Relazione di Tirocinio

FFT per grandi moli di dati

Studente: Stéphane Bisinger

Matricola 0900035365

Tirocinio svolto presso l'Istituto di Radioastronomia

1. Obiettivo del tirocinio

Il tirocinio aveva come obiettivo l'utilizzo delle librerie Integrated Performance Primitives (IPP) della Intel, disponibili presso l'indirizzo http://software.intel.com/en-us/intel-ipp/, per la creazione di un programma in grado di effettuare trasformate di fourier ed eventualmente altre trasformazioni su un segnale in entrata. L'obiettivo del programma è di riuscire a processare segnali di grandi dimensioni in tempo reale, così che si possano analizzare direttamente i segnali che provengono dalle rilevazioni dei radioscopi dell'istituto.

2. Sviluppo dell'applicazione

Lo sviluppo dell'applicazione ha previsto diverse fasi:

1. Progettazione
2. Impostazione dell'ambiente per lo sviluppo
3. Librerie e versioni
4. Tipi di dato e funzioni di inizializzazione
5. Sviluppo di un generatore di segnali
6. Sviluppo di una prima versione della FFT
7. Aggiunta del Power Spectrum
8. Test della FFT con diversi parametri
9. Sviluppo di una versione che sfrutti il threading

Segue la descrizione delle diverse fasi.

2.1 Progettazione

Durante la progettazione si è scelto di dividere il programma in piccole unità che possano poi venire composte successivamente a seconda delle necessità: l'idea era di sviluppare, in un secondo momento, un programma che permettesse di scegliere da dove acquisire il segnale (file, ftp, ecc.) e quali trasformazioni applicargli (fft, filtro passa alto/basso, ecc.) specificando anche l'ordine delle trasformazioni ed eventuali parametri. Alla fine della catena di trasformazioni si sceglie dove salvare (file, ftp, ecc.) il nuovo segnale generato. Si è scelto di lavorare su interi a 16 bit , anche se tutto rimane facilmente adattabile ad una diversa precisione.

Si è deciso inoltre di utilizzare gli autotools per facilitare la compilazione del progetto.

2.2 Impostazione dell'ambiente di sviluppo

Il primo passo nell'impostare l'ambiente di sviluppo è stato l'installazione delle librerie IPP, le istruzioni incluse sono esaustive e facili da seguire. In seguito si è passati alla configurazione di autoconf e automake: i file di automake sono di semplice scrittura, mentre per autoconf è stato necessario ricercare gli script adatti a verificare la presenza delle librerie IPP e di boost. Trovati questi, l'impostazione del progetto è grosso modo completa e permette già di compilare i sorgenti con il semplice comando ./configure && make.

2.3 Librerie e versioni

2.3.1 IPP

Versione: 6.0.2.076

Le Integrated Performance Primitives 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.), 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 multithreaded, 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.

2.3.2 Boost

Versione: 1.36.0

Le librerie boost sono molto usate nella programmazione in C++ perché implementano moltissime funzioni spesso usate 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++. In questo progetto vengono utilizzate principalmente per semplificare la lettura di argomenti da linea di comando, la lettura/scrittura di files e il multithreading. A causa di modifiche sostanziali nel modo di sfruttare il multithreading implementato da boost, è necessario usare una versione maggiore o uguale alla 1.36.0

2.3.3 Autotools: Autoconf, Automake

Versione Autoconf: 2.61

Versione Automake: 1.10.2

Gli autotools sono lo strumento di gestione progetti più diffuso nel software open source. Anche se a volte un po' complicati da usare, la semplificazione della gestione del progetto rendono il loro utilizzo indispensabile: con un solo comando, infatti, è possibile compilare, testare e creare un archivio del progetto pronto per essere redistribuito. È importante ricordarsi che questi tool tendono a cambiare comportamento tra le singole versioni, quindi va sempre controllato che non vadano fatti aggiornamenti ai files di configurazione.

2.3.4 Git

Versione: 1.6.x.x

Git è un sistema di versioning distribuito ed è usato per gestire la storia del progetto. Benché il suo apprendimento sia piuttosto difficile inizialmente, le potenzialità dello strumento e le sue ottime prestazioni diventano presto irrinunciabili. Inoltre la sua natura distribuita ha permesso di creare e gestire un repository senza bisogno di un server centrale.

Il progetto è comunque presente su github: http://github.com/Kjir/ira/

2.4 Tipi di dato e funzioni di inizializzazione

La prima funzione di libreria che deve essere richiamata è la funzione ippStaticInit(). Questa funzione inizializza internamente la libreria e va chiamata prima di qualunque altra funzione.

```
int main(int argc, char **argv) {
    IppStaticInit();
    //Usa la libreria da qui in poi
}
```

Le librerie IPP lavorano sui propri tipi di dato che vengono qui descritti.

Si è scelto di lavorare su interi a 16 bit, i quali vengono rappresentati dal tipo `Ipp16s` e la cui funzione di allocazione della memoria è `ippsMalloc_16s()` e la corrispondente funzione per liberare la memoria è `ippsFree()`

```
Ipp16s *myVal;
int length = 3;
/* Alloca lo spazio per 3 interi da 16 bit */
myVal = ippsMalloc_16s(length);
 ...
ippsFree(myVal);
```

Per calcolare una FFT servono alcuni parametri che specifichino alcune proprietà. A questo scopo viene inizializzata una variabile di tipo `IppsFFTSpec_R_16s` con i parametri della trasformata

```
IppsFFTSpec_R_16s *spec;
/*
 * La variabile hint serve ad indicare se si desidera un algoritmo
 * veloce oppure un algoritmo accurato. I valori possibili sono tre:
 *  - ippHintAccurate: algoritmo accurato
 *  - ippHintFast: algoritmo veloce
 *  - ippHintNone: algoritmo scelto automaticamente dalla libreria
 */
IppHintAlgorithm hint = ippHintAccurate;
/*
 * L'ordine della FFT indica la lunghezza del segnale con la formula
 * N = 2^order
 */
int order = 8;
/* IppStatus è il tipo di dato previsto per lo stato ritornato dalle
funzioni di IPP */
IppStatus st;
/* Praticamente tutte le funzioni delle IPP ritornano uno stato */
/*
 * Il terzo parametro indica che tipo di normalizzazione viene applicata.
 * In questo caso non viene applicata normalizzazione.
 */
st = ippsFFTInitAlloc_R_16s(&spec, order, IPP_FFT_NODIV_BY_ANY, hint);
/* Qui controlliamo che lo status sia diverso da un errore */
if( st != ippStsNoErr ) {
    /*
     * Se si entra qui, c'è stato un errore.
     * La funzione ippGetStatusString() ritorna una descrizione
     * dell'errore contenuto in st, che stampiamo a video.
     */
    std::cerr << "IPP Error in InitAlloc: " << ippGetStatusString(st);
    std::cerr << "\n";
}
```

L'algoritmo della FFT necessita anche di un buffer, il quale può essere omesso (e quindi allocato direttamente dalla libreria), oppure si può decidere di allocarlo una sola volta per diverse FFT, permettendo così un ulteriore guadagno di prestazioni. Per allocare il buffer bisogna prima scoprire quanto deve essere grande ed allocare poi lo spazio:

```
int bufsize;
Ipp8u *buffer; //Il nostro buffer da allocare
/*
 * Questa funzione calcola la dimensione (bufsize) dalla struttura
 * precedentemente allocata (cfr. listato precedente)
 */
st = ippsFFTGetBufSize_R_16s( spec, &bufsize );
if( st != ippStsNoErr ) {
    std::cerr << "IPP Error in FFTGetBufSize: ";
    std::cerr << ippGetStatusString(st) << "\n";
    exit(2);
}
/* Ora che abbiamo la dimensione del buffer, allochiamo la memoria */
buffer = ippsMalloc_8u(bufsize);
if( buffer == NULL ) {
    std::cerr << "Not enough memory\n";
    exit(3);
}
```

2.5 Sviluppo di un generatore di segnali

Il primo programma sviluppato è stato un generatore di segnali che fornisca dei risultati noti una volta effettuata la trasformata di Fourier. A questo scopo nelle librerie IPP esiste una funzione che genera una sinusoide, riducendo quindi la difficoltà della creazione del programma solamente a lettura delle opzioni e scrittura dell'output. Il programma serve anche da base per lo sviluppo degli altri programmi.

```
IppStatus st;
float phase = 0, rfreq = 0.2;
int siglen = 256;
Ipp16s *signal = ippsMalloc_16s(siglen);
Ipp16s magn;
IppHintAlgorithm hint = ippAlgHintAccurate;
if( signal == NULL ) {
    std::cerr << "Not enough memory" << std::endl;
    return -1;
}
/*
 * Genera un tono (sinusoide)
 * @param signal Il puntatore dove scrivere il segnale
 * @param siglen La lunghezza del segnale da generare
 * @param magn  L'ampiezza massima del segnale
 * @param rfreq Frequenza del segnale relativa alla frequenza di sampling
 *              Prende valori compresi tra [0.0, 0.5) per un segnale
 *              reale, tra [0.0, 1.0) per un segnale complesso.
 * @param phase La fase iniziale del segnale.
 * @param hint  Il tipo di algoritmo da utilizzare
 * */
st = ippsTone_Direct_16s(signal, siglen, magn, rfreq, &phase, hint);
if(st != ippStsNoErr) {
    std::cerr << "IPP Error in Tone Direct: " << ippGetStatusString(st);
    std::cerr << '\n';
    return -2;
}
```

2.6 Sviluppo di una prima versione della FFT

La prima versione sviluppata della FFT serve principalmente a testare la funzione delle IPP, valutare la sua correttezza e capire il suo comportamento, oltre a valutare la velocità di calcolo. Oltre ad utilizzare la funzione delle IPP, va letto l'input, vanno interpretati gli argomenti e va scritto il risultato in output. Una volta scritta la funzione, si può testarne il funzionamento dall'output del generatore. È a questo punto che si è presentata la necessità di scrivere un piccolo programma in Python che disegni un grafico dai dati di output dei precedenti programmi, sfruttando le possibilità offerte dalle librerie numpy e matplotlib.

Vengono omessi i controlli sulle funzioni di inizializzazione, fare riferimento al paragrafo 2.4

```
int siglen = 256, order = 8; // siglen == 2^order
int bufsize, scaling = 0;
IppsFFTSpec_R_16s *spec;
IppStatus st;
Ipp8u *buffer;
IppHintAlgorithm hint;

Ipp16s *signal = ippsMalloc_16s(siglen);
Ipp16s *dst = ippsMalloc_16s(siglen);

IppHintAlgorithm hint = ippAlgHintAccurate;
st = ippsFFTInitAlloc_R_16s(&spec, order, IPP_FFT_NODIV_BY_ANY, hint);
st = ippsFFTGetBufSize_R_16s( spec, &bufsize );
buffer = ippsMalloc_8u(bufsize);

/* A questo modo viene letto un segnale di lunghezza siglen da stdin */
std::istream *in = &std::cin;
(*in).read( (char *)signal, sizeof(*signal) * siglen );

/*
 * Funzione per effettuare la trasformata reale a 16 bit interi.
 * @param signal  Il vettore da cui leggere il segnale originario
 * @param dst     Il vettore in cui scrivere il segnale risultante
 * @param spec    La struttura che descrive i parametri della FFT
 * @param scaling Lo scaling da applicare al risultato. L'output viene
 *                moltiplicato per 2^-scaling
 * @param buffer  Il buffer da usare durante i calcoli (cfr. par. 2.4)
 */
st = ippsFFTFwd_RToPack_16s_Sfs(signal, dst, spec, scaling, buffer);
if( st != ippStsNoErr ) {
    std::cerr << "IPP Error in FFTFwd: " << ippGetStatusString(st);
    std::cerr << "\n";
    return -4;
}
```

2.7 Aggiunta del Power Spectrum

Dopo aver verificato il corretto comportamento della trasformata, si è scelto di aggiungere il calcolo dello spettro di potenza per rappresentare il segnale in frequenza, operazione gestita da una funzione IPP e controllata da un parametro da linea di comando. Contestualmente è stata aggiunta la possibilità di sommare tra loro i risultati di due o più trasformate consecutive sullo stream di dati in modo da far emergere chiaramente i segnali dal rumore di fondo.

```
int siglen = 256, order = 8; // siglen == 2^order
int bufsize, scaling = 0, nin = 10, pscaling = 13, iscaling = 0;
IppsFFTSpec_R_16s *spec;
IppStatus st;
Ipp8u *buffer;
IppHintAlgorithm hint;

Ipp16s *signal = ippsMalloc_16s(siglen);
Ipp16s *dst = ippsMalloc_16s(siglen);
Ipp16s *result = ippsMalloc_16s(siglen);
/* Imposta il vettore result a 0 */
ippsSet_16s(0, result, siglen);

IppHintAlgorithm hint = ippAlgHintAccurate;
st = ippsFFTInitAlloc_R_16s(&spec, order, IPP_FFT_NODIV_BY_ANY, hint);
st = ippsFFTGetBufSize_R_16s( spec, &bufsize );
buffer = ippsMalloc_8u(bufsize);

std::istream *in = &std::cin;
for( int i = 0; i < nin; i++ ) {
    (*in).read( (char *)signal, sizeof(*signal) * siglen );

    st = ippsFFTFwd_RToPack_16s_Sfs(signal, dst, spec, scaling, buffer);
    /* Ipp16sc è un numero complesso intero da 16 bit */
    Ipp16sc *vc, zero = {0, 0};
    vc = ippsMalloc_16sc(siglen);
    if( vc == NULL ) {
        return -3;
    }
    //Set the vector to zero
    ippsSet_16sc(zero, vc, siglen);
    /*
     * Questa funzione converte il risultato della FFT in un vettore di
     * numeri complessi.
     * @param dst    Il risultato della FFT
     * @param vc     Il vettore di complessi in cui inserire il risultato
     * @param siglen La lunghezza del vettore (segnale)
     */
    st = ippsConjPack_16sc(dst, vc, siglen);
    if( st != ippStsNoErr ) {
        return -5;
    }
```

```
    /*
     * Calcola il power spectrum, moltiplicandolo per 2^-pscaling
     * @param vc       Il vettore complesso contenente il segnale
     * @param dst      Il vettore di destinazione
     * @param siglen   La lunghezza del segnale
     * @param pscaling Lo scaling da applicare
     */
    st = ippsPowerSpectr_16sc_Sfs(vc, dst, siglen, pscaling);
    if( st != ippStsNoErr ) {
        return -6;
    }
    ippsFree(vc);
    vc = NULL;
    /*
     * Aggiungi il risultato della FFT in forma di power spectrum al
     * vettore del risultato. Questo viene fatto per nin volte, facendo
     * risaltare eventuali segnali sopra il rumore di fondo.
     * @param dst    Il segnale in forma di power spectrum
     * @param result Il vettore dei risultati a cui aggiungere il segnale
     * @param siglen La lunghezza del segnale
     * @param iscaling Lo scaling da adottare per questa somma. In questo
     *                 caso deve essere uguale a 0
     */
    st = ippsAdd_16s_ISfs(dst, result, siglen, iscaling);
    if( st != ippStsNoErr ) {
        return -7;
    }
}
```

2.8 Test della FFT con diversi parametri

A questo punto si è reso necessario testare l'andamento della trasformata al variare di parametri. A questo scopo è stato sviluppato un programma che misura il tempo impiegato a fare le trasformazioni e che provi diverse combinazioni di opzioni. Il primo test osserva l'andamento della trasformata al variare dell'ordine del segnale (la sua lunghezza, quindi) con e senza integrazioni. Infine si procede ad osservare il comportamento della funzione quando si aumentano il numero di somme consecutive da eseguire.

2.9 Sviluppo di una versione che sfrutti il threading

I test eseguiti al punto precedente hanno mostrato che il threading interno alla libreria non basta a sfruttare ottimamente le risorse disponibili e quindi si è deciso di creare una versione della trasformata che non lavori sequenzialmente quando deve eseguire le somme su tratti consecutivi del segnale, ma che lavori in concorrenza. La struttura del programma viene quindi leggermente modificata per fare in modo che possano essere create due versioni della stessa funzione che possano essere sostituiti per verificare il comportamento in entrambi i casi. La nuova versione ha un thread che legge l'input dal file, mentre altri thread si occupano

di calcolare la trasformata. A questo modo si sfrutta meglio la CPU durante la lettura da file.

3. Risultati dei test e conclusioni

I test mostrano che le librerie IPP sono ben adatte al tipo di performance che si ricerca per il tipo di lavoro che si vuole svolgere. In particolare si riescono ad effettuare FFT di lunghezza 2^{23} in meno di un secondo, permettendo quindi di effettuare diverse elaborazioni in tempo reale oltre alla FFT. I grafici sui tempi, inoltre mostrano che il tempo di calcolo cresce esponenzialmente all'aumentare dell'ordine di grandezza del segnale [grafico1] e che si comporta allo stesso modo sia con integrazioni che senza [grafico2], mentre cresce linearmente con l'aumentare del numero di integrazioni [grafico3]. L'attivazione o meno del threading della libreria non varia il tempo impiegato a calcolare una FFT [grafico4 e grafico5], mentre l'aver sviluppato una versione che effettui in parallelo le FFT parte di una integrazione hanno migliorato le prestazioni, anche se non di grandi valori come ci si poteva aspettare [grafico6 e grafico7]. Tuttavia l'utilizzo dei diversi core della CPU mostra come la seconda versione sfrutti meglio il multiprocessing, facendo sospettare quindi che la lettura dell'input influisca forse anche più del calcolo della trasformata.

Tutti i test sono stati eseguiti su una CPU AMD Athlon 64 X2 Dual Core 5600+ su sistema operativo Linux e kernel 2.6.25 per processori a 64 bit.

Ordine del segnale	FFT semplice	FFT con integrazione	FFT senza threading	FFT con threading
10	0 ms	1 ms	0 ms	0 ms
11	1 ms	1 ms	1 ms	0 ms
12	0 ms	1 ms	0 ms	0 ms
13	0 ms	3 ms	0 ms	1 ms
14	1 ms	8 ms	1 ms	0 ms
15	2 ms	19 ms	1 ms	1 ms
16	3 ms	43 ms	2 ms	3 ms
17	7 ms	87 ms	5 ms	5 ms
18	15 ms	219 ms	12 ms	12 ms
19	32 ms	505 ms	31 ms	25 ms
20	67 ms	1102 ms	62 ms	57 ms
21	130 ms	2039 ms	124 ms	116 ms
22	262 ms	4223 ms	263 ms	254 ms
23	556 ms	9134 ms	553 ms	548 ms
24	1191 ms	18684 ms	1206 ms	1239 ms

Numero integrazioni	FFT senza threading	FFT con threading
20	11 ms	12 ms
1020	543 ms	511 ms
2020	1118 ms	993 ms
3020	1568 ms	1566 ms
4020	2084 ms	1981 ms
5020	2603 ms	2511 ms
6020	3121 ms	3039 ms
7020	3693 ms	3527 ms
8020	4210 ms	4013 ms
9020	4713 ms	5147 ms
10020	5220 ms	5212 ms
11020	5847 ms	5554 ms
12020	6374 ms	6163 ms

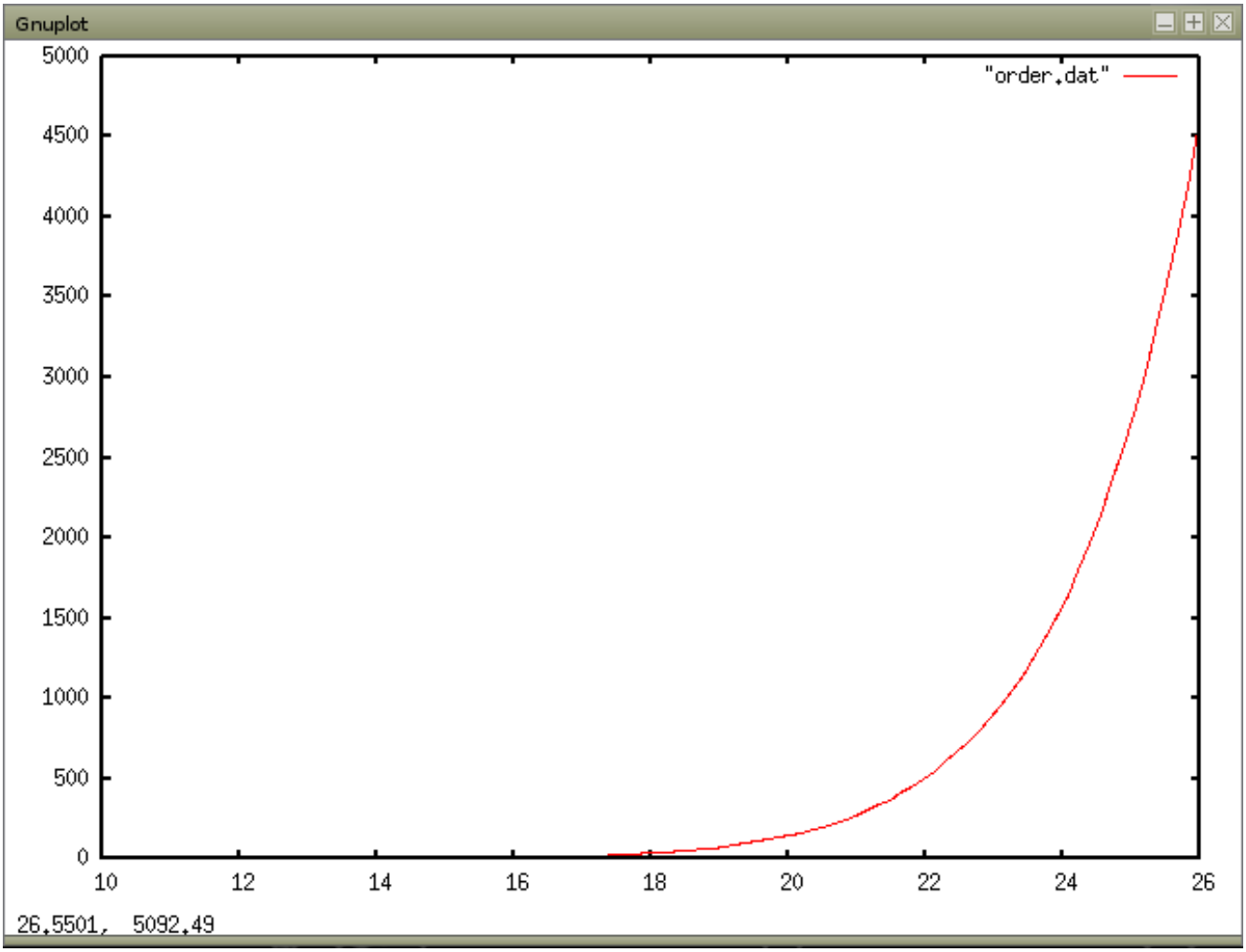

Grafico 1: Tempo impiegato all'aumentare dell'ordine della FFT

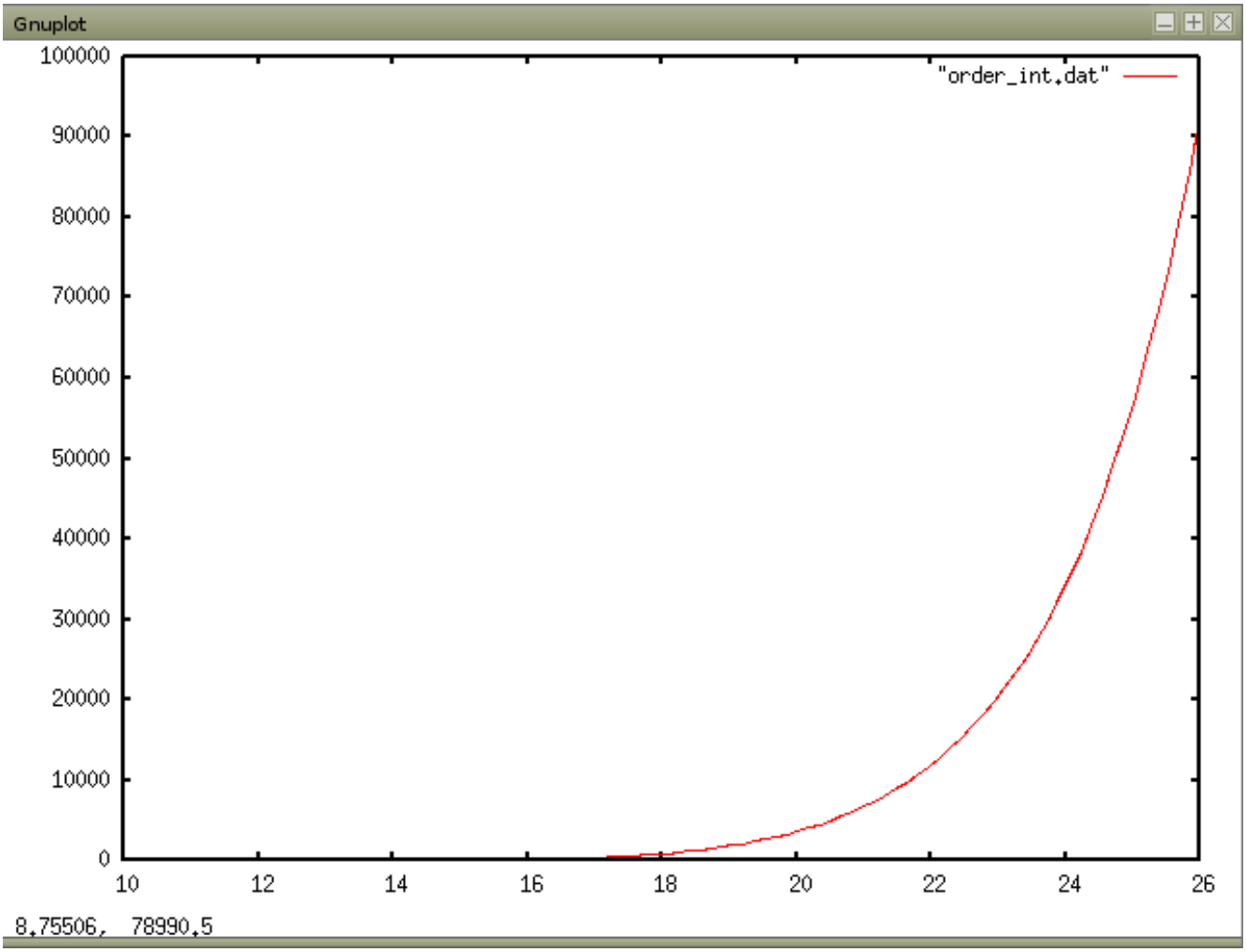

Grafico 2: Tempo impiegato all'aumentare dell'ordine della FFT, con 30 integrazioni

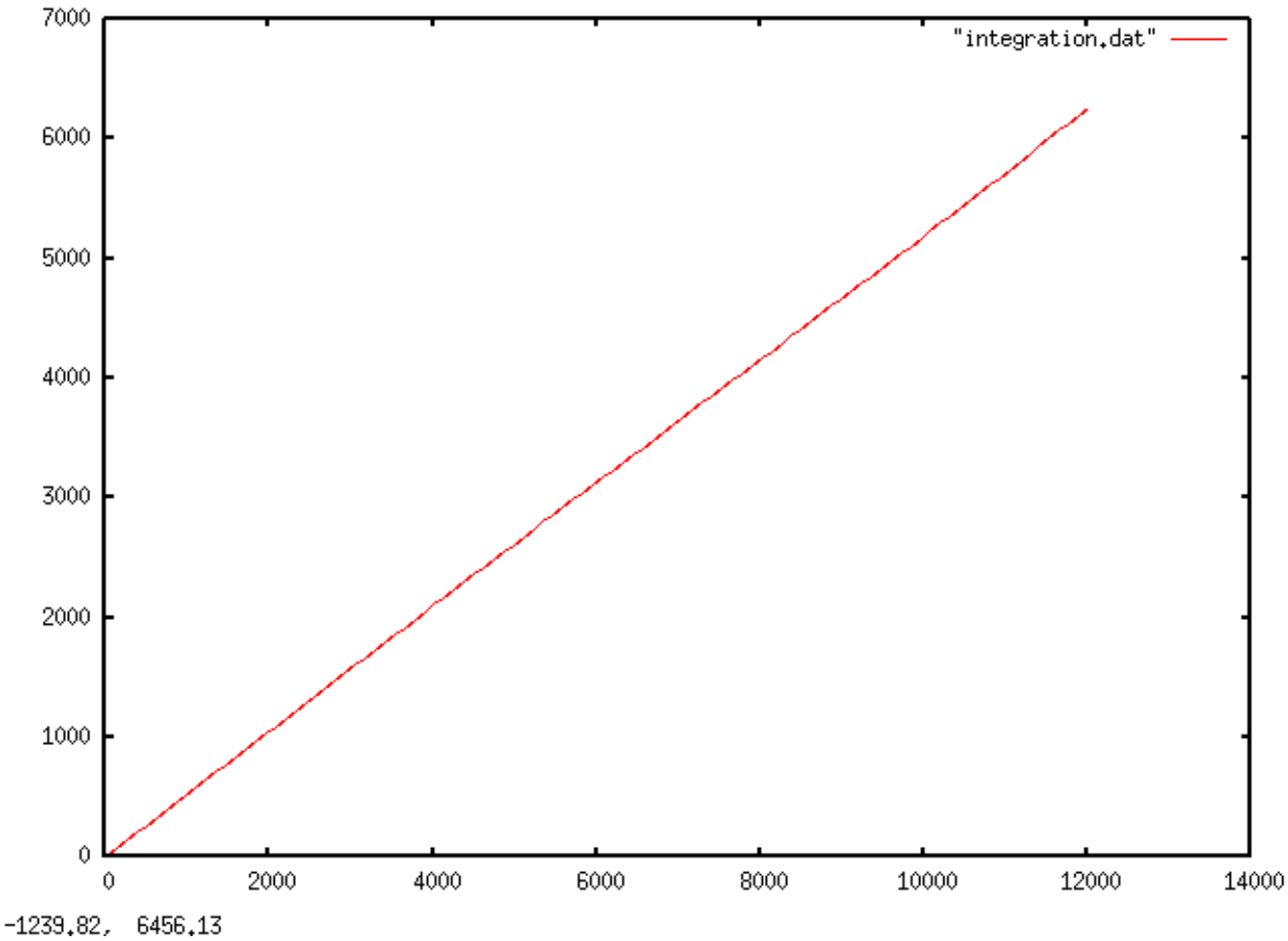

Grafico 3: Tempo impiegato all'aumentare del numero di integrazioni

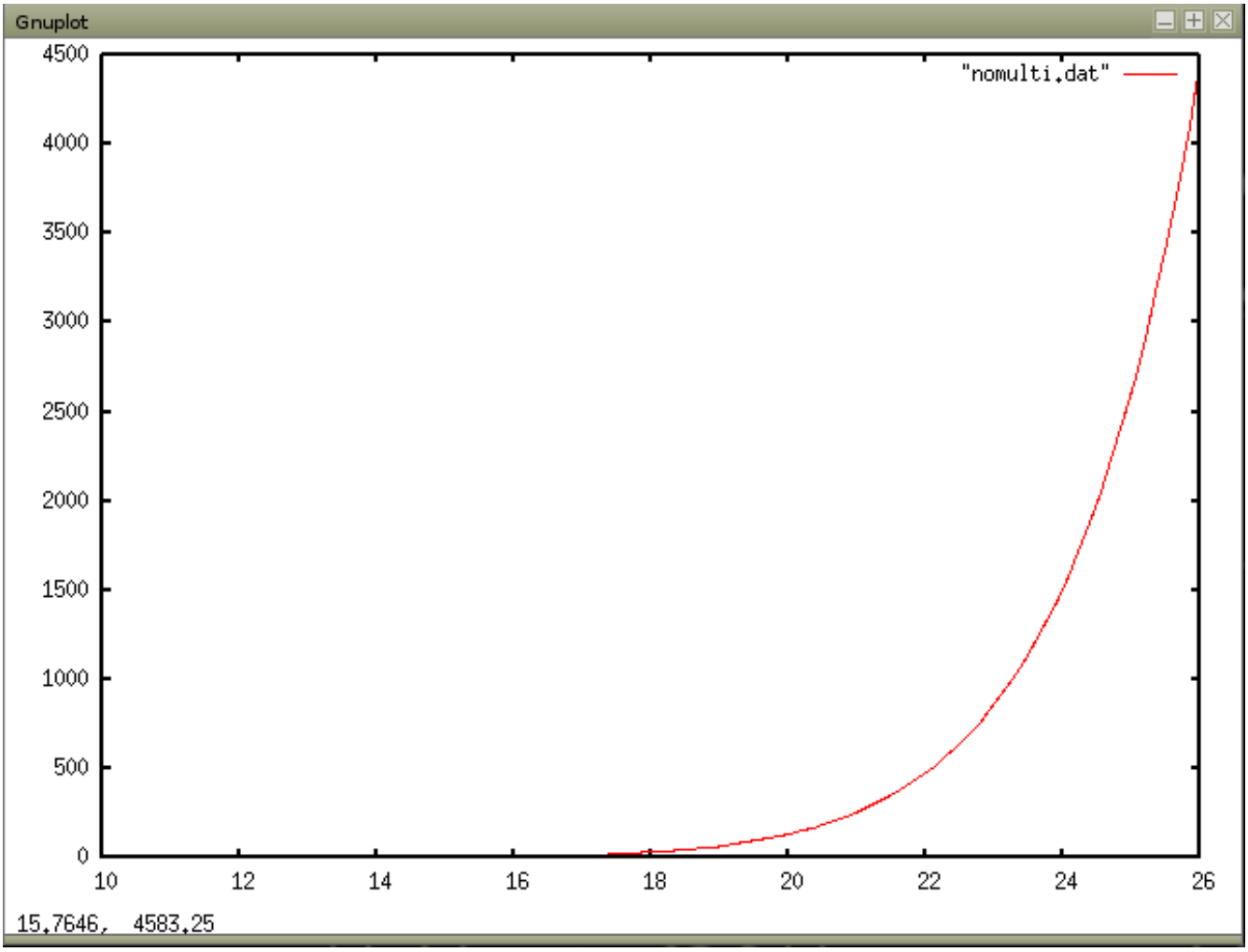

Grafico 4: Tempo impiegato a calcolare la FFT con ordine crescente, senza multithreading

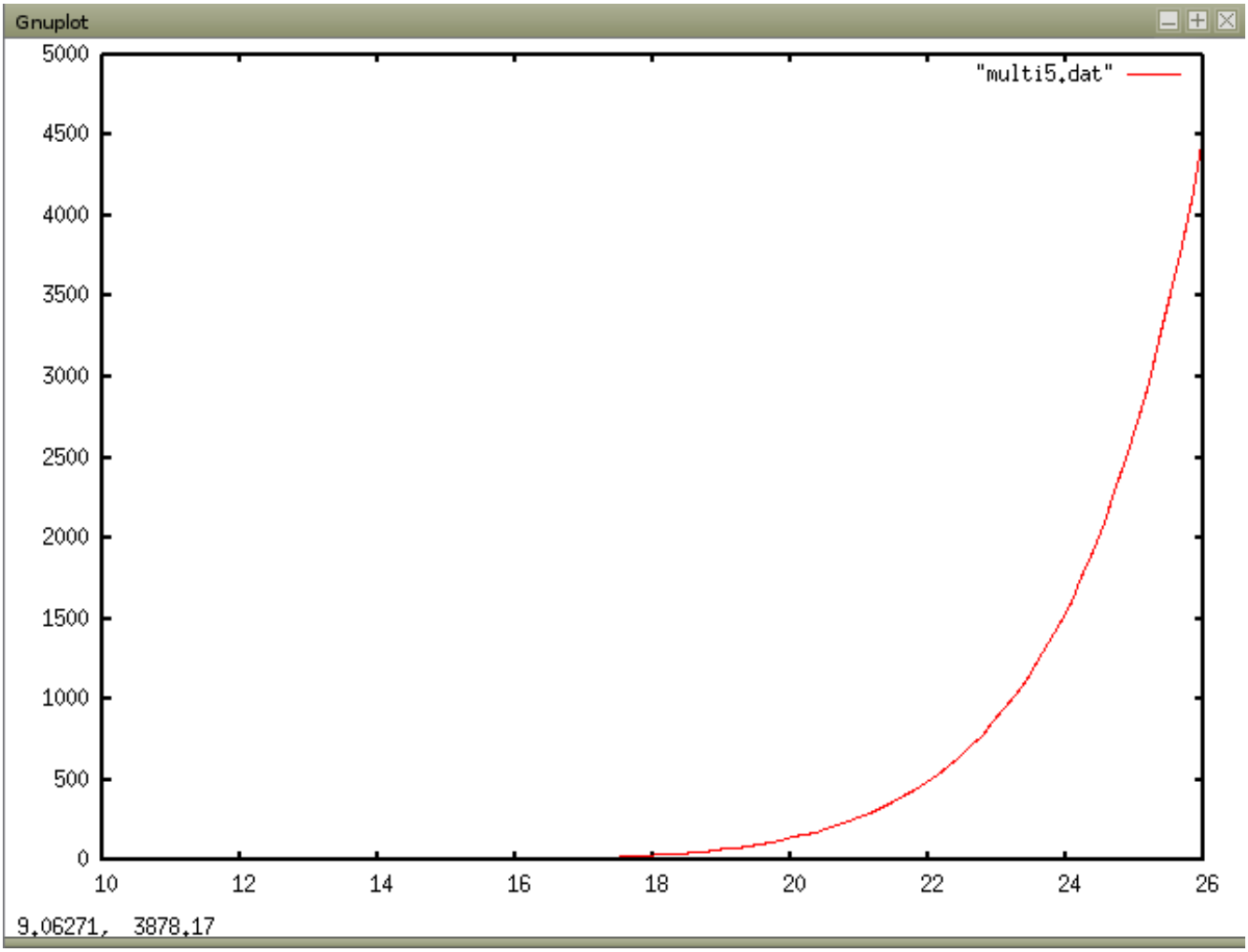

Grafico 5: Tempo impiegato a calcolare la FFT con ordine crescente, con 5 thread (libreria)

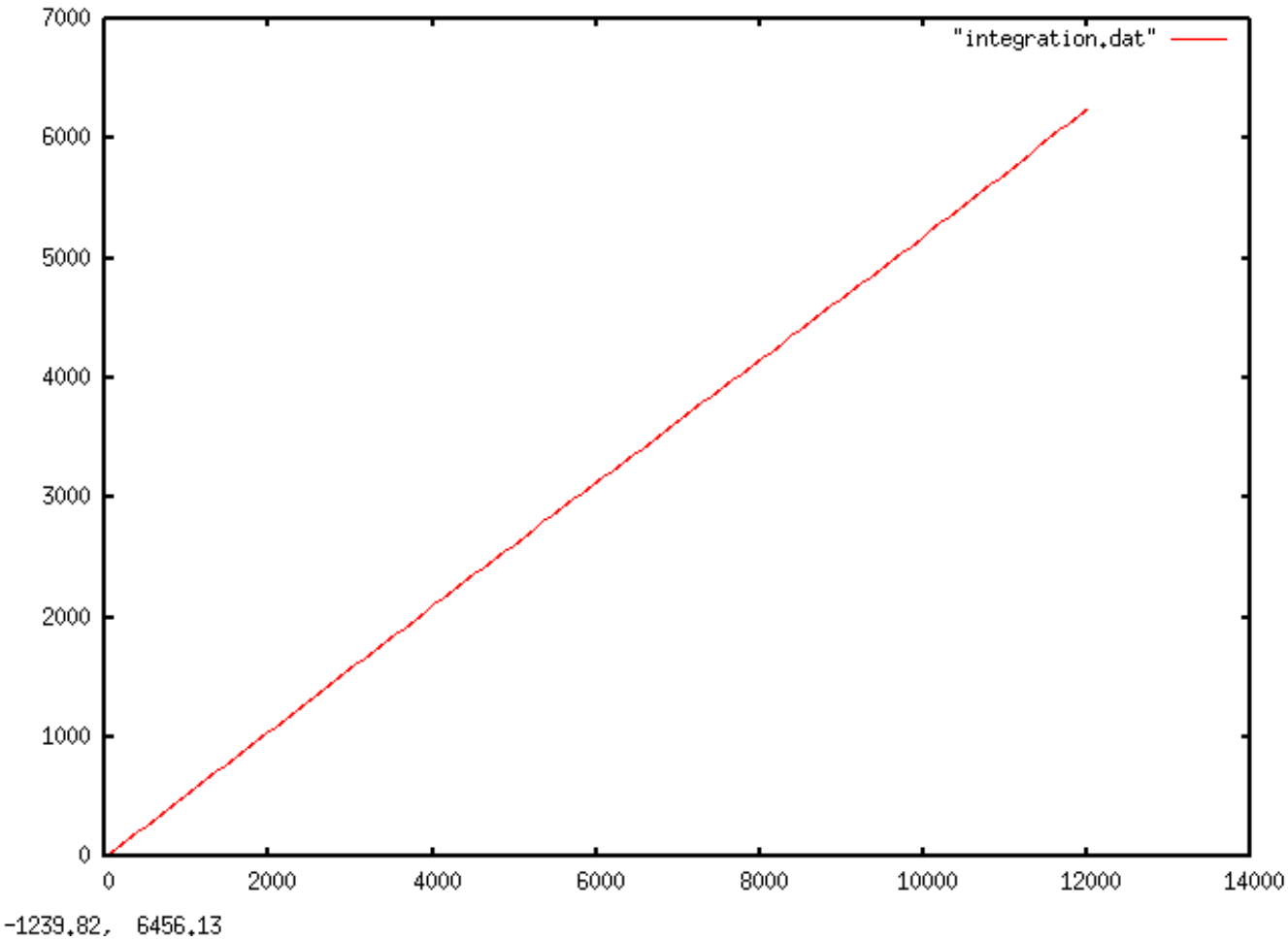

Grafico 6: Crescere del tempo all'aumentare del numero di integrazioni, senza threading

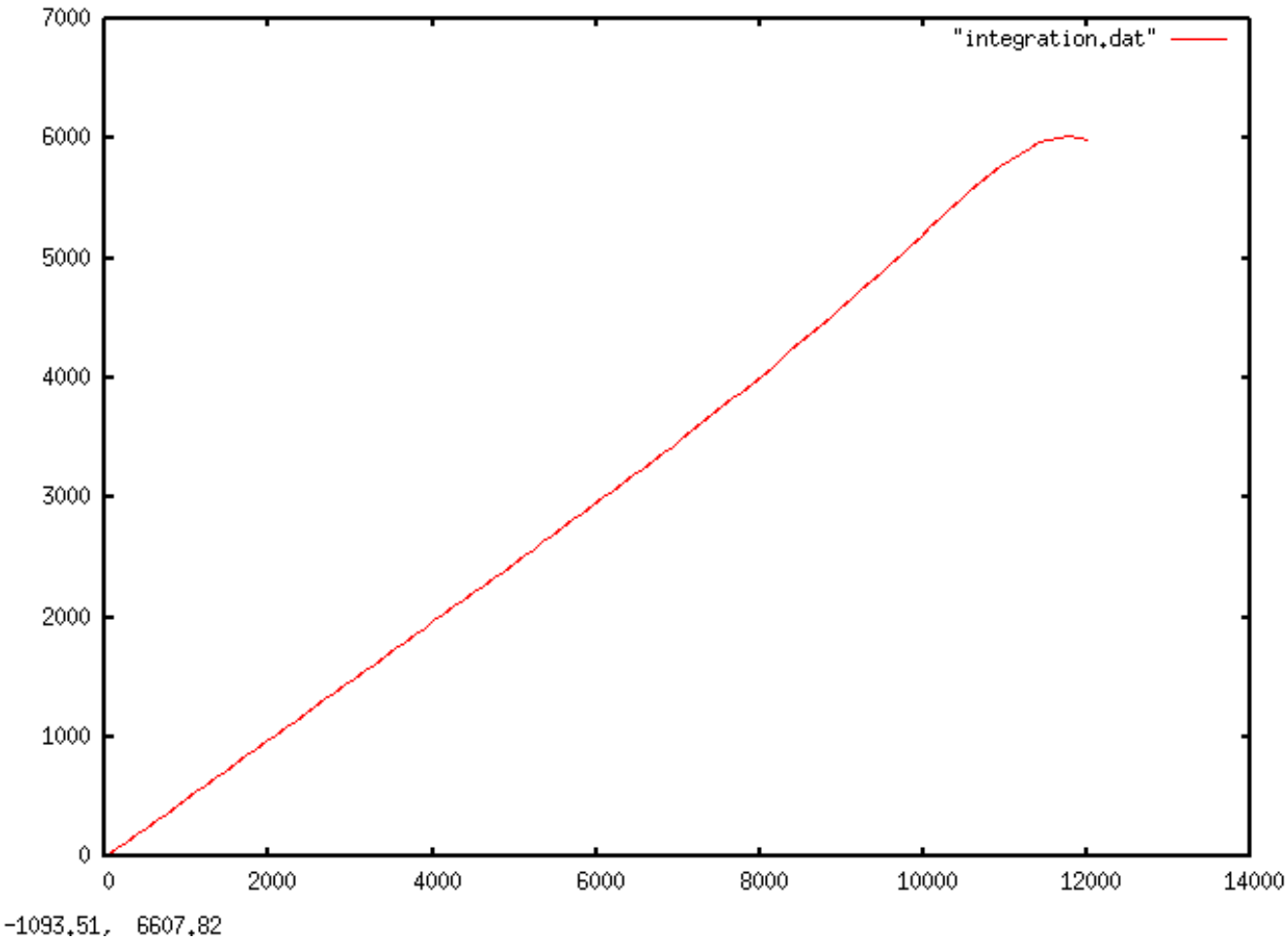

Grafico 7: Crescere del tempo all'aumentare del numero di integrazioni, con multithreading