ad un altro componente CCM da cui esso dipende per il suo corretto funzionamento. Anche queste adottano meccanismi di comunicazione sincroni e ve ne possono essere più di una per ogni componente. Le interfacce Receptacles, che hanno nomi distinti fra loro, rappresentano i punti di connessione fra i componenti CCM. Di fatto, sono i legami su cui si basano le applicazioni CCM complesse. Tali interfacce sono indipendenti dal servizio richiesto attraverso di esse, possono essere modificate staticamente in fase di inizializzazione, e possono essere gestite dinamicamente a tempo di esecuzione.

**Event Source** queste interfacce permettono di inviare eventi, generati all'interno del componente, al mondo esterno, adottando meccanismi di comunicazione

asincroni. A seconda del numero di partner della comunicazione si possono distinguere in *publishes*, quando si hanno più destinatari del messaggio, ed *emits*, quando si ha un singolo destinatario. I canali di comunicazione utilizzati da queste interfacce vengono gestiti dal Container.

**Event Sink** queste interfacce, che adottano meccanismi di comunicazione asincroni, permettono di ricevere eventi esterni da un singolo componente o, attraverso un canale condiviso, da più componenti. Per le interfacce Event Sink non si ha la distinzione a seconda del numero di partner della comunicazione. Gli eventi in ingresso possono essere generati direttamente da altri oggetti CCM o da un *Notification Service*.

**Attributes** queste interfacce permettono la configurazione statica o dinamica degli attributi del componente. Le Attributes sono la chiave fondamentale per poter far sì che i componenti possano essere riutilizzati. Di fatto, queste permettono la configurazione del componente stesso attraverso l'impostazione di parametri che ne modificano: il comportamento, la modalità di esecuzione, l'assegnazione di risorse, ed altro. Gli attributi possono essere configurati sia a tempo di implementazione del componente sia a tempo di esecuzione. Inoltre, durante l'esecuzione, queste interfacce sono in grado di generare eccezioni.

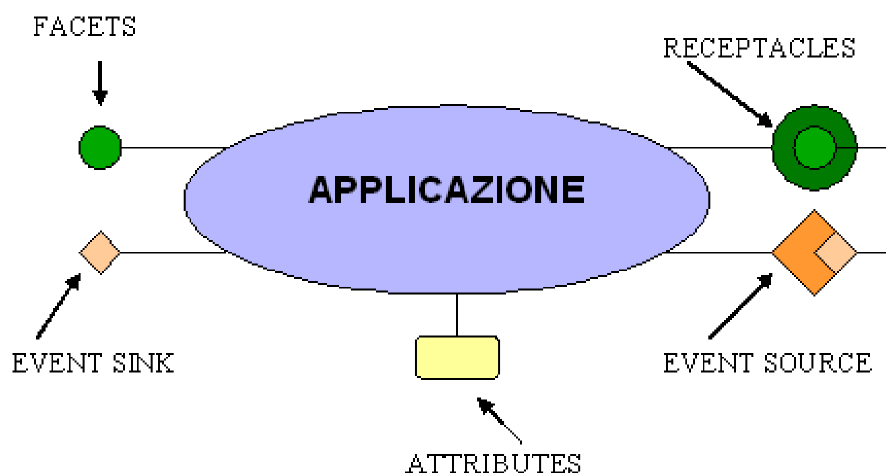



Figura 3.4: *Abstract Component Model*: nuove interfacce introdotte.

**Homes** queste interfacce permettono di accedere alle istanze dell'applicazione incapsulata dal CCM (vedi figura 3.5) e sono basate su due *design pattern* [30]: *factory*[2] e *finder*[3]. Una Home è definita per creare e richiamare uno specifico tipo di componente, e può solamente gestire istanze di questo tipo. Tuttavia, è possibile definire differenti tipi di Home per un singolo tipo di componente. Tali interfacce non identificano il componente ma sono solo dei riferimenti alle applicazioni incapsulate in modo da renderne omogenea la gestione e l'accesso all'interno. Tipicamente le interfacce Home mettono a disposizione un limitato numero di operazioni, necessario per gestire il ciclo di vita di un'applicazione.

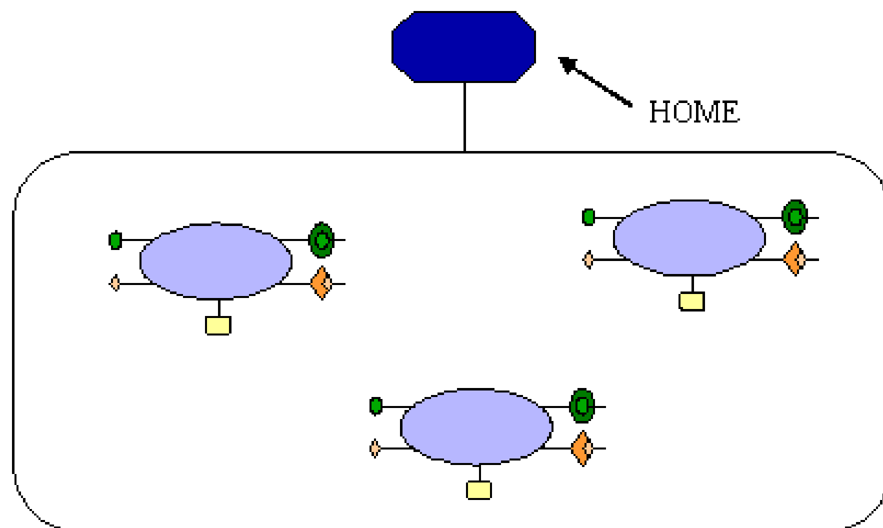



Figura 3.5: Descrizione dell'interfaccia *Home*.

### Component Implementation Framework

Come visto, il CCM definisce ed implementa molte interfacce per supportare la composizione fra componenti e le funzionalità di ogni singola applicazione. Molte imple-

[2] Il pattern Factory definisce un'interfaccia per creare un oggetto, ma lascia decidere alle interfacce quali classi istanziare. Il Factory Model permette a una classe di rinviare l'istanziazione alle sottoclassi.

[3] Il pattern Finder permette di mantenere le associazioni tra i vari tipi di componenti e le rispettive interfacce (*Component Factory*) che possono quindi essere ottenute dal client che ne fa richiesta.

mentazioni di queste interfacce possono essere generate in automatico, ed ancora, la gestione del ciclo di vita e la persistenza dello stato del componente, possono essere generalizzate, al fine di poter riutilizzare le varie applicazioni incapsulate in differenti contesti. Il *Component Implementation Framework* (CIF) è lo strumento CCM che permette di implementare tutto ciò in maniera automatica, mettendo a disposizione delle API (*Application Programming Interface*), che semplificano l'implementazione del componente stesso.

Il CIF è basato sul CIDL (*Component Implementation Definition Language*), che è un linguaggio di definizione per la composizione dei componenti, la gestione della persistenza dello stato del componente, la generazione degli *skeleton*[4] di esecuzione dei componenti. La composizione di componenti definita dal CIDL si basa su alcune caratteristiche del componente stesso, quali: tipo del ciclo di vita (*Service*, *Session*, *Process*, *Entity*, che verranno descritti nella sezione successiva), tipo delle interfacce Home di gestione dell'applicazione interna, nome dello skeleton di esecuzione generato per il componente. La gestione dello stato del componente fatta dal CIDL è basata sul PSDL (*Persistent State Definition Language*), definito dalle versioni precedenti di CORBA. Di fatto, il PSDL permette di dichiarare variabili di stato, per ogni componente, che possono essere usate attraverso i meccanismi del PPS (*Persistent State Service*).

### Container Programming Model

Il CIF genera gli skeleton d'esecuzione dei componenti implementando le specifiche CIDL. Questo fa sì che il comportamento dell'applicazione incapsulata dal CCM, come l'introspezione delle interfacce Facet, l'attivazione, la sospensione (interfacce Home), e la gestione della persistenza dello stato, possano essere eseguite automaticamente. L'Application Server è l'entità che si occupa di eseguire le operazioni appena descritte. Come detto in precedenza, l'Application Server, non riferisce di-

[4]Funzioni che consentono all'ORB di identificare il componente e quindi gestire le comunicazioni da/per il componente stesso.

rettamente le applicazioni incapsulate dal CCM, ma si interfaccia con i Container.

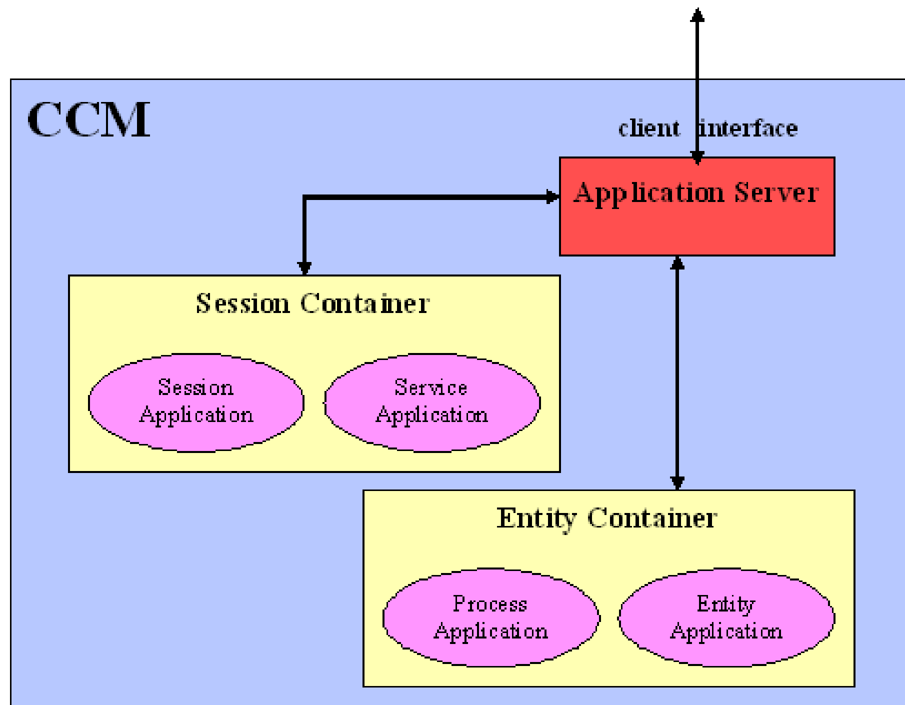



Figura 3.6: Architettura definita dal *Container Programming Model*.

Si possono distinguere le applicazioni incapsulate dai componenti, a seconda del ciclo di vita che le caratterizza:

- **Service**,
- **Session**,
- **Process**,
- **Entity**.

Le prime due categorie indicano che il ciclo di vita dell'applicazione è temporaneo, più precisamente: *Service* indica che l'applicazione viene istanziata solo nel momento in cui viene invocata una sua funzione, *Session* indica che l'applicazione è necessaria soltanto durante la sessione di uno specifico client. Le altre due indicano, invece, delle applicazioni che persistono durante tutto il tempo di esecuzione del componente CCM, di cui fanno parte.

A seconda della categoria dell'applicazione possiamo individuare diversi tipi di Container come mostrato in tabella 3.1.

| **Categoria dell'applicazione incapsulata nel componente CCM** | ***Tipo di Container*** |
|---|---|
| Service | *Session* |
| Session | *Session* |
| Process | *Entity* |
| Entity | *Entity* |

Tabella 3.1: Associazioni tra applicazione e tipo di *Container*.

Il *CCM Container Programming Model* definisce un insieme di API che semplificano lo sviluppo e la gestione di una applicazione incapsulata all'interno di un componente. In pratica, il Container, si interfaccia all'applicazione stessa invocando, in maniera automatica, l'esecuzione di operazioni quali:

- attivazione e sospensione dell'applicazione. Questo fa sì che non vengano utilizzate inutilmente risorse del sistema, come ad esempio la memoria;
- invio della richiesta di un client verso l'appropriata applicazione;
- ridirezione degli eventi esterni all'applicazione che è in grado di occuparsi della loro gestione.

Le API dell'architettura del Container Programming Model, che esportano le funzionalità degli oggetti CORBA, sono divise in due gruppi:

1. *External API*, utilizzate dal client per accedere alle interfacce esterne dell'applicazione, come Facet, Receptacle e Home. La definizione di questo tipo di API è scritta nel linguaggio IDL e, in particolare, il tipo e l'introspezione delle interfacce sono definiti dall'interfaccia Home;
2. *Container API*, API locali che permettono l'invocazione di metodi del Container da parte dell'Application Server e viceversa. Nel primo caso (*Internal*

*Interface*) si tratta di richieste indispensabili per il funzionamento corretto di un servizio messo a disposizione dall'applicazione incapsulata dal CCM. Nel secondo caso (*Callback Interface*) le operazioni consentite sono l'attivazione o la sospensione dell'applicazione, la notifica al *servant POA*[5], ed il ripristino dello stato.

## 3.4 I component framework ad alta performance

### 3.4.1 Lo standard CCA: XCat

#### Il modello CCA

L'architettura a componenti *CCA* (*Common Component Architecture*) [38, 5] è stata sviluppata come progetto tra laboratori di ricerca e istituzioni accademiche con lo scopo di definire un'architettura comune e standard con la quale costruire applicazioni scientifiche a larga scala, a partire da componenti software ben testati. Inizialmente, agli inizi del 1999, lo standard era stato pensato per sviluppare applicazioni di tipo *intensive* da far girare su super-computer paralleli. Successivamente, con la diffusione delle griglie computazionali, sono state realizzate implementazioni di CCA che tenessero anche conto di questo tipo di ambiente (XCat [31] ne è un esempio). CCA è stato pensato per superare i limiti nello sviluppo di applicazioni ad alte prestazioni delle architetture a componenti esistenti (CORBA, DOM, JavaBeans, ecc.). Per ottenere questo obiettivo, sono stati considerati vari requisiti che devono essere presi in considerazione da un ambiente di questo tipo:

**Caratteristiche dei componenti** I componenti sono pensati per generiche applicazioni, ad alte prestazioni, quindi possiamo avere una moltitudine di componenti con caratteristiche e paradigmi di parallelismo diversi.

**Eterogeneità** Le applicazioni devono poter essere costruite componendo tra loro

[5] Il POA (*Portable Object Adapter*) è responsabile dell'associazione fra gli oggetti CORBA e la loro implementazione in un linguaggio di programmazione (*servant*).

componenti scritti in linguaggi diversi e in esecuzione su architetture (hardware, SO, ambiente a run-time) diverse.

**Componenti locali e remoti** Utilizzare un servizio di componente locale, cioè in esecuzione all'interno dello stesso spazio di indirizzamento dell'applicazione, deve costare solo la chiamata ad una funzione. Nel caso di componenti remoti si devono invece sfruttare protocolli *0-copy* per il trasferimento dei dati e, ove possibile, anche sfruttare ottimizzazioni o caratteristiche particolari offerte dalla rete sottostante.

**Alte prestazioni** L'ambiente deve poter essere il più possibile efficiente, quindi, è necessario realizzare protocolli di comunicazione ottimizzati, sfruttando caratteristiche dell'architettura e del sistema operativo sottostante, evitando copie extra dei dati, sincronizzazioni inutili e qualsiasi altra cosa 'costosa' che non sia strettamente necessaria.

**Integrazione** I componenti devo essere direttamente utilizzabili in ambienti a run-time CCA differenti, in esecuzione su architetture eterogenee.

La specifica CCA definisce uno standard poco stringente, difatti definisce il comportamento di massima di un componente e lascia agli sviluppatori delle singole implementazioni la libertà di realizzare generiche funzionalità nella maniera che ritengono più opportuna.

Il concetto più importante fissato dalla specifica CCA è il modo con il quale i componenti possono comunicare ed interagire tra loro. Lo standard definisce il concetto di composizione di componenti mediante l'uso delle cosiddette *porte*. Queste non sono altro che la descrizione di una interfaccia che rappresenta un certo insieme di operazioni disponibili su un particolare oggetto. L'interfaccia definita su una porta dovrà poi essere utilizzata secondo il paradigma *uses/provides* a seconda che il componente considerato voglia utilizzare o voglia fornire un determinato servizio. Il modello descritto, piuttosto flessibile, permette con facilità di costruire *metacompo-*

*nenti*, ovvero componenti di componenti, mediante la composizione di componenti primitivi.

Lo standard CCA è implementato in diversi component framework che offrono modelli di programmazione differenti: CCAffeine [39] è focalizzato sulla costruzione di applicazioni parallele *SPMD* (*Single Program, Multiple Data*), viste come composizione locale di componenti; XCat [31], invece, è più adatto per applicazioni distribuite su griglie, in quanto si possono connettere componenti localizzati su macchine remote.

### La libreria a componenti XCat

*XCat* [31] è una implementazione dello standard CCA ideata e sviluppata dal team di Dennis Gannon della Indiana University (USA) per la realizzazione di applicazioni ad alte prestazioni in ambiente distribuito come le griglie computazionali.

XCat, oltre a supportare un ambiente di calcolo distribuito, omogeneo ed eterogeneo, permette di scrivere applicazioni eseguibili all'interno di una griglia computazionale realizzata mediante il software *Globus* [40]. L'architettura di XCat, basata sul modello dei *Web Service* [24], è articolata su uno stack a cinque livelli:

**Framework** Rappresenta lo strato più ad alto livello del modello e fornisce tutti i servizi avanzati, necessari per costruire ed eseguire l'applicazione. In XCat questo è rappresentato dalla realizzazione della specifica CCM e dai vari servizi primitivi (descritti successivamente) offerti agli applicativi.

**Discovery** Questo livello è necessario per pubblicare e recuperare i servizi offerti dagli applicativi ed utilizza un registro (di tipo *LDAP*, *Lightweight Directory Access Protocol*) per tenere traccia di tutte le funzionalità da esportare.

**Descrizione** Rappresenta il livello utilizzato per descrivere (tramite WSDL, vedi sez. 3.4.3) i servizi disponibili: interfacce, protocolli di trasporto supportati, ecc.

**Messaggio** Questo livello viene utilizzato per codificare/decodificare i dati da trasmettere sulla rete. In XCat questo compito è svolto da *XSOAP*, una libreria che realizza il modello RMI di Java sfruttando *SOAP* (*Simple Object Access Protocol*, vedi sez. 3.4.3) come meccanismo di trasporto.

**Trasporto** Questo livello indica la tecnologia utilizzata per trasportare messaggi all'interno di una applicazione parallela e distribuita. Come già detto XCat utilizza SOAP come protocollo di comunicazione.

I servizi primitivi, offerti alle applicazioni da XCat, sono:

- *CreationService*: fa parte della libreria CCA e mette a disposizione un insieme di funzionalità per istanziare e rimuovere componenti;
- *ConnectionService*: connette dinamicamente le porte di tipo *uses* di un componente con le porte di tipo *provides* di un altro;
- *NamingService*: realizza un registro in cui memorizzare le informazioni relative ai componenti istanziati che rendono disponibili le proprie funzionalità.

In XCat l'unità minima di computazione è rappresentata da un componente, il quale ha il compito di informare il 'mondo esterno' delle funzionalità che esporta e di cui ha bisogno. Per collegare e coordinare i componenti, vengono create le cosiddette applicazioni di controllo secondo due diverse possibili soluzioni:

1. Il primo modello usa direttamente i servizi messi a disposizione da XCat, attraverso una semplice applicazione di controllo Java, che usa direttamente le funzionalità messe a disposizione dai servizi base appena descritti (vedi fig. 3.7).

2. Nel secondo modello l'applicazione di controllo è rappresentata da uno script di tipo Jython[6]. L'oggetto *Application Manager Service* è un interprete Jython che ha il compito di eseguire lo script che descrive l'applicazione da modellare

[6] Implementazione di Python.

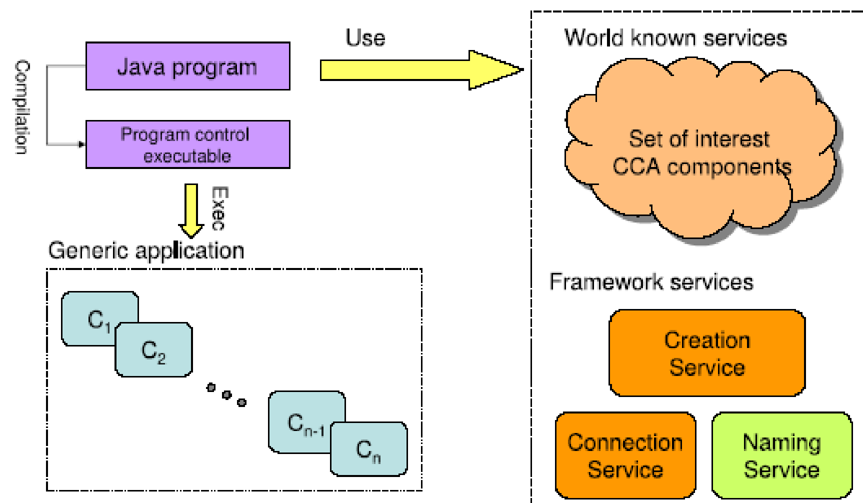



Figura 3.7: Modello di programmazione XCat che usa un programma di controllo Java.

accedendo sempre ai servizi di XCat (vedi fig. 3.8). In questo caso, però, può essere eseguito più di uno script che può andare ad incidere sul comportamento dell'applicazione da eseguire.

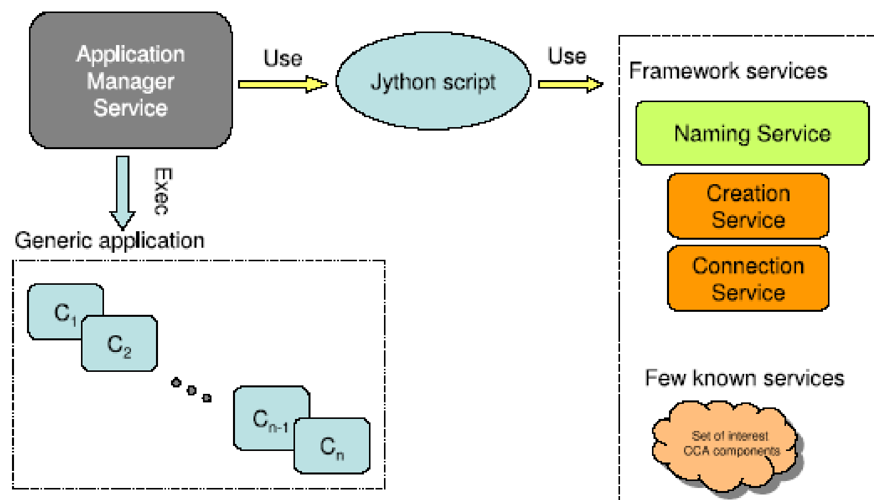



Figura 3.8: Modello di programmazione XCat che usa un programma di controllo Jython.

Il supporto alla griglia dell'ambiente XCat risulta inefficiente [13] ed ancora piuttosto limitato, poiché, fra i servizi descritti, non ne esiste alcuno che consenta di

avere un controllo esplicito sull'applicazione durante la sua esecuzione. Inoltre le interfacce delle porte possono essere specificate solo attraverso l'uso di classi Java, che penalizza l'uso di alcuni tipi di dato (e.g. numeri complessi, array dinamici multidimensionali) per applicazioni scientifiche e limita la possibilità di utilizzare componenti CCA implementati in linguaggi di programmazione diversi da Java.

### 3.4.2 Lo standard CCA: CCAffeine

CCAffeine [39, 8] è un'implementazione delle specifiche CCA orientata alle applicazioni SPMD (*Single Program Multiple Data*), che può assemblare componenti eterogenei ed internamente paralleli. Le comunicazioni fra i componenti sono gestite del framework che è responsabile della conversione dei dati se i componenti sono eterogenei (cioè eseguiti su macchine diverse o implementati in linguaggi diversi). Le comunicazioni interne al componente, invece, sono sotto la responsabilità dello sviluppatore che può utilizzare librerie di sua scelta (MPI, PVM, RPC, memoria condivisa).

Un componente CCAffeine esporta una o più porte di input e output (*uses* e *provides*), identificate dal loro tipo. L'ereditarietà e la specializzazione dei componenti permettono allo sviluppatore di caratterizzarne la definizione e la semantica. Porte distinte possono anche offrire un differente punto di vista dello stesso servizio. In CCAffeine un componente è inteso come astrazione di una procedura, quindi i componenti sono privi di stato interno. Le connessioni tra i componenti, però, non possono essere cambiate durante l'esecuzione dell'applicazione.

Le interfacce di un componente CCAffeine sono implementate utilizzando *Babel*, un *Interface Description Language* (IDL) orientato alla programmazione scientifica ad alta performance. Ciò fa sì che il framework possa essere utilizzato per comporre componenti CCA scritti in linguaggi di programmazione diversi.

Nonostante le caratteristiche di riuso e interoperabilità proprie di CCAffeine, alcuni difetti non lo rendono adatto alle griglie computazionali ed all'alta performance [13]. Di fatto manca un supporto per l'utilizzo di componenti remoti e l'alta

performance è gestita dallo sviluppatore, che deve implementare e schedulare le comunicazioni all'interno del componente.

### 3.4.3 Grid-Web Service

#### I Web Service

I *Web Service* [24] sono delle applicazioni autodescrittive che possono autonomamente interagire tramite internet per eseguire delle elaborazioni complesse. Le interfacce che espongono i Web Service possono essere trovate via internet e vengono definite nel linguaggio standard *WSDL* (*Web Services Description Language*). Per cercare un servizio fornito da un Web Service, è necessario utilizzare il repository *UDDI* (*Universal Description Discovery and Integration*). Mentre, per invocare tale servizio, è necessario utilizzare il protocollo XML standard *SOAP* (*Simple Object Access Protocol*) che si incarica della chiamata di procedura remota e dell'invio dei messaggi tra le applicazioni. I Web Service implementano un paradigma di calcolo distribuito che presenta dei vantaggi in termini di supporto industriale e di adesione agli standard utilizzati su internet.

#### Grid Service

I Web Service sono stati sviluppati per l'interazione di applicazioni commerciali, in ambiente distribuito, senza alcun obiettivo di performance.

I Grid Service [28, 27, 25], che ne rappresentano un'evoluzione, pongono invece l'obiettivo dell'alta performance alla base dello sviluppo di servizi distribuiti su griglia. Prima di entrare nel dettaglio di quello che è un Grid Service, occorre descrivere meglio due termini che ricorreranno nel seguito, ovvero *OGSA* e *OGSI*. L'*Open Grid Service Architecture* (OGSA) definisce le caratteristiche fondamentali che una griglia computazionale dovrebbe avere; definisce il modello di programmazione di una griglia; fornisce informazioni su come costruire un servizio orientato alla griglia ed infine precisa quali sono le componenti essenziali per costruire e distribuire un

valido prodotto su griglia (OGSA da una descrizione di una griglia computazionale dal punto di vista fisiologico). L'*Open Grid Service Infrastructure* (OGSI), utilizzando le tecnologie che stanno alla base delle griglie e dei Web Service, definisce i meccanismi per la creazione, la gestione e lo scambio di informazioni tra i Grid Service stessi. Le specifiche di OGSI estendono gli schemi WSDL e XML al fine di gestire le seguenti possibilità:

- Web Service con stato;
- estensione delle interfacce dei Web Service;
- notifica asincrona dei cambiamenti di stato;
- riferimenti ad istanze di servizi;
- collezione di istanze.

Nella OGSA, per definizione, un Grid Service è un Web Service, in cui è definito almeno il *portType GridService*, un'interfaccia che offre le primitive necessarie per individuare, creare dinamicamente e gestire un servizio. Tutti i Grid Service devono implementare questa interfaccia, condizione, in generale, non soddisfatta da un Web Service, nel quale le interfacce offerte dipendono dall'applicazione. Nell'architettura OGSA viene fatta una distinzione tra la descrizione di un Grid Service, in cui si prescrivono le modalità con cui si interagisce con il servizio, e la sua istanza, in cui si implementano le interfacce definite nella descrizione. Le istanze di Grid Service possono essere persistenti o transitorie ed essere, opzionalmente, accompagnate da elementi *serviceData* (*SDE Service Data Element*), informazioni aggiuntive relative all'istanza (metadati), o a proprietà dell'istanza stessa (informazioni di stato), che possono variare a run-time.

Il portType GridService implementa un ben definito insieme di operazioni:

- *FindServiceData*: accede agli elementi serviceData di una istanza;

- *RequestTermination*: accede e modifica l'istante in cui l'istanza verrà terminata;
- *Destroy*: richiede la distruzione di una istanza del servizio.

Nella definizione OGSI, sono state introdotte delle estensioni WSDL che permettono di implementare i Grid Service:

- *ServiceData*: proprietà del servizio osservabili dall'esterno;
- *ServiceDataDescription*: descrizione formale degli elementi ServiceData;
- convenzione sui nomi dei portType;
- *GSH*: *Grid Service Handle*, si tratta di un *URI* (*Uniform Resource Identifier* RFC2396), nella pratica una URL della forma *http://nomehost/nomedocumento*, che non permette di comunicare con il servizio ma solo di individuarlo in modo univoco sulla rete. Per poter comunicare con il servizio, il GSH deve essere convertito in *GSR*;
- *GSR*: *Grid Service Reference*, fornisce al client le informazioni necessarie per connettersi e dialogare con il servizio di griglia: protocolli, indirizzi, ecc. Inoltre può scadere o perdere di validità.

Attualmente vi sono nuovi sviluppi in termini di Grid Service che hanno messo in discussione l'utilizzo di alcuni standand e che quindi non permettono di ottenere una valutazione complessiva sul modello a componenti per alte prestazioni proposto.

### 3.4.4 GridCCM

*GridCCM* [21, 50] estende il CORBA Component Model con il concetto dei componenti paralleli. Il suo obiettivo è quello di permettere un'integrazione efficiente di codice parallelo (ad alta performance) in componenti GridCCM, tramite poche modifiche al codice parallelo stesso e senza introdurre profondi cambiamenti al modello a componenti di CORBA: il CORBA Interface Definition Language (IDL), infatti,

non viene modificato, in modo che i componenti paralleli rimangano interoperabili con i componenti standard sequenziali. Attualmente il modello è limitato ad integrare solamente codice SPMD (Single Program Multiple Data) per due motivi: molte applicazioni parallele sono di tipo SPMD e tale modello è facilmente gestibile, inoltre il codice SPMD, incapsulato in componenti GridCCM, usa la libreria MPI, per le comunicazioni inter-process, ed invece CORBA, per le comunicazioni con altri componenti (vedi fig. 3.9). Per evitare colli di bottiglia, tutti i processi di un com-

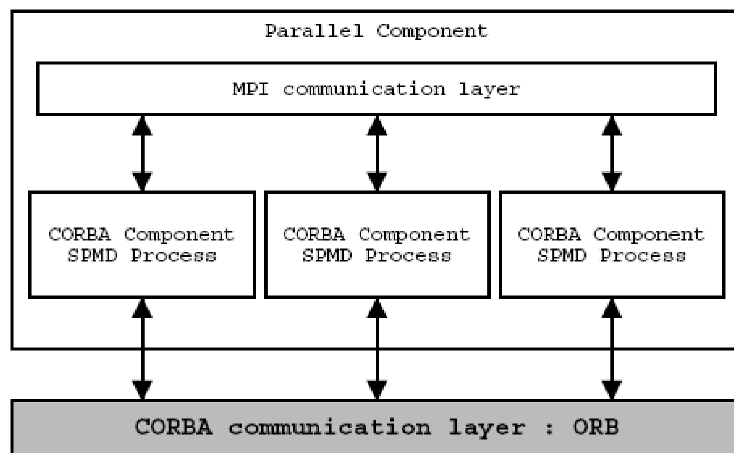



Figura 3.9: Componente parallelo introdotto nel GridCCM.

ponente parallelo partecipano alle comunicazioni tra componenti GridCCM [50]. I nodi di un componente parallelo non sono direttamente esposti ad altri componenti, poiché esistono dei proxy che li nascondono.

Per introdurre il parallelismo, e quindi l'alta performance, in un componente senza richiedere nessuna modifica all'ORB, si è introdotto un livello software, generato da un compilatore GridCCM specifico, tra il codice utente (client e server) e lo stub (vedi fig. 3.10). Una chiamata ad una operazione parallela, implementata da un componente parallelo, è intercettata da questo nuovo livello che invia i dati dai nodi client ai nodi server. Esso può inoltre eseguire una ridistribuzione dei dati sul lato cliente, sul lato server o durante la comunicazione tra client e server, in base a dei vincoli che possono includere per esempio una maggiore efficienza. Il compilatore che genera questo livello di gestione parallela utilizza due file: una descrizione IDL

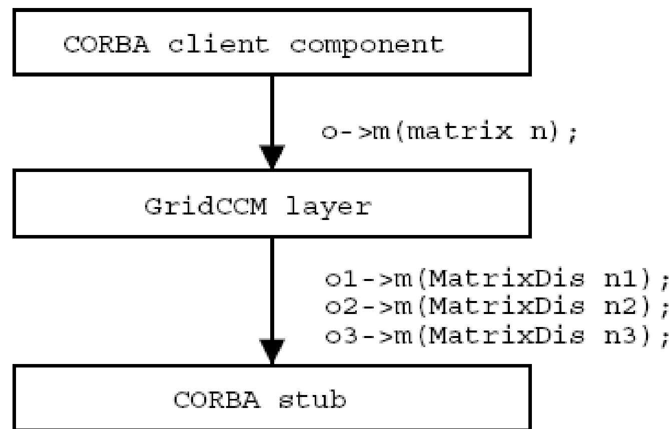



Figura 3.10: Il livello GridCCM intercetta e traduce l'invocazione remota di un metodo.

del componente ed una descrizione XML del parallelismo all'interno del componente. Per avere un livello trasparente viene generata una nuova interfaccia IDL, che deriva dall'interfaccia originale ma viene utilizzata internamente dal livello GridCCM che realmente invoca le operazioni sul lato server. L'interfaccia IDL originale viene utilizzata tra il codice utente ed il livello GridCCM, sul lato client e sul lato server.

### 3.4.5 Conclusioni

Nelle precedenti sezioni sono state analizzate le caratteristiche salienti di alcuni progetti ben conosciuti in letteratura. Tali progetti sono orientati allo sviluppo di ambienti di programmazione component-based e ad alta performance come possibile approccio alle problematiche della programmazione su griglia. Dai risultati di tale analisi, si evince però che ognuno di questi progetti è incentrato su un particolare aspetto delle problematiche della programmazione grid-aware, quindi non risulta adatto, in generale, per lo sviluppo di questo tipo di applicazioni. XCat, se da un lato permette di modellare in modo naturale e semplificato applicazioni anche di una certa complessità, fornendo all'utente una serie di servizi ad alto livello, dall'altro è limitato dal fatto che le interfacce dei componenti possono essere specificate solamente attraverso classi Java. Questo limita l'uso di alcuni tipi di dato complessi utili per le applicazioni scientifiche, e la possibilità di utilizzare componenti CCA imple-

mentati in linguaggi di programmazione diversi da Java. CCAffeine, invece, affronta il problema dell'interoperabilità dei componenti, che quindi possono essere scritti in diversi linguaggi di programmazione, ma lascia la responsabilità dell'alta performance al programmatore che deve implementarla attraverso librerie di sua scelta (MPI, PVM, RPC, memoria condivisa). GridCCM, inoltre, affronta il problema dell'ottimizzazione delle comunicazioni ma non quello dell'adattività dell'applicazione in un ambiente dinamico come quello della griglia. Per quanto riguarda i Grid Service infine, non è possibile giungere a conclusioni definitive in quanto ulteriori sviluppi sono ancora in atto.

# Capitolo 4

# Integrazione ASSIST-CCM

Sommario

In questo capitolo verranno discussi gli scopi e i problemi dell'integrazione di moduli ASSIST in componenti CCM e gli strumenti utilizzati per risolvere tali problemi.

## 4.1 Scopo dell'integrazione

Come detto nel capitolo 1, in Italia è attivo il progetto di ricerca di base MIUR-FIRB *Grid.it*: *Enabling Platforms for High-performance Computational Grids orientated to Scalable Virtual Organizations*. In particolare il gruppo di lavoro *Workpackage WP8*, *Ambienti di Programmazione ad Alta Performance basati su Componenti*, ha proposto un modello sperimentale per la definizione di un ambiente di sviluppo component-based per applicazioni grid-aware ad alta performance. L'orientamento del gruppo è di far evolvere ASSIST in un ambiente component-based [3], che supporti le peculiarità della griglia. Come primo passo si è deciso di sperimentare l'integrazione di ASSIST con un modello a componenti esistente (non ad alta performance) attraverso l'incapsulamento di moduli paralleli ASSIST in componenti, valutarne l'impatto e comprendere quali siano i meccanismi necessari da includere nel modello a componenti, per implementare applicazioni ad alta performance. Ad esempio, potrebbe essere necessaria nel modello una composizione più esplicita dei componenti, che enfatizzi l'adattività a run-time della struttura e lo sfruttamento

di questa struttura gerarchica per gestire la riconfigurazione dell'applicazione. Alla fine, attraverso i risultati della sperimentazione, si potrebbe arrivare ad una integrazione del modello a componenti che possa far coesistere componenti ASSIST paralleli e componenti già esistenti, in una applicazione parallela e distribuita su griglia ad alta performance.

Il modello a componenti scelto per la prima fase della sperimentazione è il *CORBA Component Model* [35, 54] (*CCM*), poiché rispetto ad altri *component framework* non ad alta performance (e.g. *COM/DCOM/COM+* [41] della *Microsoft*, *EJB* [46] della *Sun*), è uno standard ben affermato e svincolato da uno specifico produttore, e fornisce la nozione di componente più completa e sofisticata [6]. Tale sperimentazione ha avuto inizio con l'individuazione di problemi e potenzialità legati a performance ed espressività del modello a componenti CCM rispetto al modello ASSIST. Le fasi dell'integrazione si possono riassumere in:

- integrazione fra ASSIST e CORBA;
- adattamento dell'integrazione ASSIST-CORBA, ovvero uso di un modulo ASSIST come componente CCM;
- estensione di detta integrazione a grafi ASSIST più generali, con più punti di ingresso/uscita;
- adattamento dell'integrazione ad un singolo parmod, in questa fase vengono aggiunte al componente CCM le interfacce di controllo e configurazione del parmod.

Inizialmente è stato necessario determinare gli strumenti su cui basare l'integrazione fra ASSIST e CCM, e la scelta si è orientata verso *OpenCCM*[1] [36], in quanto è compatibile con diverse implementazioni dell'ORB di CORBA [47] (*OpenORB*, *BES*, *JacORB*, *ORBacus*) e supporta il modello CCM per i passi successivi dell'integrazione. La piattaforma OpenCCM fornisce:

[1]Implementazione *open source* del CORBA Component Model specificato dall'OMG.

- il supporto IDL per la specifica delle interfacce dei componenti;
- strumenti di compilazione per le specifiche IDL;
- la possibilità di implementare componenti e oggetti CORBA;
- la compilazione del codice sorgente Java;
- composizione della versione finale dell'applicazione.

OpenCCM, però, è stato implementato in linguaggio Java; questo ha reso necessario adattare le comunicazioni con i moduli ASSIST, interponendo dei moduli anch'essi implementati in linguaggio Java.

## 4.2 Integrazione tra ASSIST e CORBA

La strategia di integrazione è stata implementata attraverso un tool che esporta un programma ASSIST in ambienti standard CORBA [44]. La sola condizione imposta dal tool sul programma ASSIST, è stata quella di avere un unico stream di ingresso ed un unico stream d'uscita non connessi. Il tool *ASSIST2CORBA* esegue la composizione di programmi paralleli ASSIST secondo due diversi metodi supportati da CORBA:

- invocazione sincrona di un metodo;
- utilizzo del servizio degli eventi (*Event Service*), che consente di implementare un modello di comunicazione asincrono.

Le due opzioni hanno in comune la prima fase di esecuzione del tool (analisi lessicale e sintattica del programma ASSIST), durante la quale viene controllata la presenza di stream disconnessi ed il tipo di dato trasmesso su tali stream (tipo dello stream). In seguito, a seconda dei parametri impostati dall'utente, il tool esegue la conversione in base all'opzione scelta.

### 4.2.1 Architettura del meccanismo di invocazione sincrona dei metodi di un server

L'invocazione sincrona di un metodo di un server ASSIST, da parte di un client, locale o remoto, si compone di diverse fasi sia lato client, sia lato server:

- il client invoca il metodo ed attende la risposta dal server;
- il server accetta la richiesta del client, elabora l'input, ed invia la risposta.

Il tool si occupa di aggiungere al programma ASSIST le interfacce necessarie per far interoperare il client ed il server nel rispetto dello standard CORBA.

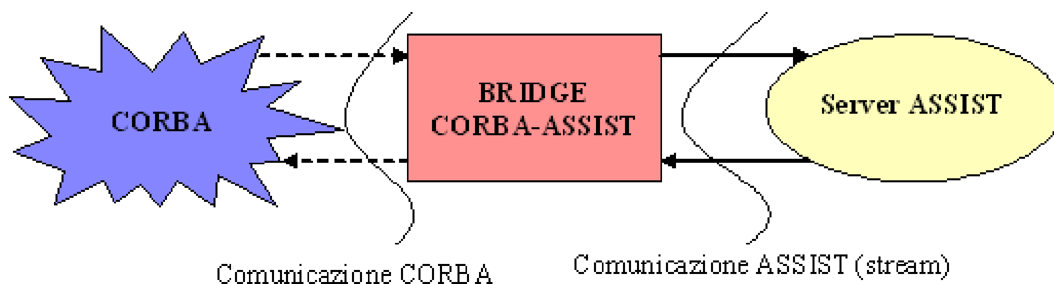



Figura 4.1: Grafo dell'invocazione sincrona di un metodo ASSIST.

La figura 4.1 mostra il grafo dell'invocazione sincrona di un metodo di un server ASSIST. Il modulo ASSIST, *Bridge CORBA-ASSIST*, generato automaticamente dal tool, è in grado di ricevere dati sia dallo stream ASSIST, sia da moduli CORBA, indipendentemente. Per fare questo, tale modulo, viene attivato nel momento in cui il programma ASSIST viene eseguito, e resta attivo finchè non viene esplicitamente terminato. In questo modo è stata eliminata la dipendenza del suo funzionamento dall'ingresso dei dati sullo stream ASSIST, ed è possibile gestire le comunicazioni con i moduli CORBA. Il funzionamento dell'invocazione sincrona di un metodo ASSIST, prevede una prima fase di inizializzazione, nella quale il modulo Bridge CORBA-ASSIST si connette al mondo CORBA e rende disponibili i servizi implementati dal server ASSIST. Dopodichè, tale modulo, rimane in attesa di eventuali

richieste. Nel momento in cui un modulo CORBA invia la propria richiesta al Bridge CORBA-ASSIST, questo spedisce il dato di input al server ASSIST, utilizzando i meccanismi su stream propri dell'ambiente ASSIST. Allo stesso modo, il modulo server ASSIST, dopo aver terminato la sua elaborazione, invia la risposta al Bridge CORBA-ASSIST. A questo punto, è quest'ultimo modulo che si occupa di inviare la risposta al client utilizzando i meccanismi CORBA, come esplicito ritorno del metodo invocato dal client.

### 4.2.2 Architettura del meccanismo ad eventi

Il meccanismo ad eventi (Event Service) consente di realizzare un sistema di comunicazione di eventi, tra oggetti distribuiti, in modo asincrono. Questo avviene attraverso l'utilizzo di un canale di trasmissione degli eventi (*Event Channel*), fornito dal meccanismo stesso. Il vantaggio principale dell'Event Service è dato dal fatto che, tramite la trasmissione di eventi, è possibile simulare l'invio di dati su stream, proprio dell'ambiente ASSIST, necessario per ottenere alte prestazioni in diverse classi di applicazioni. In questo modo è possibile connettere due programmi ASSIST, distribuiti geograficamente, senza dover modificare né i programmi stessi, né il supporto ASSIST. Lo schema di funzionamento del meccanismo ad eventi si basa sui concetti di *supplier* e *consumer* di eventi:

- il primo è un produttore che immette gli eventi in un Event Channel;
- il secondo è un consumatore che riceve gli eventi dall'Event Channel.

Nel caso specifico, il programma ASSIST, ha funzione sia di supplier sia di consumer: è consumer quando riceve i dati dal client per l'elaborazione, supplier quando restituisce i risultati dell'elaborazione. Per il corretto funzionamento, il tipo di evento in ingresso deve coincidere con il tipo dello stream in ingresso al programma ASSIST, mentre il tipo di evento in uscita deve coincidere con il tipo dello stream in uscita dal programma ASSIST.

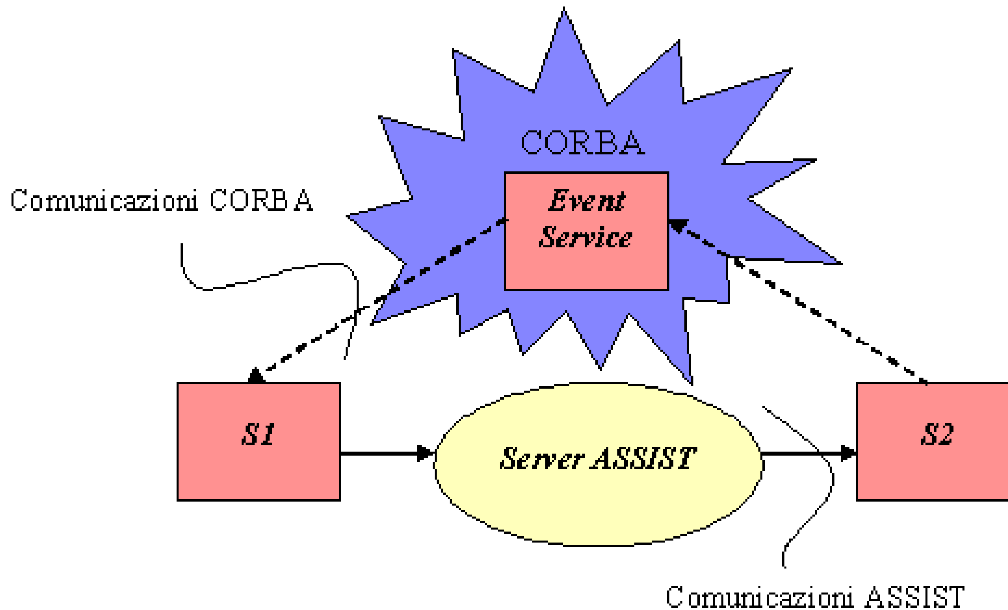



Figura 4.2: Grafo del meccanismo di comunicazione ad eventi.

In figura 4.2 è mostrato il meccanismo di comunicazione ad eventi. Di fatto il modulo ASSIST viene arricchito con due moduli sequenziali S1 e S2, che funzionano da interfaccia verso CORBA, in particolare verso l'Event Service. I due moduli, generati automaticamente dal tool ASSIST2CORBA, sono specializzati per gestire la conversione dei dati dal tipo dell'evento al tipo dello stream di ASSIST. Il funzionamento del meccanismo appena descritto avviene nel seguente modo:

- il client invia un evento che, preso in gestione dall'Event Channel, viene ridiretto verso S1. Questo modulo si occupa di estrarre i dati contenuti nell'evento ed inviarli, sullo stream di ingresso, al programma ASSIST;

- il programma ASSIST elabora il dato ricevuto ed invia il risultato sullo stream di uscita;

- il modulo S2, ricevuto il dato dallo stream, genera l'evento CORBA in uscita, del tipo corretto, e lo invia all'Event Service.

## 4.3 Integrazione tra ASSIST e CCM

L'integrazione fra ASSIST e CCM [12, 11] è stata basata sul lavoro svolto precedentemente per l'integrazione ASSIST-CORBA. Di fatto, come CCM estende CORBA, questa integrazione arricchisce gli strumenti messi a disposizione dal tool ASSIST2CORBA, dando origine al nuovo tool *ASSIST2CCM*. L'obiettivo del gruppo di ricerca, sviluppatore del tool, è stato quello di consentire l'esportazione di un generico programma ASSIST come componente CCM, in modo da poterne fare uso all'interno di un'applicazione, più complessa, basata sulla tecnologia CCM.

In generale un componente CCM che incapsula un modulo ASSIST esporta diverse interfacce:

- ingresso, queste interfacce utilizzano il meccanismo di comunicazione ad eventi fornito da CCM. Questo fa sì che possano essere implementate comunicazioni asincrone fra componenti, e che il modulo ASSIST incapsulato possa mantenere un funzionamento su stream per raggiungere un'alta performance all'interno del componente;

- uscita, anche queste interfacce, come le precedenti, utilizzano un meccanismo di comunicazione ad eventi;

- non funzionali, queste interfacce non interessano il calcolo vero e proprio eseguito dal modulo interno al componente, ma forniscono degli strumenti per la configurazione dei parametri che modificano la performance del componente stesso, quali ad esempio il cambiamento del grado di parallelismo del modulo ASSIST.

Da una visione ad alto livello dell'integrazione fra ASSIST e CCM, il tool ASSIST2CCM associa ad ogni stream in ingresso ed uscita al modulo ASSIST due thread che si occupano di trasformare i dati rendendoli compatibili con i due ambienti (ASSIST e OpenCCM), a seconda del contesto di ingresso o di uscita. Inoltre il tool si occupa della generazione e della compilazione degli IDL, degli stub e degli skeleton di

ogni componente. In seguito verranno descritte in dettaglio le fasi dell'integrazione ed i risultati prodotti.

Come per CORBA, il lavoro di esportazione è stato fatto a partire dal programma ASSIST. La sola condizione imposta dal tool sul programma ASSIST, è stata quella di avere degli stream di ingresso o degli stream d'uscita non connessi. Il primo passo necessario per trasformare un generico programma ASSIST, in un componente CCM, è stato quello di collegare gli stream non connessi, del programma ASSIST, a due nuovi moduli ASSIST: *CCM_input* e *CCM_output*. In particolare, ogni stream di ingresso $i$, è stato collegato ad un modulo dedicato $CCM_input_i$, ed ogni stream di uscita $j$, è stato collegato ad un modulo dedicato $CCM_ouput_j$. Di conseguenza esiste una corrispondenza fra il numero degli stream ed il numero dei moduli CCM_input, CCM_output. I nuovi moduli introdotti sono necessari per lo scambio dei dati tra il componente CCM ed il programma ASSIST da esso incapsulato. Il passaggio dei dati, codificati con l'utilizzo del *Common Data Representation*[2] (CDR), tra il componente CCM (Java) e il modulo ASSIST (C++), avviene tramite *socket*. Inoltre, sono presenti dei metodi $Push_i$ (uno per ogni modulo CCM_input$_i$) e $Run_j$ (uno per ogni modulo CCM_output$_j$) che hanno il compito di inviare/ricevere i dati ai/dai moduli ASSIST via socket, e ricevere/inviare i dati da/ad altri componenti CCM, utilizzando il meccanismo Event Service proprio di CCM. OpenCCM fornisce i due metodi (Push e Run) appena descritti in un unico *thread* (Push) perchè implementa chiamate di tipo RPC. Ai fini dell'integrazione, invece, è stato necessario separare i due servizi in due thread differenti per simulare la comunicazione su stream (propria dell'ambiente ASSIST).

Per descrivere l'implementazione della connessione fra il componente CCM ed ASSIST, è necessario descrivere:

- la connessione lato CCM, ossia i thread del componente CCM che gestiscono le comunicazioni con il programma ASSIST;

[2] La conversione CDR è lo standard di CORBA per il passaggio dei dati anche tra ambienti non omogenei.

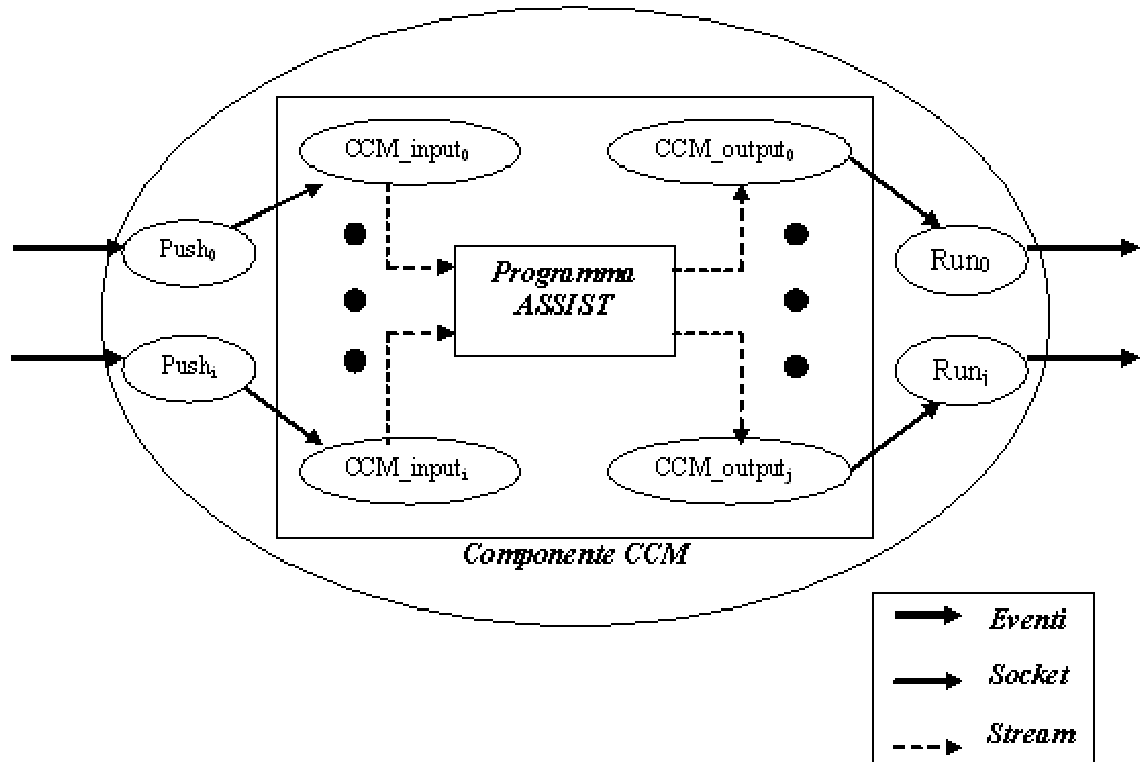



Figura 4.3: Architettura dell'integrazione tra ASSIST e CCM.

- la connessione lato ASSIST, ossia i moduli sequenziali aggiunti al programma ASSIST che hanno il compito di comunicare con i thread del componente CCM;

- la comunicazione fra CCM e ASSIST mediante l'utilizzo della codifica CDR, ossia la classe *CDRServer* (scritta in linguaggio Java), e la classe *CDR_IO* (scritta in linguaggio C++), che implementano i metodi per connettersi, inviare e ricevere dati codificati nella rappresentazione CDR.

### 4.3.1 La connessione lato CCM

Prima di procedere con la descrizione della connessione tra CCM e ASSIST lato CCM, è necessario richiamare l'attenzione del lettore sui metodi implementati ed utilizzati dai thread del componente CCM. Come detto, per garantire l'interoperabilità fra il componente CCM ed ASSIST, ed tra il componente CCM ed altri componenti CCM, sono presenti i metodi Push e Run. Questi due metodi si distin-

guono per il fatto che il metodo per inviare un evento (Run), viene implementato dal supporto del CCM ed invocato dal componente nel momento in cui vi è necessità di inviare un evento. Il metodo per ricevere un evento (Push), viene attivato dal supporto quando viene ricevuto un evento da un componente esterno.

- Metodi Push.

  Il metodo $\text{Push}_i$ implementa il codice da eseguire nel momento in cui il componente riceve un evento $i$. L'implementazione di questo metodo deve essere eseguita dal programmatore CCM. Nel caso specifico l'operazione implementata è l'inoltro dell'evento ricevuto ad ASSIST, ossia all'*i-esimo* modulo CCM_input. Il metodo Push viene attivato dall'ORB nel momento in cui viene inviato un evento verso il componente. Ad ogni attivazione di questo metodo corrisponde l'attivazione di un thread, chiamato thread *implicito*.

- Metodi Run.

  Il metodo $\text{Run}_j$ viene invocato dal componente nel momento in cui vi è necessità di inviare un dato, del tipo associato a $\text{Run}_j$, ad un componente esterno. Nel caso specifico, questo metodo viene invocato dal modulo CCM_output, nel momento in cui il programma ASSIST ha terminato la sua computazione, ed invia i risultati ottenuti all'esterno. Ad ogni metodo $\text{Run}_j$ viene associato un thread, chiamato thread *esplicito*, per la fase di esecuzione.

La sincronizzazione fra i thread impliciti ed espliciti avviene nel momento in cui vengono eseguite delle azioni di configurazione del componente CCM o del programma ASSIST incapsulato. Ad esempio la terminazione del programma ASSIST implica la sincronizzazione dei thread.

### 4.3.2 La connessione lato ASSIST

I due moduli sequenziali CCM_input e CCM_output, sono necessari al programma ASSIST per interagire con il componente CCM, in cui viene incapsulato. Di fatto, i moduli CCM_input ricevono, tramite socket, i dati da un componente CCM esterno,

mentre i moduli CCM_output, inviano, sempre tramite socket, i dati ai componenti esterni. In entrambi i casi, i moduli, ricevono ed inviano i dati utilizzando i metodi messi a disposizione dalla classe CDR_IO, implementata dal gruppo di ricerca, in modo da rendere trasparente l'uso della codifica CDR.

Come già detto il numero dei moduli CCM_input è pari al numero degli stream di ingresso, non connessi, del programma ASSIST da incapsulare: ogni modulo CCM_input gestisce uno solo degli stream di ingresso al programma ASSIST. In particolare, l'esecuzione del modulo, prevede:

- connessione con il thread opportuno. Utilizzando i metodi messi a disposizione dalla classe CDR_IO, che vedremo in seguito, viene aperto il socket verso il thread che esegue il metodo $\text{Push}_i$;

- ciclo di ricezione dati dal metodo $\text{Push}_i$ (CCM), ed inoltro dei dati su stream (ASSIST). Fintantochè la connessione con il componente CCM non viene chiusa, il modulo $\text{CCM_input}_i$ riceve i dati e li inoltra al programma ASSIST, passandoli sullo stream di uscita del modulo stesso.

Analogamente, il numero dei moduli CCM_output, è pari al numero degli stream di uscita non connessi del programma ASSIST da incapsulare: di fatto, anche in questo caso, ogni modulo gestisce uno solo degli stream di uscita del programma ASSIST. In particolare, l'esecuzione del modulo, prevede:

- connessione con il thread CCM opportuno. Utilizzando i metodi messi a disposizione dalla classe CDR_IO, la prima volta che questo modulo viene attivato, viene aperta la connessione, tramite socket, con il thread del componente CCM corrispondente ($\text{Run}_j$). Tale connessione rimane aperta fino a che non viene terminata l'esecuzione del programma ASSIST;

- invio dei dati su stream al componente CCM ($\text{Run}_j$), tramite socket. Ogni volta che il modulo CCM_output riceve dei dati dallo stream di ingresso, tali

dati vengono inoltrati verso il thread del componente CCM collegato al modulo stesso.

### 4.3.3 La comunicazione tra CCM ed ASSIST mediante CDR

Per rendere possibile la comunicazione fra i thread CCM, scritti in linguaggio Java, ed i moduli CCM_input, CCM_output, scritti in linguaggio C++, è stata utilizzata la codifica CDR (Common Data Representation). Inoltre, per rendere l'utilizzo della codifica trasparente all'utente, sono state realizzate dal gruppo di ricerca di Architetture Parallele, le seguenti classi:

- CDRServer, scritta in linguaggio Java;
- CDR_IO, scritta in linguaggio C++.

La classe CDRServer ha due caratteristiche fondamentali:

- sul socket che gestisce è possibile inviare e ricevere un evento;
- la conversione CDR su socket viene garantita direttamente dall'ambiente CORBA.

La classe CDRServer mette a disposizione i metodi per pubblicare un socket, per accettare una nuova connessione e per inviare e ricevere dati sul socket.

Analogamente a quanto fatto dal lato Java, anche dal lato C++ è stata implementata una nuova classe CDR_IO, appositamente per gestire la spedizione e ricezione dei dati su socket utilizzando la codifica CDR. Questa classe utilizza la codifica CDR messa a disposizione dall'ambiente ASSIST, che, a sua volta, utilizza la rappresentazione CDR messa a disposizione direttamente dalla libreria *Adaptive Communication Environment* (ACE). La classe CDR_IO incapsula un CDR stream implementato su socket, ed, inoltre, mette a disposizione dei metodi per la connessione, l'invio e la ricezione dei dati.

### 4.3.4 Le interfacce non funzionali del componente CCM

Attraverso l'utilizzo di alcune interfacce non funzionali del componente CCM, cioè quelle interfacce non strettamente necessarie all'esecuzione del componente, è possibile configurare il componente stesso e gestire il programma ASSIST incapsulato al suo interno [3]. Le interfacce implementate sono:

1. una per ogni parmod, in modo da poter configurare il grado di parallelismo con cui il programma ASSIST deve essere eseguito;
2. una per attivare il programma ASSIST;
3. una per terminare il programma ASSIST;
4. una per terminare istantaneamente il programma ASSIST;
5. una per attivare il componente;
6. una per terminare il componente;
7. una per configurare il nome del componente.

Le interfacce dalla 1 alla 4, sono interfacce per la gestione del programma ASSIST incapsulato dal componente CCM. Combinando in modo opportuno le azioni disponibili, è possibile riconfigurare il programma in modo che possa essere eseguito con un grado di parallelismo diverso. Questo fa sì che si possano aumentare le prestazioni dell'applicazione o evitare di occupare risorse inutilmente. Per modificare il grado di parallelismo del programma ASSIST in esecuzione, è necessario:

- terminare l'esecuzione in corso;
- modificare il grado di parallelismo per uno o più parmod;
- eseguire il programma ASSIST con la nuova configurazione.

La configurazione del grado di parallelismo di un generico parmod, avviene passando come parametro il numero di processori virtuali (*VPM*) da attivare, all'interfaccia messa a disposizione dal componente CCM. Questa operazione viene memorizzata sotto forma di stringa nel file di configurazione di ASSIST. Per questo è necessario eseguire da prima la terminazione del programma, poi la riconfigurazione, ed infine la riattivazione. L'attivazione del programma ASSIST, avviene invocando i metodi messi a disposizione dall'interfaccia predisposta a questo compito. Di fatto l'operazione implica la lettura del file di configurazione, l'esecuzione dei comandi necessari per l'inizio dell'esecuzione, l'impostazione delle variabili necessarie per comunicare al componente CCM che il programma ASSIST è attivo. L'invocazione dei metodi sull'interfaccia per la terminazione del programma ASSIST, permette di terminare l'esecuzione del programma stesso nel seguente modo: da prima vengono chiusi tutti i socket che collegano i thread impliciti ed i moduli sequenziali CCM_input; la chiusura dei socket produce la terminazione dei moduli CCM_input di ASSIST che a sua volta produce una reazione a catena all'interno del programma ASSIST. Di fatto, al procedere dello svuotamento del *pipeline*, vengono terminati tutti i moduli ASSIST attivati, procurando infine la terminazione corretta del programma ASSIST e la chiusura dei socket sui thread espliciti. La terminazione istantanea del programma ASSIST si distingue da quella appena descritta in quanto tutti i moduli in esecuzione vengono terminati senza attendere lo svuotamento del pipeline. Questo implica che tutti i risultati, in corso di elaborazione, vengono persi, mentre nel caso precedente ciò non avviene.

Tramite le altre interfacce non funzionali (5-7), è possibile gestire anche il componente stesso. È possibile attivare e terminare il componente CCM mediante i metodi sulle interfacce predisposte a questo scopo, in modo analogo a quello descritto per il programma ASSIST. Il metodo per attivare l'esecuzione del componente inizializza le strutture dati del componente stesso (porte e socket), ed attiva i thread necessari per l'esecuzione del programma ASSIST incapsulato. Il metodo per terminare l'esecuzione del componente invoca a sua volta il metodo per la terminazione del

programma ASSIST, ed in seguito chiude definitivamente i socket che collegano il componente ad ASSIST.

Un'applicazione CCM è caratterizzata da un programma principale (*main*), che ha il compito di istanziare e collegare le componenti dell'applicazione. Il main può utilizzare le interfacce non funzionali dei componenti dichiarati, in particolare può attivare o terminare un componente CCM, può attivare o terminare un programma ASSIST e può riconfigurare i programmi ASSIST incapsulati dai vari componenti.

### 4.3.5 Chiamate RMI nel componente CCM

Nell'OpenCCM è possibile inviare e ricevere informazioni, oltre che con il meccanismo degli eventi, anche attraverso il *Remote Method Invocation* (RMI). In questo caso è possibile parlare di vere e proprie applicazioni *client-server* in cui abbiamo un server che esporta uno o più servizi, o più client che richiedono i servizi esportati da un server.

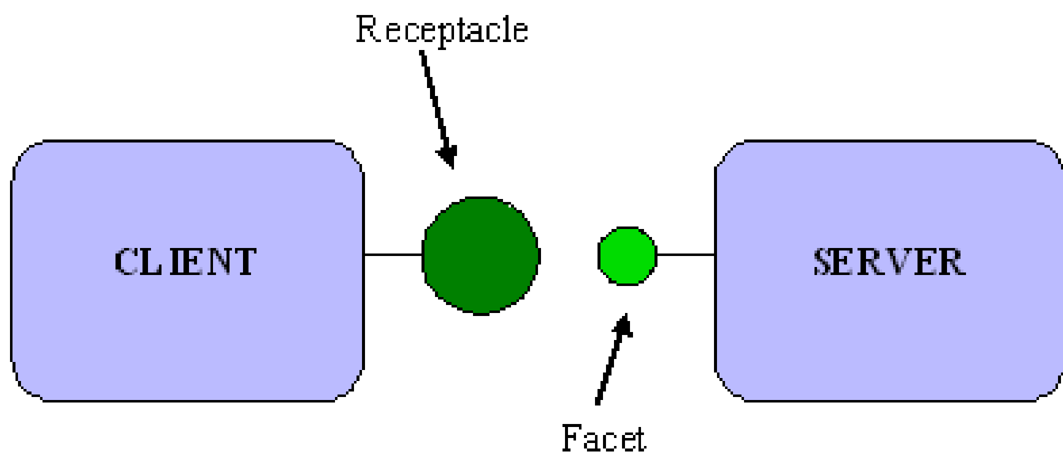



Figura 4.4: Applicazione CCM *Client-Server* mediante chiamate RMI.

Per realizzare un'implementazione con chiamate RMI, il componente, deve essere in grado di restituire i risultati, ricevuti dal programma ASSIST, al client che aveva richiesto il servizio. Questo comporta le seguenti conseguenze:

- introduzione di un *tag* univoco nei dati inviati e ricevuti tramite socket. I dati

inviati e ricevuti da ASSIST devono avere un tag, in modo da poter individuare in modo univoco ogni elaborazione, ossia in modo da poter collegare il dato in ingresso con il relativo risultato. Di conseguenza i thread impliciti del componente devono inviare ad ASSIST una struttura dati contenente un tag ed i dati ricevuti come parametro di ingresso. I thread espliciti invece devono ricevere una struttura dati contenente un tag ed i risultati, da inoltrare non più al componente esterno ma al thread implicito che è in attesa dei risultati;

- introduzione del tag nel tipo degli stream. Il programma ASSIST incapsulato deve essere scritto in modo che il tipo degli stream non connessi sia una struttura con un campo *tag* ed un campo *data*. Di fatto, il tag ricevuto tramite socket dal componente, deve seguire passo passo i dati che contrassegna.

Quest'ultima conseguenza descritta, implica che il programma ASSIST deve gestire i dati ricevuti tenendo conto che il tag deve essere propagato lungo tutto il pipeline d'esecuzione. Per questo motivo non è possibile incapsulare in un componente CCM, che esporta una o più chiamate RMI, qualsiasi programma ASSIST; di fatto la gestione di tale tag deve essere implementata dal programmatore del modulo ASSIST.

### 4.3.6 Integrazione di un'intera applicazione ASSIST

Nelle sezioni precedenti si è descritta nel dettaglio l'integrazione di un modulo ASSIST in un componente CCM, tramite il tool ASSIST2CCM. In questa sezione invece viene discussa una parte del lavoro di tesi che ha riguardato l'integrazione di un'intera applicazione ASSIST, costituita da diversi moduli paralleli, in una applicazione costituita da componenti CCM che incapsulano i moduli ASSIST. Prima di discutere ciò, di seguito sono riportate alcune importanti caratteristiche dei CCM e del tool ASSIST2CCM che hanno influenzato l'andamento del lavoro. La prima sostanziale differenza riscontrata è che l'ambiente dell'implementazione del CCM utilizzata, OpenCCM, è, al contrario di ASSIST, un ambiente Java. Inoltre, nel mondo CCM,

le comunicazioni su stream non sono implementate e il tool ASSIST2CCM ha dovuto sopperire a questa mancanza sfruttando il meccanismo delle comunicazioni ad eventi. Questo ha comportato, come detto nelle precedenti sezioni, l'introduzione di nuovi thread di interfaccia tra i due ambienti che sono stati mappati sui vari processori durante l'elaborazione.

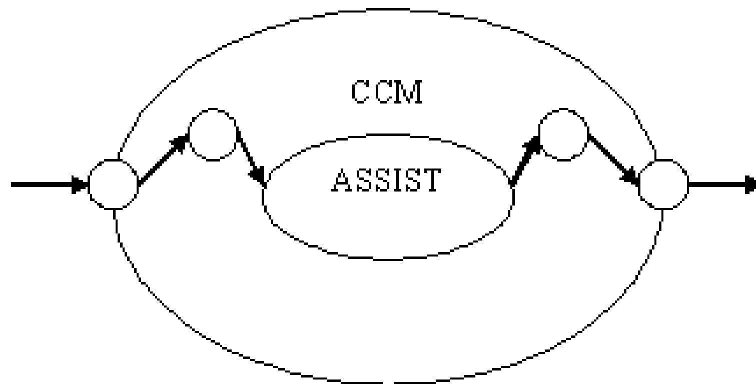



Figura 4.5: Rappresentazione di un modulo ASSIST incapsulato in un componente CCM.

Il tool utilizzato per l'integrazione è stato pensato per un'applicazione composta da un unico modulo ASSIST. Nonostante ciò il tool ASSIST2CCM, poiché implementato in maniera modulare, è stato utilizzato ugualmente in quanto, con opportune modifiche, ha permesso l'integrazione di un'applicazione ASSIST complessa in più componenti CCM (ogni modulo ASSIST viene incapsulato in un componente CCM).

In particolare, ASSIST2CCM è composto da tre fasi. Nella prima fase vengono generati i file IDL del modulo ASSIST da integrare e degli eventi ad esso correlati (stream di ingresso ed uscita del modulo). A partire da questi vengono generati gli stub, gli skeleton e i file Java necessari per la costituzione del componente CCM. Infine, nell'ultima fase, vengono compilati i file Java generati ed il modulo ASSIST vero e proprio. Queste operazioni vengono eseguite in maniera automatica dal tool ogni volta che viene invocato per l'integrazione di un singolo modulo ASSIST.

Per effettuare l'integrazione di un'intera applicazione, quindi, si è resa necessaria la separazione delle varie fasi descritte, poiché vi era l'esigenza di un singolo file

IDL degli eventi, comune a tutti i componenti dell'applicazione, che specificasse le interfacce e quindi la composizione dei componenti. A partire da questo requisito sono stati generati i file IDL degli eventi e dei moduli separatamente ed in maniera automatica. In seguito 'manualmente' sono stati uniti gli IDL degli eventi generati in un unico file utilizzato per la generazione automatica dei file Java, degli stub e degli skeleton necessari. Infine è stata eseguita la compilazione dei file Java ed ASSIST. Il lavoro finale è stato quello di creare un file *main.java*, in cui vengono localizzati, istanziati ed eseguiti i componenti CCM che costituiscono l'applicazione.

# Capitolo 5

# Metodologia proposta e sperimentazioni

Sommario

In questo capitolo verrà descritta l'intera applicazione sviluppata e la metodologia seguita negli esperimenti. Verranno enunciati e commentati, infine, i risultati raggiunti, con particolare enfasi sull'eventuale *overhead* nel calcolo e nelle comunicazioni, introdotto dai componenti CCM, e sulla bontà dell'Application Manager nel rimuovere i colli di bottiglia ed evitare lo spreco di risorse in una applicazione grid-aware.

L'obiettivo di questa tesi è quello di verificare la validità del modello sperimentale presentato in precedenza: l'integrazione fra l'ambiente ASSIST ed il framework CCM come punto di partenza per la costruzione di un ambiente di programmazione ad alte prestazioni, basato sulla tecnologia a componenti, per le griglie computazionali.

L'idea iniziale è stata quella di implementare un'applicazione complessa che richiedesse una potenza di calcolo non banale e che producesse un risultato concreto (vedi sez. 5.3). Per questo in fase di progettazione l'orientamento è stato focalizzato su due applicazioni che tradizionalmente usufruiscono della tecnologia del calcolo parallelo per la loro elaborazione: il calcolo dell'insieme di Mandelbrot (vedi sez. 5.1) e la compressione MPEG (vedi sez. 5.2). Il risultato finale voluto è stato quello di costruire un filmato compresso, rappresentante uno zoom su una particolare area dell'insieme di Mandelbrot, partendo da immagini non compresse dell'insieme.

Un requisito che l'applicazione doveva rispettare fin dalle fasi iniziali della sua progettazione è stato quello di costituirla a partire da una serie di moduli distinti, in previsione dell'integrazione con i componenti CCM. Inoltre, in questo modo, è stato possibile dare origine ad un'applicazione complessa che presentasse dei comportamenti dinamici, i quali richiedevano l'introduzione di una gestione dell'applicazione in grado di renderla adattiva ai vari contesti. Di fatto, il calcolo dell'insieme di Mandelbrot richiede un tempo di elaborazione dipendente dall'area del piano complesso in cui l'immagine deve essere calcolata. L'applicazione progettata richiede il calcolo di immagini dell'insieme sempre diverse, questo fa sì che il tempo di elaborazione sia variabile per ogni immagine calcolata. È questa la spiegazione per il comportamento dinamico che l'applicazione assume in fase di elaborazione. Per il particolare percorso scelto nell'esplorare l'insieme, l'elaborazione è caratterizzata da tre fasi:

- prima fase: il calcolo delle immagini dell'insieme di Mandelbrot risulta *veloce*;
- seconda fase: il tempo di elaborazione di ogni singola immagine sale notevolmente;
- terza fase: il tempo di elaborazione di ogni singola immagine scende nuovamente.

Questo ha comportato l'introduzione di un modulo di gestione per l'applicazione che consentisse un risparmio delle risorse allocate e/o un aumento delle prestazioni 5.3.5.

La fase di implementazione è stata guidata di volta in volta dai risultati ottenuti dagli esperimenti. Questo ha consentito il conseguimento dell'obiettivo della tesi creando le condizioni, per la fase di sperimentazione, che potessero essere il più possibile aderenti alla realtà delle griglie computazionali. La fase iniziale di analisi dei requisiti ha comportato lo studio di due applicazioni ASSIST, che verranno considerate moduli dell'applicazione più complessa, già esistenti ed indipendenti fra loro: *Mandelbrot* ed *MpegEncoder*.

Nel seguito verranno illustrati in dettaglio i moduli che sono stati utilizzati. Inoltre saranno descritti vari studi dell'applicazione ed i risultati degli esperimenti eseguiti per ogni studio, in modo da descrivere le fasi del lavoro svolto e i fattori che hanno guidato i cambiamenti nell'implementazione. Nel primo studio verrà descritta una prima versione dell'applicazione in cui i due moduli Mandelbrot ed MpegEncoder sono stati collegati attraverso l'introduzione di un nuovo modulo (*PixConverter*). Inoltre verranno presentati i risultati delle sperimentazioni eseguite inizialmente con i singoli moduli separati ed, in seguito, con l'intera applicazione. Nel secondo studio verranno illustrati i fattori che hanno portato alla costituzione di una seconda versione dell'applicazione e come questi abbiano inciso sull'implementazione della nuova versione. Di fatto, tali fattori si riferiscono alle prestazioni dell'applicazione. Anche in questo caso verranno presentati i risultati ottenuti nelle fasi di sperimentazione. Nel terzo studio verranno illustrate le innovazioni portate alla seconda versione dell'applicazione che hanno dato origine così alla terza versione. Di fatto, queste modifiche sono state decise in quanto, dagli esperimenti precedenti, si è evidenziato un comportamento dinamico dell'applicazione dovuto alla variabilità del tempo di elaborazione di ogni immagine da parte del modulo Mandelbrot, descritta precedentemente. Ciò ha motivato esperimenti sull'adattività dell'applicazione che hanno portato come detto all'introduzione di un modulo di gestione per l'applicazione che consentisse un risparmio delle risorse e/o un aumento delle prestazioni. In particolare è stato introdotto il modulo *ApplicationManager* che simula l'adattività dell'applicazione su griglia scegliendo, in base a dei tempi rilevati dagli altri moduli, quale implementazione utilizzare di due moduli MpegEncoder, introdotti per simulare un incremento o un decremento delle prestazioni nello stesso modulo. Quando il modulo Mandelbrot risulta il collo di bottiglia (seconda fase dell'elaborazione), ApplicationManager determina l'utilizzo del modulo MpegEncoder che utilizza meno risorse (più lento). Allo stesso modo, quando il collo di bottiglia risulta l'implementazione più lenta dell'MpegEncoder (prima e terza fase dell'elaborazione), ApplicationManager determina l'utilizzo dell'altra implementazione di

MpegEncoder che utilizza maggiori risorse. Il risultato voluto è che, con questo tipo di politica (vedi sez. 5.3.5), la performance dell'applicazione sia paragonabile all'utilizzo del modulo MpegEncoder più veloce, ma che allo stesso tempo non vi sia uno spreco di risorse impiegate per l'esecuzione del modulo MpegEncoder quando il collo di bottiglia è il modulo Mandelbrot. Questo ha determinato l'introduzione di un modulo *StreamStore* che si occupa di ricostituire il risultato finale dell'intera elaborazione a partire dai dati ricevuti alternativamente dalle due implementazioni dei due MpegEncoder. Anche in questo caso verranno presentati i risultati degli esperimenti eseguiti con questa versione dell'applicazione sia su cluster omogeneo sia su griglia.

Il modulo ApplicationManager appena descritto, così come gli altri moduli, sono stati progettati in modo tale da rendere agevole l'integrazione dell'intera applicazione con il framework CCM (vedi sez. 4.3.6), sperimentata fin dalla prima versione dell'applicazione. Questo ha consentito di paragonare i risultati ottenuti dagli esperimenti dell'applicazione costituita solo da moduli ASSIST, con quelli ottenuti dall'applicazione costituita da moduli ASSIST integrati in componenti CCM (vedi cap. 4). Gli studi dell'applicazione effettuati sono stati eseguiti su piattaforme diverse nelle differenti versioni. In particolare gli esperimenti relativi alla prima ed alla seconda versione dell'applicazione sono stati eseguiti su un cluster omogeneo, in quanto propedeutici allo studio dell'applicazione finale in esecuzione in ambienti eterogenei.

In questo capitolo verrà presentata in dettaglio la versione finale dell'applicazione implementata al fine della sperimentazione e verranno introdotti i risultati ottenuti dagli esperimenti eseguiti sulle tre versioni dell'applicazione. In particolare tali sperimentazioni verranno suddivise in due sezioni:

1. Esperimenti preliminari (sez. 5.5), in cui vengono enunciati i risultati raggiunti dalle sperimentazioni delle prime due versioni dell'applicazione, in ambiente omogeneo, che sono stati propedeutici per la definizione della versione finale

dell'applicazione;

2. Esperimenti sulla versione finale dell'applicazione (sez. 5.6), in cui sono riportati i risultati più importanti per gli obiettivi fissati nella tesi (vedi cap. 1): validazione del modello sperimentale proposto (integrazione ASSIST-CCM come punto di partenza per la definizione di un ambiente di programmazione per applicazioni grid-aware) e della politica adattiva implementata dall'ApplicationManager.

## 5.1 L'insieme di Mandelbrot

L'insieme di Mandelbrot è definito come l'insieme dei numeri complessi $c$ tale per cui non è divergente la successione definita da:

$$\begin{cases} z_{n+1} = & z_n^2 + c; \\ z_0 = & 0; \end{cases}$$

L'insieme consiste in un frattale (vedi fig. 5.1), la cui elaborazione richiede una

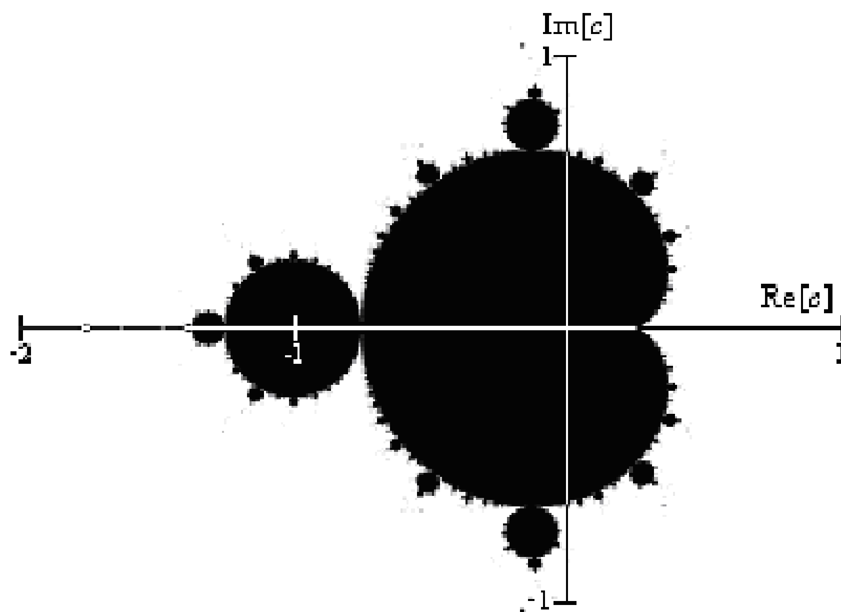



Figura 5.1: Immagine dell'insieme di Mandelbrot.

notevole potenza di calcolo. Per questo motivo è stato scelto, in questa tesi, di utilizzare un modulo parallelo che implementa questo calcolo come modulo generatore

delle immagini necessarie a formare il filmato finale. L'insieme deve il suo nome a Benoît Mandelbrot che nel 1975 introdusse il termine frattale per descrivere alcuni comportamenti matematici che sembravano avere un comportamento *caotico*. Questo genere di fenomeni nasce dalla definizione di curve o insiemi tramite funzioni o algoritmi. Si può dimostrare che se il modulo di $z_n$ è maggiore di 2 allora la successione diverge e quindi il punto $c$ è esterno all'insieme di Mandelbrot. Il minimo valore di $n$ per cui $|z_n| > 2$ è un indice di quanto 'lontano dal bordo' si trova un punto e viene spesso utilizzato, come descritto in seguito, per la visualizzazione del colore di ogni punto dell'insieme.

## 5.2 I compressori Mpeg

Il *codec* (*COdificator-DECodificator*) è un particolare software che contiene un algoritmo costruito per codificare e decodificare, in questo caso, una sequenza video. Sappiamo che la mole di dati di una sequenza video è molto elevata, quindi sarebbe troppo costoso memorizzare un filmato senza averlo prima compresso (e quindi codificato) con un algoritmo di compressione. A questa mansione risponde l'Encoder che, appunto, in fase di acquisizione trasforma le informazioni ricevute in un file più piccolo e compatto. MPEG (nelle sue varie versioni) è uno dei formati più affermati negli ultimi anni, grazie all'altissima compressione ed alla qualità decisamente buona. Per il suo carattere *lossy*, cioè la compressione rimuove alcuni dettagli meno significativi dalle immagini, MPEG raggiunge rapporti di compressione fino a 200:1. La sua tipologia è tipicamente asimmetrica, cioè la compressione avviene in un tempo decisamente maggiore della riproduzione, ciò fa sì che la compressione di un filmato richieda una notevole potenza di calcolo.

Nelle specifiche *MPEG1*, i fotogrammi che formano il filmato sono suddivisi in tre categorie: esistono gli *I-frame* (*Intra frame*), fotogrammi chiave di riferimento meno compressi, quindi i *P-frame* (*Predictive frame*), più compressi e ricavati per interpolazione dalle differenze di struttura con l'I-frame, e infine i *B-frame* (*Bidirec-*

*tional predictive frame*), la cui struttura è ricavata sia dal fotogramma precedente che da quello successivo e sui quali è applicata la massima compressione. L'immagine presente nel singolo fotogramma viene suddivisa in *macroblocchi* di 16x16 pixel ed ognuno di essi in successivi blocchi di 8x8. L'algoritmo *DCT* (*Discrete Cosine Transform* - Trasformata Discreta dei Coseni) tenta di fondere la struttura ed il colore dei blocchi 8x8 in un solo pixel per poi applicare la stessa procedura ai macroblocchi. A seconda del livello di compressione applicato tale fusione può essere elevata, media o lieve, con relativa qualità di immagine scarsa, media o ottima rispettivamente. Nelle zone di colore uniforme, per esempio una strada sgombra con sullo sfondo il cielo, l'MPEG1 risulta efficace, la compressione è alta con minima perdita di dettaglio, mentre, in una scena ricca di dettagli come la visione dell'interno di un bosco, si può percepire una certa *blocchettizzazione* dell'immagine ed artefatti attorno agli oggetti dovuti alla compressione elevata. In MPEG1 le sequenze di fotogrammi si presentano in modo ciclico (ad esempio IBBPBBPBBPBB) a gruppi di 12. Dopo il dodicesimo B-frame inizierà un nuovo I-frame. Ogni gruppo di fotogrammi è detto *GOP* (*Group Of Pictures*). L'algoritmo DTC viene usato per la compressione degli I-frame mentre per stimare le differenze strutturali sui P-frame e sui successivi B-frame si applicano anche gli algoritmi di *Motion Compensation* (compensazione del moto) e *Motion Estimation* (stima del moto). Se tra un fotogramma e l'altro c'è stato uno spostamento di una parte dell'immagine, questi algoritmi tentano di prevedere come si sposteranno i vari blocchi in modo da ricostruire correttamente la struttura del fotogramma successivo.

*MPEG2* rispetto ad MPEG1 può operare in *dual-pass*: dall'esame del filmato da comprimere si crea un database delle scene statiche e dinamiche, poi successivamente applica una compressione mirata alle varie parti e passa quindi da un valore di compressione costante (*CBR*, *Constant BitRate*) ad uno variabile (*VBR*, *Variable BitRate*).

L'ultima specifica *MPEG4* oltre alle caratteristiche di MPEG2 vanta nuovi algoritmi di matematica frattale e una struttura ad oggetti disposti su livelli di uno

spazio 3D virtuale. La compressione avviene usando *GMC* (*Global Motion Compensation*) che tiene conto della struttura dell'intero fotogramma e migliora la nitidezza nelle scene di movimento in cui la telecamera si sposta in una direzione continua.

## 5.3 L'applicazione in dettaglio

La fase iniziale di analisi dei requisiti ha comportato lo studio di due applicazioni ASSIST, che verranno considerate moduli dell'applicazione più complessa, già esistenti ed indipendenti fra loro: *Mandelbrot* ed *MpegEncoder*. In questa sezione verranno descritti i moduli in maggior dettaglio per capire quali sono stati i vincoli sulle interfacce e sul formato dei dati elaborati al fine della composizione dei moduli.

Il modulo Mandelbrot calcola l'immagine dell'insieme di Mandelbrot a partire da una data area del piano complesso. Di fatto, questo modulo è stato implementato secondo il paradigma farm:

- l'emettitore suddivide l'area del piano complesso, su cui effettuare il calcolo, per righe ed, ad ognuna, associa un indice;

- ogni riga viene affidata ad un worker per il calcolo della funzione di Mandelbrot nei singoli punti che la compongono, in questo modo viene anche determinato il colore associato ad ogni punto;

- il collettore si occupa infine di ricostituire l'immagine ricomponendo le righe elaborate dai worker grazie all'indice associato.

Il formato dei dati utilizzato durante la computazione del modulo Mandelbrot è un formato interno, definito ad hoc dagli sviluppatori. Questo formato di codifica delle immagini definisce il colore associato ad ogni punto utilizzando un gamut lineare.

Il modulo MpegEncoder genera un filmato compresso a partire da una serie di immagini non compresse. Anche questo modulo è stato implementato secondo il paradigma farm:

- l'emettitore legge da un file le immagini che devono essere elaborate e costituisce un GOP (Group Of Picture), a cui viene associato un indice, da inviare ai worker;
- i worker si occupano di comprimere i GOP ricevuti dall'emettitore ed inviare i GOP compressi al collettore, come risultato della loro computazione;
- il collettore ricostituisce il filmato nella sequenza originale grazie all'indice associato ad ogni GOP compresso. Inoltre si occupa di salvare su file il risultato dell'intera computazione.

Il formato dei dati in ingresso al modulo MpegEncoder è espresso nella codifica standard *PIX*. Tale codifica utilizza un gamut *rgb-alpha* per il colore associato ad ogni singolo punto dell'immagine che risulta suddivisa per righe.

Dall'analisi qui descritta sono state evidenziate alcune caratteristiche per le quali si sono rese necessarie delle modifiche ai moduli di partenza e l'introduzione di un nuovo modulo, *PixConverter*, che potesse rendere compatibili i risultati prodotti da Mandelbrot con i dati in ingresso ad MpegEncoder. Di fatto, il modulo Mandelbrot era implementato in modo che calcolasse un'unica immagine dell'insieme di Mandelbrot su una particolare area del piano complesso scelta staticamente. Inoltre il risultato della computazione, come già detto, veniva espresso in un formato ad hoc, mentre il tipo di dato in ingresso al modulo MpegEncoder veniva espresso nel formato PIX. Il modulo MpegEncoder era implementato in modo che leggesse da un file preesistente tutte le immagini non compresse da elaborare e costituenti il filmato.

L'applicazione utilizzata per il conseguimento dell'obiettivo della tesi è costituita, oltre che dai moduli appena menzionati, anche da altri due moduli che sono: *ApplicationManager* e *StreamStore*. Il primo implementa la politica adottata per consentire all'applicazione di adattarsi durante le diverse fasi caratterizzanti la sua computazione. Il secondo consente di raccogliere i risultati elaborati da due diversi moduli MpegEncoder (introdotti per la gestione della adattività dell'applicazione alla dinamicità della griglia computazionale, come descritto precedentemente) e ri-

comporre il risultato finale come mostrato in figura 5.2. Di seguito viene riportata

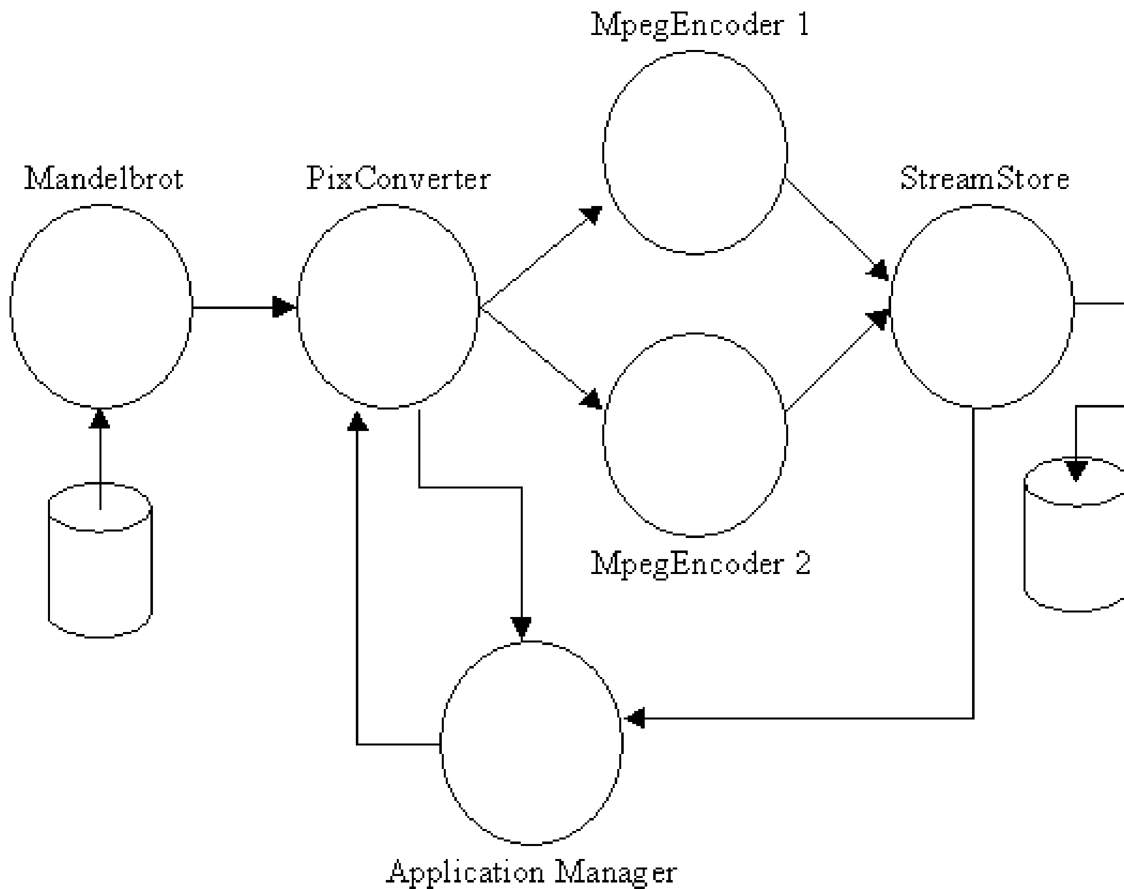



Figura 5.2: Architettura dell'applicazione.

la descrizione dell'intera applicazione e di ogni singolo modulo che la compone. L'applicazione è così strutturata:

- un modulo Mandelbrot, generatore dello stream di dati che devono essere elaborati;

- un modulo PixConverter, il cui compito è quello di convertire le immagini ricevute da Mandelbrot, generare il GOP da inviare ad uno dei moduli MpegEncoder, rilevare ed inviare al modulo ApplicationManager informazioni relative ai tempi di interarrivo degli elementi dello stream, il tempo di inattività (*idle*) del modulo stesso ed il tempo impiegato per la trasmissione dei dati ad un modulo MpegEncoder;

- due moduli MpegEncoder, la cui computazione è perfettamente identica, ma che differiscono l'uno dall'altro per il tempo di servizio impiegato nell'elaborazione di un elemento dello stream poiché utilizzano un diverso numero di risorse. Tali moduli vengono utilizzati, in fase di computazione, in modo esclusivo l'uno rispetto all'altro;

- un modulo StreamStore, il cui compito è quello di ricostruire il filmato, risultato della computazione dei moduli MpegEncoder, a partire dai GOP compressi ricevuti. Inoltre invia al modulo ApplicationManager informazioni relative al tempo di interarrivo degli elementi dello stream;

- un modulo ApplicationManager, il cui compito è quello di raccogliere i dati ricevuti da PixConverter e da StreamStore, relativi ai tempi rilevati, e di comunicare, al modulo PixConverter, a quale dei due moduli MpegEncoder debbano essere inviati gli elementi dello stream.

### 5.3.1 Modulo Mandelbrot

Le modifica apportata al modulo è stata quella di svincolarlo dal calcolo di una singola immagine e di rendere parametriche sia il numero delle aree da elaborare, e quindi delle immagini, sia la loro ubicazione e dimensione nel piano complesso in modo da ottenere uno zoom. A tal fine è stato predisposto un programma ausiliario che consentisse la creazione di un file di configurazione per Mandelbrot (*Mandel.conf*), contenente le aree su cui calcolare le immagini. Tale programma definisce le diverse aree in modo che il filmato finale risulti uno zoom, di velocità costante, in un punto fissato dell'immagine iniziale. La figura 5.3 mostra com'è costituito il modulo Mandelbrot dopo le modifiche apportate per renderlo compatibile con l'applicazione progettata. La procedura *Genera* si occupa di leggere dal file di configurazione le aree su cui calcolare l'insieme di Mandelbrot e, per ogni area, invia ai worker le righe su cui eseguire il calcolo, come descritto in precedenza. La procedura *Raccogli* si occupa di raccogliere i dati elaborati dai worker e ricostituire l'immagine di una

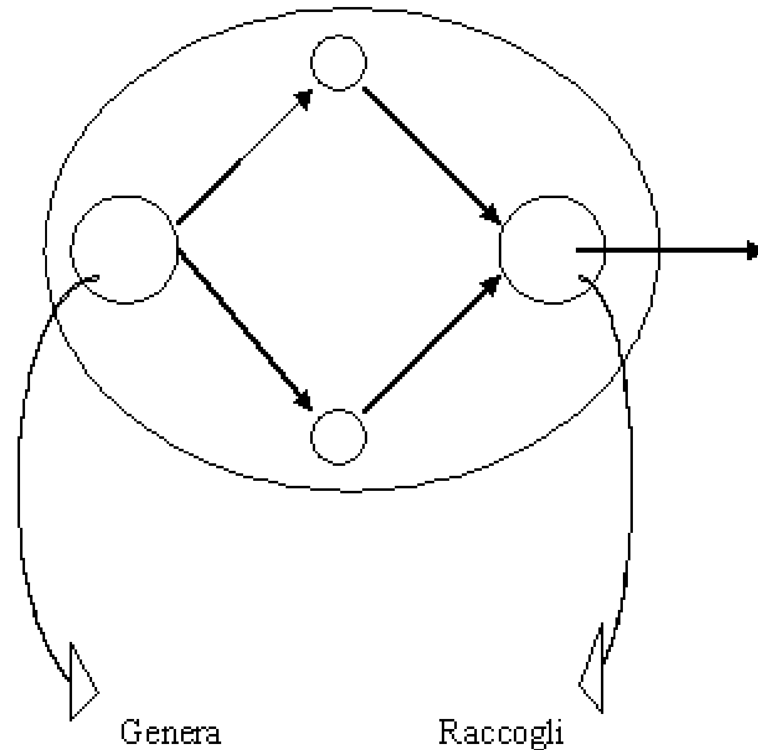



Figura 5.3: Architettura del modulo Mandelbrot.

data area. Poiché può accadere che righe appartenenti ad immagini diverse vengano calcolate in parallelo da diversi worker, e quindi che Raccogli possa assemblare immagini in modo non corretto, è stato previsto un meccanismo di code che, associato agli indici di identificazione già previsti, permette alla procedura Raccogli di assemblare ed inviare le immagini al modulo successivo correttamente. Nella versione finale dell'applicazione il modulo Mandelbrot calcola 201 immagini dell'insieme omonimo. Tale parametro è stato determinato in seguito agli esperimenti effettuati nel corso dello sviluppo dell'applicazione come sarà descritto in maggior dettaglio in seguito.

### 5.3.2 Modulo PixConverter

Il modulo PixConverter, un *parmod* con topologia *one*, è stato progettato per raccogliere le immagini provenienti dal modulo Mandelbrot ed eseguire la conversione da gamut di colori lineare a gamut di colori rgb-alpha (formato PIX). Questo in quanto

il tipo di dato in uscita dal modulo Mandelbrot deve essere compatibile con il tipo di dato in ingresso al modulo MpegEncoder. Inoltre, oltre alla conversione delle immagini nel giusto formato, PixConverter genera una transizione da un'immagine all'altra che consiste in una dissolvenza della prima e l'apparizione della seconda (*fade*). Per creare il fade vengono generate 10 nuove immagini ogni 2 inviate da Mandelbrot, inoltre per avere un migliore risultato finale, ogni singola immagine viene replicata 10 volte. Eseguite queste operazioni vengono creati e inviati due GOP ad uno dei moduli MpegEncoder. Ogni GOP inviato ha una dimensione pari a circa 15 MByte da cui consegue che le comunicazioni sono a grana *grossa*. Attraverso un file di configurazione, PixConverter può convertire le immagini ricevute da Mandelbrot nel formato standard *PPM* (*Portable Pixel Map*, un particolare formato per la codifica di immagini che utilizza un gamut di colori rgb-alpha), in alternativa all'invio delle immagini, nell'opportuno formato, all'MpegEncoder. PixConverter è stato inserito all'interno dell'applicazione collegandolo al modulo ApplicationManager e ai due moduli MpegEncoder come mostrato in figura 5.2. PixConverter comunica ad ApplicationManager il tempo di interarrivo dei dati dal modulo Mandelbrot (INPUT_PIX), il tempo necessario per inviare i dati ad uno dei moduli MpegEncoder (ASSIST_OUT_PIX), il tempo di inattività trascorso nell'attesa dei dati da Mandelbrot (IDLE_PIX). Inoltre, PixConverter è in grado di ricevere un messaggio da ApplicationManager che indica a quale dei due moduli MpegEncoder inviare i dati in uscita. Ovviamente il messaggio trasmesso su questo canale di comunicazione ha la priorità rispetto al messaggio ricevuto da Mandelbrot. Questo perché, potendo ricevere non-deterministicamente i messaggi da entrambi i canali, è necessario dare priorità ai messaggi di configurazione.

### 5.3.3 Modulo MpegEncoder

La modifica apportata ad MpegEncoder consiste nell'implementazione del modulo in modo che questo possa ricevere uno stream di immagini non compresse da elaborare. In questo modo viene eliminata la dipendenza dal file. Inoltre, così facendo, la

computazione dell'intera applicazione può avvenire secondo il paradigma pipeline. Inoltre sono state apportate modifiche per quello che riguarda la fase di salvataggio e quindi la procedura *Raccogli*. Questa è stata completamente eliminata e sostituita da un collettore che si occupa di raccogliere i dati dai vari worker ed inviarli direttamente al modulo StreamStore.

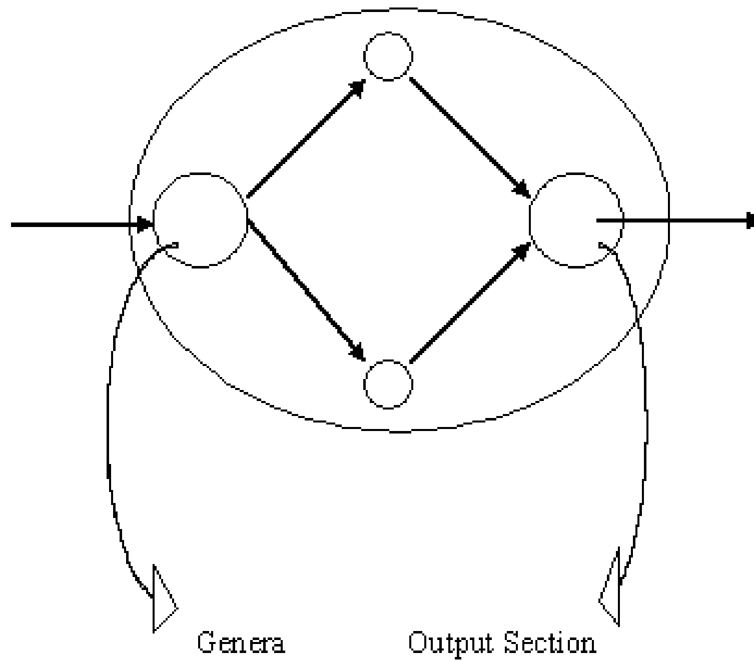



Figura 5.4: Architettura del modulo MpegEncoder.

### 5.3.4 Modulo StreamStore

Il modulo StreamStore ha il compito di raccogliere i GOP che vengono compressi da MpegEncoder e di salvare questi ultimi su disco, in modo che il risultato finale sia un unico file contenete l'intero filmato. Inoltre è in grado di rilevare il tempo di interarrivo degli elementi dello stream (INPUT_STREAM_STORE). Questi dati vengono inviati al modulo ApplicationManager per l'implementazione della politica di scheduling dei dati fra i due MpegEncoder.

### 5.3.5 Modulo ApplicationManager

Come detto nell'introduzione di questo capitolo, questo modulo è stato introdotto per sperimentare una particolare politica di adattività dell'applicazione che sfrutti la dinamicità nell'elaborazione. In particolare, come descritto in seguito, ApplicationManager indica a PixConverter di utilizzare alternativamente due implementazioni diverse di MpegEncoder in base ad una politica predefinita dipendente da alcuni tempi rilevati.

Di fatto, in base ai tempi rilevati dai moduli PixConverter e StreamStore, determina quale implementazione utilizzare di MpegEncoder. Quando il collo di bottiglia risulta l'implementazione di MpegEncoder (più lenta) che utilizza meno risorse (prima e terza fase dell'elaborazione), ApplicationManager determina l'utilizzo dell'altra implementazione di MpegEncoder, che utilizza maggiori risorse. Allo stesso modo, quando il modulo Mandelbrot risulta il collo di bottiglia (seconda fase dell'elaborazione), ApplicationManager determina l'utilizzo del modulo MpegEncoder più lento. Il risultato voluto è che, con questo tipo di politica, la performance dell'applicazione sia paragonabile all'utilizzo costante del modulo MpegEncoder più veloce, ma che allo stesso tempo non vi sia uno spreco di risorse impiegate per l'esecuzione del modulo MpegEncoder, quando il collo di bottiglia è Mandelbrot.

ApplicationManager comunica a PixConverter a quale dei due moduli MpegEncoder inviare i dati per adattare l'applicazione alla dinamicità della griglia (sovraccarico della rete o di un nodo), che, come detto, in questo caso è costituita dalla dinamicità dell'applicazione stessa (variabilità del tempo di elaborazione di ogni immagine di Mandelbrot).

Il modulo ApplicationManager raccoglie dal modulo PixConverter i seguenti tempi:

- IDLE_PIX: indica il tempo di inattività che il modulo PixConverter trascorre nell'attesa dei dati da elaborare dal modulo Mandelbrot. Se questo tempo risulta grande (ordine dei secondi), allora si può dedurre che il modulo Mandel-

brot rappresenta il collo di bottiglia dell'applicazione perché più lento rispetto all'implementazione del modulo MpegEncoder utilizzata.

- ASSIST_OUT_PIX: indica il tempo necessario per inviare due elementi dello stream al modulo MpegEncoder correntemente utilizzato. Questo può variare a seconda del contesto, cluster omogeneo o griglia. Da questo dato si può ricavare una stima dell'eventuale overhead di comunicazione introdotto dall'integrazione con il framework CCM nei contesti di svolgimento degli esperimenti. Inoltre, c'è da notare che l'aumentare di questo tempo è sintomo del fatto che il collo di bottiglia si è spostato in MpegEncoder.
- INPUT_PIX: indica il tempo di interarrivo dei messaggi dal modulo Mandelbrot. Questo tempo è necessario al modulo ApplicationManager per calcolare la banda di esecuzione media (espressa in immagini al secondo) dell'intera applicazione ed inoltre costituisce la somma dei primi due tempi rilevati.

Il modulo ApplicationManager raccoglie dal modulo StreamStore il tempo INPUT_STREAM_STORE che indica il tempo di interarrivo dei messaggi dal modulo MpegEncoder utilizzato. Ai fini dell'analisi dei risultati ottenuti non è stato preso in considerazione in quanto evidenzia soltanto il parallelismo fra i worker del modulo MpegEncoder utilizzato.

La politica adattiva implementata è stata basata sull'analisi del tempo IDLE_PIX di PixConverter per determinare il modulo collo di bottiglia dell'applicazione.

Una prima implementazione della politica adattiva ha portato a definire il seguente algoritmo che distingue il tempo di IDLE_PIX a seconda che sia inferiore o superiore ad una soglia stabilita staticamente:

- se IDLE_PIX risulta piccolo, nell'ordine dei centesimi di secondo, allora il modulo MpegEncoder utilizzato risulta il collo di bottiglia dell'intera applicazione. Questo perché una volta inviato il dato, PixConverter ne ha disponibile un altro inviato da Mandelbrot e può immediatamente proseguire la sua elaborazione. Quindi il modulo più lento è senza dubbio quello 'a valle'. In questo

caso ApplicationManager comunica a PixConverter di inviare i dati al modulo MpegEncoder che utilizza più risorse;

- se IDLE_PIX risulta grande, nell'ordine dei secondi, allora il modulo Mandelbrot risulta il collo di bottiglia dell'intera applicazione. Questo perché una volta inviato il dato ad un modulo MpegEncoder è necessario che PixConverter attenda il nuovo dato in ingresso per dei secondi. In questo caso ApplicationManager comunica a PixConverter di inviare i dati al modulo MpegEncoder che utilizza meno risorse.

La politica di base sopra descritta, però, è stata modificata in quanto particolarmente sensibile alle momentanee situazioni di carico sia della rete sia dei nodi della griglia, che avrebbero portato ad un continuo utilizzo di entrambi i moduli MpegEncoder. Per questo motivo è stato introdotto un intervallo di tolleranza ai fenomeni appena descritti. L'algoritmo sviluppato prevede perciò che il modulo ApplicationManager decida di cambiare l'implementazione di MpegEncoder da utilizzare solo nel momento in cui i valori di IDLE_PIX superino o siano inferiori alla soglia indicata per il numero di volte consecutive indicato dall'intervallo di tolleranza definito staticamente.

## 5.4 Ambienti di prova

Gli strumenti hardware che sono stati utilizzati per lo svolgimento degli esperimenti sono:

- *Pianosa*, un cluster composto da 24 macchine con processore Intel(R) Pentium(R) III Mobile con frequenza di 800 MHz, con cache da 32 KB, interconnesse da tre switched Ethernet(R) Pro 100. Ogni macchina è equipaggiata con 1 GB di memoria e il sistema operativo installato è Linux RedHat(R) 8.0, versione del kernel 2.4.22;

- *C1-C4*, un cluster composto da 4 macchine con processore Intel(R) Pentium(R)

IV con frequenza di 1996.613 MHz, con cache da 512 KB, interconnesse da una switched Gigabit Ethernet(R). Ogni macchina è equipaggiata con 512 MB di memoria e il sistema operativo installato è Linux RedHat(R) 9.0, versione del kernel 2.4.22;

- *Capraia*, desktop con processore Intel(R) Pentium(R) III con frequenza di 521.329 MHz, con cache da 512 KB. La macchina è equipaggiata con 384 MB di memoria e il sistema operativo installato è Linux Fedora Core 2(R), versione del kernel 2.6.8.1;

- *Rubentino*, *Verduzzo*, *Sangiovese*, *Cavit*, sono 4 macchine con processore AMD Athlon(tm) XP con frequenza di 2088.390 MHz, con cache da 256 KB. Ogni macchina è equipaggiata con 896 MB di memoria e il sistema operativo installato è Linux Debian(R), version del kernel 2.6.6.

I primi due cluster ed il desktop Capraia si trovano presso il Dipartimento di Informatica dell'Università degli Studi di Pisa e sono interconnessi tra loro da una rete Ethernet(R) da 100 Mb/sec intra-dipartimentale. Le altre quattro macchine si trovano presso l'ISTI-CNR di Pisa e sono anch'esse collegate fra loro da una rete Ethernet(R) da 100 Mb/sec. La rete che collega il Dipartimento di Informatica al CNR è una rete geografica sulla quale transita tutto il traffico di rete della città di Pisa.

## 5.5 Esperimenti preliminari

La descrizione degli esperimenti che seguiranno è suddivisa in tre fasi. Di fatto l'applicazione ha attraversato tre momenti fondamentali che ne hanno modificato sostanzialmente il comportamento, prima di arrivare alla costituzione della versione finale descritta in precedenza. In questa sezione sono stati analizzati gli esperimenti preliminari relativi solo alle prime due versioni dell'applicazione. I risultati ottenuti da questi esperimenti sono stati necessari per lo sviluppo e la sperimentazione

della versione finale dell'applicazione (vedi sez. 5.6). Di seguito viene riassunta la descrizione delle prime due versioni dell'applicazione oggetto della sperimentazione:

- Prima versione: i due moduli Mandelbrot ed MpegEncoder sono stati collegati attraverso l'introduzione di un nuovo modulo PixConverter. Gli esperimenti, che sono stati eseguiti su cluster omogeneo, hanno riguardato inizialmente i singoli moduli dell'applicazione e, successivamente, l'intera applicazione sia composta da moduli ASSIST sia integrata in componenti CCM, per il calcolo dell'eventuale overhead introdotto.

- Seconda versione: il modulo PixConverter è stato predisposto per calcolare un fade fra un'immagine e l'altra in modo da ridurre il numero di immagini calcolate da Mandelbrot e, di conseguenza, i tempi di elaborazione dell'intera applicazione che risultavano eccessivi nella prima versione. Gli esperimenti, che sono stati eseguiti su cluster omogeneo, hanno riguardato l'intera applicazione sia composta da moduli ASSIST sia integrata in componenti CCM, per il calcolo dell'eventuale overhead introdotto.

Gli esperimenti eseguiti, sia sull'applicazione in versione ASSIST sia in versione integrata con il CCM, sono stati focalizzati sullo studio dell'eventuale overhead introdotto nelle comunicazioni e nel calcolo dal CCM nell'ambiente omogeneo oggetto di test.

### 5.5.1 Studio dell'applicazione: prima versione

Come già detto, l'idea è stata quella di creare un filmato compresso rappresentante uno zoom su una particolare area dell'immagine dell'insieme di Mandelbrot. Per fare questo sono stati composti i due moduli, Mandelbrot ed MpegEncoder, preesistenti e descritti in precedenza. L'applicazione, in prima battuta, è stata progettata in modo da farle calcolare tutte le immagini dell'insieme necessarie per la costruzione del filmato. In questo modo, il modulo Mandelbrot ha generato 2420 immagini di dimensione 720x576 punti, e l'intervallo di appartenenza dell'indice $n$ della successione

di Mandelbrot (vedi sez. 5.1) va da 0 a 1023. Il modulo PixConverter ha costituito delle strutture dati GOP, composte ognuna da 10 immagini opportunamente convertite, ricevute dal modulo Mandelbrot, senza eseguire il fade fra un'immagine e l'altra. Infine il modulo MpegEncoder ha generato il filmato, risultato dell'intera computazione, elaborando i GOP ricevuti da PixConverter. In questa fase della sperimentazione MpegEncoder ha salvato direttamente i dati elaborati su file.

Di seguito, in figura 5.5, viene mostrata l'architettura della prima versione dell'applicazione. Come si può vedere, quest'ultima è stata progettata secondo un paradigma pipeline in cui vi sono stadi, il primo ed il terzo, implementati secondo un paradigma farm. Inoltre non sono presenti i moduli ApplicationManager e StreamStore.

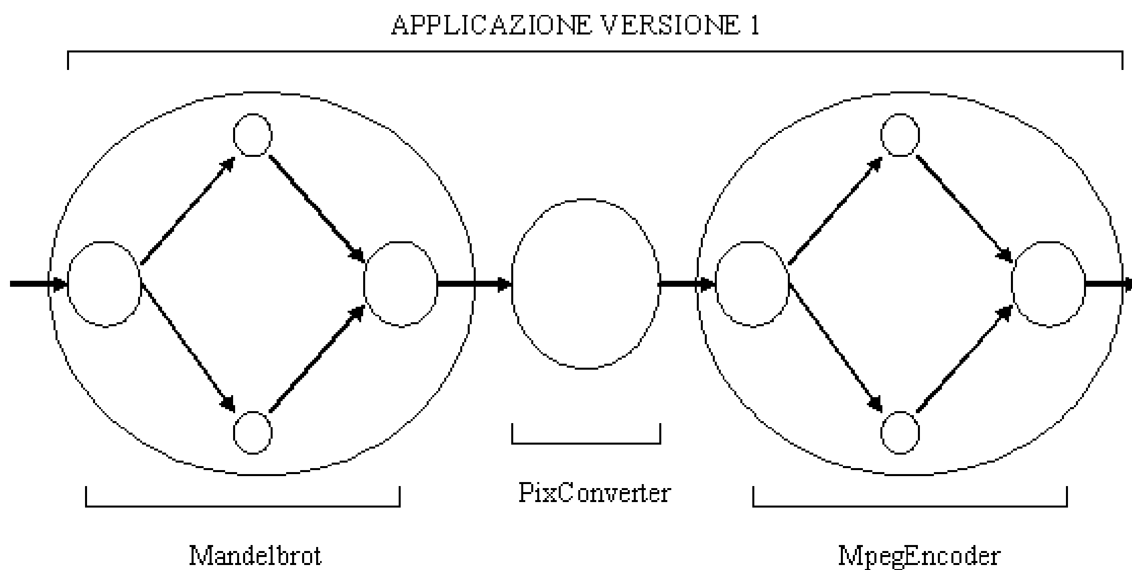



Figura 5.5: Architettura della prima versione dell'applicazione.

## Prestazioni

Per la misura delle prestazioni di tutta l'applicazione è stato necessario, da prima, misurare le prestazioni dei singoli moduli, per individuare il modulo più lento e poter così agire sul grado di parallelismo interno di quest'ultimo. Tali test sono stati effettuati sul cluster Pianosa, descritto precedentemente. Non vengono riportate

inoltre le prestazioni del modulo PixConverter, in quanto da un'analisi del codice si evince che questo modulo non risulta sicuramente il più lento di tutta l'applicazione.

### Mandelbrot

Il modulo Mandelbrot è stato compilato senza informazioni di *debug* e l'ambiente a run-time forniva il supporto della *Clam*[1]. In questa fase della sperimentazione sono state calcolate dal modulo Mandelbrot, per ogni prova, 50 immagini relative alla stessa area del piano complesso di 720x720 punti. Il calcolo di un singolo punto termina nel momento in cui il modulo del numero complesso $z_n$ (vedi sez. 5.1) diviene maggiore di 2. Il numero massimo di iterazioni per ogni punto, cioè l'indice $n$, è stato fatto variare da 1023 a 127 nelle diverse prove, per studiare l'andamento della banda di Mandelbrot. Di seguito riportiamo le varie tabelle con i tempi rilevati e il grafico riassuntivo.

| **Grado di parallelismo Mandelbrot** | **Banda (Img/Sec)** |
|---|---|
| 2 | 0.0187145 |
| 4 | 0.0374112 |
| 6 | 0.0561471 |
| 8 | 0.0745585 |
| 10 | 0.09127 |
| 12 | 0.10987 |
| 14 | 0.128532 |
| 16 | 0.144612 |

Tabella 5.1: Banda di Mandelbrot - Intervallo di appartenenza dell'indice $n$ [0-1023].

Come si può notare dal grafico 5.6, la banda del modulo Mandelbrot cambia a seconda del numero di iterazioni $n$ della successione, che vengono eseguite per ogni singolo punto dell'immagine. Tale studio è servito per determinare la banda massima del modulo Mandelbrot.

[1] *Coordination Language Abstract Machine* è una parte del run-time di ASSIST che fornisce dei servizi (ricerca e gestione delle risorse disponibili, monitoraggio della computazione) per architetture eterogenee e dinamicamente configurabili. La Clam utilizza processi POSIX e le primitive di comunicazione implementate dalla libreria *ACE Adaptive Communication Environment*.

| **Grado di parallelismo Mandelbrot** | **Banda (Img/Sec)** |
|---|---|
| 14 | 0.212295 |
| 16 | 0.230464 |

Tabella 5.2: Banda di Mandelbrot - Intervallo di appartenenza dell'indice $n$ [0-511].

| **Grado di parallelismo Mandelbrot** | **Banda (Img/Sec)** |
|---|---|
| 16 | 0.303615 |

Tabella 5.3: Banda di Mandelbrot - Intervallo di appartenenza dell'indice $n$ [0-255].

| **Grado di parallelismo Mandelbrot** | **Banda (Img/Sec)** |
|---|---|
| 16 | 0.21666 |

Tabella 5.4: Banda di Mandelbrot - Intervallo di appartenenza dell'indice $n$ [0-127].

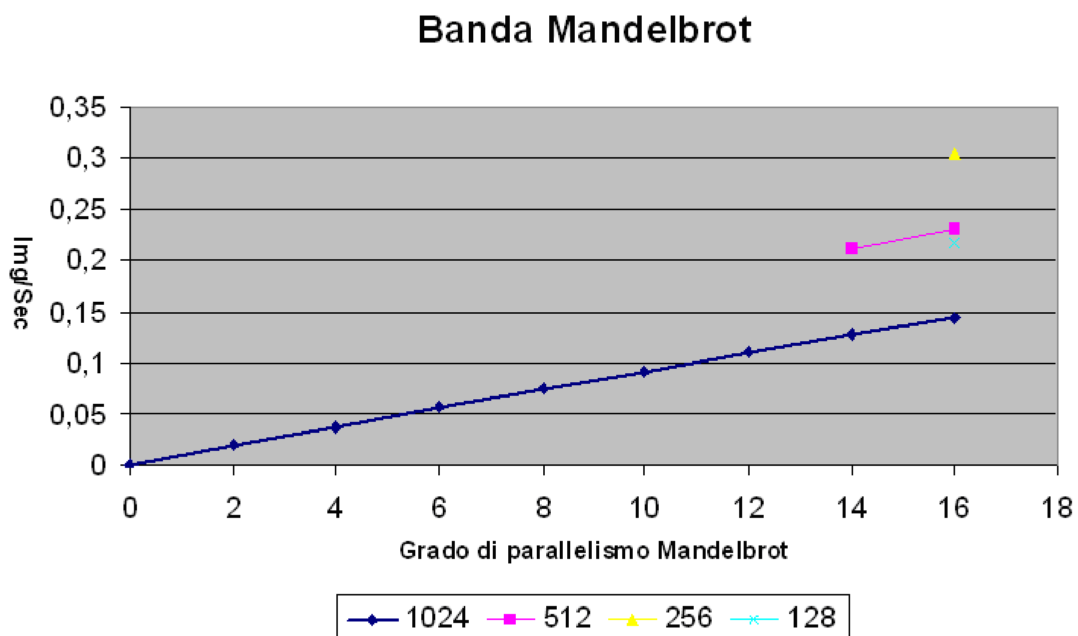



Figura 5.6: Banda del modulo Mandelbrot.

### MpegEncoder

Il modulo MpegEncoder è stato compilato senza informazioni di debug e l'ambiente a runtime forniva il supporto della Clam. In questa fase della sperimentazione è stato calcolato un filmato prendendo ingresso 100 immagini identiche, cioè 10 GOP da 10 immagini ciascuno. Non è stato necessario effettuare tutti gli esperimenti eseguiti per il modulo Mandelbrot, in quanto, dalla prova effettuata con grado di parallelismo 1 (cioè con un solo worker), si evince che è più veloce del modulo Mandelbrot con banda massima. Qui di seguito, quindi, viene riportata la banda del modulo MpegEncoder con grado di parallelismo 1.

| **Grado di parallelismo MpegEncoder** | **Banda (Img/Sec)** |
|---|---|
| 1 | 0.938288 |

Tabella 5.5: Banda di MpegEncoder.

### Prestazioni intera applicazione

Di seguito verranno evidenziati i risultati ottenuti dall'esecuzione della prima versione dell'applicazione sul cluster omogeneo Pianosa. I test sono stati eseguiti con l'applicazione compilata senza informazioni di debug e l'ambiente a run-time non forniva il supporto della Clam. La misura delle prestazioni dell'intera applicazione è stata eseguita avvalendosi di una tecnica che comporta la creazione di alcuni file, che ha permesso di avere una certa consistenza fra il tempo di inizio dell'elaborazione, rilevato da Mandelbrot, ed il tempo di fine elaborazione, rilevato da MpegEncoder. Di fatto, i due moduli vengono eseguiti su macchine diverse, e ciò avrebbe potuto causare problemi di sincronizzazione degli orologi interni di ogni singola macchina, che avrebbero potuto falsare le rilevazioni. Adottando questa tecnica è stato ovviato questo tipo di problema, in quanto, l'intero cluster Pianosa, fa uso di un file system condiviso (sistema di gestione dei file: creazione, modifica, etc.). Nel momento in cui l'applicazione inizia la sua computazione, il modulo genera di Mandelbrot crea il file *log_start* nella directory *home/presti/benchmarck* e, nel momento in cui Mandelbrot

termina il calcolo delle 2420 immagini, viene creato, nella stessa directory, un altro file *log_finish_Mandelbrot* dal modulo *Raccogli*. Il tempo di esecuzione del solo modulo Mandelbrot viene calcolato dalla differenza fra il tempo di creazione del primo file ed il tempo di creazione del secondo file. Per poter calcolare il tempo di esecuzione dell'intera applicazione viene creato un terzo file *log_finish_encoder* dal modulo *Raccogli* di MpegEncoder che esegue anche la differenza fra i tempi di creazione di questo file ed il tempo di creazione del primo file. Le prestazioni dell'intera applicazione vengono espresse nel formato ore:minuti:secondi (durata esecuzione), mentre le prestazioni del modulo Mandelbrot vengo espresse sia nel formato ore:minuti:secondi (durata esecuzione) che nel formato immagini/secondo (banda di esecuzione). La precisione delle rilevazioni non può scendere al di sotto del secondo per come è stata eseguita la rilevazione dei tempi di esecuzione. La tabella 5.6 descrive cinque prove di esecuzione, per ogni grado di parallelismo di Mandelbrot, tutte svolte nello stesso ambiente di test. L'ultima colonna, in cui viene calcolata la media fra i tempi ottenuti, riporta anche i decimi di secondo. Tale tabella evidenzia il tempo di completamento dell'intera applicazione.

| Grado di parallelismo Mandelbrot | Tempo di esecuzione (hh:mm:ss) | | | | | Media (hh:mm:ss,dec) |
|---|---|---|---|---|---|---|
| 8 | 7:01:26 | 7:01:47 | 7:01:11 | 7:02:12 | 7:00:17 | 7:01:22,6 |
| 10 | 5:43:20 | 5:42:16 | 5:42:17 | 5:42:36 | 5:42:50 | 5:42:39,8 |
| 12 | 4:44:36 | 4:43:9 | 4:42:58 | 4:42:53 | 4:42:33 | 4:43:13,8 |
| 14 | 4:19:19 | 4:19:24 | 4:19:0 | 4:19:29 | 4:19:6 | 4:19:15,6 |
| 16 | 3:45:35 | 3:45:21 | 3:45:27 | 3:45:22 | 3:45:34 | 3:45:27,8 |

Tabella 5.6: Prestazioni della prima versione dell'applicazione.

Il grafico 5.7 riporta i valori medi elencati nell'ultima colonna della precedente tabella. Tali valori sono espressi in secondi e devono essere moltiplicati per 1000 al fine di ottenere il valore originale.

La tabella 5.7 riportata descrive cinque prove di esecuzione, per ogni grado di

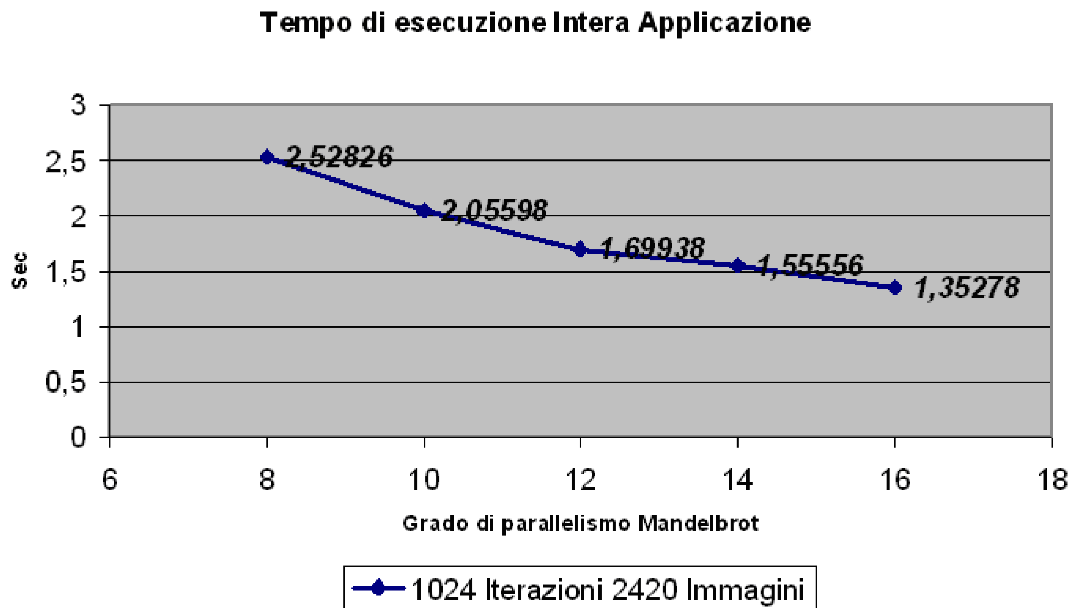



Figura 5.7: Tempo di esecuzione della prima versione dell'applicazione.

parallelismo di Mandelbrot, tutte svolte nello stesso ambiente di test. L'ultima colonna, in cui viene calcolata la media fra i tempi ottenuti, riporta anche i decimi di secondo. Tale tabella evidenzia il tempo di completamento e la banda di esecuzione del solo modulo Mandelbrot.

Il grafico 5.8 riporta i valori medi elencati nell'ultima colonna della precedente tabella. Tali valori sono espressi in immagini al secondo (banda).

| Grado di parallelismo Mandelbrot | Tempo di esecuzione (hh:mm:ss) Img/sec | | | | | Media (hh:mm:ss,dec) Img/sec |
|---|---|---|---|---|---|---|
| 8 | 7:01:14<br>0,095751 | 7:01:35<br>0,095671 | 7:00:59<br>0,095807 | 7:02:00<br>0.095577 | 7:00:05<br>0,096013 | 7:01:10,6<br>0,095764 |
| 10 | 5:43:8<br>0,117544 | 5:42:3<br>0,117916 | 5:42:4<br>0,117911 | 5:42:24<br>0,117721 | 5:42:37<br>0,117796 | 5:42:27,2<br>0,117778 |
| 12 | 4:44:23<br>0,141827 | 4:42:57<br>0,142546 | 4:42:45<br>0,142647 | 4:42:41<br>0,14268 | 4:42:20<br>0,142857 | 4:43:1,2<br>0,142511 |
| 14 | 4:19:6<br>0,155667 | 4:19:12<br>0,155607 | 4:18:48<br>0,155848 | 4:19:16<br>0,155797 | 4:18:53<br>0,155567 | 4:19:3<br>0,155697 |
| 16 | 3:45:22<br>0,178968 | 3:45:8<br>0,179153 | 3:45:15<br>0,17906 | 3:45:9<br>0,178968 | 3:45:22<br>0,17914 | 3:45:15,2<br>0,179058 |

Tabella 5.7: Prestazioni del modulo Mandelbrot nella prima versione dell'applicazione.

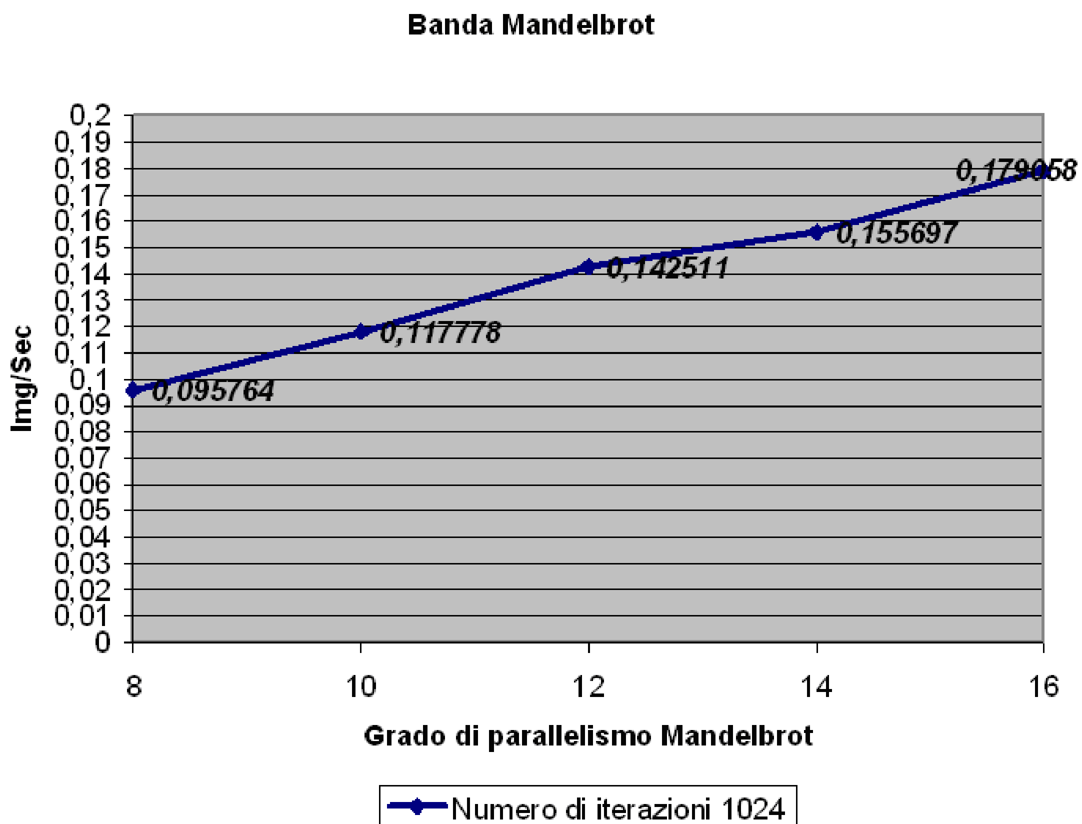



Figura 5.8: Banda di esecuzione del modulo Mandelbrot nella prima versione dell'applicazione.

### Prestazioni dell'applicazione integrata CCM

Di seguito verranno evidenziati i risultati ottenuti dall'esecuzione della prima versione dell'applicazione, integrata con il CCM, sul cluster omogeneo Pianosa. I test sono stati eseguiti con i moduli ASSIST compilati separatamente, senza informazioni di debug e con il supporto della Clam. La misura delle prestazioni dell'intera applicazione è stata eseguita avvalendosi della tecnica descritta precedentemente. Le tabelle 5.8 e 5.9 descrivono cinque prove di esecuzione, con il modulo Mandelbrot a grado di parallelismo 12, tutte svolte nello stesso ambiente di test. L'ultima colonna, in cui viene calcolata la media fra i tempi ottenuti, riporta anche i decimi di secondo.

| Grado di parallelismo Mandelbrot | Tempo di esecuzione (hh:mm:ss) | | | | | Media (hh:mm:ss,dec) |
|---|---|---|---|---|---|---|
| 12 | 4:45:15 | 4:49:35 | 4:46:22 | 4:46:41 | 4:46:29 | 4:46:52,4 |

Tabella 5.8: Prestazioni della prima versione dell'applicazione integrata con il CCM.

| Grado di parallelismo Mandelbrot | Tempo di esecuzione (hh:mm:ss) Img/sec | | | | | Media (hh:mm:ss,dec) Img/sec |
|---|---|---|---|---|---|---|
| 12 | 4:46:13<br>0,14092 | 4:46:24<br>0,14083 | 4:46:06<br>0,14141 | 4:44:58<br>0,14154 | 4:49:19<br>0,13941 | 4:46:36<br>0,14073 |

Tabella 5.9: Prestazioni del modulo Mandelbrot nella prima versione dell'applicazione integrata con il CCM.

Dalle prove svolte si può notare come l'applicazione integrata con il CCM è stata eseguita con dei tempi medi poco più elevati rispetto all'applicazione ASSIST. Questo potrebbe essere giustificato sia dall'introduzione dei meccanismi di conversione per le comunicazioni (da un ambiente Java ad uno ASSIST e viceversa), sia dall'introduzione di nuovi processi in esecuzione sugli stessi nodi. Sono stati effettuati solo questi esperimenti, quindi senza variare il grado di parallelismo di Mandelbrot,

poiché, a causa della eccessiva durata dell'esecuzione dell'intera applicazione, i dati sono risultati poco significativi.

### 5.5.2 Studio dell'applicazione: seconda versione

Il primo studio dell'applicazione ha portato a generare tutti i singoli frame componenti il filmato, cioè il modulo Mandelbrot doveva processare 2420 aree del piano complesso generando altrettante immagini. In questo modo però l'applicazione risultava troppo pesante, i tempi di esecuzione erano nell'ordine delle ore con grado di parallelismo, del modulo Mandelbrot, massimo, e poco significativa, in quanto il carico di lavoro maggiore veniva eseguito da Mandelbrot mentre gli altri due moduli avevano un carico pressoché nullo. Ciò comportava dei tempi di attesa dei dati, da parte di PixConverter e MpegEncoder, molto lunghi e quindi un'elaborazione pressoché sequenziale degli elementi dello stream. Questo ha portato a modificare l'approccio iniziale in modo da avere un carico ben distribuito fra i vari moduli e tempi di elaborazione più contenuti. Di fatto, per ottenere dei dati significativi ai fini del lavoro di tesi, dagli esperimenti eseguiti con la precedente versione dell'applicazione, sarebbe stato necessario impiegare un cluster con un numero di macchine notevolmente maggiore a quello disponibile. La stessa considerazione è stata fatta in previsione degli esperimenti su griglia. Questi fattori hanno dato origine ad un secondo studio dell'applicazione, con nuovi requisiti riguardanti il tempo di completamento e la potenza computazionale richiesta. L'obiettivo è stato quello di ridurre il carico di Mandelbrot, aumentare il carico degli altri due moduli e mantenere un risultato che rispettasse i requisiti iniziali. Per fare ciò è stato deciso che Mandelbrot elaborasse solo 201 aree del piano complesso (sempre rappresentanti uno zoom) e PixConverter convertisse le immagini nel giusto formato e calcolasse un fade fra un'immagine e l'altra, dopodiché MpegEncoder eseguisse il suo calcolo senza alcuna modifica rispetto alla versione precedente.

## Prestazioni

L'ambiente di test in cui sono state svolte le prove è esattamente lo stesso descritto per la versione precedente dell'applicazione. Sono stati eseguiti due tipi di test la cui differenza sta nel lavoro svolto da PixConverter: nel primo test interpone quattro frame d'interpolazione tra un'immagine e l'altra, nell'altro test invece ne interpone 10 (vedi sez. 5.3.2). Tali test sono stati effettuati solamente per decidere quale tipo di filmato utilizzare per proseguire gli esperimenti. Come si vedrà nelle tabelle successive, le prestazioni delle due batterie di test sono paragonabili. Per questo si è scelto di utilizzare per le sperimentazioni successive il secondo test. Come al solito vengono presentati di seguito i risultati ottenuti da tutta l'applicazione e dal solo modulo Mandelbrot, in quanto è ancora quest'ultimo a determinare la banda di elaborazione dell'intera applicazione.

| Grado di parallelismo Mandelbrot | Tempo di esecuzione Primo Test (hh:mm:ss) | Tempo di esecuzione Secondo Test (hh:mm:ss) |
|---|---|---|
| 1 | 04:38:42 | 04:38:15 |
| 2 | 02:18:49 | 02:19:05 |
| 4 | 01:09:29 | 01:09:36 |
| 6 | 00:46:33 | 00:46:54 |
| 8 | 00:35:09 | 00:35:48 |
| 10 | 00:28:27 | 00:28:41 |
| 12 | 00:23:22 | 00:24:59 |
| 14 | 00:20:11 | 00:23:52 |

Tabella 5.10: Prestazioni della seconda versione dell'applicazione.

Il grafico 5.9 riporta il tempo di completamento delle due versioni dell'applicazione al variare del grado di parallelismo di Mandelbrot. Tali valori sono espressi in secondi e devono essere moltiplicati per 1000 al fine di ottenere il valore originale.

Il grafico 5.10 riporta le bande ottenute dal modulo Mandelbrot al variare del suo grado di parallelismo. Tali valori sono espressi in immagini al secondo.

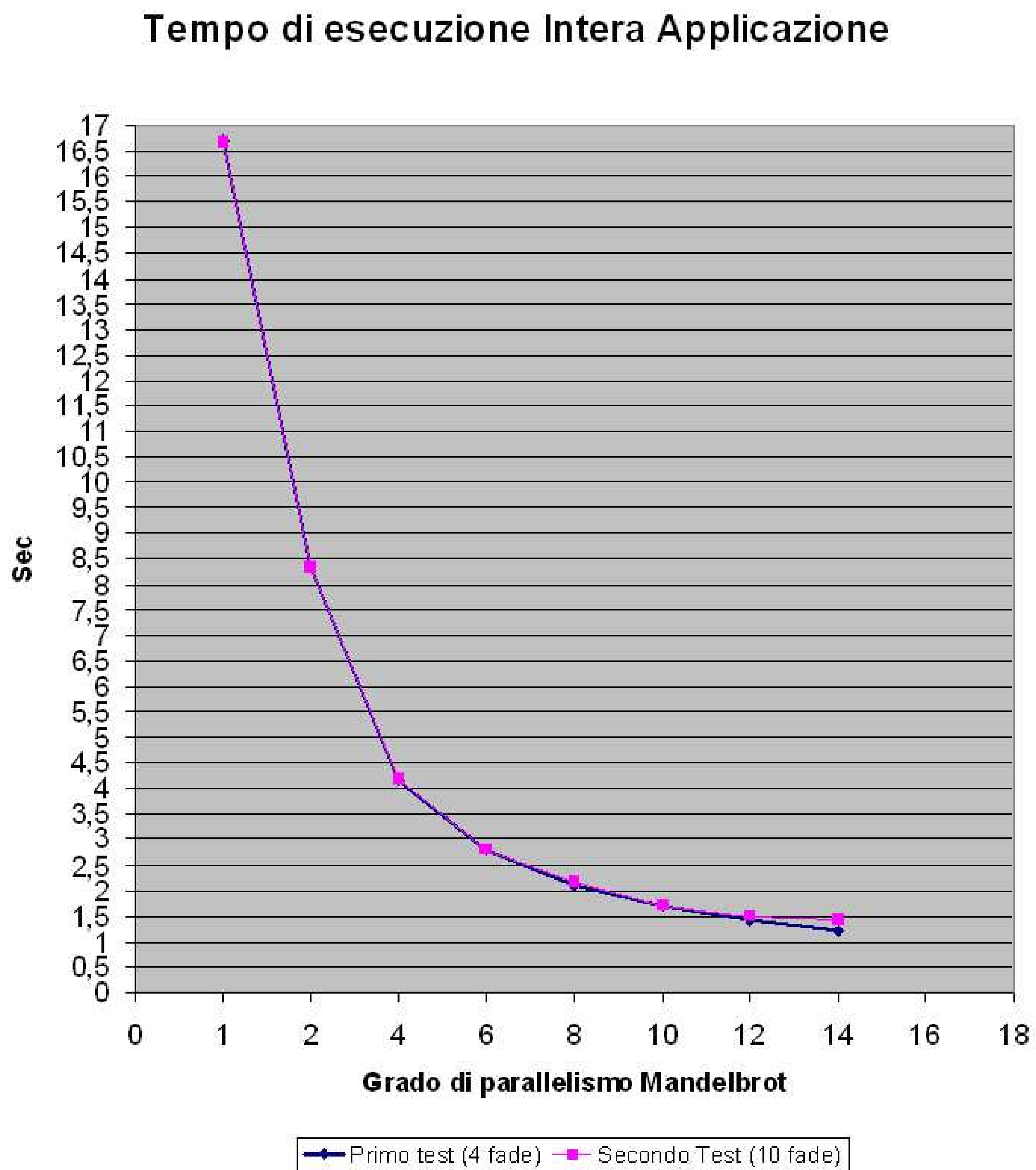



Figura 5.9: Tempo di esecuzione della seconda versione dell'applicazione.

| Grado di parallelismo Mandelbrot | Tempo di esecuzione Primo Test (hh:mm:ss) Img/Sec | Tempo di esecuzione Secondo Test (hh:mm:ss) Img/Sec |
|---|---|---|
| 1 | 04:38:30<br>0,012029 | 04:38:01<br>0,01205 |
| 2 | 02:18:37<br>0,024167 | 02:18:51<br>0,024127 |
| 4 | 01:09:17<br>0,048352 | 01:09:22<br>0,048294 |
| 6 | 00:46:20<br>0,072302 | 00:46:38<br>0,071837 |
| 8 | 00:34:56<br>0,095897 | 00:35:34<br>0,094189 |
| 10 | 00:28:14<br>0,118654 | 00:28:26<br>0,117819 |
| 12 | 00:23:19<br>0,143674 | 00:24:35<br>0,136271 |
| 14 | 00:19:59<br>0,16764 | 00:23:27<br>0,142757 |

Tabella 5.11: Prestazioni del modulo Mandelbrot nella seconda versione dell'applicazione.

| N. Worker Mandelbrot | Banda (Img/Sec) | Scalabilità | Efficienza |
|---|---|---|---|
| 1 | 0,012029 | 1 | 1 |
| 2 | 0,024167 | 2,01 | 1 |
| 4 | 0,048352 | 4,02 | 1 |
| 6 | 0,072302 | 6,01 | 1 |
| 8 | 0,095897 | 7,97 | 0,997 |
| 10 | 0,118654 | 9,86 | 0,986 |
| 12 | 0,143674 | 11,94 | 0,995 |
| 14 | 0,16764 | 13,94 | 0,995 |

Tabella 5.12: Primo Test con 4 fade tra un'immagine e l'altra: scalabilità ed efficienza dell'applicazione.

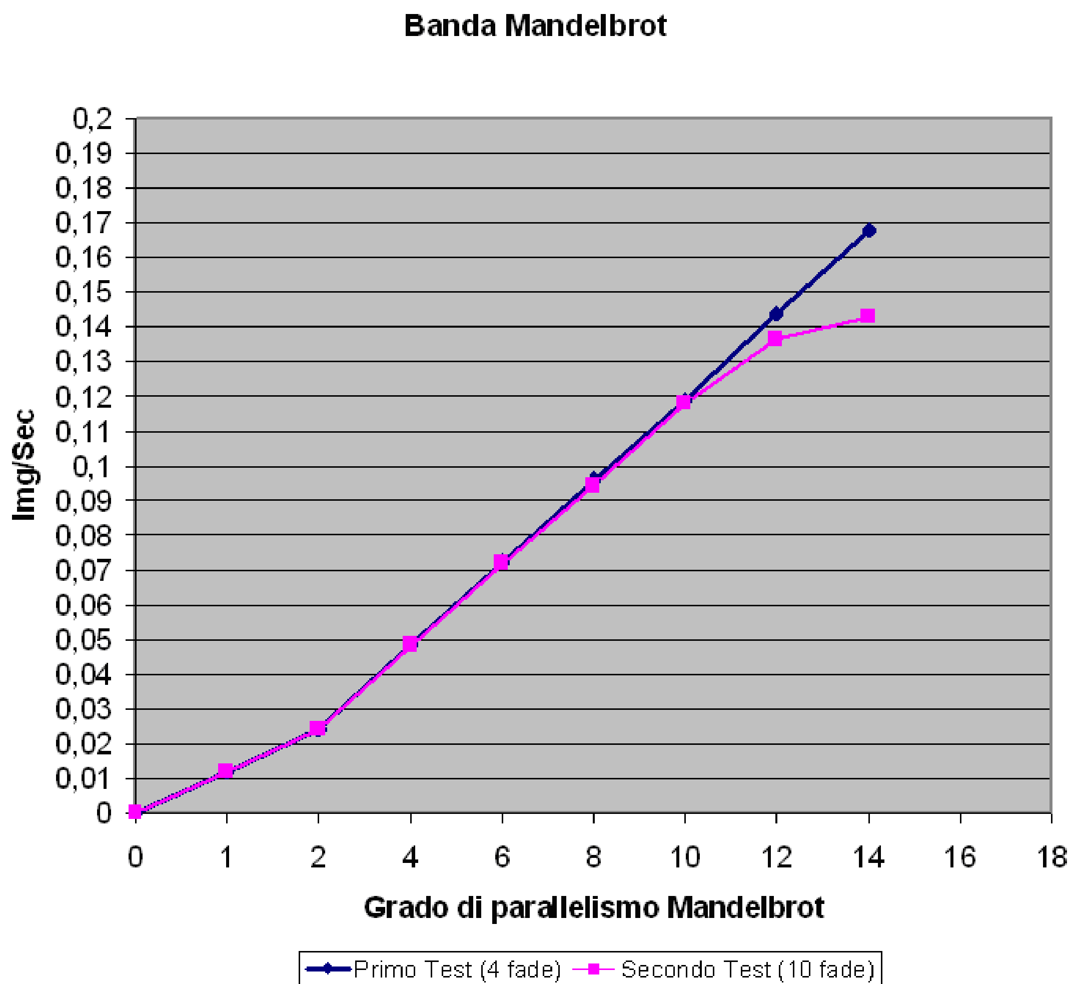



Figura 5.10: Banda di esecuzione del modulo Mandelbrot nella seconda versione dell'applicazione.

| N. Worker Mandelbrot | Banda (Img/Sec) | Scalabilità | Efficienza |
|---|---|---|---|
| 1 | 0,01205 | 1 | 1 |
| 2 | 0,024127 | 2 | 1 |
| 4 | 0,048294 | 4 | 1 |
| 6 | 0,071837 | 5,96 | 0,993 |
| 8 | 0,094189 | 7,82 | 0,977 |
| 10 | 0,117819 | 9,86 | 0,986 |
| 12 | 0,136271 | 11,31 | 0,942 |
| 14 | 0,142757 | 11,84 | 0,846 |

Tabella 5.13: Secondo Test con 10 fade tra un'immagine e l'altra: scalabilità ed efficienza dell'applicazione.

### Prestazioni dell'applicazione integrata CCM

Di seguito verranno evidenziati i risultati ottenuti dall'esecuzione della seconda versione dell'applicazione, integrata con il CCM (secondo test con 10 fade), sul cluster omogeneo Pianosa. I test sono stati eseguiti con i moduli ASSIST compilati separatamente, senza informazioni di debug e con il supporto della Clam. La misura delle prestazioni dell'intera applicazione è stata eseguita avvalendosi della tecnica descritta precedentemente. Le tabelle 5.14 e 5.15 descrivono diverse prove di esecuzione, con il modulo Mandelbrot a parallelismo variabile, tutte svolte nello stesso ambiente di test.

| Grado di parallelismo Mandelbrot | Tempo di esecuzione (hh:mm:ss) |
|---|---|
| 8 | 0:35:14 |
| 10 | 0:28:41 |
| 12 | 0:25:13 |

Tabella 5.14: Prestazioni della seconda versione dell'applicazione integrata con il CCM.

| Grado di parallelismo Mandelbrot | Tempo di esecuzione (hh:mm:ss) (Img/Sec) |
|---|---|
| 8 | 0:34:56<br>0,095943 |
| 10 | 0:28:21<br>0,118166 |
| 12 | 0:24:16<br>0,138049 |

Tabella 5.15: Prestazioni del modulo Mandelbrot nella seconda versione dell'applicazione integrata con il CCM.

Come si evince dalle prove svolte, l'applicazione integrata con il CCM è stata eseguita con dei tempi medi poco più elevati rispetto all'applicazione ASSIST. Quindi,

anche in questa versione dell'applicazione come nella precedente, l'overhead introdotto dall'integrazione con il CCM non risulta significativo in un ambiente come quello dei cluster omogenei.

## 5.6 Esperimenti sulla versione finale dell'applicazione

La terza versione dell'applicazione è quella finale che ha consentito di raggiungere gli scopi posti dalla tesi stessa, cioè la valutazione del modello sperimentale proposto: integrazione fra ASSIST ed il framework CCM per applicazioni su griglia. Inoltre, gli esperimenti sono stati orientati a validare la bontà della politica adattiva dell'applicazione proposta alla dinamicità dell'elaborazione descritta in precedenza, implementata dal modulo ApplicationManager introdotto in questa versione dell'applicazione.

In questa sezione vengono riportati i risultati degli esperimenti riguardanti l'esecuzione dell'applicazione ASSIST ed integrata CCM e la politica adottata dall'ApplicationManager nei due casi. I test sono stati eseguiti su tre diversi ambienti in modo da poter confrontare i risultati raggiunti:

- cluster omogeneo: Pianosa (Dipartimento di Informatica di Pisa);
- cluster eterogenei ma appartenenti alla stessa rete dipartimentale: Pianosa e C1 (Dipartimento di Informatica di Pisa);
- griglia composta dai due cluster Pianosa e C1, dal desktop Capraia e da alcuni desktop appartenenti ad un'altra rete (ISTI-CNR di Pisa).

Nei grafici mostrati in seguito (**A**) vengono riportati i tre tempi rilevati da PixConverter, ed inviati ad ApplicationManager, che riguardano: il tempo di interarrivo dei dati da Mandelbrot (INPUT_PIX), il tempo necessario per inviare i due GOP non compressi ad MpegEncoder (ASSIST_OUT_PIX), il tempo di inattività trascorso nell'attesa dell'immagine da Mandelbrot (IDLE_PIX). I grafici (**B**) relativi ai

tempi rilevati da StreamStore (INPUT_STREAM_STORE), poiché non necessari ai fini della sperimentazione, vengono inseriti e descritti nell'appendice B. Successivamente verranno descritti i grafici più significativi, in cui si evidenziano la dinamicità dell'elaborazione dell'applicazione, il collo di bottiglia in Mandelbrot ed in MpegEncoder. Infine vengono riportati i dati riassuntivi relativi alla banda di elaborazione dell'applicazione ottenuta nei vari esperimenti. Inoltre, per ogni test, l'applicazione è stata eseguita variando il grado di parallelismo dei moduli che la costituiscono, in modo da definire meglio i risultati ottenuti. Il titolo di ogni grafico spiega la configurazione dell'applicazione che è stata eseguita (e.g. Mandelbrot 6 - MpegEncoder 2), in quali macchine viene eseguito MpegEncoder (...su C1) e se l'applicazione è stata integrata nel CCM (....CCM).

### 5.6.1 Prestazioni sul cluster omogeneo Pianosa

#### Valutazione dell'impatto dell'integrazione ASSIST-CCM

Questa sezione è dedicata alla valutazione dell'impatto dell'integrazione di ASSIST con CCM in ambiente omogeneo (Pianosa). Vengono quindi illustrati gli esperimenti sia dell'applicazione ASSIST sia dell'applicazione integrata CCM per poter valutare l'eventuale overhead introdotto da CCM. Il modulo ApplicationManager, in questi primi esperimenti, non è stato dotato di alcuna politica adattiva poiché l'interesse è stato focalizzato sulla suddetta comparazione al variare del parallelismo interno dei moduli Mandelbrot ed MpegEncoder. Nonostante ciò i tempi registrati da ApplicationManager hanno consentito la costituzione di alcuni grafici (di seguito ne sono evidenziati una parte, ma la raccolta completa si trova nell'appendice B) dai quali risultano chiari i risultati ottenuti. I tempi registrati e calcolati da ApplicationManager sono stati memorizzati in un file di log che ha aiutato la costruzione dei grafici.

Nei grafici sono riportati i secondi sull'asse delle ordinate mentre su quelle delle ascisse gli elementi dello stream in oggetto (da Mandelbrot a PixConverter). Dopo tutti gli esperimenti su Pianosa è stato generato un ultimo grafico che evidenzia

la banda dell'applicazione, espressa in immagini al secondo, per ogni sua diversa configurazione.

Di seguito sono presentati tre grafici 5.11, 5.12 e 5.13, rispettivamente uno in cui il collo di bottiglia è Mandelbrot, uno in cui è MpegEncoder ed infine uno in cui si nota la dinamicità dell'applicazione descritta in precedenza. I tre grafici si riferiscono all'esecuzione dell'applicazione in versione ASSIST in diverse configurazioni. Dal grafico 5.11 si evince facilmente come Mandelbrot risulti il modulo collo di

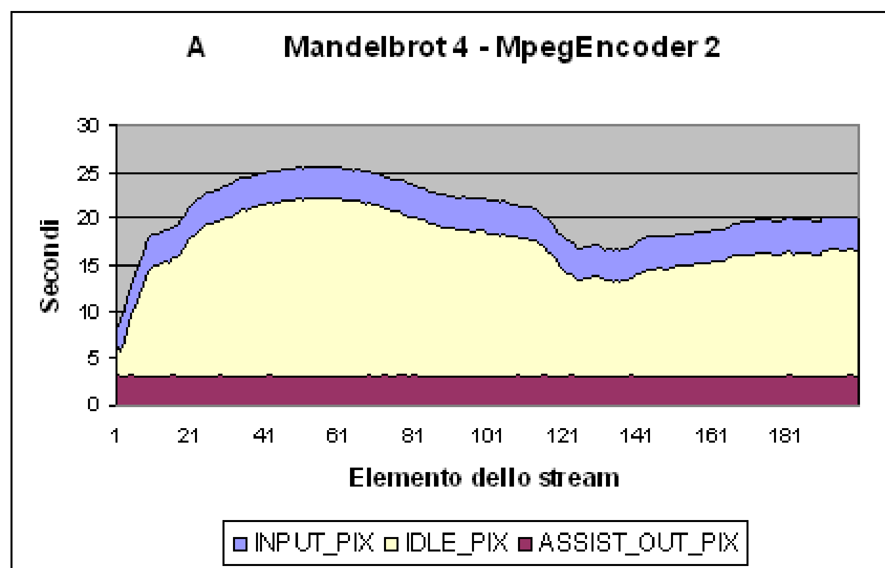



Figura 5.11: Tempi relativi all'esecuzione dell'applicazione ASSIST su Pianosa: collo di bottiglia in Mandelbrot.

bottiglia permanentemente durante l'esecuzione dell'applicazione, poiché il tempo che PixConverter trascorre aspettando i dati (IDLE_PIX) da Mandelbrot non scende mai al di sotto dei dieci secondi. Nel grafico 5.12, invece, il tempo IDLE_PIX è pressoché nullo ed il tempo ASSIST_OUT_PIX è elevato per tutta la durata della computazione. Da ciò si evince che il modulo collo di bottiglia è MpegEncoder in quanto il tempo di comunicazione dei dati da PixConverter ad MpegEncoder è prevalente rispetto al tempo di calcolo di PixConverter. Inoltre Mandelbrot deve attendere che PixConverter sia in grado di ricevere il dato trasmesso prima di poter riprendere la sua computazione. Infine, nel grafico 5.13, viene evidenziata la

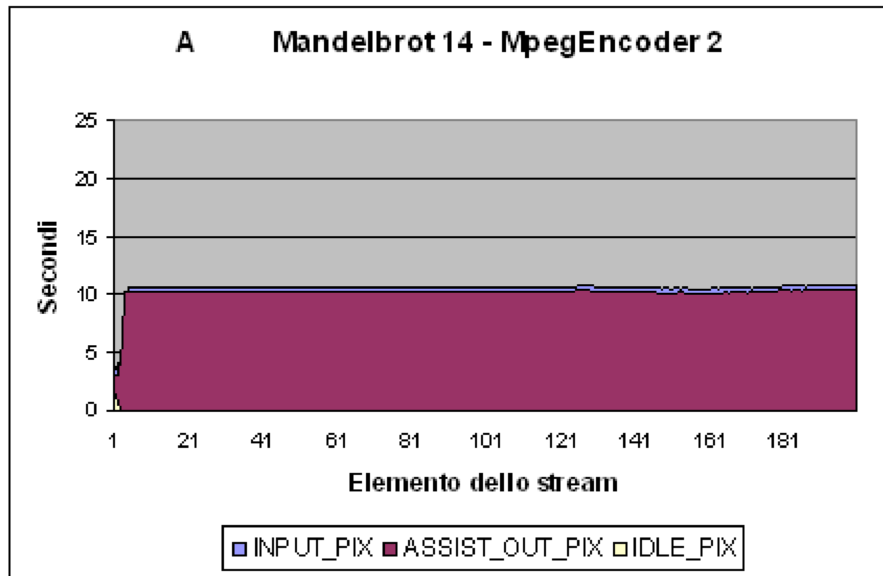



Figura 5.12: Tempi relativi all'esecuzione dell'applicazione ASSIST su Pianosa: collo di bottiglia in MpegEncoder.

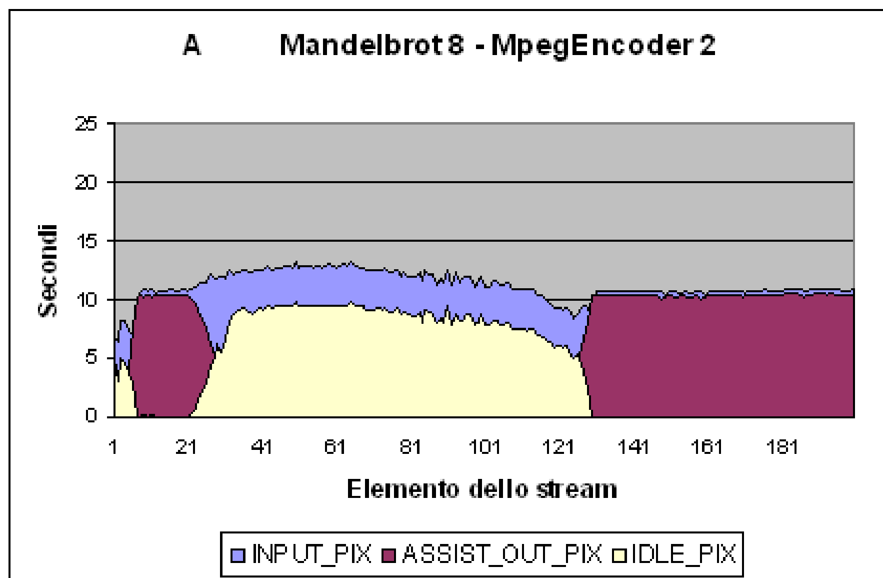



Figura 5.13: Tempi relativi all'esecuzione dell'applicazione ASSIST su Pianosa: dinamicità applicazione.

dinamicità dell'applicazione che verrà utilizzata nei successivi test di adattività. Infatti dopo una prima fase in cui il collo di bottiglia risulta MpegEncoder, poichè si verificano le condizioni appena descritte e poiché Mandelbrot, come detto, risulta costituito da tre diverse fasi di elaborazione, si passa ad una successiva fase in cui il tempo di IDLE_PIX cresce notevolmente e ASSIST_OUT_PIX diminuisce e di conseguenza il collo di bottiglia si sposta in Mandelbrot. In questa fase il calcolo delle immagini dell'insieme di Mandelbrot risulta più gravoso e quindi il carico del modulo MpegEncoder si alleggerisce. Infine, nell'ultima fase del grafico e dell'esecuzione dell'applicazione, il collo di bottiglia torna ad essere MpegEncoder, poiché il tempo di IDLE_PIX ritorna ad essere nullo, il tempo di ASSIST_OUT_PIX ritorna ad essere intorno ai dieci secondi e quindi MpegEncoder ritorna ad essere sovraccarico. Le tre situazioni finora descritte ricorrono più volte, nelle diverse configurazioni dell'applicazione che vengono riportate in appendice.

Di seguito sono presenti tre grafici 5.14, 5.15 e 5.16, che si riferiscono all'esecuzione dell'applicazione integrata con il CCM, con grado di parallelismo dei moduli Mandelbrot ed MpegEncoder uguale ai test sopra descritti. Ciò è utile per poter confrontare l'applicazione ASSIST con la versione integrata CCM nelle stesse condizioni. Dal confronto dei grafici 5.11 e 5.14, in entrambi i casi, risulta che Mandelbrot è il modulo collo di bottiglia permanentemente durante l'esecuzione dell'applicazione, poiché il tempo che PixConverter trascorre aspettando i dati (IDLE_PIX) da Mandelbrot non scende mai al di sotto dei dieci secondi. Inoltre l'integrazione con il CCM non ha introdotto overhead significativi in quanto le comunicazioni continuano ad essere ampiamente sovrapposte al calcolo e quindi il piccolo aumento del tempo di ASSIST_OUT_PIX nell'integrazione con il CCM non ha avuto impatti significativi sulla performance dell'applicazione, come evidenziato nei grafici finali riassuntivi delle bande di esecuzione (figg. 5.17 e 5.18). Dal confronto invece dei grafici 5.12 e 5.15, anche se il collo di bottiglia rimane MpegEncoder, si vede che l'overhead introdotto nelle comunicazioni è più consistente. Infatti dal grafico 5.12 tale tempo rimane costante intorno ai dieci secondi, invece nel grafico 5.15 ha delle oscillazioni

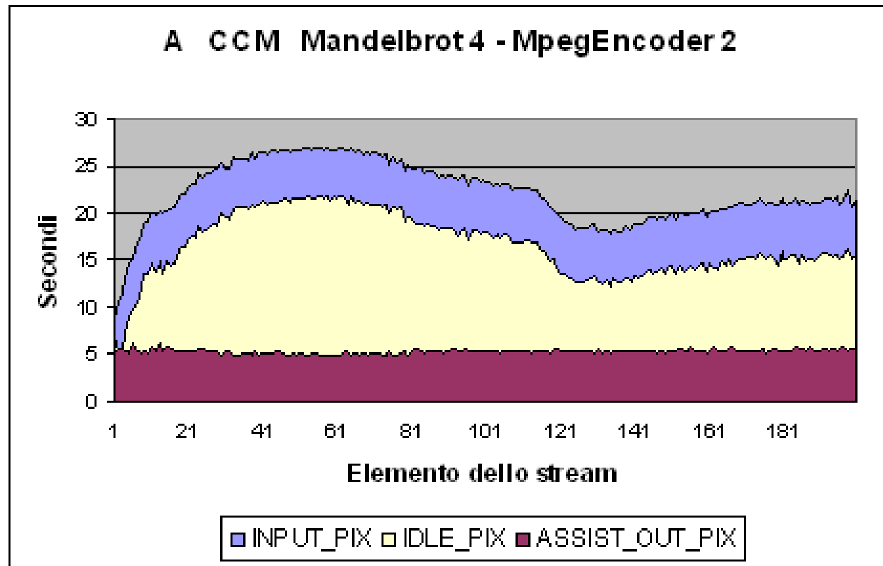



Figura 5.14: Tempi relativi all'esecuzione dell'applicazione ASSIST integrata con il CCM su Pianosa: collo di bottiglia in Mandelbrot.

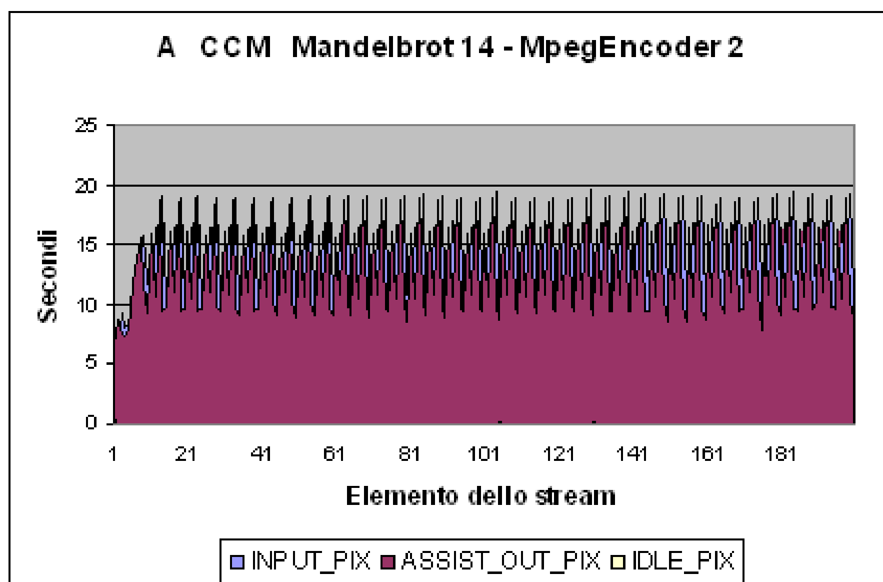



Figura 5.15: Tempi relativi all'esecuzione dell'applicazione ASSIST integrata con il CCM su Pianosa: collo di bottiglia in MpegEncoder (1).

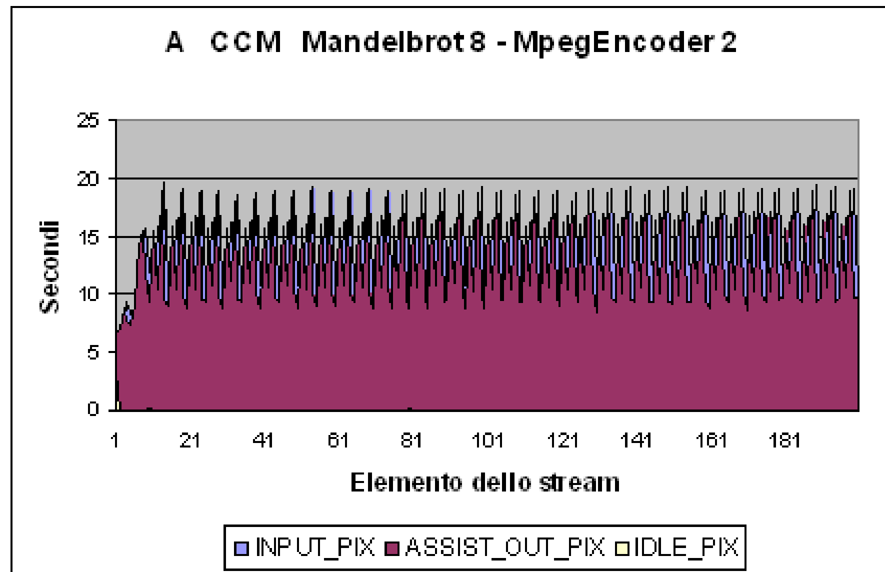



Figura 5.16: Tempi relativi all'esecuzione dell'applicazione ASSIST integrata con il CCM su Pianosa: collo di bottiglia in MpegEncoder (2).

dai dieci ai venti secondi. Come vedremo successivamente anche la banda dell'applicazione risente di questa situazione. Infine dal confronto dei grafici 5.13 e 5.16 si nota che la dinamicità dell'applicazione è stata persa a causa dell'overhead introdotto nel CCM nelle comunicazioni tra PixConverter ed MpegEncoder, che hanno spostato il modulo collo di bottiglia in MpegEncoder per tutta la computazione. Ciò ha avuto una ripercussione sulla banda totale dell'applicazione come si vede dai grafici successivi.

Di seguito vengono riportati due grafici riassuntivi (figg. 5.17 e 5.18) di tutti gli esperimenti effettuati su Pianosa con l'applicazione ASSIST ed integrata CCM. Sull'asse delle ordinate viene riportata la banda media, espressa in immagini al secondo, dell'applicazione e sull'asse delle ascisse il grado di parallelismo di Mandelbrot nei vari test. Per ogni grado di parallelismo di Mandelbrot, inoltre, sono riportati i valori relativi ai diversi gradi di parallelismo in MpegEncoder con i quali sono stati svolti gli esperimenti. I grafici vengono corredati dalle tabelle dei valori ottenuti in cui vengono messi in evidenza i casi in cui si è verificata dinamicità nell'esecuzione dell'applicazione.

| | 1 | 2 | 4 | 6 | 8 | 10 | 12 | 14 | 16 |
|---|---|---|---|---|---|---|---|---|---|
| Parallelismo Mpeg 1 | 0,011893 | 0,023963 | 0,045934 | 0,047611 | 0,048081 | 0,048081 | 0,048012 | 0,048104 | 0,048089 |
| Parallelismo Mpeg 2 | | | 0,04806 | 0,072138 | 0,089948 | 0,095375 | 0,09566 | 0,095784 | 0,095724 |
| Parallelismo Mpeg 3 | 0,011935 | | 0,048049 | 0,071626 | 0,093351 | 0,111102 | 0,135934 | 0,143236 | 0,143004 |
| Parallelismo Mpeg 4 | | | 0,048058 | 0,071419 | 0,094817 | 0,110261 | 0,140391 | 0,164836 | 0,179126 |

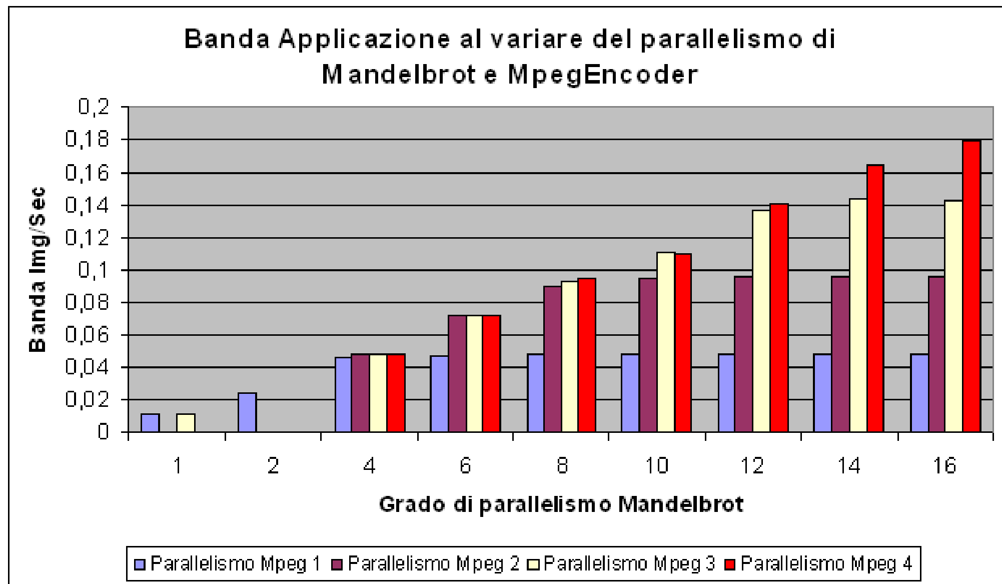



Figura 5.17: Bande dell'intera applicazione su Pianosa nelle sue diverse configurazioni.

| CCM | 1 | 2 | 4 | 6 | 8 | 10 | 12 | 14 | 16 |
|---|---|---|---|---|---|---|---|---|---|
| Parallelismo Mpeg 1 | | | 0,044187 | 0,046364 | 0,04647 | 0,046423 | | | |
| Parallelismo Mpeg 2 | | | 0,045051 | 0,06755 | 0,075377 | 0,075348 | 0,07538 | 0,075284 | |
| Parallelismo Mpeg 3 | | | | | 0,089997 | 0,104673 | 0,107169 | 0,107033 | |
| Parallelismo Mpeg 4 | | | | | | | 0,114573 | 0,112769 | |
| Parallelismo Mpeg 5 | | | | | | | 0,114931 | | |

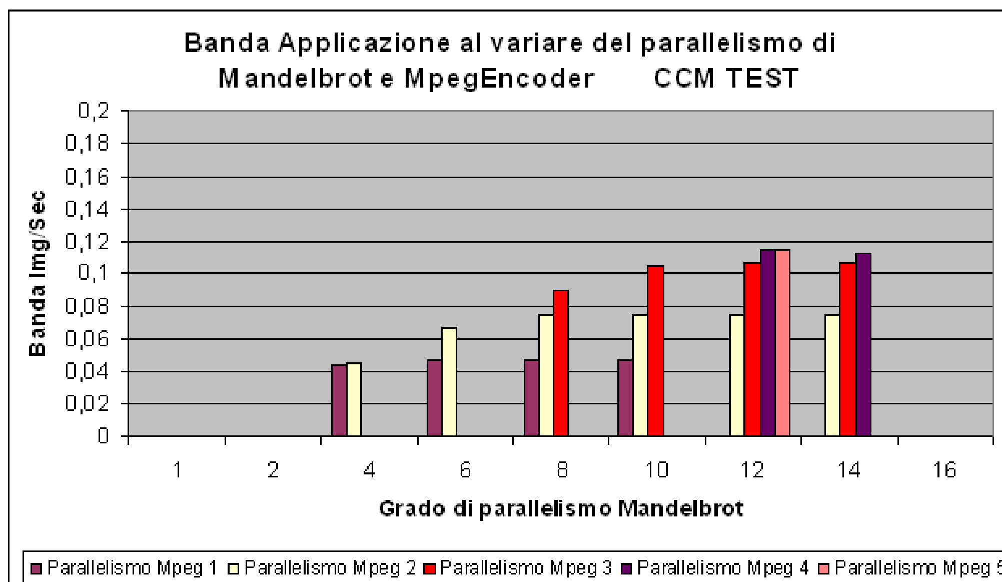



Figura 5.18: Bande dell'intera applicazione integrata con il CCM su Pianosa nelle sue diverse configurazioni.

In prima istanza si può notare dai grafici come le bande dell'applicazione raggiunte nelle due versioni (ASSIST e CCM) sono paragonabili, e quindi l'integrazione non aggiunge overhead significativi, fino a quando il grado di parallelismo di Mandelbrot non supera 8. Appena avviene ciò l'applicazione ASSIST risulta più scalabile rispetto a quella integrata in quanto le comunicazioni sono più pesanti nella versione CCM. Inoltre i casi in cui si presenta la dinamicità nell'applicazione si spostano, nella versione CCM, in gradi di parallelismo per MpegEncoder più alti rispetto alla versione ASSIST.

#### Politica adattiva dell'applicazione

Dagli esperimenti appena discussi si è evidenziato che l'applicazione in alcune configurazioni particolari mostra un comportamento dinamico dato dalla particolare elaborazione del modulo Mandelbrot precedentemente descritta. In questa sezione vengono quindi presentati i risultati degli esperimenti effettuati introducendo la politica adattiva, descritta in precedenza nel modulo ApplicationManager. Di fatto in questo caso viene fatto uso di entrambi le implementazioni dell'MpegEncoder che utilizzano un numero di risorse diverso tra loro. Successivamente verranno presentate due delle prove effettuate sull'adattività dell'applicazione in versione ASSIST e integrata nel CCM. In questo modo è stato possibile valutare la politica adottata in entrambi i casi. In ognuno dei due esperimenti l'applicazione ha iniziato la sua esecuzione con l'utilizzo della versione del modulo MpegEncoder che utilizza meno risorse (lento).

Nel grafico 5.19 vengono mostrati i tempi registrati dal modulo ApplicationManager nel suo file di log durante l'esecuzione dell'applicazione in versione ASSIST. I tempi rilevati sono quelli che si sono descritti in precedenza. Dal grafico sono evidenti i momenti in cui il modulo ApplicationManager decide di cambiare l'implementazione del modulo MpegEncoder da utilizzare in base alla politica descritta. Da questo grafico si evince che la politica adattiva utilizzata consente da un lato di risparmiare l'utilizzo delle risorse, in quanto il modulo MpegEncoder che utilizza

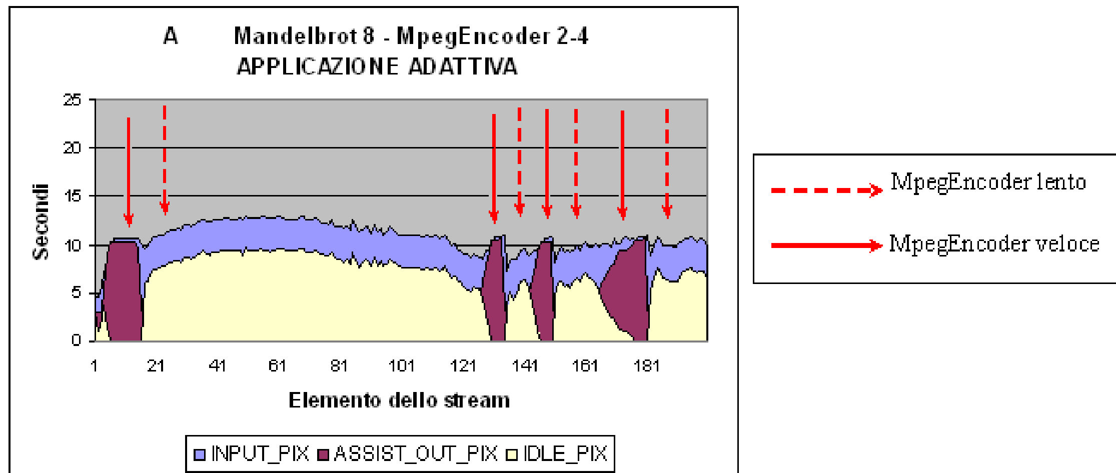



Figura 5.19: Tempi relativi all'esecuzione dell'applicazione ASSIST su Pianosa: adozione della politica di adattività.

più risorse non viene utilizzato costantemente per tutta la durata dell'elaborazione, dall'altro consente di raggiungere prestazioni paragonabili all'utilizzo costante del modulo MpegEncoder veloce, come si vede dalla tabella 5.16.

| Parallelismo MpegEncoder | Banda Img/Sec |
|---|---|
| 2 | 0,089948 |
| 4 | 0,094817 |
| 2-4 (politica adattiva) | 0,092756 |

Tabella 5.16: Bande ottenute dall'applicazione su Pianosa con Mandelbrot a grado di parallelismo 8 con e senza politica adattiva.

Nel grafico 5.20 vengono mostrati i tempi registrati dal modulo ApplicationManager nel suo file di log durante l'esecuzione dell'applicazione integrata con il CCM. Anche in questo caso dal grafico sono evidenti i momenti in cui il modulo ApplicationManager decide di cambiare l'implementazione del modulo MpegEncoder da utilizzare in base alla politica descritta. Dal grafico si evince inoltre che la politica adattiva utilizzata consente, come prima, di risparmiare l'utilizzo delle risorse e di raggiungere prestazioni paragonabili all'utilizzo costante del modulo MpegEncoder veloce, come si vede dalla tabella 5.17.

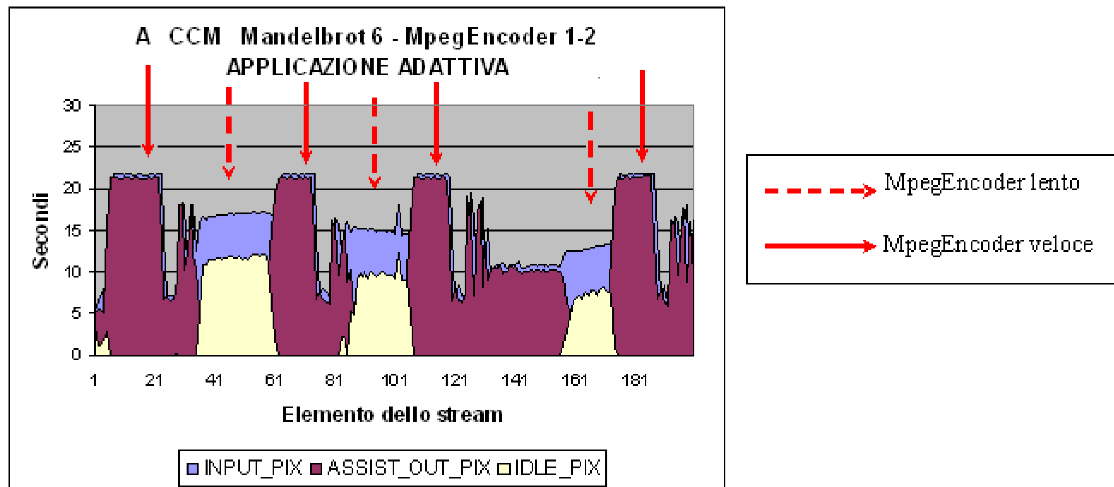



Figura 5.20: Tempi relativi all'esecuzione dell'applicazione integrata CCM su Pianosa: adozione della politica di adattività.

| Parallelismo MpegEncoder | Banda Img/Sec |
|---|---|
| 1 | 0,046364 |
| 2 | 0,06755 |
| 1-2 (politica adattiva) | 0,065098 |

Tabella 5.17: Bande ottenute dall'applicazione integrata CCM su Pianosa con Mandelbrot a grado di parallelismo 6 con e senza politica adattiva.

### 5.6.2 Prestazioni sui cluster eterogenei Pianosa e C1

#### Valutazione dell'impatto dell'integrazione ASSIST-CCM

In questa fase della sperimentazione il modulo MpegEncoder utilizzato per i test è stato allocato sul cluster C1, mentre il resto dell'applicazione sul cluster Pianosa. In questo modo i dati trasmessi a/da MpegEncoder transitano su una rete dipartimentale. Non essendo questa rete dedicata ed avendo due cluster a disposizione come piattaforma d'esecuzione, è stato possibile verificare l'entità dell'overhead introdotto dal CCM in un ambiente eterogeneo rispetto a quello precedente (solo Pianosa). Di fatto i risultati ottenuti (vedi figg. 5.25 e 5.26) confermano il trend evidenziato negli esperimenti in ambiente omogeneo. In particolare fino a quando Mandelbrot ha un grado di parallelismo inferiore a 8-10 l'overhead introdotto con l'integrazione con il CCM è piuttosto contenuto, poiché le comunicazioni hanno un minimo aggravio. Invece con Mandelbrot a grado di parallelismo elevato 12-16, l'integrazione con il CCM introduce un overhead che rende l'applicazione non scalabile. Infatti la banda dell'applicazione, in queste condizioni, si attesta su valori costanti anche cambiando il grado di parallelismo nel modulo MpegEncoder. La versione ASSIST dell'applicazione invece continua a scalare anche con grado di parallelismo in Mandelbrot uguale a 16, anche se l'aumento dei worker nel modulo MpegEncoder non produce alcun miglioramento. Anche per quello che riguarda la dinamicità dell'applicazione si può fare un discorso simile a quello fatto nella sezione precedente relativa ai test in ambiente omogeneo. Di fatto i casi di dinamicità dell'applicazione si spostano, nella versione CCM, in gradi di parallelismo per MpegEncoder più alti rispetto alla versione ASSIST.

Di seguito sono riportati alcuni grafici che mostrano il comportamento dell'applicazione, sia in versione ASSIST che integrata CCM, con il modulo Mandelbrot con grado di parallelismo dapprima basso (4, overhead introdotto non significativo, vedi figg. 5.21 e 5.22) e poi con grado di parallelismo alto (16, overhead significativo, vedi figg. 5.23 e 5.24).

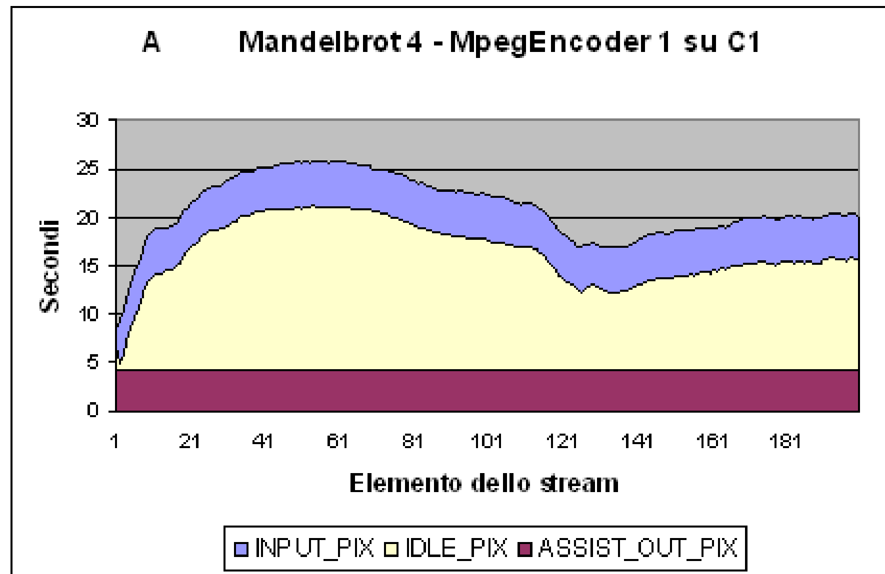



Figura 5.21: Tempi relativi all'esecuzione dell'applicazione ASSIST su Pianosa e C1: Mandelbrot con grado di parallelismo 4.

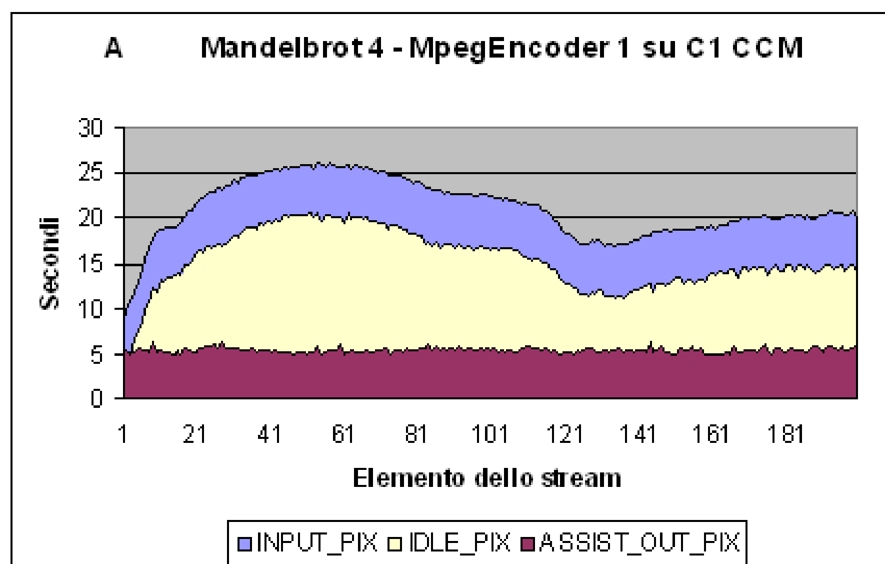



Figura 5.22: Tempi relativi all'esecuzione dell'applicazione ASSIST integrata con il CCM su Pianosa e C1: Mandelbrot con grado di parallelismo 4.

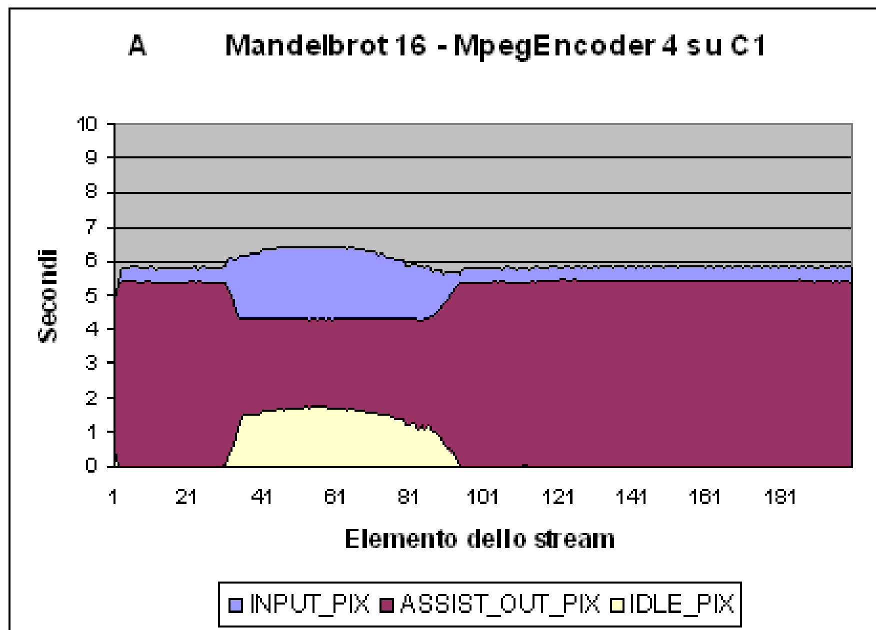



Figura 5.23: Tempi relativi all'esecuzione dell'applicazione ASSIST su Pianosa e C1: Mandelbrot con grado di parallelismo 16.

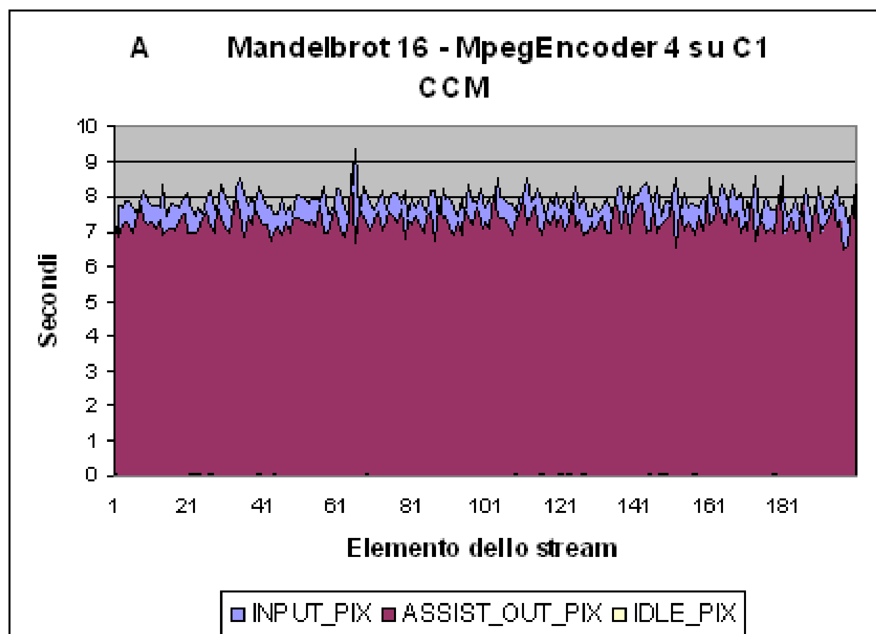
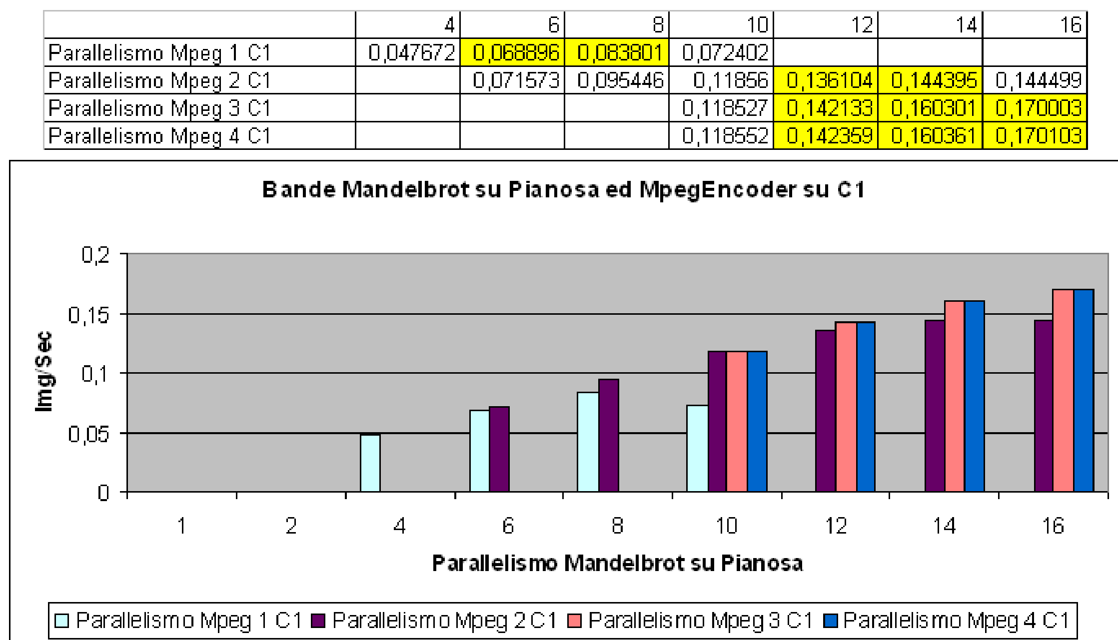



Figura 5.24: Tempi relativi all'esecuzione dell'applicazione ASSIST integrata con il CCM su Pianosa e C1: Mandelbrot con grado di parallelismo 16.

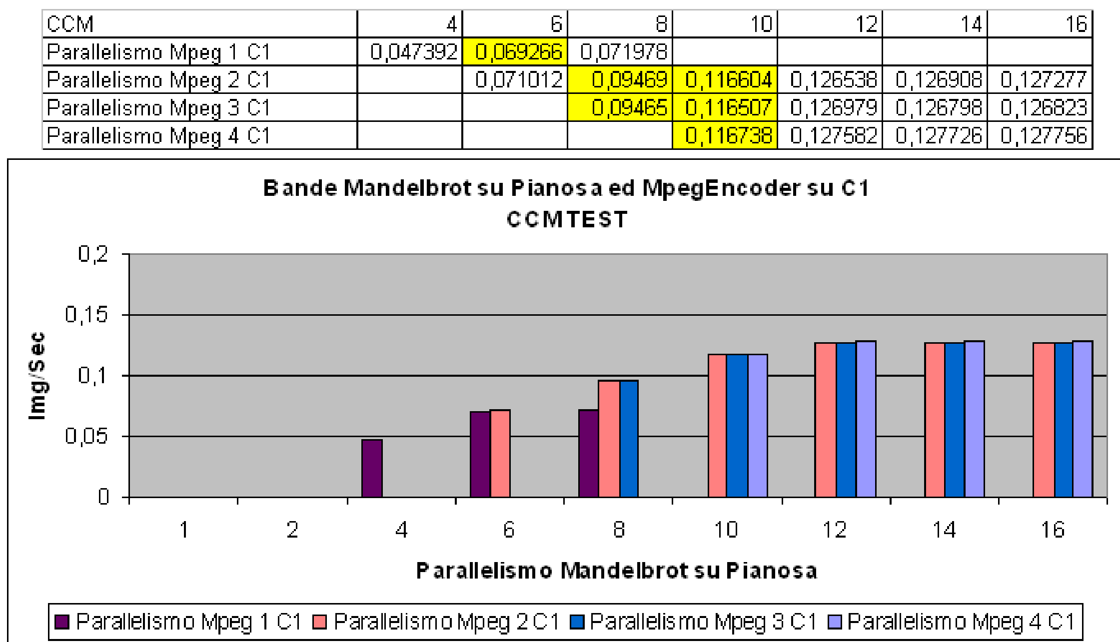
Di seguito vengono riportati anche i due grafici riassuntivi (figg. 5.25 e 5.26) di tutti gli esperimenti effettuati su Pianosa e C1 con l'applicazione ASSIST ed integrata CCM. Sull'asse delle ordinate di tali grafici viene riporta la banda media, espressa in immagini al secondo, dell'applicazione, e sull'asse delle ascisse il grado di parallelismo di Mandelbrot nei vari test. Per ogni grado di parallelismo di Mandelbrot inoltre sono riportati i valori relativi ai diversi gradi di parallelismo in MpegEncoder con i quali sono stati svolti gli esperimenti. I grafici vengono corredati dalle tabelle dei valori ottenuti in cui vengono messi in evidenza i casi in cui si è verificata dinamicità nell'esecuzione dell'applicazione.



| | 4 | 6 | 8 | 10 | 12 | 14 | 16 |
|---|---|---|---|---|---|---|---|
| Parallelismo Mpeg 1 C1 | 0,047672 | 0,068896 | 0,083801 | 0,072402 | | | |
| Parallelismo Mpeg 2 C1 | | 0,071573 | 0,095446 | 0,11856 | 0,136104 | 0,144395 | 0,144499 |
| Parallelismo Mpeg 3 C1 | | | | 0,118527 | 0,142133 | 0,160301 | 0,170003 |
| Parallelismo Mpeg 4 C1 | | | | 0,118552 | 0,142359 | 0,160361 | 0,170103 |



Figura 5.25: Bande dell'intera applicazione su Pianosa e C1 nelle sue diverse configurazioni.



| CCM | 4 | 6 | 8 | 10 | 12 | 14 | 16 |
|---|---|---|---|---|---|---|---|
| Parallelismo Mpeg 1 C1 | 0,047392 | 0,069266 | 0,071978 | | | | |
| Parallelismo Mpeg 2 C1 | | 0,071012 | 0,09469 | 0,116604 | 0,126538 | 0,126908 | 0,127277 |
| Parallelismo Mpeg 3 C1 | | | 0,09465 | 0,116507 | 0,126979 | 0,126798 | 0,126823 |
| Parallelismo Mpeg 4 C1 | | | | 0,116738 | 0,127582 | 0,127726 | 0,127756 |



Figura 5.26: Bande dell'intera applicazione integrata con il CCM su Pianosa e C1 nelle sue diverse configurazioni.

### Politica adattiva dell'applicazione

Dagli esperimenti appena discussi, si è evidenziato che l'applicazione in alcune configurazioni particolari mostra un comportamento dinamico dato dalla particolare elaborazione del modulo Mandelbrot descritta precedentemente. In questa sezione vengono quindi presentati i risultati degli esperimenti effettuati introducendo la politica adattiva descritta in precedenza nel modulo ApplicationManager. Le due diverse implementazioni del modulo MpegEncoder sono questa volta allocate sul cluster C1. Successivamente verranno presentate due prove effettuate sull'adattività dell'applicazione in versione ASSIST e integrata nel CCM. In questo modo è stato possibile valutare la politica adottata in entrambi i casi. In ognuno dei due esperimenti l'applicazione ha iniziato la sua esecuzione con l'utilizzo della versione del modulo MpegEncoder che utilizza meno risorse (lento).

Nel grafico 5.27 vengono mostrati i tempi registrati dal modulo ApplicationManager nel suo file di log. I tempi rilevati sono quelli descritti in precedenza. Dal grafico sono evidenti i momenti in cui il modulo ApplicationManager decide di cambiare l'implementazione del modulo MpegEncoder da utilizzare in base alla politica descritta. Da questo grafico si evince che la politica adattiva utilizzata consente da

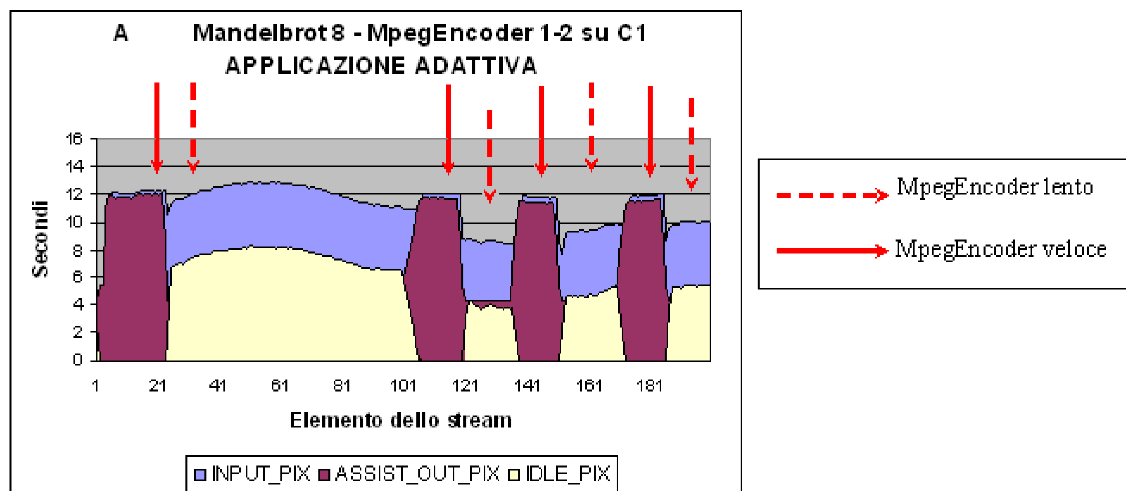



Figura 5.27: Tempi relativi all'esecuzione dell'applicazione ASSIST su Pianosa e C1: adozione della politica di adattività.

un lato di risparmiare l'utilizzo delle risorse, in quanto il modulo MpegEncoder, che utilizza più risorse, non viene utilizzato costantemente per tutta la durata dell'elaborazione, dall'altro consente di raggiungere prestazioni paragonabili all'utilizzo costante del modulo MpegEncoder veloce, come si vede dalla tabella 5.18.

| Parallelismo MpegEncoder | Banda Img/Sec |
|---|---|
| 1 | 0,083801 |
| 2 | 0,095446 |
| 1-2 (politica adattiva) | 0,090208 |

Tabella 5.18: Bande ottenute dall'applicazione su Pianosa e C1 con Mandelbrot a grado di parallelismo 8 con e senza politica adattiva.

Nel grafico 5.28 vengono mostrati i tempi registrati dal modulo ApplicationManager nel suo file di log durante l'esecuzione dell'applicazione integrata CCM. Anche in questo caso dal grafico sono evidenti i momenti in cui il modulo ApplicationManager decide di cambiare l'implementazione del modulo MpegEncoder da utilizzare in base alla politica descritta. Dal grafico risulta inoltre che la politica adattiva

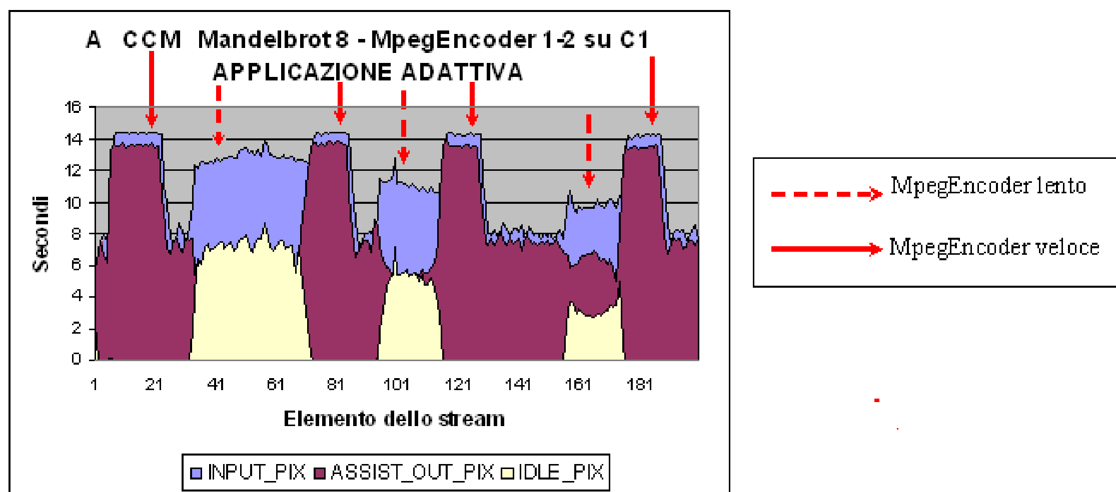



Figura 5.28: Tempi relativi all'esecuzione dell'applicazione integrata CCM su Pianosa e C1: adozione della politica di adattività.

utilizzata consente come prima di risparmiare l'utilizzo delle risorse e di raggiungere prestazioni paragonabili all'utilizzo costante del modulo MpegEncoder veloce, come si vede dalla tabella 5.19.

| Parallelismo MpegEncoder | Banda Img/Sec |
|---|---|
| 1 | 0,071978 |
| 2 | 0,09469 |
| 1-2 (politica adattiva) | 0,089081 |

Tabella 5.19: Bande ottenute dall'applicazione integrata CCM su Pianosa e C1 con Mandelbrot a grado di parallelismo 8 con e senza politica adattiva.

### 5.6.3 Prestazioni sulla griglia

#### Valutazione dell'impatto dell'integrazione ASSIST-CCM

Dopo aver sperimentato l'integrazione dell'applicazione in ambiente omogeneo ed eterogeneo, l'ultima fase di test ha riguardato la sperimentazione su griglia. Di fatto gli esperimenti precedentemente descritti sono stati propedeutici per lo svolgimento delle fasi finali del lavoro di tesi. In particolare, i risultati ottenuti hanno costituito un termine di paragone per i risultati degli esperimenti di questa fase di test. I moduli dell'applicazione sono stati allocati nel seguente modo (vedi fig. 5.29):

- Mandelbrot sul cluster omogeneo Pianosa;
- PixConverter sul cluster omogeneo C1;
- MpegEncoder (utilizzato) su alcune macchine del CNR di Pisa (Rubentino, Sangiovese, Cavit, Verduzzo);
- StreamStore sul desktop Capraia;
- ApplicationManager sul cluster omogeneo C1.

Per maggiori informazioni riguardanti la piattaforma di esecuzione dei test si faccia riferimento alla sezione 5.4.

Anche in questa fase di test l'applicazione sperimentata era composta, all'inizio, solo da moduli ASSIST, in seguito con i moduli integrati nei componenti CCM.

A differenza dei precedenti risultati, questi nuovi esperimenti hanno dato un esito ben diverso. Di fatto, mentre nei casi precedenti l'integrazione con il CCM

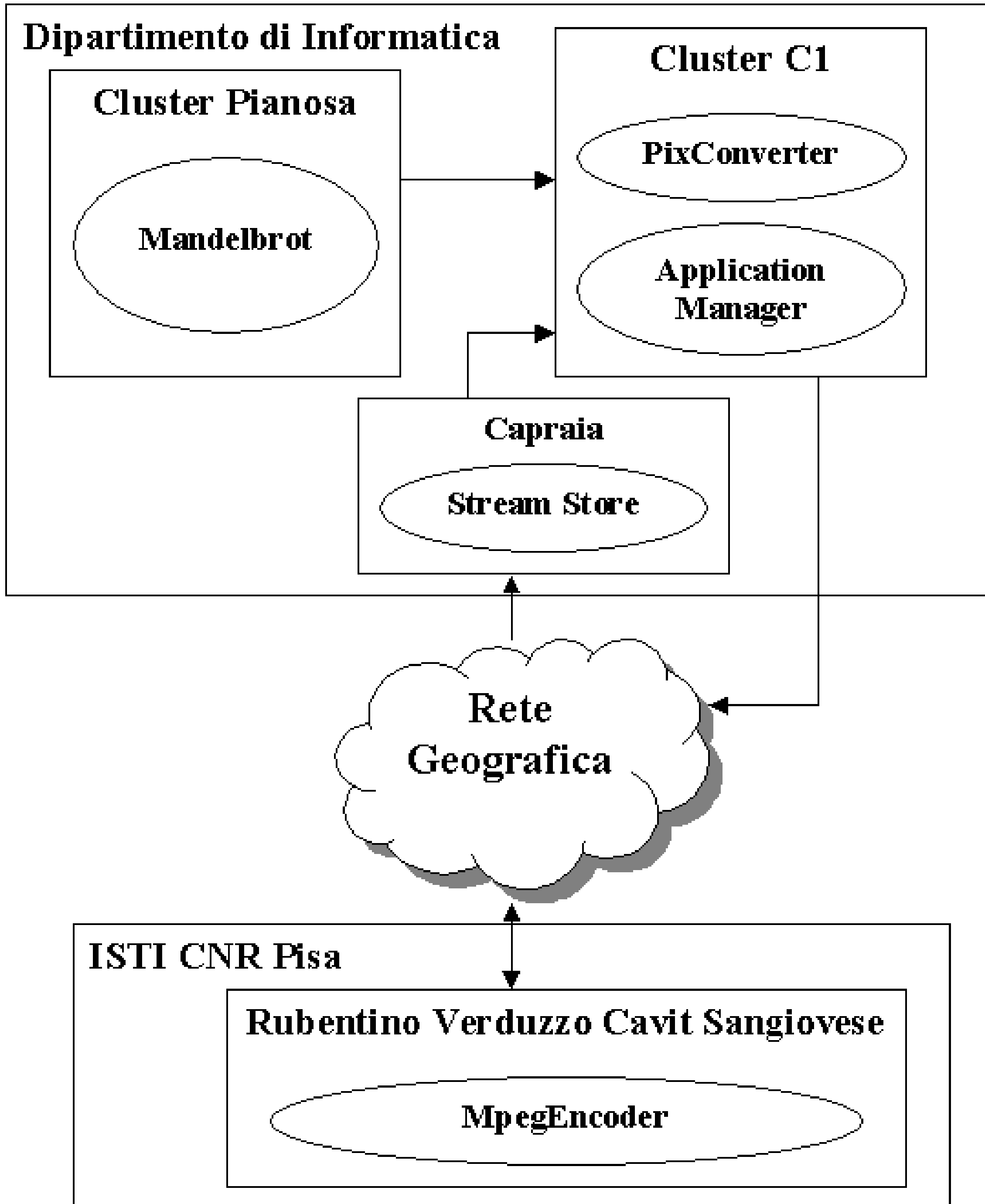



Figura 5.29: Allocazione dei moduli dell'applicazione su griglia.

introduceva un'overhead nelle comunicazioni ed una minore banda di elaborazione per i gradi di parallelismo di Mandelbrot elevati, questa batteria di test dimostra come l'integrazione con CCM è più efficiente rispetto all'applicazione ASSIST in tutte le configurazioni testate (vedi fig. 5.36).

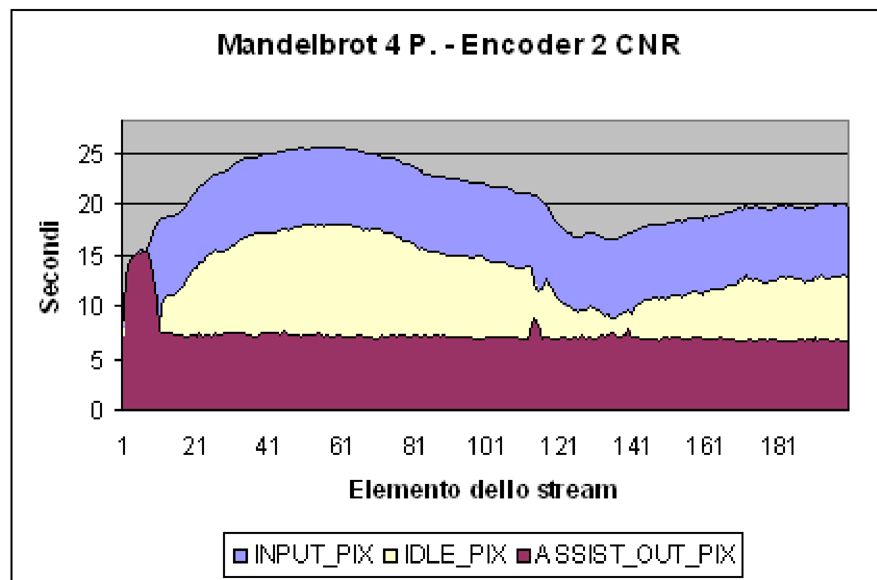



Figura 5.30: Tempi relativi all'esecuzione dell'applicazione ASSIST su griglia: Mandelbrot con grado di parallelismo 4.

Come evidenziato dai grafici, il tempo di trasmissione dei dati al modulo MpegEncoder (ASSIST_OUT_PIX) allocato sulle macchine del CNR è inferiore, nel caso dell'integrazione con CCM (fig. 5.31), rispetto a quello rilevato nel caso della sola applicazione ASSIST (fig. 5.30). In questo caso però la banda di elaborazione è la stessa in quanto, per via delle prestazioni del singolo modulo Mandelbrot, le comunicazioni vengono sovrapposte al calcolo. Non solo, all'aumentare del grado di parallelismo di Mandelbrot (figg. 5.32, 5.33, 5.34 e 5.35) questo divario risulta più evidente ed aumenta la banda di elaborazione dell'applicazione (integrata CCM) in quanto le comunicazioni fra componenti, essendo in parte sovrapposte al calcolo, sono più efficienti delle comunicazioni fra moduli ASSIST, che risultano più sincrone. La causa di questi risultati sta nel fatto che, per tutti i test eseguiti, il tempo di ASSIST_OUT_PIX è stato maggiore nella versione ASSIST dell'applicazione.

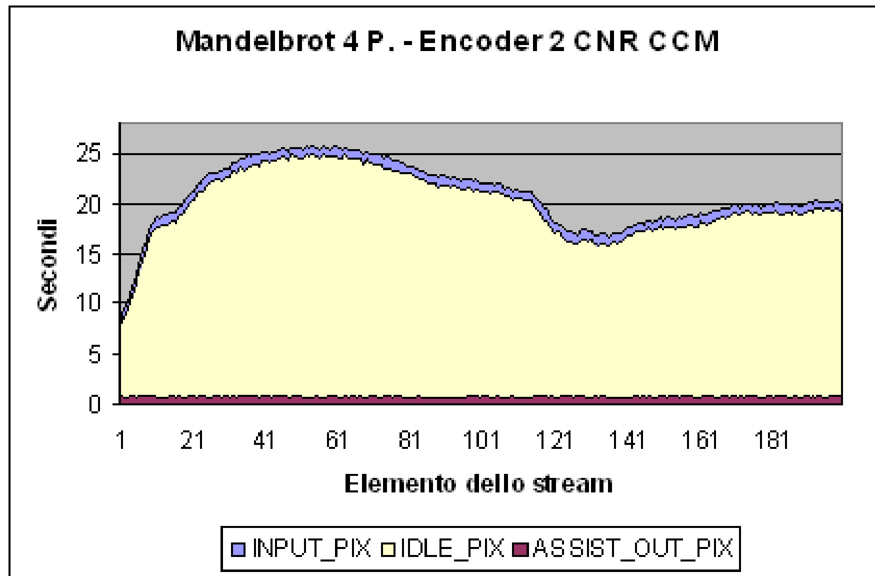



Figura 5.31: Tempi relativi all'esecuzione dell'applicazione ASSIST integrata con il CCM su griglia: Mandelbrot con grado di parallelismo 4.

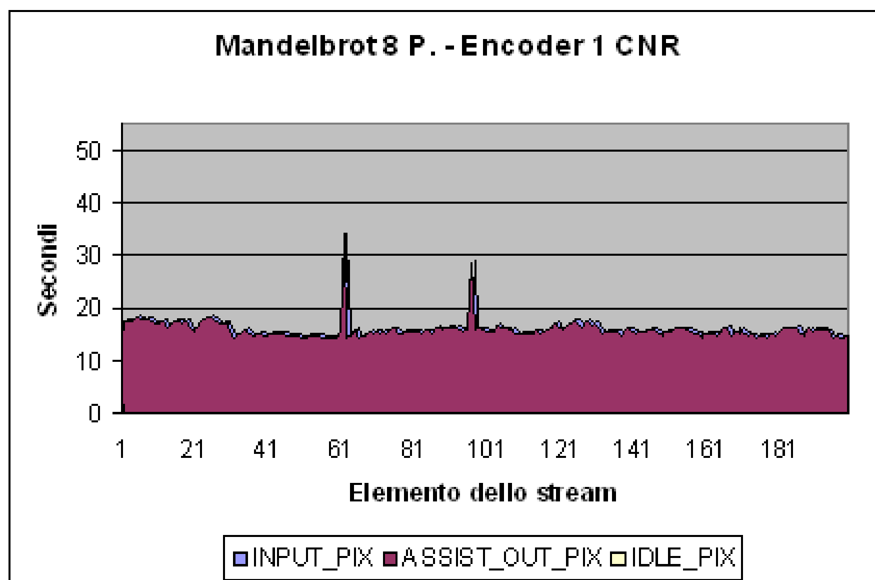



Figura 5.32: Tempi relativi all'esecuzione dell'applicazione ASSIST su griglia: Mandelbrot con grado di parallelismo 8.

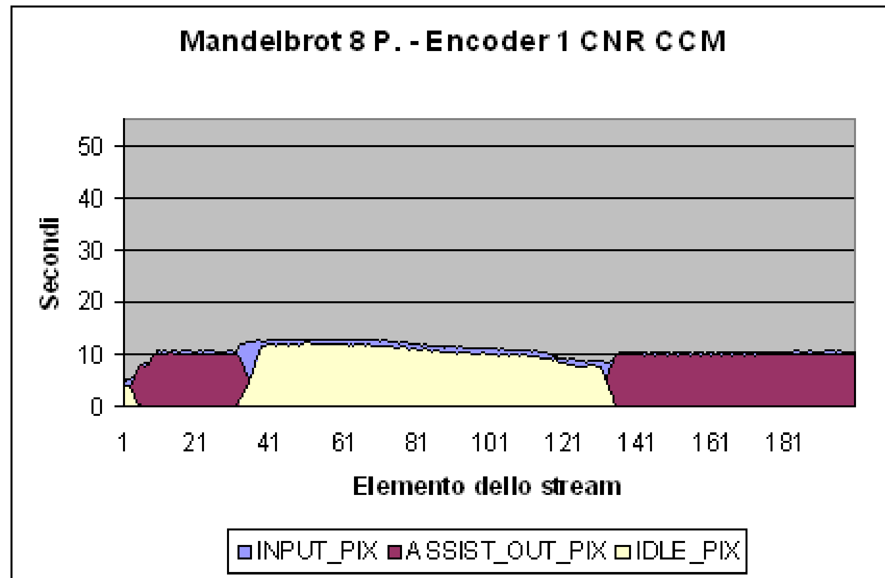



Figura 5.33: Tempi relativi all'esecuzione dell'applicazione ASSIST integrata con il CCM su griglia: Mandelbrot con grado di parallelismo 8.

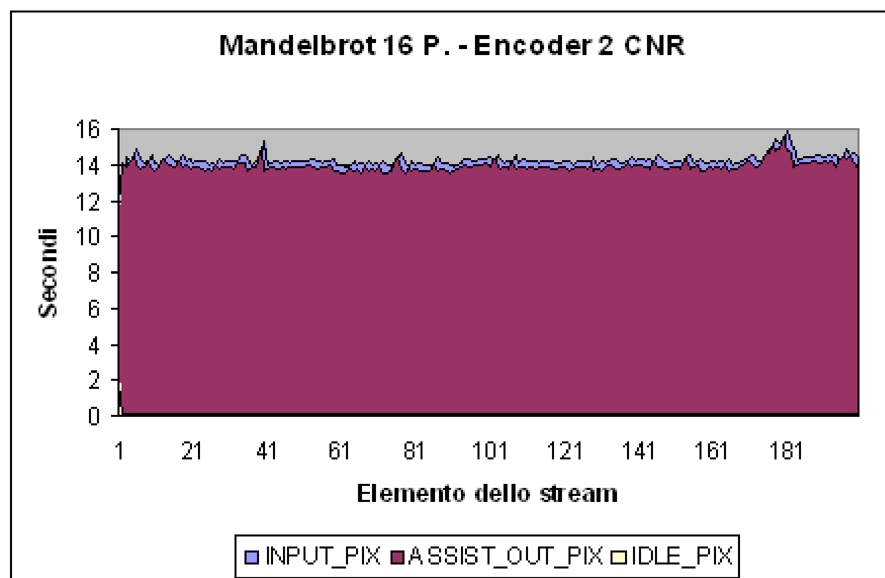



Figura 5.34: Tempi relativi all'esecuzione dell'applicazione ASSIST su griglia: Mandelbrot con grado di parallelismo 16.

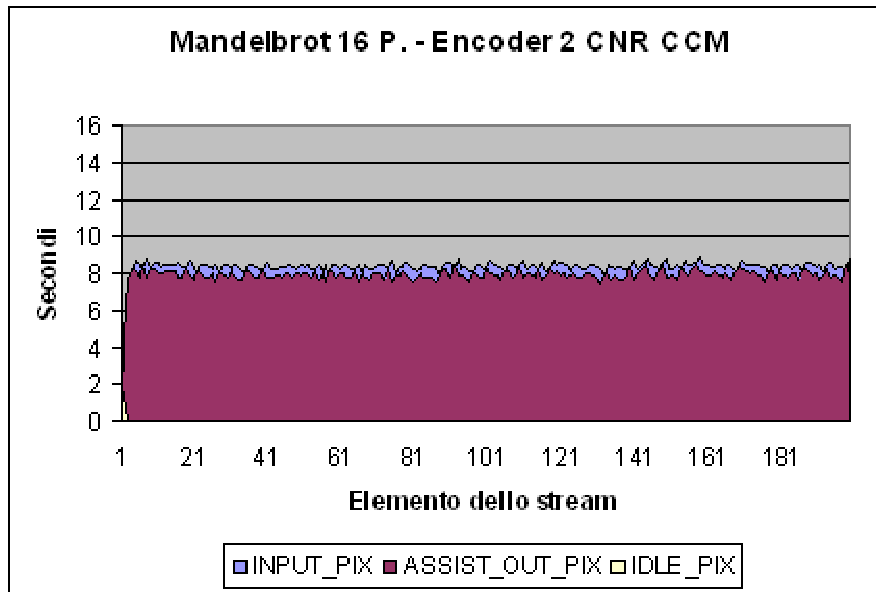



Figura 5.35: Tempi relativi all'esecuzione dell'applicazione ASSIST integrata con il CCM su griglia: Mandelbrot con grado di parallelismo 16.

Dal grafico riassuntivo 5.36, riguardante la banda di elaborazione delle due versioni dell'applicazione (ASSIST ed integrata CCM), si evince che la versione ASSIST non è scalabile su griglia, mentre per quello che riguarda la versione integrata CCM essa risulta scalabile sino al grado di parallelismo di Mandelbrot 12. In questo grafico, come negli altri, sono stati evidenziati i casi in cui l'applicazione ha presentato un comportamento dinamico.

| | 4 | 6 | 8 | 10 | 12 | 14 | 16 |
|---|---|---|---|---|---|---|---|
| Parallelismo Mpeg 1 CNR | 0,04786 | | 0,062349 | | 0,061209 | | 0,070811 |
| Parallelismo Mpeg 2 CNR | 0,048188 | | 0,056922 | | 0,064023 | | 0,07058 |
| Parallelismo Mpeg 1 CNR CCM | 0,047826 | | 0,093064 | | 0,096996 | | 0,096918 |
| Parallelismo Mpeg 2 CNR CCM | 0,047818 | | 0,095509 | | 0,120336 | | 0,120701 |

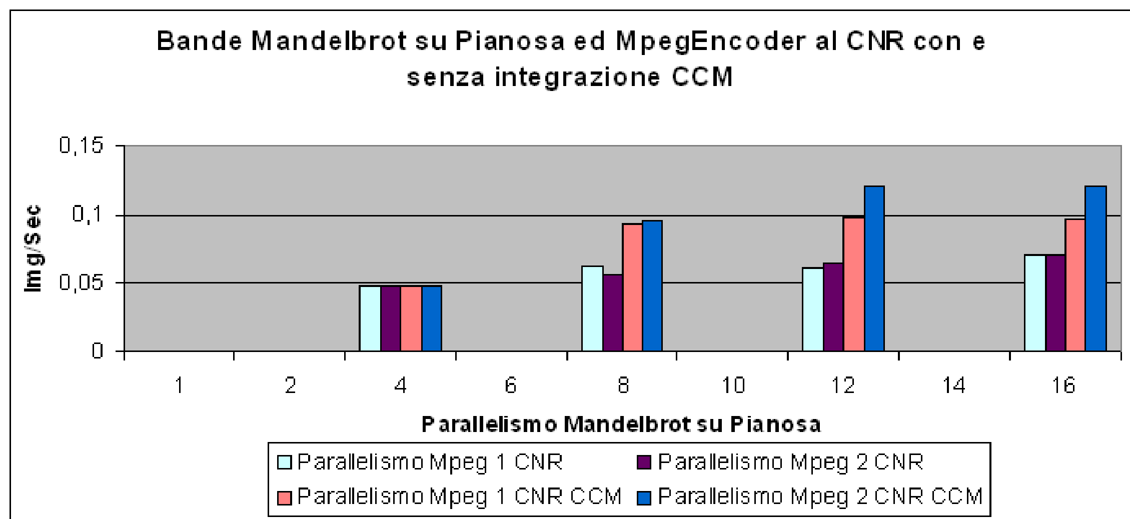



Figura 5.36: Bande dell'intera applicazione integrata con il CCM e non, su griglia, nelle sue diverse configurazioni.

A differenza degli altri ambienti di test, in questo non sono stati effettuati esperimenti sulla politica adattiva dell'applicazione, implementata dal modulo ApplicationManager, poiché dalle prove eseguite in precedenza non si sono verificati casi in cui la sperimentazione di tale politica fosse stata significativa. Infatti, nei casi in cui si sono evidenziati dei comportamenti dinamici nell'elaborazione dell'applicazione (Mandelbrot con grado di parallelismo 4 e 8, MpegEncoder con grado di parallelismo 1, vedi fig. 5.36), la banda di esecuzione è rimasta praticamente immutata all'aumentare del numero di risorse per MpegEncoder, in quanto si è raggiunto un limite nella banda di comunicazione che non consente di migliorare le prestazioni dell'applicazione in modo sensibile. Quindi l'utilizzo della politica adattiva in questi casi non avrebbe generato alcun risultato utile alla sua valutazione.

### 5.6.4 Conclusioni

Dal confronto fra gli esperimenti su griglia (sez. 5.6.3) e gli esperimenti su cluster omogeneo (sez. 5.6.1) ed eterogenei (sez. 5.6.2) si è evidenziata un'inversione di tendenza nelle prestazioni dell'applicazione ASSIST ed integrata CCM. Di fatto i risultati ottenuti hanno dimostrato che dall'applicazione integrata CCM si raggiungono migliori performance dovute alla minor durata delle comunicazioni. Una possibile spiegazione potrebbe essere il diverso supporto alle comunicazioni utilizzato dai due ambienti. Mentre ASSIST, sviluppato e sperimentato per ambienti omogenei, utilizza un protocollo di comunicazione adatto ad una rete dedicata ad alte prestazioni, CCM, sviluppato per ambienti distribuiti, utilizza un protocollo adatto a reti non dedicate e non ottimizzato per alte prestazioni. Dai test effettuati si evince che CCM è più efficiente se la banda di rete massima si aggira attorno ai 32 Mbit/sec[2], mentre ASSIST è più efficiente se la banda di rete massima si aggira attorno ai 100 Mbit/sec o superiore.

Si possono formulare diverse ipotesi sull'implementazione del supporto alle comunicazioni nei due ambienti:

[2]Banda di rete media registrata fra il Dipartimento di Informatica e il CNR di Pisa.

- in ASSIST la presenza di un buffer limitato, dove vengono memorizzati i dati prima di essere inviati, potrebbe rallentare le comunicazioni, nel caso in cui venga saturato, per due motivi: banda di rete bassa e/o modulo troppo veloce nel riempire il buffer;

- in CCM poiché il protocollo di comunicazione è più lento, nel caso in cui la rete di trasmissione è anch'essa lenta, l'efficienza, cioè il rapporto fra queste due grandezze, cresce.

Quelle appena elencate sono ipotesi formulate in base ai risultati ottenuti dagli esperimenti. È necessaria pertanto una verifica dell'implementazione del supporto alle comunicazioni dei due ambienti che possa guidare eventuali modifiche al supporto alle comunicazioni di ASSIST.

# Capitolo 6

# Conclusioni e proseguimento del lavoro

In questa tesi è stata proposta e sperimentata una metodologia per il supporto di applicazioni grid-aware espresse come moduli paralleli ASSIST *puri* e incapsulati in componenti *CCM* (*CORBA Component Model*). La metodologia sviluppa idee e risultati trattati in alcuni progetti di ricerca e impiega gli strumenti del Workpackage WP8 del progetto Grid.it. L'obiettivo del gruppo è di far evolvere *ASSIST* in un ambiente component-based per applicazioni ad alta performance su griglia computazionale. Il primo passo in tale direzione è stato quello di integrare ASSIST con un modello a componenti (non ad alta performance) esistente, attraverso l'incapsulamento di moduli paralleli ASSIST in componenti. L'obiettivo della tesi è stato quello di sperimentare l'integrazione e quindi valutare l'impatto del CCM e comprendere quali siano i meccanismi necessari da includere nel modello a componenti proposto, per implementare applicazioni ad alta performance. Inoltre la metodologia oggetto della tesi ha previsto la sperimentazione di un Application Manager, in grado di adattare l'applicazione alla dinamicità della griglia per ottenere migliori performance ed allo stesso tempo un miglior utilizzo delle risorse. Dalla sperimentazione si sono ottenuti due importanti risultati: uno riguardante l'integrazione ASSIST-CCM e l'altro la politica adattiva dell'Application Manager.

Per quanto concerne l'integrazione è emerso che l'applicazione ASSIST testata

risulta più scalabile dell'equivalente versione CCM in ambienti omogenei (cluster di PC) e in ambienti eterogenei ma sulla stessa rete locale (due cluster di PC). C'è da dire però che l'overhead introdotto dal CCM nel calcolo e nelle comunicazioni risulta poco significativo nei casi in cui l'applicazione ha gradi di parallelismo non elevati. La sperimentazione sulla griglia utilizzata, in cui i nodi di elaborazione sono distribuiti in una rete geografica, invece ha portato a risultati ben diversi. L'applicazione integrata CCM risulta più scalabile della corrispondente versione ASSIST, nelle diverse configurazioni testate. Si è notato infatti che le comunicazioni implementate in ASSIST non risultano efficienti in un ambiente dinamico ed imprevedibile come quello delle griglie computazionali, in cui le risorse sono collegate da una rete geografica. Invece nella versione integrata CCM tali comunicazioni risultano più efficienti in tutte le configurazioni dell'applicazione. Prima di poter affermare che il modello sperimentale proposto sia effettivamente valido sarà necessario orientare la sperimentazione futura verso griglie ed applicazioni più complesse che consentano di stressare maggiormente il meccanismo di comunicazione. Dagli esperimenti effettuati è però emerso che, per poter avere prestazioni paragonabili fra applicazioni eseguite su griglia e su cluster, è necessario verificare e migliorare il supporto alle comunicazioni sia nel caso CCM (cluster) che nel caso ASSIST (griglia).

Per quanto concerne invece la politica adattiva implementata nell'Application Manager si è evinto che, in tutti i test effettuati sui diversi ambienti e con l'applicazione in versione ASSIST ed integrata CCM, si sono raggiunti i risultati sperati. Le prestazioni dell'applicazione che si è adattata alla dinamicità della griglia, rappresentata nel caso specifico dalla dinamicità dell'elaborazione, sono risultate paragonabili alle prestazioni dell'applicazione che ha utilizzato costantemente un numero maggiore di risorse. Inoltre la politica ha proposto un modello non basato sulla conoscenza a priori delle performance attese dall'applicazione e dal supporto di esecuzione, che è difficile avere in un ambiente di esecuzione altamente dinamico come quello delle griglie.

La sperimentazione effettuata ha però dei limiti. Di fatto la politica adattiva

implementata è legata al comportamento dell'applicazione e quindi progettata ad hoc. Inoltre la dinamicità della griglia non è stata sperimentata completamente in quanto non sono stati considerati eventuali sovraccarichi nei nodi di elaborazione e nella rete di interconnessione.

Sicuramente tale politica ha bisogno di ulteriori miglioramenti che possano renderla meno sensibile a situazioni momentanee della rete o dei nodi che compongono la griglia. Inoltre si potrebbe sperimentare una diversa politica adattiva che modifichi la configurazione dell'applicazione non utilizzando due implementazioni di un modulo, ma cambiando il parallelismo interno del modulo stesso (e.g. aggiungere/rimuovere worker di un farm). Infine si potrebbero condurre esperimenti sulla dinamicità in presenza di sovraccarichi nei nodi di elaborazione o nella rete che li interconnette.

Tutte le tematiche appena discusse saranno oggetto dello sviluppo futuro del lavoro del gruppo di ricerca che si occupa del Workpackage WP8 del progetto Grid.it.

# Bibliografia

[1] M. Aldinucci, S. Campa, P. Ciullo, M. Coppola, M. Danelutto, P. Pesciullesi, R. Ravazzolo, M. Torquati, M. Vanneschi, and C. Zoccolo. ASSIST demo: a high level, high performance, portable, structured parallel programming environment at work. In H. Kosch, L. Boszormenyi, and H. Hellwagner, editors, *Euro-PAR 2003 Parallel Computing*, number 2790 in LNCS, pages 712–721, Kalgenfurt, Austria, Agosto 2003. Springer Verlag.

[2] M. Aldinucci, S. Campa, P. Ciullo, M. Coppola, S. Magini, P. Pesciullesi, L. Potiti, R. Ravazzolo, M. Torquati, M. Vanneschi, and C. Zoccolo. The Implementation of ASSIST, an Environment for Parallel and Distributed Programming. In H. Kosch, L. Boszormenyi, and H. Hellwagner, editors, *Euro-PAR 2003 Parallel Computing*, number 2790 in LNCS, pages 712–721, Kalgenfurt, Austria, Agosto 2003. Springer Verlag.

[3] Marco Aldinucci, Sonia Campa, Massimo Coppola, Marco Danelutto, Domenico Laforenza, Diego Puppin, Luca Scarponi, Marco Vanneschi, and Corrado Zoccolo. Components for high-performance grid programming in Grid.it, 2004.

[4] Marco Aldinucci, Massimo Coppola, Marco Danelutto, Marco Vanneschi, and Corrado Zoccolo. ASSIST as a Research Framework for High-performance Grid Programming Environments. Technical report, Dipartimento di Informatica dell'Università di Pisa, Istituto di Scienza e Tecnologie dell'Informazione CNR Pisa, 2004.

[5] Rob Armstrong, Dennis Gannon, Al Geist, Katarzyna Keahey, Scott R. Kohn, Lois McInnes, Steve R. Parker, and Brent A. Smolinski. Toward a common component architecture for high-performance scientific computing. In HPDC, 1999.

[6] Gerd Beneken, Ulrike Hammerschall, Manfred Broy, María Victoria Cengarle, Jan Jürjens, Bernhard Rumpe, and Maurce Schoenmakers. Componentware - State of the Art 2003. Background Paper for the Understanding Components Workshop of the CUE Initiative at the Università Ca' Foscari di Venezia, Ottobre 2003.

[7] K. Bergner, A. Rausch, M. Sihling, and A. Vilbig. Putting the parts together: Concepts, description techniques and development process for componentware, 2000. In 33rd Hawaii International Conference on System Sciences.

[8] D. E. Bernholdt, W. R. Elwasif, J. S. Kohl, and T. G. W. Epperly. A component architecture for high performance computing, Giugno 2002. In Workshop on Performance Optimization for High-level Languages and Libraries.

[9] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

[10] M. Broy, F. Dererichs, C. Dendorfer, M. Fuchs, T. F. Gritzner, and R. Weber. The design of distributed systems - An introduction to FOCUS. Technical report, Technische Universität München, Institut für Informatik, Sonderforshungsbereich 342: Methoden und Werkzeuge für die Nutzung paralleler Architekturen TUM-I9202, 1992.

[11] Sonia Campa, Massimo Coppola, Silvia Magini, Laura Potiti, and Diego Puppin. ASSIST2CCM: Implementazione. Technical report, Dipartimento di Informatica, Università di Pisa, 2004.

[12] Sonia Campa, Massimo Coppola, Silvia Magini, Laura Potiti, and Diego Puppin. Piano iniziale per la sperimentazione di ASSIST + componenti CCM. Technical report, Dipartimento di Informatica, Università di Pisa, 2004.

[13] Sonia Campa, Tiziano Fagni, Gianfranco Mascari, Diego Puppin, Massimo Serranò, and Nicola Tonellotto. Analysis of Component-Oriented Frameworks, 2004.

[14] John Cheesman and John Daniels. *UML Components, A Simple Process for Specifying Component-Based Systems*. Addison-Wesley, 2001.

[15] P. Ciullo, M. Danelutto, D. Guerri, M. Lettere, L. Vaglini, and M. Vanneschi. Ambiente ASSIST: modello di programmazione e linguaggio di coordinamento ASSIST-CL (versione 1.0). Technical report, Dipartimento di Informatica dell'Università di Pisa, Synapsis srl, 2001.

[16] M. Cole. *Algorithmic skeletons: structured management of parallel computation*. MIT Press, Cambridge, Mass., 1989.

[17] M. Coppola and M. Vanneschi. High Performance Data Mining with Skeleton-based Structured Programming. *Parallel Computing*, 28:793–813, 2002. special issue on Parallel Data Intensive Algorithms, Elsevier Science.

[18] H. Dail, O. Sievert, F. Berman, H. Casanova, A. Yarkhan, S. Vadhiyar, J. Dongarra, C. Liu, L. Yang, D. Angulo, and I. Foster. Scheduling in the Grid Application Development Software Project, 2003. In Resource Management in the Grid.

[19] M. Danelutto. Efficient Support for Skeletons on Workstation Clusters. *Parallel Processing Letters*, 11(1):41–56, 2001.

[20] M. Danelutto. Adaptive Task Farm Implementation Strategies. In IEEE Comp. Soc., editor, *Euromicro Conference on Parallel, Distributed and Network-based Processing*, pages 416–423, La Coruna, Novembre 2004.

[21] Alexandre Denis, Christian Pérezy, Thierry Priolz, and André Ribesx. Padico: A Component-Based Software Infrastructure for Grid Computing. Technical Report 4974, Institut National de Recherche en Informatique et en Automatique, Ottobre 2003.

[22] D. F. D'Souza and A. C. Wills. *Objects, Components and Frameworks with UML: The Catalysis Approach*. Addison-Wesley, 1998.

[23] Colin Atkinson et al. *Component-based Product Line Engineering with UML*. Addison-Wesley, 2001.

[24] F. Burbera et al. Unraveling the Web Services: an introduction to SOAP, WSDL and UDDI. *IEEE Internet Computing*, 6(2):86–93, 2002.

[25] Ian Foster et al. Grid Services for Distributed System Integration. *Computer*, 35(6):37–46, 2002.

[26] Ian Foster and Carl Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*, chapter 11. Morgan Kaufmann, San Francisco, CA, 1999.

[27] Ian Foster, Carl Kesselman, Jeffrey M. Nick, and Steven Tuecke. Grid services for distribuited system integration. *Computer*, 35(6):37–46, Giugno 2002.

[28] Ian Foster, Carl Kesselman, Jeffrey M. Nick, and Steven Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed System Integration, 2002.

[29] Ian Foster, Carl Kesselman, and Steven Tuecke. The anatomy of the Grid: Enabling scalable virtual organization. *The International Journal of High Performance Computing Applications*, 15(3):200–222, 2001.

[30] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of reusable object-oriented software*. Addison-Wesley, 1986.

[31] M. Govindaraju, S. Krishnan, K. Chiu, A. Slominski, D. Gannon, and R. Bramley. XCat 2.0: A component-based programming model for grid web services. Technical Report TR562, Department of Computer Science, Indiana University, Giugno 2002.

[32] Object Management Group. CORBA Component Model version 3.0 Specification, Settembre 2002.

[33] OMG CCM Implementers Group. CORBA Component Model Tutorial, Aprile 2002.

[34] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[35] CCM Official Website. http://ditec.um.es/ dsevilla/ccm.

[36] OpenCCM Official Website. http://openccm.objectweb.org.

[37] The ASSIST Project Home Page. https://andromeda.di.unipi.it/assist.

[38] The Common Component Architecture Technical Specification - Version 0.5. http://www.cca-forum.org/bindings/old 0.5/.

[39] CCAffeine home page. http://www.cca forum.org/ccafe.

[40] The Globus Project. http://www.globus.org/gt3, 2003.

[41] Microsoft component object model (com). http://www.microsoft.com/com.

[42] Architecture description languages. http://www.sei.cmu.edu/str/descriptions/adl_body.html.

[43] David Levine, Douglas C. Schmidt, and Nanbor Wang. Optimizing the CORBA Component Model for High-performance and Real-time Applications, 2000.

[44] Silvia Magini. Integrazione ASSIST-CORBA. Technical report, Dipartimento di Informatica, Università di Pisa, 2004.

[45] D. Mcllroy. Mass-produced software components, 1968. In Software Engineering, NATO Science Committe report.

[46] SUN Microsystems. Enterprise JavaBeans(tm) specification 2.1 proposed final draft 2, Giugno 2003.

[47] The Object Management Group Home Page. Common Object Request Broker Architecture: Core Specification. http://www.omg.org.

[48] Laura Potiti. Applicazioni ASSIST. Technical report, Dipartimento di Informatica, Università di Pisa, Ottobre 2003.

[49] Laura Potiti. Stato di ASSIST. Technical report, Dipartimento di Informatica, Università di Pisa, Gennaio 2004.

[50] C. Pérez, T. Priol, and A. Ribes. A parallel CORBA component model for numerical code coupling. In C. A. Lee, editor, *3rd Intl. Workshop on Grid Computing*, number 2536 in LNCS, pages 88–99, Baltimore, Maryland, USA, Novembre 2002. Springer Verlag.

[51] Bran Selic, Garth Gullekson, and Paul T. Ward. *Real-Time Object-Oriented Modeling*. Wiley, 1994.

[52] Mary Shaw and David Garlan. *Software Architecture, Perspectives of an Emerging Discipline*. Prentice-Hall, Upper Saddle River, NJ, 1996.

[53] Johannes Siedersleben. Quasar: Die sd&m standardarchitektur. Technical report, sd&m AG, 2002.

[54] J. Siegel. *CORBA 3: Fundamentals and Programming*. OMG Press, 2nd edition, 2001.

[55] C. Szyperski. *Component Software, Beyond object-oriented programming*. Addison-Wesley, 1998.

[56] C. Szyperski and C. Pfister. Workshop on component-oriented programming, summary. In M. Mühlhäuser, editor, *Special Issues in Object-Oriented Programming - ECOOP 96, Workshop Reader*, Heidelberg, Novembre 1997. Verlag.

[57] Marco Vanneschi. The programming model of ASSIST, an environment for parallel and distributed portable applications. *Parallel Computing*, 28(12):1709–1732, Dicembre 2002.

# Appendice A

# Codice sorgente

**bench1.ast** Page 1/2

```
// -*- C++ -*-
/*
*  Luigi Presti, Marco Pasquali
*  Benchmark
*
*  Programma principale che include i moduli dell'applicazione:
*  Mandelbrot
*  PixConverter
*  MpegEncoder_lento
*  MpegEncoder_veloce
*  ApplicationManager
*  StreamStore
*
*/

//Dimensione gamut di colori per le immagini di Mandelbrot
#define COLORS 1024
//Numero di righe presenti nelle immagini di Mandelbrot
#define RAW 576
//Numero colonne presenti nelle immagini di Mandelbrot
#define COLUMN 720
//Numero di GOP che costituiscono il filmato finale compresso
#define GOP_MAX 401
//Numero di frame per GOP
#define GOP_ITEM 10

//File di LOG in cui vengono registrati i tempi
#define TIME_FILE_PIXCONVERTER "/home/presti/benchmark/Log_tmp_PixConverter.txt"
//File in cui viene memorizzato il filmato
#define OUTPUT_FILE "/home/presti/benchmark/mandel.mpg"
//Parallelismo mandelbrot
#define VP_NUM 16
//Parallellismo Mpeg veloce
#define NPV_VELOCE 4
//parallelismo Mpeg lento
#define NPV_LENTO 3
//Dimensione in byte di un frame del filmato
#define FRAME_SIZE 720*576*4
//Dimensione in byte del GOP in ingresso all'Mpeg
#define GOP_IN GOP_ITEM*FRAME_SIZE
//Dimensione massima GOP compresso da Mpeg
#define GOP_OUT 600000

//Define utilizzate per la raccolta dei tempi
#define INPUT_TIME_PIXCONVERTER 0
#define ASS_OUT_PIXCONVERTER 1
#define IDLE_TIME_PIXCONVERTER 2

//Struttura che rappresenta il GOP in entrata all'Mpeg
typedef struct {
  char data[GOP_ITEM*RAW*COLUMN*4];
  long gop_idx;
} gop_in_t;
//Struttura che rappresenta il GOP compresso in uscita dall'Mpeg
typedef struct {
  char data[GOP_OUT];
  long size;
  long gop_idx;
} gop_out_t;

#include "mandelbrot.ast"
#include "PixConverter.ast"
#include "ApplicationManager.ast"
#include "StreamStore.ast"
#include "mpeg/MpegEncoder_lento.ast"
#include "mpeg/MpegEncoder_veloce.ast"


generic main()
{
          //stream in uscita da mandelbrot
         stream short[RAW][COLUMN] Color;

         //stream per l'MpegEncoder_lento
```

**bench1.ast** Page 2/2

```
         stream gop_in_t GopIn_lento;
         stream gop_out_t GopOut_lento;

         //stream per l'MpegEncoder_veloce
         stream gop_in_t GopIn_veloce;
         stream gop_out_t GopOut_veloce;

         //stream per l'invio dei tempi rilevati da PixConverter
         stream double[3] time_pix;

         //stream attraverso il quale ApplicationManager avvisa PixConverter di
         //utilizzare un determinato MpegEncoder
         stream bool change;

         //stream per l'invio dei tempi rilevati da StreamStore
         stream double time_Stream;

         //moduli che costituiscono l'applicazione
         Mandelbrot(output_stream Color);
         PixConverter(input_stream Color, change output_stream GopIn_lento, GopI
n_veloce, time_pix);
         StreamStore(input_stream GopOut_lento, GopOut_veloce output_stream time
_Stream);
          MpegEncoder_veloce(input_stream GopIn_veloce output_stream GopOut_velo
ce);
         MpegEncoder_lento(input_stream GopIn_lento output_stream GopOut_lento);
         ApplicationManager(input_stream time_pix, time_Stream output_stream cha
nge);

}
```

**mandelbrot.ast** Page 1/5

```
// -*- C++ -*-
/*
*  Luigi Presti, Marco Pasquali
*  Benchmark
*
*  Mandelbrot: genera delle immagini dell'insieme di mandelbrot relative
*              a particolari aree presenti in un file di configurazione
*
*/
#define MAXITER 1024
#define MAX_POINTS 1024
#define POINTS 720
#define LATO 1.2
#define X_START .1
#define Y_START -.4
#define CONF_FILE "/home/presti/benchmark/mandeConf/mandelbrot.conf"

//Struttura contenente l'area da calcolare
typedef struct {
  double x;
  double y;
  double res;
  short points;
  short i;
  short area;
  short last;
} arg_t;

//Struttura contenente l'immagine calcolata
typedef struct {
  arg_t arg;
  short col[MAX_POINTS];
} res_t;

generic Mandelbrot(output_stream short Color[RAW][COLUMN] )
{
  stream arg_t A;
  stream res_t C;
  stream bool fatto;

  mgenera     (input_stream fatto output_stream A);
  mandelbrot (input_stream A output_stream C);
  mraccogli   (input_stream C output_stream fatto, Color);
}

pragma parallelism degree VP_NUM in mandelbrot;

parmod mandelbrot (input_stream arg_t A output_stream res_t C)
{
  topology none Pv;

  do input_section {
    guard1: on , , A {
      distribution A on_demand to Pv;
    }
  } while (true)

  virtual_processors {
    mandel (in guard1 out C) {
      VP {
        Fmandel (in A output_stream C);
      }
    }
  }

  output_section {
    collects C from ANY Pv;
  }
}

proc Fmandel (in arg_t task output_stream res_t sols)
$c++{
  res_t sol={task};
  int idx=0;
```

**mandelbrot.ast** Page 2/5

```
  //Calcolo dell'immagine
  for (int n=0; n<task.points; ++n) {
    dcomplex z=0, c=dcomplex(task.x+task.res*n,task.y);
    short i=0;

    while (i<MAXITER) {
      z=z*z-c;
      if(norm(z)>4) break;
      ++i;
    }

    if(idx==MAX_POINTS) {
      sol.arg.points=idx;
      assist_out(sols,sol);
      sol.arg.x=task.x+task.res*n;
      idx=0;
    }
    sol.col[idx++] = i;
  }

  if(idx>0) {
    sol.arg.points=idx;
    assist_out(sols,sol);
  }
}c++$

parmod mgenera (input_stream bool fatto output_stream arg_t A)
{
  topology one Pv;
  attribute bool on_init;
  attribute bool cont;
  init { on_init=true; cont=true; }

  do input_section {
    guard1: on , on_init,  {
      operation {
        on_init=false;
      }
    }
    guard2: on , , fatto  {
      distribution fatto broadcast to Pv;
      operation {
        cont=false;
      }
    }
  } while (cont)

  virtual_processors {
    mandel (in guard1 out A) {
      VP {
        F_genera (output_stream A);
      }
    }
    clock (in guard2) {
      VP {
        Blocca (in fatto);
      }
    }
  }

  output_section {
    collects A from ANY Pv;
  }
}


proc F_genera (output_stream arg_t A)
path<"/home/presti/benchmark">
inc<"iostream","unistd.h">
$c++{

  static unsigned points=POINTS;
  double resolution = LATO/points;
```

**mandelbrot.ast** Page 3/5

```
  FILE * conf_file;
  char temp = (char) 1;
  fpos_t * pos = (fpos_t*) malloc(sizeof(fpos_t)); /* indicatore di posizione de
l file */


  //Settaggi delle varie aree
  double x_start;
  double y_start;
  double side;


  arg_t arg;
  arg.x = X_START;
  arg.res = resolution;
  arg.points = 0;

  std::cerr << "\nPREPARING "<<std::endl;

  //INIZIALIZZA I WORKER
  for(int i=0;i<32;++i) {
    arg.y=0;
    assist_out(A,arg);
  }
  arg.points = POINTS;
  for(int i=1;i>=0;--i) {
    std::cerr << "WAITING "<<i<<std::endl;
    sleep(1);
  }

  /*Apre e legge il file di configurazione che contiene le aree da calcolare */
  if ((conf_file = fopen(CONF_FILE,"r")) == NULL) {
    perror("Nella fopen(conf_file)");
    exit(-1);
  }


  /*Le righe che iniziano con # rappresentano commenti, quindi va avanti nella l
ettura */
  while ((int)temp){
    /*Si prende l'indicatore di posizione nel file: inizio riga  */
    if (fgetpos(conf_file,pos)) {
      perror("Nella fgetpos() su conf_file");
      exit(-1);
    }
    /* Controlla primo carattere riga */
    if ((int)temp = (getc(conf_file) == '#') ) {

      /* Se inizia con # legge tutta la riga  */
      while ((temp = getc(conf_file)) != '\n'){
        /* Se e' arrivato alla fine del file e' un errore  */
        if (temp == EOF){
          printf("Formato file 'conf_file' non valido\n");
          exit(-1);
        }
      }
    }

    /* La riga non e' un commento */
    else {
      /* Ripristino l'indicatore di posizione all'inizio riga  */
      if (fsetpos(conf_file,pos)) {
        perror("Nella fgetpos() su conf_file");
        exit(-1);
      }
      temp = (char)0;
    }
  }



  /* Controlla le aree presenti nel file */
  {
    int tmp=-1;
```

**mandelbrot.ast** Page 4/5

```
    if (fscanf(conf_file,"%d", &tmp) != 1){
      printf("Formato file 'conf_file' non valido\n");
      exit(-1);
    }
    arg.last=tmp;
  }


  std::cerr << "\n* * * MANDELBROT: STARTING \n                  NUMERO IMMAGINI
   " << arg.last << "\n                   NUMERO ITERAZIONI "<< MAXITER <<"\n
             NUMERO PROCESSORI "<< VP_NUM << "\n\n" <<std::endl;

  for (int j=0; j<arg.last; j++) {
    //Legge le configurazioni delle varie aree
    if (fscanf(conf_file,"%lf %lf %lf",&(x_start),&(y_start),&(side)) != 3){
      printf("Formato file 'conf_file' non valido\n");
      exit(-1);
    }
    arg.x = x_start;
    resolution = side/points;
    arg.res = resolution;
    arg.area=j;

    for(int i=72;i<648;++i) {
      arg.y=y_start+i*resolution;
      arg.i = (POINTS - 1) - i;

      assist_out(A,arg);
    }
  }
}c++$

proc Blocca (in bool ok)
path<"/home/presti/benchmark">
inc<"iostream">
$c++{
  if(ok) std::cerr << "ENDING with SUCCESS mandelbrot.ast" << std::endl;
  else std::cerr << "ENDING with FAILURE mandelbrot.ast" << std::endl;
}c++$

mraccogli(input_stream res_t res output_stream bool ok, short Color[RAW][POINTS]
) {
  F_raccogli(in res output_stream ok,Color);
}

proc F_raccogli (in res_t res output_stream bool ok, short Color[RAW][POINTS])
path<"/usr/local/astCC1_1/include/userlib/">
inc<"clock.h", "iostream","stdio.h","stdlib.h","RTSProc.h","math.h","queue", "sy
s/types.h", "sys/stat.h", "unistd.h">
$c++{
  static unsigned points=POINTS;
  static long total_points=RAW*COLUMN;
  static long received_points=0;
  static short color[RAW][POINTS];
  static int numImg = 0;

  //Coda per la gestione dei risultati dei worker (ogni worker calcola una riga
  //dell'immagine)
  static std::queue<res_t> my_queue;


  if (numImg == res.arg.area){
    received_points+=res.arg.points;

    int h2 = res.arg.points;

    for (int h=0;h<res.arg.points;h++) {
      color[res.arg.i-72][h] = res.col[--h2];
    }

    int j=(res.arg.y+1)/res.arg.res;
    long i=total_points-received_points;

    if (received_points==total_points) {
```

**mandelbrot.ast** Page 5/5

```
      assist_out(Color,color);

      std::cerr << "* * * MANDELBROT: Inviata a PixConverter immagine N. " << ++
numImg << std::endl;

      received_points=0;

      //Controllo gli elementi in coda
      int tmp = my_queue.size();
      while (tmp--){
                res_t tmp_res = my_queue.front();
                my_queue.pop();

                if (tmp_res.arg.area  == numImg){

                        received_points+=tmp_res.arg.points;
                        int h3 = tmp_res.arg.points;

                        for (int h=0;h<res.arg.points;h++) {
                                color[tmp_res.arg.i-72][h] = tmp_res.col[--h3];
                        }
                }else{
                        my_queue.push(tmp_res);
                }
      }

      //Se e' stata calcolata tutta l'immagine, la manda al modulo successivo
      if (res.arg.last == numImg)       {

        std::cerr << "* * * MANDELBROT: Calcolate tutte le " << res.arg.last <<
" immagini." << std::endl;
        assist_out(ok,true);
      }
    }
  }else{
    //Mette in coda la riga calcolata
    my_queue.push(res);
  }
}c++$
```

**PixConverter.ast** Page 1/5

```
// -*- C++ -*-
/*
*  Luigi Presti, Marco Pasquali
*  Benchmark
*
*  PixConverter: converte l'immagine calcolate da Mandelbrot in un formato compa
tibile
*                con l'MpegEncoder, e genera un fade, tra un'immagine e l'altra,
 che
*                costituisce il GOP da inviare all'Mpeg
*
*/

//Define riguardanti i file utilizzati da PixConverter
#define PPM_FILE "/home/presti/benchmark/mandel"
#define PIX_CONF_FILE "/home/presti/benchmark/PixConverter.conf"

//Identifica il numero di immagini da creare per la transizione (fade)
#define TRANS_ELM 11

//Dimensione dell'immagine nell'array in cui e' memorizzato un GOP
#define PIX_FRAME_SIZE RAW*COLUMN*4

//Identificano i buffer contenenti l'immagine e i fade tra un'immagine ed un'alt
ra
#define BUFFER_IMAGE 0
#define BUFFER_FADE 1

//altre macro e define sono state spostate in PixConverte_util.h

proc init_attrib (out bool channel)
  $c++{
  channel = false;
}c++$

proc fchange_channel(in bool change out bool channel)
  $c++{
  if (change)
    std::cerr<< "                                   + + + PIXCONVERTER: Ricevut
o comando di utilizzare Encoder Lento "  << std::endl;
  else
    std::cerr<< "                                   + + + PIXCONVERTER: Ricevut
o comando di utilizzare Encoder Veloce "  << std::endl;

  channel = change;
}c++$


parmod PixConverter(input_stream short Color[RAW][COLUMN], bool change output_st
ream gop_in_t GopIn_lento, gop_in_t GopIn_veloce, double time_pix[3]){

  topology one Pv;

  attribute bool channel onto Pv;

  init {
    VP {
      init_attrib (out channel);
    }
  }


  do input_section {
  guard1: on 0, , Color {
    distribution Color broadcast to Pv;
  }

  guard2: on 1, , change {
    distribution change broadcast to Pv;
  }
  } while (true)


    virtual_processors{
```

**PixConverter.ast** Page 2/5

```
    converter(in guard1)
      {
        VP {
          fconverter(in Color,channel output_stream GopIn_lento, GopIn_veloce, t
ime_pix);
        }
      }

    change_channel(in guard2)
      {
        VP {
          fchange_channel(in change out channel);
        }
      }
  }

  output_section {
    collects GopIn_lento from ANY Pv;
    collects GopIn_veloce from ANY Pv;
    collects time_pix from ANY Pv;
  }

}

proc fconverter(in short Color[RAW][COLUMN], bool channel output_stream gop_in_t
 GopIn_lento, gop_in_t GopIn_veloce,  double time_pix[3])
path<"/usr/local/astCC1_1/include/userlib/">
inc<"iostream","stdio.h","stdlib.h","RTSProc.h","math.h","PixConverter_util.h",
"clock.h">
$c++{

  //Identifica il numero di immagini arrivate da Mandelbrot
  //(utilizzato per la scrittura sul file ppm)
  static int imgNum = 0;

  //Identifica il numero di GOP generati
  static int idx = 0;

  //identifica il tipo di opearazione da effettuare:
  //1: converte nel formato PIX ed invia i GOP generati all'MpegEncoder
  //0: converte le immagini inviate da Mandelbrot in PPM
  static int op;

  //Queste variabili servono per prendere i tempi
  static double time_idle = 0;
  static double time_input = 0;
  static double time_input1 = 0;
  static double time_ass_out = 0;
  static double time_ass_out1 = 0;
  static double time[3];


  //Legge dal file che tipo di operazione effettuare con le immagini che arrivan
o
  if (!imgNum) {
    FILE * conf_file = fopen(PIX_CONF_FILE,"r");
    /* controllo le aree presenti nel file */
    if (fscanf(conf_file,"%d",&(op)) != 1){
      printf("PixConverter: Formato file 'conf_file' non valido\n");
      exit(-1);
    }
    fclose(conf_file);
  }


  //Struttura dati necessaria per la conversione ed il fade
  //il primo buffer e' quello dell'immagine replicata, il secondo invece riguard
a
  //il fade tra un'immagine e l'altra
  static gop_in_t outGop[2];


  //Coefficiente di conversione nel formato PIX (numero di bit)
  int cdc =  (int) (((log((double)COLORS)/log(2.0)) - 8.0)/2.0) + 0.1;
```

**PixConverter.ast** Page 3/5

```
  //Caso operazione numero 0: converte in formato PPM
  if (!op) {

    char ppmFileName[256];
    char charImgNum[4];

    strcpy(ppmFileName,PPM_FILE);
    sprintf(charImgNum,"%d",imgNum);
    strcat(ppmFileName,charImgNum);
    strcat(ppmFileName,".ppm");

    FILE * ppmfile = fopen(ppmFileName,"w");

    //Header ppm file
    fprintf(ppmfile,"P3\n# mandel.ppm\n%d %d\n255\n",COLUMN,RAW);

    for (int i=0;i<RAW;i++) {
      for (int j=0;j<COLUMN;j++) {

        unsigned int r,g,b;
        int tmp=Color[i][j]-1;

        if (tmp<0) tmp=0; // se numero iterazioni negativo, mettiamo a 0

        //conversione
        extractRGB(tmp,r,g,b);

        fprintf(ppmfile,"%d\n%d\n%d\n",r,g,b);

      }
    }
    fclose(ppmfile);
    std::cerr<< "+ + + PIXCONVERTER: Creato file ppm " << ppmFileName << std::en
dl;
  }

  //Caso operazione 1: conversione in formato pix ed creazione/invio GOP
  else {
    int h = 0;

    //Caso della prima immagine
    if (!imgNum) {

      //Prende il tempo di input
      time_input = get_time();

      //Converte e memorizza la prima immagine nel buffer_image
      for (int i=0;i<RAW;i++) {
        for (int j=0;j<COLUMN;h+=4, j++) {

          unsigned int r,g,b;

          int tmp = Color[i][j]-1;

          if (tmp<0) tmp=0; // se numero iterazioni negativo, si mette a 0

          //Conversione
          extractRGB(tmp,r,g,b);


          //Salva la conversione del punto in tutto il gop/buffer relativo all'i
mmagine
          for (int y=0;y<GOP_ITEM;y++) {
            outGop[BUFFER_IMAGE].data[h   +(y*PIX_FRAME_SIZE)] = (char) r;
            outGop[BUFFER_IMAGE].data[h+1 +(y*PIX_FRAME_SIZE)] = (char) g;
            outGop[BUFFER_IMAGE].data[h+2 +(y*PIX_FRAME_SIZE)] = (char) b;
            outGop[BUFFER_IMAGE].data[h+3 +(y*PIX_FRAME_SIZE)] = PIX_ALPHA;
          }
        }
      }


      //Invia il gop relativo all'immagine
```

**PixConverter.ast** Page 4/5

```
      outGop[BUFFER_IMAGE].gop_idx = idx++;

      if (channel){
        assist_out(GopIn_lento, outGop[BUFFER_IMAGE]);
      }else{
        assist_out(GopIn_veloce, outGop[BUFFER_IMAGE]);
      }

      //Prende il tempo di idle
      time_idle = get_time();


      std::cerr<< "+ + + PIXCONVERTER: Creato ed inviato a ENCODER gop N. " << i
dx <<std::endl;

    }else{

      //Prende il tempo di input
      time_input1 = get_time();

      //Tempo di input
      time_input = time_input1 - time_input;

      //Tempo di idle
      time_idle = time_input1 - time_idle;

      //Deve raccogliere il tempo di idle
      time[IDLE_TIME_PIXCONVERTER] = time_idle;

      //Devo raccogliere il tempo di input
      time[INPUT_TIME_PIXCONVERTER] = time_input;

      time_input = time_input1;

      // Dalla seconda immagine in poi converte l'immagine ricevuta memorizzando
la
      //nel buffer_image ed al volo la interpola per ottenere anche il buffer_fa
de
      for (int i=0;i<RAW;i++) {
        for (int j=0;j<COLUMN;h+=4, j++) {

          //Contengono il punto di arrivo dell'interpolazione
          unsigned int r,g,b;

          //Punto di partenza dell'interpolazione
          unsigned int r1, g1, b1;

           int tmp = Color[i][j]-1;

           if (tmp<0) tmp=0; // se numero iterazioni negativo, mette a 0

           //Conversione
           extractRGB(tmp,r,g,b);

          //Pixel di origine dell'interpolazione
          r1 = (unsigned char) outGop[BUFFER_IMAGE].data[h  ];
          g1 = (unsigned char) outGop[BUFFER_IMAGE].data[h+1];
          b1 = (unsigned char) outGop[BUFFER_IMAGE].data[h+2];

          //Memorizza l'interpolazione nel buffer_fade e l'immagine ricevuta
          //e convertita nel buffer_image
          for (int y=0;y<GOP_ITEM;y++) {
            unsigned int r2,g2,b2;
            //Calcola il nuovo pixel che fara' parte del fade
            r2 = ((y+1) * r + (TRANS_ELM-(y+1)) * r1) / TRANS_ELM;
            g2 = ((y+1) * g + (TRANS_ELM-(y+1)) * g1) / TRANS_ELM;
            b2 = ((y+1) * b + (TRANS_ELM-(y+1)) * b1) / TRANS_ELM;

            //debug: non deve succedere
            if (r2<0 || g2 <0 || b2 <0 || r2 > 255 || g2 > 255 || b2 > 255)
              std::cerr << " ################ limiti superati!!! " << std::endl;
            if (r2>255) r2=255;
            if (g2>255) g2=255;
            if (b2>255) b2=255;
```

**PixConverter.ast** Page 5/5

```
            // fine debug

            //Memorizza il fade
            outGop[BUFFER_FADE].data[h   +(y*PIX_FRAME_SIZE)] = (char) r2;
            outGop[BUFFER_FADE].data[h+1 +(y*PIX_FRAME_SIZE)] = (char) g2;
            outGop[BUFFER_FADE].data[h+2 +(y*PIX_FRAME_SIZE)] = (char) b2;
            outGop[BUFFER_FADE].data[h+3 +(y*PIX_FRAME_SIZE)] = PIX_ALPHA;

            //Memorizza l'immagine
            outGop[BUFFER_IMAGE].data[h   +(y*PIX_FRAME_SIZE)] = (char) r;
            outGop[BUFFER_IMAGE].data[h+1 +(y*PIX_FRAME_SIZE)] = (char) g;
            outGop[BUFFER_IMAGE].data[h+2 +(y*PIX_FRAME_SIZE)] = (char) b;
            outGop[BUFFER_IMAGE].data[h+3 +(y*PIX_FRAME_SIZE)] = PIX_ALPHA;

          }
        }
      }

      //Spedisce prima il gop relativo al fade e poi quello relativo all'immagin
e
      outGop[BUFFER_FADE].gop_idx = idx++;
      outGop[BUFFER_IMAGE].gop_idx = idx++;

      //Prende i tempi dell'assist_out
      time_ass_out = get_time();


      //Invia i GOP generati all'MpegEncoder prescelto
      if (channel){
        assist_out(GopIn_lento, outGop[BUFFER_FADE]);
        assist_out(GopIn_lento, outGop[BUFFER_IMAGE]);
      }else{
        assist_out(GopIn_veloce, outGop[BUFFER_FADE]);
        assist_out(GopIn_veloce, outGop[BUFFER_IMAGE]);
      }

      time_ass_out1 = get_time();

      //Spedisce i tempi raccolti all'ApplictionManager
      time[ASS_OUT_PIXCONVERTER] = time_ass_out1 - time_ass_out;

      assist_out(time_pix, time);
      std::cerr<< "\n+ + + PIXCONVERTER: Inviato tempi ad Application Manager" <
< std::endl;
      //Prende il tempo di idle
      time_idle = get_time();
    }
  }
  imgNum++;
}c++$
```

**PixConverter_util.h** Page 1/1

```
/*
*  Luigi Presti, Marco Pasquali
*  Benchmark
*
*  PixConverter_util.h
*  Definisce le macro utili per il lavoro di conversione di PixConverter
*
*/


#define COLUMN 720
#define RAW    576
//Dimensione dell'immagine nell'array
#define PIX_FRAME_SIZE RAW*COLUMN*4


//Valore del canale alfa
#define PIX_ALPHA ((char) 1)
//Numero di colori per canale
#define PIX_COL_CHAN 256

//Prende 4 variabili in ingresso
//Usa la variabile cdc e altre define fatte da PixConverter
#define extractRGB(col, r, g, b) {                  \
          r = col % (PIX_COL_CHAN);                 \
          g = (col >> cdc) & (PIX_COL_CHAN-1);      \
          b = (col >> 2 * cdc) & (PIX_COL_CHAN-1); }


void savePPM (unsigned char *frame, char ppmFileName[256])
{
  unsigned char * tmp;

  FILE * ppmfile = fopen(ppmFileName,"w");

  //Header ppm file
  fprintf(ppmfile,"P3\n# mandel.ppm\n%d %d\n255\n",COLUMN,RAW);

  for (int i=0;i<PIX_FRAME_SIZE;i+=4) {
      tmp= frame+i;
      fprintf(ppmfile,"%d\n%d\n%d\n",*tmp,*(tmp+1),*(tmp+2));

  }

  fclose(ppmfile);
  std::cerr<< "+++PixConverter: Creato file ppm " << ppmFileName << std::endl;
}

void saveGOP (gop_in_t & myGop, char * fname )
{
  char ppmFileName[256];
  char charImgNum[4];
  std::cerr<< "+++PixConverter: saveGOP" << std::endl;

  for (int i=0; i<10;i++)
    {
      strncpy (ppmFileName, fname, 256);
      sprintf(charImgNum,"_%d",myGop.gop_idx*10+i);
      strcat(ppmFileName,charImgNum);
      strcat(ppmFileName,".ppm");

      savePPM (((unsigned char *) myGop.data)+i*PIX_FRAME_SIZE, ppmFileName);
    }
}
```

**MpegEncoder_veloce.ast** Page 1/3

```
// -*- C++ -*-
/*
*  Luigi Presti, Marco Pasquali
*  Benchmark 1
*
*  MpegEncoder_veloce: Raccoglie dei GOP non compressi e li comprime
*
*/

generic MpegEncoder_veloce(input_stream gop_in_t GopIn_veloce output_stream gop_
out_t GopOut_veloce)
{
  stream gop_in_t  A;

  genera  (input_stream GopIn_veloce output_stream A);
  encoder   (input_stream A output_stream GopOut_veloce);
}


genera(input_stream gop_in_t GopIn_veloce output_stream gop_in_t A) {
  Fgenera(in GopIn_veloce output_stream A);
}


proc Fgenera(in gop_in_t GopIn_veloce output_stream gop_in_t A)
inc<"iostream","stdio.h", "stdlib.h">
$c++{
  if (GopIn_veloce.gop_idx == 0){
    std::cerr << "\n- - - - STARTING ENCODER \n" << std::endl;
  }
  assist_out(A,GopIn_veloce);
  std::cerr << "- - - - ENCODER: Raccolto da PIXCONVERTER gop N. " << GopIn_velo
ce.gop_idx+1 << std::endl;
}c++$


proc encode(in gop_in_t A out gop_out_t GopOut_veloce)
inc<"WinCompatibility.h", "Picture.h","MPEG2Stream.h", "iostream">
$c++{
  static CMPEG2Stream * pEnc=0;

  if (pEnc==0) {
    // ***** Caratteristiche dello stream da generare *****
    CMPEG2Stream::m_StreamFormatInfo.m_StreamType               = IP_STREAM ;
    // Tipo dello stream prodotto I-Frame, IP-Frame, IBP-Frame
    CMPEG2Stream::m_StreamFormatInfo.m_bConstatBitRate  = false;        // Gener
a stream CBR = TRUE, VBR = FALSE
    CMPEG2Stream::m_StreamFormatInfo.m_StartTC.h                = 0;
    CMPEG2Stream::m_StreamFormatInfo.m_StartTC.m                = 0;
    CMPEG2Stream::m_StreamFormatInfo.m_StartTC.s                = 0;
    CMPEG2Stream::m_StreamFormatInfo.m_StartTC.fr               = 0;
    CMPEG2Stream::m_StreamFormatInfo.m_BitRate                  = 7*1024*1024;
    CMPEG2Stream::m_StreamFormatInfo.m_GOPSize                  = GOP_ITEM;
    CMPEG2Stream::m_StreamFormatInfo.m_I_P_Distance             = 6;//0;
// M (I/P frame distance)
    CMPEG2Stream::m_StreamFormatInfo.m_AspectRatio              = 2; // aspect_r
atio_information 1=square pel, 2=4:3, 3=16:9, 4=2.11:1
    CMPEG2Stream::m_StreamFormatInfo.m_FrameRateCode    = 3;          // frame_ra
te_code 1=23.976, 2=24, 3=25, 4=29.97, 5=30 frames/sec.
    CMPEG2Stream::m_StreamFormatInfo.m_Profile                  = 4; // Profile
ID: Simple = 5, Main = 4, SNR = 3, Spatial = 2, High = 1
    CMPEG2Stream::m_StreamFormatInfo.m_Level                    = 8; // Level ID
:   Low = 10, Main = 8, High 1440 = 6, High = 4
    CMPEG2Stream::m_StreamFormatInfo.m_bProgrSequence   = true;       // progress
ive_sequence
    CMPEG2Stream::m_StreamFormatInfo.m_VideoFormat              = 1; // video_fo
rmat: 0=comp., 1=PAL, 2=NTSC, 3=SECAM, 4=MAC, 5=unspec.
    CMPEG2Stream::m_StreamFormatInfo.m_ColorPrimaries   = 5;          // color_pr
imaries (5)
    CMPEG2Stream::m_StreamFormatInfo.m_TranferCharacteristics = 5;     // transfe
r_characteristics (5)
    CMPEG2Stream::m_StreamFormatInfo.m_MatrixCoefficients = 5;         // matrix_
coefficients (5)
    CMPEG2Stream::m_StreamFormatInfo.m_IntraDCPrecision = 3;          // intra_d
```

**MpegEncoder_veloce.ast** Page 2/3

```
c_precision (0: 8 bit, 1: 9 bit, 2: 10 bit, 3: 11 bit
    CMPEG2Stream::m_StreamFormatInfo.m_bIsFieldBased    = false;      // Vero se
 il filmato é fiels based
    CMPEG2Stream::m_StreamFormatInfo.m_bLow_delay               = false;
    CMPEG2Stream::m_StreamFormatInfo.m_bProg_frame              = true;
// Progressive frame
    CMPEG2Stream::m_StreamFormatInfo.m_bRepeatfirst             = false;
// Repeat first field after second field
    CMPEG2Stream::m_StreamFormatInfo.m_bFieldpic                = false;
// Use field pictures
    CMPEG2Stream::m_StreamFormatInfo.m_bTopfirst                = true;
// Display top field first
    CMPEG2Stream::m_StreamFormatInfo.m_vbv_buffer_size  = 112;          // Size
of VBV buffer (* 16 kbit)
    CMPEG2Stream::m_StreamFormatInfo.m_IntraVlcUse.m_bI = TRUE;         // intra
_vlc_format (I P B)
    CMPEG2Stream::m_StreamFormatInfo.m_IntraVlcUse.m_bB = TRUE;         // intra
_vlc_format (I P B)
    CMPEG2Stream::m_StreamFormatInfo.m_IntraVlcUse.m_bP = TRUE;         // intra
_vlc_format (I P B)
    CMPEG2Stream::m_StreamFormatInfo.m_FramePredDctUse.m_bI = TRUE;     // frame
_pred_frame_dct (I P B)
    CMPEG2Stream::m_StreamFormatInfo.m_FramePredDctUse.m_bB = TRUE;     // frame
_pred_frame_dct (I P B)
    CMPEG2Stream::m_StreamFormatInfo.m_FramePredDctUse.m_bP = TRUE;     // frame
_pred_frame_dct (I P B)
    // **** Rate Control
    CMPEG2Stream::m_StreamFormatInfo.m_r                = 0;    // Rate control:
 r (reaction parameter)
    CMPEG2Stream::m_StreamFormatInfo.m_Xi               = 0;    // Rate control:
 Xi (initial I frame global complexity measure)
    CMPEG2Stream::m_StreamFormatInfo.m_Xp               = 0;    // Rate control:
 Xp (initial P frame global complexity measure)
    CMPEG2Stream::m_StreamFormatInfo.m_Xb               = 0;    // Rate control:
 Xb (initial B frame global complexity measure)
    CMPEG2Stream::m_StreamFormatInfo.m_d0i              = 0;    // Rate control:
 d0i (initial I frame virtual buffer fullness)
    CMPEG2Stream::m_StreamFormatInfo.m_d0p              = 0;    // Rate control:
 d0p (initial P frame virtual buffer fullness)
    CMPEG2Stream::m_StreamFormatInfo.m_d0b              = 0;    // Rate control:
 d0b (initial B frame virtual buffer fullness)
    CMPEG2Stream::m_StreamFormatInfo.m_avg_act  = 0;    // Rate control: avg_act
 (initial average activity)
    // **** Caratteristiche del frame
    CMPEG2Stream::m_StreamFormatInfo.m_FrameInfo.m_ChromaComponents     = 4;
// 1 per 4:2:0 ; 2 per 4:2:2 ; 4 per 4:4:4
    CMPEG2Stream::m_StreamFormatInfo.m_FrameInfo.m_DimX
= 720;
    CMPEG2Stream::m_StreamFormatInfo.m_FrameInfo.m_DimY
= 576;
    CMPEG2Stream::m_StreamFormatInfo.m_FrameInfo.m_DisplayDimX          = 720;
        // display_horizontal_size
    CMPEG2Stream::m_StreamFormatInfo.m_FrameInfo.m_DisplayDimY          = 576;
        // display_vertical_size
    // **** Caratteristiche/Parametri del MotionEstimation
    CMPEG2Stream::m_StreamFormatInfo.m_forw_hor_f_code  = 2;            // |
    CMPEG2Stream::m_StreamFormatInfo.m_forw_vert_f_code = 2;            // | mot
ion vector ranges
    CMPEG2Stream::m_StreamFormatInfo.m_back_hor_f_code  = 2;            // |
    CMPEG2Stream::m_StreamFormatInfo.m_back_vert_f_code = 2;            // |
    CMPEG2Stream::m_StreamFormatInfo.m_MV_DimX                  = 15; // Dimensi
one dello spostamento del vettore movimento sull'asse X
    CMPEG2Stream::m_StreamFormatInfo.m_MV_DimY                  = 15; // Dimensi
one dello spostamento del vettore movimento sull'asse Y

    pEnc = new CMPEG2Stream();
  }
  GopOut_veloce.gop_idx=A.gop_idx;

  CPicture * pPict = NULL;
  char * pFrame = (char *)A.data;       // Puntatore al frame corrente

  // *** Comprime tutti i frame del GOP ***
  // Avvia l'encoder per una nuova compressione...
```

**MpegEncoder_veloce.ast** Page 3/3

```
  pEnc->BeginStreaming("");


  // Imposta il time code relativo del primo frame del gop nello stream che risu
ltato
  // i t.c. vanno da [0...GOPSize-1], [GOPSize...2GopSize-1], [2GOPSize....
  pEnc->m_StreamFrameCount = A.gop_idx * CMPEG2Stream::m_StreamFormatInfo.m_GOPS
ize;

  int i;
  for (i=CMPEG2Stream::m_StreamFormatInfo.m_GOPSize; i; i--, pFrame+=FRAME_SIZE)
 {
    try {
      pPict = pEnc->NewFrame();
      memcpy ( pPict->m_VetRGBFrame, pFrame, FRAME_SIZE );
    } catch(...) {
      std::cerr << "NewFrame ERROR!" << std::endl;
    }
    try {
      pEnc->EncodePicture(pPict);
    } catch (...) {
      std::cerr << "EncodePicture ERROR!" << std::endl;
    }
  }

  pEnc->m_OutputFileStream.alignbits(); // NOTA BENE

  GopOut_veloce.size=pEnc->m_OutputFileStream.GetBufferedInfoByteCount();

  assert(GopOut_veloce.size<GOP_OUT);

  pEnc->m_OutputFileStream.StreamCopy((char *)GopOut_veloce.data);      // Estra
zione del GOP compresso

  std::cerr <<"- - - - ENCODER: inviato il gop numero "<< GopOut_veloce.gop_idx+
1 <<" a SALVA"<< std::endl;

}c++$


pragma parallelism degree NPV_VELOCE in encoder;

parmod encoder (input_stream gop_in_t A output_stream gop_out_t GopOut_veloce) {
  topology none Pv;

  do input_section {
    guard1: on , , A {
      distribution A on_demand to Pv;
    }
  } while (true)

  virtual_processors {
    elab1 (in guard1 out GopOut_veloce) {
      VP {
        encode(in A out GopOut_veloce);
      }
    }
  }

  output_section {
    collects GopOut_veloce from ANY Pv;
    }
}
```

**StreamStore.ast** Page 1/4

```
//* -*- C++ -*-
/*
*  Luigi Presti, Marco Pasquali
*  Benchmark
*
*  StreamStore: prende in ingresso il filmato compresso dall'MpegEncoder
*               e lo salva su disco
*/

parmod StreamStore(input_stream  gop_out_t GopOut_lento, gop_out_t GopOut_veloce
 output_stream double time_Stream)
{

  topology one Pv;

  do input_section {
  guard1: on , , GopOut_lento {
    distribution GopOut_lento broadcast to Pv;
  }
  guard2: on , , GopOut_veloce {
    distribution GopOut_veloce broadcast to Pv;
  }
  } while (true)

    virtual_processors{
    SalvaL(in guard1)
      {
        VP {
          fsalva(in GopOut_lento output_stream time_Stream);
        }
      }

    SalvaV(in guard2)
      {
        VP {
          fsalva(in GopOut_veloce output_stream time_Stream);
        }
      }
  }

  output_section {
    collects time_Stream from ANY Pv;
  }

}

proc fsalva(in gop_out_t B output_stream double time_Stream)
  inc<"WinCompatibility.h","math.h", "Picture.h","MPEG2Stream.h", "iostream", "s
ys/types.h", "sys/stat.h", "unistd.h", "../clock.h">
$c++{
  static int count = 0;
  static int ok = 0;
  //Relativa al GOP
  static CMPEG2Stream * m_pStreamOut =0;
  static int gop_processed=0;

  //Servono per prendere i tempi relativi a tutto il progetto
  static double time;
  static double time_input = 0;
  static double time_input1 = 0;

  struct gop_list_t {
    gop_out_t            gop;
    struct gop_list_t * next;
    struct gop_list_t * prec;
  };

  static struct gop_list_t * pending = 0;



  if (B.gop_idx == 0){

    std::cerr << "\nSTARTING STREAM_STORE\n" << std::endl;
```

**StreamStore.ast** Page 2/4

```
    time_input = get_time();

  }else{

    time_input1 = get_time();

    //Tempo di input
    time_input = time_input1 - time_input;

    //Deve inviare il tempo di input
    time = time_input;

    time_input = time_input1;

    //Invio vero e proprio
    assist_out(time_Stream, time);
    std::cerr<< "\n/ / / /STREAM_STORE: Inviato tempo di INPUT ad Application Ma
nager" << std::endl;
  }


  std::cerr << "/ / / /STREAM_STORE: Raccolto da MpegEncoder gop N. " << B.gop_i
dx+1 <<std::endl;




  if (m_pStreamOut==0) {
    // ***** Caratteristiche dello stream da generare *****
    CMPEG2Stream::m_StreamFormatInfo.m_StreamType              = IP_STREAM ;
    // Tipo dello stream prodotto I-Frame, IP-Frame, IBP-Frame
    CMPEG2Stream::m_StreamFormatInfo.m_bConstatBitRate  = false;        // Gener
a stream CBR = TRUE, VBR = FALSE
    CMPEG2Stream::m_StreamFormatInfo.m_StartTC.h               = 0;
    CMPEG2Stream::m_StreamFormatInfo.m_StartTC.m               = 0;
    CMPEG2Stream::m_StreamFormatInfo.m_StartTC.s               = 0;
    CMPEG2Stream::m_StreamFormatInfo.m_StartTC.fr              = 0;
    CMPEG2Stream::m_StreamFormatInfo.m_BitRate                 = 7*1024*1024;
    CMPEG2Stream::m_StreamFormatInfo.m_GOPSize                 = GOP_ITEM;
    CMPEG2Stream::m_StreamFormatInfo.m_I_P_Distance            = 6;//0;
// M (I/P frame distance)
    CMPEG2Stream::m_StreamFormatInfo.m_AspectRatio             = 2; // aspect_r
atio_information 1=square pel, 2=4:3, 3=16:9, 4=2.11:1
    CMPEG2Stream::m_StreamFormatInfo.m_FrameRateCode    = 3;         // frame_ra
te_code 1=23.976, 2=24, 3=25, 4=29.97, 5=30 frames/sec.
    CMPEG2Stream::m_StreamFormatInfo.m_Profile                 = 4; // Profile
ID: Simple = 5, Main = 4, SNR = 3, Spatial = 2, High = 1
    CMPEG2Stream::m_StreamFormatInfo.m_Level                   = 8; // Level ID
:   Low = 10, Main = 8, High 1440 = 6, High = 4
    CMPEG2Stream::m_StreamFormatInfo.m_bProgrSequence   = true;      // progress
ive_sequence
    CMPEG2Stream::m_StreamFormatInfo.m_VideoFormat             = 1; // video_fo
rmat: 0=comp., 1=PAL, 2=NTSC, 3=SECAM, 4=MAC, 5=unspec.
    CMPEG2Stream::m_StreamFormatInfo.m_ColorPrimaries   = 5;         // color_pr
imaries (5)
    CMPEG2Stream::m_StreamFormatInfo.m_TranferCharacteristics = 5;    // transfe
r_characteristics (5)
    CMPEG2Stream::m_StreamFormatInfo.m_MatrixCoefficients = 5;        // matrix_
coefficients (5)
    CMPEG2Stream::m_StreamFormatInfo.m_IntraDCPrecision = 3;          // intra_d
c_precision (0: 8 bit, 1: 9 bit, 2: 10 bit, 3: 11 bit
    CMPEG2Stream::m_StreamFormatInfo.m_bIsFieldBased    = false;      // Vero se
 il filmato é fiels based
    CMPEG2Stream::m_StreamFormatInfo.m_bLow_delay              = false;
    CMPEG2Stream::m_StreamFormatInfo.m_bProg_frame             = true;
// Progressive frame
    CMPEG2Stream::m_StreamFormatInfo.m_bRepeatfirst            = false;
// Repeat first field after second field
    CMPEG2Stream::m_StreamFormatInfo.m_bFieldpic               = false;
// Use field pictures
    CMPEG2Stream::m_StreamFormatInfo.m_bTopfirst               = true;
// Display top field first
    CMPEG2Stream::m_StreamFormatInfo.m_vbv_buffer_size  = 112;          // Size
of VBV buffer (* 16 kbit)
```

**StreamStore.ast** Page 3/4

```
    CMPEG2Stream::m_StreamFormatInfo.m_IntraVlcUse.m_bI = TRUE;         // intra
_vlc_format (I P B)
    CMPEG2Stream::m_StreamFormatInfo.m_IntraVlcUse.m_bB = TRUE;         // intra
_vlc_format (I P B)
    CMPEG2Stream::m_StreamFormatInfo.m_IntraVlcUse.m_bP = TRUE;         // intra
_vlc_format (I P B)
    CMPEG2Stream::m_StreamFormatInfo.m_FramePredDctUse.m_bI = TRUE;     // frame
_pred_frame_dct (I P B)
    CMPEG2Stream::m_StreamFormatInfo.m_FramePredDctUse.m_bB = TRUE;     // frame
_pred_frame_dct (I P B)
    CMPEG2Stream::m_StreamFormatInfo.m_FramePredDctUse.m_bP = TRUE;     // frame
_pred_frame_dct (I P B)
    // **** Rate Control
    CMPEG2Stream::m_StreamFormatInfo.m_r               = 0;    // Rate control:
 r (reaction parameter)
    CMPEG2Stream::m_StreamFormatInfo.m_Xi              = 0;    // Rate control:
 Xi (initial I frame global complexity measure)
    CMPEG2Stream::m_StreamFormatInfo.m_Xp              = 0;    // Rate control:
 Xp (initial P frame global complexity measure)
    CMPEG2Stream::m_StreamFormatInfo.m_Xb              = 0;    // Rate control:
 Xb (initial B frame global complexity measure)
    CMPEG2Stream::m_StreamFormatInfo.m_d0i             = 0;    // Rate control:
 d0i (initial I frame virtual buffer fullness)
    CMPEG2Stream::m_StreamFormatInfo.m_d0p             = 0;    // Rate control:
 d0p (initial P frame virtual buffer fullness)
    CMPEG2Stream::m_StreamFormatInfo.m_d0b             = 0;    // Rate control:
 d0b (initial B frame virtual buffer fullness)
    CMPEG2Stream::m_StreamFormatInfo.m_avg_act  = 0;    // Rate control: avg_act
 (initial average activity)
    // **** Caratteristiche del frame
    CMPEG2Stream::m_StreamFormatInfo.m_FrameInfo.m_ChromaComponents     = 4;
// 1 per 4:2:0 ; 2 per 4:2:2 ; 4 per 4:4:4
    CMPEG2Stream::m_StreamFormatInfo.m_FrameInfo.m_DimX
= 720;
    CMPEG2Stream::m_StreamFormatInfo.m_FrameInfo.m_DimY
= 576;
    CMPEG2Stream::m_StreamFormatInfo.m_FrameInfo.m_DisplayDimX          = 720;
        // display_horizontal_size
    CMPEG2Stream::m_StreamFormatInfo.m_FrameInfo.m_DisplayDimY          = 576;
        // display_vertical_size
    // **** Caratteristiche/Parametri del MotionEstimation
    CMPEG2Stream::m_StreamFormatInfo.m_forw_hor_f_code  = 2;            // |
    CMPEG2Stream::m_StreamFormatInfo.m_forw_vert_f_code = 2;            // | mot
ion vector ranges
    CMPEG2Stream::m_StreamFormatInfo.m_back_hor_f_code  = 2;            // |
    CMPEG2Stream::m_StreamFormatInfo.m_back_vert_f_code = 2;            // |
    CMPEG2Stream::m_StreamFormatInfo.m_MV_DimX                  = 15; // Dimensi
one dello spostamento del vettore movimento sull'asse X
    CMPEG2Stream::m_StreamFormatInfo.m_MV_DimY                  = 15; // Dimensi
one dello spostamento del vettore movimento sull'asse Y

    m_pStreamOut = new CMPEG2Stream(false);
    m_pStreamOut->BeginStreaming(OUTPUT_FILE);
  };

  if (B.gop_idx != gop_processed) {

    struct gop_list_t * tmp = new struct gop_list_t;
    tmp->gop.size    = B.size;
    tmp->gop.gop_idx = B.gop_idx;
    tmp->next        = 0;
    tmp->prec        = 0;
    memcpy(tmp->gop.data,B.data,GOP_OUT);

    if (pending == 0) {
        pending = tmp;
    }
    else {
      struct gop_list_t * cursor = pending;
      while (cursor != 0) {
        if (B.gop_idx > cursor->gop.gop_idx && cursor->next !=0 ) {
          cursor = cursor->next;
          continue;
        };
```

**StreamStore.ast** Page 4/4

```
        if (B.gop_idx < cursor->gop.gop_idx) {
          if (cursor->prec !=0 ) { cursor->prec->next = tmp; };
          if (pending == cursor) pending = tmp;
          tmp->prec    = cursor->prec;
          tmp->next    = cursor;
          cursor->prec = tmp;
          cursor       = 0;
        } else {
          tmp->next    = 0;
          tmp->prec    = cursor;
          cursor->next = tmp;
          cursor       = 0;
        };
      }
    }
  } else {
    m_pStreamOut->m_OutputFileStream.alignbits();
    m_pStreamOut->m_OutputFileStream.PutBuffer ((unsigned char *) B.data,B.size)
;
    gop_processed++;

    while (pending!=0 && pending->gop.gop_idx == gop_processed) {
      m_pStreamOut->m_OutputFileStream.alignbits();
      m_pStreamOut->m_OutputFileStream.PutBuffer ((unsigned char *)pending->gop.
data,pending->gop.size);
      struct gop_list_t * cursor = pending;
      pending = pending->next;
      delete cursor;
      gop_processed++;
    };
  };

  std::cerr << "- - - - MpegEncoder: Processato e salvato gop N. " << B.gop_idx+
1 << " dimensione " << B.size << std::endl;

}c++$
```

**ApplicationManager.ast** Page 1/3

```
// -*- C++ -*-
/*
*  Luigi Presti, Marco Pasquali
*  Benchmark
*
*  ApplicationManager: raccoglie nondeterministicamente le prestazioni dei vari
moduli e
*  decide la politica da adottare per adattare l'applicazione
*
*/

//File di configurazione di ApplicationManager, da qui si puo' cambiare
//la soglia per la politica di scheduling fra i due MpegEncoder
#define MANAGER_FILE "/l/disc1/home/presti/benchmark/Manager.conf"

//Valori utilizzati affinche' l'ApplicationManager sappia in quale fase si trova
#define UNDER_PHASE 0
#define OVER_PHASE 1

parmod ApplicationManager(input_stream double time_pix[3], double time_Stream ou
tput_stream bool change)
{
  topology one Pv;

  do input_section {
  guard1: on , , time_pix {
    distribution time_pix broadcast to Pv;
  }

  guard2: on , , time_Stream {
    distribution time_Stream broadcast to Pv;
  }
  } while (true)


  virtual_processors{
    managerPix(in guard1)
      {
        VP {
          fmanagerPix(in time_pix output_stream change);
        }
      }

    managerSalva(in guard2)
      {
        VP {
          fmanagerSalva(in time_Stream);
        }
      }
  }

  output_section {
    collects change from ANY Pv;
  }
}
//Gestisce i tempi calcolati da PixConverter
proc fmanagerPix(in double time_pix[3]  output_stream bool change)
inc<"iostream","stdio.h">
$c++{

  //Variabile necessaria al conteggio dei tempi ricevuti da PixConverter
  static int imgNum = 0;
  //Indica la soglia per la politica di scheduling
  static double threshold;
  //Indica l'intervallo di risposta da parte dell'ApplicationManager
  static int answer_interval;
  //Indica il numero di volte consecutive in cui il tempo di idle e' andato
  //sotto la soglia o sopra (questa fase viene indicata da current_phase)
  static int o_u_times = 0;
  static int current_phase = UNDER_PHASE;

  //Si parte dall'encoder lento (true), altrimenti per l'encoder veloce
  //sarebbe stato false.
  static bool used_mpeg = true;
```

**ApplicationManager.ast** Page 2/3

```
  //Accumulatore
  static double sum_pix = 0;


  //Scrive nel file le prestazioni misuarate da PixConverter
  FILE * time_file = fopen(TIME_FILE_PIXCONVERTER,"a");


  if (!imgNum) {
    fprintf(time_file,"INPUT_PIX\tBANDA_MEDIA_MA\tASSIST_OUT_PIX\tIDLE_PIX\tINPU
T_STREAM_STORE\tBANDA_MEDIA_MP\n");

    //Legge dal file la soglia di determinazione della politica di scheduling
    //e l'intervallo di risposta
    FILE * conf_file = fopen(MANAGER_FILE,"r");
    if (fscanf(conf_file,"%lf",&(threshold)) != 1){
      printf("Application Manager: Formato file 'Manager.conf' non valido\n");
      exit(-1);
    }
    if (fscanf(conf_file,"%d",&(answer_interval)) != 1){
      printf("Application Manager: Formato file 'Manager.conf' non valido\n");
      exit(-1);
    }
    printf("........................APP_MANAGER soglia %lf, intervallo di rispos
ta %d",threshold,answer_interval);
    fclose(conf_file);
  }


  //Raccoglie i tempi e li memorizza nel file
  sum_pix += time_pix[INPUT_TIME_PIXCONVERTER];
  fprintf(time_file,"%lf\t%lf\t%lf\t%lf\n",time_pix[INPUT_TIME_PIXCONVERTER],1.0
/(sum_pix/(++imgNum)),time_pix[ASS_OUT_PIXCONVERTER],time_pix[IDLE_TIME_PIXCONVE
RTER]);

  //Scarta i primi 10 tempi perche' influenzati dalla stabilizzazione iniziale
  if (imgNum > 10){

    if (time_pix[IDLE_TIME_PIXCONVERTER] > threshold){

      //Il collo di bottiglia e' Mandelbrot passa ad un MpegEncoder lento se non
 c'e'
      //gia' per risparmiare le risorse e se il numero di volte in cui ha
      //superato la soglia e' uguale dell'intervallo di risposta
      if (current_phase != OVER_PHASE) {
        o_u_times = 1;
        current_phase = OVER_PHASE;
      }
      else {
        o_u_times++;
      }

      if ((o_u_times == answer_interval) && (!used_mpeg)) {
          used_mpeg = true;
          o_u_times = 0;
          //Segnala nel file che ha ordinato a PixConverter di passare
          //all'MpegEncoder lento
          fprintf(time_file,"\t\t\t\t\t\t\t1\n");
          assist_out(change, used_mpeg);
      }
    }else if (time_pix[IDLE_TIME_PIXCONVERTER] < threshold){

      //Il collo di bottiglia e' l'MpegEncoder, passo ad uno piu' veloce, se non
 ci
      //sono gia', per aumentare la banda, e se il numero si volte in cui l'idle
      //e' andato sotto la soglia e' uguale dell'intervallo di risposta
      if (current_phase != UNDER_PHASE) {
        o_u_times = 1;
        current_phase = UNDER_PHASE;
      }
      else {
        o_u_times++;
      }
```

**ApplicationManager.ast** Page 3/3

```
      if ((o_u_times == answer_interval) && (used_mpeg)) {
        used_mpeg = false;
        o_u_times = 0;
        //Segnalo nel file che ha ordinato a PixConverter di passare
        //all'MpegEncoder veloce
        fprintf(time_file,"\t\t\t\t\t\t\t0\n");
        assist_out(change, used_mpeg);
      }
    }
  }

  //Chiude il file relativo a PixConverter
  fclose(time_file);

}c++$

//Gestisce i tempi rilevati da StreamStore
proc fmanagerSalva(in double time_Stream)
  inc<"iostream","stdio.h">
$c++{
    //Scrive nel file le prestazioni misuarate da StreamStore
    FILE * time_file = fopen(TIME_FILE_PIXCONVERTER,"a");

    //Accumulatori
    static double sum_stream_store = 0;
    static int div_stream_store = 1;


    sum_stream_store += time_Stream;
    fprintf(time_file,"\t\t\t\t\t\t\t\t%lf\t\t%lf\n",time_Stream,0.5/(sum_stream
_store/(div_stream_store)));
    div_stream_store++;


    //Chiude il file relativo a StreamStore
    fclose(time_file);
}c++$
```

**Table of Content** Page 1/1

**Table of Contents**


```
 1 bench1.ast.......... sheets   1 to   2 ( 2) pages   1-  2  101 lines
 2 mandelbrot.ast...... sheets   3 to   7 ( 5) pages   3-  7  329 lines
 3 PixConverter.ast.... sheets   8 to  12 ( 5) pages   8- 12  329 lines
 4 PixConverter_util.h. sheets  13 to  13 ( 1) pages  13- 13   65 lines
 5 MpegEncoder_veloce.ast sheets  14 to  16 ( 3) pages  14- 16  167 lines
 6 StreamStore.ast..... sheets  17 to  20 ( 4) pages  17- 20  213 lines
 7 ApplicationManager.ast sheets  21 to  23 ( 3) pages  21- 23  178 lines
```

# Appendice B

# Grafici delle sperimentazioni

Di seguito sono riportati tutti i grafici prodotti dall'esecuzione della terza versione dell'applicazione nelle diverse configurazioni e nei tre diversi ambienti di test. I grafici generati da ogni esecuzione dell'applicazione sono di due tipi:

1. il primo, identificato dalla lettera **A**, mostra i tempi rilevati dal modulo PixConverter e registrati da ApplicationManager che riguardano: il tempo di interarrivo dei dati da Mandelbrot (INPUT_PIX), il tempo necessario per inviare i due GOP non compressi ad MpegEncoder (ASSIST_OUT_PIX), il tempo di inattività trascorso nell'attesa dell'immagine da Mandelbrot (IDLE_PIX). La somma degli ultimi due tempi è pressoché uguale al primo tempo a meno del tempo di conversione di PixConverter che è piccolo e costante;

2. il secondo, identificato dalla lettera **B**, mostra i tempi rilevati dal modulo StreamStore e registrati da ApplicationManager che riguardano i tempi di interarrivo dai moduli MpegEncoder dei GOP compressi (INPUT_STREAM_STORE).

Per quanto riguarda i grafici di tipo **A**, nei casi in cui non si applica la politica adattiva, si possono presentare tre diversi casi:

1. collo di bottiglia nel modulo Mandelbrot: il tempo IDLE_PIX è costantemente nell'ordine dei secondi;

2. collo di bottiglia nel modulo MpegEncoder: il tempo IDLE_PIX è costantemente quasi nullo;

3. dinamicità dell'applicazione: il tempo IDLE_PIX varia da una situazione in cui è nell'ordine dei secondi ad una in cui è quasi nullo e viceversa.

Per quanto riguarda i grafici di tipo **B**, nei casi in cui non si applica la politica adattiva, si possono presentare due diversi casi:

1. INPUT_STREAM_STORE costante: si verifica principalmente quando l'MpegEncoder utilizza un solo worker e quindi il tempo di interarrivo dei GOP compressi da tale modulo risulta costante;

2. INPUT_STREAM_STORE oscillante: si verifica principalmente quando l'MpegEncoder utilizza più worker in parallelo e quindi il tempo di interarrivo risulta alternativamente molto alto e subito dopo molto basso. Di fatto i GOP vengono spediti dal modulo PixConverter due alla volta e quindi ci si può aspettare che MpegEncoder elabori due elementi dello stream in tempi ravvicinati mentre impieghi più tempo per elaborare il terzo in quanto deve attenderlo. Un'altra causa dell'oscillazione di tale tempo potrebbe riguardare la dinamicità dell'applicazione.

I grafici in cui invece si applica la politica adattiva sono descritti nel capitolo 5.

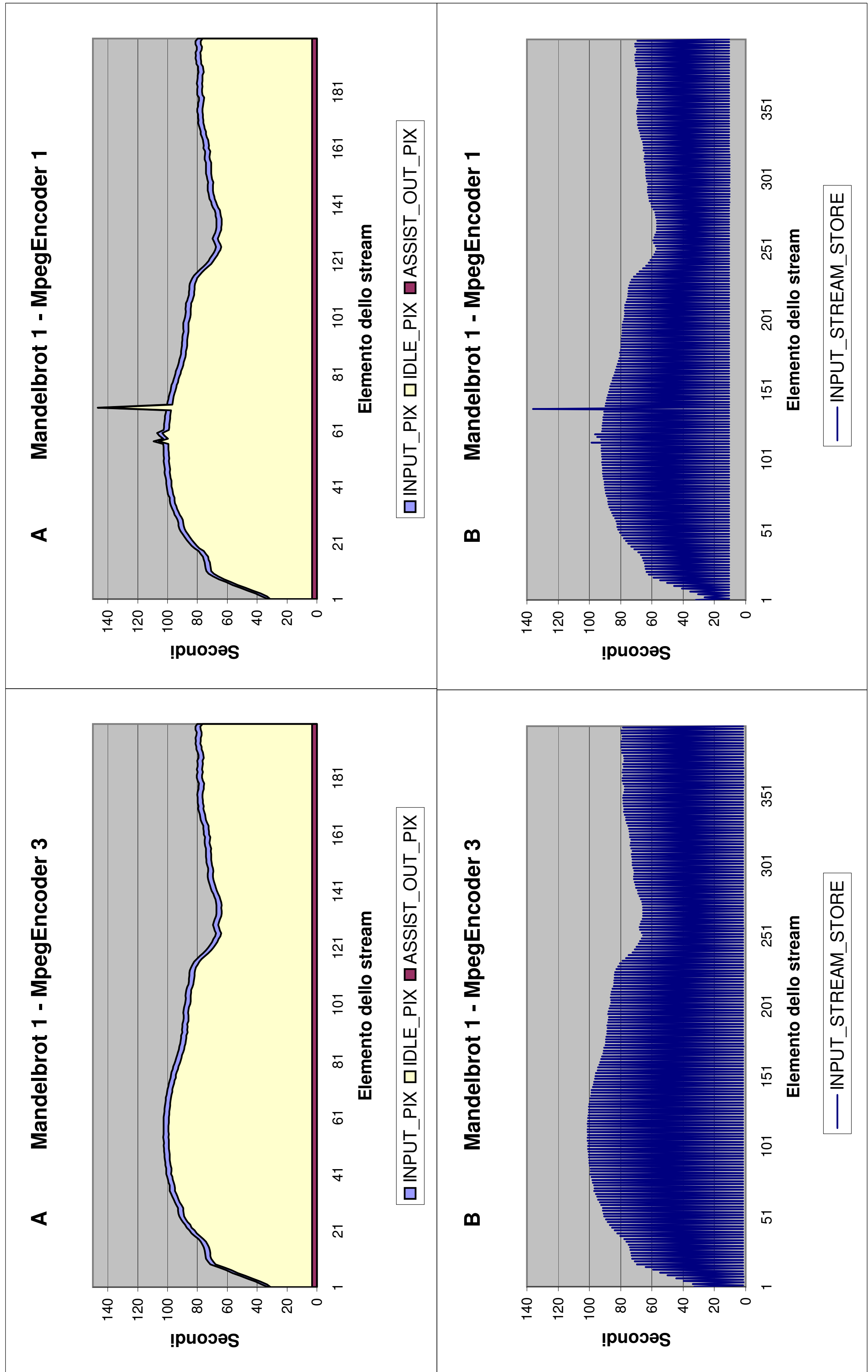
Il titolo di ogni grafico spiega la configurazione dell'applicazione che è stata eseguita (e.g. *Mandelbrot 6 - MpegEncoder 2*), in quali macchine viene eseguito MpegEncoder (*...su C1* o *...CNR*), se l'applicazione è stata integrata nel CCM (*...CCM*) e se l'ApplicationManager ha applicato la politica adattiva (*...APPLICAZIONE ADATTIVA*). Nei grafici sono riportati i secondi sull'asse delle ordinate mentre su quello delle ascisse gli elementi dello stream in oggetto.

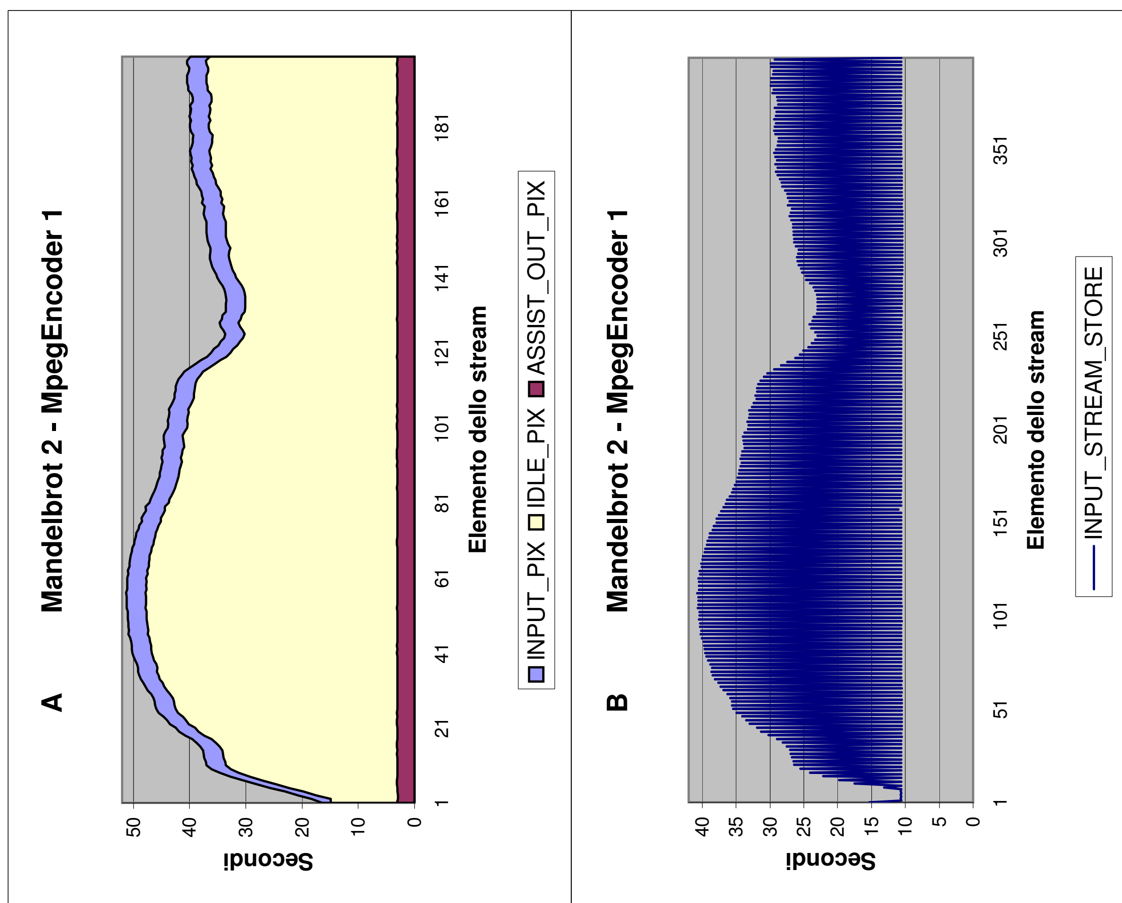
All'inizio, per ogni ambiente di esecuzione, viene mostrata una tabella che mette in evidenza dove si trovano i grafici di una particolare situazione dell'applicazione. Alla fine invece, per ogni ambiente di esecuzione, viene presentata una tabella ed

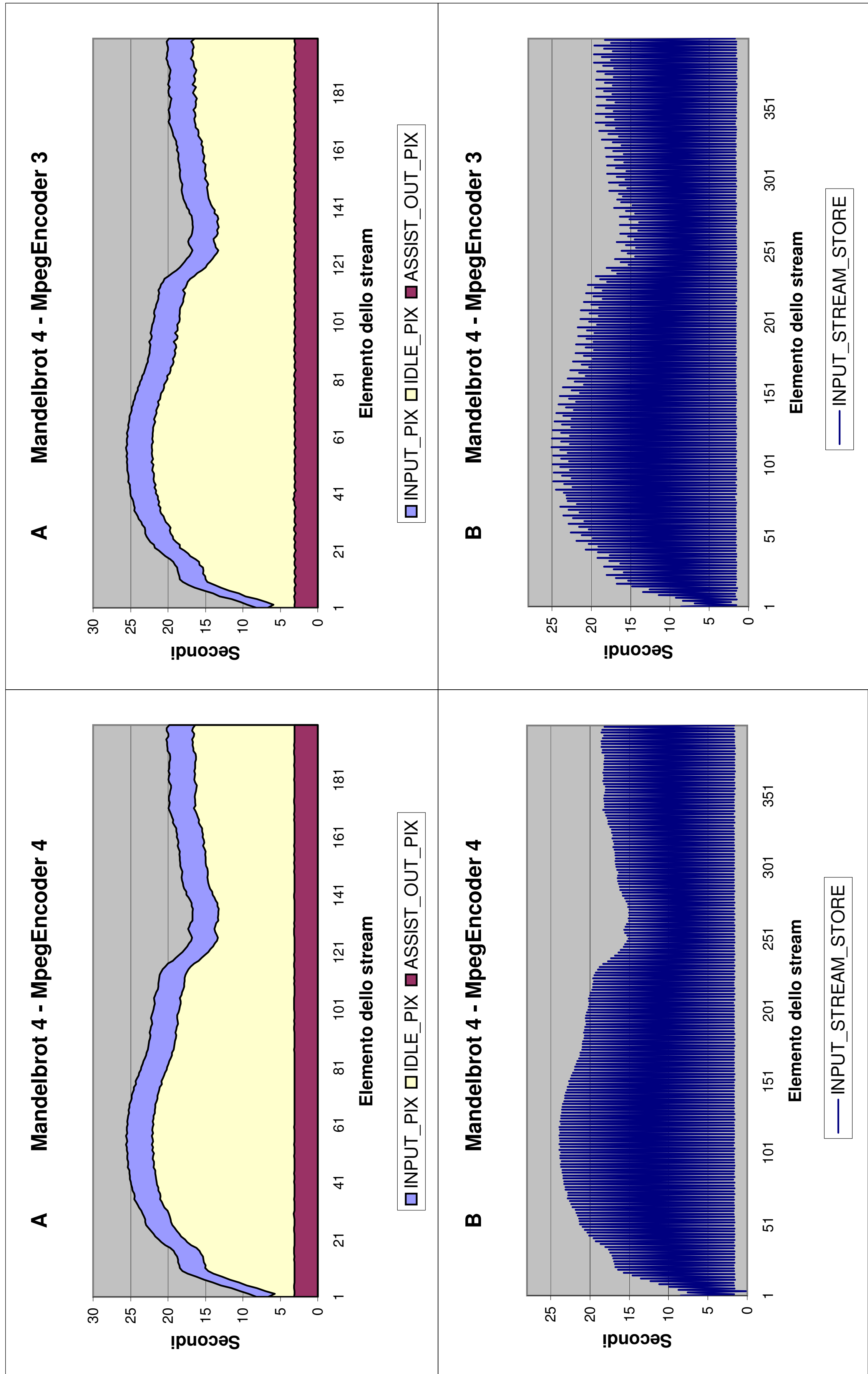
un grafico riassuntivi delle bande di esecuzione ottenute dall'applicazione nelle varie configurazioni testate. Vengono inoltre messi in evidenza i casi in cui si presenta la dinamicità nell'elaborazione dell'applicazione.

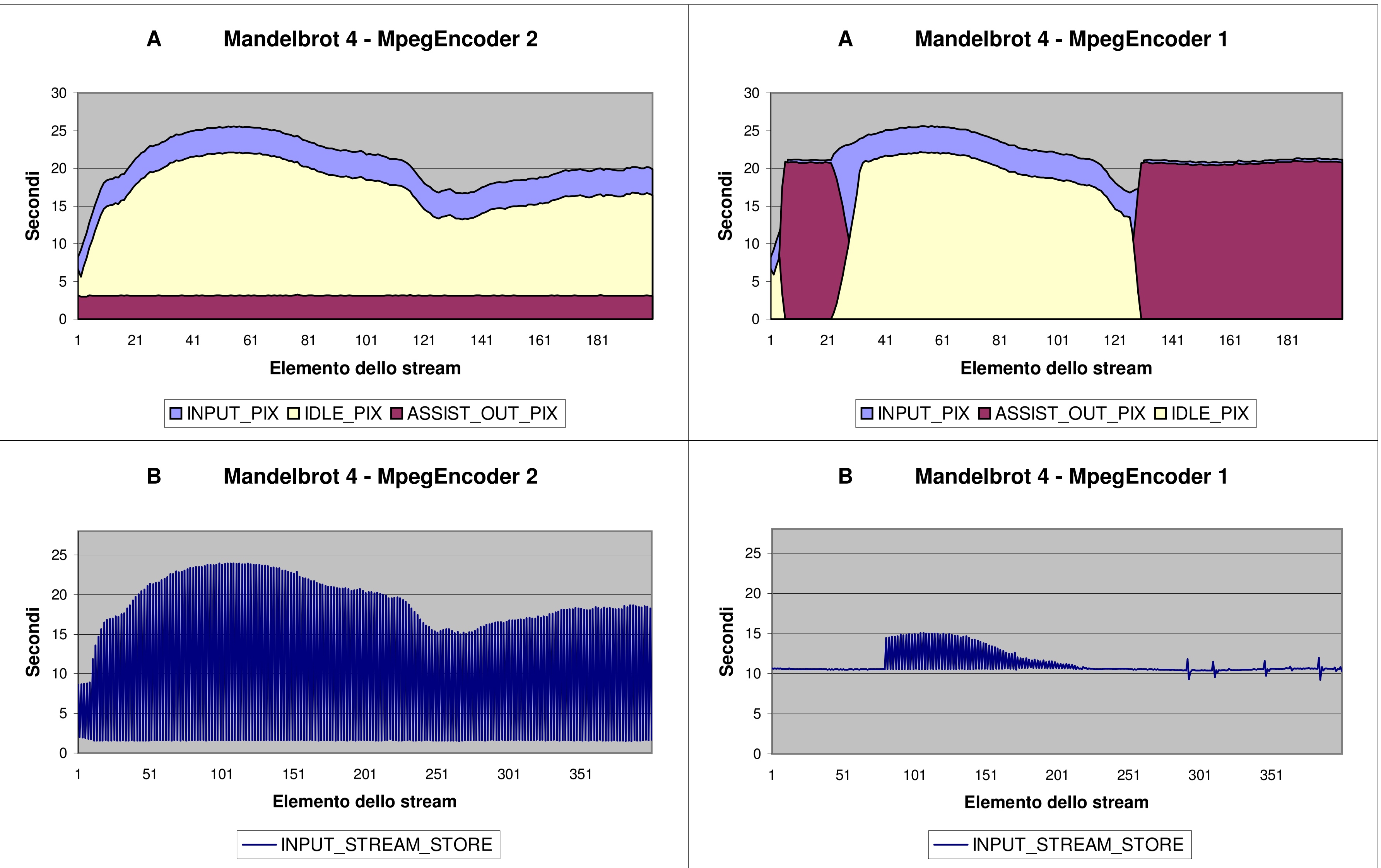
## B.1 Prove sul cluster Pianosa

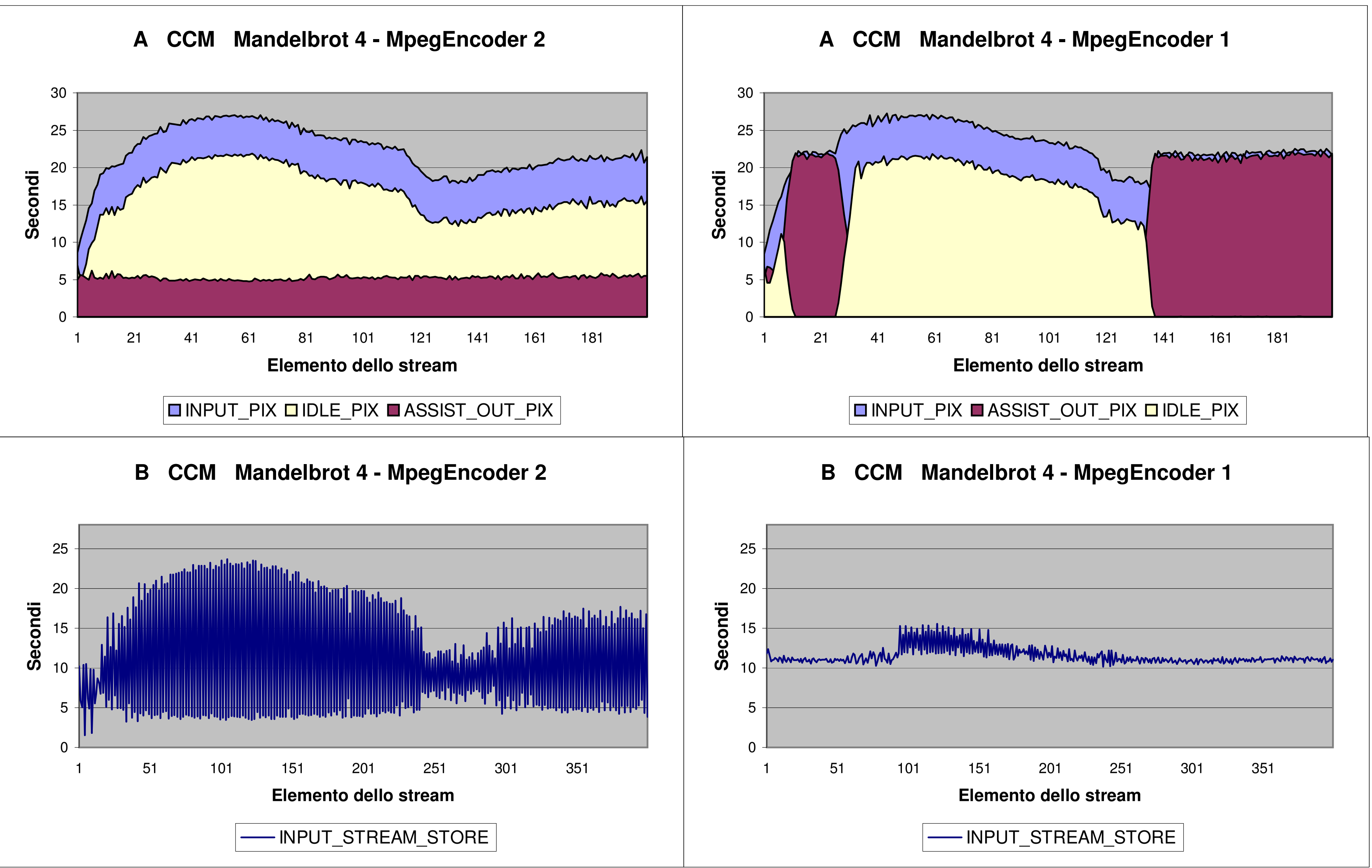
| Pagina | Collo di bottiglia Mandelbrot | Collo di bottiglia MpegEncoder | Dinamicità Applicazione | Applicazione Politica Adattiva |
|---|---|---|---|---|
| 187 | * | | | |
| 188 | * | | | |
| 189 | * | | | |
| 190 | * | | * | |
| 191 | * | | * | |
| 192 | * | | | |
| 193 | * | * | | |
| 194 | | * | * | |
| 195 | | | | * |
| 196 | * | | | |
| 197 | | * | * | |
| 198 | | * | * | |
| 199 | | * | | * |
| 200 | * | | | |
| 201 | | * | | |
| 202 | | * | * | |
| 203 | | * | | |
| 204 | * | | * | |
| 205 | | * | | |
| 206 | | * | | |
| 207 | | * | | |
| 208 | | | * | |
| 209 | | * | | |
| 210 | | * | | |
| 211 | | * | | |
| 212 | | * | * | |
| 213 | | * | | |

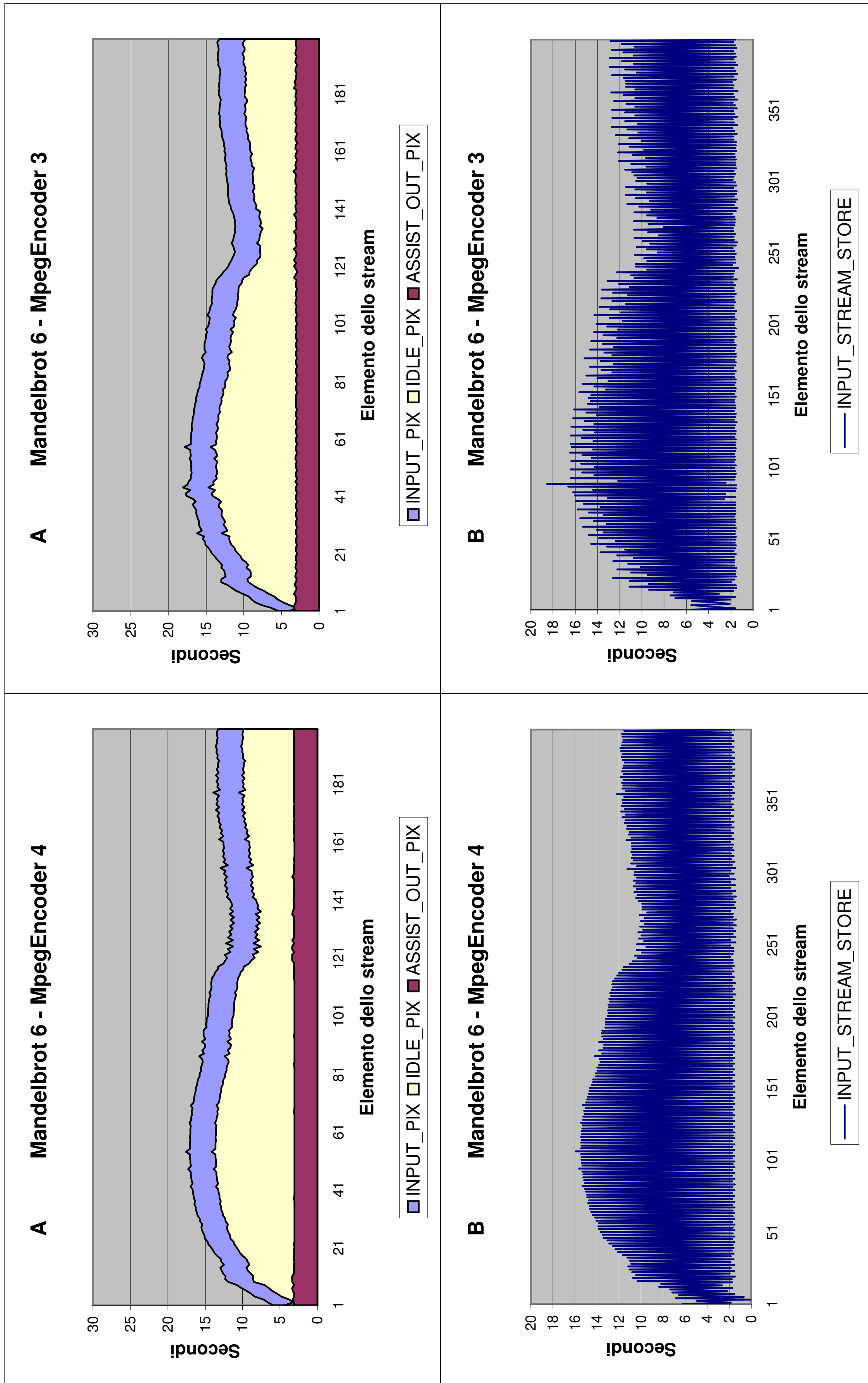

A
Mandelbrot 1 - MpegEncoder 3
Secondi
Elemento dello stream
INPUT_PIX
IDLE_PIX
ASSIST_OUT_PIX
A
Mandelbrot 1 - MpegEncoder 1
Secondi
Elemento dello stream
INPUT_PIX
IDLE_PIX
ASSIST_OUT_PIX
B
Mandelbrot 1 - MpegEncoder 3
Secondi
Elemento dello stream
INPUT_STREAM_STORE
B
Mandelbrot 1 - MpegEncoder 1
Secondi
Elemento dello stream
INPUT_STREAM_STORE

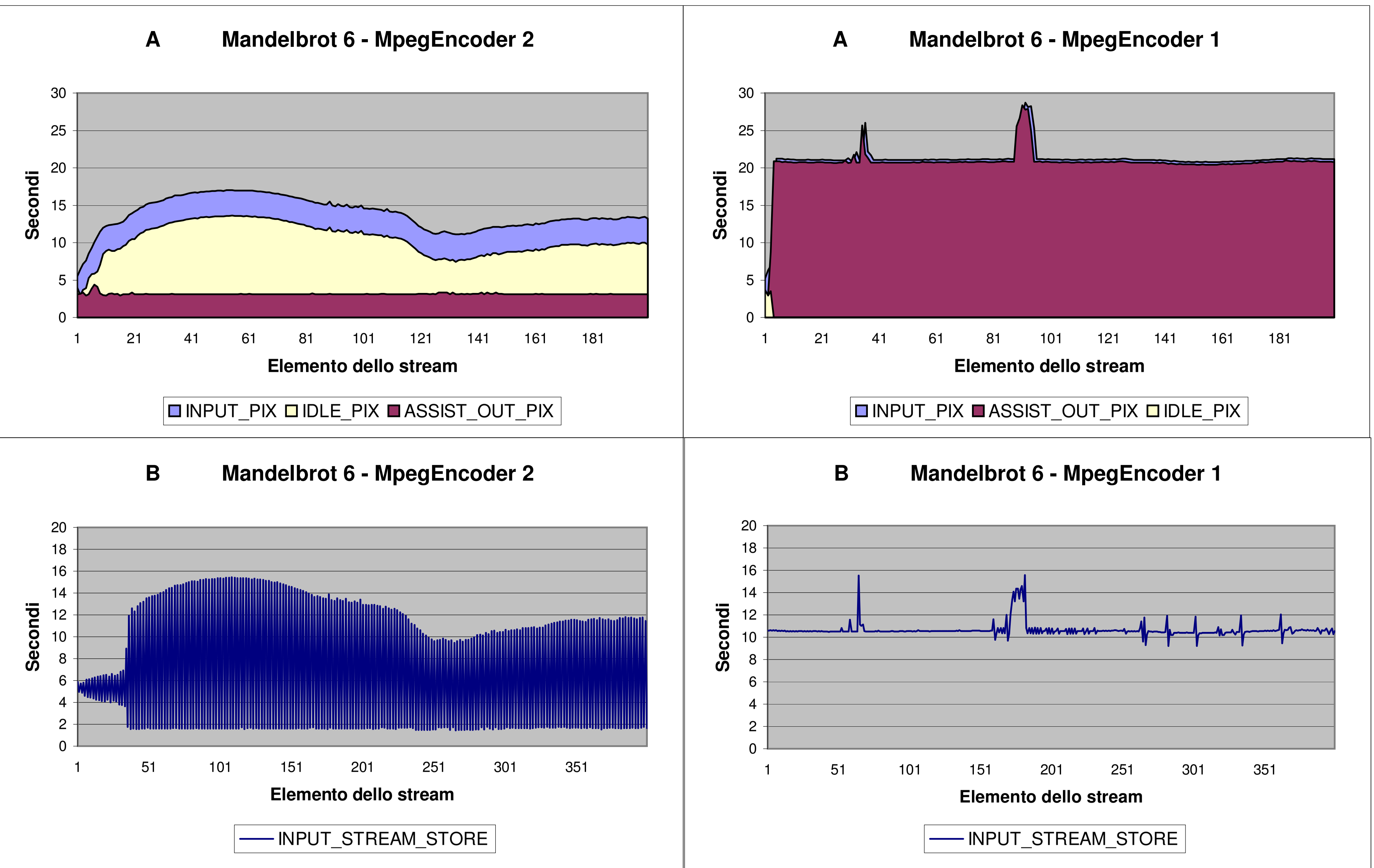

A
Mandelbrot 2 - MpegEncoder 1
Secondi
0
10
20
30
40
50
1
21
41
61
81
101
121
141
161
181
Elemento dello stream
INPUT_PIX
IDLE_PIX
ASSIST_OUT_PIX
B
Mandelbrot 2 - MpegEncoder 1
Secondi
0
5
10
15
20
25
30
35
40
1
51
101
151
201
251
301
351
Elemento dello stream
INPUT_STREAM_STORE

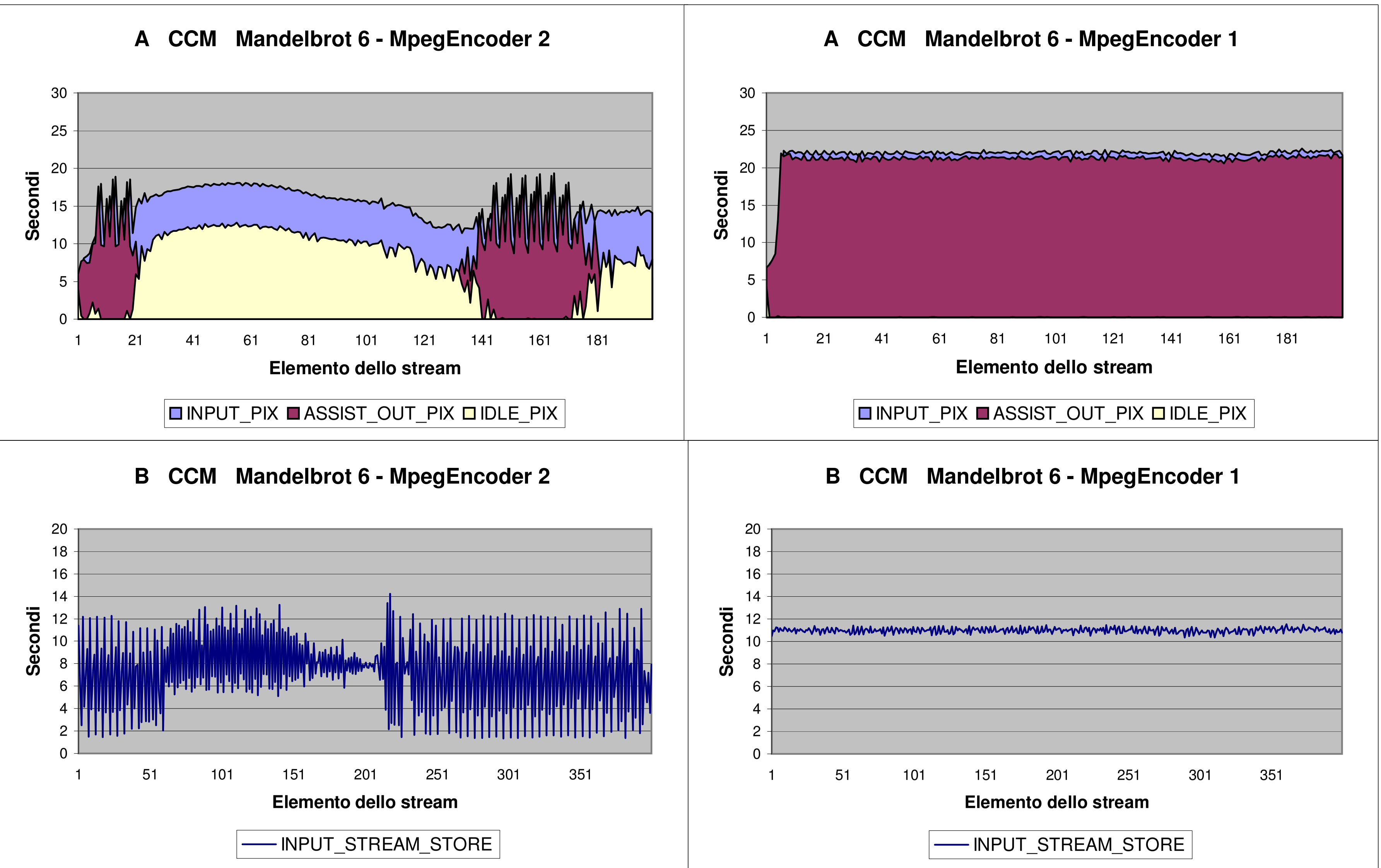

A
Mandelbrot 4 - MpegEncoder 4
Secondi
Elemento dello stream
INPUT_PIX
IDLE_PIX
ASSIST_OUT_PIX
A
Mandelbrot 4 - MpegEncoder 3
Secondi
Elemento dello stream
INPUT_PIX
IDLE_PIX
ASSIST_OUT_PIX
B
Mandelbrot 4 - MpegEncoder 4
Secondi
Elemento dello stream
INPUT_STREAM_STORE
B
Mandelbrot 4 - MpegEncoder 3
Secondi
Elemento dello stream
INPUT_STREAM_STORE

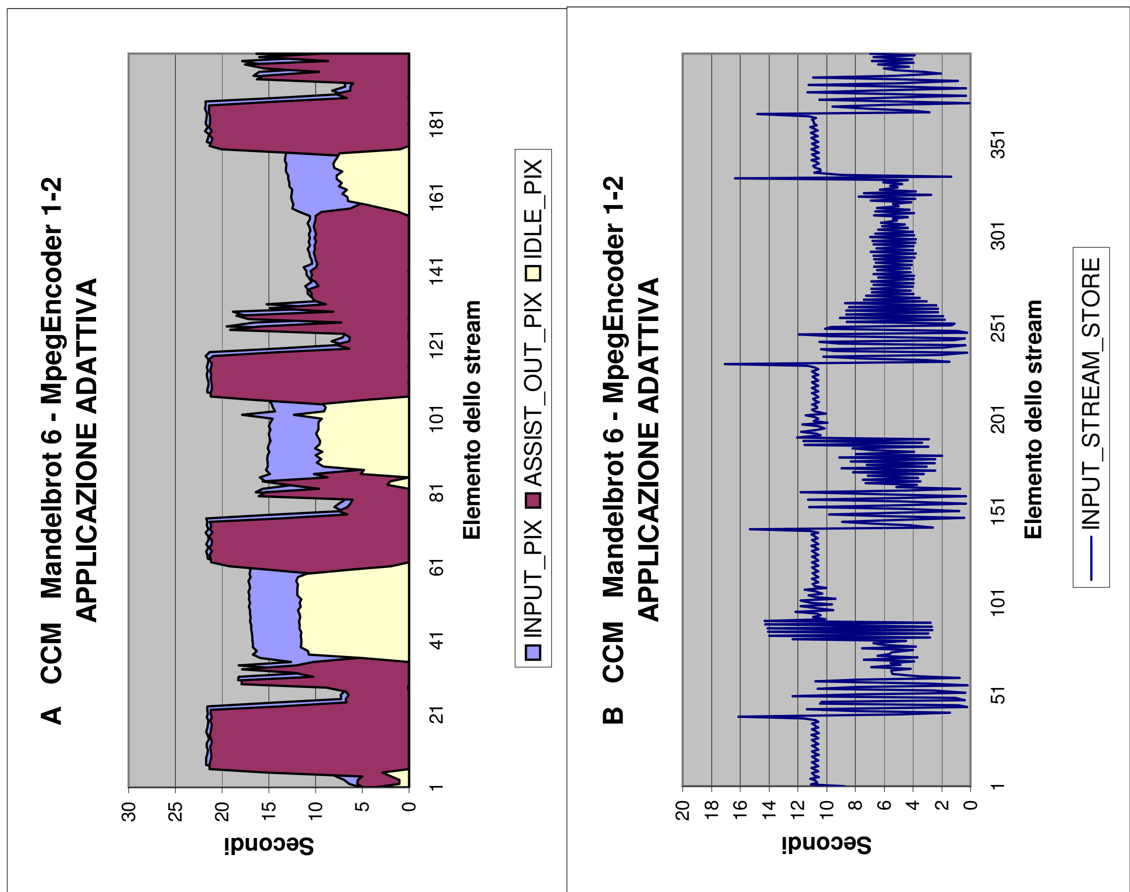

A Mandelbrot 4 - MpegEncoder 2
Secondi
Elemento dello stream
INPUT_PIX
IDLE_PIX
ASSIST_OUT_PIX
A Mandelbrot 4 - MpegEncoder 1
Secondi
Elemento dello stream
INPUT_PIX
ASSIST_OUT_PIX
IDLE_PIX
B Mandelbrot 4 - MpegEncoder 2
Secondi
Elemento dello stream
INPUT_STREAM_STORE
B Mandelbrot 4 - MpegEncoder 1
Secondi
Elemento dello stream
INPUT_STREAM_STORE

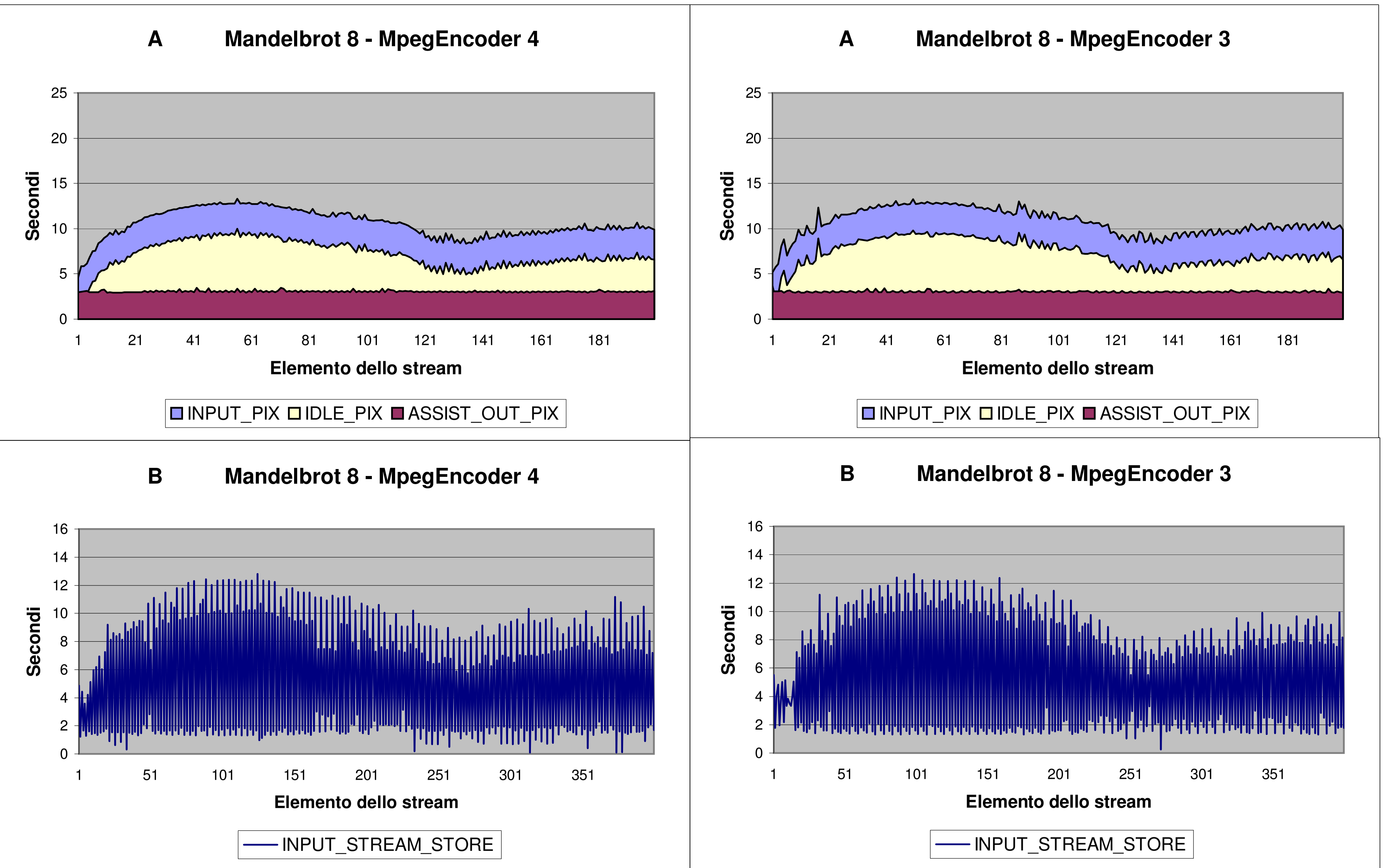

A CCM Mandelbrot 4 - MpegEncoder 2
Secondi
Elemento dello stream
INPUT_PIX
IDLE_PIX
ASSIST_OUT_PIX
A CCM Mandelbrot 4 - MpegEncoder 1
Secondi
Elemento dello stream
INPUT_PIX
ASSIST_OUT_PIX
IDLE_PIX
B CCM Mandelbrot 4 - MpegEncoder 2
Secondi
Elemento dello stream
INPUT_STREAM_STORE
B CCM Mandelbrot 4 - MpegEncoder 1
Secondi
Elemento dello stream
INPUT_STREAM_STORE

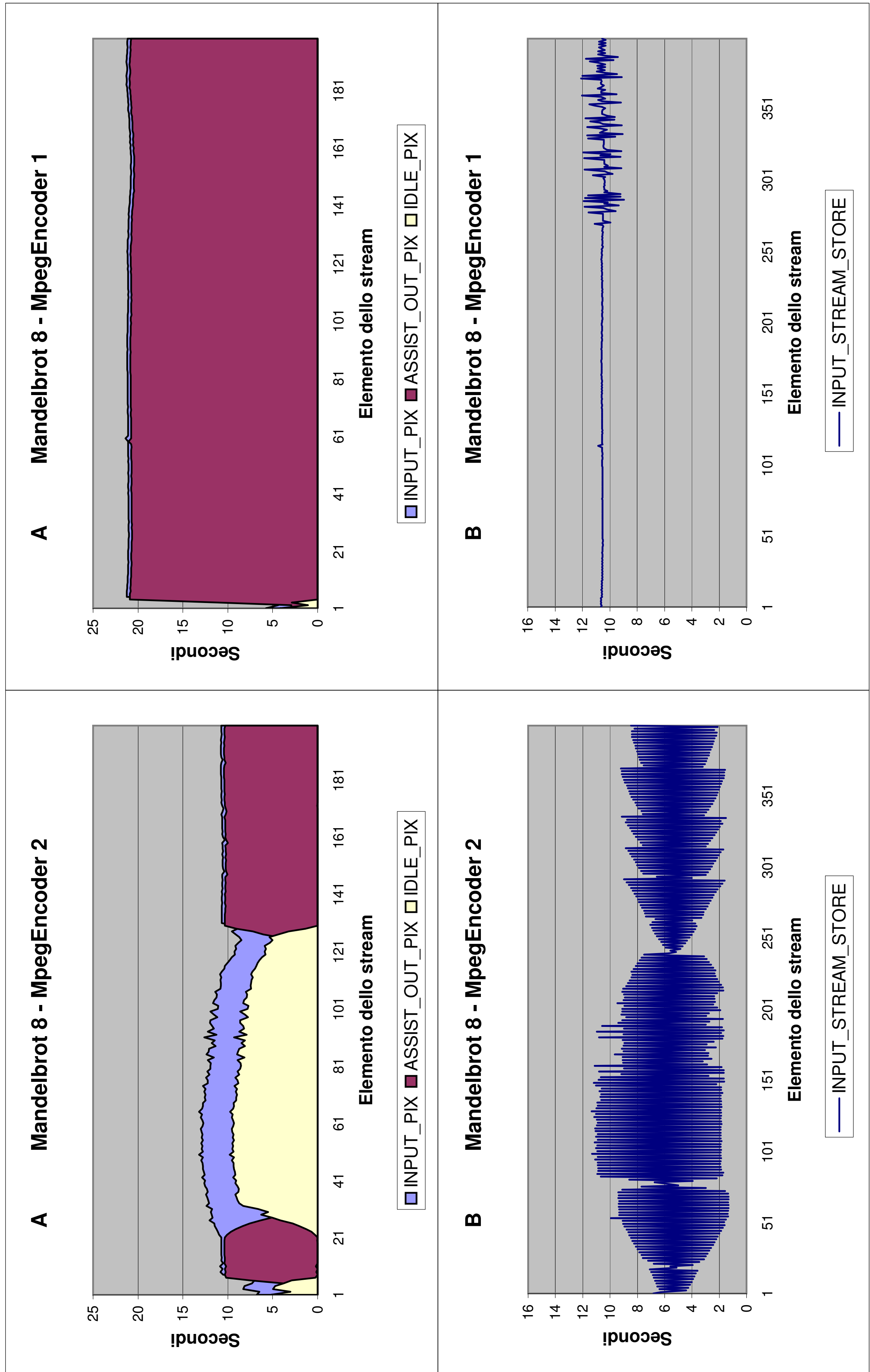

A
Mandelbrot 6 - MpegEncoder 4
Secondi
30
25
20
15
10
5
0
1
21
41
61
81
101
121
141
161
181
Elemento dello stream
INPUT_PIX
IDLE_PIX
ASSIST_OUT_PIX
A
Mandelbrot 6 - MpegEncoder 3
Secondi
Elemento dello stream
INPUT_PIX
IDLE_PIX
ASSIST_OUT_PIX
B
Mandelbrot 6 - MpegEncoder 4
Secondi
20
18
16
14
12
10
8
6
4
2
0
1
51
101
151
201
251
301
351
Elemento dello stream
INPUT_STREAM_STORE
B
Mandelbrot 6 - MpegEncoder 3
Secondi
Elemento dello stream
INPUT_STREAM_STORE

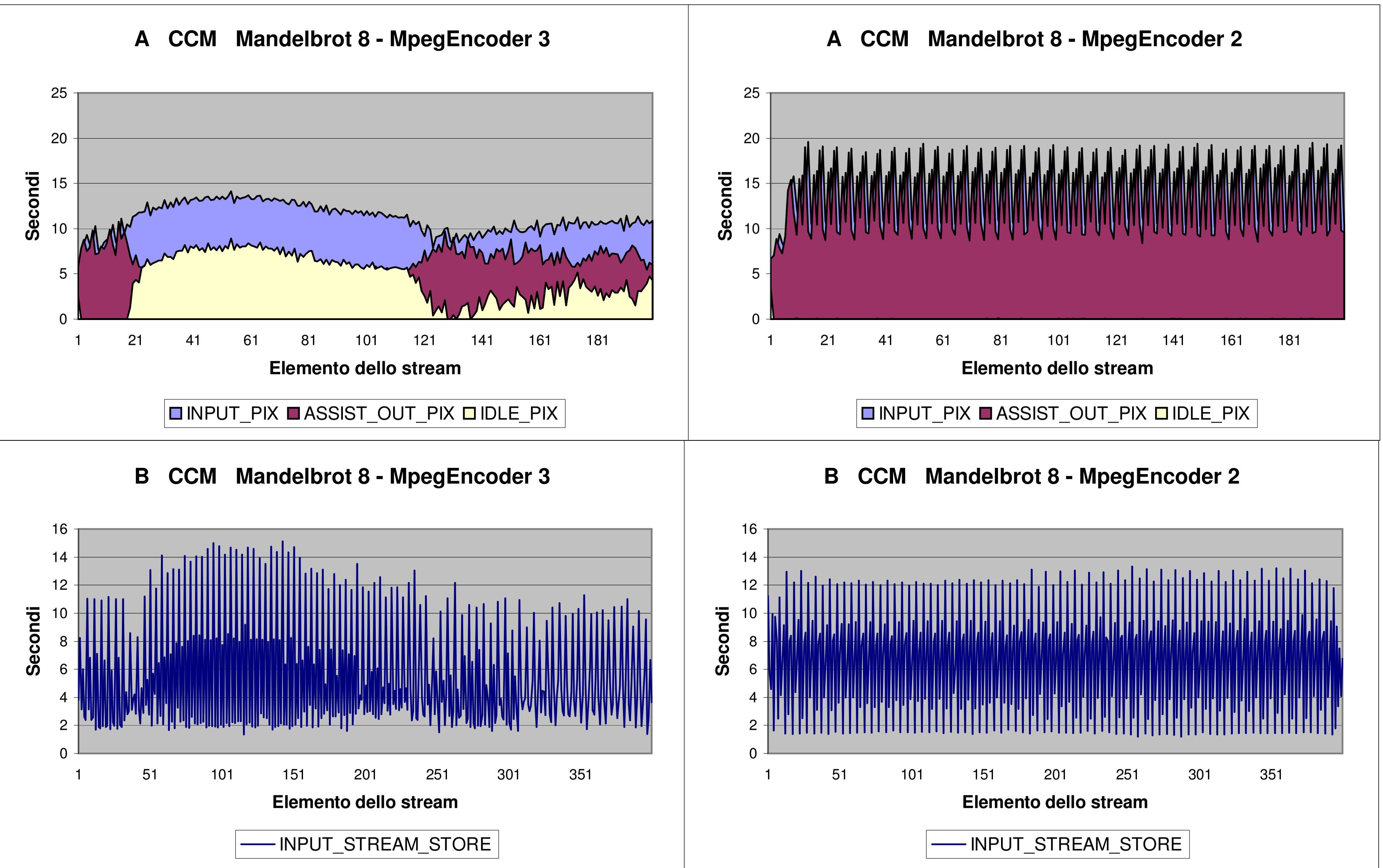

A Mandelbrot 6 - MpegEncoder 2
Secondi
Elemento dello stream
INPUT_PIX IDLE_PIX ASSIST_OUT_PIX
A Mandelbrot 6 - MpegEncoder 1
Secondi
Elemento dello stream
INPUT_PIX ASSIST_OUT_PIX IDLE_PIX
B Mandelbrot 6 - MpegEncoder 2
Secondi
Elemento dello stream
INPUT_STREAM_STORE
B Mandelbrot 6 - MpegEncoder 1
Secondi
Elemento dello stream
INPUT_STREAM_STORE

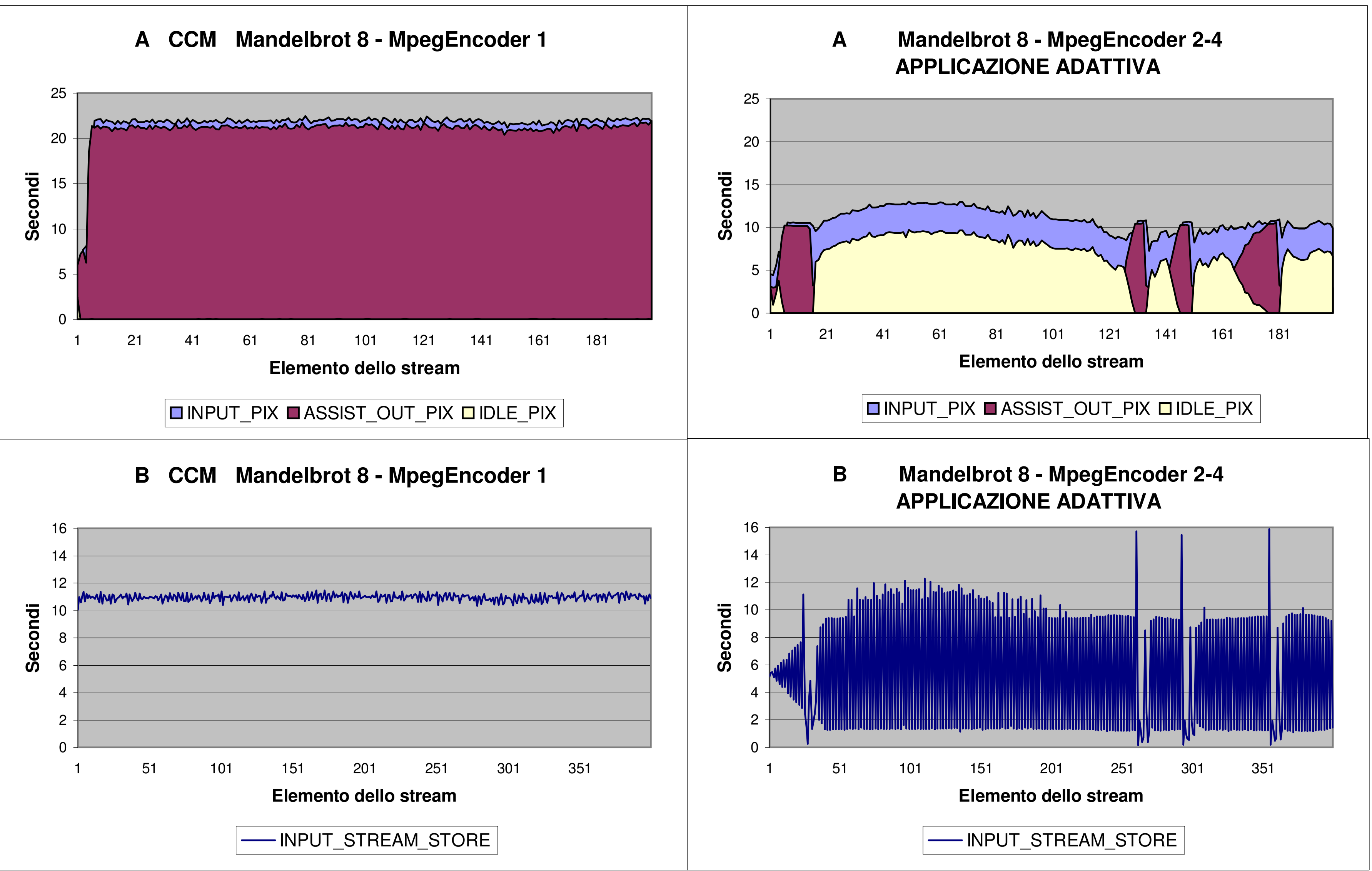

A CCM Mandelbrot 6 - MpegEncoder 2
Secondi
Elemento dello stream
INPUT_PIX
ASSIST_OUT_PIX
IDLE_PIX
A CCM Mandelbrot 6 - MpegEncoder 1
Secondi
Elemento dello stream
INPUT_PIX
ASSIST_OUT_PIX
IDLE_PIX
B CCM Mandelbrot 6 - MpegEncoder 2
Secondi
Elemento dello stream
INPUT_STREAM_STORE
B CCM Mandelbrot 6 - MpegEncoder 1
Secondi
Elemento dello stream
INPUT_STREAM_STORE

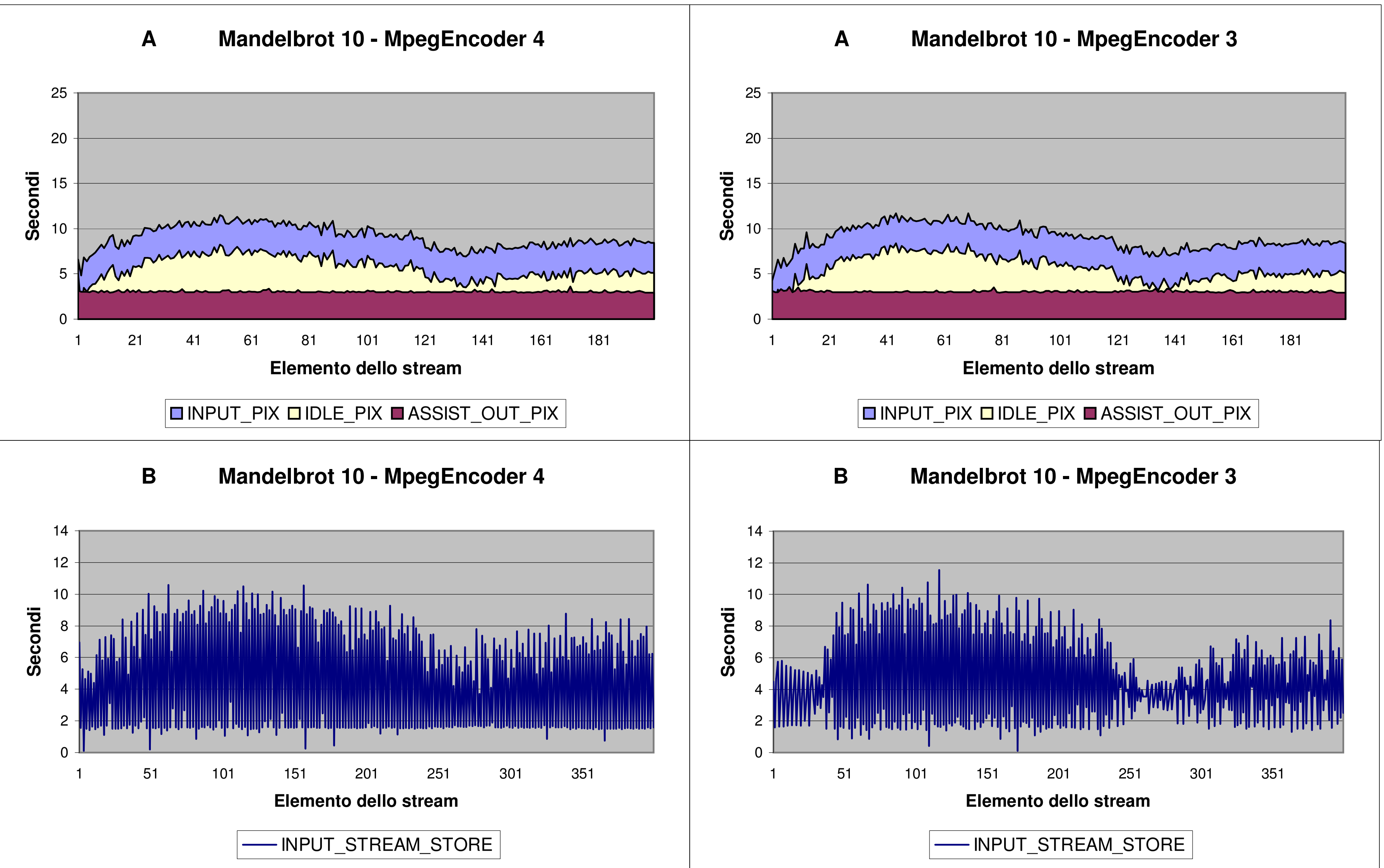

A CCM Mandelbrot 6 - MpegEncoder 1-2
APPLICAZIONE ADATTIVA
Secondi
Elemento dello stream
INPUT_PIX
ASSIST_OUT_PIX
IDLE_PIX
B CCM Mandelbrot 6 - MpegEncoder 1-2
APPLICAZIONE ADATTIVA
Secondi
Elemento dello stream
INPUT_STREAM_STORE

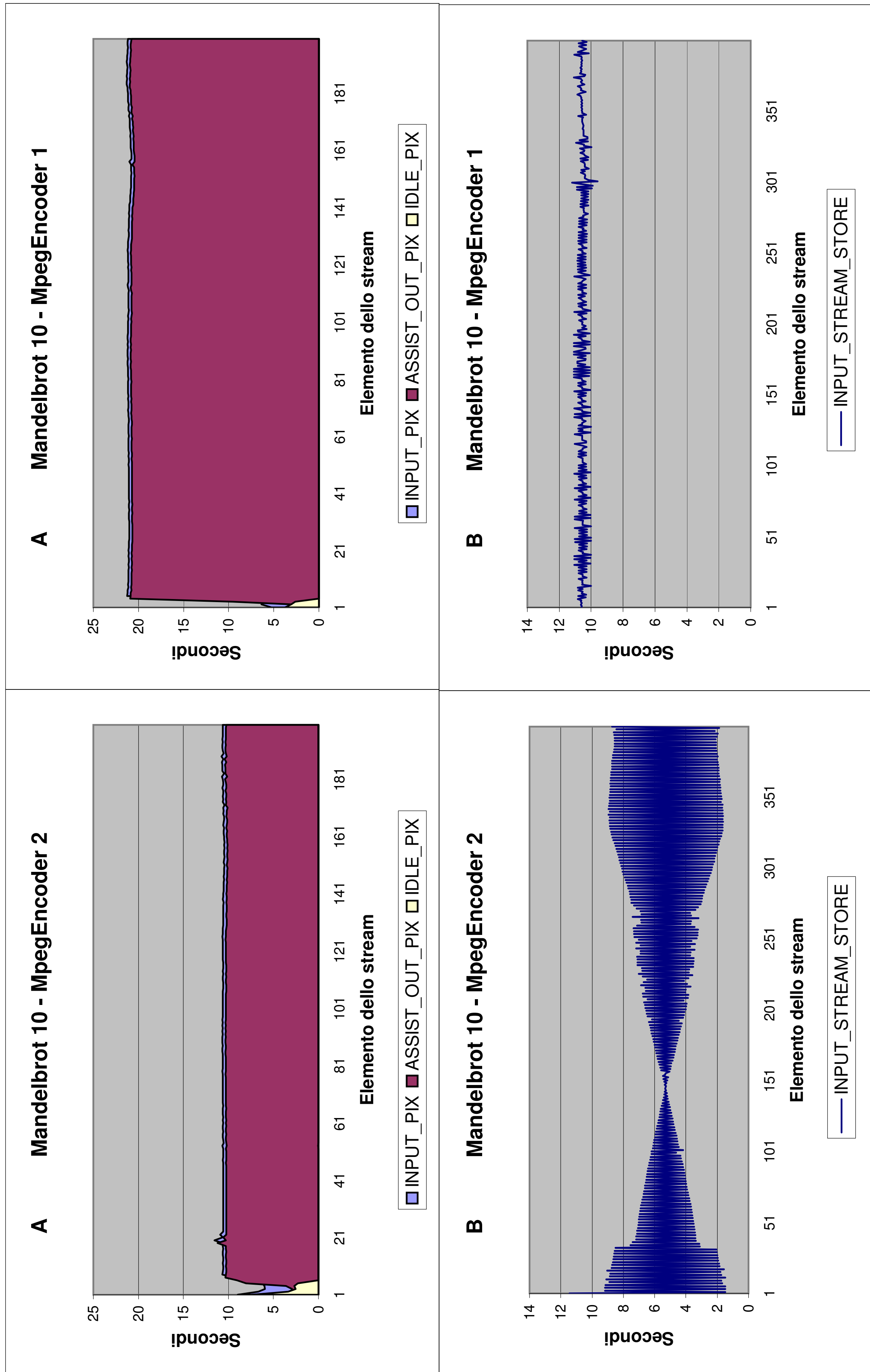

A Mandelbrot 8 - MpegEncoder 4
Secondi
Elemento dello stream
INPUT_PIX IDLE_PIX ASSIST_OUT_PIX
A Mandelbrot 8 - MpegEncoder 3
Secondi
Elemento dello stream
INPUT_PIX IDLE_PIX ASSIST_OUT_PIX
B Mandelbrot 8 - MpegEncoder 4
Secondi
Elemento dello stream
INPUT_STREAM_STORE
B Mandelbrot 8 - MpegEncoder 3
Secondi
Elemento dello stream
INPUT_STREAM_STORE

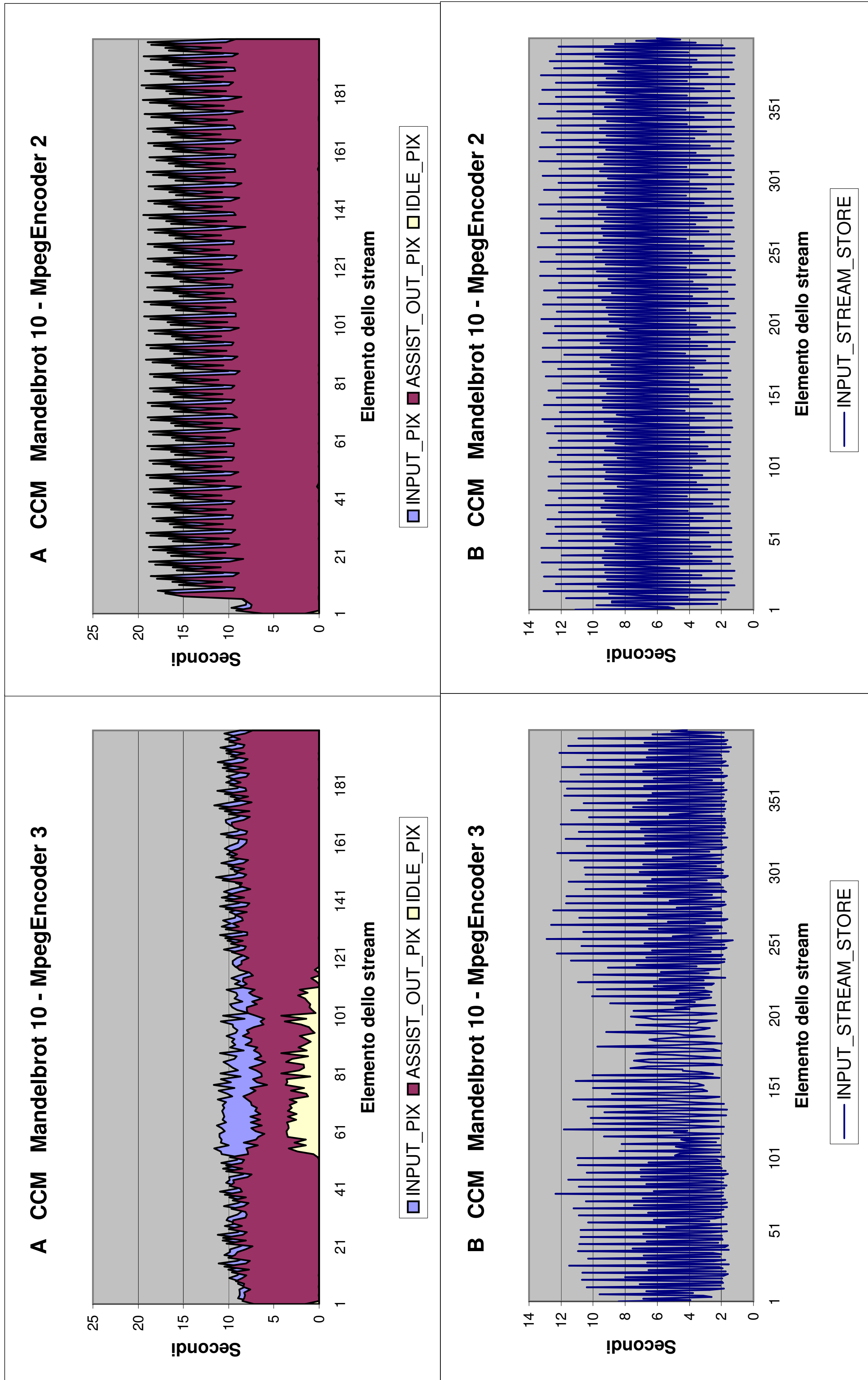

A Mandelbrot 8 - MpegEncoder 1
Secondi
Elemento dello stream
INPUT_PIX ASSIST_OUT_PIX IDLE_PIX
B Mandelbrot 8 - MpegEncoder 1
Secondi
Elemento dello stream
INPUT_STREAM_STORE
A Mandelbrot 8 - MpegEncoder 2
Secondi
Elemento dello stream
INPUT_PIX ASSIST_OUT_PIX IDLE_PIX
B Mandelbrot 8 - MpegEncoder 2
Secondi
Elemento dello stream
INPUT_STREAM_STORE

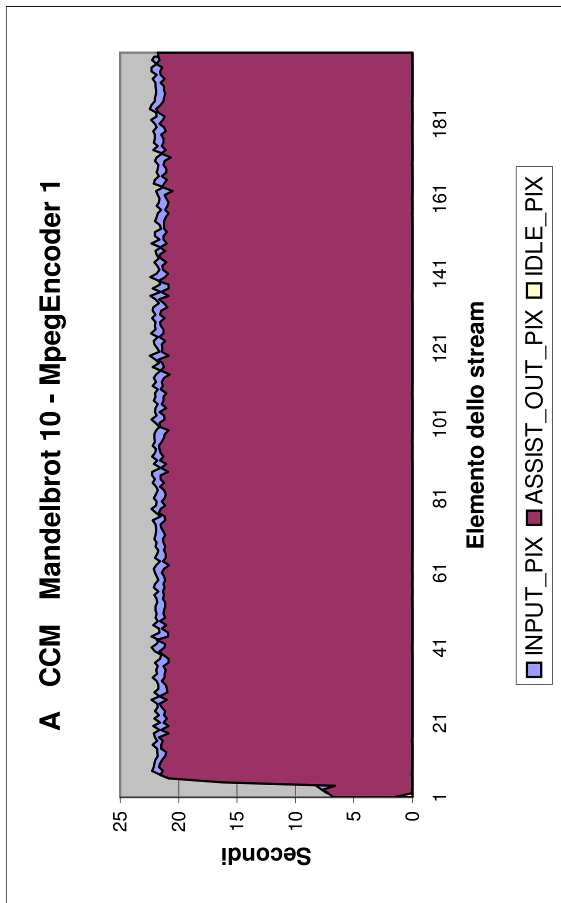

A CCM Mandelbrot 8 - MpegEncoder 3
Secondi
Elemento dello stream
INPUT_PIX
ASSIST_OUT_PIX
IDLE_PIX
A CCM Mandelbrot 8 - MpegEncoder 2
Secondi
Elemento dello stream
INPUT_PIX
ASSIST_OUT_PIX
IDLE_PIX
B CCM Mandelbrot 8 - MpegEncoder 3
Secondi
Elemento dello stream
INPUT_STREAM_STORE
B CCM Mandelbrot 8 - MpegEncoder 2
Secondi
Elemento dello stream
INPUT_STREAM_STORE

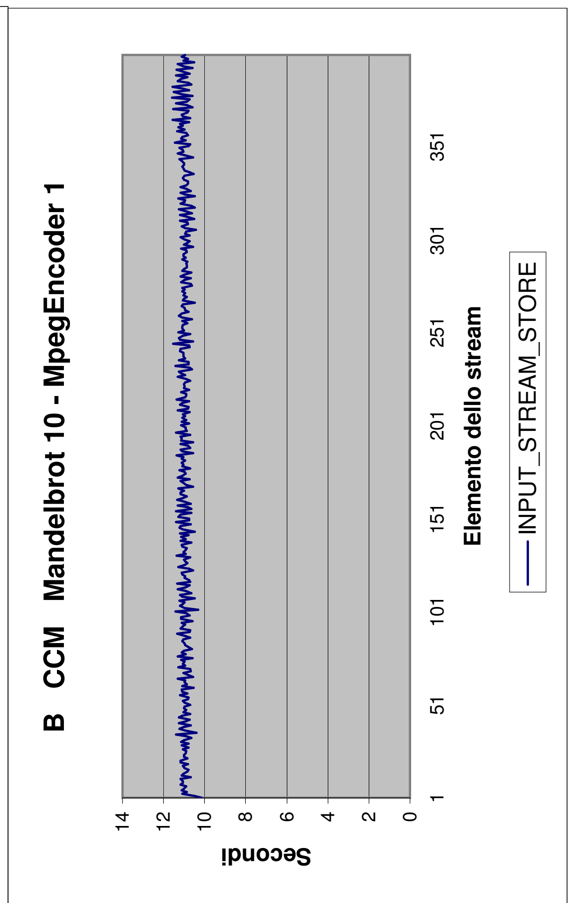

A CCM Mandelbrot 8 - MpegEncoder 1
Secondi
Elemento dello stream
INPUT_PIX ASSIST_OUT_PIX IDLE_PIX
A Mandelbrot 8 - MpegEncoder 2-4
APPLICAZIONE ADATTIVA
Secondi
Elemento dello stream
INPUT_PIX ASSIST_OUT_PIX IDLE_PIX
B CCM Mandelbrot 8 - MpegEncoder 1
Secondi
Elemento dello stream
INPUT_STREAM_STORE
B Mandelbrot 8 - MpegEncoder 2-4
APPLICAZIONE ADATTIVA
Secondi
Elemento dello stream
INPUT_STREAM_STORE

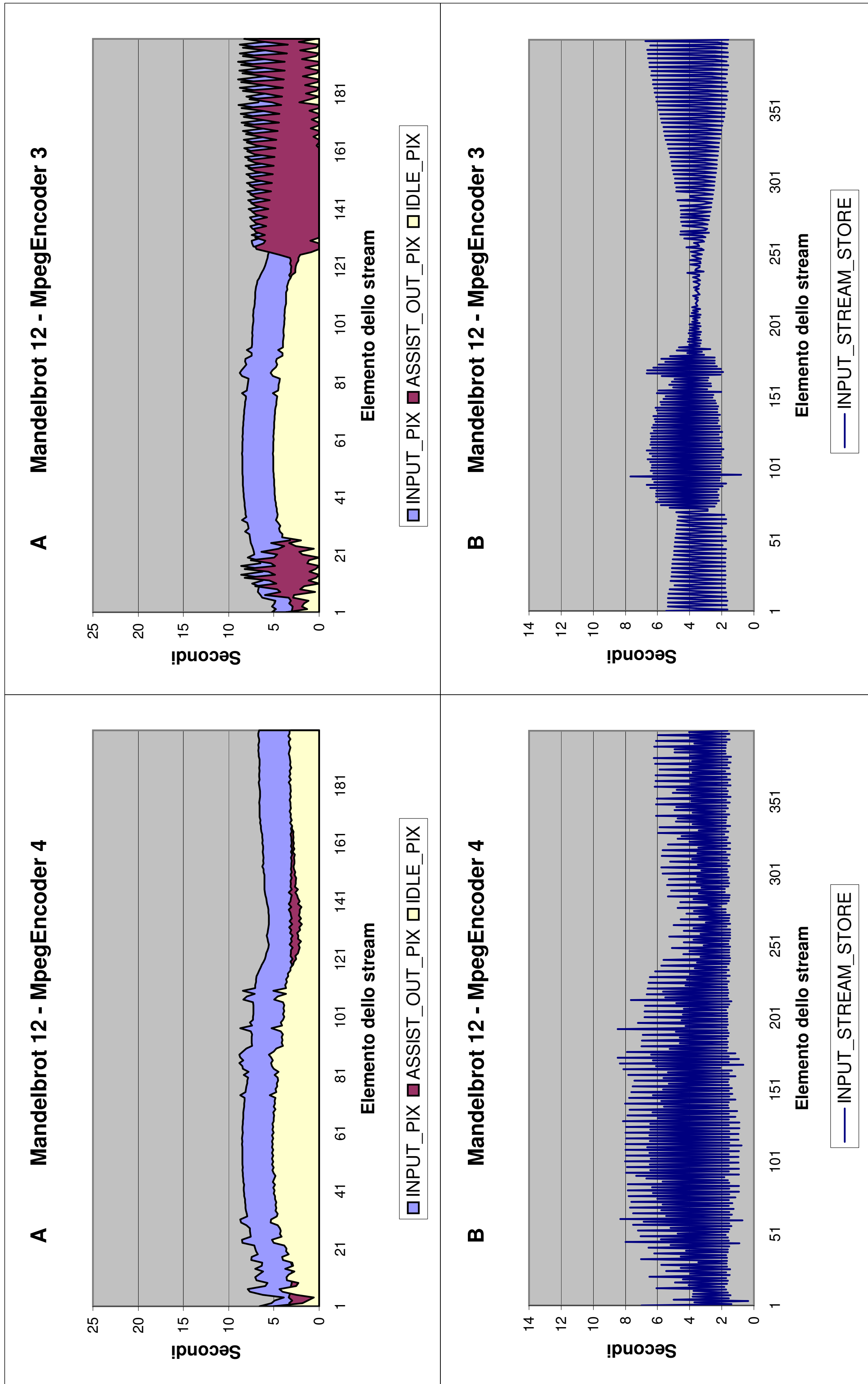

A Mandelbrot 10 - MpegEncoder 4
Secondi
25 20 15 10 5 0
1 21 41 61 81 101 121 141 161 181
Elemento dello stream
INPUT_PIX IDLE_PIX ASSIST_OUT_PIX
A Mandelbrot 10 - MpegEncoder 3
Secondi
25 20 15 10 5 0
1 21 41 61 81 101 121 141 161 181
Elemento dello stream
INPUT_PIX IDLE_PIX ASSIST_OUT_PIX
B Mandelbrot 10 - MpegEncoder 4
Secondi
14 12 10 8 6 4 2 0
1 51 101 151 201 251 301 351
Elemento dello stream
INPUT_STREAM_STORE
B Mandelbrot 10 - MpegEncoder 3
Secondi
14 12 10 8 6 4 2 0
1 51 101 151 201 251 301 351
Elemento dello stream
INPUT_STREAM_STORE

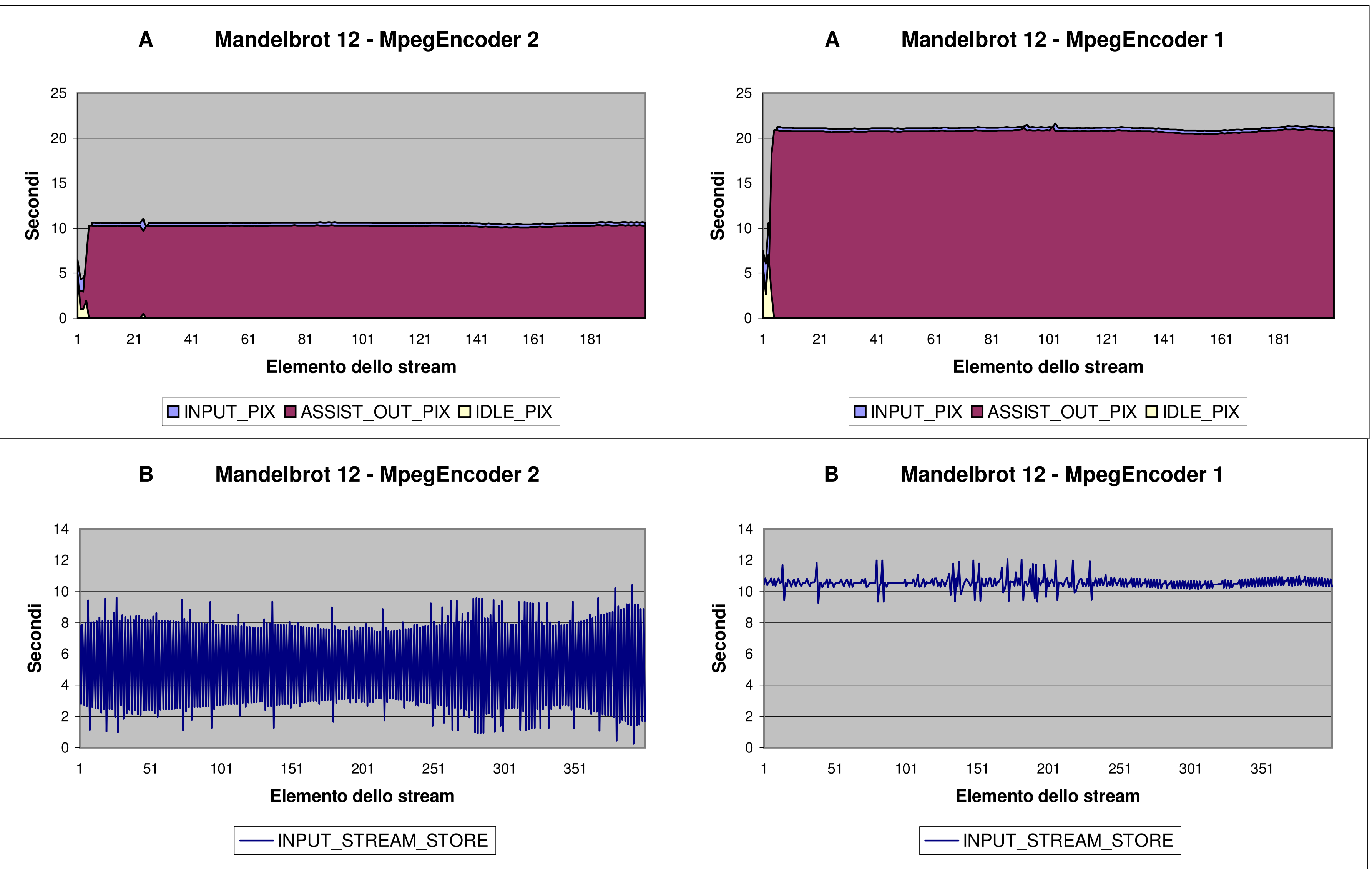

A Mandelbrot 10 - MpegEncoder 1
Secondi
Elemento dello stream
INPUT_PIX
ASSIST_OUT_PIX
IDLE_PIX
A Mandelbrot 10 - MpegEncoder 2
Secondi
Elemento dello stream
INPUT_PIX
ASSIST_OUT_PIX
IDLE_PIX
B Mandelbrot 10 - MpegEncoder 1
Secondi
Elemento dello stream
INPUT_STREAM_STORE
B Mandelbrot 10 - MpegEncoder 2
Secondi
Elemento dello stream
INPUT_STREAM_STORE

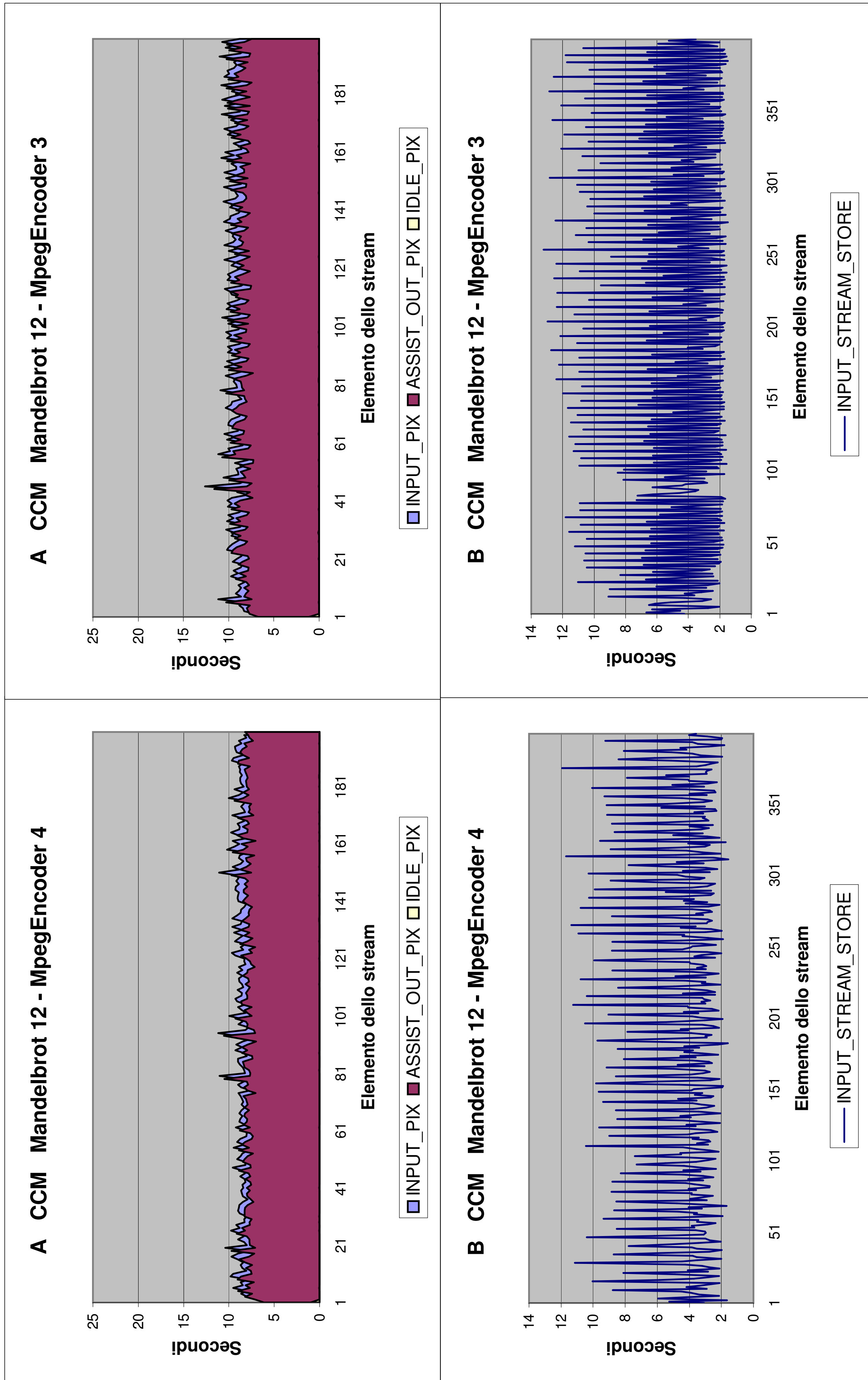

A CCM Mandelbrot 10 - MpegEncoder 3
Seconds
Secondi
25
20
15
10
5
0
1
21
41
61
81
101
121
141
161
181
Elemento dello stream
INPUT_PIX
ASSIST_OUT_PIX
IDLE_PIX
A CCM Mandelbrot 10 - MpegEncoder 2
Secondi
Elemento dello stream
INPUT_PIX
ASSIST_OUT_PIX
IDLE_PIX
B CCM Mandelbrot 10 - MpegEncoder 3
Secondi
14
12
10
8
6
4
2
0
1
51
101
151
201
251
301
351
Elemento dello stream
INPUT_STREAM_STORE
B CCM Mandelbrot 10 - MpegEncoder 2
Secondi
Elemento dello stream
INPUT_STREAM_STORE

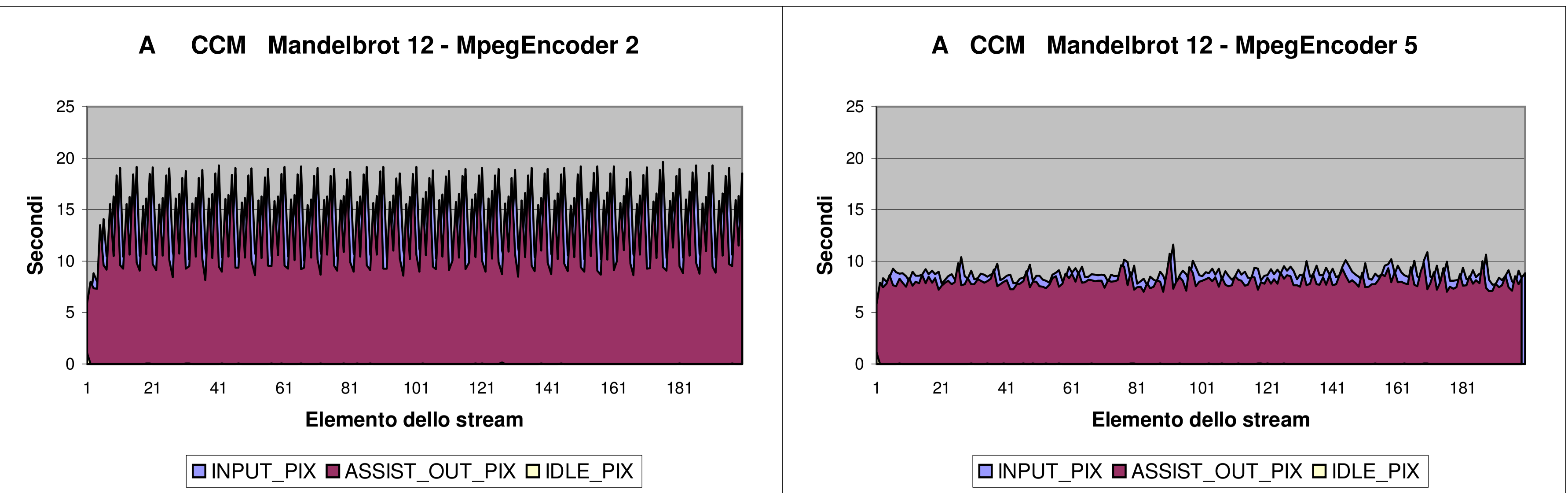

A CCM Mandelbrot 10 - MpegEncoder 1
Secondi
25
20
15
10
5
0
1
21
41
61
81
101
121
141
161
181
Elemento dello stream
INPUT_PIX
ASSIST_OUT_PIX
IDLE_PIX

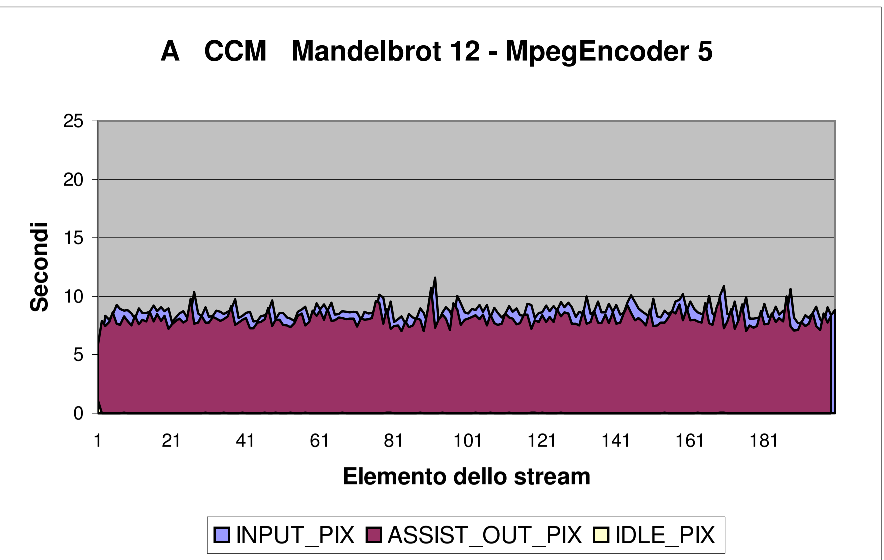

B CCM Mandelbrot 10 - MpegEncoder 1
Secondi
14
12
10
8
6
4
2
0
1
51
101
151
201
251
301
351
Elemento dello stream
INPUT_STREAM_STORE

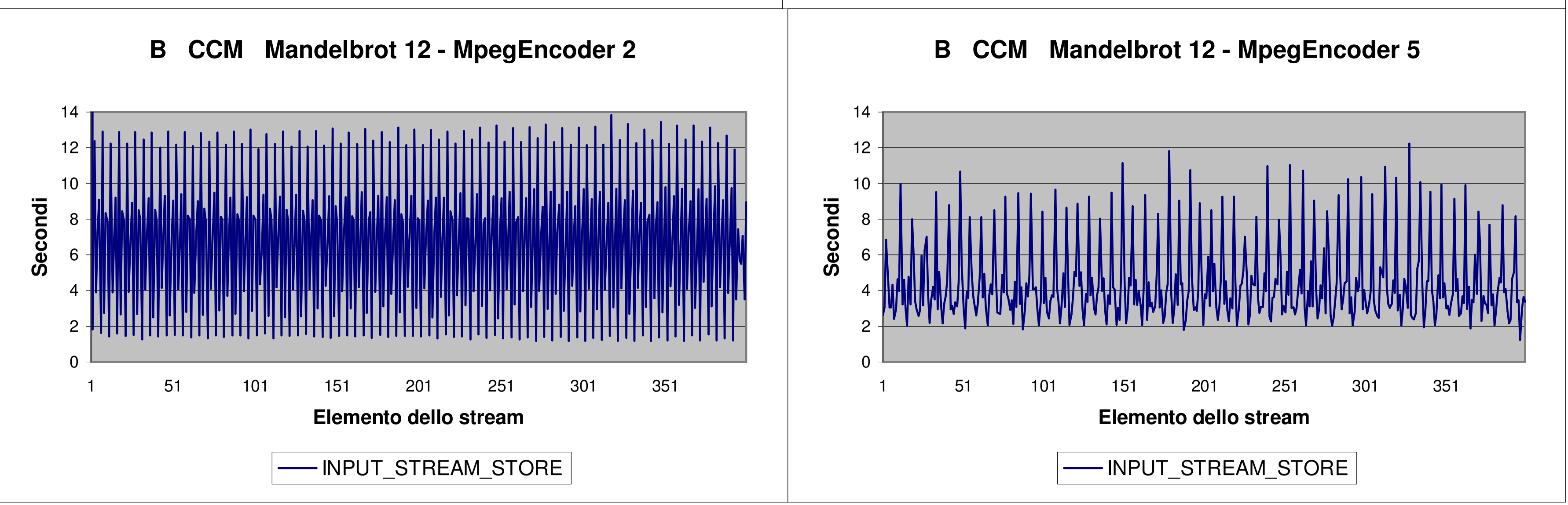

A Mandelbrot 12 - MpegEncoder 4
Seconldi
Elemento dello stream
INPUT_PIX ASSIST_OUT_PIX IDLE_PIX
A Mandelbrot 12 - MpegEncoder 3
Secondi
Elemento dello stream
INPUT_PIX ASSIST_OUT_PIX IDLE_PIX
B Mandelbrot 12 - MpegEncoder 4
Secondi
Elemento dello stream
INPUT_STREAM_STORE
B Mandelbrot 12 - MpegEncoder 3
Secondi
Elemento dello stream
INPUT_STREAM_STORE

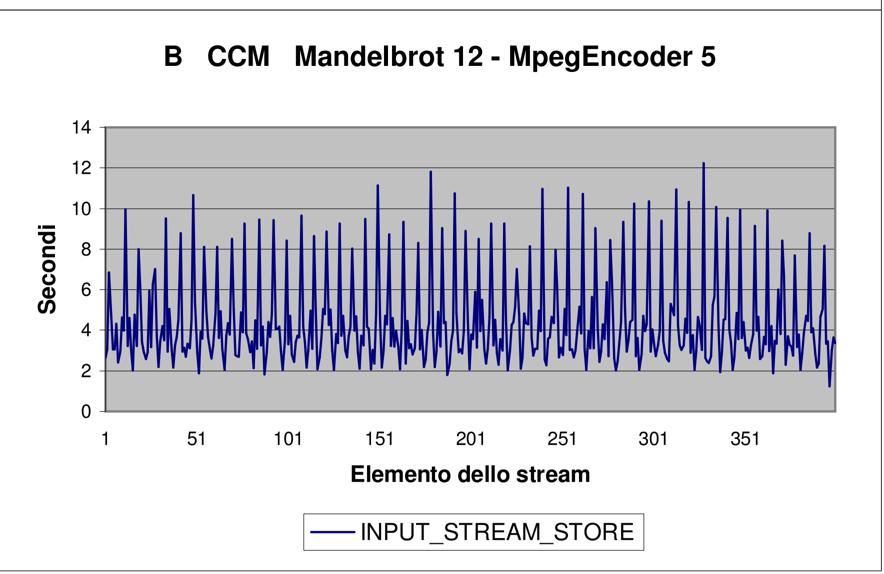

A Mandelbrot 12 - MpegEncoder 2
Secondi
Elemento dello stream
INPUT_PIX
ASSIST_OUT_PIX
IDLE_PIX
A Mandelbrot 12 - MpegEncoder 1
Secondi
Elemento dello stream
INPUT_PIX
ASSIST_OUT_PIX
IDLE_PIX
B Mandelbrot 12 - MpegEncoder 2
Secondi
Elemento dello stream
INPUT_STREAM_STORE
B Mandelbrot 12 - MpegEncoder 1
Secondi
Elemento dello stream
INPUT_STREAM_STORE

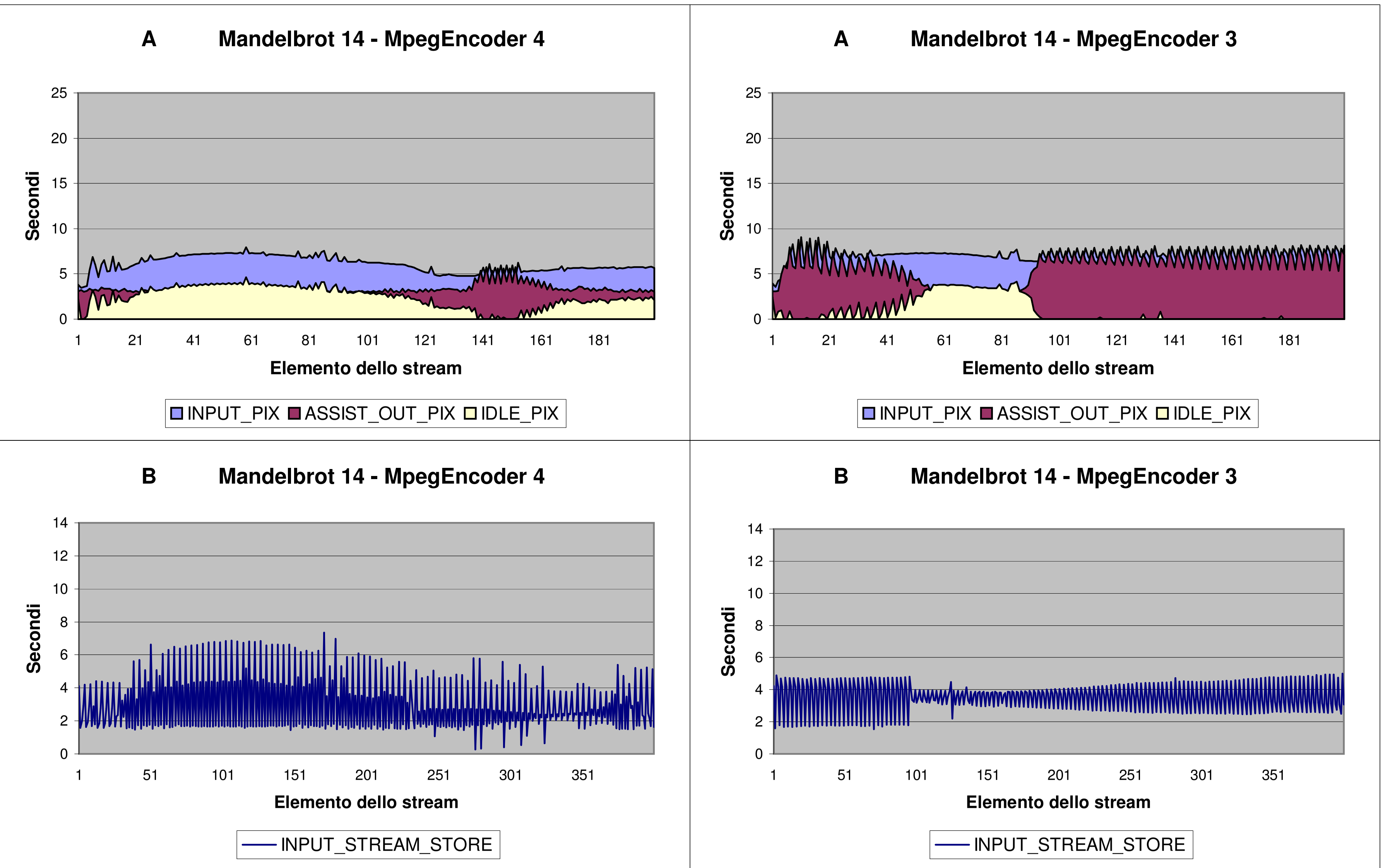

A CCM Mandelbrot 12 - MpegEncoder 4
Seconds
25
20
15
10
5
0
1
21
41
61
81
101
121
141
161
181
Elemento dello stream
INPUT_PIX
ASSIST_OUT_PIX
IDLE_PIX
A CCM Mandelbrot 12 - MpegEncoder 3
Seconds
25
20
15
10
5
0
1
21
41
61
81
101
121
141
161
181
Elemento dello stream
INPUT_PIX
ASSIST_OUT_PIX
IDLE_PIX
B CCM Mandelbrot 12 - MpegEncoder 4
Seconds
14
12
10
8
6
4
2
0
1
51
101
151
201
251
301
351
Elemento dello stream
INPUT_STREAM_STORE
B CCM Mandelbrot 12 - MpegEncoder 3
Seconds
14
12
10
8
6
4
2
0
1
51
101
151
201
251
301
351
Elemento dello stream
INPUT_STREAM_STORE

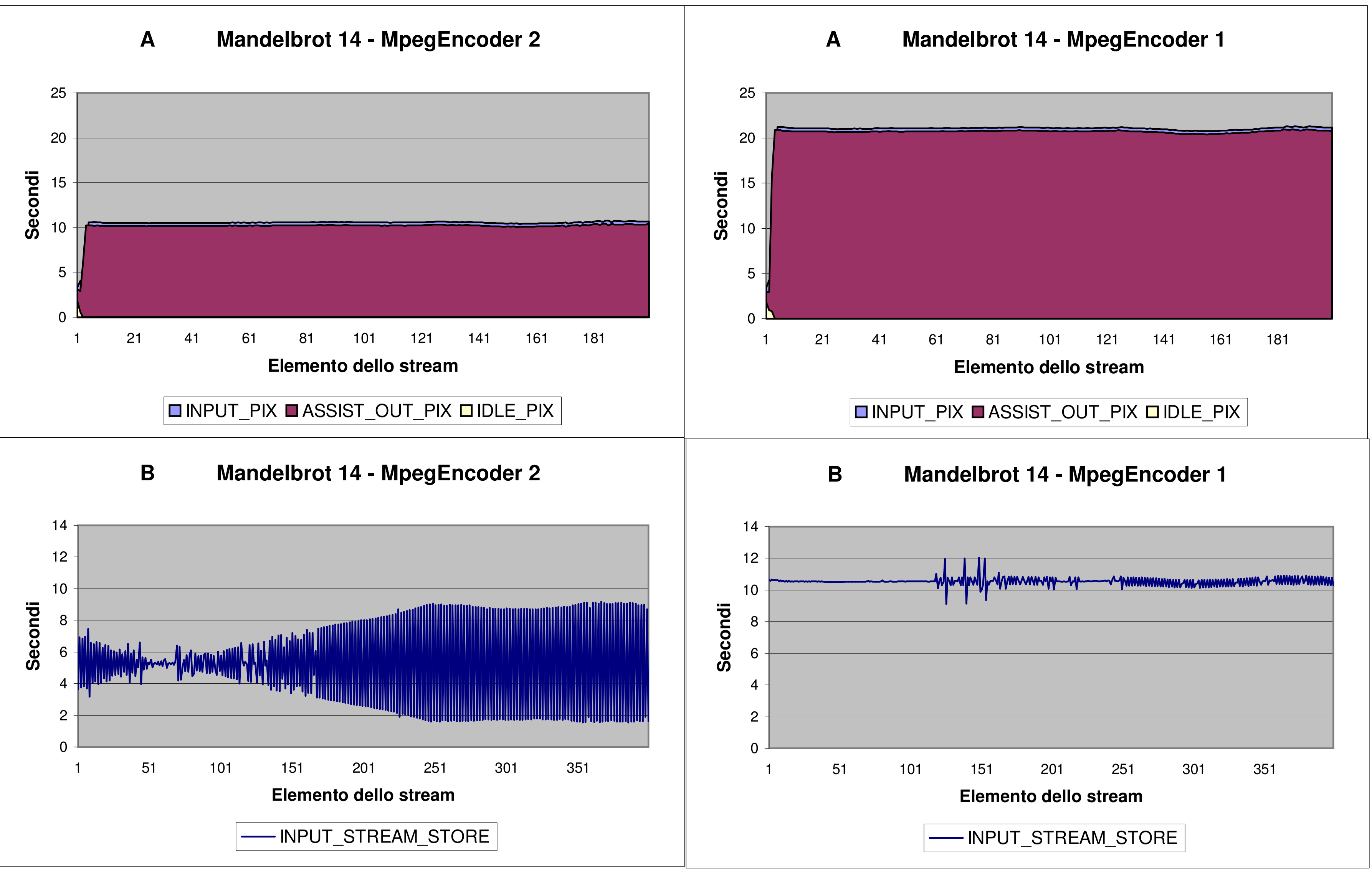

A CCM Mandelbrot 12 - MpegEncoder 2
Secondi
0 5 10 15 20 25
1 21 41 61 81 101 121 141 161 181
Elemento dello stream
INPUT_PIX ASSIST_OUT_PIX IDLE_PIX

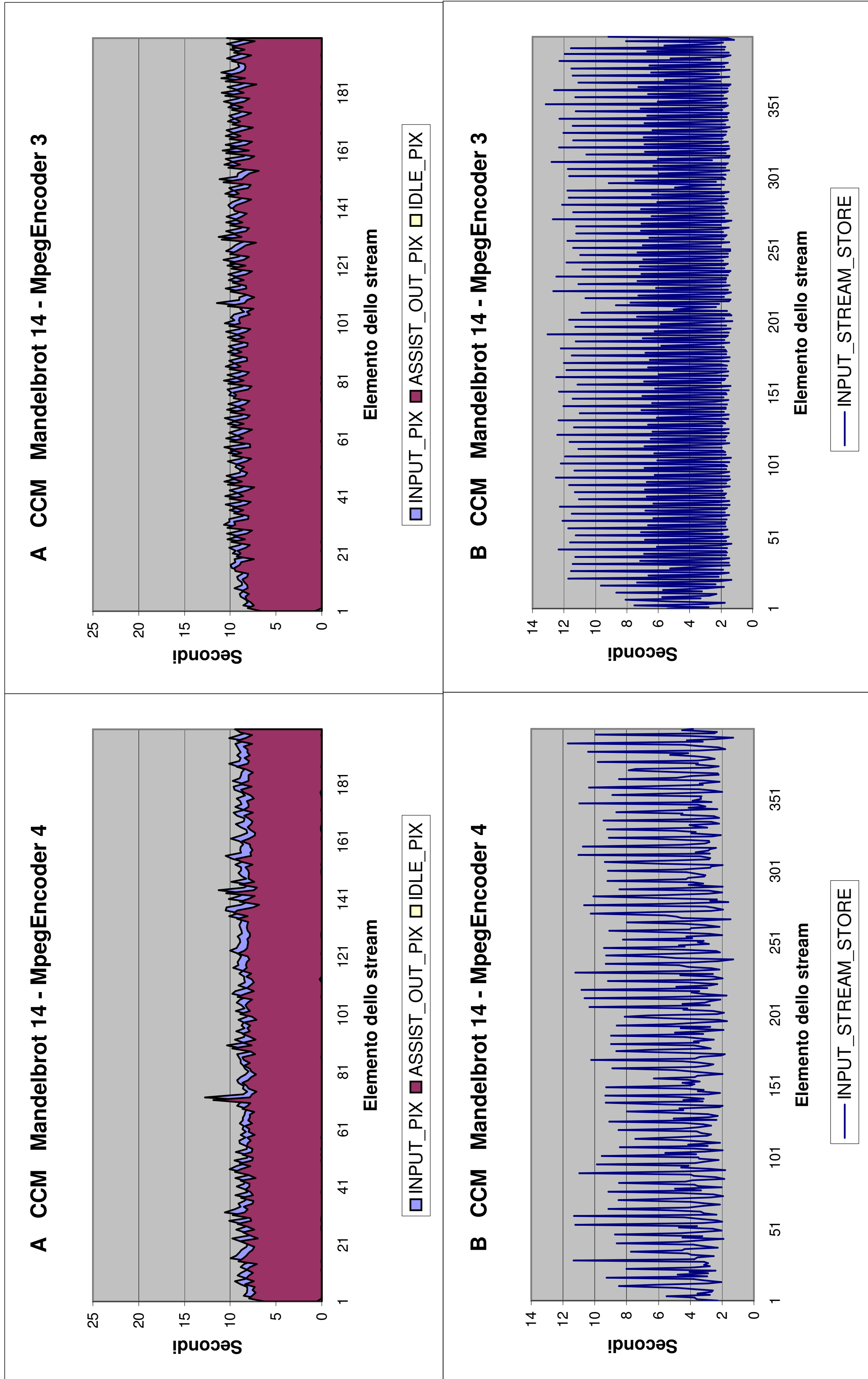

A CCM Mandelbrot 12 - MpegEncoder 5
Secondi
0 5 10 15 20 25
1 21 41 61 81 101 121 141 161 181
Elemento dello stream
INPUT_PIX ASSIST_OUT_PIX IDLE_PIX

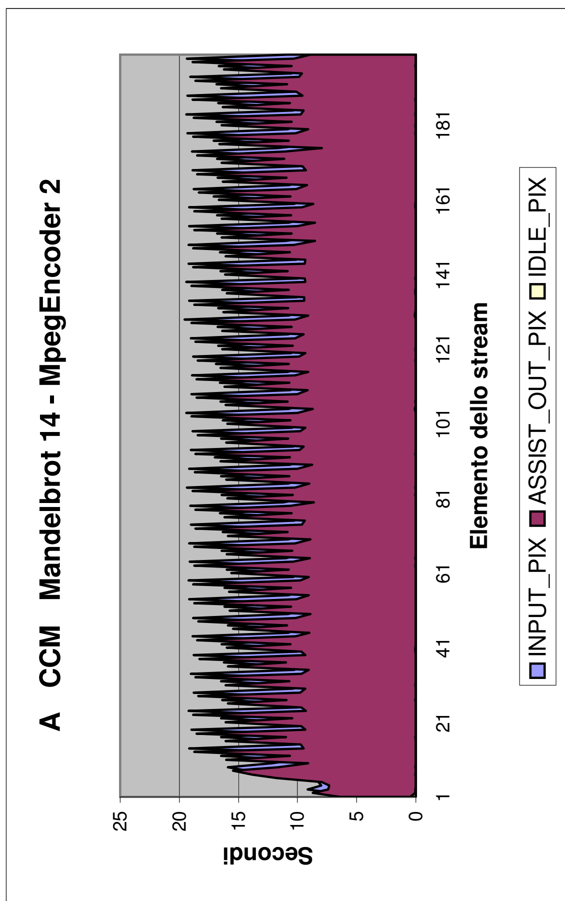

B CCM Mandelbrot 12 - MpegEncoder 2
Secondi
0 2 4 6 8 10 12 14
1 51 101 151 201 251 301 351
Elemento dello stream
INPUT_STREAM_STORE

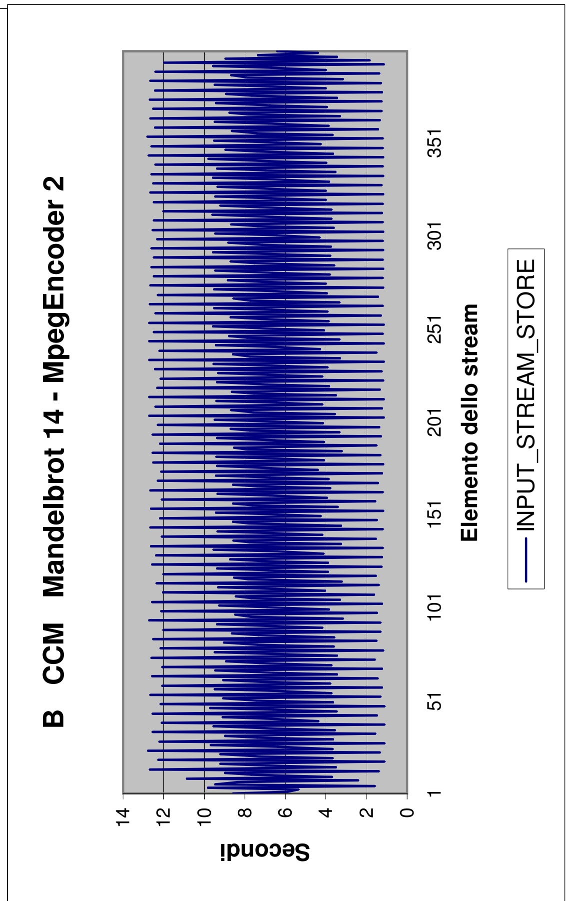

B CCM Mandelbrot 12 - MpegEncoder 5
Secondi
0 2 4 6 8 10 12 14
1 51 101 151 201 251 301 351
Elemento dello stream
INPUT_STREAM_STORE

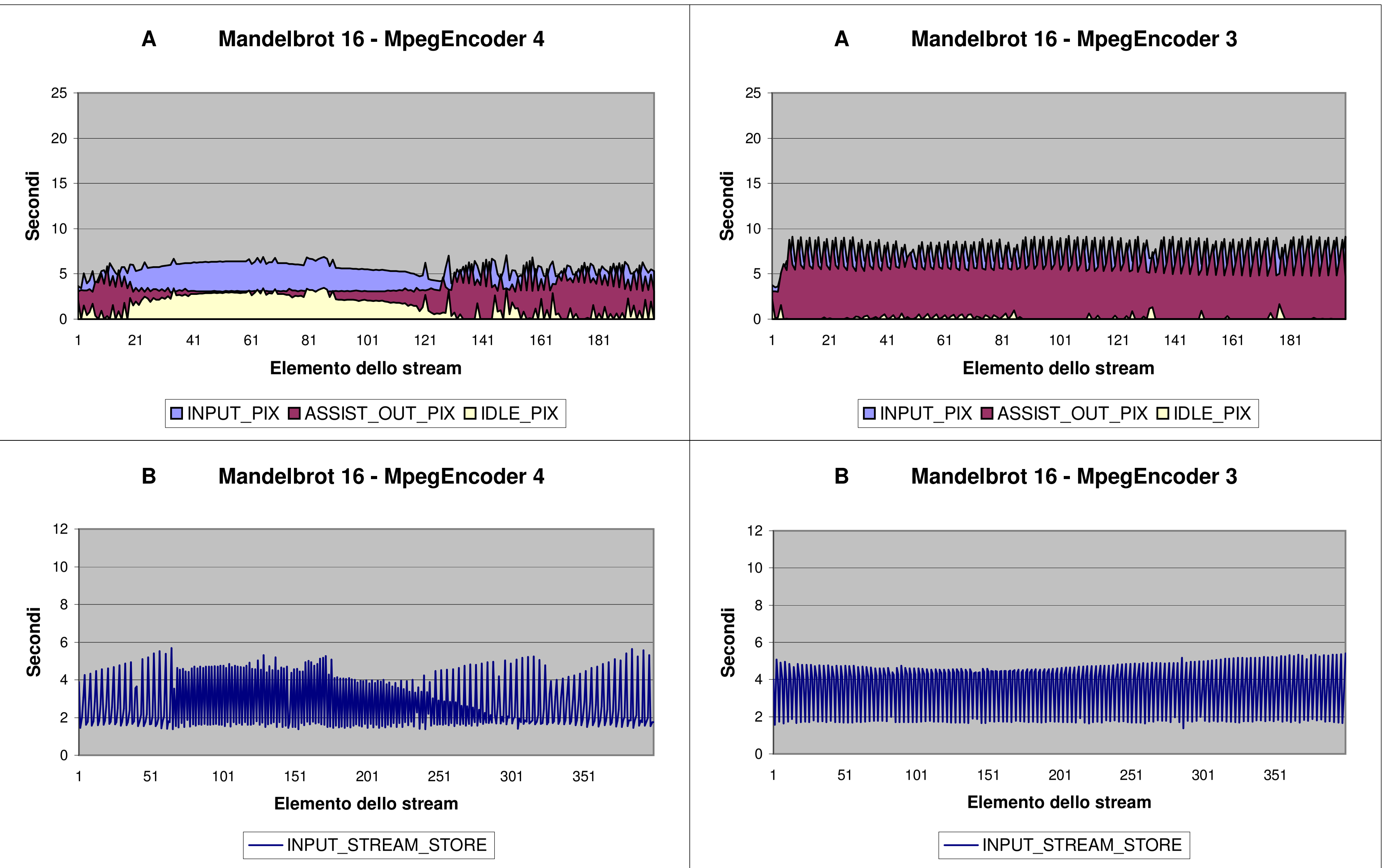

A
Mandelbrot 14 - MpegEncoder 4
Secondi
Elemento dello stream
INPUT_PIX
ASSIST_OUT_PIX
IDLE_PIX
A
Mandelbrot 14 - MpegEncoder 3
Secondi
Elemento dello stream
INPUT_PIX
ASSIST_OUT_PIX
IDLE_PIX
B
Mandelbrot 14 - MpegEncoder 4
Secondi
Elemento dello stream
INPUT_STREAM_STORE
B
Mandelbrot 14 - MpegEncoder 3
Secondi
Elemento dello stream
INPUT_STREAM_STORE

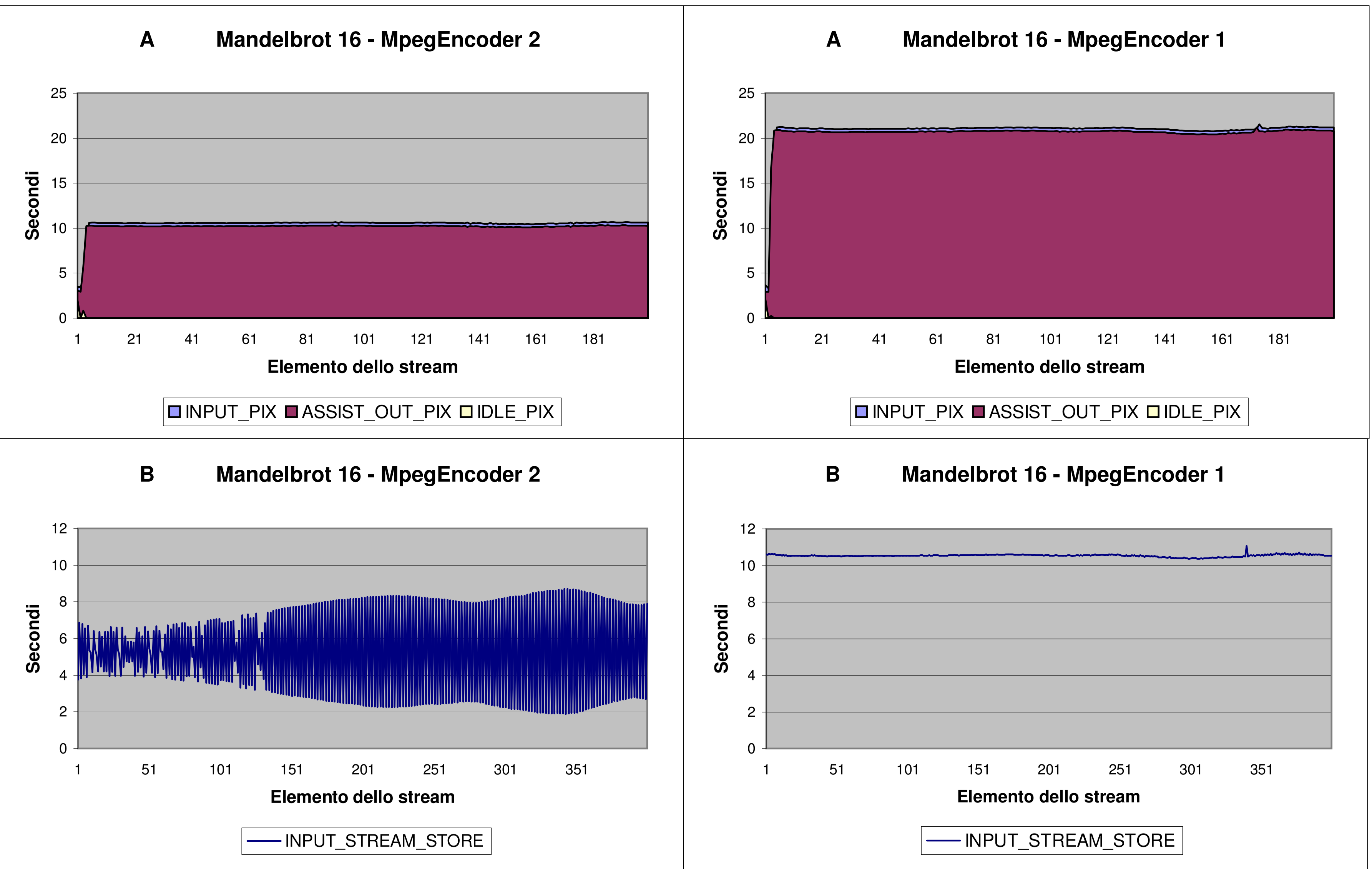

A
Mandelbrot 14 - MpegEncoder 2
Secondi
25
20
15
10
5
0
1
21
41
61
81
101
121
141
161
181
Elemento dello stream
INPUT_PIX
ASSIST_OUT_PIX
IDLE_PIX
A
Mandelbrot 14 - MpegEncoder 1
Secondi
25
20
15
10
5
0
1
21
41
61
81
101
121
141
161
181
Elemento dello stream
INPUT_PIX
ASSIST_OUT_PIX
IDLE_PIX
B
Mandelbrot 14 - MpegEncoder 2
Secondi
14
12
10
8
6
4
2
0
1
51
101
151
201
251
301
351
Elemento dello stream
INPUT_STREAM_STORE
B
Mandelbrot 14 - MpegEncoder 1
Secondi
14
12
10
8
6
4
2
0
1
51
101
151
201
251
301
351
Elemento dello stream
INPUT_STREAM_STORE


A CCM Mandelbrot 14 - MpegEncoder 4
Secondi
Elemento dello stream
INPUT_PIX
ASSIST_OUT_PIX
IDLE_PIX
A CCM Mandelbrot 14 - MpegEncoder 3
Secondi
Elemento dello stream
INPUT_PIX
ASSIST_OUT_PIX
IDLE_PIX
B CCM Mandelbrot 14 - MpegEncoder 4
Secondi
Elemento dello stream
INPUT_STREAM_STORE
B CCM Mandelbrot 14 - MpegEncoder 3
Secondi
Elemento dello stream
INPUT_STREAM_STORE


A CCM Mandelbrot 14 - MpegEncoder 2
Secondi
25
20
15
10
5
0
1
21
41
61
81
101
121
141
161
181
Elemento dello stream
INPUT_PIX
ASSIST_OUT_PIX
IDLE_PIX

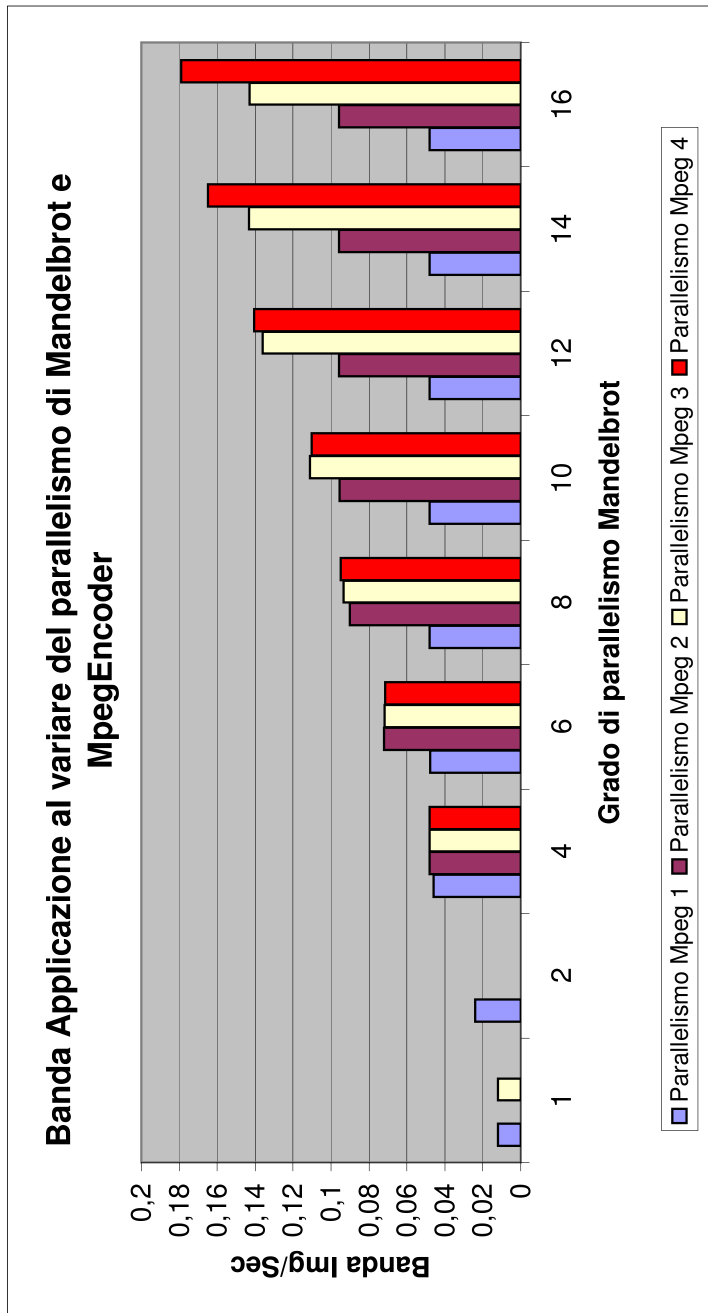

B CCM Mandelbrot 14 - MpegEncoder 2
Secondi
14
12
10
8
6
4
2
0
1
51
101
151
201
251
301
351
Elemento dello stream
INPUT_STREAM_STORE


A Mandelbrot 16 - MpegEncoder 4
Secondi
Elemento dello stream
INPUT_PIX
ASSIST_OUT_PIX
IDLE_PIX
A Mandelbrot 16 - MpegEncoder 3
Secondi
Elemento dello stream
INPUT_PIX
ASSIST_OUT_PIX
IDLE_PIX
B Mandelbrot 16 - MpegEncoder 4
Secondi
Elemento dello stream
INPUT_STREAM_STORE
B Mandelbrot 16 - MpegEncoder 3
Secondi
Elemento dello stream
INPUT_STREAM_STORE

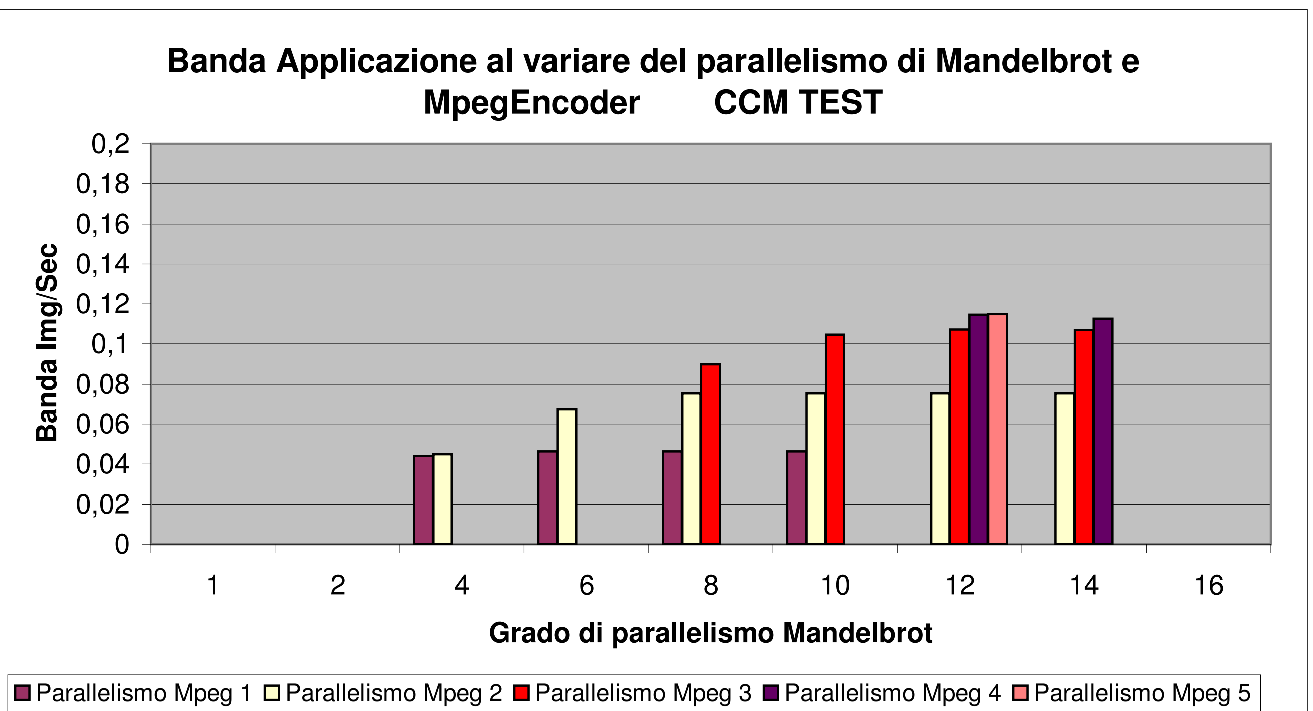


A Mandelbrot 16 - MpegEncoder 2
Secondi
25
20
15
10
5
0
1
21
41
61
81
101
121
141
161
181
Elemento dello stream
INPUT_PIX
ASSIST_OUT_PIX
IDLE_PIX
A Mandelbrot 16 - MpegEncoder 1
Secondi
25
20
15
10
5
0
1
21
41
61
81
101
121
141
161
181
Elemento dello stream
INPUT_PIX
ASSIST_OUT_PIX
IDLE_PIX
B Mandelbrot 16 - MpegEncoder 2
Secondi
12
10
8
6
4
2
0
1
51
101
151
201
251
301
351
Elemento dello stream
INPUT_STREAM_STORE
B Mandelbrot 16 - MpegEncoder 1
Secondi
12
10
8
6
4
2
0
1
51
101
151
201
251
301
351
Elemento dello stream
INPUT_STREAM_STORE

| | 1 | 2 | 4 | 6 | 8 | 10 | 12 | 14 | 16 |
|---|---|---|---|---|---|---|---|---|---|
| Parallelismo Mpeg 1 | 0,011893 | 0,023963 | 0,045934 | 0,047611 | 0,048081 | 0,048081 | 0,048012 | 0,048104 | 0,048089 |
| Parallelismo Mpeg 2 | | | 0,04806 | 0,072138 | 0,089948 | 0,095375 | 0,09566 | 0,095784 | 0,095724 |
| Parallelismo Mpeg 3 | 0,011935 | | 0,048049 | 0,071626 | 0,093351 | 0,111102 | 0,135934 | 0,143236 | 0,143004 |
| Parallelismo Mpeg 4 | | | 0,048058 | 0,071419 | 0,094817 | 0,110261 | 0,140391 | 0,164836 | 0,179126 |

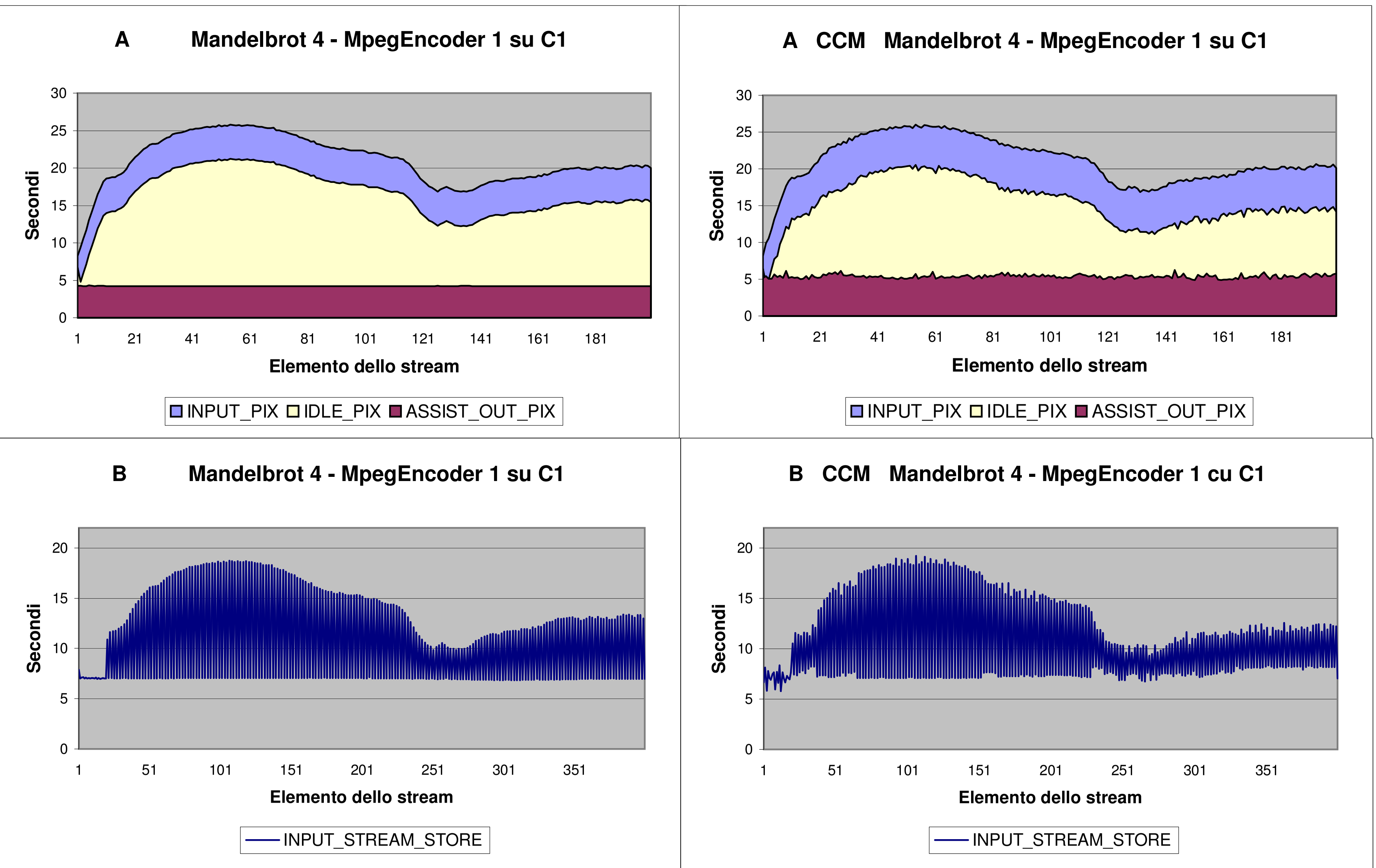


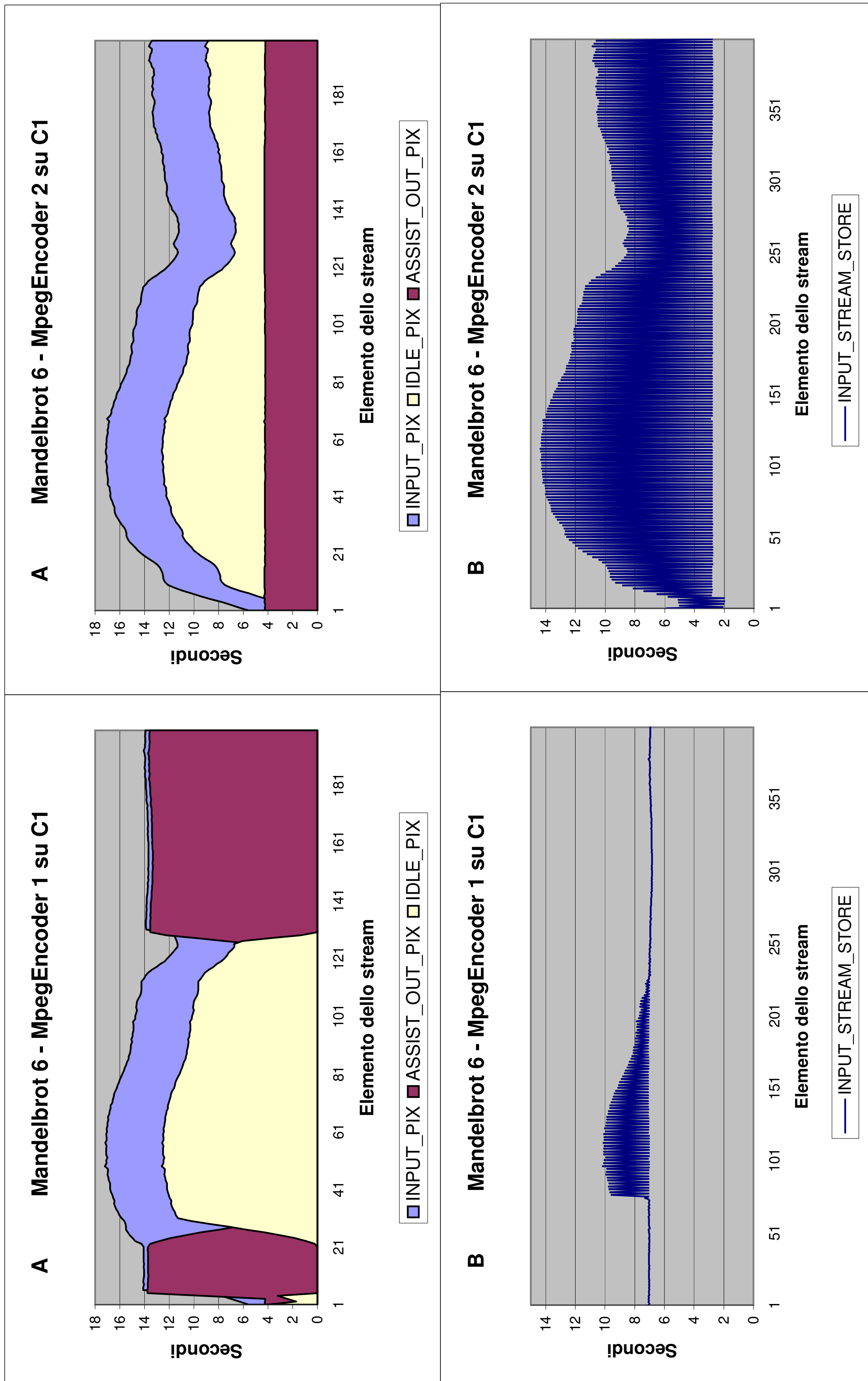
| CCM | 1 | 2 | 4 | 6 | 8 | 10 | 12 | 14 | 16 |
|---|---|---|---|---|---|---|---|---|---|
| Parallelismo Mpeg 1 | | | 0,044187 | 0,046364 | 0,04647 | 0,046423 | | | |
| Parallelismo Mpeg 2 | | | 0,045051 | 0,06755 | 0,075377 | 0,075348 | 0,07538 | 0,075284 | |
| Parallelismo Mpeg 3 | | | | | 0,089997 | 0,104673 | 0,107169 | 0,107033 | |
| Parallelismo Mpeg 4 | | | | | | | 0,114573 | 0,112769 | |
| Parallelismo Mpeg 5 | | | | | | | 0,114931 | | |

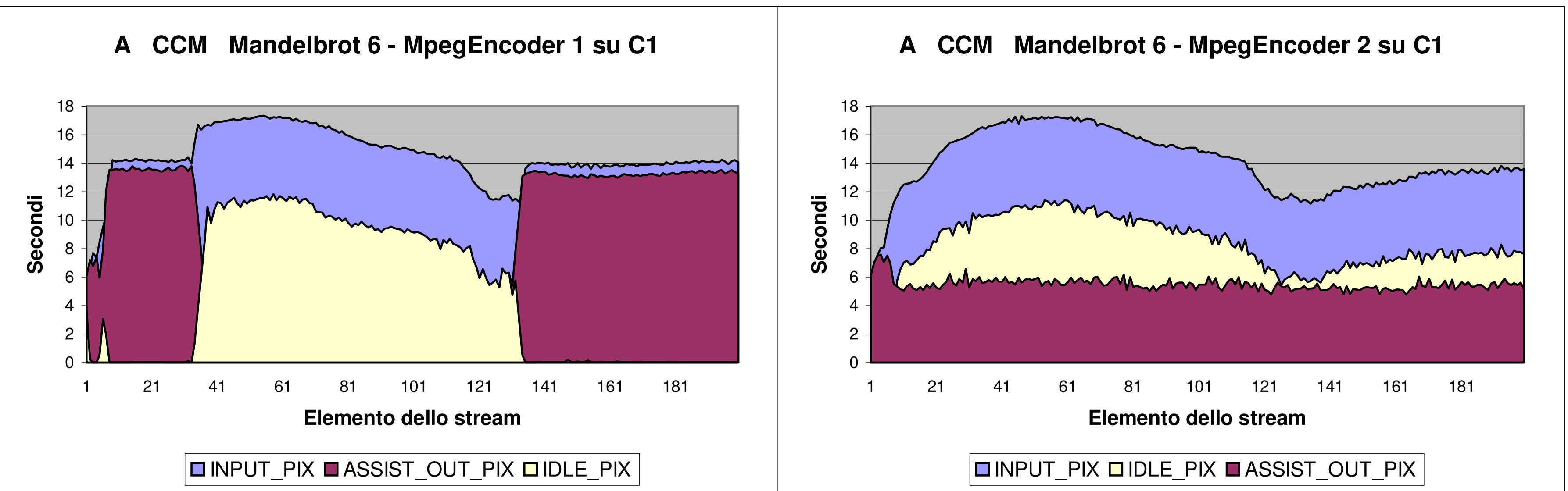


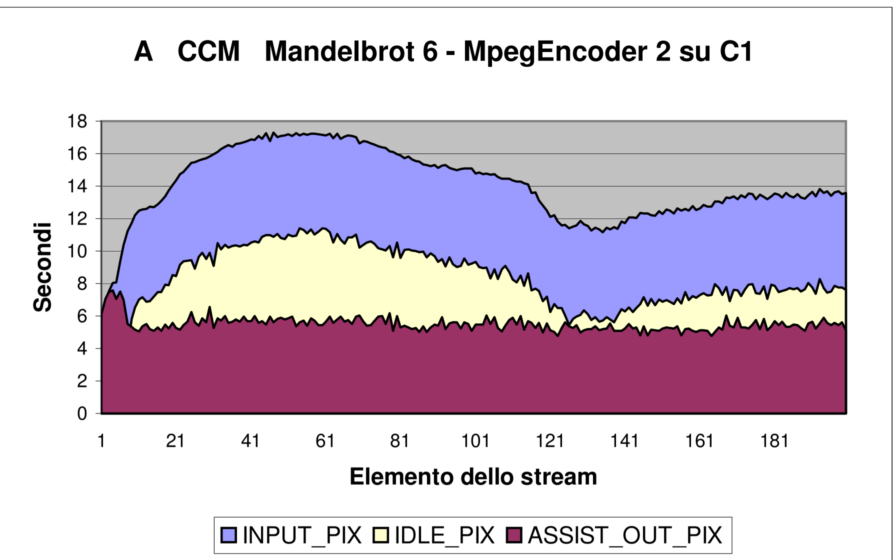
## B.2 Prove sui cluster Pianosa e C1

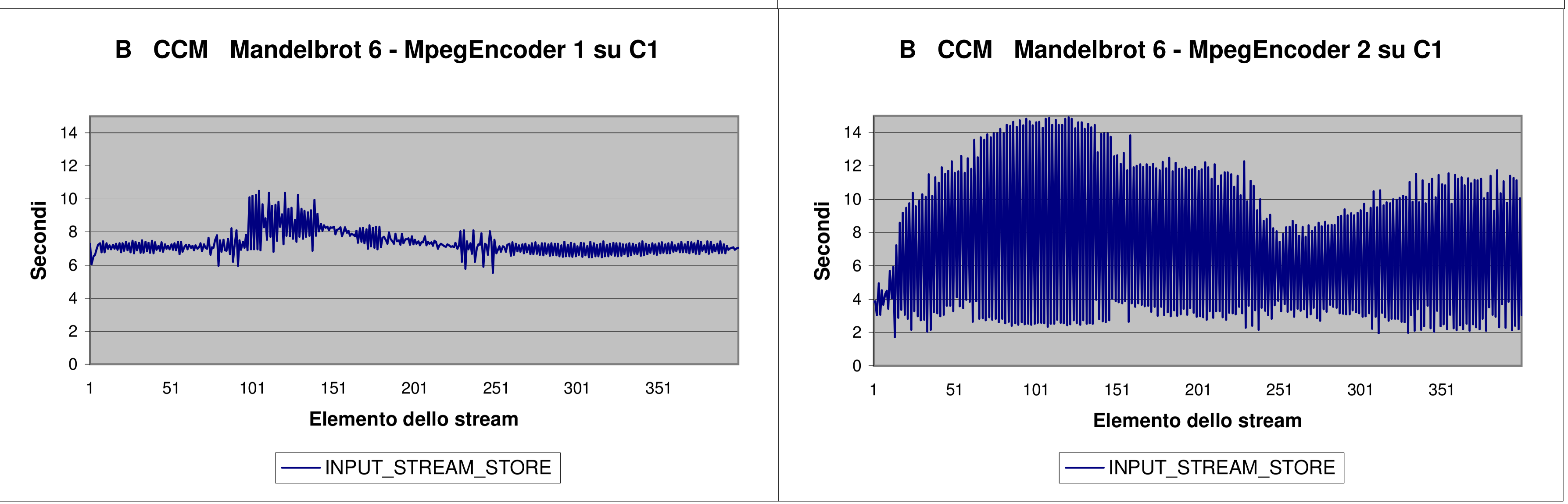
| Pagina | Collo di bottiglia Mandelbrot | Collo di bottiglia MpegEncoder | Dinamicità Applicazione | Applicazione Politica Adattiva |
|---|---|---|---|---|
| 217 | * | | | |
| 218 | * | | * | |
| 219 | * | | * | |
| 220 | * | | * | |
| 221 | | * | * | |
| 222 | | | * | * |
| 223 | | | | * |
| 224 | * | * | | |
| 225 | * | | | |
| 226 | | | * | |
| 227 | | | * | |
| 228 | | | * | |
| 229 | | * | * | |
| 230 | | * | | |
| 231 | | | * | |
| 232 | | * | * | |
| 233 | | * | | |
| 234 | | * | * | |
| 235 | | * | * | |
| 236 | | * | | |

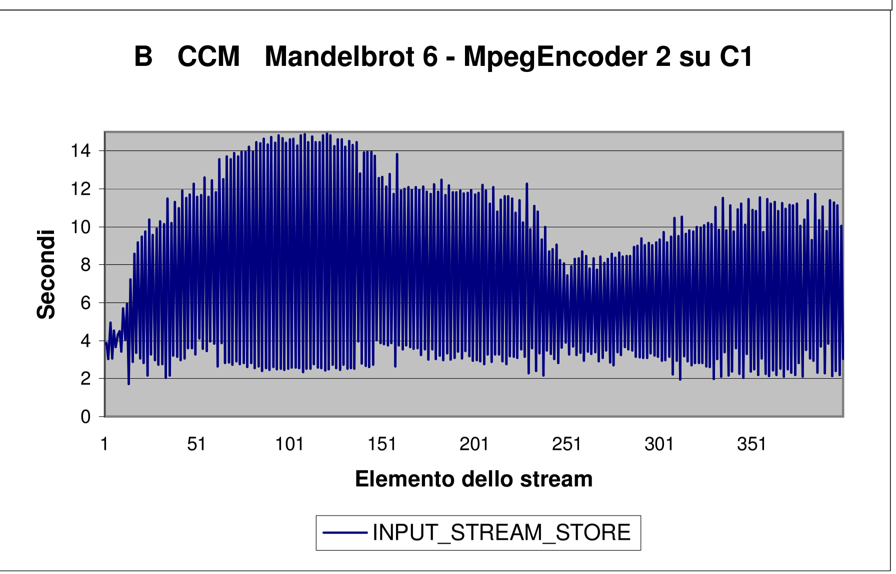

A Mandelbrot 4 - MpegEncoder 1 su C1
Secondi
Elemento dello stream
INPUT_PIX
IDLE_PIX
ASSIST_OUT_PIX
A CCM Mandelbrot 4 - MpegEncoder 1 su C1
Secondi
Elemento dello stream
INPUT_PIX
IDLE_PIX
ASSIST_OUT_PIX
B Mandelbrot 4 - MpegEncoder 1 su C1
Secondi
Elemento dello stream
INPUT_STREAM_STORE
B CCM Mandelbrot 4 - MpegEncoder 1 cu C1
Secondi
Elemento dello stream
INPUT_STREAM_STORE

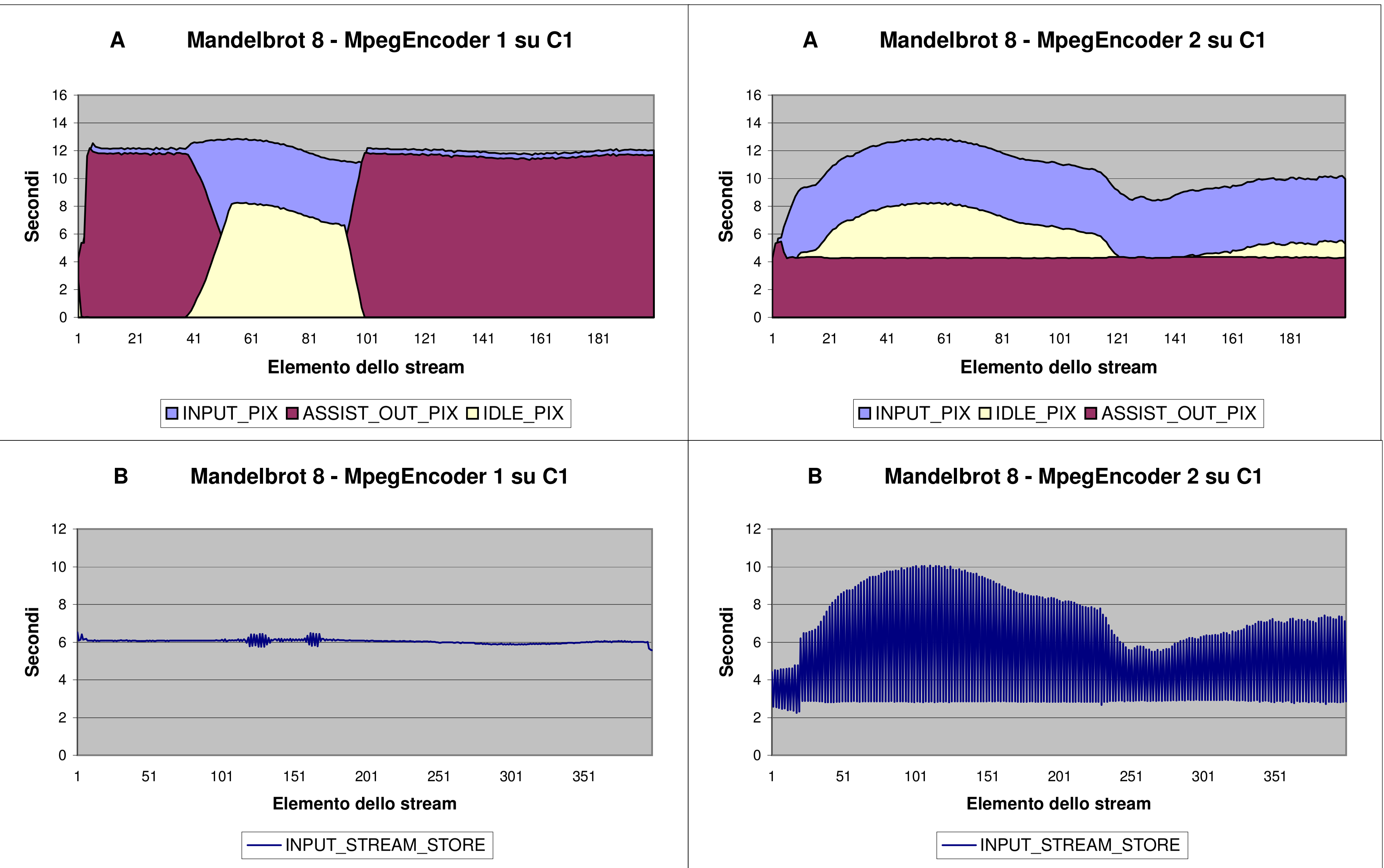

A Mandelbrot 6 - MpegEncoder 1 su C1
Secondi
Elemento dello stream
INPUT_PIX ASSIST_OUT_PIX IDLE_PIX
A Mandelbrot 6 - MpegEncoder 2 su C1
Secondi
Elemento dello stream
INPUT_PIX IDLE_PIX ASSIST_OUT_PIX
B Mandelbrot 6 - MpegEncoder 1 su C1
Seconi
Elemento dello stream
INPUT_STREAM_STORE
B Mandelbrot 6 - MpegEncoder 2 su C1
Secondi
Elemento dello stream
INPUT_STREAM_STORE


A CCM Mandelbrot 6 - MpegEncoder 1 su C1
Secondi
Elemento dello stream
INPUT_PIX ASSIST_OUT_PIX IDLE_PIX

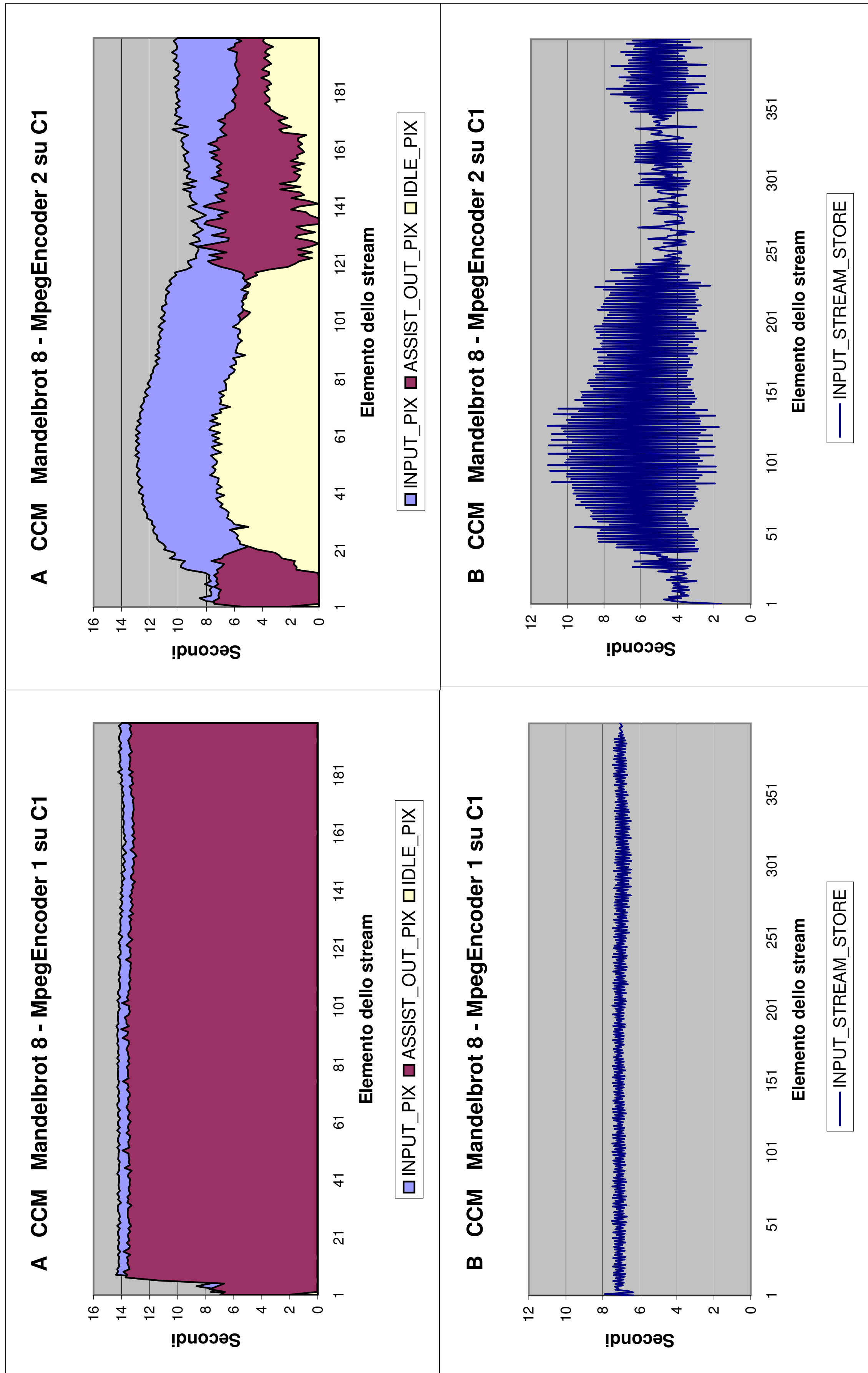

A CCM Mandelbrot 6 - MpegEncoder 2 su C1
Secondi
Elemento dello stream
INPUT_PIX IDLE_PIX ASSIST_OUT_PIX

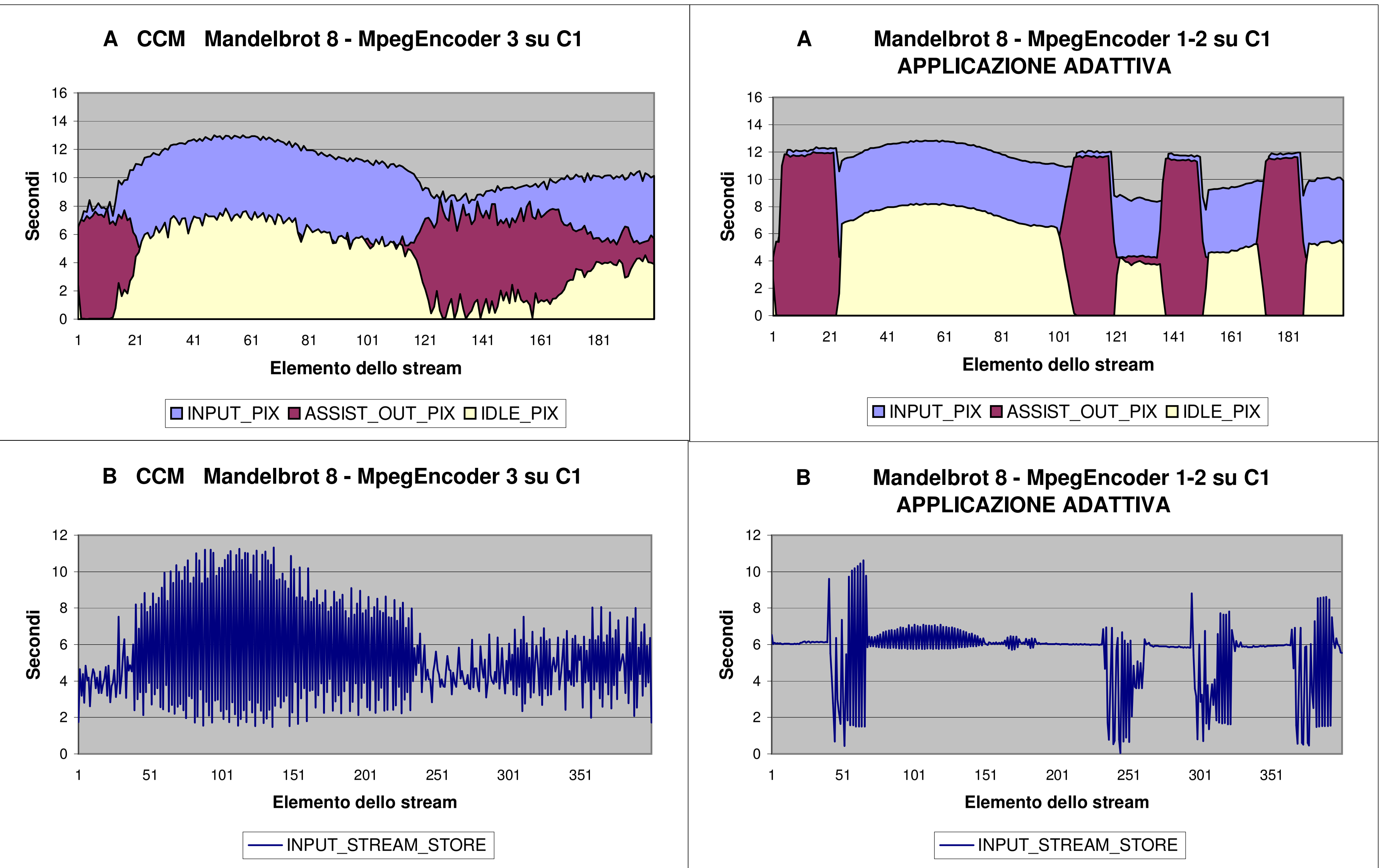

B CCM Mandelbrot 6 - MpegEncoder 1 su C1
Secondi
Elemento dello stream
INPUT_STREAM_STORE

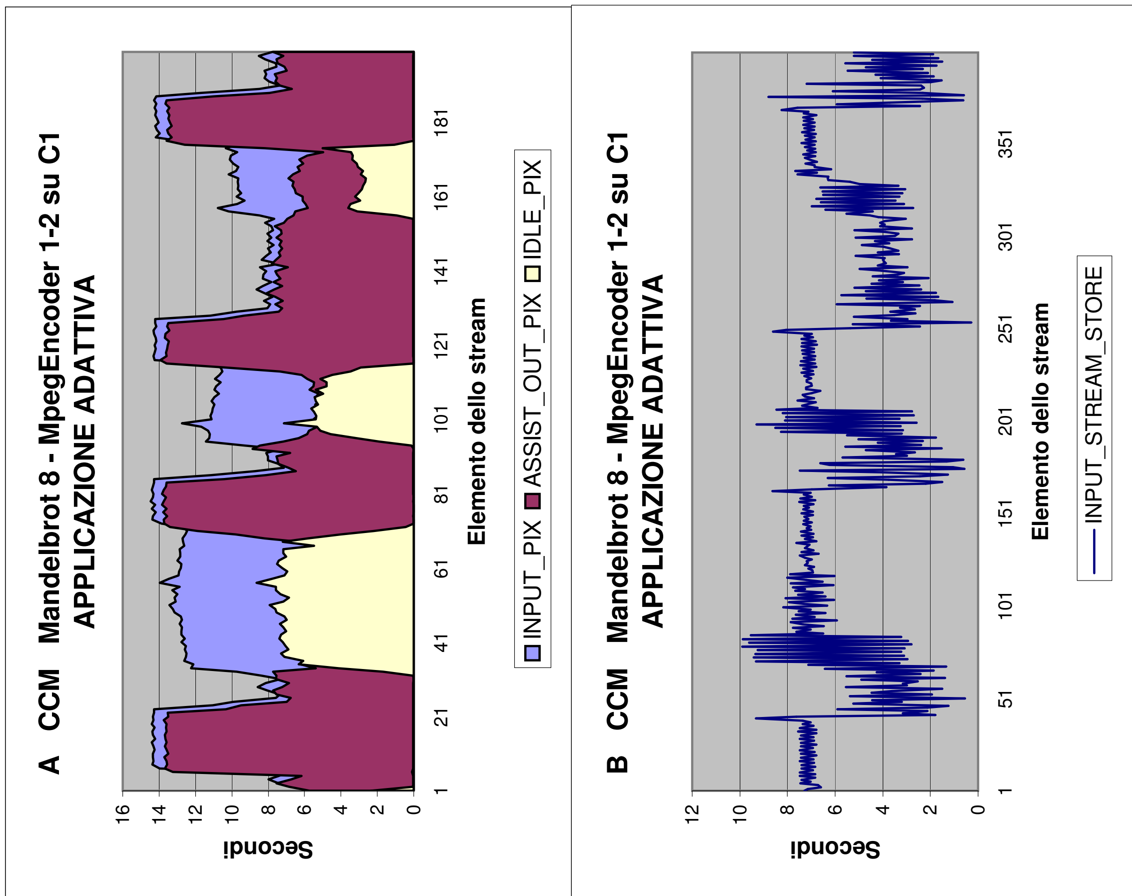

B CCM Mandelbrot 6 - MpegEncoder 2 su C1
Secondi
Elemento dello stream
INPUT_STREAM_STORE

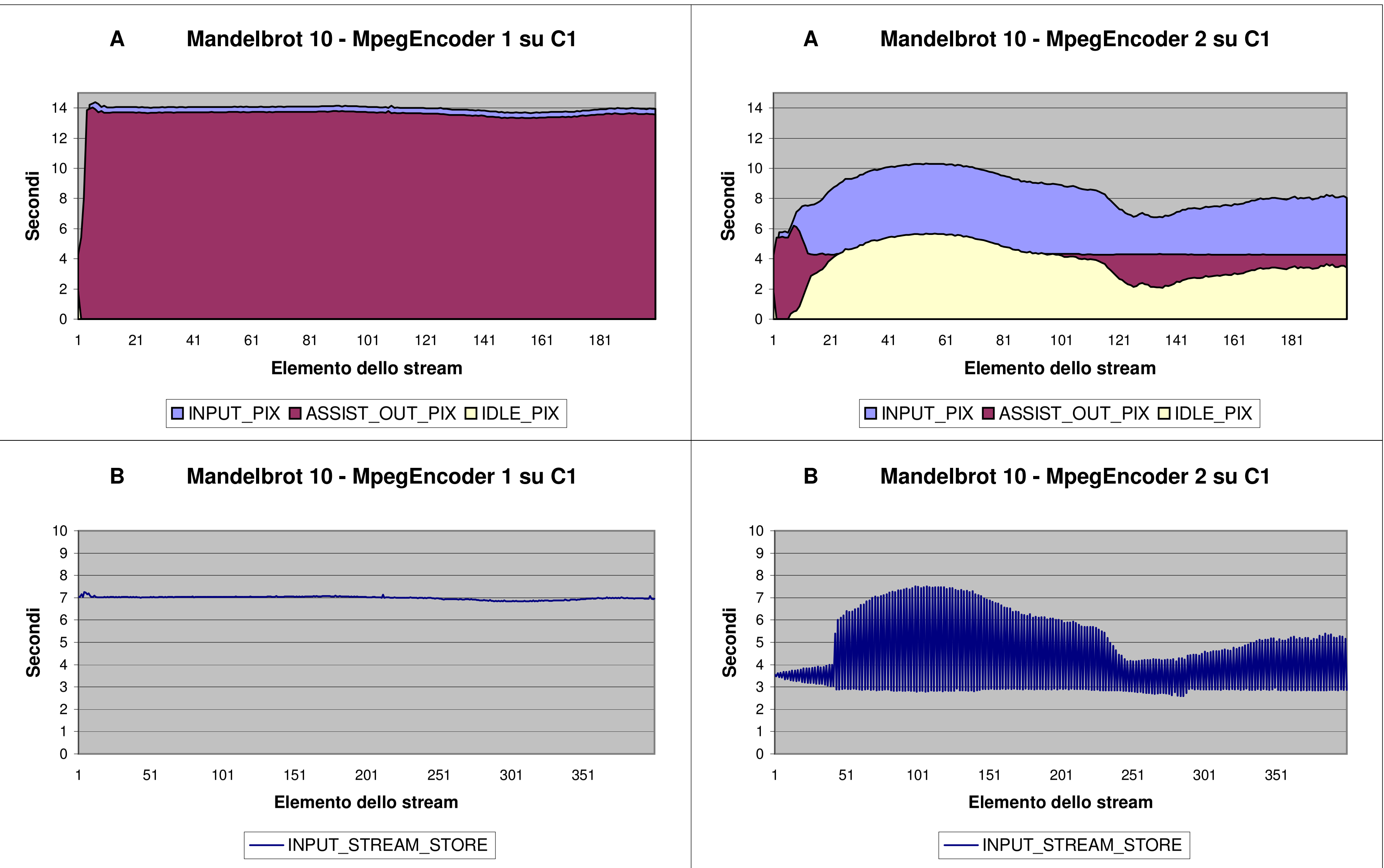


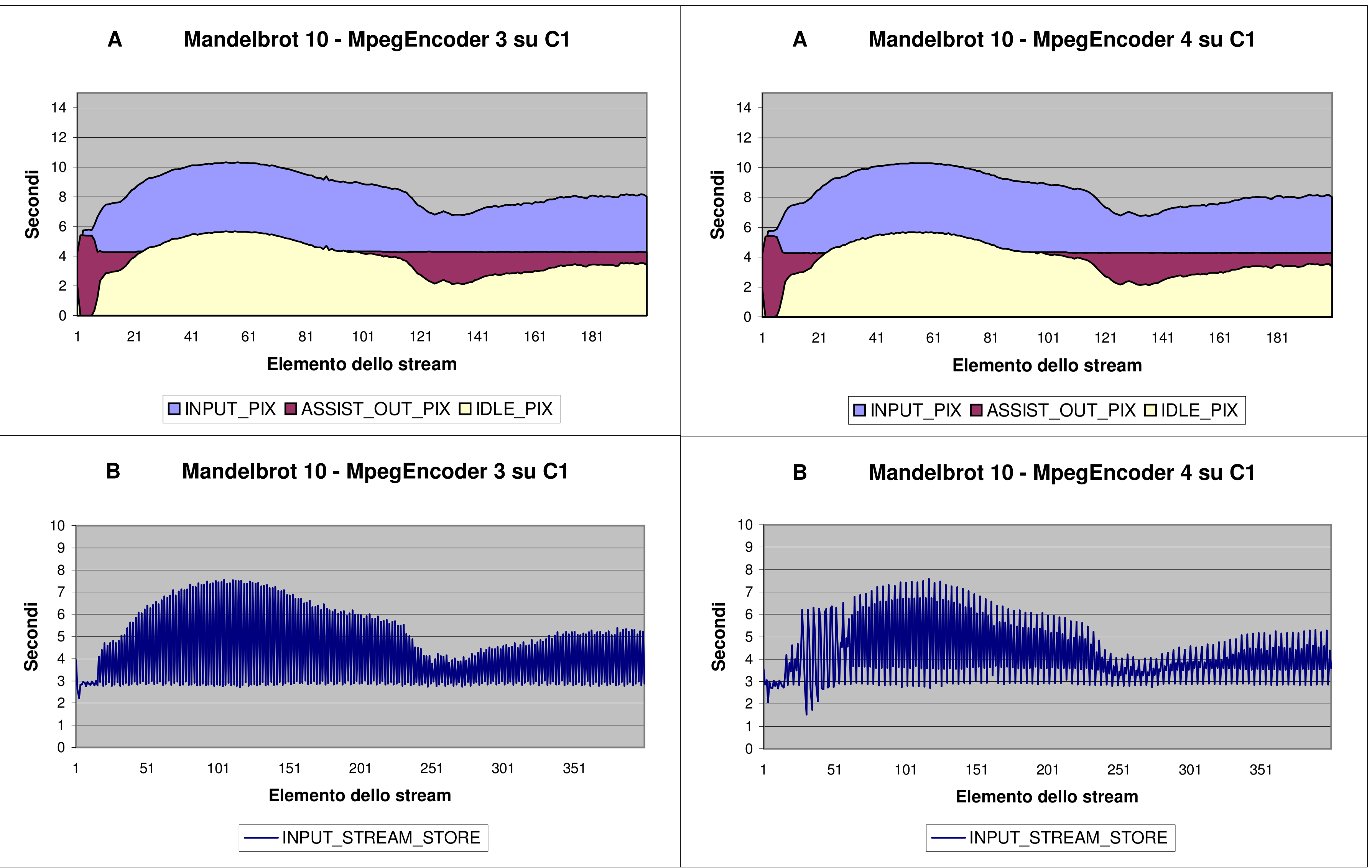
A Mandelbrot 8 - MpegEncoder 1 su C1
Secondi
16
14
12
10
8
6
4
2
0
1
21
41
61
81
101
121
141
161
181
Elemento dello stream
INPUT_PIX
ASSIST_OUT_PIX
IDLE_PIX
A Mandelbrot 8 - MpegEncoder 2 su C1
Secondi
16
14
12
10
8
6
4
2
0
1
21
41
61
81
101
121
141
161
181
Elemento dello stream
INPUT_PIX
IDLE_PIX
ASSIST_OUT_PIX
B Mandelbrot 8 - MpegEncoder 1 su C1
Secondi
12
10
8
6
4
2
0
1
51
101
151
201
251
301
351
Elemento dello stream
INPUT_STREAM_STORE
B Mandelbrot 8 - MpegEncoder 2 su C1
Secondi
12
10
8
6
4
2
0
1
51
101
151
201
251
301
351
Elemento dello stream
INPUT_STREAM_STORE

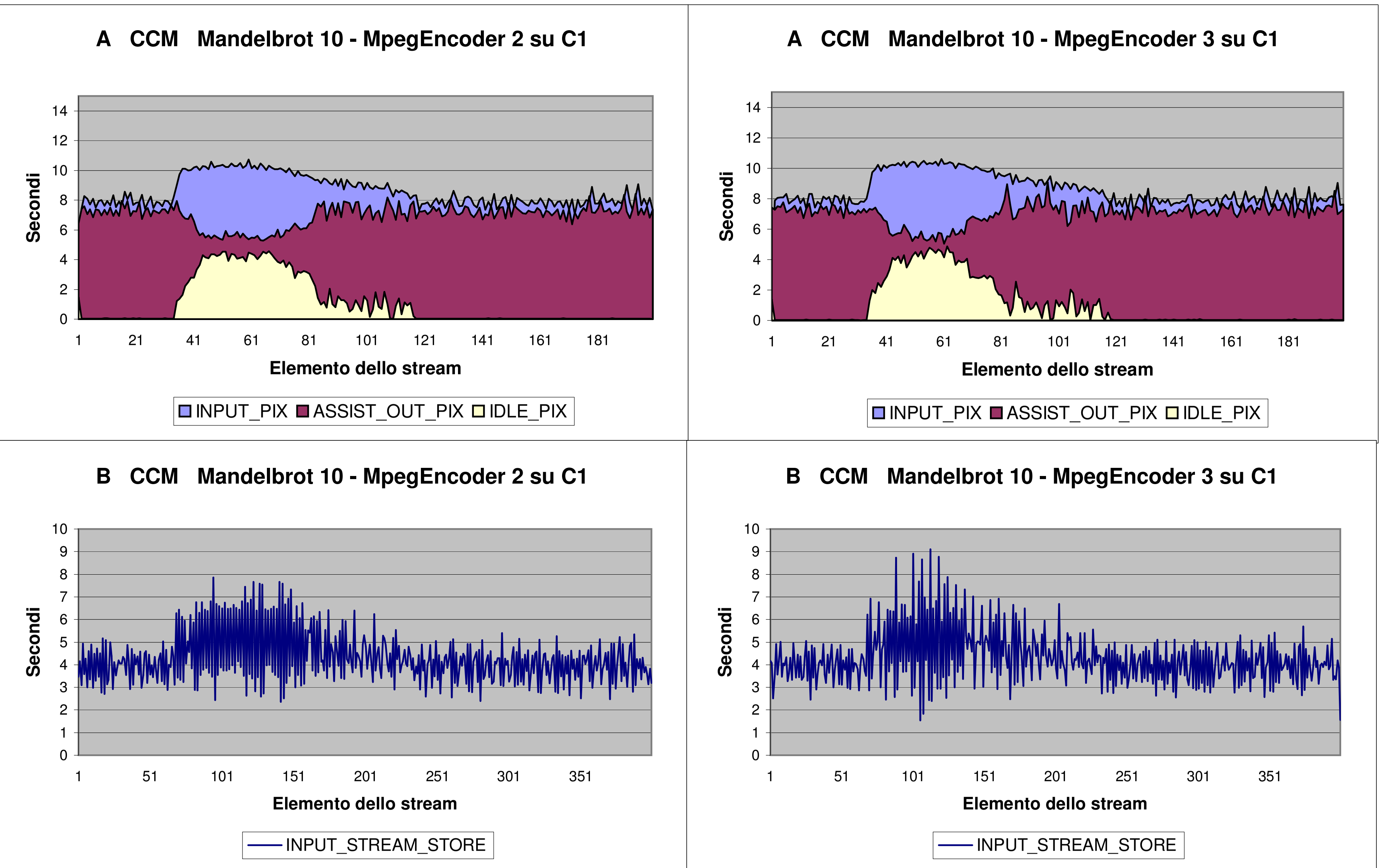

A CCM Mandelbrot 8 - MpegEncoder 1 su C1
Secondi
Elemento dello stream
INPUT_PIX ASSIST_OUT_PIX IDLE_PIX
A CCM Mandelbrot 8 - MpegEncoder 2 su C1
Secondi
Elemento dello stream
INPUT_PIX ASSIST_OUT_PIX IDLE_PIX
B CCM Mandelbrot 8 - MpegEncoder 1 su C1
Secondi
Elemento dello stream
INPUT_STREAM_STORE
B CCM Mandelbrot 8 - MpegEncoder 2 su C1
Secondi
Elemento dello stream
INPUT_STREAM_STORE

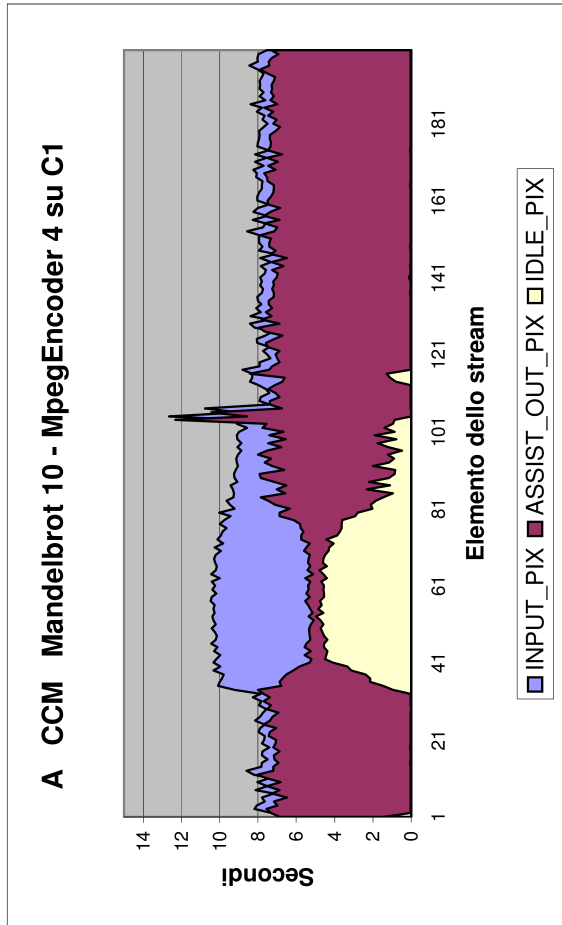

A CCM Mandelbrot 8 - MpegEncoder 3 su C1
Secondi
Elemento dello stream
INPUT_PIX
ASSIST_OUT_PIX
IDLE_PIX
A Mandelbrot 8 - MpegEncoder 1-2 su C1
APPLICAZIONE ADATTIVA
Secondi
Elemento dello stream
INPUT_PIX
ASSIST_OUT_PIX
IDLE_PIX
B CCM Mandelbrot 8 - MpegEncoder 3 su C1
Secondi
Elemento dello stream
INPUT_STREAM_STORE
B Mandelbrot 8 - MpegEncoder 1-2 su C1
APPLICAZIONE ADATTIVA
Secondi
Elemento dello stream
INPUT_STREAM_STORE

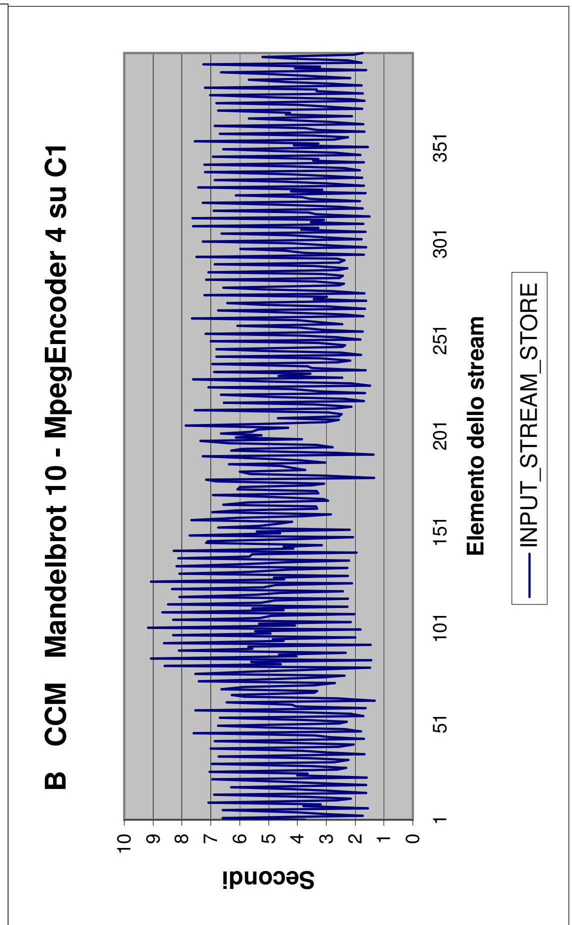

A CCM Mandelbrot 8 - MpegEncoder 1-2 su C1
APPLICAZIONE ADATTIVA
Secondi
Elemento dello stream
INPUT_PIX
ASSIST_OUT_PIX
IDLE_PIX
B CCM Mandelbrot 8 - MpegEncoder 1-2 su C1
APPLICAZIONE ADATTIVA
Secondi
Elemento dello stream
INPUT_STREAM_STORE

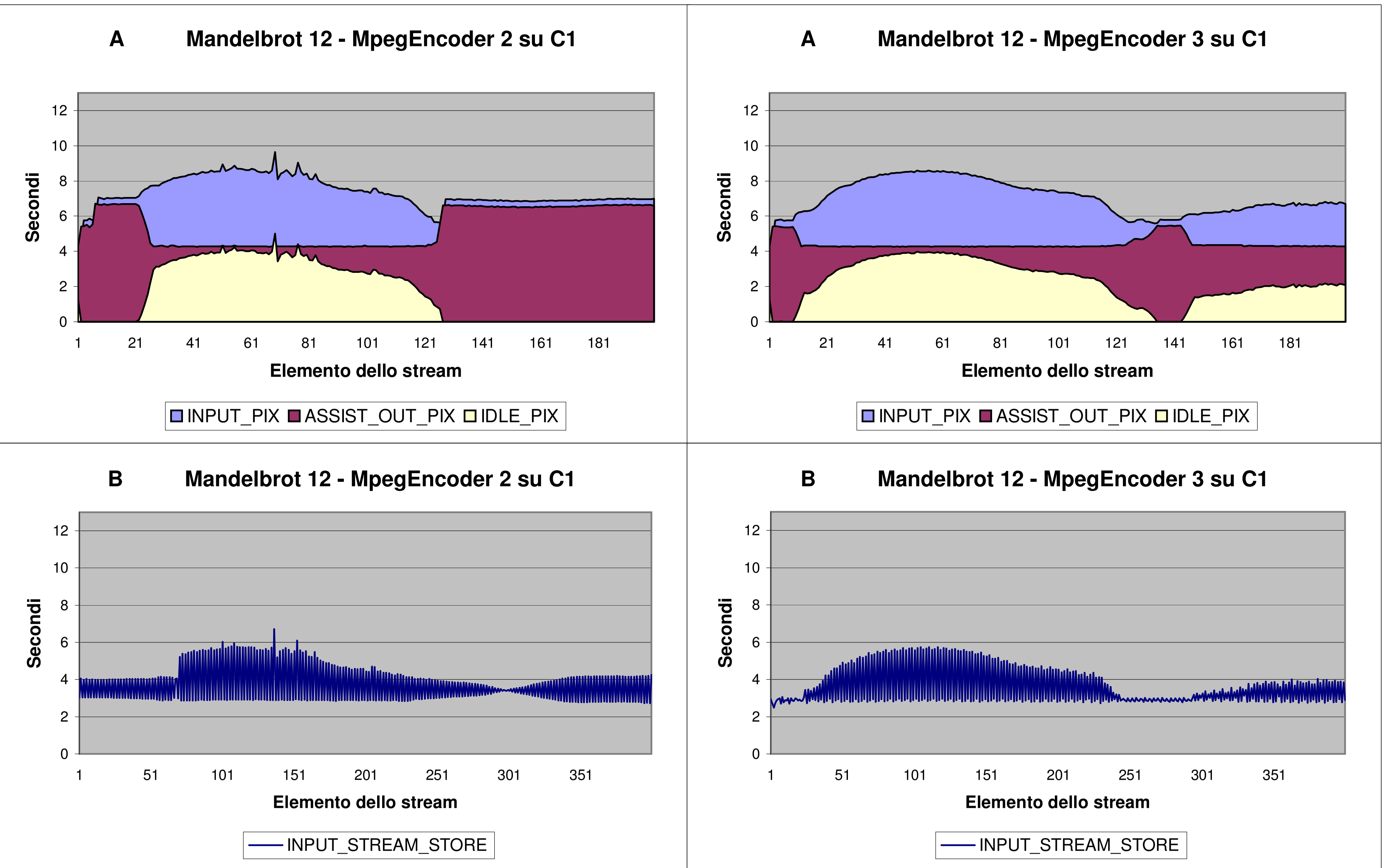

A Mandelbrot 10 - MpegEncoder 1 su C1
Secondi
Elemento dello stream
INPUT_PIX ASSIST_OUT_PIX IDLE_PIX
A Mandelbrot 10 - MpegEncoder 2 su C1
Secondi
Elemento dello stream
INPUT_PIX ASSIST_OUT_PIX IDLE_PIX
B Mandelbrot 10 - MpegEncoder 1 su C1
Secondi
Elemento dello stream
INPUT_STREAM_STORE
B Mandelbrot 10 - MpegEncoder 2 su C1
Secondi
Elemento dello stream
INPUT_STREAM_STORE

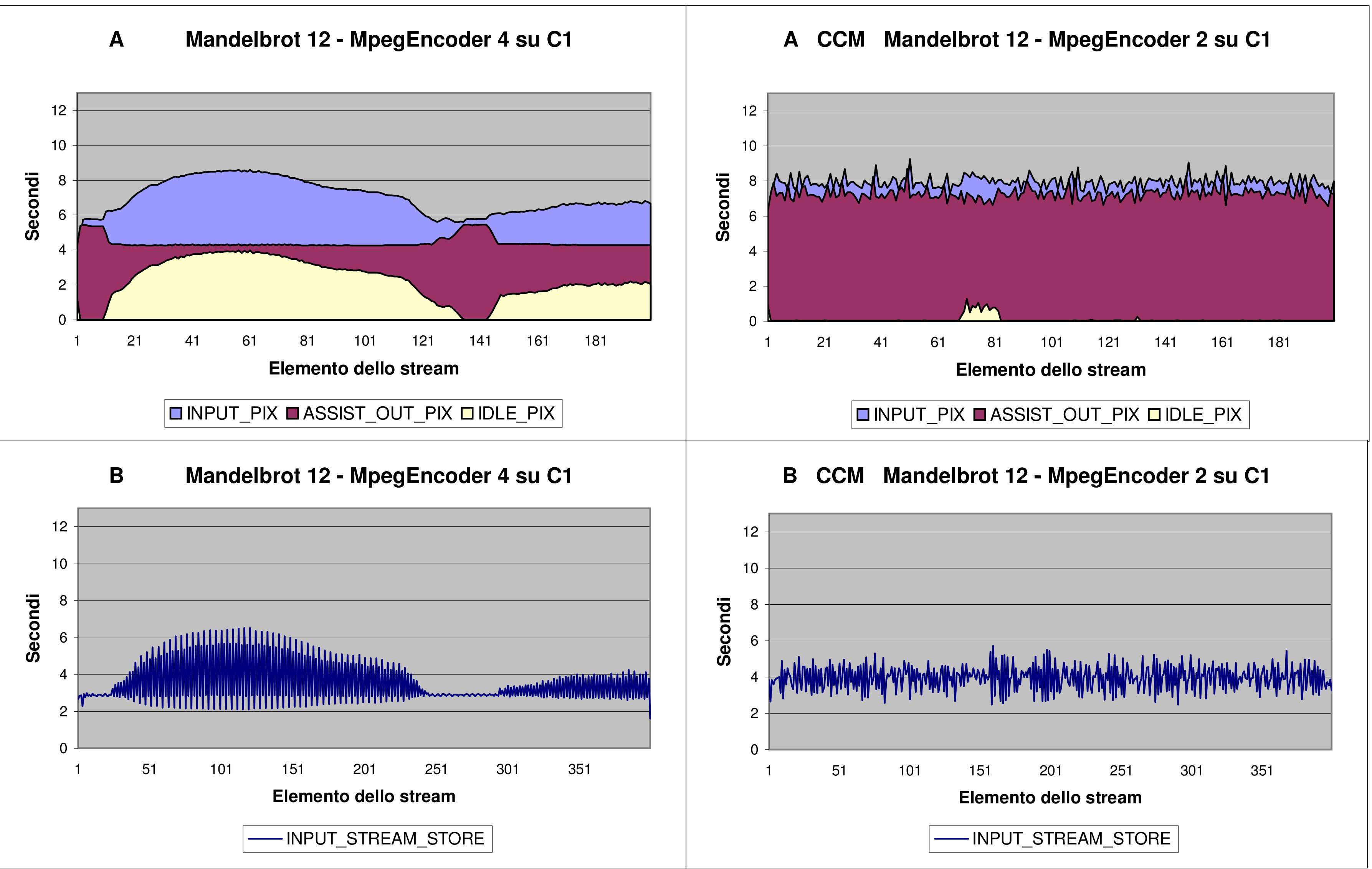

A
Mandelbrot 10 - MpegEncoder 3 su C1
Secondi
Elemento dello stream
INPUT_PIX
ASSIST_OUT_PIX
IDLE_PIX
A
Mandelbrot 10 - MpegEncoder 4 su C1
Secondi
Elemento dello stream
INPUT_PIX
ASSIST_OUT_PIX
IDLE_PIX
B
Mandelbrot 10 - MpegEncoder 3 su C1
Secondi
Elemento dello stream
INPUT_STREAM_STORE
B
Mandelbrot 10 - MpegEncoder 4 su C1
Secondi
Elemento dello stream
INPUT_STREAM_STORE

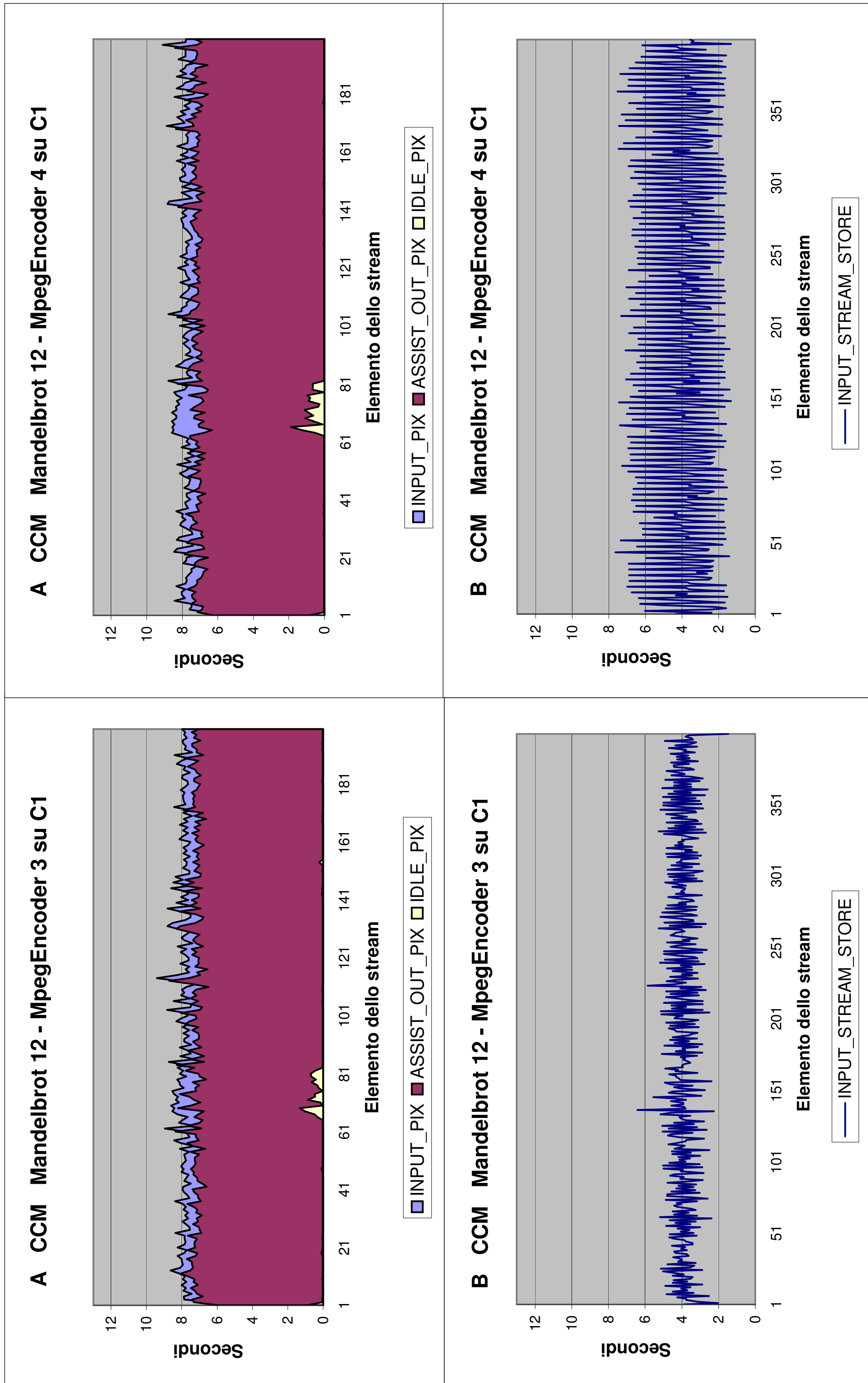

A CCM Mandelbrot 10 - MpegEncoder 2 su C1
Secondi
14 12 10 8 6 4 2 0
1 21 41 61 81 101 121 141 161 181
Elemento dello stream
INPUT_PIX
ASSIST_OUT_PIX
IDLE_PIX
A CCM Mandelbrot 10 - MpegEncoder 3 su C1
Secondi
14 12 10 8 6 4 2 0
1 21 41 61 81 101 121 141 161 181
Elemento dello stream
INPUT_PIX
ASSIST_OUT_PIX
IDLE_PIX
B CCM Mandelbrot 10 - MpegEncoder 2 su C1
Secondi
10 9 8 7 6 5 4 3 2 1 0
1 51 101 151 201 251 301 351
Elemento dello stream
INPUT_STREAM_STORE
B CCM Mandelbrot 10 - MpegEncoder 3 su C1
Secondi
10 9 8 7 6 5 4 3 2 1 0
1 51 101 151 201 251 301 351
Elemento dello stream
INPUT_STREAM_STORE

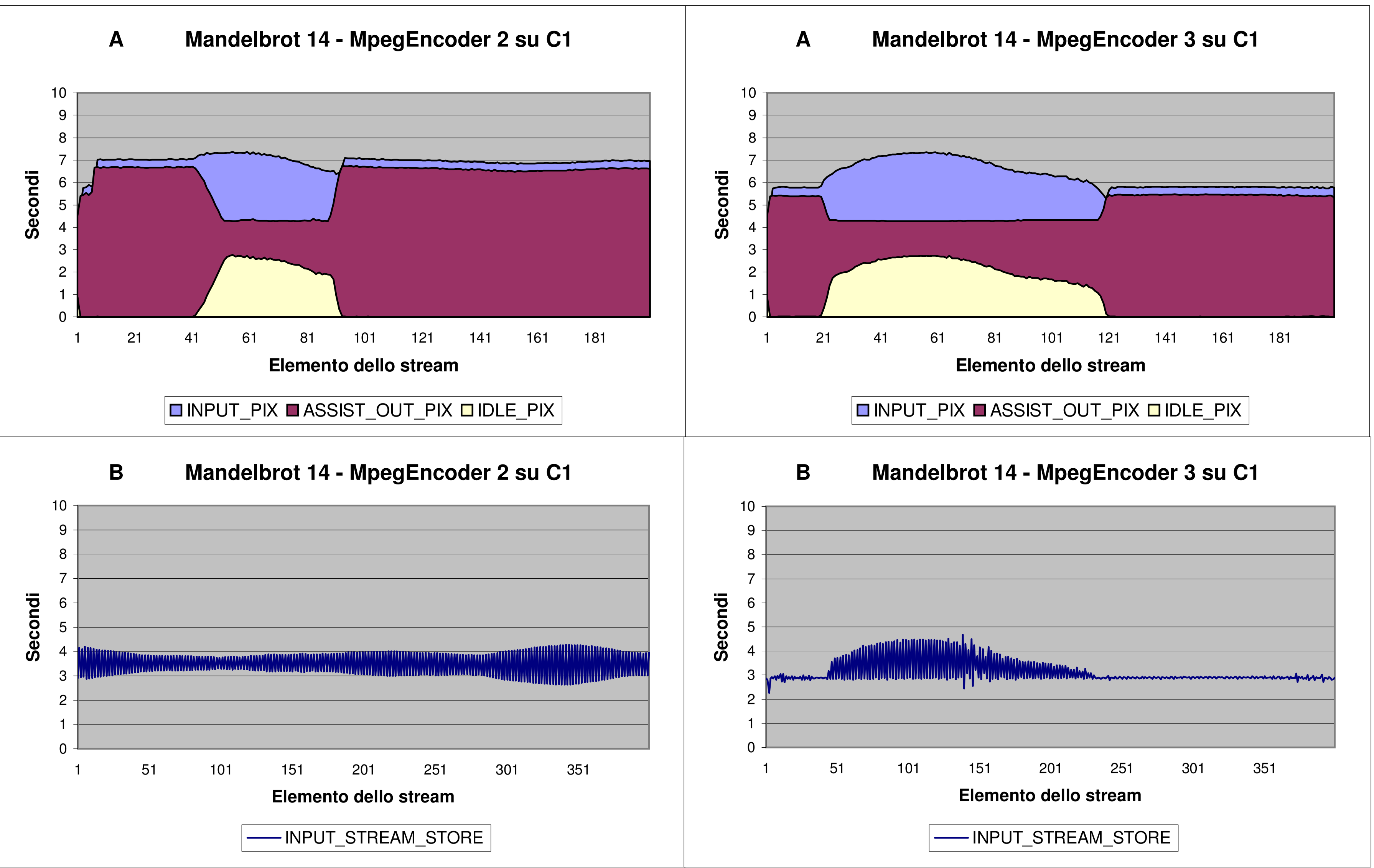

A CCM Mandelbrot 10 - MpegEncoder 4 su C1
Secondi
0
2
4
6
8
10
12
14
1
21
41
61
81
101
121
141
161
181
Elemento dello stream
INPUT_PIX
ASSIST_OUT_PIX
IDLE_PIX

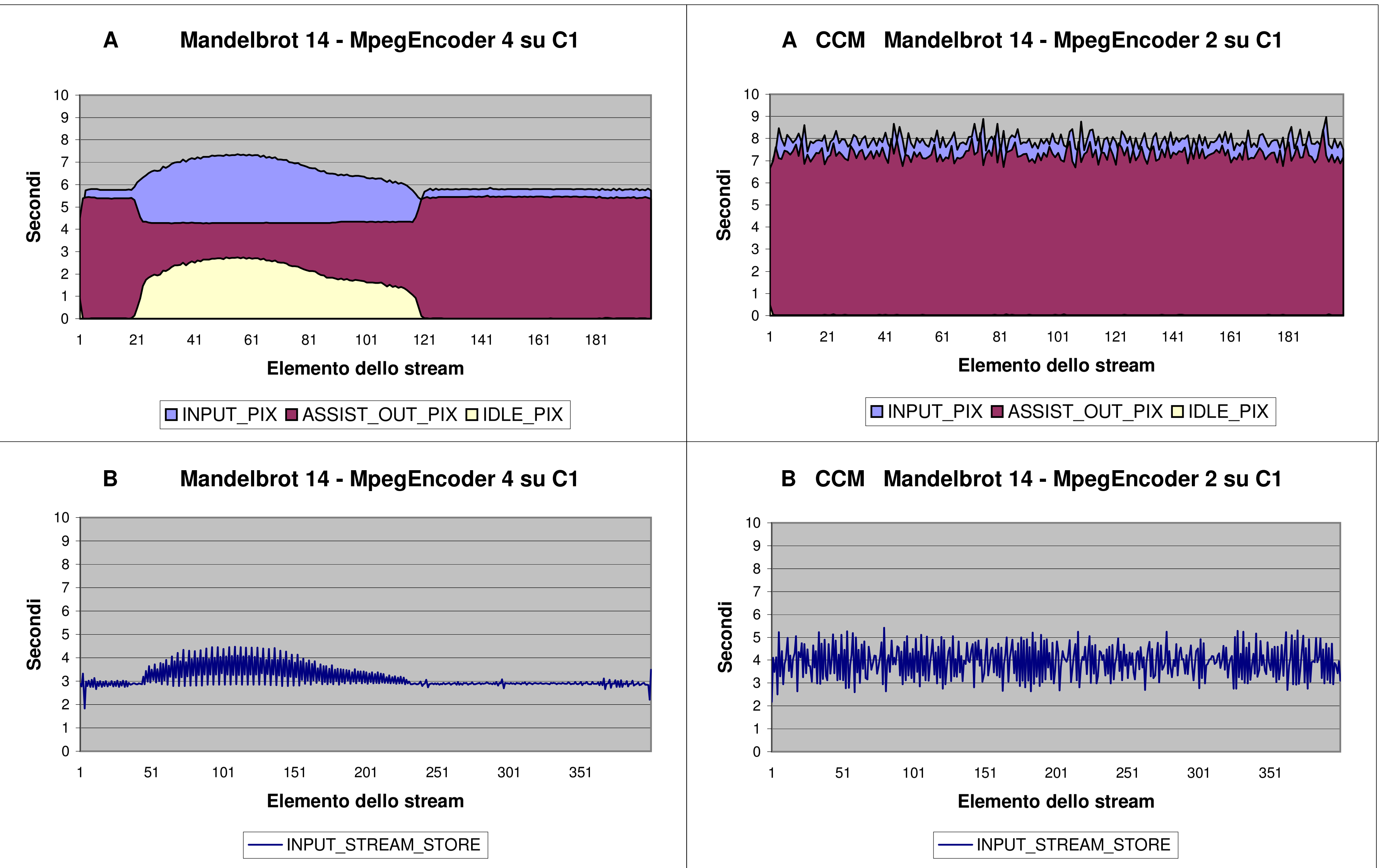

B CCM Mandelbrot 10 - MpegEncoder 4 su C1
Secondi
0
1
2
3
4
5
6
7
8
9
10
1
51
101
151
201
251
301
351
Elemento dello stream
INPUT_STREAM_STORE


A
Mandelbrot 12 - MpegEncoder 2 su C1
Secondi
Elemento dello stream
INPUT_PIX
ASSIST_OUT_PIX
IDLE_PIX
A
Mandelbrot 12 - MpegEncoder 3 su C1
Secondi
Elemento dello stream
INPUT_PIX
ASSIST_OUT_PIX
IDLE_PIX
B
Mandelbrot 12 - MpegEncoder 2 su C1
Secondi
Elemento dello stream
INPUT_STREAM_STORE
B
Mandelbrot 12 - MpegEncoder 3 su C1
Secondi
Elemento dello stream
INPUT_STREAM_STORE

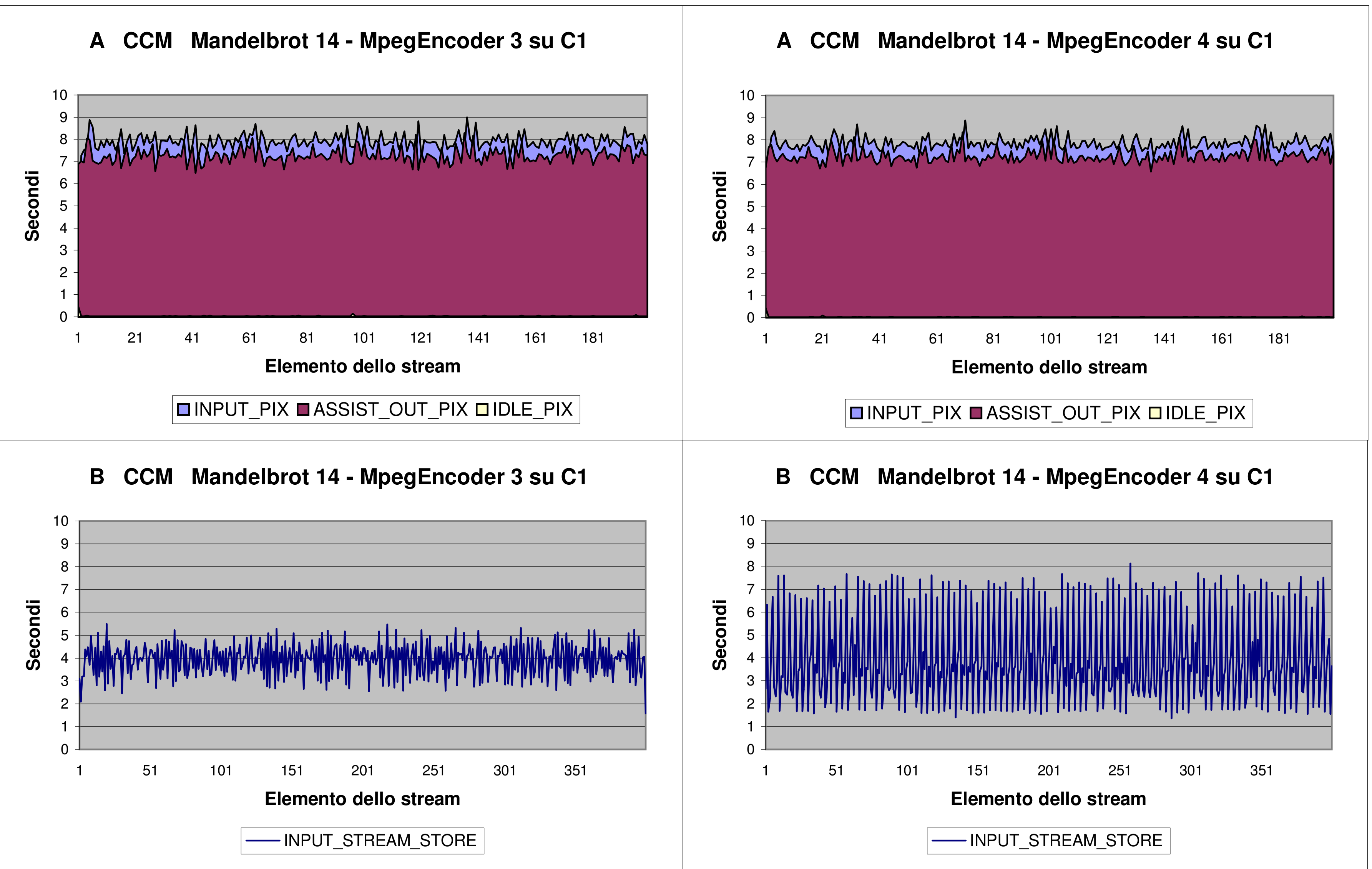

A Mandelbrot 12 - MpegEncoder 4 su C1
Secondi
Elemento dello stream
INPUT_PIX ASSIST_OUT_PIX IDLE_PIX
A CCM Mandelbrot 12 - MpegEncoder 2 su C1
Secondi
Elemento dello stream
INPUT_PIX ASSIST_OUT_PIX IDLE_PIX
B Mandelbrot 12 - MpegEncoder 4 su C1
Secondi
Elemento dello stream
INPUT_STREAM_STORE
B CCM Mandelbrot 12 - MpegEncoder 2 su C1
Secondi
Elemento dello stream
INPUT_STREAM_STORE

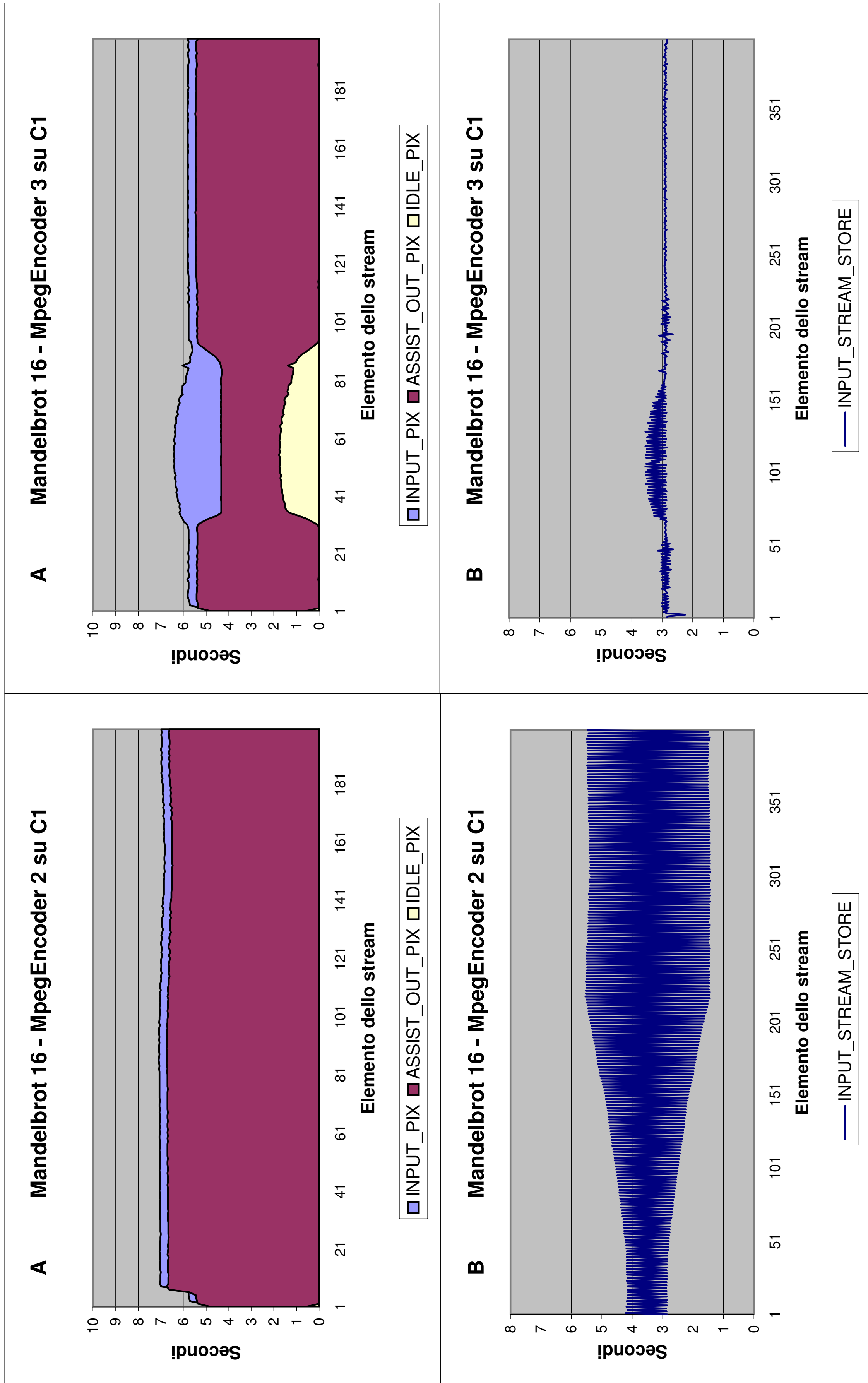

A CCM Mandelbrot 12 - MpegEncoder 3 su C1
Secondi
0
2
4
6
8
10
12
1
21
41
61
81
101
121
141
161
181
Elemento dello stream
INPUT_PIX
ASSIST_OUT_PIX
IDLE_PIX
A CCM Mandelbrot 12 - MpegEncoder 4 su C1
Secondi
0
2
4
6
8
10
12
1
21
41
61
81
101
121
141
161
181
Elemento dello stream
INPUT_PIX
ASSIST_OUT_PIX
IDLE_PIX
B CCM Mandelbrot 12 - MpegEncoder 3 su C1
Secondi
0
2
4
6
8
10
12
1
51
101
151
201
251
301
351
Elemento dello stream
INPUT_STREAM_STORE
B CCM Mandelbrot 12 - MpegEncoder 4 su C1
Secondi
0
2
4
6
8
10
12
1
51
101
151
201
251
301
351
Elemento dello stream
INPUT_STREAM_STORE

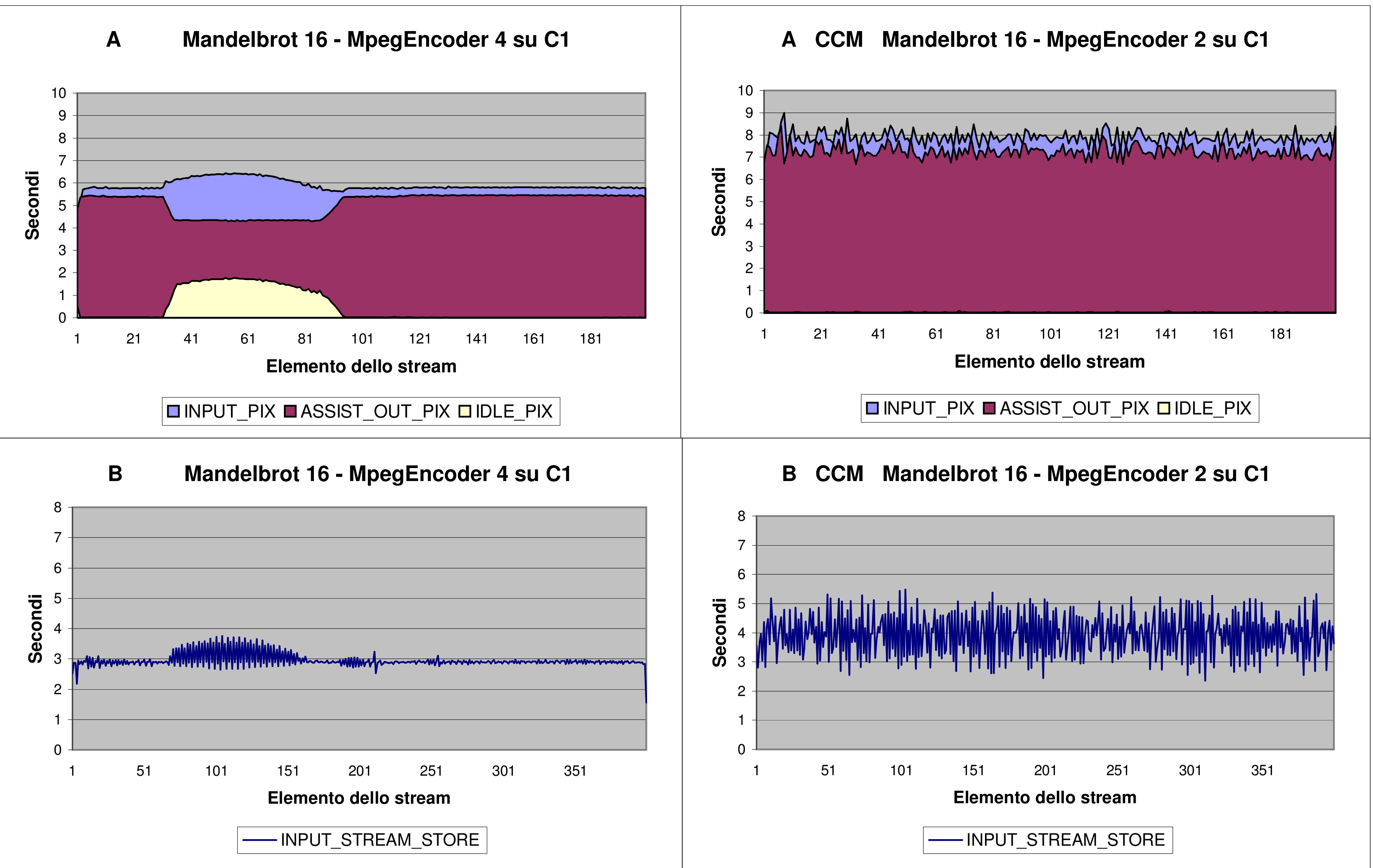

A
Mandelbrot 14 - MpegEncoder 2 su C1
Secondi
Elemento dello stream
INPUT_PIX
ASSIST_OUT_PIX
IDLE_PIX
A
Mandelbrot 14 - MpegEncoder 3 su C1
Secondi
Elemento dello stream
INPUT_PIX
ASSIST_OUT_PIX
IDLE_PIX
B
Mandelbrot 14 - MpegEncoder 2 su C1
Secondi
Elemento dello stream
INPUT_STREAM_STORE
B
Mandelbrot 14 - MpegEncoder 3 su C1
Secondi
Elemento dello stream
INPUT_STREAM_STORE

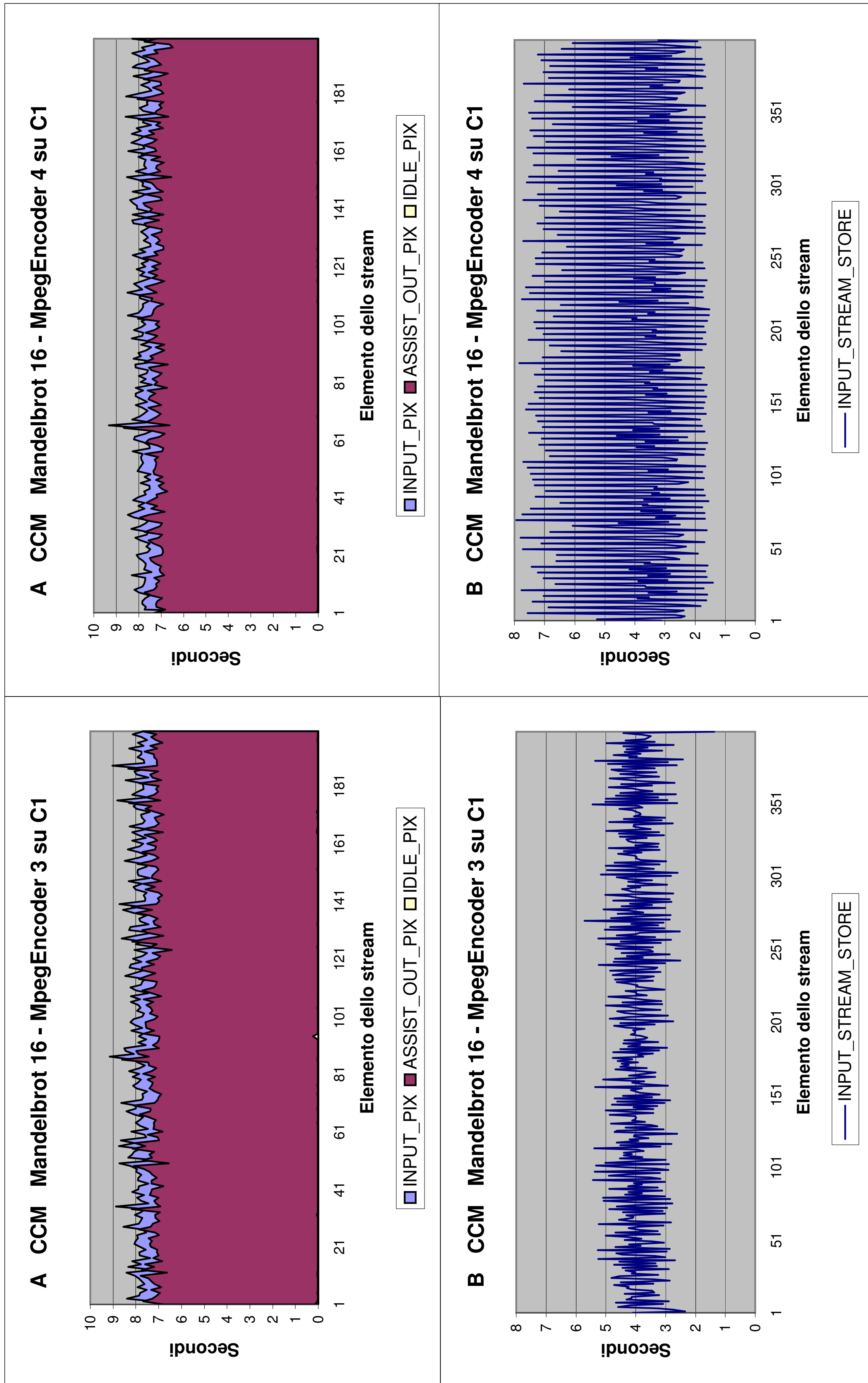


A Mandelbrot 14 - MpegEncoder 4 su C1
Secondi
Elemento dello stream
INPUT_PIX
ASSIST_OUT_PIX
IDLE_PIX
A CCM Mandelbrot 14 - MpegEncoder 2 su C1
Secondi
Elemento dello stream
INPUT_PIX
ASSIST_OUT_PIX
IDLE_PIX
B Mandelbrot 14 - MpegEncoder 4 su C1
Secondi
Elemento dello stream
INPUT_STREAM_STORE
B CCM Mandelbrot 14 - MpegEncoder 2 su C1
Secondi
Elemento dello stream
INPUT_STREAM_STORE

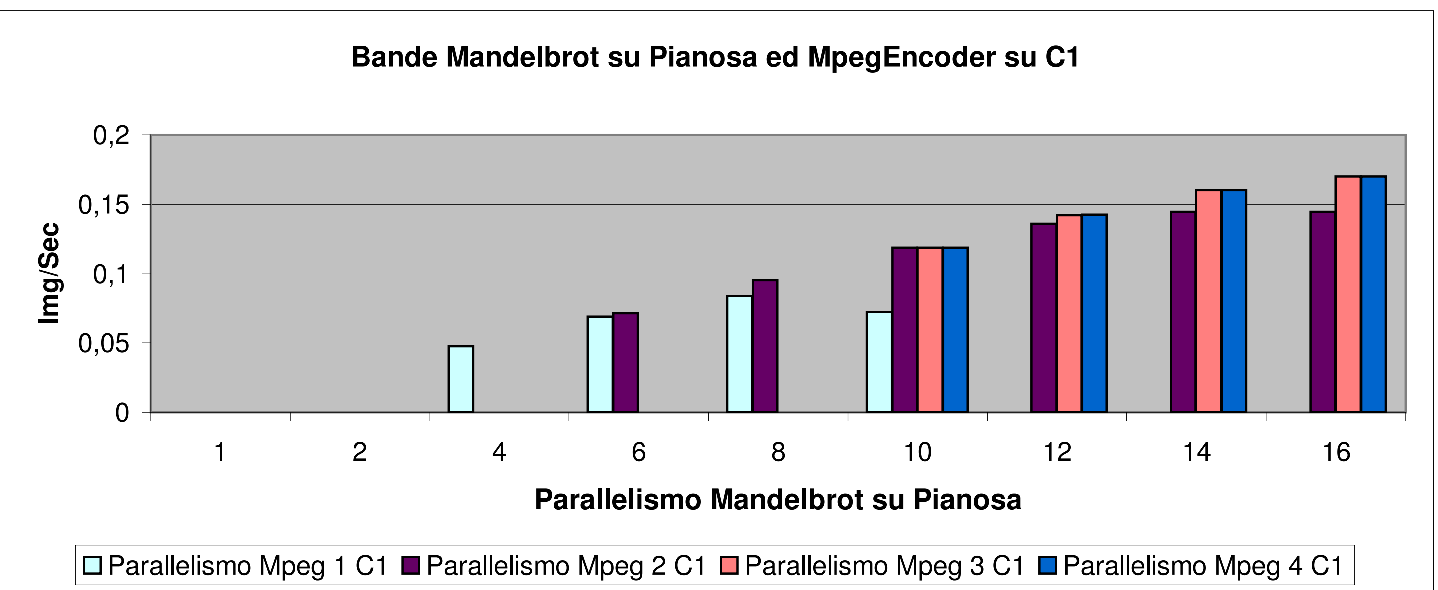

A CCM Mandelbrot 14 - MpegEncoder 3 su C1
Secondi
Elemento dello stream
INPUT_PIX
ASSIST_OUT_PIX
IDLE_PIX
A CCM Mandelbrot 14 - MpegEncoder 4 su C1
Secondi
Elemento dello stream
INPUT_PIX
ASSIST_OUT_PIX
IDLE_PIX
B CCM Mandelbrot 14 - MpegEncoder 3 su C1
Secondi
Elemento dello stream
INPUT_STREAM_STORE
B CCM Mandelbrot 14 - MpegEncoder 4 su C1
Secondi
Elemento dello stream
INPUT_STREAM_STORE


A
Mandelbrot 16 - MpegEncoder 3 su C1
Secondi
Elemento dello stream
INPUT_PIX
ASSIST_OUT_PIX
IDLE_PIX
B
Mandelbrot 16 - MpegEncoder 3 su C1
Secondi
Elemento dello stream
INPUT_STREAM_STORE
A
Mandelbrot 16 - MpegEncoder 2 su C1
Secondi
Elemento dello stream
INPUT_PIX
ASSIST_OUT_PIX
IDLE_PIX
B
Mandelbrot 16 - MpegEncoder 2 su C1
Secondi
Elemento dello stream
INPUT_STREAM_STORE

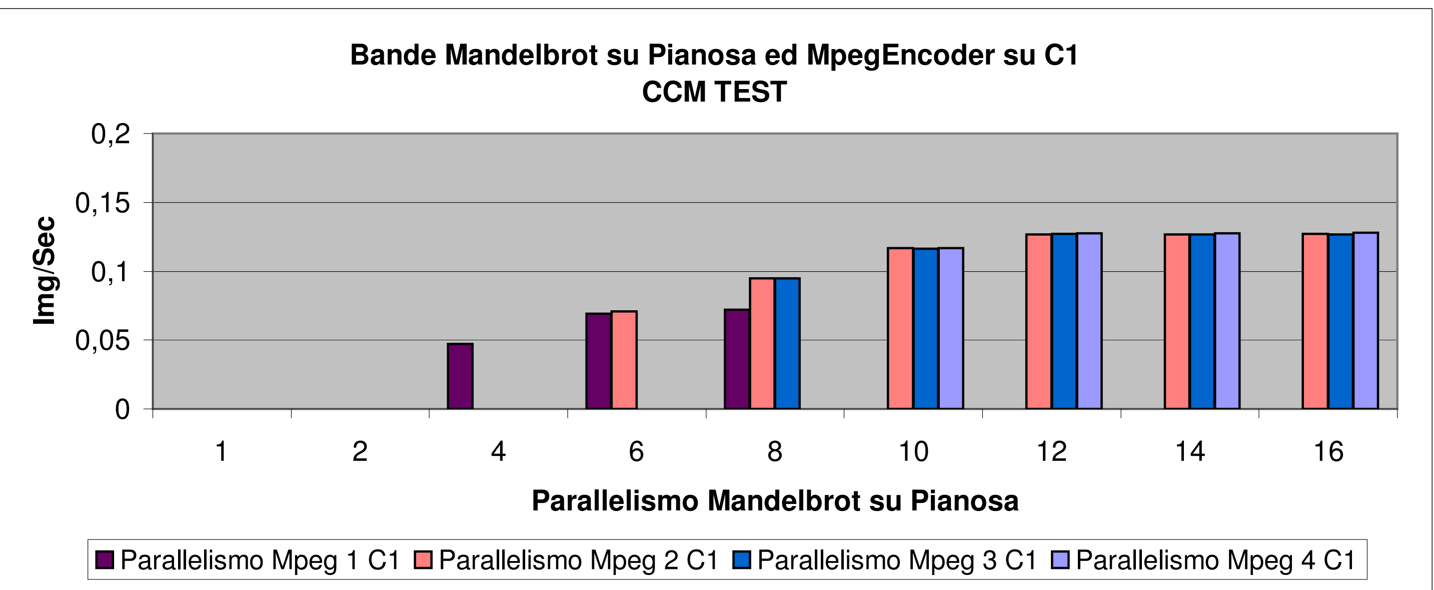

A Mandelbrot 16 - MpegEncoder 4 su C1
Secondi
Elemento dello stream
INPUT_PIX
ASSIST_OUT_PIX
IDLE_PIX
A CCM Mandelbrot 16 - MpegEncoder 2 su C1
Secondi
Elemento dello stream
INPUT_PIX
ASSIST_OUT_PIX
IDLE_PIX
B Mandelbrot 16 - MpegEncoder 4 su C1
Secondi
Elemento dello stream
INPUT_STREAM_STORE
B CCM Mandelbrot 16 - MpegEncoder 2 su C1
Secondi
Elemento dello stream
INPUT_STREAM_STORE


A CCM Mandelbrot 16 - MpegEncoder 3 su C1
Secondi
Elemento dello stream
INPUT_PIX
ASSIST_OUT_PIX
IDLE_PIX
A CCM Mandelbrot 16 - MpegEncoder 4 su C1
Secondi
Elemento dello stream
INPUT_PIX
ASSIST_OUT_PIX
IDLE_PIX
B CCM Mandelbrot 16 - MpegEncoder 3 su C1
Secondi
Elemento dello stream
INPUT_STREAM_STORE
B CCM Mandelbrot 16 - MpegEncoder 4 su C1
Secondi
Elemento dello stream
INPUT_STREAM_STORE

| | 1 | 2 | 4 | 6 | 8 | 10 | 12 | 14 | 16 |
|---|---|---|---|---|---|---|---|---|---|
| Parallelismo Mpeg 1 C1 | | | 0,047672 | 0,068896 | 0,083801 | 0,072402 | | | |
| Parallelismo Mpeg 2 C1 | | | | 0,071573 | 0,095446 | 0,11856 | 0,136104 | 0,144395 | 0,144499 |
| Parallelismo Mpeg 3 C1 | | | | | | 0,118527 | 0,142133 | 0,160301 | 0,170003 |
| Parallelismo Mpeg 4 C1 | | | | | | 0,118552 | 0,142359 | 0,160361 | 0,170103 |



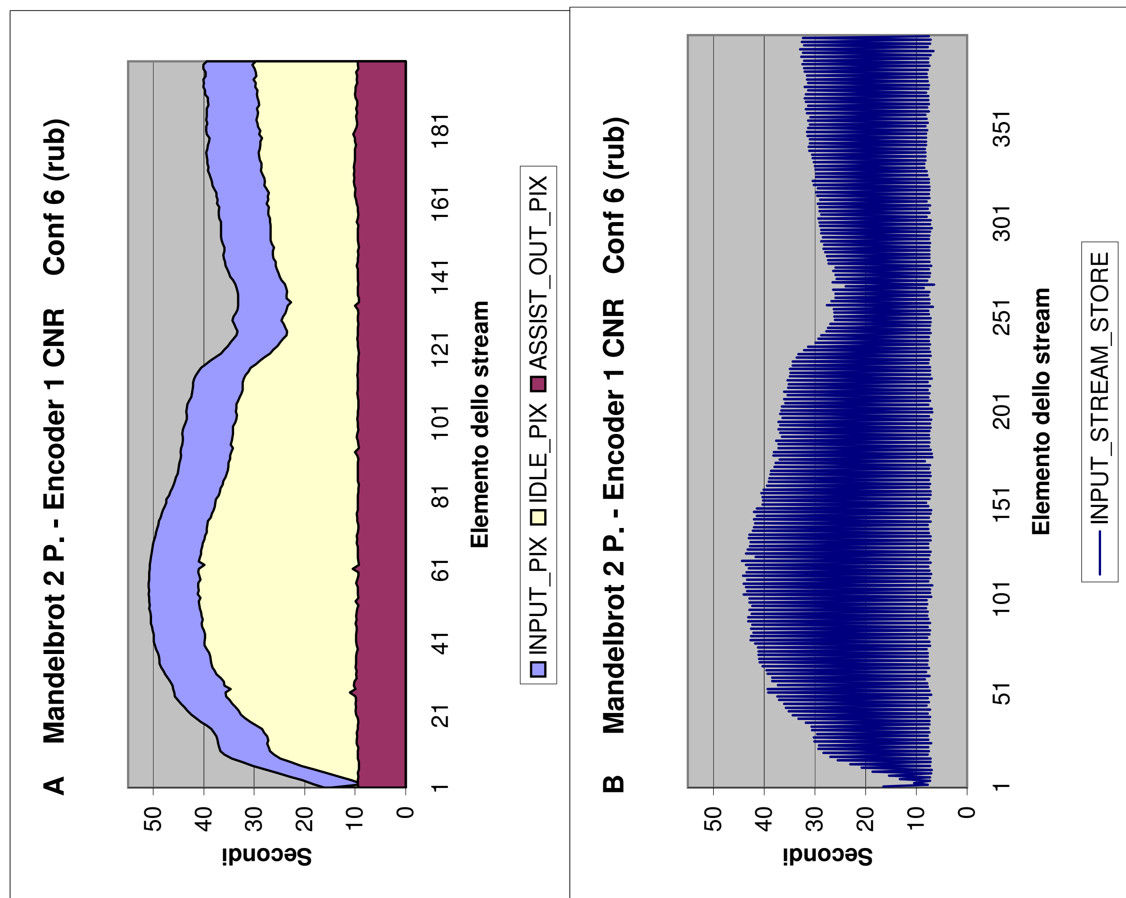
| CCM | 1 | 2 | 4 | 6 | 8 | 10 | 12 | 14 | 16 |
|---|---|---|---|---|---|---|---|---|---|
| Parallelismo Mpeg 1 C1 | | | 0,047392 | 0,069266 | 0,071978 | | | | |
| Parallelismo Mpeg 2 C1 | | | | 0,071012 | 0,09469 | 0,116604 | 0,126538 | 0,126908 | 0,127277 |
| Parallelismo Mpeg 3 C1 | | | | | 0,09465 | 0,116507 | 0,126979 | 0,126798 | 0,126823 |
| Parallelismo Mpeg 4 C1 | | | | | | 0,116738 | 0,127582 | 0,127726 | 0,127756 |

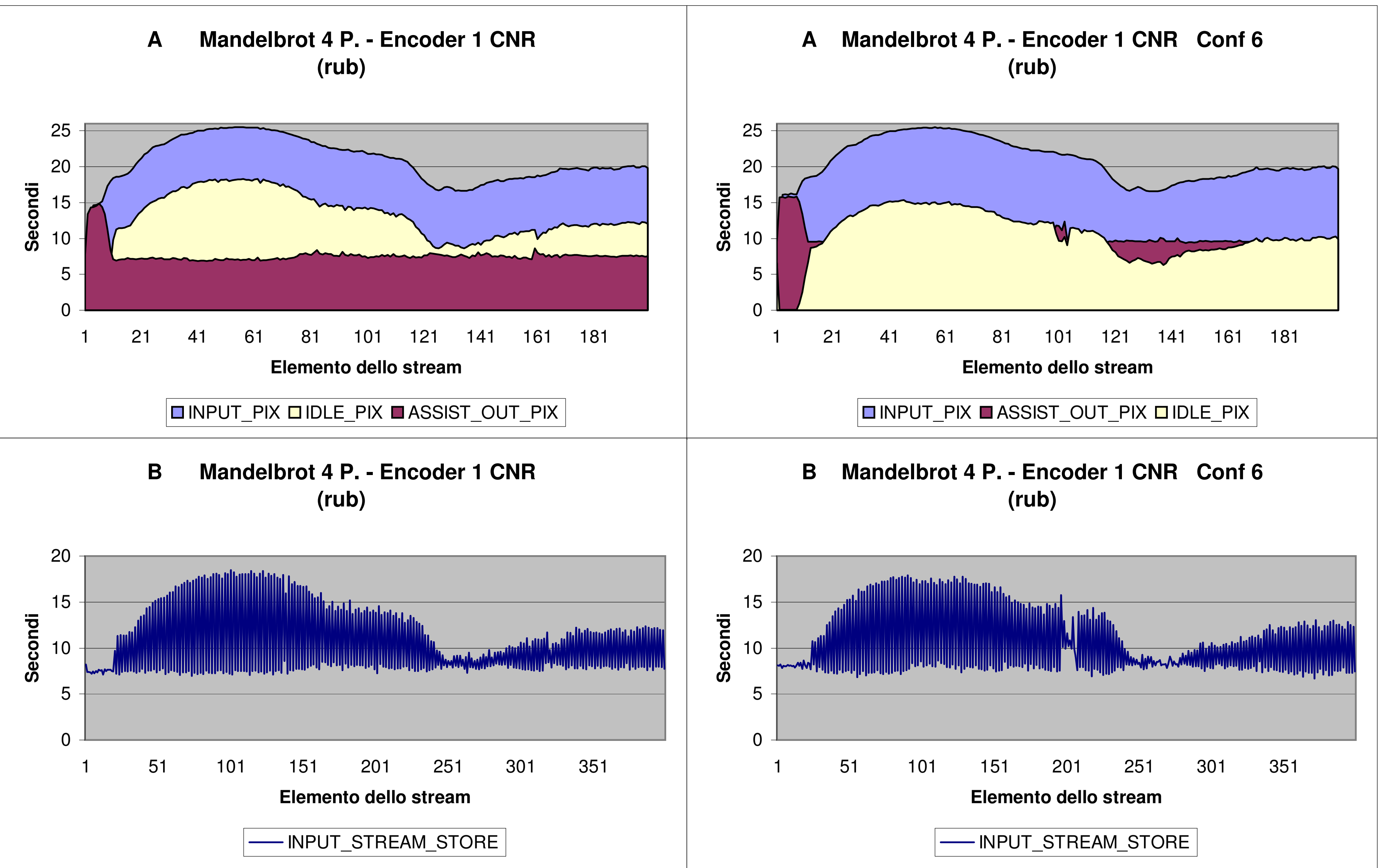


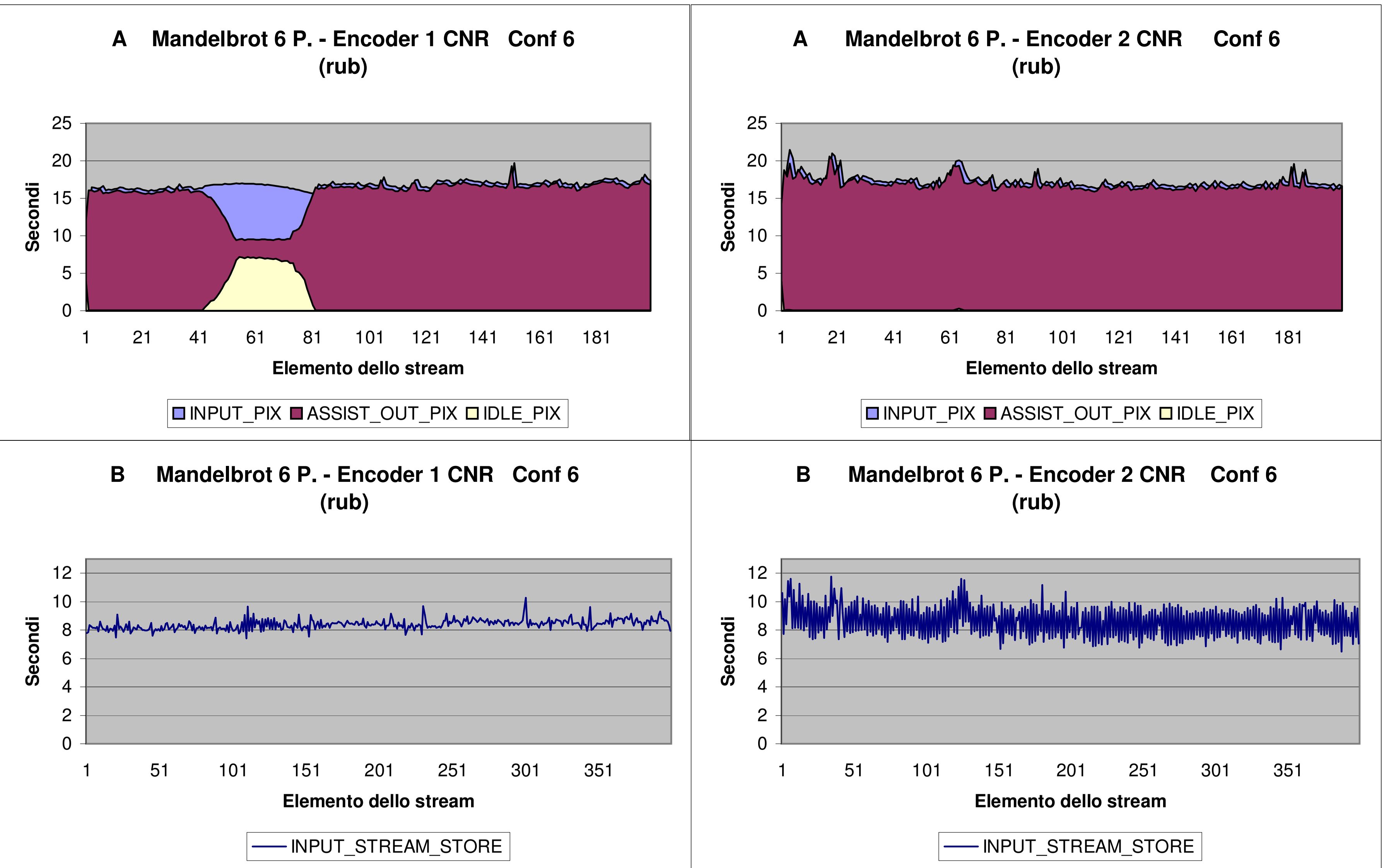
# B.3 Prove sulla griglia

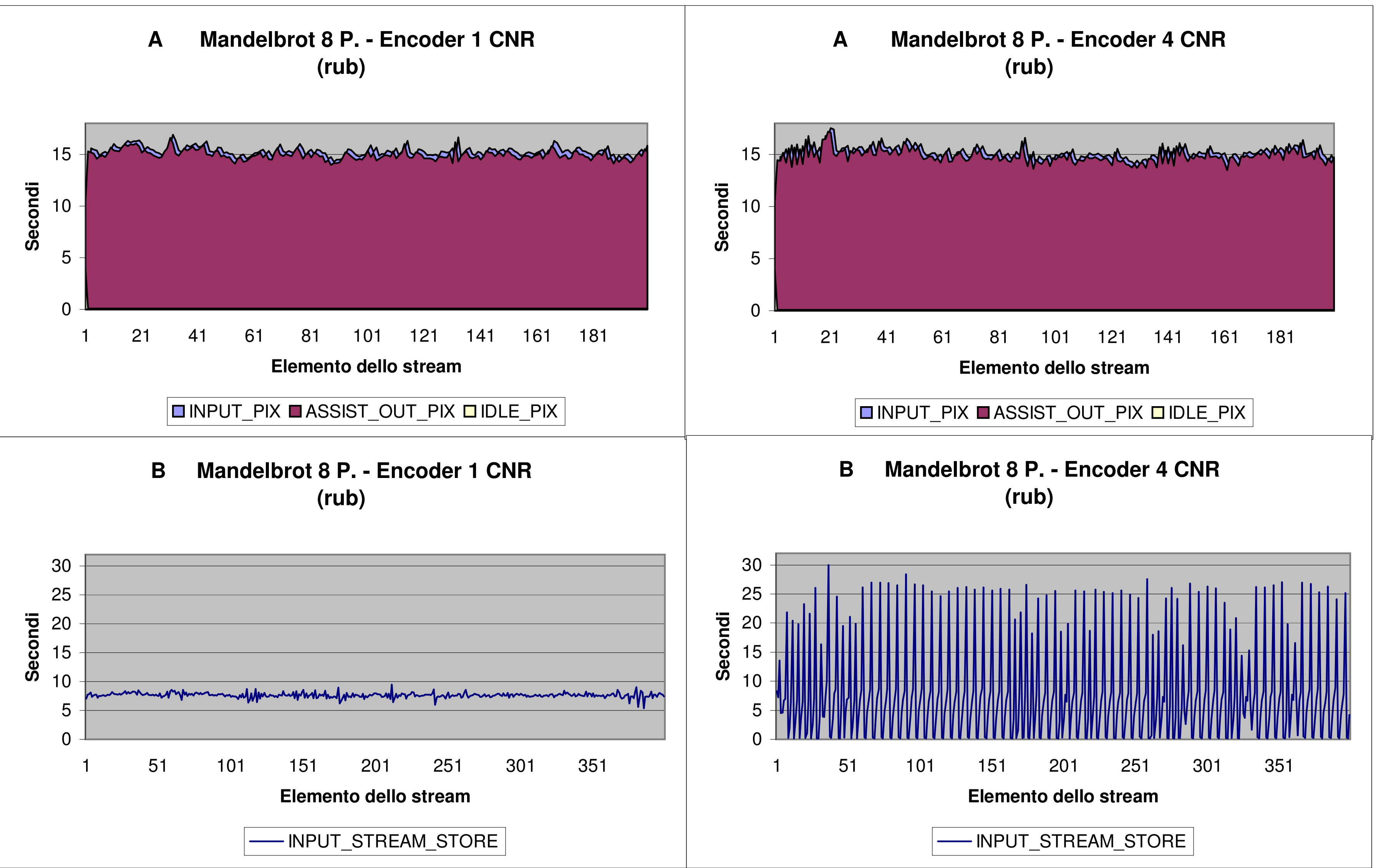
## Macchina master al CNR: Rubentino

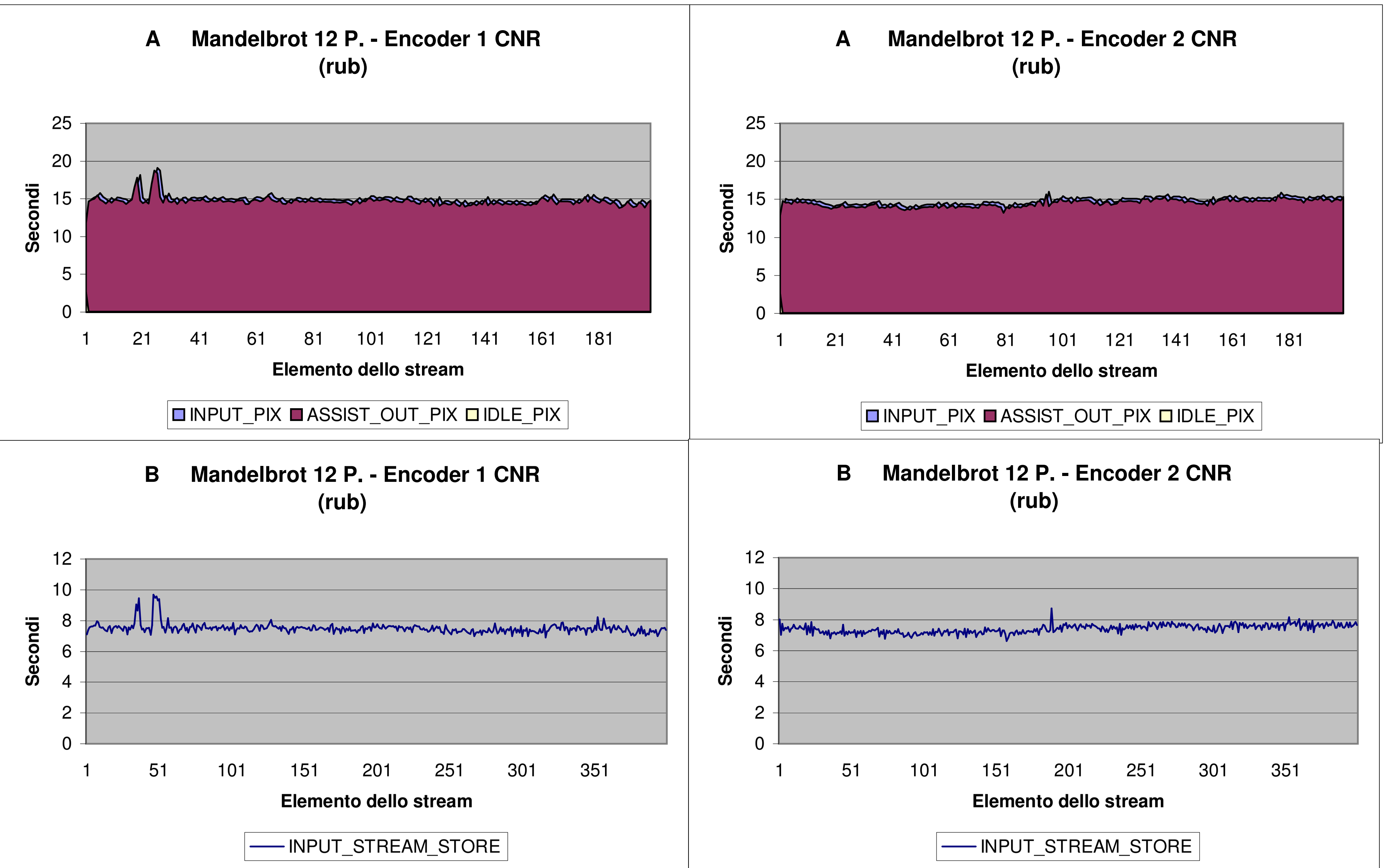
| Pagina | Collo di bottiglia Mandelbrot | Collo di bottiglia MpegEncoder | Dinamicità Applicazione | Applicazione Politica Adattiva |
|---|---|---|---|---|
| 240 | * | | | |
| 241 | * | | | |
| 242 | | * | * | |
| 243 | | * | | |
| 244 | | * | | |
| 245 | | * | | |
| 246 | | * | | |
| 247 | | * | | |
| 248 | | * | | |

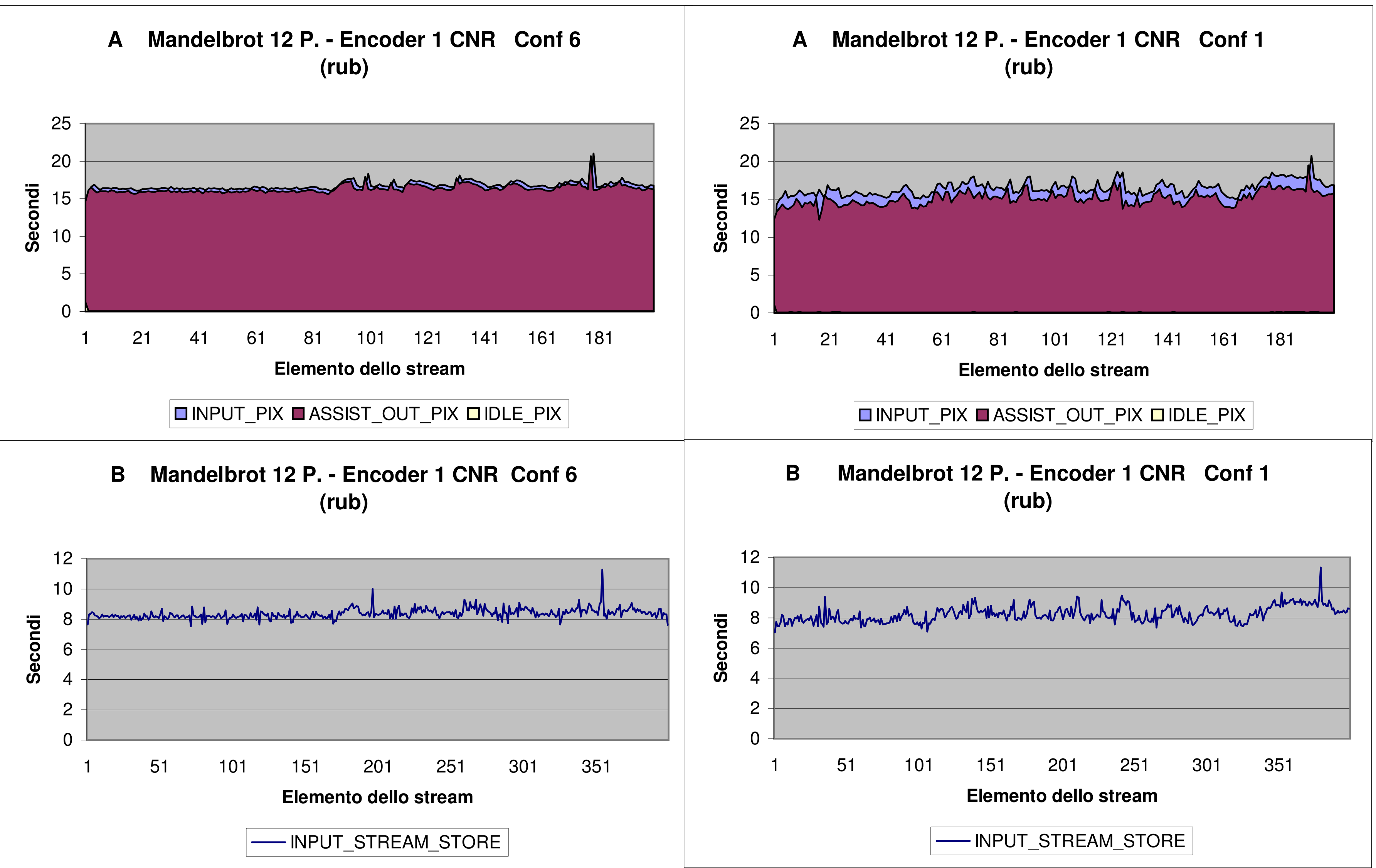

A Mandelbrot 2 P. - Encoder 1 CNR Conf 6 (rub)
Secondi
0
10
20
30
40
50
1
21
41
61
81
101
121
141
161
181
Elemento dello stream
INPUT_PIX
IDLE_PIX
ASSIST_OUT_PIX
B Mandelbrot 2 P. - Encoder 1 CNR Conf 6 (rub)
Secondi
0
10
20
30
40
50
1
51
101
151
201
251
301
351
Elemento dello stream
INPUT_STREAM_STORE

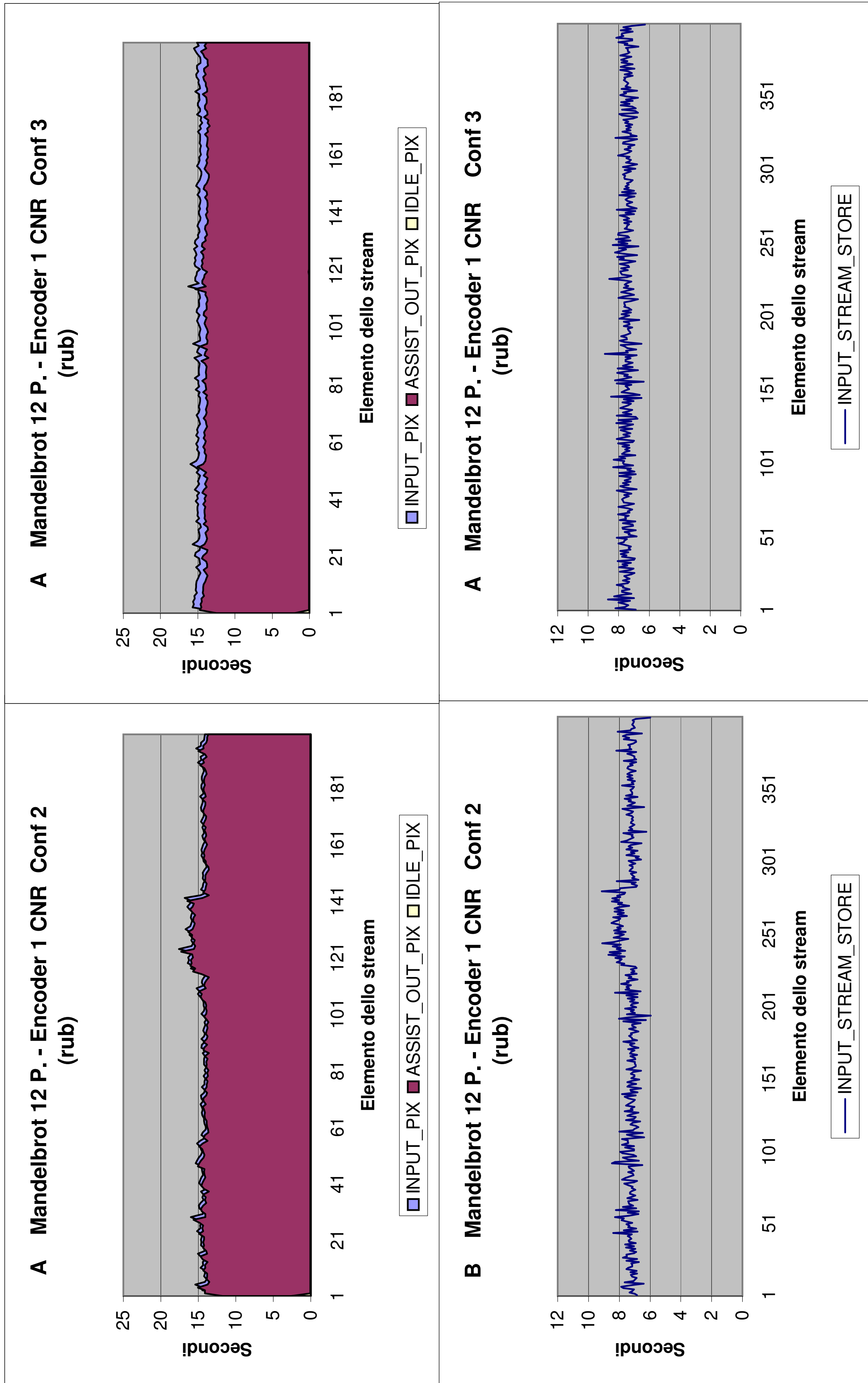

A Mandelbrot 4 P. - Encoder 1 CNR
(rub)
Secondi
Elemento dello stream
INPUT_PIX IDLE_PIX ASSIST_OUT_PIX
A Mandelbrot 4 P. - Encoder 1 CNR Conf 6
(rub)
Secondi
Elemento dello stream
INPUT_PIX ASSIST_OUT_PIX IDLE_PIX
B Mandelbrot 4 P. - Encoder 1 CNR
(rub)
Secondi
Elemento dello stream
INPUT_STREAM_STORE
B Mandelbrot 4 P. - Encoder 1 CNR Conf 6
(rub)
Secondi
Elemento dello stream
INPUT_STREAM_STORE

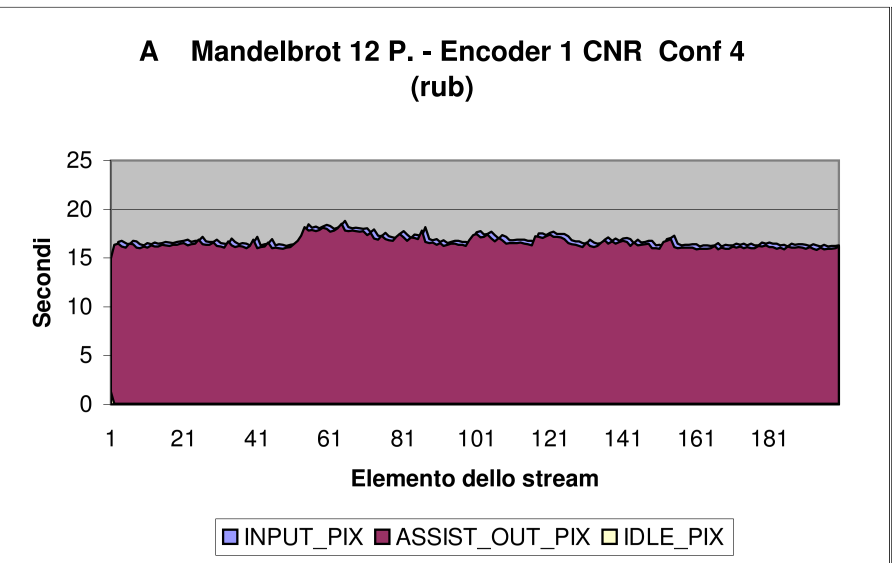

A Mandelbrot 6 P. - Encoder 1 CNR Conf 6
(rub)
Secondi
Elemento dello stream
INPUT_PIX ASSIST_OUT_PIX IDLE_PIX
A Mandelbrot 6 P. - Encoder 2 CNR Conf 6
(rub)
Secondi
Elemento dello stream
INPUT_PIX ASSIST_OUT_PIX IDLE_PIX
B Mandelbrot 6 P. - Encoder 1 CNR Conf 6
(rub)
Secondi
Elemento dello stream
INPUT_STREAM_STORE
B Mandelbrot 6 P. - Encoder 2 CNR Conf 6
(rub)
Secondi
Elemento dello stream
INPUT_STREAM_STORE

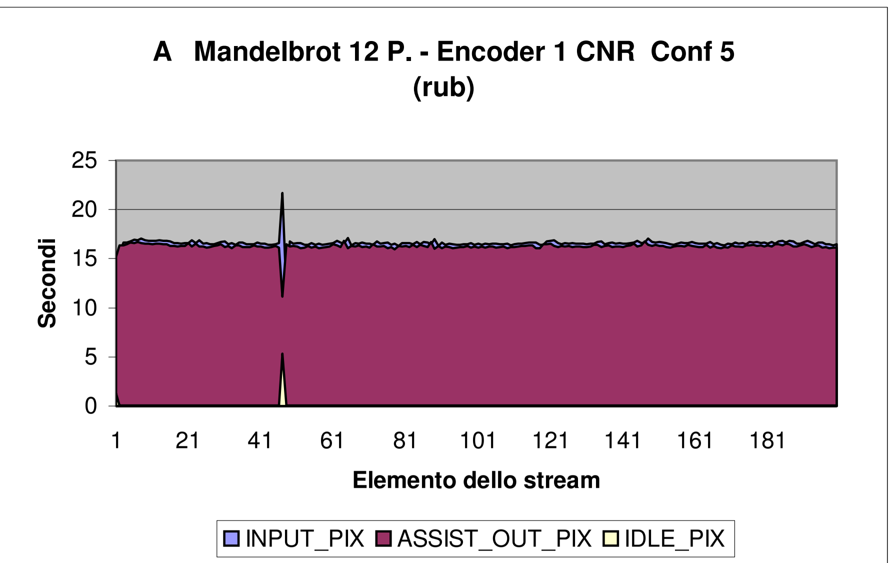

A Mandelbrot 8 P. - Encoder 1 CNR (rub)
Secondi
0
5
10
15
1
21
41
61
81
101
121
141
161
181
Elemento dello stream
INPUT_PIX
ASSIST_OUT_PIX
IDLE_PIX
A Mandelbrot 8 P. - Encoder 4 CNR (rub)
Secondi
0
5
10
15
1
21
41
61
81
101
121
141
161
181
Elemento dello stream
INPUT_PIX
ASSIST_OUT_PIX
IDLE_PIX
B Mandelbrot 8 P. - Encoder 1 CNR (rub)
Secondi
0
5
10
15
20
25
30
1
51
101
151
201
251
301
351
Elemento dello stream
INPUT_STREAM_STORE
B Mandelbrot 8 P. - Encoder 4 CNR (rub)
Secondi
0
5
10
15
20
25
30
1
51
101
151
201
251
301
351
Elemento dello stream
INPUT_STREAM_STORE

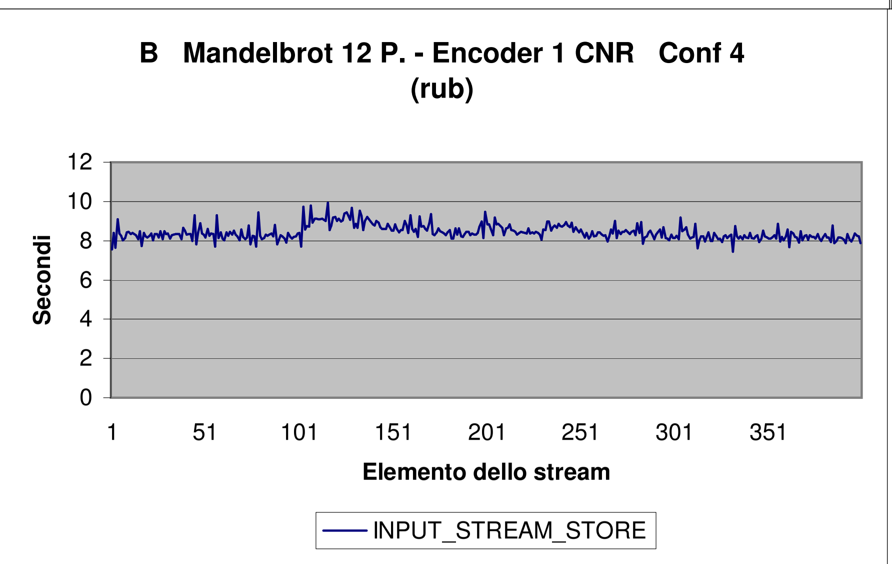

A Mandelbrot 12 P. - Encoder 1 CNR (rub)
Secondi
Elemento dello stream
INPUT_PIX
ASSIST_OUT_PIX
IDLE_PIX
A Mandelbrot 12 P. - Encoder 2 CNR (rub)
Secondi
Elemento dello stream
INPUT_PIX
ASSIST_OUT_PIX
IDLE_PIX
B Mandelbrot 12 P. - Encoder 1 CNR (rub)
Secondi
Elemento dello stream
INPUT_STREAM_STORE
B Mandelbrot 12 P. - Encoder 2 CNR (rub)
Secondi
Elemento dello stream
INPUT_STREAM_STORE

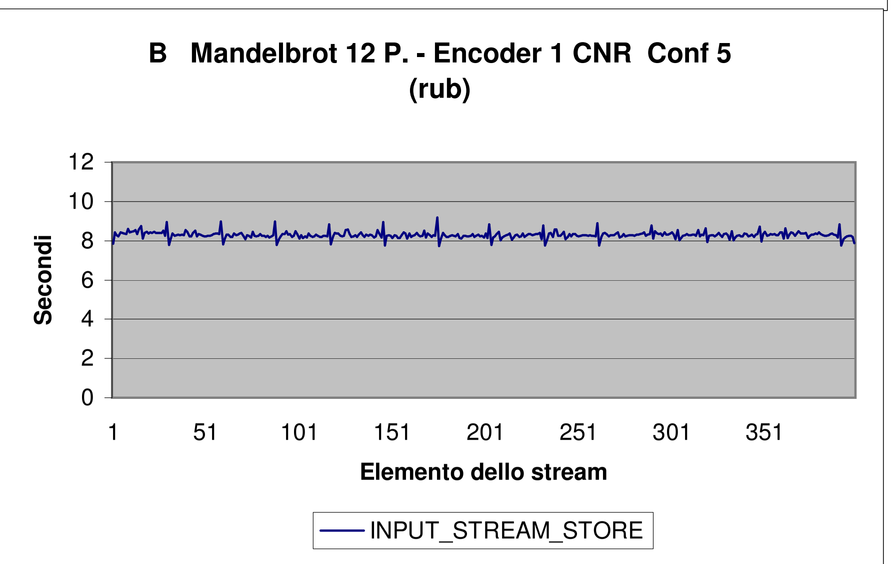

A Mandelbrot 12 P. - Encoder 1 CNR Conf 6 (rub)
Secondi
25
20
15
10
5
0
1
21
41
61
81
101
121
141
161
181
Elemento dello stream
INPUT_PIX
ASSIST_OUT_PIX
IDLE_PIX
A Mandelbrot 12 P. - Encoder 1 CNR Conf 1 (rub)
Secondi
Elemento dello stream
INPUT_PIX
ASSIST_OUT_PIX
IDLE_PIX
B Mandelbrot 12 P. - Encoder 1 CNR Conf 6 (rub)
Secondi
12
10
8
6
4
2
0
1
51
101
151
201
251
301
351
Elemento dello stream
INPUT_STREAM_STORE
B Mandelbrot 12 P. - Encoder 1 CNR Conf 1 (rub)
Secondi
Elemento dello stream
INPUT_STREAM_STORE

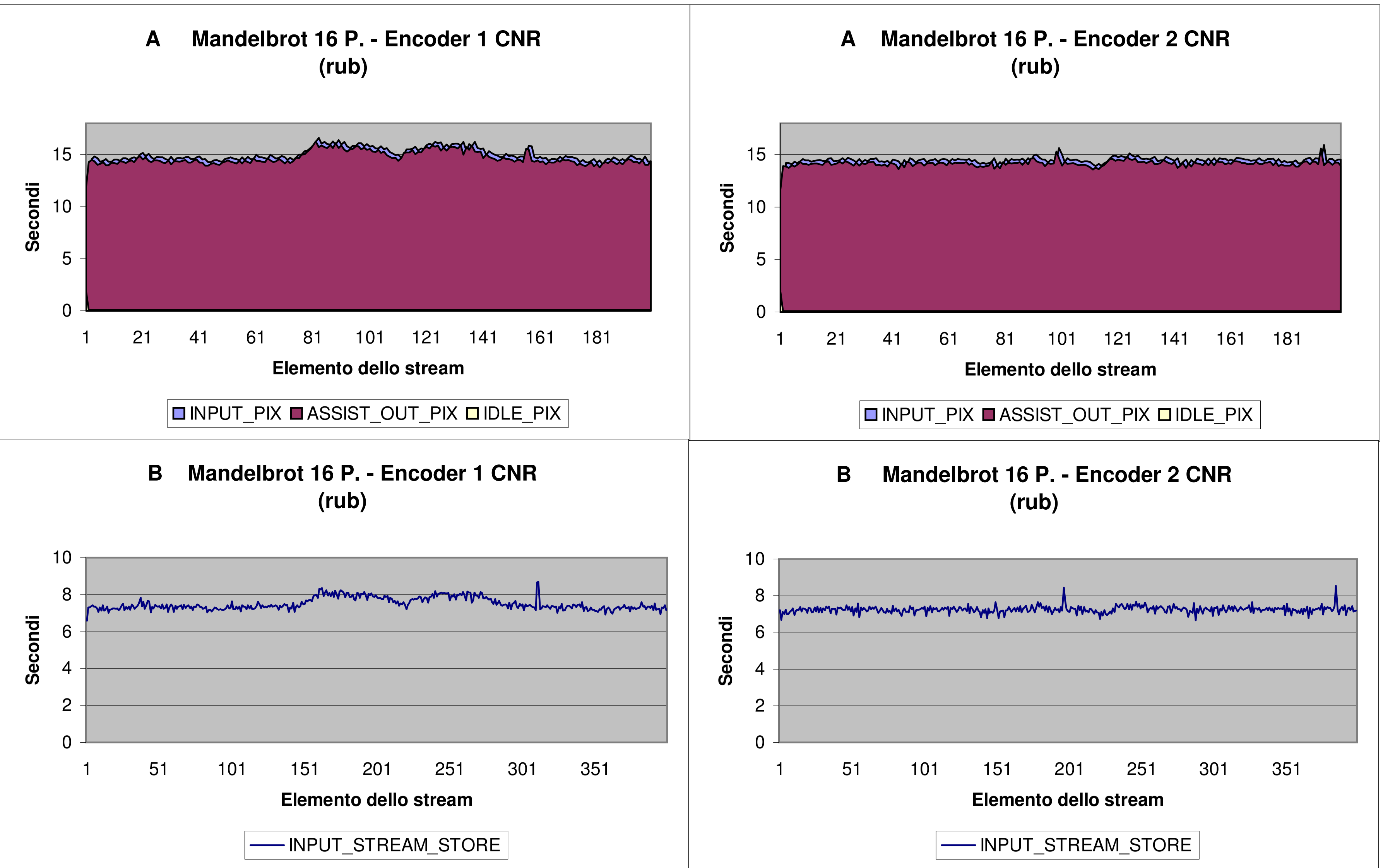

A Mandelbrot 12 P. - Encoder 1 CNR Conf 2
(rub)
Secondi
25
20
15
10
5
0
1
21
41
61
81
101
121
141
161
181
Elemento dello stream
INPUT_PIX
ASSIST_OUT_PIX
IDLE_PIX
A Mandelbrot 12 P. - Encoder 1 CNR Conf 3
(rub)
Secondi
25
20
15
10
5
0
1
21
41
61
81
101
121
141
161
181
Elemento dello stream
INPUT_PIX
ASSIST_OUT_PIX
IDLE_PIX
B Mandelbrot 12 P. - Encoder 1 CNR Conf 2
(rub)
Secondi
12
10
8
6
4
2
0
1
51
101
151
201
251
301
351
Elemento dello stream
INPUT_STREAM_STORE
A Mandelbrot 12 P. - Encoder 1 CNR Conf 3
(rub)
Secondi
12
10
8
6
4
2
0
1
51
101
151
201
251
301
351
Elemento dello stream
INPUT_STREAM_STORE


A Mandelbrot 12 P. - Encoder 1 CNR Conf 4
(rub)
Secondi
25
20
15
10
5
0
1
21
41
61
81
101
121
141
161
181
Elemento dello stream
INPUT_PIX
ASSIST_OUT_PIX
IDLE_PIX


A Mandelbrot 12 P. - Encoder 1 CNR Conf 5
(rub)
Secondi
25
20
15
10
5
0
1
21
41
61
81
101
121
141
161
181
Elemento dello stream
INPUT_PIX
ASSIST_OUT_PIX
IDLE_PIX

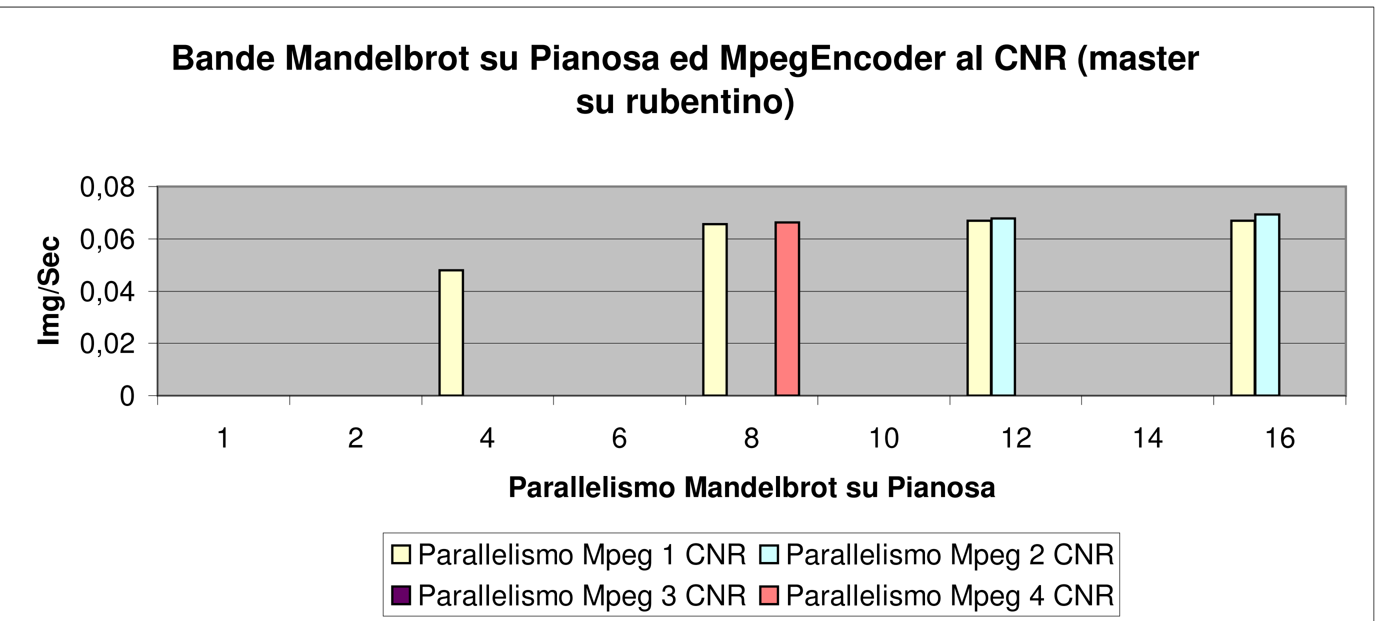

B Mandelbrot 12 P. - Encoder 1 CNR Conf 4
(rub)
Secondi
12
10
8
6
4
2
0
1
51
101
151
201
251
301
351
Elemento dello stream
INPUT_STREAM_STORE


B Mandelbrot 12 P. - Encoder 1 CNR Conf 5
(rub)
Secondi
12
10
8
6
4
2
0
1
51
101
151
201
251
301
351
Elemento dello stream
INPUT_STREAM_STORE



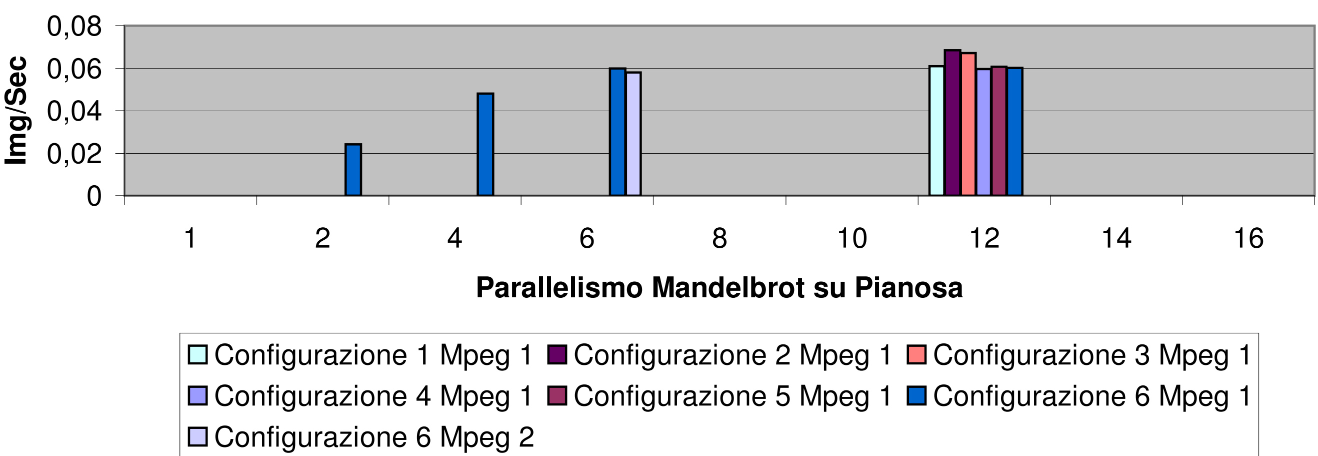
A Mandelbrot 16 P. - Encoder 1 CNR (rub)
Secondi
0
5
10
15
1
21
41
61
81
101
121
141
161
181
Elemento dello stream
INPUT_PIX
ASSIST_OUT_PIX
IDLE_PIX
A Mandelbrot 16 P. - Encoder 2 CNR (rub)
Secondi
0
5
10
15
1
21
41
61
81
101
121
141
161
181
Elemento dello stream
INPUT_PIX
ASSIST_OUT_PIX
IDLE_PIX
B Mandelbrot 16 P. - Encoder 1 CNR (rub)
Secondi
0
2
4
6
8
10
1
51
101
151
201
251
301
351
Elemento dello stream
INPUT_STREAM_STORE
B Mandelbrot 16 P. - Encoder 2 CNR (rub)
Secondi
0
2
4
6
8
10
1
51
101
151
201
251
301
351
Elemento dello stream
INPUT_STREAM_STORE

| | 1 | 2 | 4 | 6 | 8 | 10 | 12 | 14 | 16 |
|---|---|---|---|---|---|---|---|---|---|
| Parallelismo Mpeg 1 CNR | | | 0,048052 | | 0,065711 | | 0,067023 | | 0,066925 |
| Parallelismo Mpeg 2 CNR | | | | | | | 0,0678 | | 0,06934 |
| Parallelismo Mpeg 3 CNR | | | | | | | | | |
| Parallelismo Mpeg 4 CNR | | | | | 0,066249 | | | | |



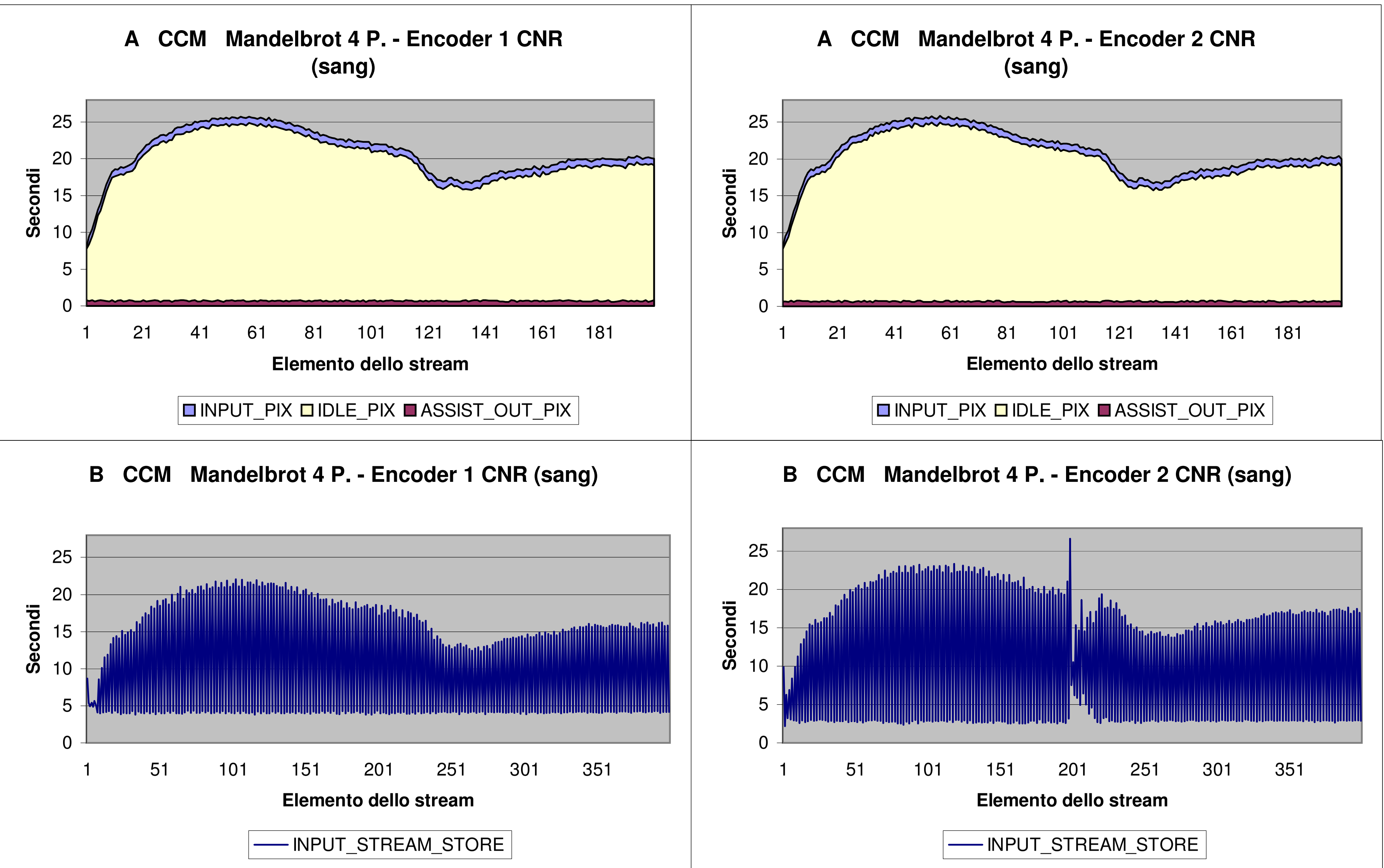
C1: configurazione 1: i processi pix in esecuzione su capraia e stream_store su c1
C2: configurazione 2: i processi pix in esecuzione su c1 e stream_store su capraia
C3: configurazione 3: i processi pix e stream_store in esecuzione su capraia
C4: configurazione 4: il processo pix_ivp su c2, pix_osm su c1, stream_ivp su capraia, stream_osm c3
C5: configurazione 5: il processo pix_ivp su c2, pix_osm su capraia, stream_ivp su c1, stream_osm c3
C6: configurazione 6: il processo pix_ivp su u2, pix_osm su c1, stream_ivp su capraia, stream_osm c3
quelli senza suffisso stanno ad indicare che i processi in oggetto stanno tutti su c1

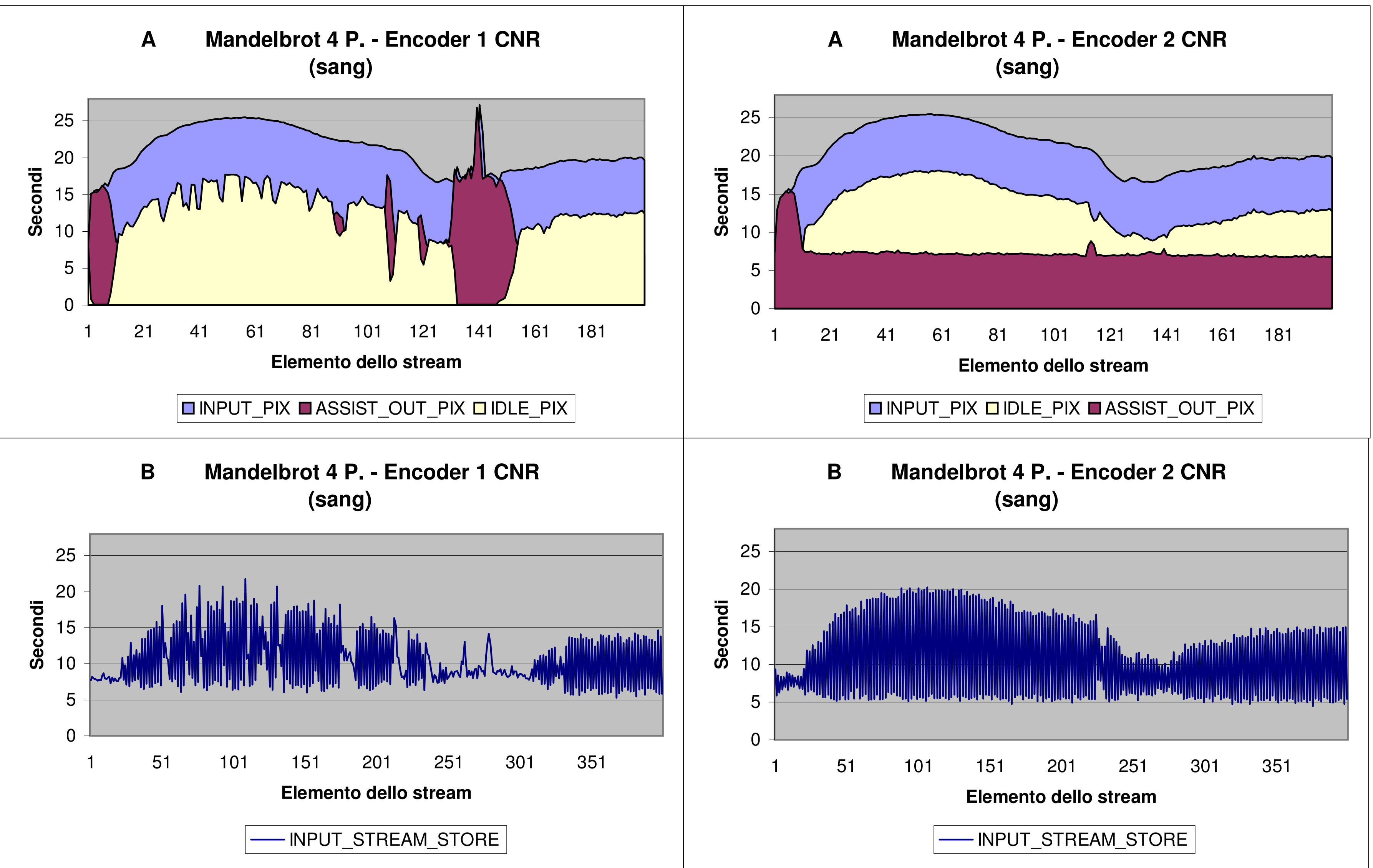
| Parallelismo Mpeg | 1 | 2 | 4 | 6 | 8 | 10 | 12 | 14 | 16 |
|---|---|---|---|---|---|---|---|---|---|
| Configurazione 1 Mpeg 1 | | | | | | | 0,061 | | |
| Configurazione 2 Mpeg 1 | | | | | | | 0,068459 | | |
| Configurazione 3 Mpeg 1 | | | | | | | 0,067149 | | |
| Configurazione 4 Mpeg 1 | | | | | | | 0,059551 | | |
| Configurazione 5 Mpeg 1 | | | | | | | 0,060545 | | |
| Configurazione 6 Mpeg 1 | | 0,02415 | 0,048102 | 0,059786 | | | 0,060035 | | |
| Configurazione 6 Mpeg 2 | | | | 0,05808 | | | | | |

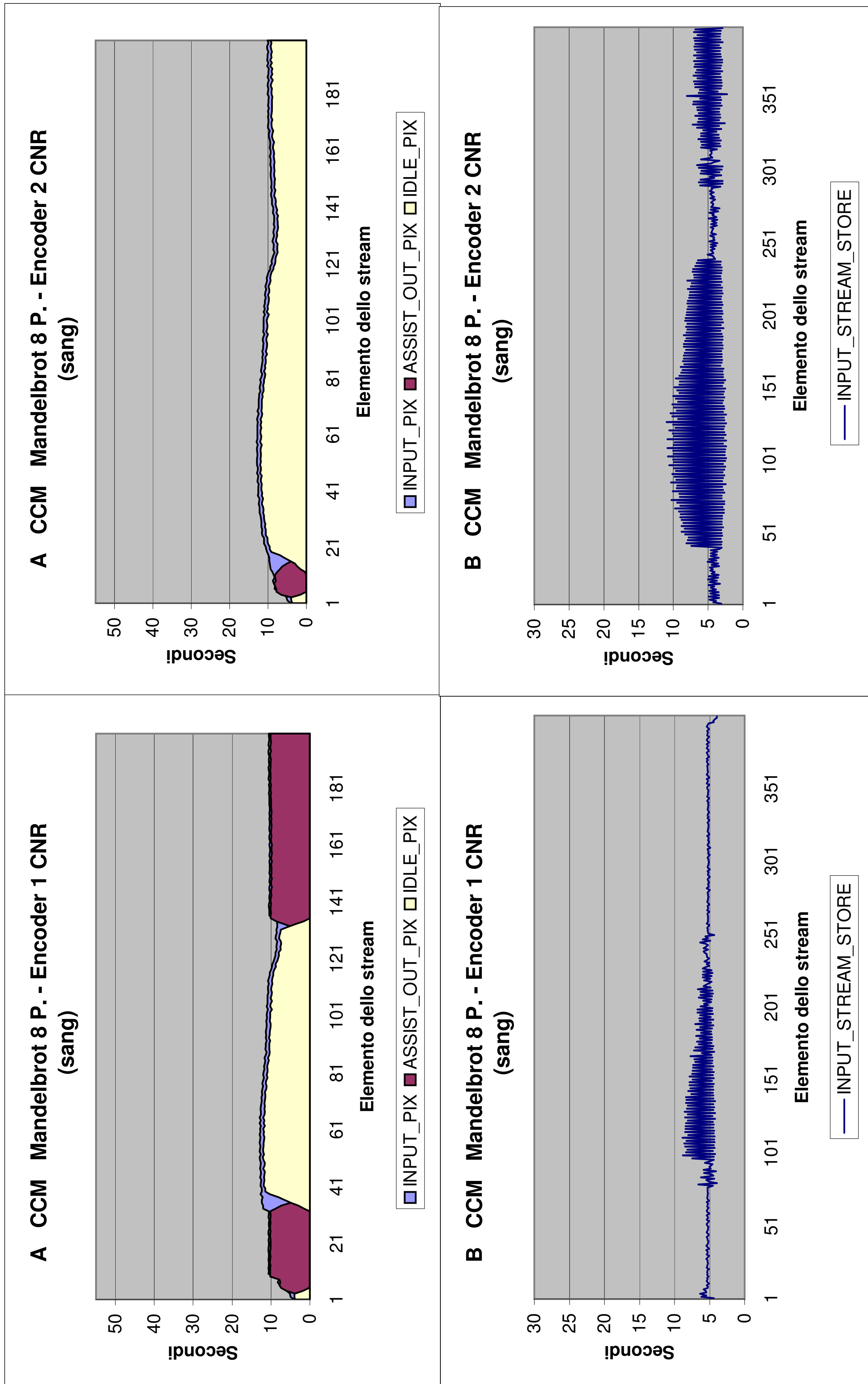


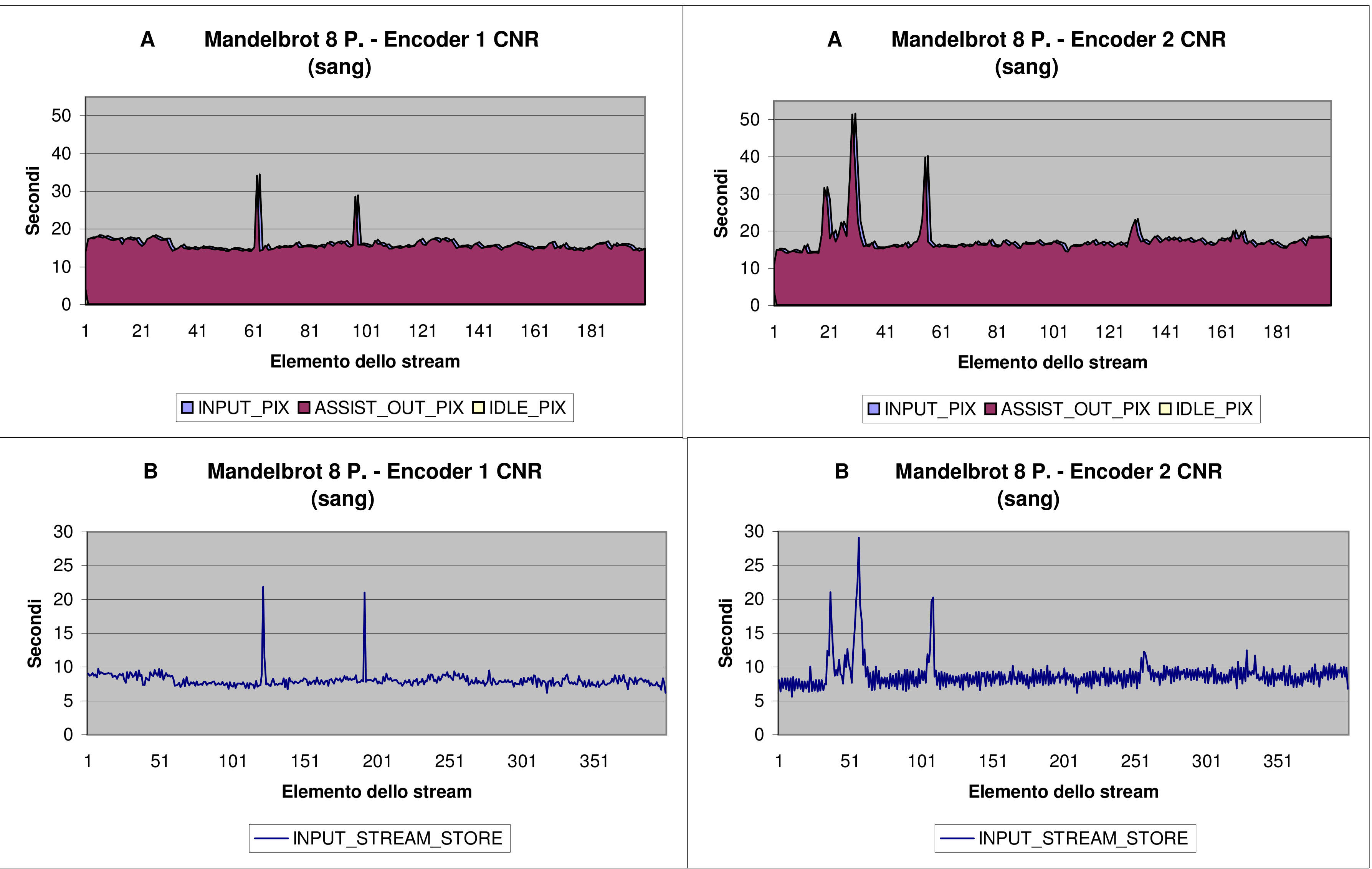
## Macchina master al CNR: Sangiovese

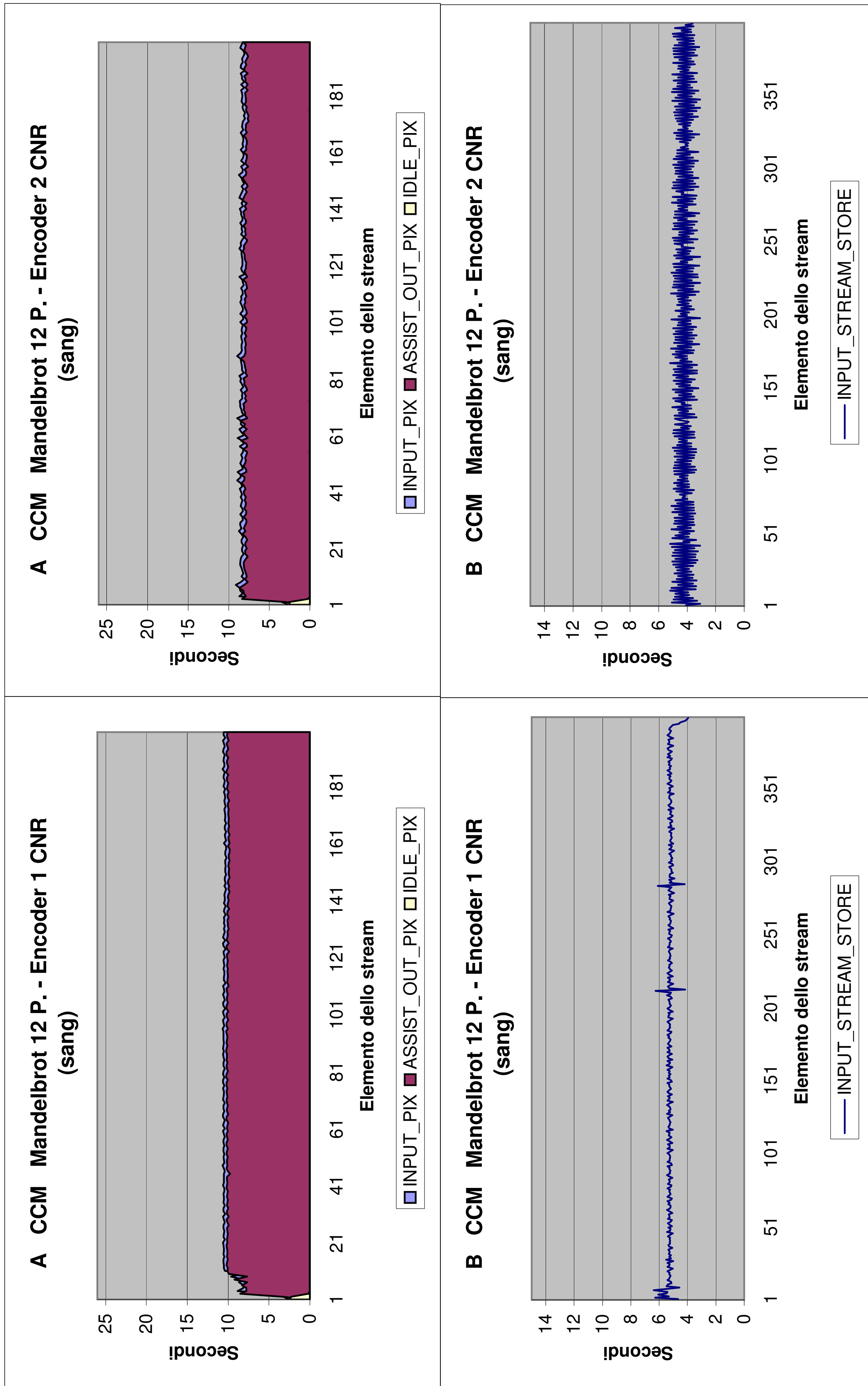
| Pagina | Collo di bottiglia Mandelbrot | Collo di bottiglia MpegEncoder | Dinamicità Applicazione | Applicazione Politica Adattiva |
|---|---|---|---|---|
| 252 | * | | | |
| 253 | * | | * | |
| 254 | * | | * | |
| 255 | | * | | |
| 256 | | * | | |
| 257 | | * | | |
| 258 | | * | | |
| 259 | | * | | |
| 260 | | * | | |

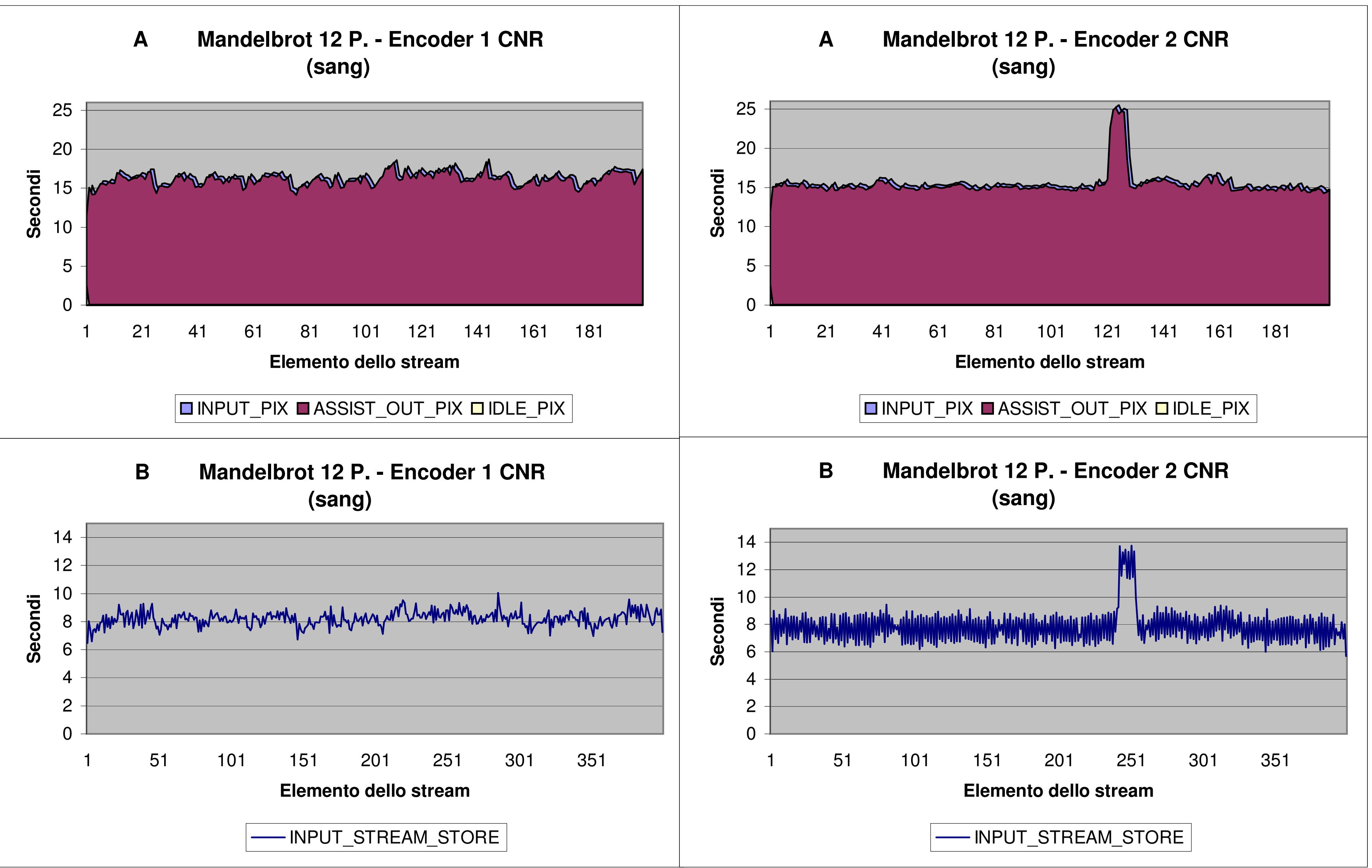

A CCM Mandelbrot 4 P. - Encoder 1 CNR (sang)
Secondi
Elemento dello stream
INPUT_PIX
IDLE_PIX
ASSIST_OUT_PIX
A CCM Mandelbrot 4 P. - Encoder 2 CNR (sang)
Secondi
Elemento dello stream
INPUT_PIX
IDLE_PIX
ASSIST_OUT_PIX
B CCM Mandelbrot 4 P. - Encoder 1 CNR (sang)
Secondi
Elemento dello stream
INPUT_STREAM_STORE
B CCM Mandelbrot 4 P. - Encoder 2 CNR (sang)
Secondi
Elemento dello stream
INPUT_STREAM_STORE

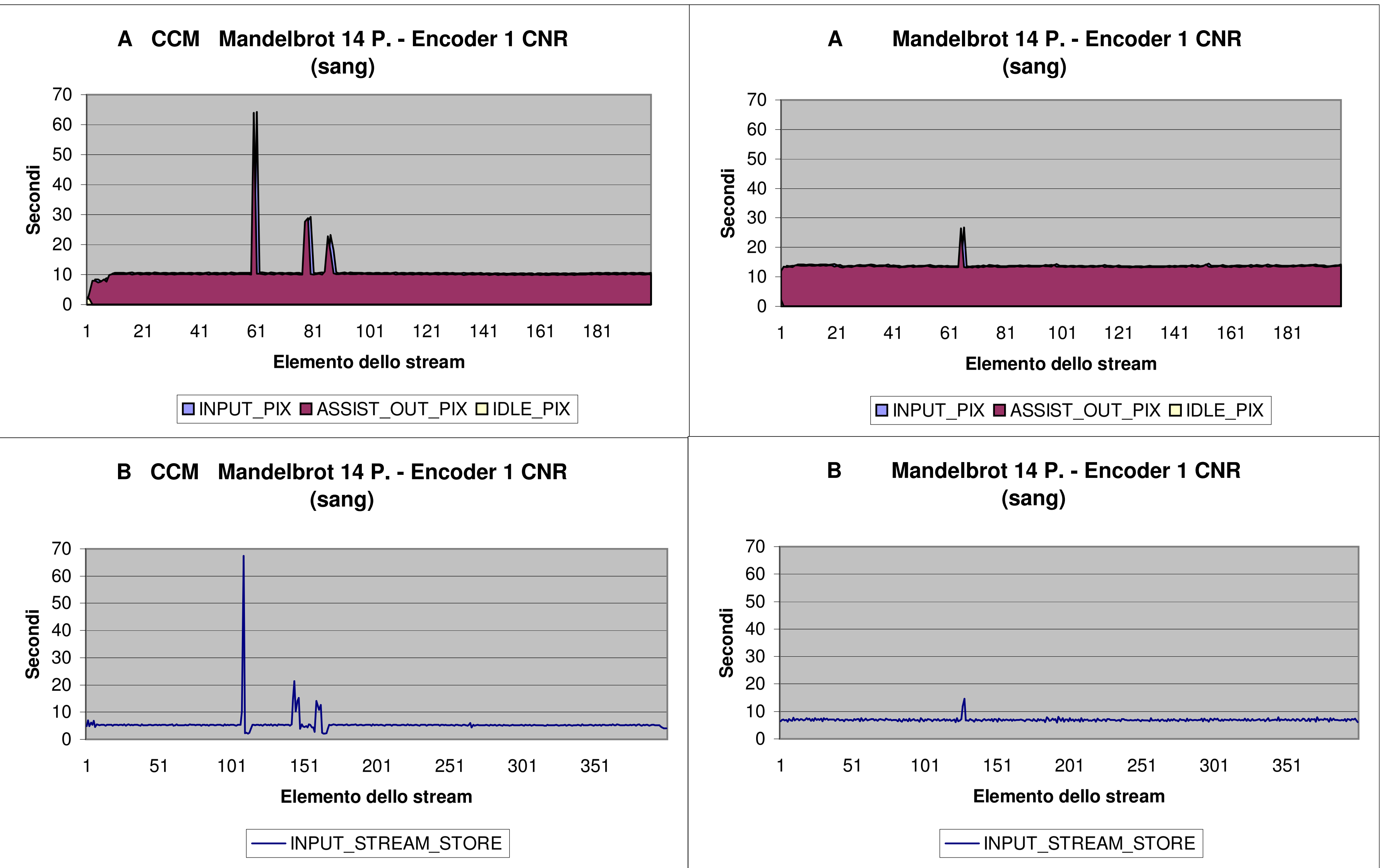

A Mandelbrot 4 P. - Encoder 1 CNR (sang)
Secondi
Elemento dello stream
INPUT_PIX
ASSIST_OUT_PIX
IDLE_PIX
A Mandelbrot 4 P. - Encoder 2 CNR (sang)
Secondi
Elemento dello stream
INPUT_PIX
IDLE_PIX
ASSIST_OUT_PIX
B Mandelbrot 4 P. - Encoder 1 CNR (sang)
Secondi
Elemento dello stream
INPUT_STREAM_STORE
B Mandelbrot 4 P. - Encoder 2 CNR (sang)
Secondi
Elemento dello stream
INPUT_STREAM_STORE

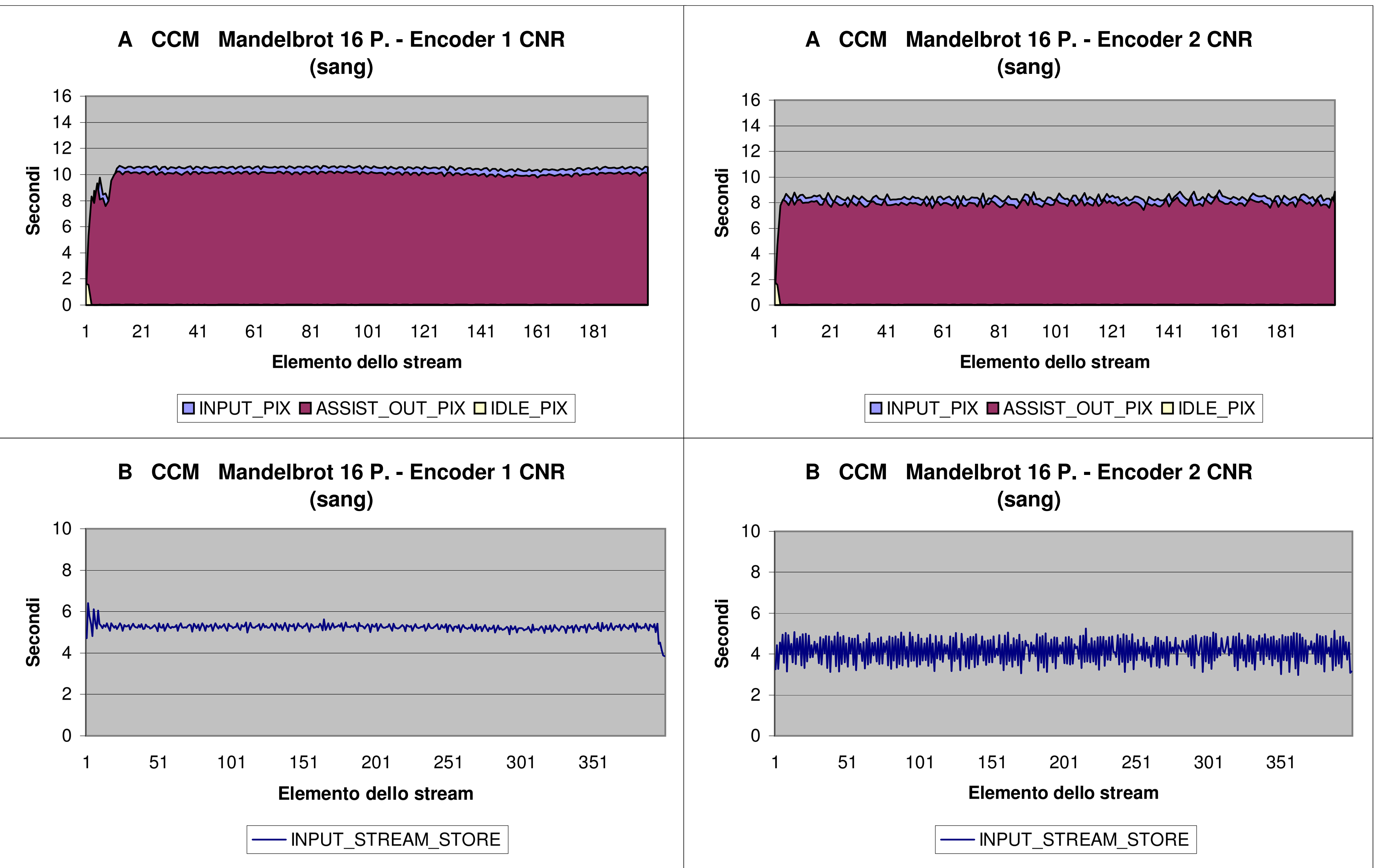

A CCM Mandelbrot 8 P. - Encoder 1 CNR (sang)
Secondi
Elemento dello stream
INPUT_PIX
ASSIST_OUT_PIX
IDLE_PIX
A CCM Mandelbrot 8 P. - Encoder 2 CNR (sang)
Secondi
Elemento dello stream
INPUT_PIX
ASSIST_OUT_PIX
IDLE_PIX
B CCM Mandelbrot 8 P. - Encoder 1 CNR (sang)
Secondi
Elemento dello stream
INPUT_STREAM_STORE
B CCM Mandelbrot 8 P. - Encoder 2 CNR (sang)
Secondi
Elemento dello stream
INPUT_STREAM_STORE

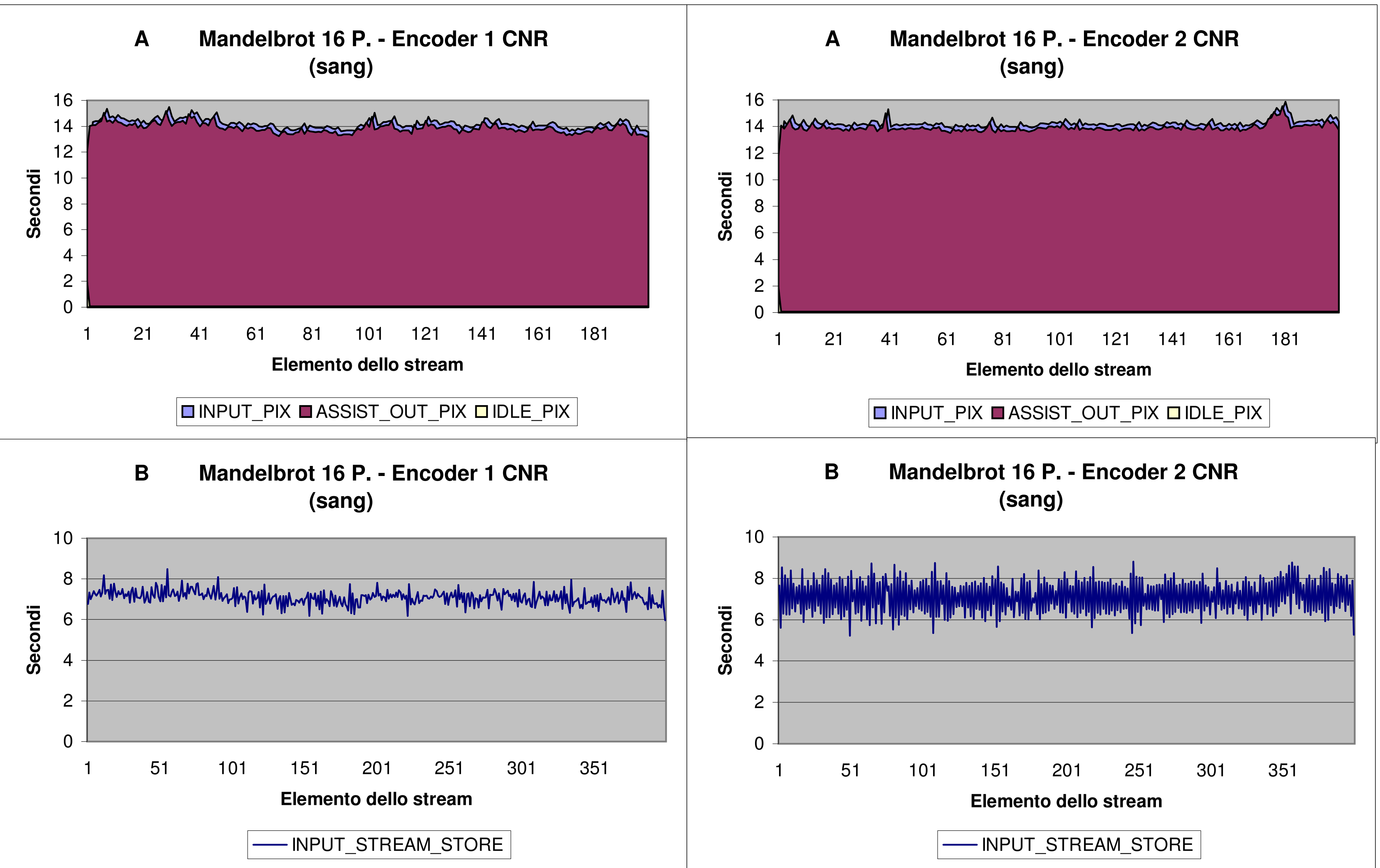

A
Mandelbrot 8 P. - Encoder 1 CNR
(sang)
Secondi
Elemento dello stream
INPUT_PIX
ASSIST_OUT_PIX
IDLE_PIX
A
Mandelbrot 8 P. - Encoder 2 CNR
(sang)
Secondi
Elemento dello stream
INPUT_PIX
ASSIST_OUT_PIX
IDLE_PIX
B
Mandelbrot 8 P. - Encoder 1 CNR
(sang)
Secondi
Elemento dello stream
INPUT_STREAM_STORE
B
Mandelbrot 8 P. - Encoder 2 CNR
(sang)
Secondi
Elemento dello stream
INPUT_STREAM_STORE


A CCM Mandelbrot 12 P. - Encoder 1 CNR
(sang)
Secondi
25
20
15
10
5
0
1
21
41
61
81
101
121
141
161
181
Elemento dello stream
INPUT_PIX
ASSIST_OUT_PIX
IDLE_PIX
A CCM Mandelbrot 12 P. - Encoder 2 CNR
(sang)
Secondi
Elemento dello stream
INPUT_PIX
ASSIST_OUT_PIX
IDLE_PIX
B CCM Mandelbrot 12 P. - Encoder 1 CNR
(sang)
Secondi
14
12
10
8
6
4
2
0
1
51
101
151
201
251
301
351
Elemento dello stream
INPUT_STREAM_STORE
B CCM Mandelbrot 12 P. - Encoder 2 CNR
(sang)
Secondi
Elemento dello stream
INPUT_STREAM_STORE


A Mandelbrot 12 P. - Encoder 1 CNR (sang)
Secondi
Elemento dello stream
INPUT_PIX
ASSIST_OUT_PIX
IDLE_PIX
A Mandelbrot 12 P. - Encoder 2 CNR (sang)
Secondi
Elemento dello stream
INPUT_PIX
ASSIST_OUT_PIX
IDLE_PIX
B Mandelbrot 12 P. - Encoder 1 CNR (sang)
Secondi
Elemento dello stream
INPUT_STREAM_STORE
B Mandelbrot 12 P. - Encoder 2 CNR (sang)
Secondi
Elemento dello stream
INPUT_STREAM_STORE


A CCM Mandelbrot 14 P. - Encoder 1 CNR
(sang)
Secondi
70 60 50 40 30 20 10 0
1 21 41 61 81 101 121 141 161 181
Elemento dello stream
INPUT_PIX ASSIST_OUT_PIX IDLE_PIX
A Mandelbrot 14 P. - Encoder 1 CNR
(sang)
Secondi
70 60 50 40 30 20 10 0
1 21 41 61 81 101 121 141 161 181
Elemento dello stream
INPUT_PIX ASSIST_OUT_PIX IDLE_PIX
B CCM Mandelbrot 14 P. - Encoder 1 CNR
(sang)
Secondi
70 60 50 40 30 20 10 0
1 51 101 151 201 251 301 351
Elemento dello stream
INPUT_STREAM_STORE
B Mandelbrot 14 P. - Encoder 1 CNR
(sang)
Secondi
70 60 50 40 30 20 10 0
1 51 101 151 201 251 301 351
Elemento dello stream
INPUT_STREAM_STORE


A CCM Mandelbrot 16 P. - Encoder 1 CNR (sang)
Secondi
Elemento dello stream
INPUT_PIX ASSIST_OUT_PIX IDLE_PIX
A CCM Mandelbrot 16 P. - Encoder 2 CNR (sang)
Secondi
Elemento dello stream
INPUT_PIX ASSIST_OUT_PIX IDLE_PIX
B CCM Mandelbrot 16 P. - Encoder 1 CNR (sang)
Secondi
Elemento dello stream
INPUT_STREAM_STORE
B CCM Mandelbrot 16 P. - Encoder 2 CNR (sang)
Secondi
Elemento dello stream
INPUT_STREAM_STORE


A Mandelbrot 16 P. - Encoder 1 CNR
(sang)
Secondi
Elemento dello stream
INPUT_PIX
ASSIST_OUT_PIX
IDLE_PIX
A Mandelbrot 16 P. - Encoder 2 CNR
(sang)
Secondi
Elemento dello stream
INPUT_PIX
ASSIST_OUT_PIX
IDLE_PIX
B Mandelbrot 16 P. - Encoder 1 CNR
(sang)
Secondi
Elemento dello stream
INPUT_STREAM_STORE
B Mandelbrot 16 P. - Encoder 2 CNR
(sang)
Secondi
Elemento dello stream
INPUT_STREAM_STORE

| | 1 | 2 | 4 | 6 | 8 | 10 | 12 | 14 | 16 |
|---|---|---|---|---|---|---|---|---|---|
| Parallelismo Mpeg 1 CNR | | | 0,04786 | | 0,062349 | | 0,061209 | 0,072378 | 0,070811 |
| Parallelismo Mpeg 2 CNR | | | 0,048188 | | 0,056922 | | 0,064023 | | 0,07058 |
| Parallelismo Mpeg 1 CNR CCM | | | 0,047826 | | 0,093064 | | 0,096996 | 0,091779 | 0,096918 |
| Parallelismo Mpeg 2 CNR CCM | | | 0,047818 | | 0,095509 | | 0,120336 | | 0,120701 |