UNIVERSITÀ DEGLI STUDI DI PADOVA
DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE
LAUREA IN INGEGNERIA INFORMATICA

Corso di Calcolo Parallelo

A.A. 2008/2009

PROF.: *Gianfranco Bilardi*

Implementazione della FFT parallela

STUDENTI: *Marco Bettiol* 586580
Ludovik Çoba 607286
Francesco Locascio 604120

19 agosto 2009

Introduzione

Un vettore x di lunghezza N può essere visto come una sequenza $x_0, ..., x_{N-1}$ di N numeri complessi. Si definisce X *trasformata discreta di Fourier (DFT, discrete Fourier transform)* di x la sequenza $X_0, ..., X_{N-1}$ espressa dalla seguente relazione:

$$X_k = \sum_{n=0}^{N-1} x_n e^{-\jmath \frac{2\pi}{N} kn} \quad k = 0, ..., N-1$$

La *FFT* o *trasformata veloce di Fourier* individua una famiglia di algoritmi in grado di calcolare la DFT in tempo $O(N \log N)$ al posto dell'usuale $O(N^2)$ ottenibile banalmente attraverso la definizione.
Molti di questi algoritmi, oltre che ad essere computazionalmente efficienti, si prestano bene anche ad un'implementazione parallela.
In questo lavoro è stata realizzata un'implementazione del noto algoritmo di Cooley-Tukey.

1 Algoritmo

L'algoritmo di Cooley-Tukey prevede di calcolare la DFT del vettore x attraverso il calcolo separato delle DFT delle sottosequenze di indice pari e dispari della sequenza originaria e quindi la loro ricombinazione attraverso *operazioni butterlfy* tra elementi in posizione analoga delle due sottosequenze per ottenere la trasformata della originale.
L'immagine seguente illustra quanto appena enunciato per una sequenza di lunghezza 8

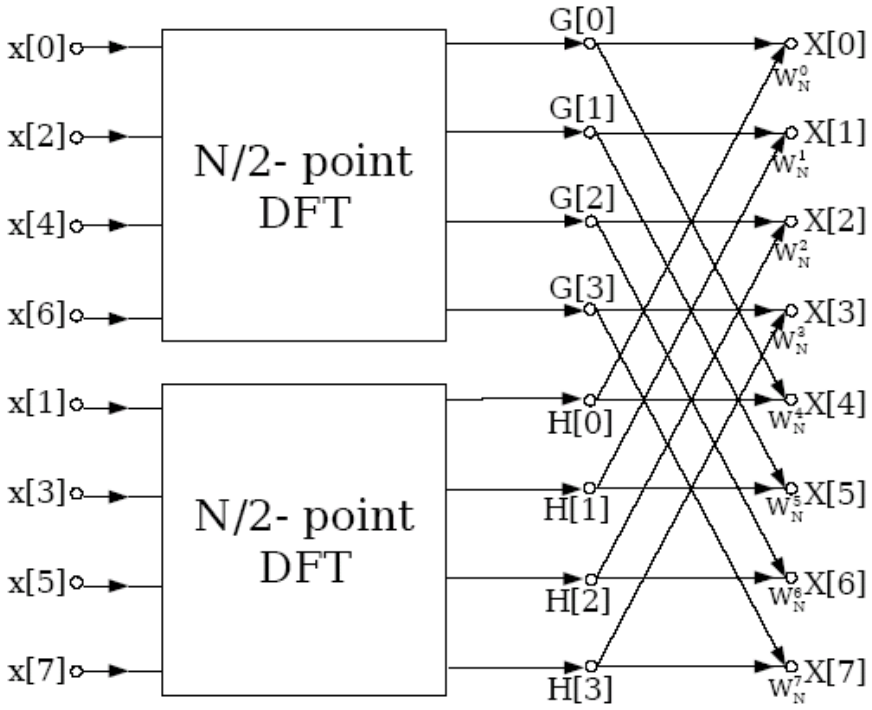


Figura 1: Ricombinazione in Cooley-Tukey per una sequenza di lunghezza 8

Questo rimescolamento dell'input iniziale, operato a ritroso fino al caso base di sequenze di lunghezza 2, permette di notare come l'algoritmo di CT sia costituito da una rete ascendente che ha come input il vettore originale rimescolato in ordine *bitreversal*.

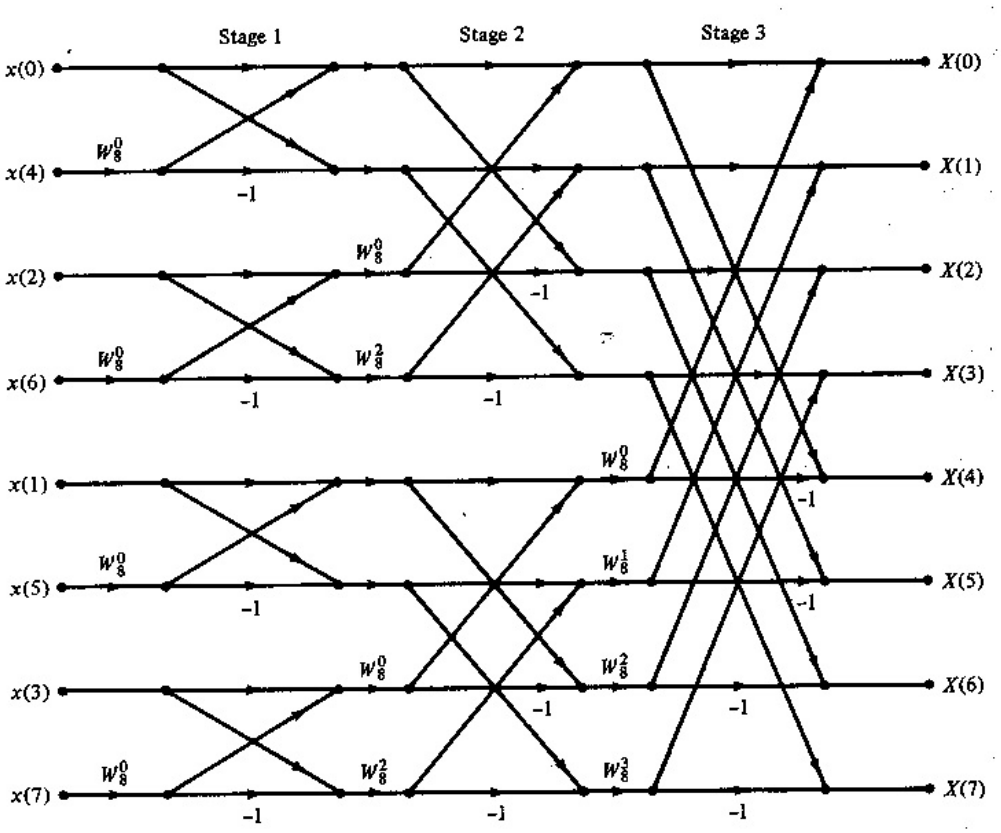


Figura 2: bitreversal di una sequenza di lughezza 8 e computazione su rete ascendente

Ora, considerando una sequenza di lunghezza N e avendo a disposizione P processori, è possibile dividere la sequenza iniziale in P blocchi da $\frac{N}{P}$ elementi. Il processore i-esimo sarà responsabile dell'i-esimo blocco. Notiamo come la computazione sia inizialmente priva di comunicazioni in quanto ogni processore dispone localmente dei dati da elaborare. Soltanto le ultime $\log_2 P$ fasi richiedono infatti l'utilizzo di dati di responsabilità di un altro processore. In queste fasi che chiameremo "esterne" (analogamente consideriamo "interne" le fasi precedenti), il processore x, utilizzando la notazione vista a lezione, comunicherà prima con $C_0 x$, quindi con $C_1 x$,... In generale, nella j-esima fase esterna, si avrà lo scambio dei rispettivi blocchi di responsabilità tra

$$x \longleftrightarrow C_j x \quad \text{per} \quad j = 0, ..., \log_2 P - 1$$

2 Implementazione

Per prima cosa è stata sviluppata un'implementazione sequenziale iterativa dell'algoritmo di CT cercando di seguire i suggerimenti visti a lezione per quanto riguarda l'ottimizzazione del codice. I cicli (3) utilizzati per la computazione sono stati modificati in modo da essere decrescenti (es: da N-1 a 0) e ordinati in modo da elaborare il vettore dei dati in maniera contigua (posizione i, i+1, e così via) al fine di ridurre gli accessi ed utilizzare nel modo migliore RAM e soprattutto cache.
Particolare attenzione è stata inoltre riposta nella gestione dei *twiddle factors*. Per elaborare una sequenza di lunghezza N è necessario calcolare $\frac{N}{2}$ *twiddle factors* $(\omega_N^0, ..., \omega_N^{\frac{N}{2}-1})$. Tuttavia soltanto nell'ultima fase vengono utilizzati tutti i *twiddle factors* mentre nelle fasi precedenti si utilizzano solo alcuni di questi (ottenibili come un sottocampionamento del vettore appena calcolato). Ad esempio nella prima fase si utilizza, banalmente, $\omega_N^0 = 1$, nella seconda ω_N^0 e $\omega_N^{\frac{N}{4}} = -j$, nella terza ω_N^0, $\omega_N^{\frac{N}{8}}$, $\omega_N^{\frac{N}{4}}$, $\omega_N^{\frac{3N}{8}}$ e così via.
Pertanto sottocampionando progressivamente il vettore *twiddle factors* mantenendo solo gli elementi in posizione dispari si è prodotto un vettore di lunghezza 2N-1 che ha in posizioni contigue i *twiddle factors* opportuni che saranno impiegati in ciascun stadio. Con uno "spreco" di $\frac{N}{2}$ di RAM ulteriormente occupata è stato pertanto ottimizzato l'accesso in RAM/cache ai fattori di ricombinazione. Questo ha fornito un ulteriore boost prestazionale molto evidente.
L'implementazione sequenziale così ottenuta è stata quindi utilizzata come base di partenza per:

1. un algoritmo ascendente con chiamate bloccanti
2. un algoritmo ascendente con chiamate non-bloccanti

2.1 Sceletro generale dell'algoritmo

Il processore con rank 0 è eletto come coordinatore del gruppo di processori e si occupa della gestione, distribuzione e raccolta dei dati elaborati.

Le fasi principali che possiamo individuare dall'algoritmo sono:

1. P_0 legge l'input da file o lo genera se richiesto
2. P_0 dispone l'input in bit-reversal
3. Ogni processore calcola localmente i *twiddle factors* e li memorizza in un vettore
4. P_0 distribuisce a ciascun processore la porzione di input di cui è responsabile attraverso una chiamata MPI_Scatter
5. Calcolo della FFT parallela prima nelle dimensioni interne e poi in quelle esterne
6. P_0 raccoglie i risultati attraverso un MPI_Gather
7. P_0 scrive l'eventuale output

2.1.1 Algoritmo ascendente bloccante

Per l'implementazione della modalità bloccante, poichè ogni processore manda dati e li riceve dallo stesso "collega" si è utilizzata la primitiva

```
MPI_Sendrecv(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf,
             recvcount, recvtype, source, recvtag, comm, status)
```

che prevede lo scambio mutuo di dati tra due processori.

2.1.2 Algoritmo ascendente non-bloccante

Come già detto ogni processore gestisce $\frac{N}{P}$ elementi memorizzati in un vettore.
Denominiamo con A e B rispettivamente la prima e la seconda metà del vettore (A e B contengono quindi $\frac{N}{2P}$ elementi ciascuno). Indichiamo inoltre con A' e B' i dati provenienti dal processore "collega". Per ogni stadio (completo) in cui è necessaria la comunicazione con altri processori questa avviene nel seguente modo:

1. Attesa per il completamento della ricezione di B' e l'invio di B
2. Aggiornamento di B
3. Invio di B e ricezione di B'
4. Attesa per il completamento della ricezione di A' e l'invio di A
5. Aggiornamento di A
6. Invio di A e ricezione di A'

In questo caso le primitive utilizzate sono quelle non bloccanti, non bufferizzate

`MPI_Isend, MPI_Irecv, MPI_Wait`

La scelta di suddividere il vettore in due metà e comunicarne prima l'una e poi l'altra è nata dall'idea di

1. limitare l'impatto del costo della gestione della comunicazione. Infatti tanto più sovrapposte sono calcolo e comunicazione (pacchetti scambiati piccoli) tanto più sforzo è richiesto alla macchina per la gestione dei buffer della comunicazione
2. cercare di limitare il ritardo dovuto alla comunicazione a quello dell'invio di $\frac{N}{2P}$ elementi + delay iniziale.

2.2 Modalità d'uso

Il software sviluppato funziona per

- $N = 2^n$; lunghezza della sequenza da elaborare
- $P = 2^p$; numero di processori impiegati
- $P < N$; quindi risulta $\frac{N}{P} \geq 2$ in modo da avere come caso base sempre operazioni tra due elementi.

Il programma prevede 2 distinte modalità di esecuzione.
La prima è utile per la valutazione delle performance dell'algoritmo, la seconda per l'effettivo impiego. Consideriamo $n = \log_2 N$

Test su sequenza nota Il nodo P_0 genera un'input di lunghezza 2^n e quindi procede con il calcolo parallelo della sua FFT.
La sintassi è:

```
./fft-asc-* test n
```

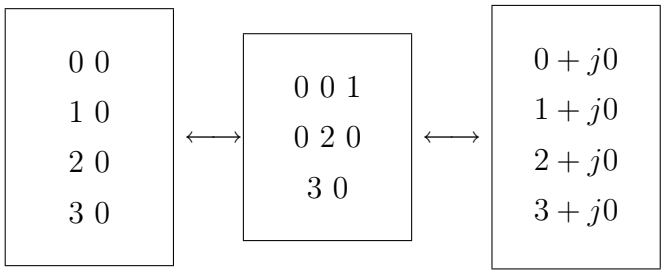
Elaborazione dell'input fornito P_0 legge da file l'input che dopo l'elaborazione viene salvato in un file di output

```
./fft-asc-* read n file_input file_output
```

dove *∈{sp, cp} per richiamare rispettivamente la versione bloccante o non bloccante.

2.3 Formato dei file elaborati

Il i file sono letti/scritti con *fscanf* / *fprintf*. I file devono essere costituiti da una sequenza di coppie di numeri intervallati da spazi o return. Per ciascuna coppia il primo elemento viene interpretato come parte la reale mentre il secondo come parte immaginaria dell'effettivo valore da elaborare.
Ad esempio:



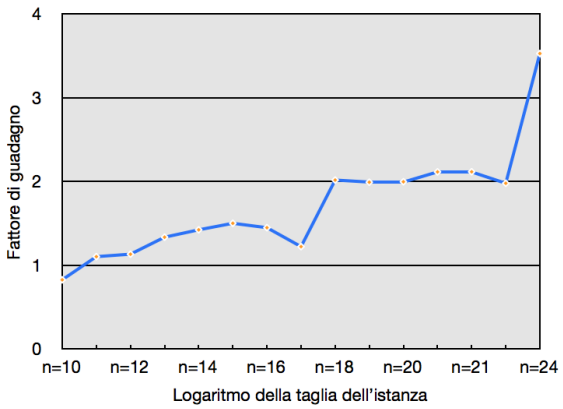
3 Prestazioni

Breve confronto tra le due implementazioni sequenziali

Notiamo come all'aumentare della taglia del problema l'ottimizzazione e la migliore gestione della cache aumentino considerevolmente le prestazioni.

Taglia	NORMALE	OTTIMIZZATO
n=10	0.000129936	0.000157723
n=11	0.000332915	0.000302535
n=12	0.000778489	0.000689069
n=13	0.001804	0.00135363
n=14	0.00406251	0.00285965
n=15	0.00910774	0.00607799
n=16	0.0228943	0.0158313
n=17	0.053337	0.0437193
n=18	0.24052	0.119367
n=19	0.679818	0.341773
n=20	1.82025	0.912718
n=21	4.35264	2.06058
n=22	10.1937	4.82321
n=23	22.957	11.6157
n=24	79.5295	22.5562

(a) 1. Tempi di calcolo assoluti in secondi tra le due implementazioni



(b) 4. Guadagno dell'algoritmo ottimizzato

3.1 Analisi Parallela

L'algoritmo parallelo "naive" (basato sul codice non ottimizzato) e quello ottimizzato sono stati eseguiti un numero di volte tra 25 e 200 (in funzione della taglia dell'input) in modo da ricavare i tempi medi di esecuzione con un'affidabilità adeguata.

I test sono stati tutti eseguiti con la comunicazione attraverso memoria condivisa abilitata ($MP\_SHARED\_MEMORY = yes$) e selezionando lo switch ad alte prestazioni per la comunicazione tra nodi ($network.mpi = switch,shared,US$)

3.1.1 Test con carico massimo sui nodi

Per prima cosa entrambi gli algoritmi sono stati lanciati con il parametro *blocking* = *unlimited* nei file .job. In questa modalità vengono allocati per ogni nodo il maggior numero di processori possibile in modo da ridurre il più possibile il tempo dovuto alle comunicazioni tra processori di nodi diversi.

Algoritmo naive: tempo complessivo di calcolo in secondi

	Numero di processori				
	1	**2**	**4**	**8**	**16**
Taglia					
n=10	0.000129936	0.000149962	0.000226345	0.00033327	0.000785987
n=11	0.000332915	0.000258177	0.000346991	0.000480192	0.00185124
n=12	0.000778489	0.00059017	0.000619149	0.00087158	0.00772419
n=13	0.001804	0.00132147	0.00128626	0.00145136	0.0312449
n=14	0.00406251	0.00284767	0.00229943	0.00302159	0.0293407
n=15	0.00910774	0.00610827	0.00471642	0.00585453	0.0356274
n=16	0.0228943	0.0154807	0.00984966	0.0119216	0.0411835
n=17	0.053337	0.0351614	0.022934	0.024873	0.0723321
n=18	0.24052	0.139784	0.0775715	0.0634866	0.162552
n=19	0.679818	0.39229	0.232645	0.155495	0.329262
n=20	1.82025	1.13313	0.761394	0.459175	0.600513
n=21	4.35264	2.852	2.1373	1.20351	1.05444
n=22	10.1937	6.49403	5.15223	2.85529	2.57329
n=23	22.957	14.59	10.9924	6.50547	5.47629
n=24	79.5295	48.7643	36.319	19.9424	15.5504

Figura 3: In verde scuro il miglior tempo. In verde chiaro i tempi descrescenti

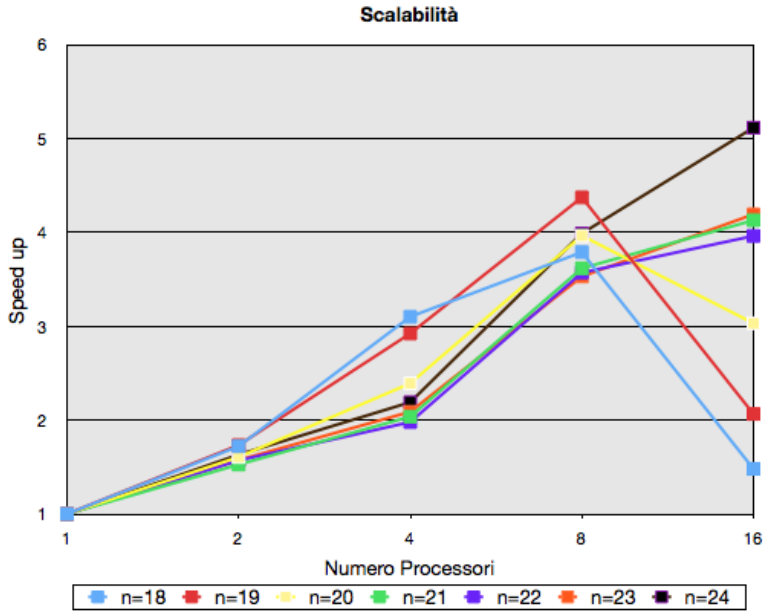


Figura 4: Algoritmo naive con *blocking* = *unlimited*, scalabilità per taglie tra 2^{18}-2^{24}

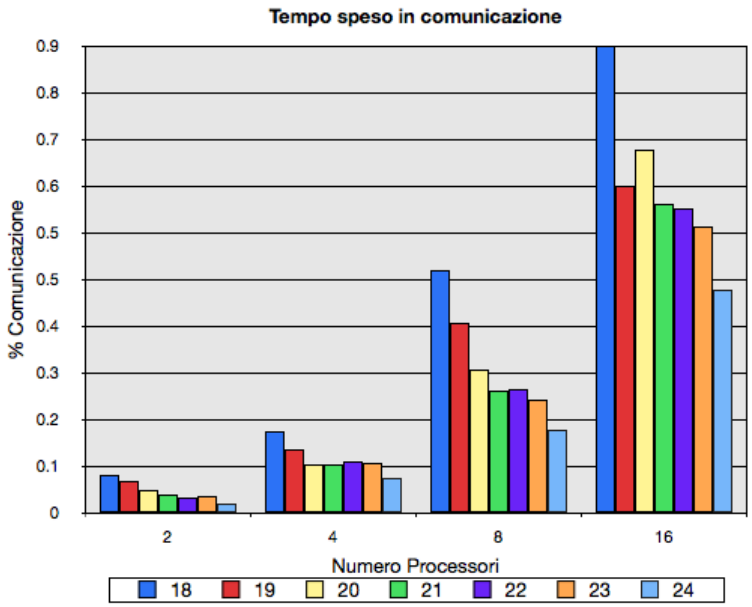


Figura 5: Algoritmo naive con *blocking* = *unlimited*, % di tempo speso in comunicazione per taglie tra 2^{18}-2^{24}

Algoritmo ottimizzato: tempo complessivo di calcolo in secondi

	Numero di processori				
	1	2	4	8	16
Taglia					
n=18	0.118242	0.0809752	0.0520556	0.0528282	0.0886482
n=19	0.346147	0.207697	0.125466	0.109042	0.176459
n=20	0.917483	0.608879	0.336681	0.242719	0.377141
n=21	2.06864	1.63816	1.18064	0.592305	0.614492
n=22	4.89169	3.76885	3.40489	1.65131	1.35391
n=23	11.6645	9.32642	7.4083	4.34579	3.65514
n=24	22.6815	19.1633	18.2875	9.16994	7.29964

Figura 6: In verde scuro il miglior tempo. In verde chiaro i tempi descrescenti

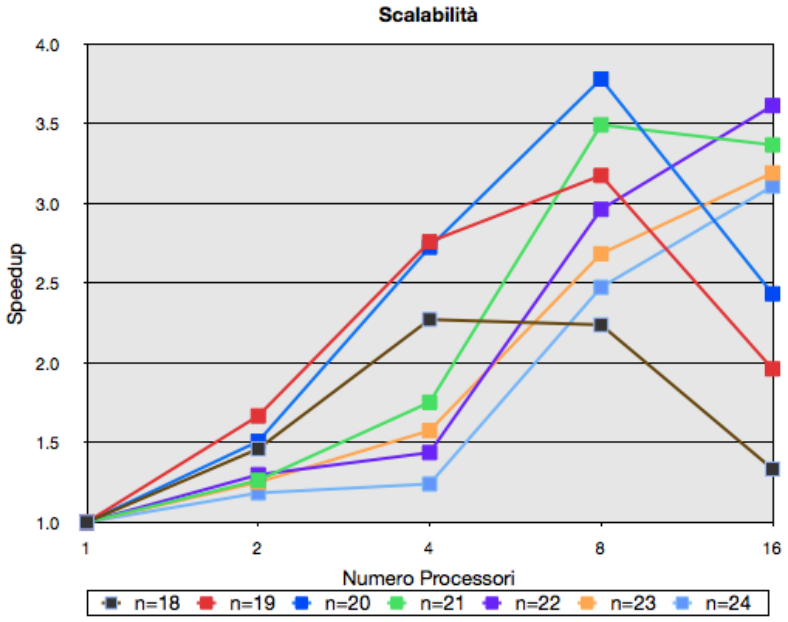


Figura 7: Algoritmo ottimizzato con *blocking* = *unlimited*, scalabilità per taglie tra 2^{18}-2^{24}

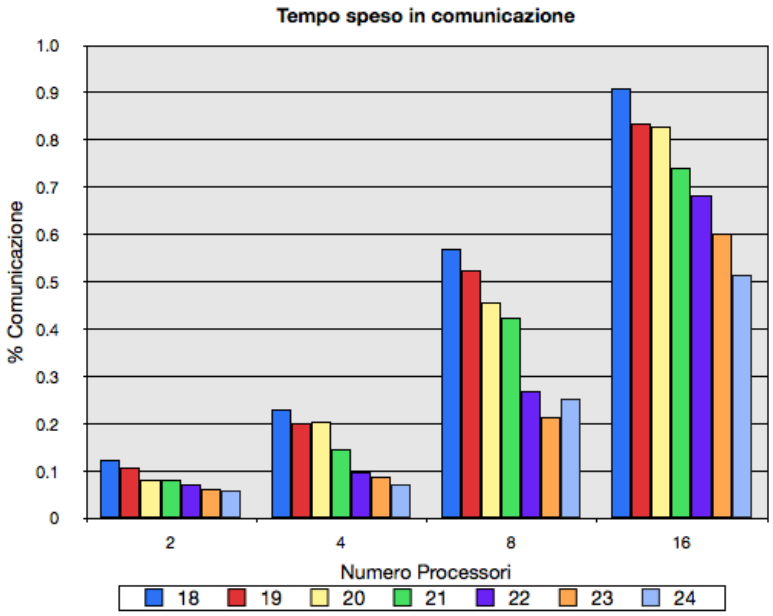


Figura 8: Algoritmo ottimizzato con *blocking* = *unlimited*, % di tempo speso in comunicazione per taglie tra 2^{18}-2^{24}

Dalle tempistiche assoulte emerge come per entrambe le versioni al crescere della taglia ci sia un continuo miglioramento fino al punto critico in cui il costo della comunicazione sovrasta quello effettivo di calcolo. Poichè l'algoritmo ottimizzato è più efficiente a livello sequenziale l'impatto della comunicazione lo penalizza maggiormente: questo è evidente dal fatto che sebbene i risultati assoluti siano sempre molto migliori, lo speedup parallelo è inferiore. Inoltre, per problemi di taglia modesta o piccola, si ha che il punto di trade-off in cui è controproducente utilizzare più processori è anticipato rispetto all'algoritmo naive.

I grafici mostrano, inaspettatamente, che con pochi processori, sono proprio i problemi con taglia n=18, n=19 a trarre il maggior beneficio dall'utilizzo di 2 e 4 processori.

Osservando la tabella dei tempi di comunicazione della tabella seguente si osserva come nel passaggi da 2 a 4 processori il tempo, pur raddoppiando, rimanga contenuto e tuttavia il tempo complessivo impiegato dall'algoritmo

migliora solo marginalmente : con un raddoppio di processori si ha un miglioramento solo lineare.

Algoritmo naive: tempo complessivo di comunicazione in secondi

	Numero di processori				
	1	2	4	8	16
Taglia					
n=10	-	0.0000781276	0.000184457	0.000298347	0.000763018
n=11	-	0.000110689	0.000264494	0.000427258	0.00181667
n=12	-	0.000198081	0.00044847	0.00076781	0.00766201
n=13	-	0.00033873	0.000799326	0.00121833	0.0146425
n=14	-	0.000654502	0.0011915	0.00244736	0.0290643
n=15	-	0.001277	0.00221974	0.00456754	0.0349638
n=16	-	0.00240612	0.00441024	0.00900482	0.0396967
n=17	-	0.00462758	0.00723319	0.017954	0.0685832
n=18	-	0.0101013	0.0122053	0.0296805	0.146114
n=19	-	0.0233875	0.028545	0.0568916	0.207139
n=20	-	0.0486471	0.070787	0.126273	0.42018
n=21	-	0.0972076	0.200624	0.281445	0.626083
n=22	-	0.19137	0.510398	0.683093	1.51098
n=23	-	0.451813	1.06641	1.42284	3.01829
n=24	-	0.888409	2.44638	3.1574	6.68983

Passando ad 8 processori si verifica però un miglioramento più che doppio rispetto al caso con 4. Questo ci fa supporre che ci sia qualche collo di bottiglia che deve essere considerato (vedremo in seguito).

Come ultime osservazioni notiamo come nell'algoritmo ottimizzato non sia ancora arrivato al limite di scalabilità con le taglie di problemi prese in esame: infatti lo speedup per 8 o 16 processori risulta inferiore al crescere della taglia (es: per n=22,n=23,n=24) ma taglie maggiori hanno un trend di miglioramento più promettente. Sebbene il costo di comunicazione sia proporzionalmente maggiore per taglie inferiori (e questo dovrebbe pertanto penalizzarle) si ha che, raddoppiando il numero di processori dimezza il rapporto $\frac{N}{P}$, il guadagno a livello sequenziale è più che doppio (anche triplo in alcuni casi) e questo boost dovuto ad un utilizzo migliore della cache fornisce risultati significativi.

Possiamo dire, prima di passare all'analisi del caso seguente, che i grafici non rispecchiano minimamente quanto ci si aspetterebbe dalla teoria e sono assai dissimili dal caso ideale.

3.1.2 Test con carico minimo sui nodi

Questi test sono stati condotti con *blocking = 1* per 2 e 4 processi. Avendo solo al più 6 nodi per 8 e 16 processi si è dovuto ripiegare su *blocking = 2* nel primo caso e *blocking = 4* equivalente ad *unlimited* nel secondo (pertanto con 16 processi si ricade nel test precedente).

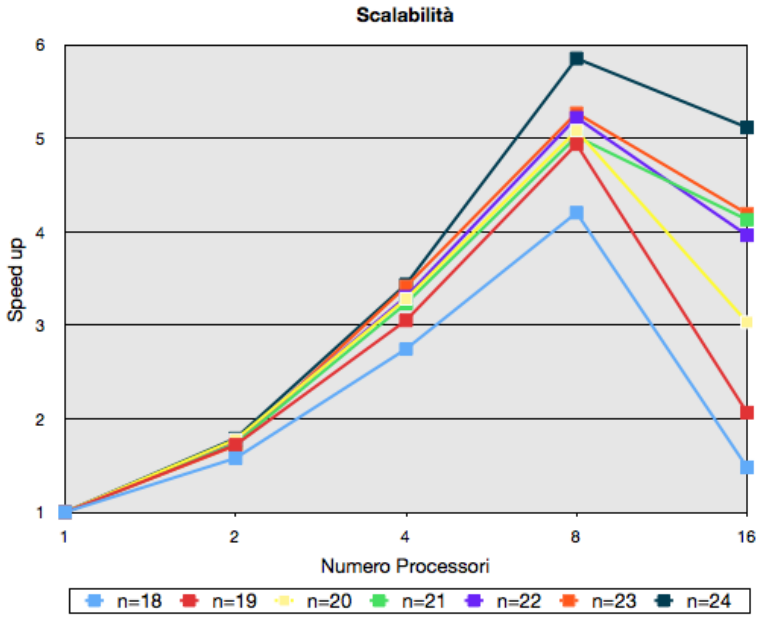


Figura 9: Algoritmo naive con *blocking = "minimo"*, scalabilità per taglie tra 2^{18}-2^{24}

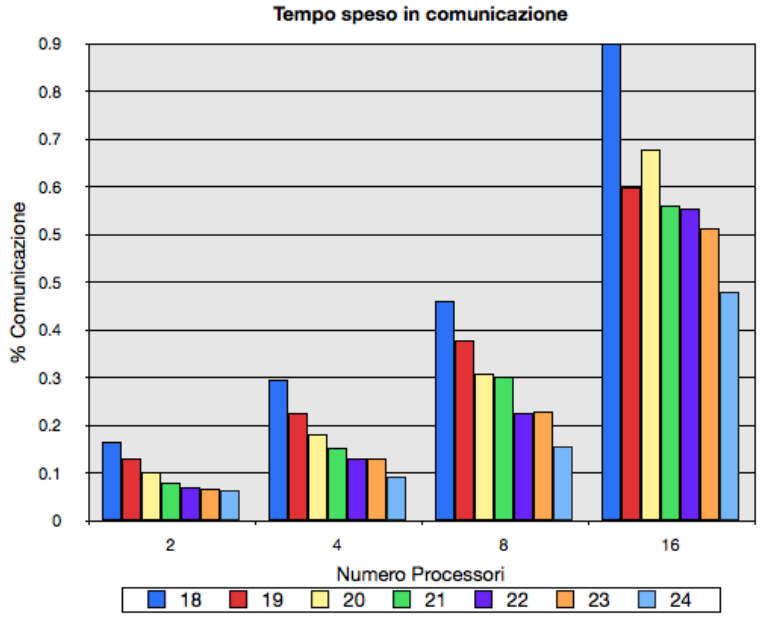


Figura 10: Algoritmo naive con *blocking* = *"minimo"*, % di tempo speso in comunicazione per taglie tra 2^{18}-2^{24}

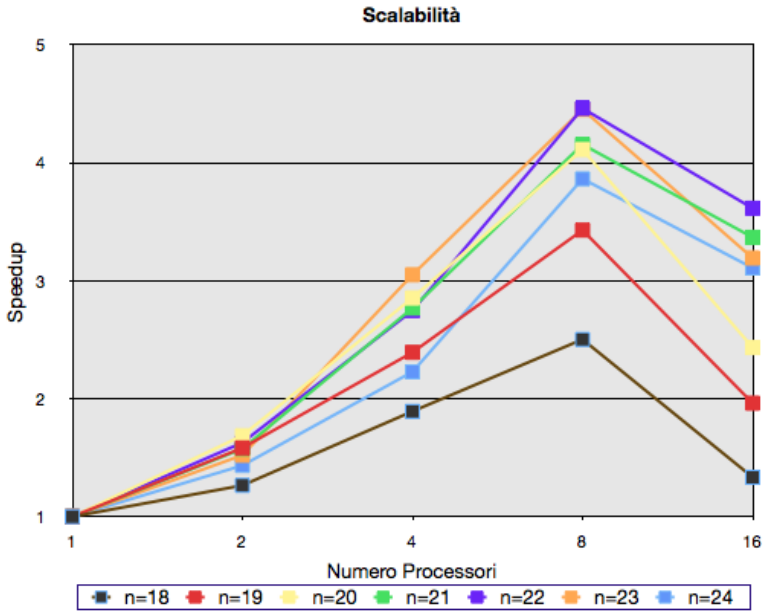


Figura 11: Algoritmo ottimizzato con *blocking* = *"minimo"*, scalabilità per taglie tra 2^{18}-2^{24}

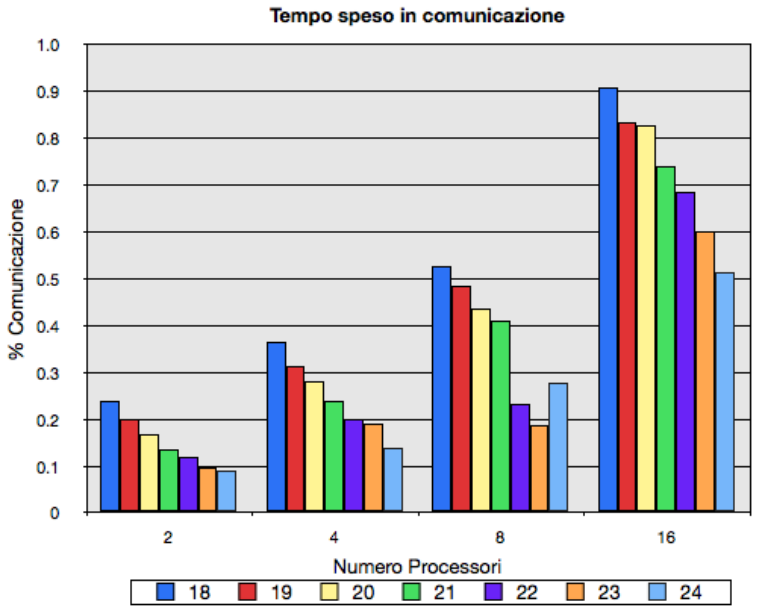


Figura 12: Algoritmo ottimizzato con *blocking* = *"minimo"*, % di tempo speso in comunicazione per taglie tra 2^{18}-2^{24}

Nei grafici ora l'effettto "sella" del passaggio da 4 a 8 processori è sparito.

L'algoritmo naive ripercorre l'andamendo del grafico di scalabilità ideale in cui a taglie maggiori corrispondono speedup più elevati. Notiamo come, sebbene il costo di comunicazione con 2 processori sia ora più elevato (circa il doppio) in quanto per la comunicazione viene ora sfruttata lo switch, le prestazioni fornite siano migliori rispetto a quelle ottenute in precedenza al variare di tutte le taglie/ numero di processori impegati. Questo è possibile poichè ora, visto che non abbiamo più contesa tra i processori per l'accesso alla RAM (almeno con 2 e 4 processi), l'algoritmo sequenziale opera con piena efficienza. Notiamo altresi osservando la tabella dei tempi sottostante come il tempo di comunicazione rimanga invariato al crescere del numero di processori impiegati da 2 a 8.

Algoritmo naive: tempo complessivo di comunicazione in secondi

	Numero di processori				
Taglia	**1**	**2**	**4**	**8**	**16**
n=18	-	0.0226757	0.0232123	0.023731	0.146114
n=19	-	0.0464662	0.0453859	0.0468016	0.207139
n=20	-	0.0922557	0.0896293	0.0991871	0.42018
n=21	-	0.177082	0.185297	0.236428	0.626083
n=22	-	0.365746	0.358138	0.397229	1.51098
n=23	-	0.8069	0.793064	0.892307	3.01829
n=24	-	2.49426	1.93924	1.89317	6.68983

Ora entrambi gli algoritmi con soli 8 processori forniscono risultati migliori di quanto ottenuto nel test precedente.

Osservando la tabella dei tempi di calcolo totali (sotto) per l'algoritmo ottimizzato con *blocking* = *"minimo"* notiamo come il costo di comunicazione via rete, essendo molto elevato per le taglie di problema più piccole, renda l'esecuzione dell'algoritmo in questa configurazione controproducente per n=10,...,n=14 e sia poco scalabile per n=15,...,17. Il mantenersi stabile

Algoritmo ottimizzato: tempo complessivo di calcolo in secondi

	Numero di processori			
	1	**2**	**4**	**8**
Taglia				
n=10	0.00012386	0.00024509	0.0002312	0.00028197
n=11	0.00026562	0.00040552	0.00047829	0.00041947
n=12	0.00061414	0.00084179	0.00074716	0.0008455
n=13	0.00132134	0.00170171	0.00144698	0.00133847
n=14	0.00282619	0.00332304	0.00280186	0.00254316
n=15	0.00603222	0.00664814	0.00531045	0.00479623
n=16	0.0157337	0.016336	0.0105193	0.00917623
n=17	0.0429662	0.0395373	0.0282537	0.0190364
n=18	0.118242	0.0934808	0.0624862	0.0472584
n=19	0.346147	0.218727	0.144714	0.100984
n=20	0.917483	0.543995	0.321588	0.22322
n=21	2.06864	1.31276	0.747594	0.498002
n=22	4.89169	3.00635	1.78098	1.09688
n=23	11.6645	7.66828	3.82528	2.62123
n=24	22.6815	15.8213	10.1824	5.8719

del costo di comunicazione fa si che vi siano dimensioni dell'istanza

(n=14,n=15,n=16) per cui l'utilizzo di 2 processori è penalizzante mentre l'utilizzo di ben 8 risulta essere il caso più vantaggioso.

3.1.3 Note su FFT Sequenziale e Cache

Durante l'implementazione dell'algoritmo è emersa la critica dipendenza tra le prestazioni ottenute dall'algoritmo FFT-DIT2 rispetto alla dimensione della cache della macchina su cui viene lanciato.
L'algoritmo implementato infatti, per la natura stessa degli operatori *butterfly*, risulta assolutamente non-cachefriendly.
L'accesso in memoria non è locale: ad ogni stadio dell'algoritmo un valore dell'input viene letto/sovrascritto una ed una sola volta. Quando l'accesso ai dati non è spazialmente localizzato e la struttura di computazione è rigida la classica modalità di mapping

$$cache\ address = memory\ address \text{ modulo } cache\_size$$

perde di efficacia. Nel nostro caso si verifica quando la distanza tra i dati da elaborare (potenza di due) è vicina alla dimensione della cache (anch'essa potenza di due).
Proponiamo qui di seguito i grafici sullo Cache Miss e MFlops per illustrare la drammaticità del problema.
In entrambi i casi le performance diminuiscono drasticamente quando la taglia del problema supera $2^{15} = 32768$ (che corrisponde a un vettore di *twiddle factors* di esattamente 64KB: pari alla dimensione della cache L1 del processore) per mantenersi successivamente costanti fino a 2^{22} (vettore di 8MB pari alla cache L2).
La diminuzione praticamente lineare delle performance per $n = 16$ visibile in Figura 13 è conseguenza raddoppio del numero di *twiddle factor* utilizzati ad ogni passo.
Le performance inferiori, a parità di stage, per le istanze di taglia maggiore sono probabilmente dovute allo swap in/out dei twiddle factor il cui address

mapping entra in conflitto con quello dei valori nel vettore da elaborare.
Una volta oltrepassata la soglia critica il Loads per Miss si stabilizza sui $5,92$ più di 100 volte peggiore rispetto agli stage in-cache. Analogamente per gli stage out-of-cache si hanno circa 81/60 Mflops/s.

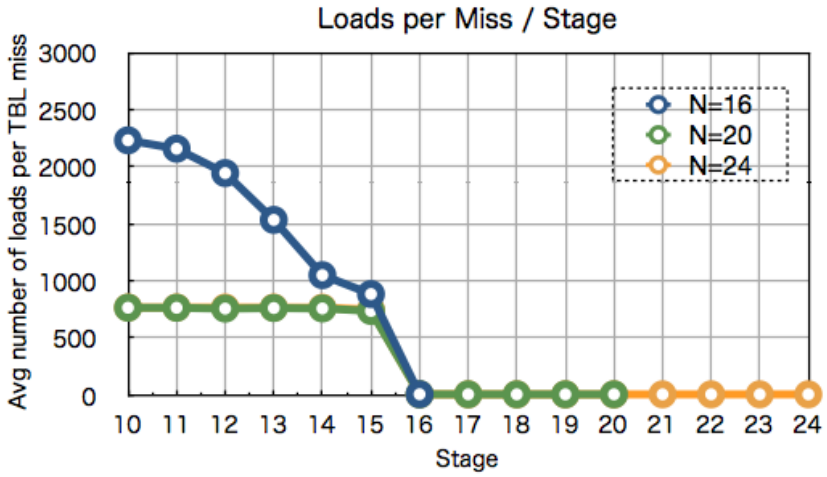


Figura 13: Caricamento di dati utili nei registri / accessi in ram. Maggiore è migliore

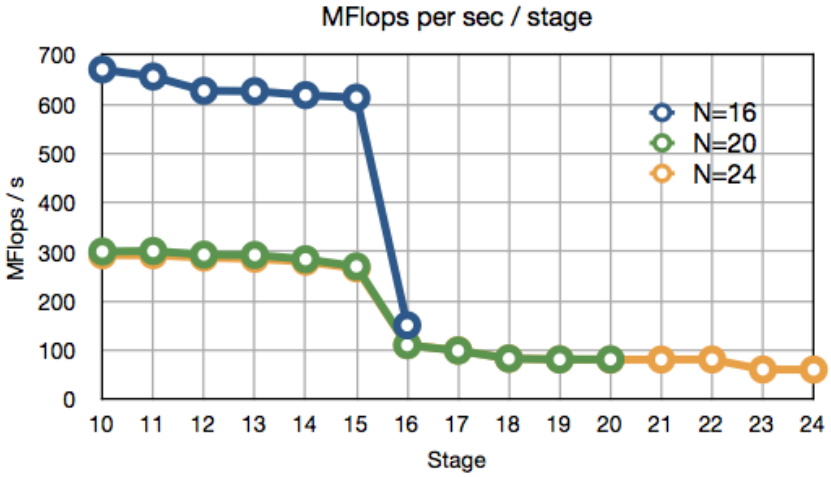


Figura 14: Potenza di calcolo sfruttata. Maggiore è migliore

Notiamo come $n = 20$ e $n = 24$ abbiano esattamente lo stesso andamento.

Conclusioni

Dai test condotti è emersa la critica dipendenza dell'algoritmo $\mathcal{FFT}$ rispetto alla banda con la RAM. Infatti esso non gode di località spaziale e deve continuamente scorrere tutti gli elementi del vettore da elaborare. Le macchine parallele dotate di nodi con processori veloci e accesso alla memoria a bus condiviso evidenziano particolarmente questo limite.

Come conseguenza di ciò si è mostrato come, con lo stesso algoritmo, in funzione della taglia del problema da risolvere e delle caratteristiche tecniche della macchina (banda con la memoria, banda di rete, latenza di rete,prestazioni floating point, dimensioni delle cache) possa risultare conveniente oppure sconvente concentrare il calcolo in un nodo o distribuirlo omogeneamente su tutto il calcolatore.

Nel nostro caso per taglie di problema "piccole", indicativamente fino a n=16, conviene concentrare il calcolo su un nodo mentre per taglie maggiori è conveniente distribuirlo il più possibile.

Dalle due implementazioni algoritmi (ottimizzato e non) che condividono la parte di comunicazione ma differiscono nel codice sequenziale è emerso come codici meno ottimizzati risultino più scalabili. Tuttavia le loro performance assolute (almeno nel nostro caso) rimangono inferiori rispetto ad implementazioni sequenzialmente efficienti.
Con un'implementazione più efficiente infatti è possibile ottenere le stesse performance assolute ma utilizzando un minor numero di processori/risorse.

Per ultimo mostriamo le performance assolute che è possibile ottenere selezionando opportunamente il miglior risultato al variare della taglia del problema e della modalità di *blocking* e i corrispondenti MFlips/s totali spremuti alla macchina.

Migliori prestazioni assolute ottenute (in secondi)

Taglia	Numero di processori 1	2	4	8
n=10	***0.00012386***	0.00013174	0.00021177	0.00028197
n=11	0.00026562	***0.00023768***	0.00036661	0.00041947
n=12	0.00061414	***0.0005588***	0.00059675	0.0008455
n=13	0.00132134	***0.00117567***	0.00127375	0.00133847
n=14	0.00282619	***0.0023679***	0.00237923	0.00254316
n=15	0.00603222	0.00492627	***0.00454904***	0.00479623
n=16	0.0157337	0.0129345	***0.00895791***	0.00917623
n=17	0.0429662	0.0331265	0.0229555	***0.0190364***
n=18	0.118242	0.0809752	0.0520556	***0.0472584***
n=19	0.346147	0.207697	0.125466	***0.100984***
n=20	0.917483	0.543995	0.321588	***0.22322***
n=21	2.06864	1.31276	0.747594	***0.498002***
n=22	4.89169	3.00635	1.78098	***1.09688***
n=23	11.6645	7.66828	3.82528	***2.62123***
n=24	22.6815	15.8213	10.1824	***5.8719***

Figura 15: Tabella con i migliori tempi ottenuti con l'implementazione ottimizzata al variare di taglia,processori e *blocking*. In verde i risultati sequenziali. In viola quelli con *blocking ūnlimited*. In azzurro *blocking ūnlimited*. In bianco nessun miglioramento. In grassetto il miglior tempo a parità di taglia

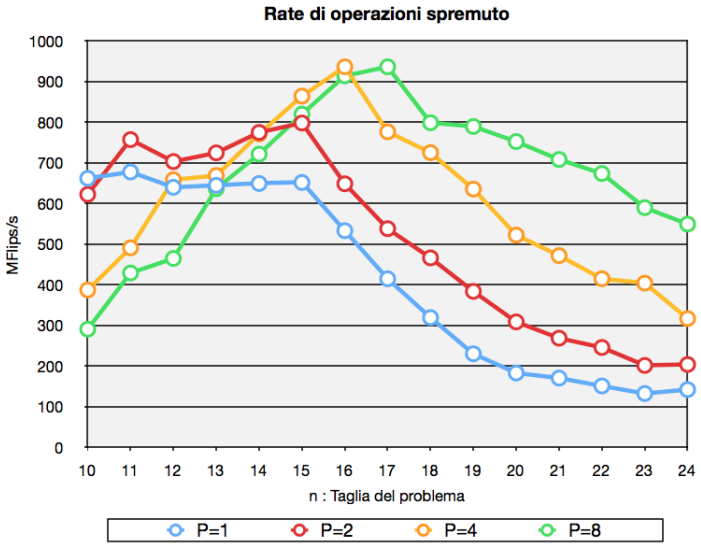


Figura 16: MFlips/s al variare di taglia e #processori