Alma Mater Studiorum · Università di Bologna
Sede di Cesena

FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI
Corso di Laurea in Scienze dell'Informazione

SVILUPPO DI UN SOFTWARE PER L'ANALISI REAL-TIME DI DATI RADIOASTRONOMICI SU MACCHINE MULTICORE

Relazione Finale in Programmazione

Relatore:
Chiar.ma Prof.ssa ANTONELLA CARBONARO

Presentata da:
STÉPHANE BISINGER

Sessione Seconda
Anno Accademico 2009 – 2010

Indice

Elenco delle figure

Listings

Ringraziamenti

Questo lavoro è frutto di uno sforzo collettivo: numerose sono le persone incontrate lungo la strada che mi hanno portato fin qui, senza le quali niente di tutto questo esisterebbe. Innanzitutto vorrei ringraziare la professoressa Antonella Carbonaro per aver creduto nelle mie capacità, nonostante non avessimo avuto precedenti occasioni di conoscersi. Ringrazio anche l'Ing. Stelio Montebugnoli per avermi accolto nella sua struttura e per avermi permesso di lavorare in un ambiente stimolante ed interessante. Grande riconoscenza va anche al Dott. Marco Bartolini per avermi seguito riuscendo a trovare sempre tempo da dedicarmi. Un ulteriore ringraziamento a tutte le persone che ho incontrato nella stazione di Medicina che mi hanno fatto sentire di casa e condiviso con me qualche piatto di pasta.

Oltre alle persone che hanno contribuito direttamente, ci sono tutte quelle persone che mi hanno stimolato ad andare avanti e portare a termine questa grande esperienza che è stata l'università: ringrazio il Presidente Michele Munno, che è riuscito meglio di chiunque altro a stimolarmi e spingermi ad andare avanti, nonostante la mia innata pigrizia; ringrazio il Segretario Alice De Vecchi, che mi dà sempre la soddisfazione di farmi sentire un buon amico; il mio compagno di studi Simic, a cui auguro una lunga carriera universitaria; Nicola Munno e Marisa Mackiewicz, che mi fanno sempre sembrare un tipo puntuale; Michela Zanni, per la sua ostinazione a volermi bene nonostante il mio essere orso. Infine un ringraziamento sentito va alla mia famiglia, che in tutti questi anni non ha mai perso fiducia in me e ha fatto numerosi sacrifici per permettermi di studiare.

Vorrei ringraziare anche i vecchi compagni di università a Milano, numerosissimi, che hanno reso più sopportabile vivere in quella orribile città, e la professoressa Paola Campadelli, che con la passione e competenza sulla sua materia è riuscita a farmi comprendere ed apprezzare i concetti fondamentali necessari a questo tipo di lavoro.

Ringrazio chi mi ha portato a cambiare, senza cui sarei ancora un ragazzino impreparato.

Introduzione

L'analisi dei segnali è un aspetto fondamentale per la cultura moderna: grazie ad essa esistono telefoni, radio, televisioni, Internet e tanto altro. Grazie all'analisi dei segnali è possibile riprodurre, modificare e creare audio, immagini, video e tanto altro con l'utilizzo di un computer. Inoltre lo studio dei segnali elettromagnetici permette di capire molto meglio il mondo attorno a noi ed ha aperto nuove possibilità per lo studio dei corpi celesti: osservare un oggetto astronomico con il solo ausilio di strumenti ottici, infatti, non permette di cogliere moltissimi aspetti che possiamo dedurre dallo studio della radiazione elettromagnetica proveniente dalla sorgente e dalle zone limitrofe. Inoltre l'analisi dei segnali offre la possibilità di studiare tanti altri oggetti come buchi neri, stelle di neutroni, pulsar, l'effetto redshift e tanto altro. Lo studio di questi grandi oggetti astronomici ci permette anche di capire molto di più la natura della materia ed il suo comportamento in situazioni estreme.

L'analisi dei segnali con mezzi informatici è precedente all'introduzione di personal computers: gran parte di questo lavoro viene svolto su circuiti elettronici costruiti su misura per effettuare le operazioni necessarie, senza il bisogno di un calcolatore generico e di programmazione software. Questa soluzione, ancora largamente in uso al giorno d'oggi, permette di ottenere delle prestazioni altissime, ma ha il grande difetto di essere poco flessibile e molto costosa: una piccola modifica quasi insignificante all'algoritmo può significare ore e ore di lavoro specializzato. Per questo motivo si cercano al giorno d'oggi delle soluzioni software in grado di ottenere delle prestazioni sufficientemente buone da poter fare a meno di elaboratori DSP, così che si possa sostituire almeno in parte questo hardware specializzato con generici elaboratori. Il progetto sviluppato nasce proprio nell'ottica di ottenere buone prestazioni da un normale elaboratore allo scopo di abbattere i costi in alcuni campi di ricerca.

Stazione Radioastronomica di Medicina

Inaugurata nel 1964, la stazione "Croce del Nord" di Medicina (BO) ha sancito l'inizio dell'osservazione radioastronomica in Italia, essendo il primo

Figura 1: Veduta aerea della Croce del Nord e dell'antenna parabolica

osservatorio astronomico fornita di radiotelescopio invece di telescopi ottici. Un radiotelescopio, a differenza di un telescopio ottico, non osserva i corpi celesti tramite la parte visibile dello spettro elettromagnetico, ma li osserva attraverso la captazione di onde radio e la loro trasformazione in segnali elettrici. Questi segnali elettrici vengono poi elaborati per permettere l'analisi dello spettro elettromagnetico degli oggetti osservati. I radiotelescopi presenti a Medicina sono due: una è la "Croce del Nord", che dà il nome alla stazione, formata da due serie di antenne; l'altra è l'antenna parabolica, dal diametro di 32 metri.

La "Croce del Nord" è un array di antenne ad apertura, con specchio di forma cilidrico-parabolica, composto da un braccio orientato in direzione Est–Ovest ed un braccio orientato in direzione Nord–Sud. Il braccio E/W è costituito da 24 cilindri dal diametro di 29 metri e lunghi 24 metri ciascuno, mentre il ramo N/S è costituito da 64 cilindri di 7,5 metri di diametro e 22 metri di lunghezza, per un totale di 30000 m^2 di area collettrice, che rende la "Croce del Nord" uno dei più grandi radiotelescopi attualmente esistenti sul pianeta. La "Croce del Nord" lavora su segnali centrati a 408 Mhz, con larghezza di banda che varia dai 2,5 ai 16 Mhz. La sua sensibilità a segnali anche piuttosto deboli la rende molto utile nello studio di corpi molto distanti, al di fuori della nostra galassia. Potendo ruotare in una unica

direzione, questo radioscopio può osservare solamente segnali che transitano sulla verticale del meridiano in cui si trova, cioè è un radiotelescopio di *transito*.

L'antenna parabolica, invece, può essere orientata in tutte le direzioni e per questo motivo è possibile osservare tutti gli oggetti celesti presenti nel cielo visibile. Questa antenna, con un diametro di 32 metri, lavora con frequenze comprese tra i 327 Mhz ed i 43 Ghz. Ha due principali modalità operative: in rete o in *single dish*.

La parabola partecipa al progetto Very Large Baseline Interferometry (VLBI) che, tramite la rilevazione simultanea di diverse antenne paraboliche situate in posti diversi, permette di raggiungere una precisione maggiore: correlando i risultati dei diversi radiotelescopi si riesce ad ottenere una risoluzione equivalente ad una antenna di dimensione pari alla distanza delle antenne coinvolte. Le osservazioni particolarmente interessanti da effettuare in questa maniera sono due: la prima prevede l'osservazione di un stesso oggetto noto da parte di radiotelescopi situati su placche tettoniche diverse, in modo da studiare il fenomeno della deriva dei continenti: questo tipo di osservazioni non rientrano nell'astronomia, ma nella *geodinamica*, anche se sfrutta un radiotelescopio per fare i suoi rilevamenti. Il secondo tipo di osservazioni prevede l'osservazione della posizione e delle proprietà chimico-fisiche di un corpo celeste. Durante circa 4 mesi all'anno l'antenna parabolica di Medicina è riservata al progetto VLBI.

Nella modalità *single dish* l'antenna parabolica viene utilizzata in modo indipendente rispetto ad altre antenne, senza correlare i dati rilevati con altri dati. In questa modalità si effettuano ad esempio ricerche di comete e il progetto Search for Extraterrestrial Intelligence (SETI).

Il progetto SETI

Nel 1959 due fisici, Giuseppe Cocconi e Philip Morrison, pubblicarono su *Nature* un articolo [CM59] in cui ponevano le basi teoriche con cui condurre ricerche su forme di vita intelligenti extra-terrestri. Secondo questi due scienziati, se una civiltà extra-terrestre evoluta volesse mettersi in contatto con altri mondi, invierebbe segnali a banda stretta alla frequenza di 1420 Mhz: questa frequenza corrisponde all'emissione spontanea di radiazione degli atomi di idrogeno neutro, l'elemento più diffuso nell'universo e il più importante sotto molti aspetti di ricerca astronomica. Questa frequenza ha anche il vantaggio di essere poco soggetta al rumore di fondo presente nell'universo, quindi di riuscire a viaggiare a grande distanza. Inoltre un segnale a banda stretta è riconosibile molto facilmente rispetto ai segnali a banda larga emessi naturalmente dai corpi celesti, quindi la ricezione di un segnale a banda stretta ha buone probabilità di indicare una qualche emissione artificiale.

In seguito all'articolo di Cocconi e Morrison, nel 1960 Frank Drake puntò un radiotelescopio sulle stelle Tau Ceti ed Epsilon Erinadi: queste due stelle sono di dimensioni simili al sole ed hanno probabilmente un sistema planetario simile al sistema solare, quindi hanno una certa probabilità di aver sviluppato la vita. Benché la sua ricerca non produsse esiti positivi, essa generò entusiasmo nella comunità astronomica e diversi progetti simili furono avviati da appassionati durante i tempi liberi dei propri radiotelescopi. Nel 1961 Drake propose un'equazione per valutare quante forme di vita potrebbero essere presenti nell'universo:

$$N = R_* \cdot f_p \cdot n_e \cdot f_l \cdot f_i \cdot f_t \cdot L$$

Dove:

- R_* è il numero di stelle di tipo solare che nascono nell'universo ogni anno.
- f_p è la frazione di queste stelle che possiedono un sistema planerario.
- n_e è il numero di pianeti per ogni sistema che si trovano ad una distanza adeguata per renderli abitabili.
- f_l è la frazione di questi pianeti che hanno forme di vita, seppur primitive.
- f_i è la frazione di pianeti con forme di vita intelligente.
- f_t è la frazione di pianeti su cui l'avanzamento tecnologico permette la comunicazione con altri pianeti.
- L indica la durata media in anni di tali civiltà.

Nonostante le stime di Drake indichino numerose civiltà presenti nell'universo, questa equazione mostra anche i grandi problemi legati alla ricerca SETI: molte di queste variabili sono quasi impossibili da stimare basandosi sulla nostra conoscenza attuale dell'universo e molti altri fattori (come ad esempio la distanza dalla Terra) non sono inclusi. Nonostante questo il progetto prende piede e negli anni '70 anche la NASA inizia ad interessarsene, finché nel 1992 non lancia il proprio progetto SETI. In seguito la NASA se ne disinteresserà ed il progetto continuerà solamente grazie a finanziamenti privati ed il lavoro di volontari all'interno di osservatori radioastronomici.

Italian Search for Extraterrestrial Life (ITASEL)

In Italia, ed in particolare a Medicina, la ricerca di forme di vita extraterrestri avviene attraverso il progetto ITASEL. Questo progetto, finanziato dall'Agenzia Spaziale Italiana, ha lo scopo principale di sviluppare nuove

tecnologie di spettrometria radio per la rilevazione di acqua e molecole prebiotiche nelle comete e nelle atmosfere exoplanetarie. La scoperta di queste molecole indicherebbero una buona probabilità che la vita si sia sviluppata su altri pianeti, in quanto si ritiene fondamentale il ruolo dell'acqua allo stato liquido nella formazione e nel sostentamento della vita.

La strumentazione sviluppata nell'ambito del progetto ITASEL viene anche utilizzata per la partecipazione italiana nel progetto SETI.

Search for Extraterrestrial Radio from Nearby Developed Intelligent Population (SERENDIP)

Analizzare i segnali per il progetto SETI richiede un sistema informatico in grado di effettuare i calcoli e le trasformazioni necessarie. L'Università di Berkeley, California, ha sviluppato il sistema SERENDIP, attualmente alla versione IV, che permette di analizzare i segnali in modalità *piggy back*, cioè contemporaneamente ad altre elaborazioni senza modificare il segnale originario. In questo modo non è necessario dedicare del tempo di antenna al progetto SETI, ma è possibile fare le osservazioni durante altri progetti di ricerca senza interferenze: questo ha permesso di abbattere notevolmente i costi del progetto, oltre a sfruttare tutto il tempo di utilizzo dell'antenna. Con questo spettrometro è possibile analizzare segnali a 24 milioni di canali con larghezza di banda di 15 Mhz.

Il progetto

Il progetto consiste nello sviluppo di uno spettrometro, scritto nel linguaggio di programmazione C++, allo scopo di ottenere le migliori prestazioni possibili sfruttando unicamente un elaboratore generico in contrapposizione ad un processore DSP. Questo spettrometro può trovare applicazione in diversi ambiti di ricerca, incluso il progetto SETI, dove può essere paragonato alle prestazioni di SERENDIP IV.

Un requisito fondamentale di questo progetto è che sia sufficientemente modulare da permettere una sua estensione e per permettere l'introduzione di filtri e altre trasformazioni sui segnali senza dover riscrivere codice già esistente, ma sfruttando delle interfacce predefinite.

Per raggiungere questi scopi è stato dedicato ampio tempo alla progettazione del software, in modo che si potesse procedere allo sviluppo con le idee chiare sulla struttura da adottare nel progetto.

Nei capitoli seguenti verranno illustrati alcuni concetti matematici di base fondamentali per la comprensione del lavoro svolto, per poi procedere all'analisi della fase di progettazione e quella di sviluppo. Verranno poi mostrati i risultati raggiunti e tratte le relative conclusioni anche in relazione agli obiettivi posti. Saranno poi indicati alcuni dei possibili sviluppi

futuri, ottimi punti di partenza per altri lavori di tesi o di tirocinio per altri studenti.

Capitolo 1

Basi di elaborazione del segnale

In questo capitolo introdurremo alcuni concetti basilari sull'elaborazione dei segnali necessari alla comprensione del funzionamento di uno spettrometro e del suo campo di applicazione. Sapere cosa sia un segnale, come si estraggono ed elaborano le informazioni in esso contenuto o quali siano i concetti matematici utilizzati è fondamentale per capire il lavoro svolto.
Questo materiale introduttivo è sufficiente per avere una panoramica sui concetti teorici utilizzati.

1.1 Segnali

Il nostro corpo è in grado di percepire, attraverso i sensi, alcune delle variazioni nelle proprietà fisiche del mondo che ci circonda. Queste variazioni che percepiamo vengono elaborate dal nostro cervello che riesce a ricavarne informazioni utili: In questo modo siamo in grado di percepire se fa caldo o freddo, se c'è luce, se un oggetto è di un colore piuttosto che un altro, ecc. Va notato che in questi segnali c'è una componente fisica (la temperatura) e l'informazione veicolata (freddo/caldo). [BCR09]

Definizione 1.1. *Si dice segnale una qualunque quantità che varia nel tempo o nello spazio.*

Secondo la precedente definizione, quindi, è possibile che un segnale non contenga informazioni utili: in questo caso ci si riferisce al segnale come *rumore*. Il rumore è anche una componente di interferenza o errore su di un segnale che si cerca di interpretare.

Un segnale viene quindi rappresentato da una funzione $g = f(c)$, ove:

- g è una variabile dipendente su di una grandezza fisica in relazione al tempo o spazio.

- c è una varaibile indipendente che rappresenta lo spazio o il tempo.
- f è una funzione che associa ad un valore temporale o spaziale c la corrispondente quantità g.

Per semplicità di esposizione, d'ora in avanti si farà riferimento unicamente a segnali in relazione al tempo.

1.1.1 Segnali periodici

Una importante classe di segnali sono i segnali periodici. Un segnale è periodico se si ripete in un certo periodo di tempo: definendo T il periodo con cui si ripete il segnale e con t un qualunque momento nel tempo, per ogni k, vale $f(t) = f(t + kT)$. Un segnale periodico $f(t)$ è quindi univocamente individuato nell'intervallo $-\frac{T}{2} \leq t \leq \frac{T}{2}$.

Si definisce *frequenza* di un segnale periodico $f(t)$ il numero di ripetizioni del periodo T nell'unità di tempo. Risulta quindi:

$$\nu = \frac{1}{T}$$

L'unità di misura della frequenza è l'Hertz ed indica il numero di cicli (ripetizioni) al secondo.

1.1.2 Frequenza radio

I segnali elettromagnetici vengono classificati a seconda della loro frequenza in diverse categorie:

- Frequenze minori di 3 GHz vengono denominate *frequenze radio*.
- Frequenze comprese tra 3 GHz e 300 GHz sono denominate *microonde*.
- Frequenze comprese tra 300 GHz e 400 000 GHz (400 THz) sono denominate *infrarossi*.
- Frequenze comprese tra 400 THz e 750 THz sono parte dello *spettro visibile*.
- Frequenze comprese tra 750 THz e 30 000 THz (30 PHz) sono denominate *raggi ultravioletti*.
- Frequenze comprese tra 30 Phz e 30 000 PHz (30 EHz) sono denominate *Raggi X*.
- Frequenze superiori ai 30 EHz sono denominate *Raggi Gamma*.

Va notato che questi limiti sono solo convenzionali e non hanno una definizione standard.

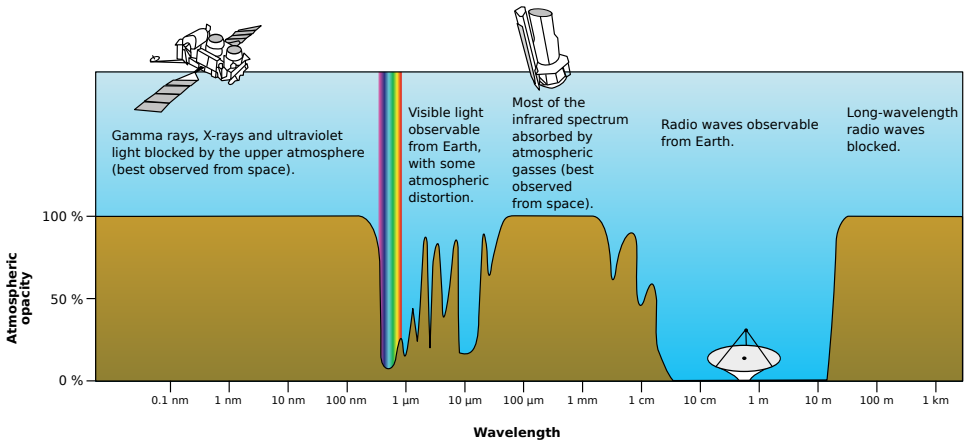

Figura 1.1: Permeazione dei segnali elettromagnetici nell'atmosfera

Studio dei corpi celesti

I segnali elettromagnetici sono fondamentali per lo studio dei corpi celesti. Tuttavia va considerata la permeabilità dell'atmosfera terrestre rispetto ai segnali che ci arrivano dallo spazio, come illustrato in Figura 1.1[1]: come si può notare, lo studio dei corpi celesti da terra è possibile solo nello spettro visibile (telescopi, con qualche distorsione) oppure nello spettro delle frequenze radio (radiotelescopi). Per altre lunghezze d'onda, è necessario studiare questi segnali direttamente dallo spazio con l'ausilio di satelliti. I moderni radiotelescopi lavorano tra i 70 Mhz ed i 43 Ghz, a seconda del radiotelescopio, quindi anche le microonde rivestono una qualche importanza nello studio dei corpi celesti. Questi tipi di segnale sono oggetto di elaborazione del software sviluppato.

1.2 ADC: Analog–to–Digital Conversion

Per interpretare i segnali del mondo che ci circonda, possiamo sfruttare la velocità di calcolo di un elboratore; tuttavia per poterlo usare, bisogna essere in grado di rappresentare le informazioni che riceviamo in modo che l'elaboratore sia in grado di capirle. Per questo motivo dobbiamo trovare un modo di trasformare un segnale *analogico* in un segnale *digitale*.

1.2.1 Segnali analogici, segnali digitali

Come abbiamo visto nella definzione 1.1, i segnali hanno due componenti: il tempo e il valore assunto dalla grandezza fisica che osserviamo. Queste

[1] Fonte: `http://coolcosmos.ipac.caltech.edu/cosmic_classroom/cosmic_reference/irwindows.html`

due componenti possono assumere un qualunque valore reale, cioè il segnale ha *tempo continuo* e *valori continui*. Tuttavia, in questo modo il segnale non può essere rappresentato da un elaboratore, in quanto un elaboratore può contenere solamente quantità finite, mentre le componenti del segnale sono quantità infinite. Per questo motivo bisogna cercare di far rientrare il segnale in un range di valori finiti e renderlo così rappresentabile da un calcolatore.

Per rendere finita la misurazione del tempo, si possono salvare i valori rilevati ad un intervallo prefissato, ad esempio una volta ogni secondo. Misurando un segnale per 10 secondi si otterranno così 10 valori associati ad ogni secondo. Un segnale di questo tipo ha *tempo discreto*, ma valori ancora *continui*. Un dispositivo che raccoglie valori con una determinata frequenza si chiama *campionatore*.

Per rendere i valori finiti, si può scegliere di utilizzare un certo insieme di grandezze, ad esempio 1 e -1, e per ogni valore rilevato, associare la grandezza più vicina. Quindi se rileviamo i valori 4, 10, -5 e -13, salviamo il segnale con i valori 1, 1, -1, -1. Così facendo, il segnale assume *valori finiti* che sono rappresentabili in un elaboratore, nell'esempio fatto con l'uso di 1 bit. Un dispositivo che associa ai valori rilevati il più vicino valore rappresentabile dall'elaboratore si chiama *quantizzatore*.

Un segnale che ha tempo *continuo* e valori *continui* si chiama *segnale analogico*, mentre un segnale con tempo *discreto* e valori *finiti* si chiama *segnale digitale*.

Come è facile intuire, la conversione da analogico a digitale può introdurre degli *errori*, cioè del *rumore*, in quanto la versione digitale di un segnale analogico è una approssimazione. Fortunatamente è possibile valutare questo margine di errore e ridurlo a seconda delle necessità sia aumentando il numero di bit usati per rappresentare i valori nel tempo, sia aumentando il numero di rilevazioni effettuate nello spazio temporale di osservazione.

1.2.2 Teorema del Campionamento

Quando si effettua il campionamento di un segnale $f(t)$, si sceglie un periodo τ e si fanno n misurazioni nel tempo, ottenendo quindi un segnale di tipo $f(n\tau)$. Come possiamo assicurarci che da $f(n\tau)$ sia possibile ricostruire il segnale originale $f(t)$? Il teorema del campionamento fornisce una risposta definendo un limite basato sulla frequenza massima del segnale:

Teorema 1.1. *Un segnale $f(t)$ con frequenza massima ν_m può essere ricostruito dal segnale campionato $s(n\tau)$ con frequenza $\nu_s = \frac{1}{\tau}$ se $\nu_s > 2\nu_m$*

Cioè la frequenza di campionamento deve essere doppia rispetto alla frequenza massima presente nel segnale.[2] Nel caso in cui un segnale venga campionato con frequenza di campionamento $\nu_s < 2\nu_m$, si verifica un

[2]Per una dimostrazione del teorema del campionamento, fare riferimento a [Smi07]

fenomeno di *aliasing*, cioè le componenti a frequenza maggiore di $\frac{\nu_s}{2}$ verranno ricostruite come se avessero un'altra frequenza compresa tra 0 e $\frac{\nu_s}{2}$; a questo modo l'intero segnale verrebbe compromesso e sarebbe impossibile recuperare il segnale originario. Per evitare che ciò accada si applica un filtro al segnale che deve essere campionato per eliminare tutte le frequenze maggiori di $\frac{\nu_s}{2}$, evitando che frequenze non di nostro interesse possano interferire con le frequenze che andremo ad elaborare.

1.3 La Trasformata Discreta di Fourier

1.3.1 Introduzione alla Trasformata di Fourier

Jean Baptiste Joseph Fourier, vissuto a cavallo tra il XVIII e XIX secolo, era un matematico francese che, presentando i suoi studi sulla propagazione del calore, affermava che qualunque segnale periodico continuo potesse essere rappresentato come somme pesate di sinusoidi. Questa teoria fu molto osteggiata da Joseph Louis Lagrange, il quale sosteneva che per segnali contenenti angoli, come ad esempio le onde quadre, ciò non fosse possibile e per questo il lavoro di Fourier non fu pubblicato per 15 anni, fino alla morte dello stesso Lagrange.[3] In realtà l'approccio di Fourier non era sbagliato, ma anche l'obiezione di Lagrange è valida: se da un lato è vero che un segnale con angoli non potesse essere rappresentato come somme di sinusoidi, si riesce ad ottenere una approsimazione così accurata del segnale da avere una differenza energetica pari a zero. Questo fenomeno è conosciuto come *Fenomeno di Gibbs*[4]

1.3.2 Tipi di Trasformate di Fourier

Esistono quattro tipi di segnali possibili, ognuno con la sua trasformata di Fourier. Un segnale può essere periodico o aperiodico, continuo o discreto. Ne conseguono i quattro tipi di trasformate:

- **Aperiodico - Continuo**: Il segnale non è periodico ed assume infiniti valori. Per analizzarlo si utilizza la *Trasformata di Fourier*.
- **Periodico - Continuo**: Il segnale assume infiniti valori che si ripetono con un certo periodo. In questo caso si usa la *Serie di Fourier*.
- **Aperiodico - Discreto**: Il segnale non ha periodo e assume solo un numero finito di valori. In questo caso si utilizza la *Trasformata di Fourier a tempo discreto*.

[3] cfr. [Smi97], capitolo 8
[4] cfr. [Smi97], capitolo 11

- **Periodico - Discreto**: Il segnale assume un numero finito di valori che si ripetono con un dato periodo. La trasformata utilizzata è la *Trasformata discreta di Fourier*.

Ogni segnale usato per queste trasformate deve essere infinito; nel caso in cui ci fosse un segnale discreto con un numero di punti finito, come succede quando abbiamo un segnale rappresentato all'interno di un elaboratore, si può immaginare questo segnale come se fosse infinito. Se si immagina che prima e dopo del segnale in nostro possesso il segnale abbia un valore pari a 0, si userà la trasformata di Fourier a tempo discreto, altrimenti se si immagina il segnale infinito come una ripetizione dei dati in nostro possesso, si utlizza la trasformata discreta di Fourier. Purtroppo, un qualunque segnale aperiodico ha infinite componenti sinusoidali, quindi la trasformata di Fourier a tempo discreto non è utilizzabile con un elaboratore, lasciando unicamente la trasformata discreta di Fourier a nostra disposizione.

Ognuna delle quattro trasformate può essere eseguita nella sua forma complessa, dove un insieme di numeri complessi vengono trasformati in un'altro insieme di numeri complessi, oppure nella forma reale, dove un insieme di numeri reali vengono trasformati in due insiemi di numeri reali. La forma complessa di una trasformata è molto più difficile da comprendere, perciò verrà esposta unicamente la forma reale della trasformata. Analizzeremo unicamente la versione discreta della trasformata di Fourier in quanto è di più semplice comprensione, pur essendo concettualmente identica agli altri tipi di trasformata, oltre ad essere la trasformata utilizzata direttamente nel progetto.

1.3.3 Analisi e Sintesi

Quando applichiamo la trasformata di Fourier, non facciamo altro che passare tra due rappresentazioni diverse dello stesso segnale: il segnale originario è nel dominio del tempo, mentre il segnale trasformato è nel dominio delle frequenze. Esiste anche la trasformata inversa, che permette di passare dal dominio della frequenza al dominio del tempo. L'operazione che trasforma un segnale nel dominio del tempo ad uno nel dominio delle frequenze si chiama *analisi*, *decomposizione* o semplicemente trasformata DFT, mentre l'operazione inversa si chiama *sintesi* o trasformata inversa. Entrambe le operazioni possono essere espresse come algoritmi ed utilizzate da elaboratori.

Quando rappresentiamo un segnale in un elaboratore, avremo un vettore, `x[]`, contente le ampiezze del segnale nei vari intervalli di tempo. Se al momento del campionamento abbiamo scelto di misurare N campioni, avremo ora un vettore di N elementi che vanno da 0 a N. Di solito per facilitare

la computazione tramite elaboratore N viene scelto tra le potenze di 2.[5] La controparte di questo segnale è suddivisa in due parti, `ReX[]` contenente i valori per le ampiezze dei coseni presenti nel segnale, e `ImX[]` contenente i valori per le ampiezze dei seni presenti nel segnale.[6] I due vettori `ReX[]` e `ImX[]` contengono $N/2+1$ valori ciascuno, partendo da 0 e arrivando a $N/2$.

1.3.4 Funzioni base della DFT

Quando effettuiamo la DFT di un segnale, otteniamo due vettori che rappresentano l'ampiezza delle funzioni di base della DFT. Queste funzioni base sono le funzioni coseno e seno di diversa frequenza e ampiezza unitaria che fanno parte del segnale. Quindi avremo che:

$$c_k[i] = \cos\left(\tfrac{2\pi ki}{N}\right)$$
$$s_k[i] = \sin\left(\tfrac{2\pi ki}{N}\right)$$
$$i = 0 \dots N-1, \quad k = 0 \dots N/2$$

Al variare di k, varia la frequenza della sinusoide negli N punti: per $k = 0$ avremo 0 cicli negli N punti, per $k = 1$ avremo 1 ciclo, per $k = 2$ ci saranno due cicli e così via. Ad ogni c_k ed s_k, si associano le relative ampiezze `ReX[k]` ed `ImX[k]`.

Tre aspetti importanti vanno notati:

- $\boldsymbol{c_0}$ ha valore 1 per tutti i punti N: $\cos\left(\tfrac{2\pi 0i}{N}\right) = \cos(0) = 1, \quad \forall i \in (0 \dots N-1)$
- $\boldsymbol{s_0}$ ha valore 0 in tutti i punti N: $\sin\left(\tfrac{2\pi 0i}{N}\right) = \sin(0) = 0, \quad \forall i \in (0 \dots N-1)$
- $\boldsymbol{s_{N/2}}$ ha valore 0 in tutti i punti N: $\sin\left(\frac{2\pi(N/2)i}{N}\right) = \sin(\pi i) = 0, \quad \forall i \in (0 \dots N-1)$

In particolare si nota che i valori `ImX[0]` e `ImX[N/2]` non influiscono in nessun modo, quindi vengono generalmente impostati a 0.

1.3.5 DFT inversa

Per ottenere il segnale originario dal segnale nel dominio delle frequenze basta effettuare la somma dei coseni e dei seni pesati:

$$x[i] = \sum_{k=0}^{N/2} Re\bar{X}[k] \cos\left(\frac{2\pi ki}{N}\right) + \sum_{k=0}^{N/2} Im\bar{X}[k] \sin\left(\frac{2\pi ki}{N}\right)$$

[5] La Fast Fourier Transform (FFT) richiede segnali che abbiano lunghezza pari ad una potenza di due, per poter funzionare.

[6] Le notazioni `ReX[]` e `ImX[]` significano, rispettivamente, "parte reale di X" e "parte immaginaria di X". Questi nomi provengono dalla versione complessa della trasformata, ma in questo caso i due vettori vanno considerati semplicemente come ampiezze dei coseni e dei seni presenti nel segnale.

Nell'equazione si usano le versioni normalizzate dei vettori: `ReX[]` e `ImX[]`:

$$Re\bar{X}[k] = \frac{ReX[k]}{N/2}$$
$$Im\bar{X}[k] = -\frac{ImX[k]}{N/2}$$

Ci sono due casi speciali:

$$Re\bar{X}[0] = \frac{ReX[0]}{N}$$
$$Re\bar{X}[N/2] = \frac{ReX[N/2]}{N}$$

Questa normalizzazione viene introdotta perché il dominio delle frequenze è definito come densità spettrale, cioè le frequenze sono divise in bande e ogni valore `ReX[]` e `ImX[]` rappresenta la quantità di segnale (ampiezza) presente per ogni unità dell'ampiezza di banda: ad esempio il valore `ReX[1]` rappresenta l'ampiezza del segnale nella banda 1, che corrisponde all'intervallo $0,5 \ldots 1,5$, `ReX[2]` rappresenza l'ampiezza del segnale nella banda 2 $(1,5 \ldots 2,5)$ e così via. Ci sono $N/2+1$ bande, quindi ogni banda è larga $\frac{1}{N/2} = \frac{2}{N}$, tranne la prima e l'ultima: la banda 0 $(0 \ldots 0.5)$ e la banda $N/2$ $(N/2 - 0,5 \ldots N/2)$ sono larghe esattamente la metà, $\frac{1}{N}$.

1.3.6 Calcolo della DFT

Il primo metodo per calcolare la DFT ha solo lo scopo di far comprendere il procedimento, ma non ha nessuna applicazione pratica in quanto troppo complesso a livello computazionale. Dato un segnale `x[]` per calcolare la sua trasformata possiamo creare N equazioni semplicemente prendendo il primo punto delle sinusoidi di base e sommarli tra loro. Sappiamo che la loro somma, moltiplicata per i rispettivi pesi, devono essere uguali al primo punto del segnale originario. Andando a creare delle equazioni simili per gli altri punti, otterremo le N equazioni di cui abbiamo bisogno e potremo risolverle usando un qualunque metodo algebrico, ad esempio il metodo di eliminazione di Gauss. Supponiamo che il nostro segnale abbia 4 punti:

$$Re\bar{X}[0] + Re\bar{X}[1] + Re\bar{X}[2] = x[0]$$
$$Re\bar{X}[0] + Re\bar{X}[1]\cos\left(\frac{\pi}{2}\right) + Re\bar{X}[2]\cos(\pi) + Im\bar{X}[1]\sin\left(\frac{\pi}{2}\right) = x[1]$$
$$Re\bar{X}[0] + Re\bar{X}[1]\cos(\pi) + Re\bar{X}[2]\cos(2\pi) + Im\bar{X}[1]\sin(\pi) = x[2]$$
$$Re\bar{X}[0] + Re\bar{X}[1]\cos\left(\frac{3\pi}{2}\right) + Re\bar{X}[2]\cos(3\pi) + Im\bar{X}[1]\sin\left(\frac{3\pi}{2}\right) = x[3]$$

Ci ritroviamo con 4 variabili e 4 equazioni linearmente indipendenti, quindi siamo in grado di risolvere queste equazioni e trovare i valori delle variabili. Purtroppo questo metodo di risoluzione non è efficiente e non viene usato per calcolare la trasformata con l'ausilio di un elaboratore.

1.3.7 FFT: Fast Fourier Transform

Quando si sfrutta un elaboratore per fare dei calcoli, è importante valutare l'efficienza dell'algoritmo utilizzato, cioè valutare quanto tempo impiega il nostro algoritmo per eseguire il calcolo e di quanto cresce il tempo di esecuzione all'aumentare dei dati inseriti in ingresso. Il calcolo della DFT tramite correlazione[7] richiede un tempo di esecuzione all'incirca di $k_{DFT}N^2$. k_{DFT} è una costante di proporzionalità rispetto all'algoritmo che con un processore a 100Mhz equivale a 7 microsecondi, con tutte le ottimizzazioni possibili. Considerando un segnale di $N = 1024$ punti il tempo di esecuzione è di circa 7 secondi. Questo tempo è enorme, soprattutto considerando la piccola dimensione del segnale; per questo motivo è fondamentale trovare un algoritmo più efficiente nel calcolo della trasformata di Fourier. La *Fast Fourier Transform* è un algoritmo che, utilizzando un approccio completamente diverso al calcolo della trasformata, riesce ad ottenere un tempo di esecuzione pari a $k_{FFT}N\log_2 N$. In questo caso, k_{FFT} equivale a 10 microsecondi su processore a 100Mhz, quindi un segnale di $N = 1024$ punti viene trasformato in 70 millisecondi.[8] Comparando i due tempi si capisce che la FFT permette di effettuare calcoli anche su segnali di dimensioni abbastanza grandi, poiché la crescita logaritmica assicura che il tempo di calcolo non aumenti troppo bruscamente all'aumentare dei punti.

Un altro vantaggio non indifferente della FFT è la sua scarsa sensibilità agli errori di arrotondamento: poiché l'algoritmo diminuisce drasticamente il numero di operazioni da effettuare sui numeri, anche gli errori di arrotondamento rimangono molto contenuti. Questo assicura una buona stabilità numerica dell'algoritmo anche con grandi moli di dati.

[7] cfr. [Smi97], capitoli 6—8

[8] Per i dati dei test, cfr. [Smi97], capitolo 12

Capitolo 2

Progettazione e architettura

Nel momento della progettazione, sono stati prefissati alcuni punti importanti: il programma avrebbe dovuto essere veloce, modulare e parametrizzabile sulle caratteristiche più rilevanti dei segnali. L'utilizzo del C++ come linguaggio di sviluppo ha portato alla scelta di scrivere preferibilmente codice orientato agli oggetti e di sfruttare al massimo gli strumenti offerti dalla libreria boost[1], praticamente uno standard per qualunque programma C++ di una certa complessità.

2.1 Obiettivi

Il funzionamento desiderato per l'applicazione è piuttosto semplice: il programma legge dei dati da una sorgente, li trasforma con una FFT ed eventualmente applica qualche filtro, infine scrive l'output su una qualche destinazione. Un importante obiettivo è che la manipolazione dei segnali avvenga con elaborazione lossless, cioè che le operazioni vengano calcolate abbastanza velocemente da riuscire ad elaborare i dati man mano che arrivano dalla sorgente.

Uno degli utilizzi principali ipotizzati per lo spettrometro è l'elaborazione di dati per il progetto SETI. In questo caso, i dati vengono recuperati da una speciale scheda di acquisizione dedicata[2]. Le caratteristiche fondamentali della FFT sono una lunghezza del segnale piuttosto importante (circa 2^{23}), un basso numero di integrazioni e la possibilità di ricevere i dati sporadicamente, cioè il data rate può essere mantenuto basso.

[1] `http://www.boost.org/`

[2] L'hardware necessario a sviluppare questo aspetto dell'applicazione non era disponibile, perciò l'implementazione dell'acquisizione di dati SETI è stata lasciata come possibile sviluppo futuro. cfr. 5.2.1

Un'altro utilizzo è l'analisi di *space debris*[3] dove la sorgente di dati sono dei pacchetti UDP e la FFT è caratterizzata da segnali di lunghezza contenuta (circa 2^{11}) con poche integrazioni (circa 100), quindi una frequente scrittura su disco e tanti dati per periodi di tempo anche brevi. In questo caso è importante riuscire a mantenere il data rate alto, quindi l'elaborazione deve essere il più veloce possibile.

2.2 Gerarchia di classi

Per raggiungere gli obiettivi prefissati, sono necessarie alcune classi per astrarre alcuni concetti. In particolare ci sono tre componenti fondamentali:

- Una sorgente da cui prendere dati
- Filtri o trasformazioni da applicare al segnale
- Una destinazione su cui scrivere l'output

Per questo motivo ci sono tre classi astratte che rappresentano questi tre concetti, oltre a delle classi utili come contenitori di dati. Insieme formano le basi su cui viene implementato l'algoritmo rappresentato in figura 2.1.

2.2.1 SourceFilter

Una sorgente ha un unico metodo assolutamente necessario che serve a leggere i dati. Una sottoclasse di `SourceFilter`, ad esempio, è il client UDP (`udp_sock`), che può essere usata da qualunque codice generico che faccia riferimento unicamente all'interfaccia astratta. Se si creasse un'altra sottoclasse di `SourceFilter` e la si sostituisse a `udp_sock` si otterrebbero i dati da un'altra fonte senza dover cambiare una riga di codice al di fuori della inizializzazione.

2.2.2 ProcessFilter

Un `ProcessFilter` rappresenta una qualunque elaborazione intermedia del segnale, sia essa una trasformata di Fourier o un filtro passa-basso o una qualunque altra manipolazione. Siccome in questa fase si prende un segnale e lo si manipola in qualche maniera, l'interfaccia richiede un metodo `transform` che abbia in input i dati da elaborare e scriva su un vettore di output specificato. I filtri sono per natura concatenabili, quindi per gestire questa coda serve una classe `FilterChain` che si occupi di passare l'elaborazione attraverso una coda di filtri precedentemente selezionati.

[3] Gli space debris, letteralmente detriti spaziali, sono oggetti di media e piccola dimensione generalmente di origine artificiale che orbitano a distanza ravvicinata dalla Terra: si tratta generalmente di vecchi satelliti e altre apparecchiature abbandonate. Siccome possono creare problemi di varia natura, vanno tenuti sotto osservazione.

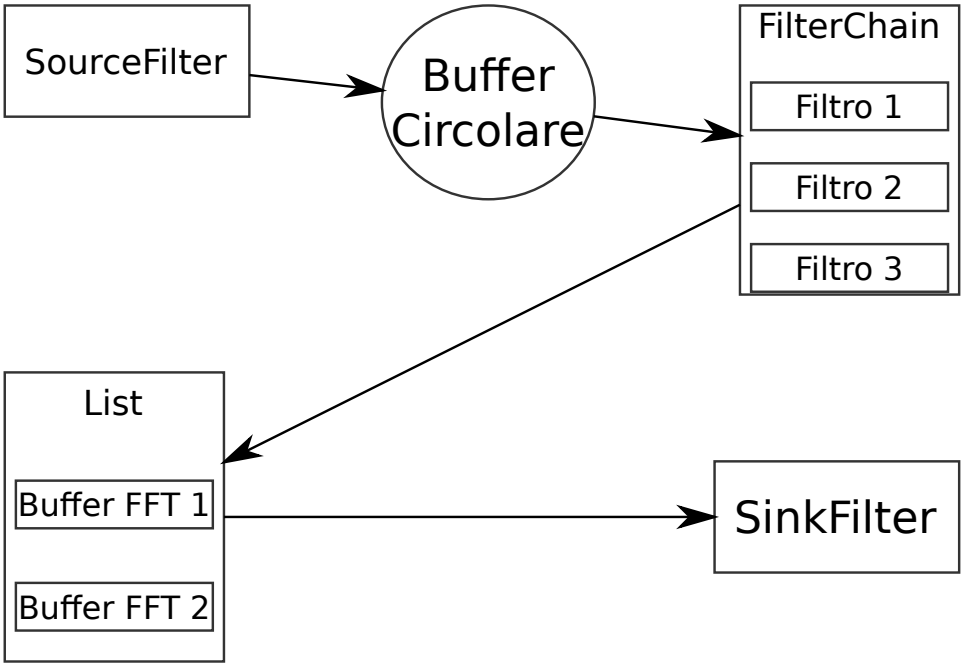

Figura 2.1: Flusso dei dati all'interno del programma

2.2.3 SinkFilter

Le destinazioni hanno bisogno di un unico metodo di interfaccia: `write`. In modo simile al `SourceFilter`, una qualunque implementazione della classe astratta `SinkFilter` può essere intercambiata con un'altra senza dover apportare modifiche al codice, permettendo di scegliere il dispositivo di memorizzazione più adeguato agli scopi perseguiti.

2.2.4 Contenitori di dati

Sono state progettate alcune classi al fine di astrarre i dati e assicurarne la corretta sincronizzazione.

SrcType

`SrcType` è una struttura con un costruttore, un destruttore ed un operatore di assegnamento. Tra i membri di questa struttura c'è una mutex per eseguire operazioni in mutua esclusione, ove necessario, permettendo un utilizzo thread safe dei dati. Questa struttura è un wrapper di un puntatore, ideale per essere usata all'interno di un circular buffer.

FFTBuf

La classe `FFTBuf` è un vettore su cui vanno scritti i risultati di una o più trasformazioni di Fourier. Al momento della costruzione, si indica la lunghezza del segnale da salvare e il numero di integrazioni che si prevede di fare; a questa maniera è possibile assegnare questo buffer a tanti thread quante dovranno essere le integrazioni da fare. La classe offre strumenti anche per tener traccia di quante integrazioni sono già state effettuate e sapere così se il buffer è pronto per essere scritto su output.

List

La classe `List` è un semplice wrapper attorno ad una lista della libreria standard. Questa classe assicura un comportamento corretto in ambienti multithreading e aggiunge alcuni metodi per semplificare la coordinazione con gli altri thread.

2.2.5 Wrapper di librerie

La classe `IPP` esiste solo per fornire un wrapper C++ alle librerie IPP, scritte in C. Tramite overloading è stato possibile utilizzare lo stesso nome per funzioni che operano su tipi di dati differenti, lasciando al compilatore il compito di invocare la funzione specifica più adeguata.

2.3 Threading

Per incrementare le prestazioni soprattutto su elaboratori forniti di processore multi-core o di più processori, il programma è stato progettato per funzionare con differenti thread di esecuzione per le diverse parti. Questo permette di sfruttare al massimo il processore senza dover attendere sui tempi di lettura dei dati in ingresso (ad esempio, una rete locale) o sui tempi di scrittura dei dati in uscita (ad esempio, un file locale). Inoltre è possibile eseguire più di una trasformazione in parallelo, utile ad esempio nel caso in cui si voglia sommare tra loro diversi segnali trasformati allo scopo di attutire il rumore di fondo.

2.3.1 Inizializzazione

All'avvio dell'applicazione vengono lette le opzioni passate da linea di comando, inizializzando le variabili con valori appropriati ed allocando alcuni buffer. In seguito viene inizializzato un buffer circolare utilizzato per i dati di input, una lista per i dati di output, un thread pool il cui compito sarà calcolare le trasformazioni descritte dalla FilterChain ed un thread che si occupa di scrivere l'output.

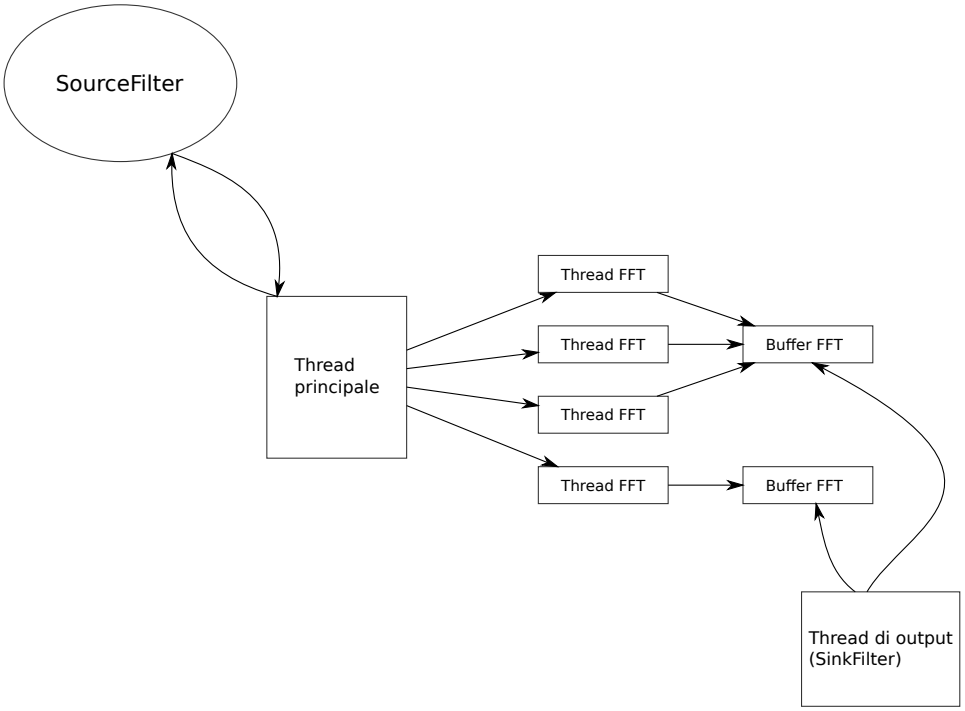

Figura 2.2: Interazione dei diversi thread tra loro.

2.3.2 Thread di output

Il thread di output controlla i dati presenti in una lista, fornita durante l'inizializzazione, dove verranno accodati i dati da scrivere in uscita. Quando la lista è vuota, il thread si mette semplicemente in attesa, altrimenti procede alla scrittura, rimuove dalla lista i dati scritti e ricomincia il proprio ciclo da capo.

2.3.3 Thread principale

Avviato il thread di output, il thread principale entra nel suo ciclo principale, dove effettua tre operazioni che esploriamo nel dettaglio: *lettura dei dati*, *selezione del buffer di output* e *assegnazione al thread pool*.

Lettura dei dati

La prima operazione effettuata all'interno del ciclo principale è verificare che ci sia dello spazio disponibile nel buffer circolare. Se il buffer è pieno — cioè se tutti i dati nel buffer devono ancora essere elaborati — il thread si mette in attesa che il carico di lavoro venga smaltito. Quando c'è dello

spazio disponibile nel buffer circolare, si leggono i dati dal SourceFilter e si accodano. Si procede poi alla selezione del buffer di output.

Selezione del buffer di output

Una volta ottenuti i dati da elaborare, si verifica se esistono buffer accodati nella lista di output. Se non ce ne sono, si crea un nuovo buffer, lo si accoda alla lista e lo si seleziona come buffer di output; se, invece, la lista contiene dei buffer si seleziona il buffer di testa e si verifica se è utilizzabile o meno: siccome allo scopo di attutire il rumore di fondo in un segnale spesso di fanno le somme di diversi segnali trasformati, esiste un parametro che determina quante di queste somme si vogliono effettuare. Per ogni buffer si tiene traccia del numero di thread a cui è stato assegnato e se questo numero è minore rispetto al numero di somme desiderate, il buffer è selezionabile, altrimenti se ne crea uno nuovo e lo si accoda in lista.

Assegnazione al thread pool

Dopo aver letto i dati del segnale da trasformare ed aver individuato il buffer di output su cui scrivere i dati elaborati, si può assegnare il lavoro al thread pool. Per farlo, è sufficiente indicare i puntatori ai due buffer ed un puntatore alla funzione che deve occuparsi di elaborare i dati. Il thread pool si occuperà automaticamente di assegnare il lavoro al primo thread libero presente nel suo pool.

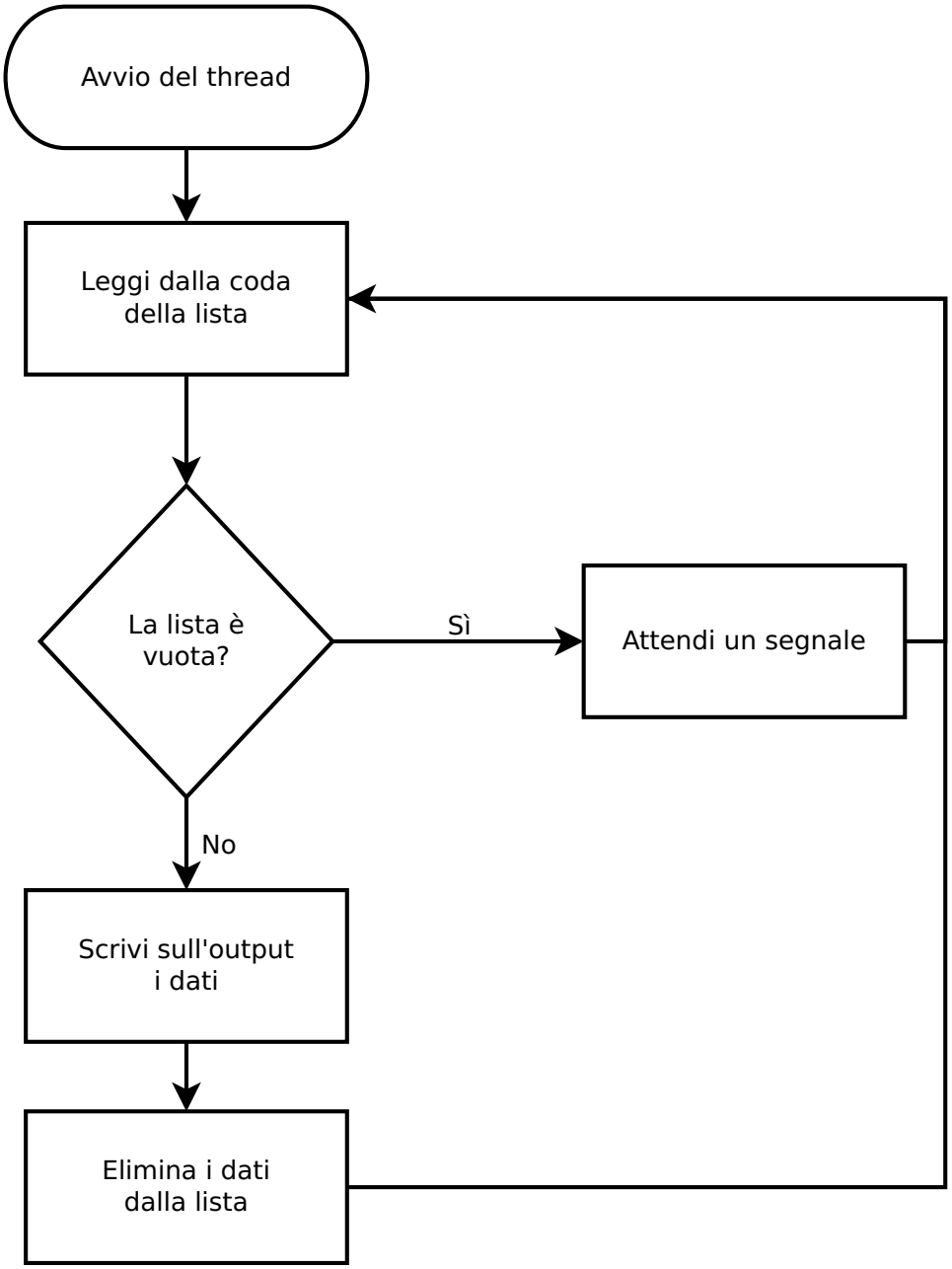

Figura 2.3: Algoritmo del thread di output

Capitolo 3

Sviluppo del progetto

Durante lo sviluppo del progetto si è fatto uso di librerie e strumenti software per semplificare il più possibile il lavoro da svolgere ed ottenere il massimo delle prestazioni possibili.

3.1 Librerie

L'utilizzo delle giuste librerie permette di sviluppare più velocemente i propri progetti, ottenere prestazioni spesso migliori rispetto ad una soluzione sviluppata in proprio e sicuramente avere più certezze riguardo al corretto funzionamento delle parti attenenti alla libreria.

3.1.1 Boost

Le librerie Boost[1] sono molto usate nella programmazione in C++ perché implementano moltissime funzioni di comune utilizzo fornendo una interfaccia molto semplice. L'ottima qualità di queste librerie è confermato dal fatto che spesso alcune sue componenti vengono integrate nello standard C++. Queste librerie sono basate sulla versione ANSI del C++, rendendole compatibili con qualunque compilatore che implementi gli standard ANSI. In questo progetto vengono utilizzate principalmente per semplificare la lettura di argomenti da linea di comando, la lettura/scrittura di files, le comunicazioni di rete e il multi-threading.

3.1.2 IPP

Le Integrated Performance Primitives[2] sono delle librerie che implementano un insieme molto completo di funzioni e algoritmi relativi al processamento di segnali e crittografia. In particolare implementano funzioni utili al processamento di segnali audio (filtri, codifiche audio, compressione dei dati, ecc.),

[1] `http://www.boost.org/`

[2] `http://software.intel.com/en-us/intel-ipp/`

funzioni per il processamento di immagini (trasformazioni, codifica video, ecc.), funzioni per calcoli su matrice e funzioni crittografiche (crittografia simmetrica, asimmetrica, funzioni hash, ecc.). Il vantaggio delle IPP è che offrono un ampio spettro di funzioni che coprono praticamente tutte le necessità nel processamento dei segnali, oltre ad essere ottimizzate per processori Intel. Inoltre sono state scritte in modo da essere naturalmente utilizzabili in ambienti multi-threaded, sfruttando tutte le potenzialità di processori multicore o processori multipli. Tutti questi fattori hanno concorso nell'adozione di questa libreria al posto di altre librerie concorrenti. Questa libreria è l'unico componente non Open Source utilizzato nell'intero progetto, anche se il suo utilizzo è gratuito per usi accademici. Le librerie alternative potrebbero essere comparate per verificare l'effettiva qualità a livello di prestazioni[3].

Multithreading e prestazioni nelle IPP

Le IPP sono state sviluppate con l'intenzione di sfruttare a pieno le istruzioni Single Instruction, Multiple Data (SIMD) e Streaming SIMD Extensions (SSE) presenti nei moderni processori. Queste istruzioni permettono di effettuare operazioni su vettori in parallelo, guadagnando grandi vantaggi prestazionali in alcuni tipi di operazioni, tra cui anche i calcoli necessari per l'elaborazione di segnali. Siccome diversi processori supportano diversi tipi di istruzioni, possono esistere diverse implementazioni di alcune funzioni in base al tipo di istruzioni supportate. Per selezionare automaticamente la funzione corretta, le IPP dispongono di un dispatcher che, durante l'inizializzazione a run-time, individuano le capacità del processore e quindi la categoria di libreria da utilizzare. Questo permette di usare trasparentemente le funzioni di libreria sfruttando sempre l'implementazione più efficiente per il processore utilizzato.

Alcune funzioni di base della libreria sfruttano il threading per migliorare le proprie prestazioni, in particolare sfruttando le librerie OpenMP sviluppate da Intel. Quando si utilizza la versione multi-threaded della libreria, i thread vengono automaticamente inizializzati al numero di processori e core presenti: a questo modo la libreria avvierà al massimo tanti thread quante sono le unità di calcolo presenti nel sistema, evitando di sovracaricare il sistema e ottenendo così la migliore prestazione possibile dal sistema in uso. Sono presenti anche due funzioni, `ippSetNumThreads()` e `ippGetNumThreads()`, che permettono di manipolare il numero di threads utilizzati dalla libreria. Il numero di thread non sarà mai maggiore rispetto al numero di unità di calcolo del sistema, anche se impostati manualmente ad un valore maggiore.

[3] cfr. paragrafo 5.2.2

Tutte le funzioni IPP sono thread-safe, cioè possono essere usate da diversi thread di esecuzione contemporaneamente senza che questo implichi un qualunque malfunzionamento.[4]

3.2 Codice pre-esistente

Parte del codice del progetto è stato prelevato dal software sviluppato in fase di tirocinio, sempre presso l'Istituto di Radioastronomia di Medicina. Questo codice ha permesso di ridurre notevolmente il tempo dedicato allo sviluppo del calcolo della FFT lasciando maggior tempo per una corretta implementazione delle altre parti. Questo codice è stato adattato per funzionare in un contesto multi-threaded e per accettare input da diverse fonti.

3.3 Gestione di problemi comuni

3.3.1 Selezione del tipo di dato

Per rendere il programma il più versatile possibile, non si poteva legare il suo funzionamento ad un unico tipo di dati. Siccome le IPP includono funzioni per diversi tipi di dati, il programma è stato sviluppato in modo da riuscire a selezionare il giusto tipo di funzione a seconda del tipo di dato in input e in output. L'implementazione attuale permette di selezionare il tipo di dato impostando delle costanti e ricompilando il codice, un possibile miglioramento potrebbe essere la selezione dinamica senza bisogno di ricompilazione del tipo di dato, di cui parleremo nel paragrafo 5.2.1.

Dal numero di bit al tipo di variabile

Il primo passo da compiere per poter lavorare sul tipo di dato desiderato è associare al numero di bit selezionati nella costante il corretto tipo di dato. Il listato 3.1 illustra il metodo di selezione dei numeri di bit e la definizione delle costanti contenenti i tipi di dato corretti. Come si può osservare, si usano i template per ottenere il risultato che vogliamo e con delle `typedef` definiamo dei nuovi tipi corrispondenti alle dimensioni richieste. Parte del codice di `typer` è riportato nel listato 3.2. Si può osservare come molto semplicemente al variare dei due parametri del template, si definisce un tipo denominato `type` corrispondente alle caratteristiche richieste.

[4] Per approfondimenti, cfr. `http://software.intel.com/en-us/articles/threading-and-intel-integrated-performance-primitives/`

Listing 3.1: Definizione di costanti per la selezione del tipo di dato

```
/* CONSTANTS to define data type.
 * Change this & recompile to change data types!
 */

const int IPP_DATA_LENGTH = 16;
const int IPP_OUTPUT_DATA_LENGTH = 32;
const bool IS_COMPLEX_TYPE = false;
typedef typer<IPP_DATA_LENGTH, IS_COMPLEX_TYPE>::type IppType;
typedef typer<IPP_OUTPUT_DATA_LENGTH, IS_COMPLEX_TYPE>::type DstIppType;

typedef boost::shared_ptr< FFTBuf<DstIppType> > FFTBufPtr;
```

Listing 3.2: La magia dietro `typer`

```
template<int size, bool is_complex> struct typer;
template<> struct typer<16, false>
{
    typedef Ipp16s type;

};

template<> struct typer<16, true>
{
    typedef Ipp16sc type;

};

template<> struct typer<32, false>
{
    typedef Ipp32f type;

};
```

Listing 3.3: Alcune implementazioni della funzione `IPP::malloc`

```
/* Alloc */

Ipp8u *IPP::alloc(Ipp8u *d, long int length) {
    d = ippsMalloc_8u(length);
    if( d == NULL ) {
        std::stringstream ss;
        ss << "Not enough memory" << std::endl;
        error(ss);
        exit(3);
    }
    return d;
}

Ipp16s *IPP::alloc(Ipp16s *d, long int length) {
    d = ippsMalloc_16s(length);
    if( d == NULL ) {
        std::stringstream ss;
        ss << "Not enough memory" << std::endl;
        error(ss);
        exit(3);
    }
    return d;
}
```

Gestione della memoria

La libreria IPP fornisce alcune funzioni per allocare memoria allineata[5] ed effettuare altre operazioni di basso livello. Siccome la libreria è scritta in C, ogni tipo di dato ha una funzione diversa, quindi per poter scrivere codice generico che funzioni con qualunque tipo di dato è stato necessario creare un'interfaccia C++ che richiami automaticamente le funzioni corrette. Il metodo adottato (listato 3.3) prevede che la funzione generica di interfaccia prenda due argomenti invece di uno solo: il primo argomento è un puntatore al vettore per cui si sta allocando la memoria ed il secondo argomento è la lunghezza di questo vettore. Il primo argomento in realtà non viene utilizzato, ma con la tecnica dell'`overloading` permette di selezionare la funzione corretta.

[5] Gli elaboratori quando accedono ai dati in memoria, lo fanno leggendo un certo numero di bytes alla volta. Per ottenere migliori prestazioni, è ideale che la memoria per le variabili venga allocata allineata al punto in cui il processore inizierà a leggerla.

Listing 3.4: Dichiarazione della struttura `SrcType`

```
template<class T> struct SrcType {
    public:
        T *data;
        mutable bool erasable;
        mutable boost::mutex *mutex;
        SrcType();
        SrcType(const SrcType &other);
        SrcType<T> & operator=(const SrcType<T> &rhs);
        ~SrcType();
};
```

Listing 3.5: Dichiarazione dell'interfaccia `SourceFilter`

```
#include <stddef.h>

class SourceFilter {
    public:
        virtual ~SourceFilter() {};
        virtual void *read(void *destination, size_t size) = 0;
};
```

Buffer di dati

Per le classi e strutture contenente i dati è stata scelta una strategia diversa: siccome non è necessario chiamare funzioni di libreria, ma i metodi sono generici ed indipendenti dal tipo di dato su cui si opera, sono stati adottati i template come soluzione generica. Come si può vedere nella definizione della struttura `SrcType` (listato 3.4), i membri della struttura sono assolutamente neutri rispetto al tipo di dato utilizzato.

3.3.2 Client UDP come SourceFilter

Nel paragrafo 2.2.1 abbiamo spiegato come in fase di progettazione si sia pensato ad una interfaccia generica per leggere i dati da una fonte esterna; il listato 3.5 mostra l'implementazione di questa interfaccia.

Per implementare questa interfaccia è sufficiente ereditare dalla classe virtuale `SourceFilter` ed implementare il metodo `read()`. Vediamo ad esempio la dichiarazione della classe `udp_sock`, presentata nel listato 3.6: si può notare che vengono definite due funzioni read, una delle quali corrisponde a quella dichiarata nella classe virtuale `SourceFilter`. L'implementazione della `read()` virtuale, presentata nel listato 3.7, non fa altro

Listing 3.6: Dichiarazione della classe `udp_sock`

```
template<class T>
class udp_sock : public SourceFilter
{
    public:
        udp_sock(std::string host, unsigned short port);
        size_t read_dgram(T *dgram);
        T *read(T *ret, size_t length);
        virtual void *read(void *ret, size_t length);
    private:
        boost::asio::io_service _io_service;
        boost::asio::ip::udp::socket _sock;
        boost::array<T,UDP_MAX_DGRAM> _buf;
        size_t _remaining;
        uint64_t _counter;

        void _move_to_front(T *arr, int start, int end = UDP_MAX_DGRAM);
};
```

Listing 3.7: Implementazione della `read()` virtuale

```
template<class T> void *udp_sock<T>::read(void *ret, size_t size)
{
    return read(reinterpret_cast<T *>(ret), size);
}
```

che convertire il puntatore a void in un puntatore del tipo specificato dal template. Questo permette di usare la classe `udp_sock` con qualunque tipo di dato.

3.3.3 Gestione del threading

Quando si introduce il threading in una applicazione, ci sono delle situazioni che vanno prese in considerazione: avere diversi thread può portare problemi normalmente assenti e non facilmente identificabili. Inoltre alcuni problemi legati al threading possono essere difficili da riprodurre perché dipendono dalla schedulazione dei thread da parte del sistema operativo, quindi sono al di fuori dal controllo del programmatore. Per questo motivo esistono alcune tecniche di programmazione che permettono di evitare problemi quando thread diversi devono condividere alcuni dati.

Listing 3.8: Variabili della lista

```
#include <list>
#include <boost/thread/mutex.hpp>
#include <boost/thread/condition_variable.hpp>

template<class T> class List {
    private:
        std::list<T> _dst;
        boost::mutex _mut;
        boost::condition_variable _empty;
    public:
```

Listing 3.9: Implementazione di `empty()`

```
template<class T> bool List<T>::empty() {
    boost::mutex::scoped_lock lock(_mut);
    return _dst.empty();
}
```

Mutua esclusione

Con la mutua esclusione si assicura che quando un thread accede ad una parte di dati protetta, nessun altro thread può accedervi contemporaneamente. La mutua esclusione è stata implementata, ad esempio, come wrapper attorno alla struttura standard `std::list` per renderla utilizzabile da più thread contemporaneamente.

Per rendere la lista thread-safe, servono tre variabili: una contiene la lista standard, una per la mutex ed una per la condition variable, come mostrato nel listato 3.8

Una mutex, abbreviazione di **mut**utal **ex**clusion, serve appunto per poter bloccare l'accesso all'oggetto da parte di altri thread e per rilasciare il blocco quando si ha finito. Le funzioni presenti in `std::list` vengono reimplementate sfruttando la mutex per assicurare un comportamento corretto in caso di multi-threading: troviamo un buon esempio nel metodo `empty()` presentato nel listato 3.9.

La creazione dell'oggetto di tipo `boost::mutex::scoped_lock` sospende l'esecuzione del thread se ci sono già altri thread che possiedono l'esclusiva sull'oggetto `_mut`, altrimenti acquisisce il diritto di esclusiva sulla mutex e procede con l'operazione. Nel distruttore dell'oggetto `lock` avviene il rilascio della mutex, quindi non appena l'oggetto finisce fuori contesto — cioè dopo il `return` — il distruttore viene chiamato e la mutex viene liberata, lasciando

Listing 3.10: Uso della `boost::condition_variable`

```
template<class T> void List<T>::wait() {
    boost::mutex::scoped_lock lock(_mut);
    return _empty.wait(lock);
}
```

libero accesso ad altri threads.

Wait/notify

Abbiamo visto nel paragrafo 2.3.2 come il thread di output si metta in attesa sulla lista quando questa è vuota. Il codice presente nel thread di output è simile a questo:

```
while(list.empty()) {
    list.wait();
}
//Do something
```

Questo codice mette in uno stato "dormiente" il thread, lasciando il posto agli altri thread e ottimizzando così il tempo di scheduling assegnato al programma: l'alternativa sarebbe fare un ciclo continuo — detto *busy waiting* — per verificare la stessa condizione ripetutamente, ma in questo modo si sprecherebbero tantissime risorse su un controllo che non cambierà risultato prima che avvengano altre cose. Per monitorare lo stato di questa condizione, si può utilizzare una `boost::condition_variable`, come mostrato nel listato 3.10. Il codice inizialmente acquisisce un lock esclusivo sulla mutex, poi si mette in attesa sulla `condition_variable _empty`. A questo punto il lock viene rilasciato ed il thread si mette in attesa, senza occupare risorse sul sistema. Per risvegliare questo thread bisogna notificarlo del fatto che la condizione su cui stava aspettando 'e stata modificata, quindi quando si inserisce un nuovo elemento nella lista bisogna anche notificare il thread di output che la lista non è più vuota, come illustrato nel listato 3.11. Il metodo `notify_one()` è definito come:

```
void notify_one() { _empty.notify_one(); }
```

Utilizza, quindi, la `condition_variable` per notificare uno dei thread in attesa sulla variabile stessa. A questo punto il thread di output si risveglia, aspetta di poter riacquisire il lock sull'oggetto e continua con l'esecuzione. A questo punto la condizione `list.empty()` dovrebbe essere falsa e il thread può procedere con il suo lavoro.

Listing 3.11: Notifica di inserimento di un nuovo elemento in lista

```
template<class T> void List<T>::push_back(const T &x) {
    boost::mutex::scoped_lock lock(_mut);
    _dst.push_back(x);
    notify_one();
}
```

Listing 3.12: Codice problematico nel caso di più threads attivi

```
1 if( list.empty() || list.back()->is_src_full() ) {
2         FFTBuf<DstIppType> *buffer( new FFTBuf<DstIppType>(siglen , sur
3         *buffer = IPP::alloc(buffer->cdata(), siglen);
4         IPP::zero_mem(buffer->cdata(), siglen);
5         list.push_back(buffer);
6 }
7 FFTBuf<DstIppType> *work_buffer = list.back();
```

Debugging

Per capire meglio i problemi che possono sorgere osserviamo il codice nel listato 3.12: questo codice faceva parte del thread principale ed è il controllo effettuato per verificare se è necessario creare un nuovo buffer di output o se si può assegnare l'ultimo buffer in lista come punto di scrittura della trasformazione da effettuare, meccanismo descritto nel paragrafo 2.3.3.

Analisi del codice Siccome il listato non è banale, procediamo ad analizzare passo per passo le operazioni effettuate:

Nella riga 1, si controlla innanzitutto se la lista di output è vuota o se l'ultimo elemento nella lista è stato assegnato tante volte quanto sono il numero di somme che si vogliono effettuare. Se almeno una delle due condizioni è vera, significa che dobbiamo accodare un nuovo elemento nella lista. La riga 2 crea il puntatore `buffer` ed inizializza l'oggetto di tipo `FFTBuf<DstIppType>` assegnando come parametri la lunghezza del segnale elaborato (`siglen`) ed il numero di somme che si intendono fare su quel buffer (`sums`). `FFTBuf` è un oggetto che contiene il buffer in cui viene salvato il segnale una volta applicate le trasformazioni e alcuni metodi ausiliari. Uno di questi metodi, usato nella riga 1, è `is_src_full()` che restituisce un valore booleano vero se l'istanza dell'oggetto è stata assegnata ad un numero di thread pari al numero di somme volute, falso altrimenti. Con questo metodo, quindi, si può sapere se l'istanza dell'oggetto è riutilizzabile per un altro thread o meno. La costante DstIppType segnala il tipo di dato su cui si opera, in particolare il numero di bit usati per ogni punto

del segnale: avere questa costante permette di adattare il programma molto velocemente per tipi di dati diversi. Alle righe 3–4 si alloca la memoria per il buffer, utilizzando i metodi delle IPP per fare sì che la memoria sia allineata e la si inizializza a zero. Nella riga 5 si accoda il puntatore al buffer in quanto ora l'inizializzazione è completata. La riga 7 mostra come viene selezionato il buffer su cui lavorare: grazie al codice precedente si tratterà sempre dell'ultimo buffer inserito nella lista.

Comportamento in ambiente multi-threaded Il codice appena descritto funziona perfettamente se non si utilizza il threading. Tuttavia, durante la fase di test, il programma funzionava il più delle volte, ma presto o tardi l'esecuzione terminava con un errore di *segmentation fault*[6]. Osservando il problema con l'ausilio di un debugger, il problema non risultava chiaro perché si presentava in punti differenti del codice e sempre in situazioni in cui non ci si aspetta di avere un puntatore nullo, rendendo l'investigazione tramite debugger praticamente inutile. Il fatto che il problema si mostrasse in modo casuale, in momenti diversi e in punti diversi del codice lasciava intuire che si trattasse di un problema di concorrenza, quindi riesaminando l'interazione dei vari thread tra di loro si è riusciti a scovare e risolvere la fonte del problema: come descritto nel paragrafo 2.3.2, il thread di output lavora sulla lista di buffer e quando finisce di scrivere i dati sull'output, elimina il buffer dalla lista. Se osserviamo in particolare la riga 1 del listato 3.12, vediamo che si chiede prima se la lista è vuota e in seguito si accede all'ultimo elemento in lista. Ma cosa succede se subito dopo l'istruzione `list.empty()` avviene un *context switch* e il thread di output riprende l'esecuzione, scrive l'ultimo buffer presente il lista sull'output e lo elimina? In questo scenario la successiva chiamata a `list.back()` ritornerebbe un puntatore nullo e la chiamata a `is_src_full()` provoca la segmentation fault. Per evitare che questo avvenga, bisogna richiedere il lock sulla mutex della lista di buffer di output prima di testare le condizioni, a questo modo mentre si verifica questa condizione lo stato della lista non può essere alterato.

Conclusioni e osservazioni La soluzione al problema, presentata nei listati 3.13 e 3.14 è stata raggiunta dopo molto tempo perso a ricercare la fonte di questo problema causato da una non corretta analisi dei comportamenti in ottica multi-threaded. Individuare questo genere di errori è molto difficile, mentre commettere questo tipo di errori è molto facile: per questo motivo è generalmente consigliabile evitare la concorrenza a meno che non vi siano chiari vantaggi nell'adottarla; e quando si decide di sfruttare la concorrenza

[6]La segmentation fault è un errore tipico dei programmi C/C++. Significa che un programma ha tentato di accedere ad una zona di memoria non di sua competenza e il sistema operativo non concede di farlo per ovvi motivi di sicurezza. Tipicamente questo problema avviene quando si legge oltre la fine di un array o si tenta di accedere ad un puntatore non allocato o impostato a NULL.

Listing 3.13: Codice funzionante anche in ambiente multi-threaded

```
if(list.item_needed()) {
        FFTBuf<DstIppType> *buffer( new FFTBuf<DstIppType>(siglen , sur
        *buffer = IPP::alloc(buffer->cdata(), siglen);
        IPP::zero_mem(buffer->cdata(), siglen);
        list.push_back(buffer);
}
FFTBuf<DstIppType> *work_buffer = list.back();
```

Listing 3.14: Implementazione del metodo `item_needed()`

```
template<class T> bool List<T>::item_needed() {
    boost::mutex::scoped_lock lock(_mut);
    return _dst.empty() || _dst.back()->is_src_full();
}
```

nel proprio programma, è importante considerare tutti gli scenari possibili nelle interazioni tra i thread soprattutto quando si effettuano operazione non atomiche[7].

3.3.4 Memory leak

Un altro problema molto diffuso nella programmazione in C++ sono i *memory leak*. Con questo termine si indicano quelle perdite di memoria che avvengono perché ad allocazione di memoria da una parte, non corrisponde una deallocazione da un'altra parte del codice. Se un problema di questo genere esiste e non viene corretto, un programma che rimane attivo per un certo periodo di tempo incrementa sempre più il suo utilizzo di memoria, fino a che la memoria disponibile nel sistema non viene interamente consumata. A questo punto non solo il programma non può più funzionare, ma anche gli altri programmi presenti sul sistema possono riscontrare problemi per la mancanza di memoria da allocare; in alcuni casi il sistema potrebbe diventare addirittura inutilizzabile e potebbe diventare necessario un riavvio del sistema. Purtroppo, se la perdita di memoria è sufficientemente piccola, è facile che un problema di questo tipo possa sfuggire in fase di test e creare problemi solo dopo un lungo utilizzo in fase di produzione. Per questo mo-

[7] Una operazione, ad esempio **int** i = 5; è atomica se la sua esecuzione non può essere interrotta a metà: o l'operazione è ancora da effettuare, oppure è completata. In C++ l'operazione di cui sopra viene tradotta in diverse istruzioni macchina e quindi può avvenire un context switch a metà dell'operazione. Le istruzioni in C++, quindi, non sono atomiche. Esistono altri linguaggi di programmazione, pensati appositamente per la programmazione multi-threaded, che garantiscono l'atomicità delle loro istruzioni.

tivo esistono dei tool, come ad esempio la suite di Valgrind[8], che aiutano nell'individuazione di questi problemi. Durante lo sviluppo del progetto è stato incontrato un errore di questo tipo nell'implementazione del buffer per i dati di output.

Implementazione originaria

La classe `FFTBuf` contiene un buffer e qualche metodo per gestire l'accesso al buffer stesso. Nella sua prima versione il codice per inizializzarlo era il seguente:

Listing 3.15: Inizializzazione di un FFTBuf (vecchia versione)

```
FFTBuf<DstIppType> *buffer( new FFTBuf<DstIppType>(siglen, sums) );
*buffer = IPP::alloc(buffer->cdata(), siglen);
IPP::zero_mem(buffer->cdata(), siglen);
```

Come si può vedere, l'allocazione della memoria veniva effettuata manualmente dopo l'inizializzazione dell'oggetto, mentre non era presente alcuna deallocazione della memoria. Per risolvere questo problema ci sono due strade possibili: o si aggiunge una deallocazione della memoria dopo che i dati sono stati scritti su output, o si sposta il meccanismo di allocazione/deallocazione all'interno del costruttore/distruttore della classe. Per rendere l'utilizzo di `FFTBuf` più trasparente ed evitare che il problema si ripeta se questa classe dovesse essere riutilizzata da qualche altra parte, la scelta è ricaduta sulla seconda soluzione.

Implementazione corretta

La nuova versione dell'allocazione di memoria per il buffer interno a `FFTBuf` è la seguente:

```
template<class T> FFTBuf<T>::FFTBuf(long int siglen) : _dst(NULL),
    _siglen(siglen), _expected_sums(1), _processed_sums(0),
    _assigned_sources(0), _written(false)
{
    _dst = IPP::alloc(_dst, siglen);
    IPP::zero_mem(_dst, siglen);
}

template<class T> FFTBuf<T>::FFTBuf(long int siglen, int sums) :
    _dst(NULL), _siglen(siglen), _expected_sums(sums),
    _processed_sums(0), _assigned_sources(0), _written(false)
{
    _dst = IPP::alloc(_dst, siglen);
```

[8] `http://www.valgrind.org/`

```
    IPP::zero_mem(_dst, siglen);
}

template<class T> FFTBuf<T>::~FFTBuf()
{
    notify_all();
    IPP::free(_dst);
    _dst = NULL;
}
```

Ora l'allocazione è legata alla creazione di un nuovo oggetto di tipo `FFTBuf` e la deallocazione viene effettuata automaticamente quando l'oggetto viene distrutto: a questo modo si ha la certezza che tutti i buffer creati vengono distrutti correttamente e che non ci sarà perdita di memoria. Dopo questa modifica, il programma è passato da pochi minuti di operatività con consumo di memoria sempre crescente a un consumo di memoria stabile e conseguente stabilità del programma stesso.

3.4 Programmi di supporto

Allo scopo di testare l'applicazione e verificarne il funzionamento, sono stati sviluppati alcuni programmi di supporto. Trattandosi di programmi in cui le prestazioni non sono essenziali, il linguaggio di programmazione utilizzato è Python per favorire uno sviluppo più rapido.

3.4.1 Server UDP

Uno dei programmi sviluppati serve ad inviare dei dati nel formato corretto tramite il protocollo UDP. Siccome l'ordinamento è importante, anche se non è grave l'eventuale perdita di alcuni pacchetti, all'inizio di ogni pacchetto UDP compare un contatore progressivo lungo 4 byte. Il client controlla questo contatore e se il numero non è in progressione con il pacchetto precedente, sa quanti pacchetti ha perso e svuota i buffer eventualmente pieni per ricominciare ad elaborare i dati a partire dal nuovo pacchetto ricevuto. Questa operazione viene svolta per evitare di elaborare dati incompleti che potrebbero compromettere i risultati. Il server in Python permette di leggere i dati da un file ed inviarli tramite rete ad una determinata destinazione.

3.4.2 Visualizzatore di grafici

Un altro programma di supporto utlissimo per verificare il funzionamento del programma principale è il visualizzatore di grafici. Questo programma prende i dati elaborati e visualizza un grafico in scala logaritmica(?). La scala logaritmica viene utilizzata in quanto permette di visualizzare i dati in

maniera più facilmente comprensibile rispetto ad una scala naturale, senza cambiarne il significato. Visualizzare i dati è un modo semplice per verificare la correttezza delle trasformazioni, se il segnale trasformato è noto.

Capitolo 4

Risultati e considerazioni

Una volta terminato lo sviluppo dell'applicazione e verificato che non ci siano problemi durante l'esecuzione, sono stati eseguiti diversi test per verificare correttezza e prestazione dell'applicazione.

4.1 Correttezza dell'applicazione

Per verificare la correttezza dell'applicazione si visualizza un segnale trasformato dal programma; questo segnale è noto o comunque si conosce la forma approssimativa che si dovrebbe ottenere, come accennato nel paragrafo 3.4.2. In figura 4.1 si vede la visualizzazione di un segnale con larghezza di banda di 5 Mhz ed elaborato a 4096 canali di frequenza. Come si vede dalla figura, tra i canali 1000 e 3100 circa c'è una grande curva con piccoli picchi sopra di essa. La curva, o *panettone*, rappresenta una elaborazione del segnale: prima di arrivare allo spettrometro, il segnale è pre-processato con alcuni filtri che gli danno l'aspetto in figura. I dati rilevanti sono quelli presenti sulla zona piatta in cima al panettone, mentre i picchi presenti sopra il panettone rappresentano verosimilmente delle interferenze.

Va notato che uno strumento di misura introduce un qualche tipo di "distorsione" detta *bandshape*, o banda caratteristica del ricevitore. Questo implica che nell'interpretazione di un segnale va considerata questa bandshape per evitare di trarre conclusioni erronee sulla natura del segnale: normalmente è possibile "correggere" il segnale registrato in modo da ottenere la forma reale del segnale stesso.

Da questa immagine si capisce quindi che il programma elabora correttamente i dati e produce una trasformazione corretta dei segnali.

4.1.1 Riduzione del rumore

Verifichiamo anche il corretto funzionamento del metodo delle somme dello stesso segnale per attutire il rumore: nella figura 4.2 si vede un segnale con

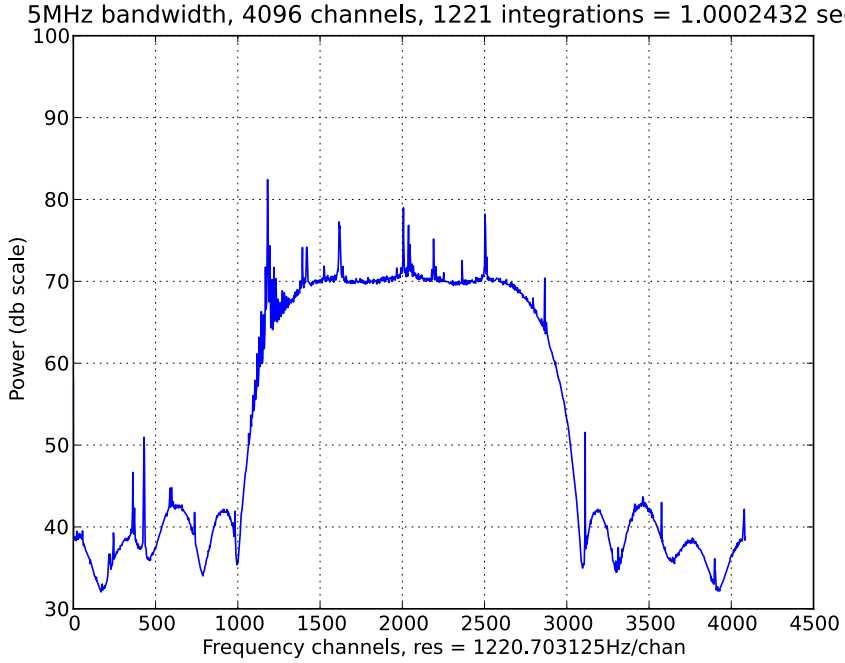

Figura 4.1: Visualizzazione di un segnale elaborato con il programma

5 Mhz di banda ed elaborato a 512 canali, senza effettuare alcuna somma di altre elaborazioni. Si può osservare con il grafico risultante sia estremamente frastagliato e a malapena si riesce ad intuire la forma del segnale, mentre le eventuali interferenze non sono nemmeno distringuibili dal rumore di fondo.

Confrontiamo questa immagine con la figura 4.3 dove vengono effettuate 97656 elaborazioni successive di un segnale nel tempo e poi sommate tra di loro: in questa seconda immagine è perfettamente delineata la forma del segnale così come sono perfettamente visibili le interferenze presenti. Si intuisce anche come le due immagini rappresentino la stessa cosa, anche se la prima è un po' "sfuocata", non sufficientemente dettagliata, mentre la seconda è molto più netta. Questo avviene perché integrando un segnale periodico migliora il rapporto segnale/rumore (SNR).

Tuttavia va considerato anche che aumentando il numero di somme, aumenta anche il tempo di elaborazione necessario: nel primo caso per calcolare i dati rappresentati nell'immagine è stato richiesto un tempo di $0,0001$ secondi, mentre nel secondo caso sono stati impiegati $10,9$ secondi. Chiaramente va ricercato un compromesso tra riduzione del rumore e tempo di calcolo, compromesso che varia a seconda del tipo di analisi che si desidera

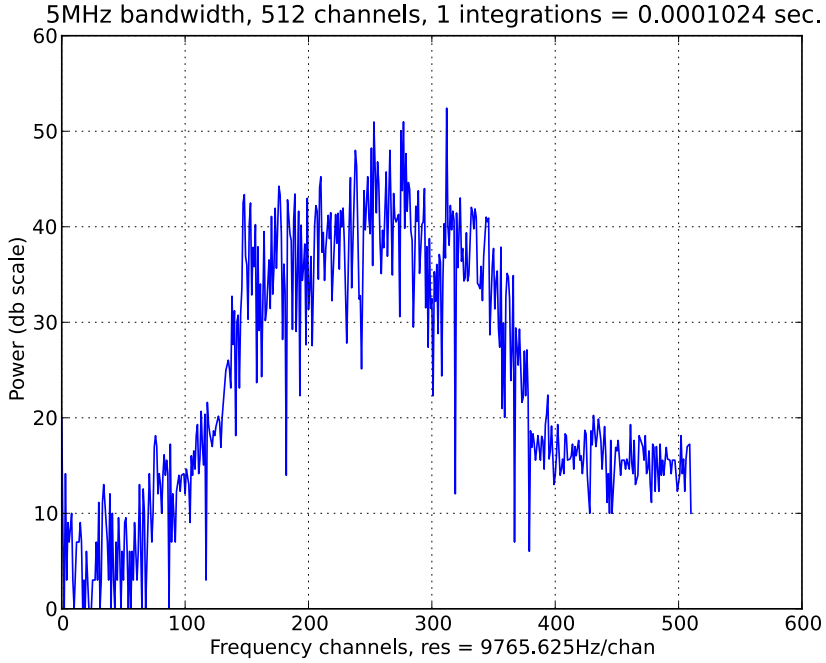

Figura 4.2: Segnale senza nessuna integrazione

fare: per alcuni tipi di osservazioni va bene anche un segnale non molto ben definito, purché sia calcolato molto in fretta; per altri tipi di osservazioni, il segnale deve essere definito molto precisamente, al costo di un maggiore tempo di calcolo.

4.1.2 Variazione del numero di canali

Il numero di canali usati nel calcolo della FFT influiscono sulla risoluzione di ognuno di questi canali: se i canali sono pochi, ogni canale rappresenta una banda di frequenza piuttosto ampia, mentre con molti canali la banda di frequenza si restringe. Aumentare il numero di canali permette di analizzare un segnale più nel dettaglio, osservando anche variazioni tra frequenze molto vicine tra loro. Al contrario, con pochi canali l'energia del segnale viene distribuita in porzioni di banda vicine tra loro. Per verificare questo fenomeno, si confrontino la figura 4.4 con la figura 4.5: nella seconda figura tutta la zona colorata di blu è composta da oscillazioni vicinissime tra loro, presenti anche nella prima figura, ma abbastanza distanziate da essere ben visibili. Nella prima figura, ogni canale ha una risoluzione di $19531,25 Hz/canale$, mentre nella seconda ogni canale ha una risoluzione di $1,192 Hz/canale$.

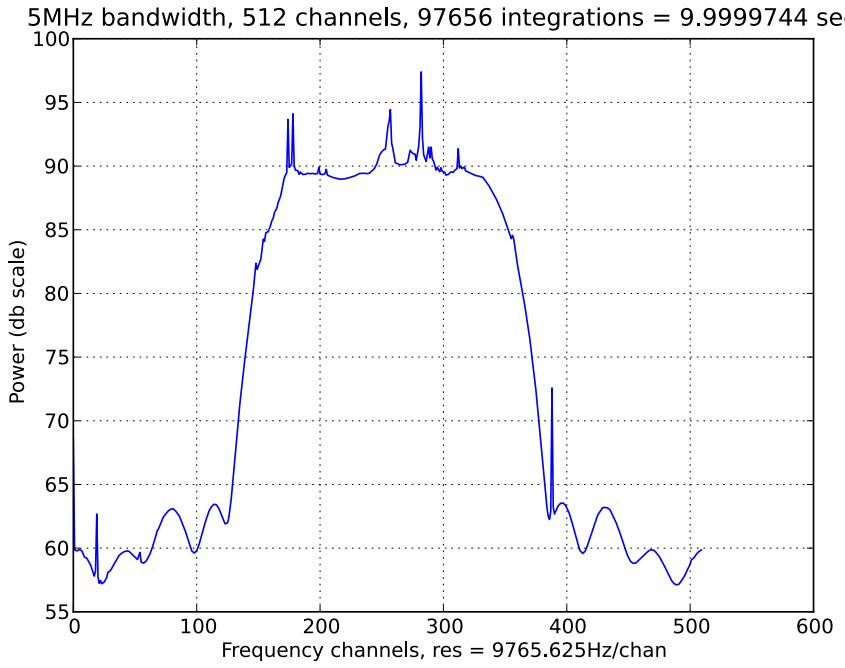

Figura 4.3: Segnale con un elevato numero di integrazioni

Come per la riduzione del rumore di fondo, aumentando il numero di canali aumenta il tempo di calcolo; allo stesso modo il compromesso tra numero di canali e tempo di calcolo dipende dal tipo di analisi che si intende effettuare.

4.2 Verifica delle prestazioni

4.2.1 Caratteristiche dell'elaboratore

Nel verificare le prestazioni del programma, è importante tenere in considerazione il tipo di elaboratore su cui vengono effettuati questi test. Il server collegato ai dati provenienti dall'antenna ha un processore Intel Core 2 Duo a 3 Ghz, 3 Gb di memoria RAM e come sistema operativo Ubuntu Linux. È collegato ad una rete a 10 Gbps con i Jumbo Frames[1] attivati. Tutti i segnali su cui si sono effettuati i test contenevano valori di 16 bit per ogni punto, con banda del segnale di 5 Mhz.

[1] I Jumbo Frames sono dei frame ethernet con più di 1500 bytes di payload. Aumentando il payload, si riduce il carico della CPU e si aumenta il throughput.

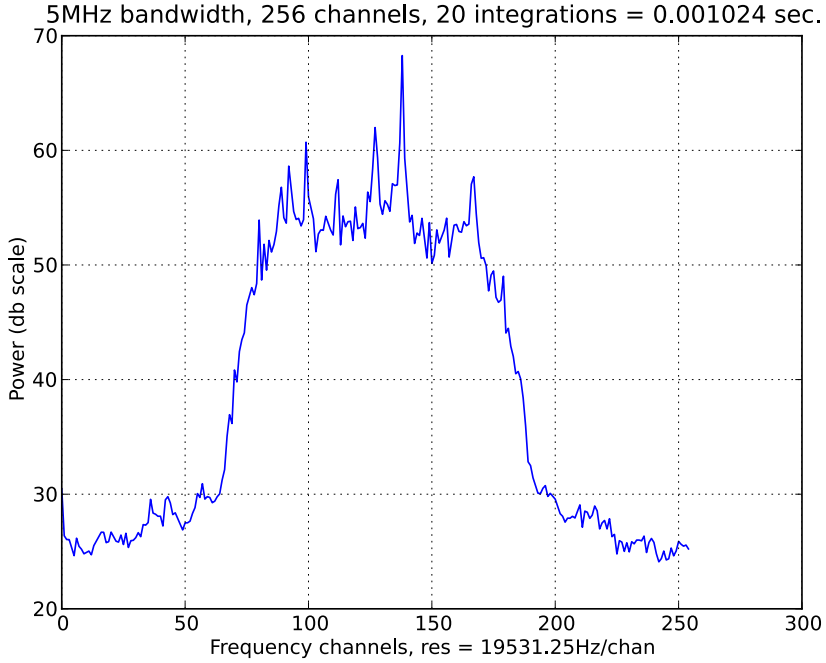

Figura 4.4: Segnale analizzato con pochi canali

4.2.2 Tests eseguiti

Una volta determinato che il programma funziona, è stabile e produce dei risultati corretti, si può verificare il livello di prestazioni. Per questo sono stati fatti dei test dove si aumentava man mano il numero di canali e il numero di integrazioni. L'aumento del numero di integrazioni è stato studiato in modo che l'ordine di grandezza del tempo di calcolo previsto aumentasse ad ogni test: se si osserva le prime righe della tabella A.1 si nota che all'aumentare del numero di integrazioni il tempo si moltiplica all'incirca per 10.[2] Per valutare il numero di integrazioni necessarie, la formula utilizzata è $integrations = \frac{bandwidth}{channels} * time$. Considerando che tutti i segnali utilizzati hanno 5 Mhz di banda, è facile ricavare il numero di integrazioni richieste. I tempi testati oscillano tra 10^{-3} e 10. I canali aumentano raddoppiando, per motivi visti nel paragrafo 1.3.3, e ogni volta che si incrementa il numero di canali, conseguentemente incrementa il tempo di calcolo.

L'obiettivo di questi test è di verificare un eventuale limite superiore del programma nel calcolare le trasformate. Avendo predeterminato i tempi di

[2]Notare che nella prima riga si impiega 20 volte meno tempo rispetto alla seconda. Questo perché le integrazioni avrebbero dovuto essere 2 per ottenere un tempo 10 volte inferiore.

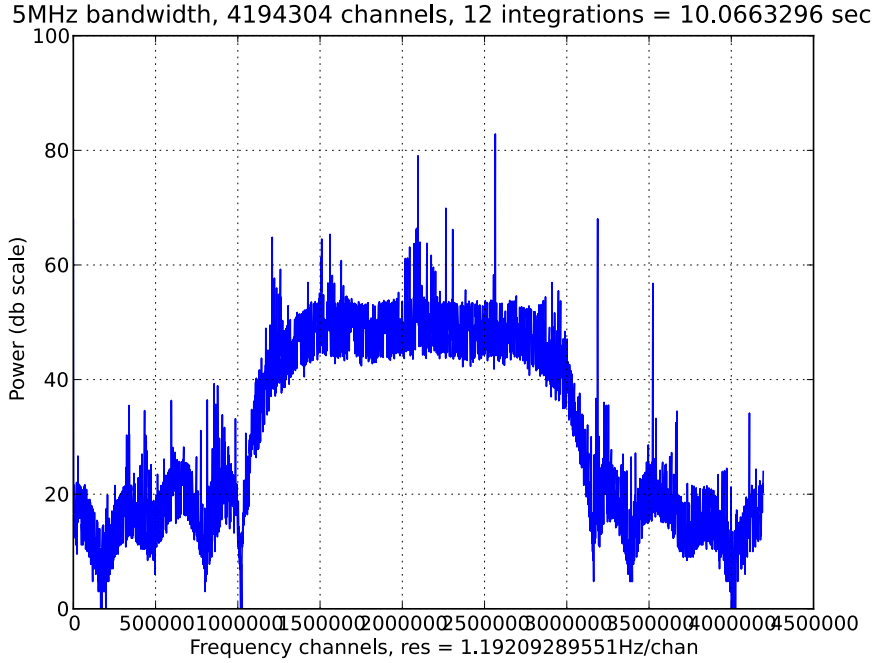

Figura 4.5: Segnale analizzato con molti canali

calcolo desiderati, è facile verificare la tenuta dell'algoritmo: si osservino le tabelle nell'appendice A, si noterà che tutti i tempi rientrano nelle previsioni.

Si è arrivati a testare al massimo $2^{22} = 4194304$ canali perch/'e essendo il segnale a 5 Mhz di banda, non avrebbe senso analizzarlo ad una risoluzione minore di $1\frac{Hz}{canale}$. Tuttavia per verificare in quali condizioni il programma smette di funzionare, si sono effettuati dei test con un numero di canali ancora maggiore, arrivando sino a $2^{27} = 134217728$ canali: attorno ai valori 2^{26}–2^{27} il programma non riesce ad allocare tutta la memoria necessaria all'elaborazione. Inoltre elaborando a 2^{25} canali si inizia a perdere qualche pacchetto UDP dalla rete, segno che il processore non riesce ad elaborare il segnale con sufficiente rapidità. Con questi limiti, il programma è in grado di elaborare segnali fino a 20 Mhz di banda e, con un hardware più potente, potrebbe essere possibile elaborare segnali a banda ancora più larga, ben oltre le reali necessità.

Tabella 4.1: Esempio di alcuni tempi di calcolo

Can.	Integr.	Tempo totale[a]	Dim. output[b]	Segnali output	Tempo trasf.[a c]
256	1	4.831	96423936	94164	5.13041e-05
256	20	5.984	5974016	5834	0.00102571
256	195	3.952	403456	394	0.0100305
256	195313	24.182	2048	2	12.091
512	1	5.087	101545984	49583	0.000102596
512	98	9.040	1841152	899	0.0100556
65536	1	5.115	101974016	389	0.0131491
65536	76	6.267	1572864	6	1.0445
524288	1	3.962	77594624	37	0.107081
524288	95	38.645	6291456	3	12.8817
4194304	1	6.242	100663296	6	1.04033
4194304	12	38.022	50331648	3	12.674

[a] Tempo espresso in secondi

[b] Dimensione dei files in bytes

[c] Si intende il tempo impiegato per effettuare tutte le FFT e le somme dei risultati.

Capitolo 5

Conclusioni e sviluppi futuri

5.1 Conclusioni

Lo scopo principale dello sviluppo di questo programma è capire che tipo di prestazioni ci si può aspettare da una soluzione software sull'elaborazione di segnali radioastronomici. Basandosi sui risultati dei test, si vede che il programma è in grado di elaborare correttamente delle quantità importanti di dati; se confrontate con le prestazioni di una macchina DSP[1] sicuramente saranno inferiori, ma se si aggiunge al confronto anche i fattori del costo dell'attrezzatura e la flessibilità della soluzione alle esigenze si capisce come le prestazioni di questo software siano importanti: il software lavora su un normale elaboratore il cui prezzo, nel caso di macchine comunque high-end e piuttosto potenti, si aggira attorno ai 6000 euro. L'hardware specializzato per il DSP ha costi molto più elevati, nell'ordine dei 10000 euro, anche se spesso i consumi sono minori. Inoltre, modificare un programma scritto in C++ è un'operazione relativamente semplice e che molte persone sono in grado di fare, mentre per modificare l'algoritmo in una macchina DSP bisogna modificare l'hardware, operazione per nulla semplice, né rapida, quindi anche più costosa. Avere una soluzione un po' meno performante, ma comunque di buon livello a costi contenuti è importante sia in quei campi di ricerca dove le prestazioni non sono un aspetto fondamentale del lavoro, sia per progetti a budget molto ristretto, dove avere degli strumenti a basso costo è fondamentale. Un altro aspetto importante è il fatto che un elaboratore generico può essere riutilizzato per altri scopi, ad esempio alla fine del progetto o nei tempi "morti" in cui non serve fare elaborazioni sui segnali, mentre hardware specifico per il DSP è difficile da riutilizzare in altri ambiti.

In generale, dipende molto dalla natura della ricerca e delle analisi da effettuare: in alcuni casi una soluzione software è più conveniente, in altri

[1] DSP significa, in questo caso, Digital Signal Processor. Con questo nome si indicano degli elaboratori specializzati che effettuano solamente le trasformate per cui sono costruiti.

il costo iniziale di un DSP viene ammortizzato nel tempo. Va considerato, anche, che difficilmente una soluzione software può esulare completamente dall'utilizzo di un DSP, ma può ridurne comunque il numero o aumentare il tipo di elaborazioni fattibili sui dati pre-elaborati dal DSP. I risultati raggiunti mostrano come la soluzione software possa essere affiancata, o in alcuni casi addirittura sostituita, ai tradizionali DSP per offire una soluzione più adeguata alle esigenze della ricerca. Come in tanti altri casi, non c'è una soluzione unica per risolvere tutti i problemi, ma questo spettrometro è sicuramente uno strumento addizionale a disposizione del ricercatore.

5.2 Sviluppi futuri

Gli ottimi risultati conseguiti aprono la strada a diversi sviluppi possibili: si pu adattare il programma ad altri scopi, provare implementazioni diverse, cercare di migliorarne ancora le prestazioni e tanto altro ancora.

5.2.1 Estensione del programma

Implementazione di altre trasformazioni

Il modo più semplice per estendere le funzionalità del programma è scrivere qualche trasformazione o filtro aggiuntivo da accodare nella `FilterChain`. Essendo già pronte le interfacce, ed esistendo già una sua implementazione, la parte di lavoro necessaria riguarda solamente la scrittura delle funzioni di trasformazione, senza dover sviluppare un modo per gestire il flusso dei dati.

Selezione dinamica del tipo di dato

Nell'implementazione attuale, la selezione del tipo di dati in ingresso ed in uscira è basato su delle costanti all'interno del codice. Per modificare queste costanti bisogna modificare il programma e ricompilarlo, operazione che, per quanto semplice, potrebbe essere rimossa completamente permettendo di selezionare il tipo di dato ad esempio da linea di comando.

Controllo interattivo dei parametri

Un altro possibile sviluppo del programma è la possibilità di controllare in modo interattivo i parametri del programma ed avviare/fermare il processo tramite una console accessibile tramite rete. Il funzionamento è piuttosto semplice: connettendosi tramite rete alla console, si possono impartire vari comandi per modificare il tipo di trasformazioni effettuate ed i loro parametri, la fonte di acquisizione e la destinazione dei dati. Si pu anche avviare e fermare il ciclo principale di elaborazione e leggere delle statistiche sul suo funzionamento.

Output di rete

Attualmente l'unica implementazione del `SinkFilter` è una classe che scrive l'output su file. Molto utile potrebbe essere la scrittura su una interfaccia di rete, così che i dati possano essere letti da un computer remoto. Questo permetterebbe anche di visualizzare i dati in tempo reale spostando sul client il programma di visualizzazione.

Acquisizione dati SETI

L'acquisizione di dati per il progetto SETI non avviene tramite rete, ma utilizza uno speciale hardware dedicato. Leggere i dati da questo hardware richiede un certo lavoro via software che potrebbe essere integrato all'interno dell'interfaccia `SourceFilter`, così che il programma possa essere utilizzato per elaborare i dati per il progetto SETI.

5.2.2 Librerie alternative

Esistono diverse liberie alternative alle IPP che svolgono le stesse operazioni. Per avere parametri di confronto, sarebbe interessante utilizzarle nello stesso tipo di programma per verificare se effettivamente le ottimizzazioni presenti nelle IPP offrono grandi vantaggi rispetto ad implementazioni alternative.

FFTW: The Fastest Fourier Transform in the West

La libreria FFTW è un progetto Open Source che sostiene di avere la più rapida implementazione della FFT.

Framewave

La libreria Open Source Framewave è stata sviluppata da AMD per fare concorrenza alle IPP. L'interfaccia di Framewave è molto simile a quella delle IPP, rendendo quindi più semplice l'adattamento del programma all'utilizzo di questa libreria.

FFT sulla GPU

Esistono diverse librerie che invece di sfruttare il processore per calcolare la FFT, utilizzano la GPU della scheda grafica. Alcuni benchmark[2] mostrano un miglioramento interessante delle prestazioni rispetto a librerie per CPU, soprattutto con l'aumentare il numero di canali. Esistono diverse librerie che permettono di sfruttare la GPU, tra cui CUDA[3] e GPUFFTW[4]. Se queste librerie si dovessero mostrare realmente efficaci, i requisiti hardware si

[2] `http://www.cv.nrao.edu/~pdemores/gpu/`
[3] `http://www.nvidia.com/object/cuda_home_new.html`
[4] `http://gamma.cs.unc.edu/GPUFFTW/`

limiterebbero fondamentalmente alla scheda grafica, permettendo di ridurre ulteriormente il costo di una soluzione software.

5.2.3 Compilatori

Un altro aspetto da esplorare sono i compilatori: nel progetto è stato utilizzato il compilatore g++, parte della suite di compilatori GNU gcc, con le sue opzioni standard. Siccome dalla qualità del compilatore e dalle opzioni utilizzate dipende anche la qualità del codice macchina generato e quindi la sua rapidità, un miglioramento di prestazioni potrebbe essere possibile esplorando diverse opzioni.

GCC/g++

Il compilatore GNU gcc, nella sua versione per il C++ g++, è il più diffuso in ambiente *NIX. Supporta numerose architetture e per ogni architettura ha delle specifiche opzioni per ottimizzare il codice prodotto. Siccome nel progetto è stato utilizzato senza particolari opzioni, sarebbe interessante verificare se è possibile spremere prestazioni migliori con l'utilizzo delle funzionalità avanzate di g++.

ICC

Il compilatore ICC è stato sviluppato da Intel ed è il compilatore con cui sono state compilate le liberie IPP stesse. Essendo stato sviluppato dall'azienda produttrice di processori, potrebbe essere in grado di sfruttare molto meglio l'architettura dei processori Intel rispetto ad altri compilatori.

Clang e LLVM

Clang e LLVM formano insieme un compilatore e la sua interfaccia; sono un progetto relativamente giovane e sono stati sviluppati da Apple con una licenza di tipo BSD. Questo nuovo compilatore sembra produrre codice molto pi rapido rispetto gli altri compilatori e sta guadagnando un seguito molto importante nella comunità FreeBSD e non solo.

5.2.4 Altri sviluppi

La natura modulare del programma lo prestano ad un'infinità di manipolazioni ed utilizzi nuovi. Con i dovuti arrangiamenti potrebbe addirittura essere utilizzato per scopi completamente differenti, come ad esempio la codifica di file audio o l'elaborazione di immagini. A seconda delle necessità si possono apportare piccoli miglioramenti e l'architettura pensata cerca di essere il più possibile flessibile verso esigenze diverse. Come per ogni progetto

software, le vere potenzialità ed i veri limiti diventeranno evidenti solamente nel tempo e tentando di utilizzare il programma negli ambiti più differenti.

Appendice A

Risultati dei test

Tabella A.1: Tempo di calcolo a 256 canali

Can.	Integr.	Tempo totale[a]	Dim. output[b]	Segnali output	Tempo trasf.[a c]
256	1	4.831	96423936	94164	5.13041e-05
256	20	5.984	5974016	5834	0.00102571
256	195	3.952	403456	394	0.0100305
256	1953	6.926	70656	69	0.100377
256	19531	6.252	6144	6	1.042
256	195313	24.182	2048	2	12.091

[a] Tempo espresso in secondi

[b] Dimensione dei files in bytes

[c] Si intende il tempo impiegato per effettuare tutte le FFT e le somme dei risultati.

Tabella A.2: Tempo di calcolo a 512 canali

Can.	Integr.	Tempo totale[a]	Dim. output[b]	Segnali output	Tempo trasf.[a c]
512	1	5.087	101545984	49583	0.000102596
512	10	4.691	9361408	4571	0.00102625
512	98	9.040	1841152	899	0.0100556
512	977	5.844	118784	58	0.100759
512	9766	6.437	12288	6	1.07283
512	97656	21.845	4096	2	10.9225

[a] Tempo espresso in secondi

[b] Dimensione dei files in bytes

[c] Si intende il tempo impiegato per effettuare tutte le FFT e le somme dei risultati.

Tabella A.3: Tempo di calcolo a 1024 canali

Can.	Integr.	Tempo totale[a]	Dim. output[b]	Segnali output	Tempo trasf.[a c]
1024	1	4.981	98603008	24073	0.000206912
1024	5	5.682	22687744	5539	0.00102582
1024	49	5.420	2207744	539	0.0100557
1024	488	6.635	270336	66	0.10053
1024	4883	6.538	24576	6	1.08967
1024	48828	31.378	12288	3	10.4593

[a] Tempo espresso in secondi

[b] Dimensione dei files in bytes

[c] Si intende il tempo impiegato per effettuare tutte le FFT e le somme dei risultati.

Tabella A.4: Tempo di calcolo a 2048 canali

Can.	Integr.	Tempo totale[a]	Dim. output[b]	Segnali output	Tempo trasf.[a c]
2048	1	3.209	63979520	7810	0.000410883
2048	2	4.828	48185344	5882	0.000820809
2048	24	4.951	4112384	502	0.00986255
2048	244	7.417	606208	74	0.10023
2048	2441	6.900	49152	6	1.15
2048	24414	31.149	24576	3	10.383

[a] Tempo espresso in secondi

[b] Dimensione dei files in bytes

[c] Si intende il tempo impiegato per effettuare tutte le FFT e le somme dei risultati.

Tabella A.5: Tempo di calcolo a 4096 canali

Can.	Integr.	Tempo totale[a]	Dim. output[b]	Segnali output	Tempo trasf.[a c]
4096	1	3.522	70238208	4287	0.000821554
4096	12	3.789	6291456	384	0.00986719
4096	122	4.611	753664	46	0.100239
4096	1221	5.200	81920	5	1.04
4096	12207	31.146	49152	3	10.382

[a] Tempo espresso in secondi

[b] Dimensione dei files in bytes

[c] Si intende il tempo impiegato per effettuare tutte le FFT e le somme dei risultati.

Tabella A.6: Tempo di calcolo a 8192 canali

Can.	Integr.	Tempo totale[a]	Dim. output[b]	Segnali output	Tempo trasf.[a c]
8192	1	3.261	64946176	1982	0.00164531
8192	6	4.353	14450688	441	0.00987075
8192	61	4.639	1376256	42	0.110452
8192	610	5.950	163840	5	1.19
8192	6104	30.472	98304	3	10.1573

[a] Tempo espresso in secondi

[b] Dimensione dei files in bytes

[c] Si intende il tempo impiegato per effettuare tutte le FFT e le somme dei risultati.

Tabella A.7: Tempo di calcolo a 16384 canali

Can.	Integr.	Tempo totale[a]	Dim. output[b]	Segnali output	Tempo trasf.[a c]
16384	1	4.293	85655552	1307	0.00328462
16384	3	4.723	30212096	461	0.0102451
16384	30	5.841	3801088	58	0.100707
16384	305	6.282	393216	6	1.047
16384	3052	29.566	131072	2	14.783

[a] Tempo espresso in secondi

[b] Dimensione dei files in bytes

[c] Si intende il tempo impiegato per effettuare tutte le FFT e le somme dei risultati.

Tabella A.8: Tempo di calcolo a 32768 canali

Can.	Integr.	Tempo totale[a]	Dim. output[b]	Segnali output	Tempo trasf.[a c]
32768	1	4.215	83755008	639	0.00659624
32768	2	4.764	47448064	362	0.0131602
32768	15	4.316	5636096	43	0.100372
32768	153	4.413	524288	4	1.10325
32768	1526	30.914	393216	3	10.3047

[a] Tempo espresso in secondi

[b] Dimensione dei files in bytes

[c] Si intende il tempo impiegato per effettuare tutte le FFT e le somme dei risultati.

Tabella A.9: Tempo di calcolo a 65536 canali

Can.	Integr.	Tempo totale[a]	Dim. output[b]	Segnali output	Tempo trasf.[a c]
65536	1	5.115	101974016	389	0.0131491
65536	8	4.789	11796480	45	0.106422
65536	76	6.267	1572864	6	1.0445
65536	768	36.736	786432	3	12.2453

[a] Tempo espresso in secondi

[b] Dimensione dei files in bytes

[c] Si intende il tempo impiegato per effettuare tutte le FFT e le somme dei risultati.

Tabella A.10: Tempo di calcolo a 131072 canali

Can.	Integr.	Tempo totale[a]	Dim. output[b]	Segnali output	Tempo trasf.[a c]
131072	1	4.305	85458944	163	0.026411
131072	4	4.575	22020096	42	0.108929
131072	38	5.616	2621440	5	1.1232
131072	381	37.403	1572864	3	12.4677

[a] Tempo espresso in secondi

[b] Dimensione dei files in bytes

[c] Si intende il tempo impiegato per effettuare tutte le FFT e le somme dei risultati.

Tabella A.11: Tempo di calcolo a 262144 canali

Can.	Integr.	Tempo totale[a]	Dim. output[b]	Segnali output	Tempo trasf.[a c]
262144	1	5.628	111149056	106	0.0530943
262144	2	7.484	74448896	71	0.105408
262144	19	5.533	5242880	5	1.1066
262144	191	37.702	3145728	3	12.5673

[a] Tempo espresso in secondi

[b] Dimensione dei files in bytes

[c] Si intende il tempo impiegato per effettuare tutte le FFT e le somme dei risultati.

Tabella A.12: Tempo di calcolo a 524288 canali

Can.	Integr.	Tempo totale[a]	Dim. output[b]	Segnali output	Tempo trasf.[a c]
524288	1	3.962	77594624	37	0.107081
524288	10	5.982	10485760	5	1.1964
524288	95	38.645	6291456	3	12.8817

[a] Tempo espresso in secondi

[b] Dimensione dei files in bytes

[c] Si intende il tempo impiegato per effettuare tutte le FFT e le somme dei risultati.

Tabella A.13: Tempo di calcolo a 1048576 canali

Can.	Integr.	Tempo totale[a]	Dim. output[b]	Segnali output	Tempo trasf.[a c]
1048576	1	4.602	88080384	21	0.219143
1048576	5	8.509	29360128	7	1.21557
1048576	48	33.681	12582912	3	11.227

[a] Tempo espresso in secondi

[b] Dimensione dei files in bytes

[c] Si intende il tempo impiegato per effettuare tutte le FFT e le somme dei risultati.

Tabella A.14: Tempo di calcolo a 2097152 canali

Can.	Integr.	Tempo totale[a]	Dim. output[b]	Segnali output	Tempo trasf.[a c]
2097152	1	3.675	67108864	8	0.459375
2097152	2	4.564	41943040	5	0.9128
2097152	24	37.731	25165824	3	12.577

[a] Tempo espresso in secondi

[b] Dimensione dei files in bytes

[c] Si intende il tempo impiegato per effettuare tutte le FFT e le somme dei risultati.

Tabella A.15: Tempo di calcolo a 4194304 canali

Can.	Integr.	Tempo totale[a]	Dim. output[b]	Segnali output	Tempo trasf.[a c]
4194304	1	6.242	100663296	6	1.04033
4194304	12	38.022	50331648	3	12.674

[a] Tempo espresso in secondi

[b] Dimensione dei files in bytes

[c] Si intende il tempo impiegato per effettuare tutte le FFT e le somme dei risultati.

Bibliografia

[AG04] David Abrahams and Aleksey Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series)*. Addison-Wesley Professional, 2004.

[BCR09] Alberto Bertoni, Paola Campadelli, and Giuliano Rossi. Introduzione all'elaborazione dei segnali. `http://homes.dsi.unimi.it/~bertoni/Introduzione%20elaborazione%20segnali.pdf`, 2009. 1.1

[BJER98] Ravi Bhargava, Lizy K. John, Brian L. Evans, and Ramesh Radhakrishnan. Evaluating mmx technology using dsp and multimedia applications. In *MICRO 31: Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, pages 37–46, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[Bur10] C. Sidney Burrus. *Fast Fourier Transforms*. Connexions, 2010. `http://cnx.org/content/col10550/1.21/`.

[Chu07] Wesley J. Chun. *Core Python Programming*. Prentice Hall, second edition, 2007.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2nd revised edition, 2001.

[CM59] Giuseppe Cocconi and Philip Morrison. Searching for interstellar communications. *Nature*, 184:844 – 846, 1959. `http://www.coseti.org/morris_0.htm`. (document)

[CT65] James Cooley and John Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19(90):297–301, 1965.

[Emb95] Paul M. Embree. *C algorithms for real-time DSP*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1995.

[Gib89] Jerry D. Gibson. *Principles of digital and analog communications*. Macmillan Publishing Co., Inc., Indianapolis, IN, USA, 1989.

[HJB84] M. Heideman, D. Johnson, and C. Burrus. Gauss and the history of the fast fourier transform. *ASSP Magazine, IEEE [see also IEEE Signal Processing Magazine]*, 1(4):14–21, 1984.

[Knu97] Donald E. Knuth. *Fundamental algorithms*, volume 1 of *The Art of Computer Programming*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, third edition, 1997.

[KP99] Brian W. Kernighan and Rob Pike. *The practice of programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[LF98] Paul A. Lynn and Wolfgang Fuerst. *Introductory Digital Signal Processing with Computer Applications*. John Wiley & Sons, Inc., New York, NY, USA, 1998.

[Lyo04] Richard G. Lyons. *Understanding Digital Signal Processing (2nd Edition)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.

[MY97] James H. McClellen and Mark A. Yoder. *DSP First: A Multimedia Approach*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.

[NJ99] Huy Nguyen and Lizy Kurian John. Exploiting simd parallelism in dsp and multimedia algorithms using the altivec technology. In *ICS '99: Proceedings of the 13th international conference on Supercomputing*, pages 11–20, New York, NY, USA, 1999. ACM.

[OSB99] Alan V. Oppenheim, Ronald W. Schafer, and John R. Buck. *Discrete-time signal processing (2nd ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1999.

[OWN96] Alan V. Oppenheim, Alan S. Willsky, and S. Hamid Nawab. *Signals & systems (2nd ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[PPR07] Charles L Phillips, John Parr, and Eve Riskin. *Signals, Systems, and Transforms*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2007.

[PTVF92] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical recipes in C (2nd ed.): the art of scientific computing*. Cambridge University Press, New York, NY, USA, 1992.

[PTVF07] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. Cambridge University Press, New York, NY, USA, 2007.

[Ray04] Eric Steven Raymond. *The Art of UNIX Programming*. Pearson Education, 2004.

[Smi97] Steven W. Smith. *The Scientist and Engineer's Guide to Digital Signal Processing*. California Technical Publications, `http://www.dspguide.com/`, 1997. 3, 4, 7, 8

[Smi07] Julius O. Smith. *Mathematics of the Discrete Fourier Transform (DFT)*. W3K Publishing, `http://www.w3k.org/books/`, 2007. 2

[Ste04] E. Stewart. *Intel Integrated Performance Primitives: How to Optimize Software Applications Using Intel IPP*. Intel Press, 2004.

[Str00] Bjarne Stroustrup. *The C++ Programming Language: Special Edition*. Addison-Wesley Professional, third edition, 2000.

[Tan02] Andrew Tanenbaum. *Computer Networks*. Prentice Hall Professional Technical Reference, 2002.

[VL92] Charles Van Loan. *Computational frameworks for the fast Fourier transform*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1992.