Performance di un cluster Linux M.P.I. in configurazione spettrometro digitale.

Andrea Orlati[1]

1 - Istituto di Radioastronomia.

Ottobre 2004

IRA 365/04

Capitolo 1: Introduzione.

1.1 Clusters vs supercomputers.

Recentemente lo sviluppo tecnologico nel campo dei microprocessori, delle reti d'intercomunicazioni digitale tramite rame o fibra ottica e, più in generale del mondo dei personal computer, ha costituito una notevole spinta, per le grandi istituzioni che hanno necessità di grosse risorse di calcolo, a collegare in rete computer "commerciali" in configurazione di cluster.

Un cluster può essere definito come un insieme di computer (nodi) che cooperano per il calcolo di funzioni, che per dimensioni o per quantità di risorse richieste, non possono essere trattati da calcolatori tradizionali. Il mondo dell'HPC (High Performance Computing), tempo fa dominio delle macchine MIMD (Multiple Instruction Multiple Data) quali i Mainframe o comunque di sistemi monolitici, già da qualche anno è sempre più popolato da cluster di PC Linux. Le ragioni sono molteplici:

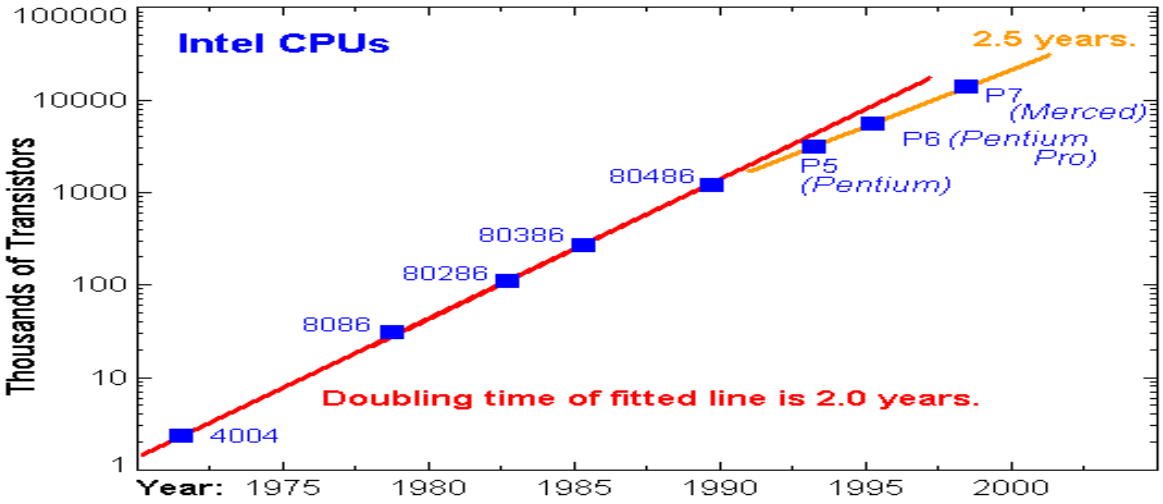
1. I cluster offrono buone prestazioni ad un decimo del costo dei Mainframe. Ciò consente non solo un forte risparmio iniziale, ma anche di tenere aggiornato il parco macchine con costi minimi.
2. La legge di Moore[1], anche se non pienamente rispettata lascia sperare d'avere performance migliori a costi sempre minori (vedi Figura 1).
3. Sono sicuramente più scalabili rispetto ad un sistema chiuso. L'aggiunta di un nodo non implica infatti nessuna modifica all'architettura generale.
4. I PC hanno un rapporto $\frac{GFlops}{Space + KJoule}$ migliore. La questione del calore prodotto è sempre più pressante con l'aumentare delle frequenze operative dei microprocessori; riuscire a dissiparlo è perciò una questione sempre più critica per il corretto funzionamento delle macchine. Il rischio è di perdere investimenti di svariate migliaia d'euro per il guasto di una semplice ventola.
5. Il sistema operativo Linux, a contrario dei sistemi proprietari (Solaris, AIX) ai quali sono legate le tecnologie di tipo monolitico, non richiede forti competenze specifiche ed è gratuito.
6. Lo sviluppo del software richiede uno sforzo iniziale ma definitivo: se anche le macchine dovessero essere cambiate, il sistema operativo rimarrebbe lo stesso e di conseguenza anche il software applicativo.

Per questa serie di motivi si è fatto il tentativo d'assemblare un cluster di minima per misurarne le prestazioni in una delle configurazioni (programmato come) più usata in radioastronomia, quella di uno spettrometro digitale. La prova è stata fatta senza acquistare alcun materiale ma usando semplicemente l'hardware già esistente alla stazione Medicina e software GNU[2].

[1] La legge dice che il volume d'informazione su una data quantità di silicio è approssimativamente raddoppiato ogni due anni.

[2] General Pubblic License. Il software che aderisce a questa tipo di licenza può essere liberamente copiato, usato, o modificato secondo le proprie esigenze.

Figura 1 – Quantità di transistor integrata nei processori Intel. L'andamento si discosta di poco da quello previsto dalla legge di Moore.



In questo rapporto verranno riassunte le fasi del lavoro: prima il punto di vista sistemistico, poi quello di programmazione e infine sarà dato un sunto ed un'interpretazione delle prestazioni ottenute.

1.2 Performance a confronto.

E' bene cercare di capire la differenza di prestazioni che un processore "general purpose" può offrire nei confronti di un hardware specializzato. Per far questo è stata scelta un'applicazione (la DFT per rimanere in tema) ed è stata fatta una stima dei tempi di risposta sulle seguenti piattaforme:

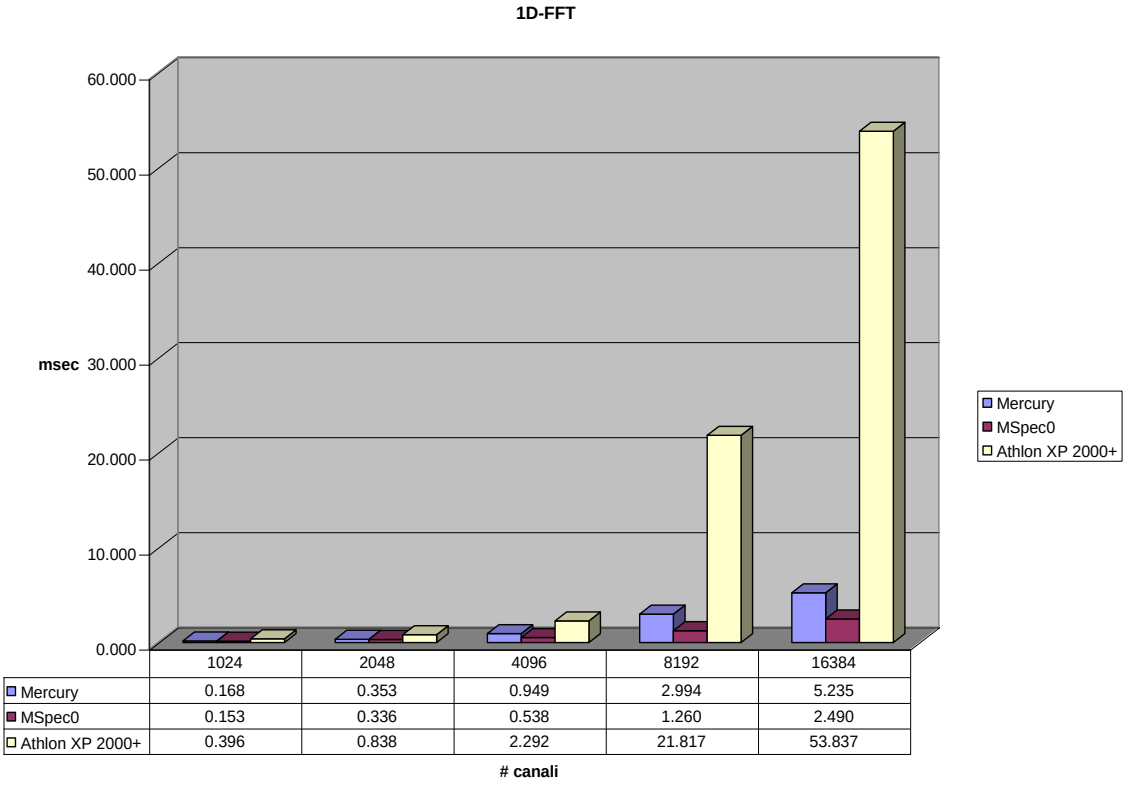
1) **Mspec0**. Spettrometro digitale in uso alla stazione di Medicina, basato su DSP Sharp.
2) **Mercury**. Macchina multi processore, prodotta dall'azienda americana Mercury, basata su due chip RISC[3] MPC7400 a 400 MHz, dotati di unità aritmetico logica ALTIVEC di tipo vettoriale. Ogni processore è equipaggiato con 64 MB di ram, 32KB di cache di primo livello e 2MB di cache di secondo livello.
3) **Athlon XP 2000+**. Macchina desktop equipaggiata con processore AMD della famiglia x86 a 1667 MHz, 32KB cache L1 e con 512 MB ram.

Le prove sono state condotte variando il numero di canali con cui viene calcolata la DFT, in altre parole aumentando le dimensioni del problema. La Figura 2 riporta i risultati ottenuti. Come ben si può vedere, la macchina desktop ha prestazioni paragonabili alle altre due finché i dati hanno dimensioni contenute e possono essere alloggiati nella cache (fino a 4096 canali); oltre questo limite le prestazioni degradano in modo sensibile. Il rapporto tra millisecondi impiegati e numero di canali, invece, rimane abbastanza costante per le altre due piattaforme.

E' evidente che se si trascura il fattore "costo" le piattaforme con hardware dedicato sono ancora di gran lunga preferibili. Se però viene valutato il rapporto tra prezzo e prestazioni e si considera che negli ultimi anni i processori di tipo commerciale hanno avuto un grosso incremento in termini di potenza di calcolo (vedi Figura 3), ecco che l'idea di un cluster di PC torna ad avere un fondamento pratico anche in questo tipo d'applicazioni.

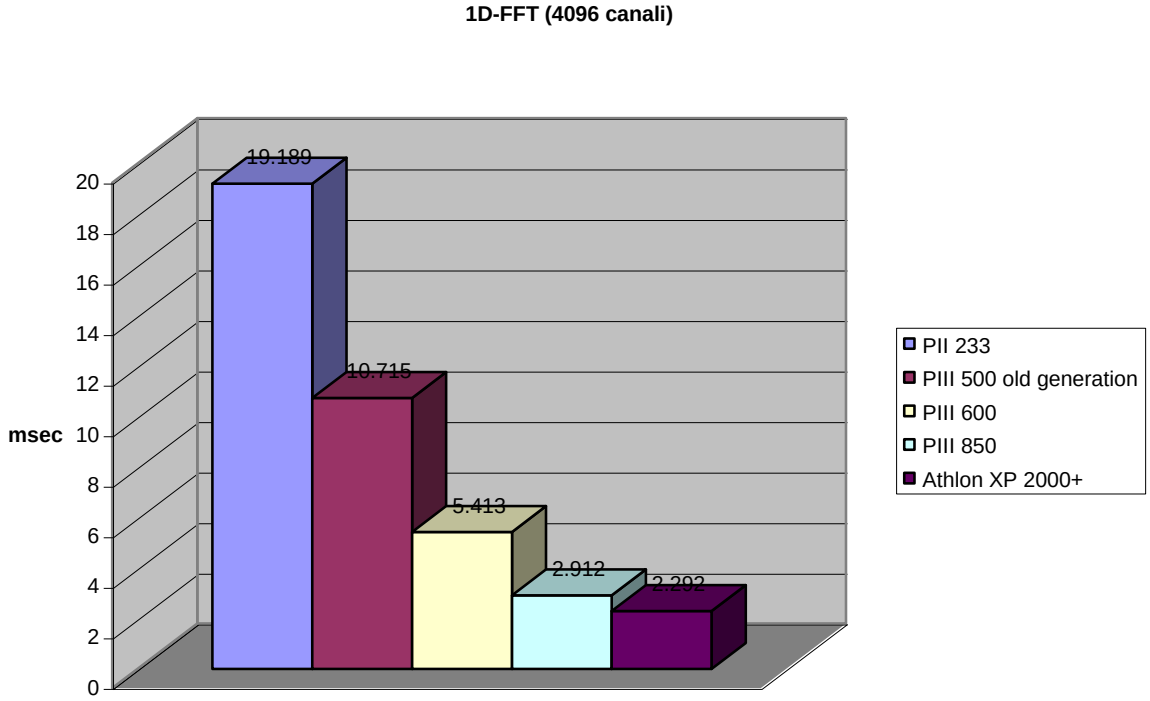
[3] Reduced Instruction Set Computer. Sono microprocessori che riconoscono ed eseguono un numero limitato di istruzioni. Uno dei vantaggi di questa architettura è che le istruzioni vengono eseguite più velocemente.

Figura 2 - Tempi impiegati (ordinate) su tre diverse piattaforme per produrre DFT a grandezza crescente (ascisse).



	1024	2048	4096	8192	16384
Mercury	0.168	0.353	0.949	2.994	5.235
MSpec0	0.153	0.336	0.538	1.260	2.490
Athlon XP 2000+	0.396	0.838	2.292	21.817	53.837

Figura 3 - Tempi d'esecuzioni di processori di varie generazioni per calcolare una DFT da 4096 canali.



Capitolo 2: Configurazione del cluster.

2.1 Configurazione hardware.

Il cluster è stato allestito con tre PC e con materiale già a disposizione. Dal punto di vista logico una macchina è stata disegnata essere il nodo master mentre le altre due i nodi di calcolo. La Tabella 1 riassume la configurazione hardware delle tre macchine impiegate.

Tabella 1 – Equipaggiamento hardware dei nodi.

Nome	Logico	CPU	Frequency	RAM	Chipset/southbridge	Disk
PcMed14	Master	Intel PentiumIII	500 MHz	128MB	Intel i440BX Intel 82371	40 GB e 10 GB
Giovedì	Slave	Intel PentiumIV	2.4 GHz	512MB	SiS648FX SiS963L	80 GB
Venerdi	Slave	Intel Celeron	2.0 GHz	256MB	SiS651 SiS962L	40 GB

I nomi assegnati alle singole macchine sono quelli con i quali sono identificate nella LAN di Medicina durante il loro normale servizio presso la stazione. Per questo motivo ai nodi sono stati assegnati due indirizzi IP, uno relativo alla rete locale, l'altro relativo alla sotto rete virtuale creata appositamente per far dialogare privatamente il cluster. Questa scelta ha consentito di configurare i nodi in modo che considerassero "trusted[4]" tutte i pacchetti IP provenenti dalle macchine facenti parte di quella sottorete. La Tabella 2 riassume queste assegnazioni.

Tabella 2 – Indirizzi ed identificazione di rete dei nodi.

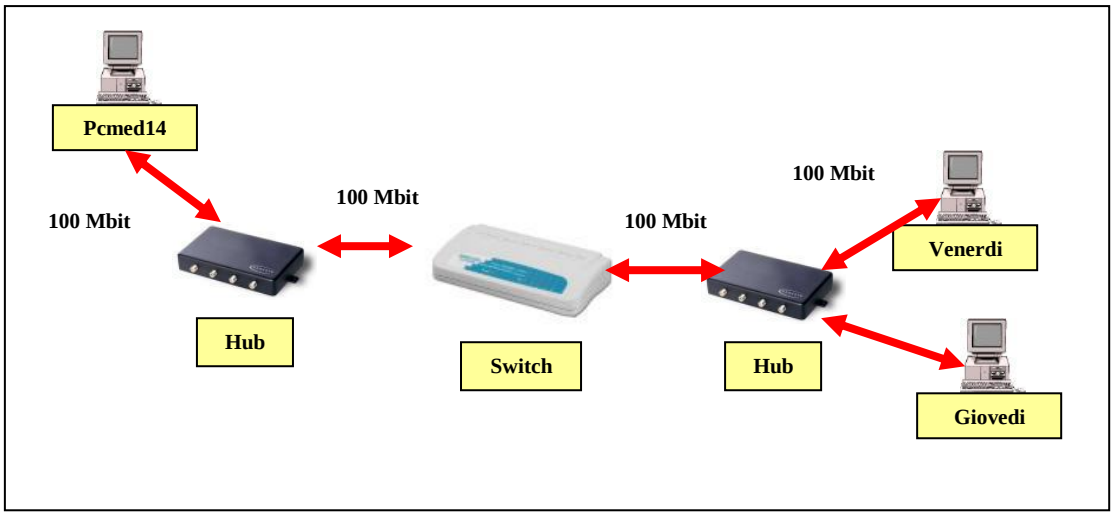
Nome completo	Indirizzo primario	Indirizzo sottorete "virtuale"
Pcmed14.med.ira.cnr.it	192.167.189.82	192.168.1.3
Venerdi.med.ira.cnr.it	192.167.189.121	192.168.1.2
Giovedì.med.ira.cnr.it	192.167.189.120	192.168.1.1

Tutte e tre le macchine sono equipaggiate con una scheda di rete 10/100 Ethernet utilizzata a 100 Mbit. La topologia dei collegamenti è esemplificata dalla Figura 4, dalla quale si nota che i due nodi di calcolo fanno parte di un ramo diverso da quello del nodo master. Questa configurazione, nata dalla disposizione originale delle macchine, che ricordo erano già in servizio al momento della stesura di questo lavoro, non è ottimale. Al fine di ottenere prestazioni migliori sarebbe stata sicuramente preferibile una rete privata tra i nodi, senza che ci fosse traffico aggiuntivo al di fuori di quello tra i nodi stessi ad occupare banda.

Sono state valutate le prestazioni del sottosistema formato da interfaccia di I/O, scheda di rete, e connessione di rete per ogni PC. A tale scopo è stato usato il programma *TTCP* distribuito con la versione di Linux installata sulle macchine. Questo programma va lanciato sulle due macchine di cui si vuole testare la connessione di rete; su una va configurato il modo ricevitore sull'altra il modo trasmettitore. Il programma trasmettitore non fa altro che spedire al ricevitore un buffer dati (di

[4] Sono I pacchetti per i quali non si applicano misure di sicurezza quali firewall o altro.

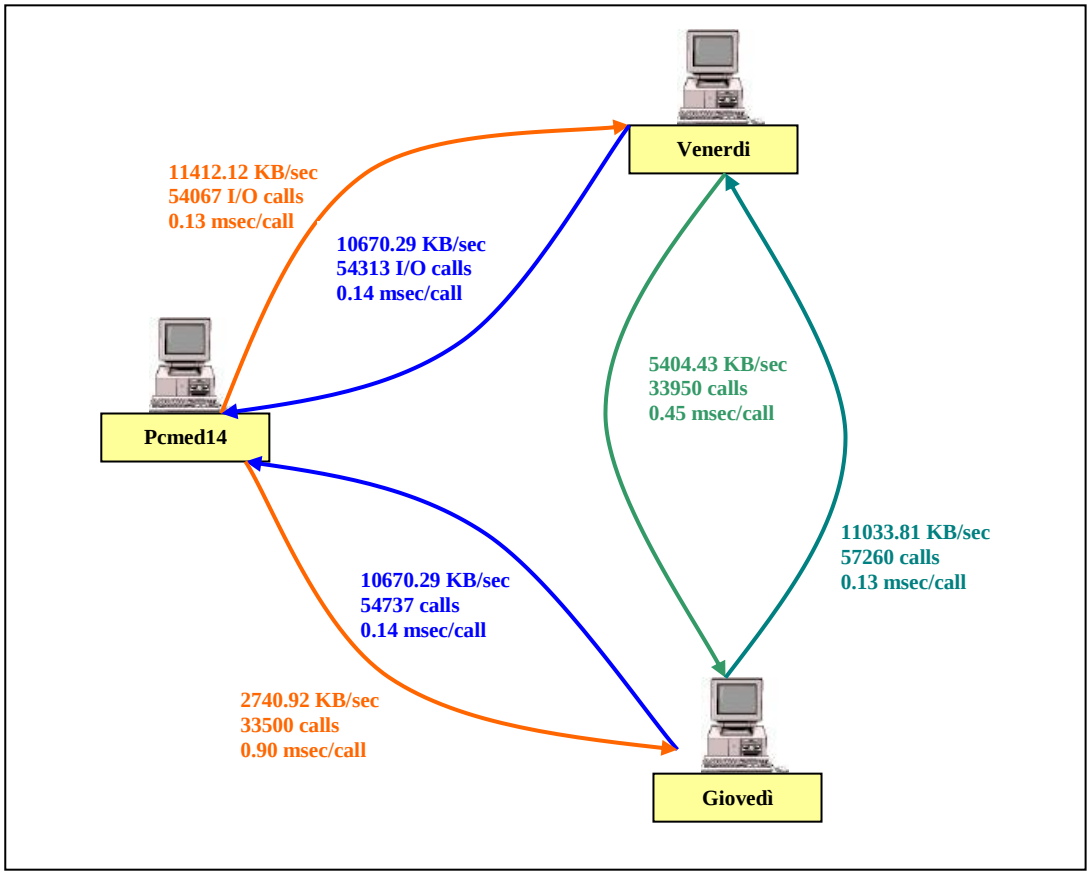
Figura 4 – Geometria dei collegamenti tra i nodi che compongono il cluster.



dimensione impostabile) per un determinato numero di volte. Alla fine del processo vengono ritornate le statistiche di tutto il processo. Per questa misura si è usato un buffer di 81920000 byte utilizzando questi due comandi *ttcp –r –s –n10000* per il modo ricevitore e *ttcp –t –s –n10000* per quella trasmettitore.

I risultati sono sintetizzati in Figura 5. Per tutte le possibili direzioni di comunicazione sono state fatte le misure con la linea impegnata da un solo trasmettitore alla volta; per ogni arco viene riportata rispettivamente, la banda in KB/sec, il numero d'operazioni di I/O fatte dalla macchina ricevente e il tempo medio (in millisecondi) per ognuna di queste operazioni. L'interconnessione di rete è basta su interfaccia a 100 Mbit con un throughput teorico di 12.5 MB al secondo. In quattro

Figura 5 – Misura delle performance delle connessioni di rete.



casi su sei la misura si è avvicinata parecchio al limite, negli altri due (quelli in cui era Giovedì la macchina ricevente) il valore misurato è di gran lunga inferiore. Escluso un problema di linea: in trasmissione, infatti, i risultati sono simili agli altri, e verificata la configurazione sia a livello di BIOS[5] sia di sistema operativo, l'unica conclusione rimasta è che la causa delle scarse prestazioni mostrate sia dovuta al chipset montato sulla macchina.

Come si vedrà di seguito, le scarse performance di rete mostrate da giovedì causeranno un forte sbilanciamento del cluster con conseguente decadimento generale delle prestazioni.

2.2 Configurazione software.

Il sistema operativo che si è scelto di installare sulle macchine è Fedora Core 1 con kernel versione 2.4.22. La scelta è ricaduta su questo pacchetto perché è una distribuzione gratuita supportata direttamente dalla Red-Hat. Inoltre con la Red-Hat 7.3 si erano verificati alcuni problemi nella configurazione delle schede di rete SiS900 integrate nelle schede madri di Giovedì e Venerdi.

Prima di installare il sistema operativo i tre dischi sono stati formattati e partizionati, tramite il wizard (*Disk Druid)* del programma d'installazione come illustrato dalla Tabella 3. Su Pcmed14, nelle partizioni con NTFS è stata installata anche una versione di Windows 2000 Pro. Il doppio sistema operativo è stato gestito tramite il boot loader di Windows, mentre il secondo disco è utilizzato come condivisione file tra i due sistemi. In ambiente Linux il disco è stato montato editando il file */etc/fstab* aggiungendo la linea

/dev/hdd1 /mnt/scambio vfat uid=500,gid=100,umask=002 0 0

Tabella 3 – Sintesi del partizionamento dei dischi delle tre macchine.

Giovedi				
Disco	**Dim(MB)**	**File System**	**Mount Point**	**Partizione**
/dev/hda1	2996	Ext2	/	Principale
/dev/hda2	50132	Win Fat32	/c	Principale
/dev/hda3	12001	Ext2	/usr	Principale
/dev/hda5	12001	Ext2	/home	Espansa
/dev/hda6	1027	Swap		Espansa
Venerdi				
Disco	**Dim(MB)**	**File System**	**Mount Point**	**Partizione**
/dev/hda1	2000	Ext2	/	Principale
/dev/hda2	23000	Win Fat32	/c	Principale
/dev/hda3	6000	Ext2	/usr	Principale
/dev/hda5	6000	Ext2	/home	Espansa
/dev/hda6	1027	Swap		Espansa
Pcmed14				

[5] Basic Input/Output Stream. Parte software integrata che contiene il codice per avviare la macchine e per gestire i dispositivi come tastiera, disco e interfaccia di I/O. Nei computer moderni è solitamente alloggiato in una memoria flash che può essere quindi aggiornata.

Disco	Dim(MB)	File System	Mount Point	Partizione
/dev/hda1	7172	NTFS		Principale
/dev/had2	3831	Ext2	/	Principale
/dev/hda3	522	Swap		Principale
/dev/hda4	7172	NTFS		Espansa
/dev/hda5	14337	NTFS		Espansa
/dev/hda6	7052	Ext2	/usr	Principale
/dev/hadd1	10000	Win Fat32	/mnt/scambio	Principale

L'indirizzo IP primario (vedi Tabella 2) è stato configurato durante la fase d'installazione del sistema operativo, quello secondario è stato impostato manualmente. Per Venerdi, ad esempio. il comando da linea dei comandi, con privilegi di root è: *ifconfig eth0:1 192.168.1.2 netmask 255.255.255.0*. Questa istruzione non fa altro che mappare sulla prima interfaccia di rete (eth0) un alias come IP virtuale. Per rendere permanente la modifica, vale a dire far si che ad ogni successivo riavvio la macchina risponda anche all'indirizzo secondario si è aggiunto uno script di configurazione. Il nuovo script è stato prodotto partendo da quello di configurazione dell'interfaccia primaria modificando i campi necessari. Questa la sequenza dei comandi:

cd /etc/sysconfig/network-script/

cp ifcfg-eth0 ifcfg-eth0:1

Il file ifcfg-eth0:1 di Venerdi è riportato sotto a titolo di esempio:

 # Silicon Integrated Systems [SiS]|SiS900 10/100 Ethernet

DEVICE=eth0:1

BOOTPROTO=static

BROADCAST=192.168.1.255

HWADDR=00:0D:87:16:0A:6A

IPADDR=192.168.1.2

NETMASK=255.255.255.0

NETWORK=192.168.1.0

ONBOOT=yes

TYPE=Ethernet

Infine con privilegi di root si è riavviato il servizio di rete: *service network restart*. La risposta al comando *ifconfig* è la seguente:

eth0 Link encap:Ethernet HWaddr 00:0D:87:16:0A:6A

inet addr:192.167.189.121 Bcast:192.167.189.255 Mask:255.255.255.0

UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1

RX packets:214908 errors:0 dropped:0 overruns:0 frame:0

TX packets:36163 errors:0 dropped:0 overruns:0 carrier:0

collisions:0 txqueuelen:1000

RX bytes:33921408 (32.3 Mb) TX bytes:33348282 (31.8 Mb)

Interrupt:11 Base address:0xe800

eth0:1 Link encap:Ethernet HWaddr 00:0D:87:16:0A:6A

inet addr:192.168.1.2 Bcast:192.168.1.255 Mask:255.255.255.0

UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1

RX packets:214908 errors:0 dropped:0 overruns:0 frame:0

TX packets:36163 errors:0 dropped:0 overruns:0 carrier:0

collisions:0 txqueuelen:1000

RX bytes:33921408 (32.3 Mb) TX bytes:33348282 (31.8 Mb)

Interrupt:11 Base address:0xe800

lo Link encap:Local Loopback

inet addr:127.0.0.1 Mask:255.0.0.0

UP LOOPBACK RUNNING MTU:16436 Metric:1

RX packets:7190 errors:0 dropped:0 overruns:0 frame:0

TX packets:7190 errors:0 dropped:0 overruns:0 carrier:0

collisions:0 txqueuelen:0

RX bytes:26786343 (25.5 Mb) TX bytes:26786343 (25.5 Mb)

La configurazione di rete è stata completata modificando sulle tre macchine il file */etc/hosts* in modo da renderle "note" l'una alle altre ed evitare così, continui accessi al DNS[6] per le comunicazioni via rete. La Tabella 4 riporta i tre file.

Tabella 4 – Contenuto dei file per la risoluzione dei nomi per ciascuna macchina del cluster.

Pcmed14	
127.0.0.1	Localhost.localdomain localhost
192.168.1.3	Pcmed14.med.ira.cnr.it Pcmed14
192.168.1.2	Venerdi.med.ira.cnr.it Venerdi
192.168.1.1	Giovedì.med.ira.cnr.it Giovedi
192.167.189.82	Pcmed14.med.ira.cnr.it Pcmed14
Giovedi	
127.0.0.1	Giovedi.med.ira.cnr.it Giovedi localhost.localdomain localhost
192.168.1.2	Venerdi.med.ira.cnr.it Venerdi
192.168.1.3	Pcmed14.med.ira.cnr.it Pcmed14
Venerdi	

[6] Domain Name Server. Servizio fornito che serve a tradurre i nomi di dominio in indirizzi IP. I nomi di dominio sono stringhe alfabetiche perciò più facili da ricordare, internet tuttavia è basato su indirizzi IP. Tutte le volte quindi che viene usato un nome di dominio il DNS deve tradurlo nel corrispondente indirizzo IP.

127.0.0.1	Venerdi.med.ira.cnr.it Venerdi localhost.localdomain localhost
192.168.1.1	Giovedì.med.ira.cnr.it Giovedi
192.168.1.3	Pcmed14.med.ira.cnr.it Pcmed14

Essendo le macchine collegate alla rete locale di Medicina si è deciso comunque di adottare misure per la sicurezza, mantenendo però una porta "privilegiata" per l'intercomunicazione tra le tre macchine che compongono il cluster. Il primo passo è stato quello di editare il file */etc/hosts.allow* su ciascuna macchina secondo quanto illustrato nella Tabella 5. In altre parole si permette l'accesso a tutte le macchine facenti parte della rete (classe C) 192.168.1.X; in più si è istruito il sistema operativo dei due nodi di calcolo per accettare l'accesso dall'indirizzo principale di Pcmed14 che è la macchina utilizzata per programmare il cluster.

Tabella 5 – Contenuto dei files /etc/hosts.allow per ciascuna macchina.

Pcmed14	ALL: 192.168.1.
Giovedì	ALL: 192.168.1. ALL 192.167.189.82
Venerdi	ALL: 192.168.1. ALL:192.167.189.82

E' stato inoltre configurato un firewall su ciascuna macchina (*iptables*). Con privilegi di root si è utilizzato il programma *redhat-securitylevel* per attivare il firewall e configurarlo in modo da lasciare aperta la porta SSH (Secure SHell). Il passo successivo è stato quello di impostarlo in modo da lasciare passare tutti i pacchetti TCP e UDP provenienti dalle macchine del cluster, il comando è: *ipatalbes –I INPUT –s 192.168.1.0/24 –p all –j ACCEPT*. Per salvare la configurazione in modo permanente solitamente si usa il comando *iptable-save > /etc/sysconfig/iptables*, in realtà si è costatato che questo sistema non salva con un formato corretto le vecchie impostazioni, rendendo di conseguenza impossibile avviare il firewall. Per questo motivo si è concatenato al vecchio file di configurazione quello nuovo, *iptable-save >> /etc/sysconfig/iptables*, poi si è rimpiazzato tutte le righe corrispondenti alle vecchie configurazioni con quelle del file originale, mantenendo solo la riga relativa alla configurazione nuova. Il firewall è stato poi riavviato: *service iptables restart.* Come esempio viene riportato il file di configurazione:

:INPUT ACCEPT [0:0]

:FORWARD ACCEPT [0:0]

:OUTPUT ACCEPT [0:0]

:RH-Firewall-1-INPUT - [0:0]

-A INPUT -j RH-Firewall-1-INPUT

-A FORWARD -j RH-Firewall-1-INPUT

-A RH-Firewall-1-INPUT -s 192.168.1.0/255.255.255.0 -j ACCEPT

-A RH-Firewall-1-INPUT -i lo -j ACCEPT

-A RH-Firewall-1-INPUT -p icmp --icmp-type any -j ACCEPT

-A RH-Firewall-1-INPUT -p 50 -j ACCEPT

-A RH-Firewall-1-INPUT -p 51 -j ACCEPT

-A RH-Firewall-1-INPUT -m state --state ESTABLISHED,RELATED -j ACCEPT

-A RH-Firewall-1-INPUT -m state --state NEW -m tcp -p tcp --dport 22 -j ACCEPT

-A RH-Firewall-1-INPUT -j REJECT --reject-with icmp-host-prohibited

COMMIT

E' stato aggiunto l'utente *prog* su ogni macchina con cui si procederà alla programmazione del cluster: *adduser –g 100 –u 500 –d /home/prog –m –s /bin/bash prog*. Da notare che lo user id (uid) e il group id (gid) sono stati esplicitamente indicati, in modo che l'utente risulti effettivamente lo stesso su tutti i nodi.

A questo punto si è reso necessario completare l'installazione dei nodi aggiungendo pacchetti come il server telnet e quello rsh (Remote SHell). Da notare che, per le impostazioni di sicurezza descritte prima, le macchine sono accessibili tramite questi due servizi solo da un'altra appartenente al cluster. I passi per l'installazione dei servizi sono stati:

1. Inserito CD3 di Fedora.
2. *mount /dev/ cdrom*
3. *cd /mnt/cdrom/Fedora/RPMS*
4. *rpm –ivh telenet-server-0.17-26.2.i386.rpm*
5. *rpm –ivh rsh-server.0.17-19.i385.rpm*

L'avvio dei servizi ha imposto nei due casi la modifica dei file */etc/xinitd.d/telnet, /etc/xnitd.d/rsh /etc/xnitd.d/rlogin* e */etc/xinitd.d/rexec* in modo da cambiare la dicitura "*disable=yes*" in "*disable=no*" seguita dal comando *service xinitd restart*. Per quel che riguarda rsh la configurazione ha richiesto qualche passo aggiuntivo dovuto alla necessità per l'utente prog di lanciare comandi remotamente a partire dal nodo master:

1. Dentro la home di prog è stato aggiunto il file *.rhosts* col seguente contenuto (per Pcmed14):

   127.0.0.1

   192.168.1.1

   192.168.1.2

   Questo file serve perché un utente possa connettersi a ciascun nodo senza dover fornire una password.

2. Impostati i privilegi giusti al file appena creato: *chmod 600 .rhosts*
3. Aggiunto al file *etc/hosts.equiv* l'elenco delle macchine che sono "fidate" per il servizio RSHD, ovvero aggiunto l'elenco degli indirizzi IP dei nodi del cluster.
4. Modificato il file *etc/securetty* che è la lista dei *tty* dai quali l'utente root è abilitato a collegarsi alla macchina. Alla fine del file sono state aggiunte le righe *rlogin, rsh, rexec*.
5. La directory */etc/pam.d/* contiene le configurazioni che influenzano il login dei vari servizi; i file *rsh, rlogin, rexec* all'interno di questa cartella sono stati modificati in modo da avere le linee con "rhosts" al primo posto, quelle con "securetty" al secondo e cambiando le parole "required" con "sufficient". Un tipico file è:

   #%PAM-1.0

   # For root login to succeed here with pam_securetty, "rsh" must be

   # listed in /etc/securetty.

   auth sufficient pam_rhosts_auth.so

   auth sufficient pam_securetty.so

   auth sufficient pam_nologin.so

   auth sufficient pam_env.so

account sufficient pam_stack.so service=system-auth

session sufficient pam_stack.so service=system-auth

La correttezza della configurazione è stata provata lanciando un comando remoto, ad esempio: *rsh -n host echo $SHELL*.

2.3 Implementazione LAM delle specifiche MPI.

MPI è l'acronimo di Message Passing Interface, ovvero la definizione di un insieme di funzioni API che consentono al programmatore di scrivere programmi ad elevate performance, che passano messaggi tra processi seriali in modo da ottenere un lavoro eseguito in forma parallela. Lo standard MPI è stato ideato per supportare la portabilità e l'indipendenza dalla piattaforma, per questo può essere impiegato sulle grandi macchine parallele che sulla singola workstation che sui PC che compongono un cluster. Per il lavoro in esame si è utilizzato questo paradigma di programmazione arrivato alla versione 2.0, in particolare si è sfruttata l'implementazione LAM versione 7.0.4, resa disponibile dall'Università dell'Indiana sotto licenza GNU.

2.3.1 Note sull'implementazione LAM.

L'implementazione di LAM/MPI è basata sul System Service Interface (SSI) che influenza come tutti i processi MPI sono eseguiti. SSI provvede dei componenti su cui si basa l'ambiente run-time di LAM (RTE) e lo strato di comunicazione MPI. Ciascun componente o modulo si differenzia per tipo e può essere selezionato per fornire servizi a run-time. Ci sono attualmente quattro tipi di componenti:

- Boot: usato per avviare l'RTE.
- Coll: utilizzato per le comunicazioni collettive tra i processi MPI.
- CR: implementa la funzionalità checkpoint/restart, usato sia dai comandi LAM sia dai processi MPI, serve per riprendere le esecuzioni a partire dai punti di controllo. Per funzionare questo tipo di modulo necessita che tutti gli altri (rpi e coll) siano in grado di gestire il checkpoint.
- RPI: utilizzato per le comunicazioni punto a punto tra i processi MPI.

Uno dei principi fondamentali dell'SSI è di consentire il passaggio di parametri all'SSI stesso a run-time. Questo consente sia di selezionare il modulo da usare per ciascuna tipologia di componente, sia di sintonizzare le performance impostando i parametri da passare ai moduli stessi. La scelta del modulo, se non esplicitata dall'utente, avviene tramite un meccanismo basato sulle priorità assegnate a ciascun modulo. Come scegliere i moduli ed esplicitare i parametri riferirsi a [2] nella trattazione dei comandi *lamboot* e *mpirun*.

I moduli disponibili per il boot del sistema sono diversi, quello effettivamente usato è uno solo ovvero l'*rsh/ssh* che ha come unico requisito quello che l'utente che esegue il boot sia abilitato ad eseguire comandi remoti senza dover fornire una password. Per le altre tipologie di componenti i moduli disponibili sono più significativi e vengono descritti nella Tabella 6, senza però soffermarsi sui parametri di configurazione per i quali si rimanda a [2].

Tabella 6 – Elenco dei moduli MPI disponibili.

RPI(Request Progression Interface)	
Crtcp	Studiato per essere utilizzato in congiunzione con un modulo CR. Si basa

	essenzialmente sulla comunicazione via socket TCP, ma proprio per la sua caratteristica di supportare i punti di controllo presenta maggiore overhead rispetto al modulo tcp.
Gm	Modulo usato in caso il cluster sia collegato attraverso reti Myrinet[7].
Lamd	La comunicazione interprocesso avviene sfruttando un demone, questo permette di passare i messaggi in maniera completamente asincrona (infatti la trasmissione del messaggio prosegue anche quando il programma utente sta eseguendo altre operazioni). Tutto ciò viene pagato con una maggiore latenza e di una minor larghezza di banda, per questo motivo è utilizzabile solo in quei rari casi in cui il tempo di calcolo supera quello di trasmissione e comunicazione.
Sysv	E' la combinazione di due meccanismi, esso utilizza la tecnica della memoria condivisa per la comunicazione tra processi all'interno dello stesso nodo e il TCP per quella inter nodo. Per la sincronizzazione dell'area di memoria condivisa vengono utilizzati semafori System V[8].
Usysv	Simile al precedente con la differenza che per la sincronizzazione vengono usati gli spin locks[9]. La natura di questi meccanismi fa si che tutto il sistema abbia prestazioni peggiori nel caso in cui ci siano più processi che processori. Gli spin locks infatti non sospendono il processo nell'attesa della risorsa, questo dispensa il sistema operativo dal fare switching tra processi e in più consente al processo stesso di "reagire" immediatamente quando la risorsa diventa disponibile.
Tcp	Basato interamente sull'implementazione dei socket TCP.
Coll	
Lam_basic	Questo modulo può essere usato in qualsiasi ambiente. Non sono fatti tentativi per determinare la località dei processi e si assume che la latenza tra i vari processi sia identica; per questo motivo tutte le comunicazioni sono eseguite secondo l'ordine predefinito. Il modulo è poco efficiente e non viene solitamente usato se altri moduli coll sono disponibili.
Smp	Il modulo determina la località dei processi prima di eseguire le funzioni collettive. Si appoggia ancora ai moduli punto a punto.
CR	
BLCR	Il modulo è prodotto dai Berkley Lab. Quando selezionato controlla che tutti gli altri moduli SSI supportino il checkpoint/restart.

Per compilare i programmi MPI è necessario utilizzare esattamente lo stesso compilatore usato per produrre i binari della versione LAM, inoltre è necessario indicare al compilatore dove trovare le librerie dinamiche e una notevole mole di parametri. Tutta questa complessità viene mascherata da dei "wrapper compilers" forniti assieme alla versione LAM. Questi compilatori (*mpicc*, *mpiCC*, *mpif77*) non fanno altro che invocare il compilatore giusto (C, C++, F77) passandogli i giusti parametri assieme a quelli eventualmente specificati dall'utente.

[7] Tecnologia di intercomunicazione a bassa latenza e basso overhead imposto alle macchine host rispetto alle classiche reti Ethernet.
[8] System V tipica implementazione del meccanismo dei semafori nei kernel UNIX.
[9] Spin locks o Mutex sono le più semplici e comuni primitive di sincronizzazione fornite dal Kernel. Sono solitamente utilizzati nella scrittura dei drivers.

2.3.2 Installazione di LAM/MPI.

Come prima cosa è stata creata nella home dell'utente *prog* di ciascuna macchina una cartella in cui salvare i file necessari alla programmazione: *mkdir ~ /ClusterFFT*. L'implementazione LAM, infatti, si aspetta di trovare su ciascun nodo l'eseguibile che implementa il task lanciato sul cluster. La cartella appena creata è stata poi aggiunta al PATH nel file *.bash_profile*.

Al fine di tenere sincronizzate le cartelle sui vari nodi, si è deciso di mantenere i file solo nella cartella */home/prog/ClusterFFT* di Pcmed14 che è il nodo master e di esportarla tramite server NFS. La configurazione del server NFS su Pcmed14 è stata fatta tramite il tool *NFSServerConfigurationTool* che corrisponde al pacchetto *redhat-config-nfs.rpm* della distribuzione Fedora; il software necessario al funzionamento del servizio NFS era già stato aggiunto al momento dell'istallazione del sistema operativo. Col tool prima menzionato si è esportata la cartella */home/prog/ClusterFFT* con privilegi di lettura e scrittura per l'utente *ID=500* e *GID=100* da tutte le macchine con indirizzo del tipo 192.168.1.0/24. Sui rimanenti due nodi la cartella è stata importata aggiungendo al file */etc/fstab* la riga:

pcmed14:/home/prog/ClusterFFT /home/prog/ClusterFFT nfs rw,retry=10,timeo=10 0 0

In questo modo, tutte le modifiche apportate dall'utente prog ai file della cartella *ClusterFFT* saranno immediatamente visibili anche sulle altre macchine.

Per l'installazione vera e propria di LAM/MPI si è scelto il pacchetto *lam-7.04-1.i586.rpm*, scaricabile al sito [3], contenente la versione precompilata. Con questa scelta non c'è possibilità di agire sulla configurazione (specialmente dei moduli SSI) compensata, però dalla facilità con cui il sistema va in funzione. Il comando *laminfo* restituisce informazioni sulla configurazione dell'ambiente:

LAM/MPI: 7.0.4

Prefix: /usr

Architecture: i686-pc-linux-gnu

Configured by: jsquyres

Configured on: Tue Jan 13 20:49:26 EST 2004

Configure host: traal.osl.iu.edu

C bindings: yes

C++ bindings: yes

Fortran bindings: yes

C profiling: yes

C++ profiling: yes

Fortran profiling: yes

ROMIO support: yes

IMPI support: no

Debug support: no

Purify clean: no

SSI boot: globus (Module v0.5)

SSI boot: rsh (Module v1.0)

SSI coll: lam_basic (Module v7.0)

SSI coll: smp (Module v1.0)

SSI rpi: crtcp (Module v1.0.1)

SSI rpi: lamd (Module v7.0)

SSI rpi: sysv (Module v7.0)

SSI rpi: tcp (Module v7.0)

SSI rpi: usysv (Module v7.0)

La partenza dell'ambiente LAM (LAM UNIVERSE) avviene tramite il comando *lamboot –v –d –ssi boot rsh clusterhosts* lanciato dal nodo master. Le opzioni –v e -d aggiungono alcune informazioni utili per il debug nel caso qualcosa non funzioni; il flag –ssi non fa altro che confermare che per il boot va usato il modulo *rsh/ssh*. Il file *clusterhosts*, che contiene alcune informazioni necessarie per eseguire il boot, è riportato sotto:

pcmed14 cpu=1

giovedi cpu=1

venerdi cpu=1

In questo caso è riportato l'elenco delle macchine che compongono il cluster e il numero di processori montati su ciascuna macchina ovvero quanti processi MPI possono essere lanciati su quella macchina; in realtà è possibile specificare altri parametri quali l'utente autorizzato ad eseguire programmi, i nodi non schedulabili ecc. Nel caso, come questo, si utilizzino i nomi degli host anziché gli indirizzi IP, è fondamentale che il nodo padre riconosca tutti gli altri nodi (compreso se stesso) con gli indirizzi della sottorete virtuale (vedi file */etc/hosts*). Nel caso di problemi il comando *recon* effettua un controllo veloce se l'ambiente è configurato opportunamente per eseguire il boot. Una volta che il boot ha avuto successo il comando *lamnodes* ritorna l'elenco dei nodi che compongono il cluster:

n0 pcmed14.med.ira.cnr.it:1:origin,this_node

n1 giovedi.med.ira.cnr.it:1:

n2 venerdi.med.ira.cnr.it:1:

Mentre il comando *tping –n –c 10* verifica per dieci volte che tutti i nodi siano vivi:

1 byte from 2 remote nodes and 1 local node: 0.002 secs

1 byte from 2 remote nodes and 1 local node: 0.001 secs

1 byte from 2 remote nodes and 1 local node: 0.002 secs

1 byte from 2 remote nodes and 1 local node: 0.001 secs

1 byte from 2 remote nodes and 1 local node: 0.001 secs

1 byte from 2 remote nodes and 1 local node: 0.001 secs

1 byte from 2 remote nodes and 1 local node: 0.001 secs

1 byte from 2 remote nodes and 1 local node: 0.001 secs

1 byte from 2 remote nodes and 1 local node: 0.001 secs

1 byte from 2 remote nodes and 1 local node: 0.001 secs

10 messages, 10 bytes (0.010K), 0.015 secs (1.343K/sec)

roundtrip min/avg/max: 0.001/0.001/0.002

Capitolo 3: Programma di test.

3.1 Introduzione.

Per testare il Cluster MPI allestito si è deciso di scrivere un programma per riuscire a valutare le performance ottenibili con una delle applicazioni più diffuse in ambito radioastronomico: la trasformata di Fourier. L'applicazione non fa altro che "emulare" uno spettrometro digitale calcolando e mediando spettri d'ampiezza. Il programma è di tipo Master/Slave, in altre parole è un task in cui c'è un nodo che distribuisce agli altri i compiti, mentre i nodi di calcolo svolgono il lavoro vero e proprio. Nello specifico il nodo master (Pcmed14) colleziona i dati da processare, li invia ai nodi di calcolo (Giovedì, Venerdi) e attende che questi abbiano finito prima di ricevere gli spettri di potenza e calcolare le performance del sistema. Non essendo dotato di scheda d'acquisizione, Pcmed14 non campiona i dati nel dominio del tempo ma semplicemente li genera. Questa non è ovviamente una limitazione in quanto questo programma è orientato alla valutazione delle capacità di calcolo di un cluster non all'utilizzo pratico.

3.2 Manuale d'uso.

Il programma, dopo aver impostato l'ambiente LAM, funziona a riga di comando. Per lanciarlo è stata preparata una procedura batch *goFFT*:

mpirun C clusterFFT "$1" "$2" "$3" "$4" "$5" "$6"

Questa procedura esegue il programma *clusterFFT* su ogni processore schedulabile passandogli l'elenco dei parametri passati alla procedura stessa. L'elenco dei parametri riconosciuti si ottiene col comando *goFFT –h*:

rank 0 goFFT [-bBins] [-aAccumulations] [-fFrequency] [-sSNRatio] [-gFrames] [-w] [-h]

rank 0 -bBins Number of bins of the FFT (32-65536), default 2048

rank 0 -aAccumulations Number of accumulations (1-10000), default 50

rank 0 -fFrequency Frequency of generated fake signal, default 512.0

rank 0 -sSNRatio Signal to noise ratio of generated fake signal, default 0.2

rank 0 -gFrames Number of frames sent to a single nodes at the same time, default 1

rank 0 -wFileName Write computed spectrum into a file

rank 0 -h Shows this help

rank 0 Execution halted!

rank 2 Execution halted!

rank 1 Execution halted!

Il comando *goFFT –b1024 –a500 –f128 –s0.1-g2 –wOut.txt* ad esempio misura le performance del cluster calcolando e mediando lo spettro d'ampiezza di una FFT da 1024 canali. Il segnale digitale fittizio che viene generato è tale da avere un rapporto segnale rumore di 0.1 e da essere rilevato, nel dominio delle frequenze, in corrispondenza del centoventottesimo canale. Specificare il flag .*-w* significa che si vuole che i valori (nell'esempio 1024), che rappresentano lo spettro calcolato, siano scritti in formato testuale, nel file specificato. Quest'ultima scelta risulta utile per verificare se il

software è corretto, in altre parole se produce risultati simili a quelli che produrrebbe una spettrometro tradizionale.

Un possibile output del programma è riportato di seguito:

rank 0 number of bins : 512

rank 0 number of accumulations : 1000

rank 0 signal frequency : 512.000000

rank 0 signal to noise ratio : 0.200000

rank 0 number of frames : 1

rank 0 number of tasks : 3

rank 0 Computations begin.....

rank 0 Spectrum completed.....

rank 0 ...now fetching data

rank 0 Total execution time (us): 1070624

rank 0 Done blocks: 1000

rank 0 Lost blocks: 0

rank 0 Mean computation time (us): 242378

rank 0 Mean idle time (us): 815494

rank 0 Node 1 stats:

rank 0 Computation time (us): 126007

rank 0 Idle time (us): 938187

rank 0 Done blocks: 328

rank 0 Node 2 stats:

rank 0 Computation time (us): 358750

rank 0 Idle time (us): 692801

rank 0 Done blocks: 672

rank 0 Execution halted!

rank 1 Execution halted!

rank 2 Execution halted!

Le prime righe riassumono i valori dei parametri con cui il programma è stato eseguito; nel seguito vengono riportati tempo d'esecuzione totale, tempo di idle (impiegato per la maggior parte al trasferimento dei dati via rete) e tempo di calcolo vero e proprio così come stimati dal nodo master.

Successivamente sono riportati gli stessi valori riferiti questa volta alle misure effettuate dai nodi di calcolo.

3.3 Note sull'implementazione.

Come già accennato, il programma è strutturato secondo il paradigma master/slave in cui il nodo master genera dei dati fittizi (secondo una funzione parametrizzabile) e li distribuisce ai nodi di calcolo; appena uno di essi ha completato, il master spedisce subito un altro frame. I nodi slave, una volta ricevuti i dati ne calcolano l'FFT e quindi lo spettro d'ampiezza (modulo delle coppie complesse prodotte dalla routine di FFT); gli spettri d'ampiezza vengono poi mediati con quelli calcolati in precedenza. Quando l'intero processo è completato (sono state fatte tutte le accumulazioni) i nodi di calcolo inviano al master il loro spettro risultante, il numero di blocchi che hanno processato e le stime sui tempi di calcolo. Il nodo master "pesa" gli spettri ricevuti e produce il definitivo, cioè quello scritto a file nel caso sia specificata l'opzione *–w*, e stampa i tempi richiesti per l'intero processo. Il nodo master è quello con rank 0, solitamente è il nodo indicato per primo nel file di boot (vedi *lamboot*).

Nel seguito si da una breve descrizione delle parti ritenute interessanti del programma di test; i dettagli possono essere trovati nell'Appendice A dove sono riportati i sorgenti.

3.3.1 Inizializzazione.

Prima di poter chiamare qualsiasi routine MPI un programma utente deve inizializzare l'ambiente run time chiamando l'API *MPI_Init*. Tra le varie finalità di questa routine c'è la registrazione del processo al comunicatore *MPI_COMM_WORLD*. Un comunicatore può essere descritto come un gruppo di processi che vogliono comunicare tra loro; esso offre meccanismi di sincronizzazione e interscambio. Con la chiamata *MPI_Init* il processo viene inserito nel comunicatore di default, ma è anche possibile crearne uno nuovo e privato per evitare di interferire con altri processi coi quali non si vuole comunicare. Prima di chiudere il processo occorre usare la routine *MPI_Finalize* per rilasciare le risorse allocate da *MPI_Init*.

3.3.2 Gestione degli errori MPI.

Ogni API messa a disposizione dall'ambiente MPI fa riferimento ad un comunicatore. Associata a ciascun comunicatore c'è una routine di gestione degli errori; quella di default è la *MPI_ERRORS_ARE_FATAL* che non fa altro che, in caso d'errore, terminare tutti i task di quel comunicatore. Soprattutto in fase di debug è invece utile tenere traccia degli eventuali errori, perciò per quest'applicativo si è creato un nuovo comunicatore, duplicandolo da quello di default (*MPI_Comm_dup*) e gli si è assegnata una nuova routine di gestione degli errori che, rispetto a quella di default, ricava dal codice d'errore una stringa esplicativa (*MPI_Error_string*) e la stampa sullo standard error (stderr). La routine deve prima essere allocata (*MPI_Errhandler_create*), poi assegnata al comunicatore (*MPI_Errhandler_set*). Alla chiusura del processo la funzione di gestione degli errori deve essere rilasciata con una chiamata a *MPI_Errhandler_free*.

3.3.3 Comunicazione e sincronizzazione.

Il paradigma MPI mette a disposizione dei processi diverse routine di comunicazione, esse coprono tutte le combinazioni tra chiamate bloccanti e non bloccanti, messaggi punto a punto ed uno a molti (broadcast).

Le chiamate non bloccanti sono routine che restituiscono il controllo al blocco di programma che le ha invocate senza che necessariamente l'operazione di trasferimento del messaggio sia completata. Questo tipo di semantica implica sia una maggior difficoltà nella sincronizzazione dei processi, sia che i buffer utilizzati per immagazzinare il messaggio non possano essere riusati in sicurezza; di contro offrono performance migliori specialmente in quelle applicazioni (rare) in cui la parte di calcolo è preponderante rispetto all'attività d'interscambio.

Il nodo master manda a tutti i nodi di calcolo un messaggio d'inizializzazione, con cui sincronizza tutti i processi, informa gli slave sul numero di canali con cui dovranno calcolare l'FFT, sul numero d'accumulazioni che si prevedono per quel nodo, su quanti frame dati gli verranno trasferiti con un unico messaggio dati. A questo punto parte la fase di calcolo vera e propria; il master genera il primo frame di dati e lo invia al primo nodo, questo nodo viene marcato "occupato". Un secondo frame dati viene inoltrato al primo nodo non "occupato" nella lista e così via finché tutti i nodi sono "occupati". Quando tutti gli slave sono impegnati nella fase di calcolo, il master rimane in attesa del primo che si dichiara non "occupato" e gli invia altri dati finché tutti i frame richiesti non sono stati processati. Come ultimo passo raccoglie i risultati che ciascun nodo di calcolo gli invia. Il nodo slave esegue un algoritmo più semplice: rimane in attesa di un messaggio da parte del master, se questo è un messaggio d'inizializzazione si predispone al calcolo della FFT, se contiene dati lo elabora, se è un messaggio di stop risponde con un altro messaggio con cui restituisce al master i suoi risultati. Ogni messaggio inviato è marcato con un TAG che definisce il tipo di messaggio come illustrato in Tabella 7.

Tabella 7 – Tipi di messaggi e relativi TAG scambiati tra nodo master e nodi slave.

TAG	CODICE	MITTENTE	USO
TERMINATE	100	Master	Forza i nodi slave a terminare l'esecuzione
STOP	200	Master	Informa il nodo slave che non ci sono più dati da processare e che il master attende i risultati del calcolo.
INIT	300	Master	Usato per sincronizzare tutti i processi prima dell'inizio del calcolo.
DATA	400	Master	Il messaggio contiene i dati da processare.
SPECTRA	500	Slave	Usato per ritornare al master lo spettro risultato dell'elaborazione dei dati svolta su quel nodo
INFO	600	Slave	Usato per informare il master sui tempi di calcolo, sul numero di frame elaborati ecc.

Per com'è strutturata l'applicazione che si sta descrivendo la scelta è caduta logicamente sull'utilizzo di tutte chiamate punto a punto (tra master e nodo slave e viceversa), bloccanti per quel che riguarda i nodi di calcolo, non bloccanti per il master. Un nodo slave attende indefinitivamente che il master gli dica cosa fare, cioè esegue una ricezione bloccante (*MPI_Recv)*. Il master invece non può attendere che un nodo slave abbia ricevuto i dati, ma deve immediatamente preparare un altro buffer per spedirlo al nodo successivo (*MPI_Isend)*.

La sincronizzazione per le chiamate non bloccanti avviene tramite la routine *MPI_Test*, anche questa non bloccante, che ritorna un flag che indica se il destinatario ha, di fatto, ricevuto il messaggio. Con questo sistema il master non fa altro che eseguire il test sui vari nodi; se il messaggio precedente risulta ricevuto significa che il nodo in questione non è più "occupato" e può ricevere un altro messaggio. Questo meccanismo funziona anche se il nodo è, di fatto, ancora impegnato a fare calcoli: in questo caso sono i meccanismi della rete d'intercomunicazione che fanno da buffer per il messaggio.

L'ultima sincronizzazione importante va fatta prima della chiusura di tutti i processi. L'implementazione LAM invalida il comunicatore se uno dei processi membri si chiude; questo implica che anche gli altri processi escano con un errore. Se dopo aver ricevuto il messaggio di terminazione, un processo chiude prima degl'altri (ad esempio perché gira su una macchina più veloce) si può incorrere nel caso sopra descritto. Al fine di evitare quest'evenienza i processi vengono allineati (*MPI_Barrier*) prima delle istruzioni di chiusura.

3.4 Compilazione

Come già detto, tutti gli oggetti e i binari che costituiscono l'implementazione LAM sono prodotti col compilatore ed il linker nativi del sistema operativo. Anche gli applicativi vanno perciò compilati coi medesimi strumenti avendo però cura di indicare tutte le librerie e gli header LAM. Per di più se il cluster è composto di architetture non omogenee il lavoro si complica ulteriormente. Per questo motivo esistono dei wrapper compiler che implicitamente fanno tutto questo lavoro ed esplicitamente passano al compilatore nativo le opzioni che essi ricevono in input. Il compilatore che si è usato è il wrapper dello standard C *mpicc*, come si vede dal listato del contenuto del *makefile* riportato sotto:

```
CC = mpicc
LIBPATH=.
CFLAGS=
LIBES = -lm
#
OBJECTS = clusterFFT.o parsecommandline.o outmessage.o outerror.o slave.o master.o FFT.o
#
clusterFFT : $(OBJECTS)
        mpicc -o clusterFFT $(OBJECTS) $(LIBES)
#
clean:
        touch clusterFFT
        rm -f *.o
```

3.5 Debugging.

Il problema del debugging di applicazioni parallele non è di facile soluzione in quanto la distribuzione MPI che si è utilizzata mette a disposizione pochi tool oltre alla compatibilità con *TotalView*. *TotalView* è un debugger che supporta il debug di programmi in parallelo in quanto è in grado di "collegarsi" a più processi contemporaneamente. Sfortunatamente è un tool commerciale

per cui non a disposizione. Gli altri strumenti sono il comando *mpitask* che visualizza alcune informazioni sui processi in esecuzione sul cluster e il comando *mpimsg* che esamina i messaggi in uso sul cluster.

Il problema quindi, è di debuggare un programma parallelo con un debugger seriale (*gdb*). Come prima cosa è stato creato un file di script (*debug*) da fare eseguire a *mpirun* invece dell'eseguibile vero e proprio. Lo script determina quale nodo e come deve eseguire un determinato processo. Ad esempio per debuggare l'algoritmo del nodo master(rank 0) lo script risulta essere:

n0 gdb -x gdbcmds clusterFFT

n1,2 clusterFFT

La prima riga stabilisce che sul nodo di rango 0 sia eseguito il debugger, la seconda che sugli altri due nodi l'applicazione sia eseguita normalmente l'applicazione. Se il comando *mpirun debug* è eseguito da un terminale sul nodo 0 (Pcmed14) è possibile controllare esattamente il comportamento della parte di codice riservata al master mentre quella per gli slave gira normalmente sugli altri nodi. L'opzione *–x* passata al *gdb* lo istruisce affinché prima di lasciare il controllo all'utente esegua le istruzioni contenute nel file *gdbcmds*. Quest'opzione, ad esempio, può essere utile per fissare dei break-point o per passare dei parametri al programma:

break clusterFFT.c:22

break master.c:52

run -g4

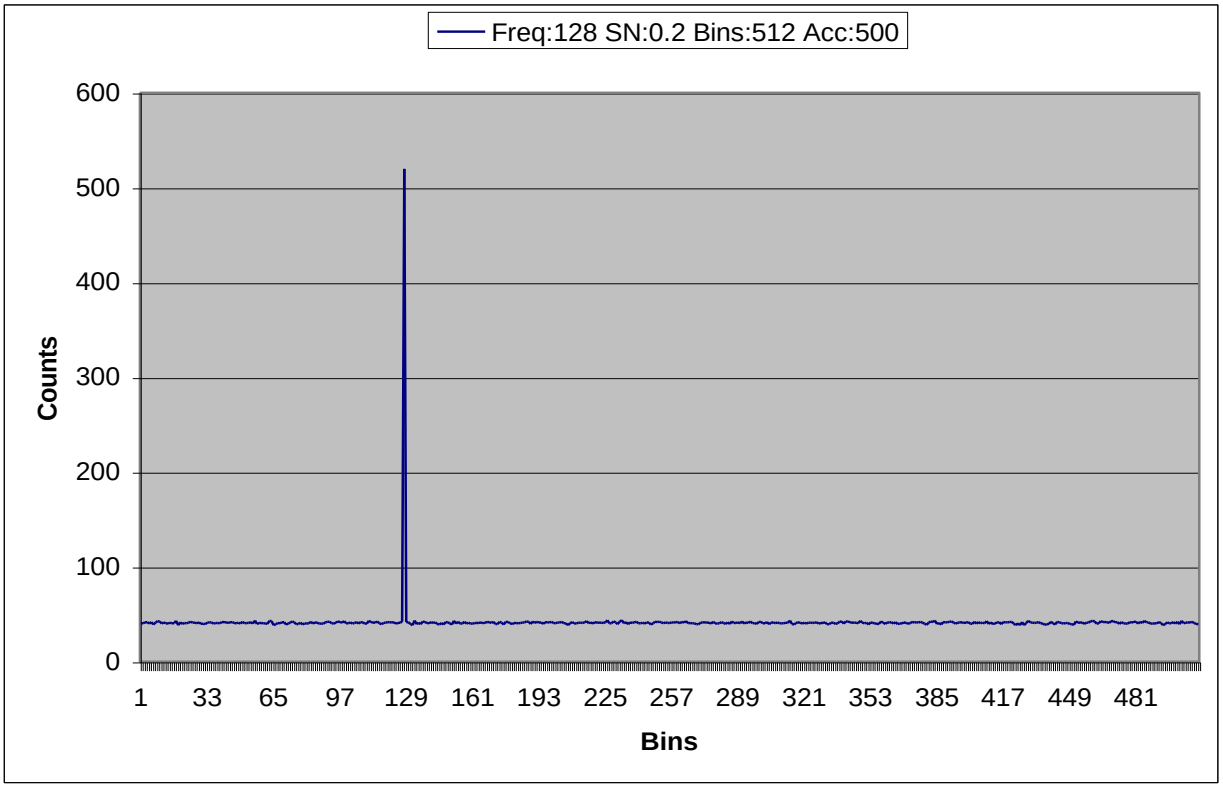
Con un file *gdbcmds* costruito come illustrato sopra si imposta un break-point alla riga 22 del modulo *clusterFFT.c* uno alla riga 52 del modulo *master.c* e si lancia il programma con l'opzione *g4* (vedi Capitolo 3.2).

Capitolo 4: Analisi dei risultati.

4.1 Verifica della correttezza.

La valutazione delle performance di un programma deve essere preceduta dalla verifica della sua correttezza al fine di evitare di stimare dei tempi su un problema che non è quello desiderato. Il programma di test offre la possibilità di impostare diversi parametri tra cui la frequenza del segnale sinusoidale auto generato, il suo rapporto segnale rumore, il numero di canali della trasformata di Fourier e le integrazioni. Variando questi parametri e sfruttando la possibilità di salvare su file lo spettro risultante è stato possibile verificarne il comportamento rispetto a quello di uno spettrometro digitale vero e proprio.

Figura 6 - Spettro d'ampiezza calcolato con il programma di test.



In Figura 6 ed in Figura 7 sono riportati due spettri ottenuti fissando la frequenza e impostando il numero di canali dell'FFT a 512 e 1024 rispettivamente. Come aspettato il segnale viene rilevato in corrispondenza del centoventottesimo canale in entrambi i casi; non solo, aumentando la risoluzione in frequenza il livello con cui viene rilevato aumenta a sua volta.

La Figura 8 mostra una situazione analoga a quella della Figura 6 con la differenza che la frequenza del segnale sinusoidale è stata impostata 128.5. In questo caso la riga non occupa più un solo canale ma cade a cavallo del centoventottesimo e del centoventinovesimo.

Figura 7 – Spettro d'ampiezza calcolato con il programma di test.

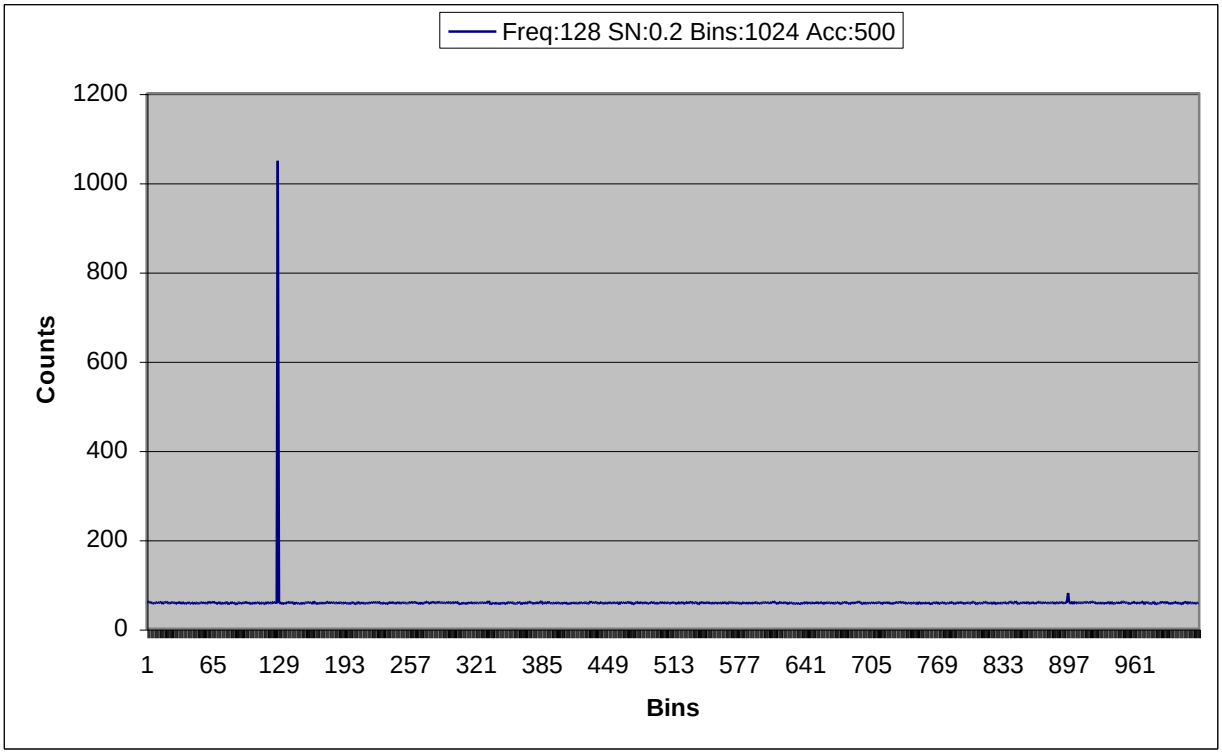


Figura 8 – Spettro d'ampiezza calcolato con il programma di test.

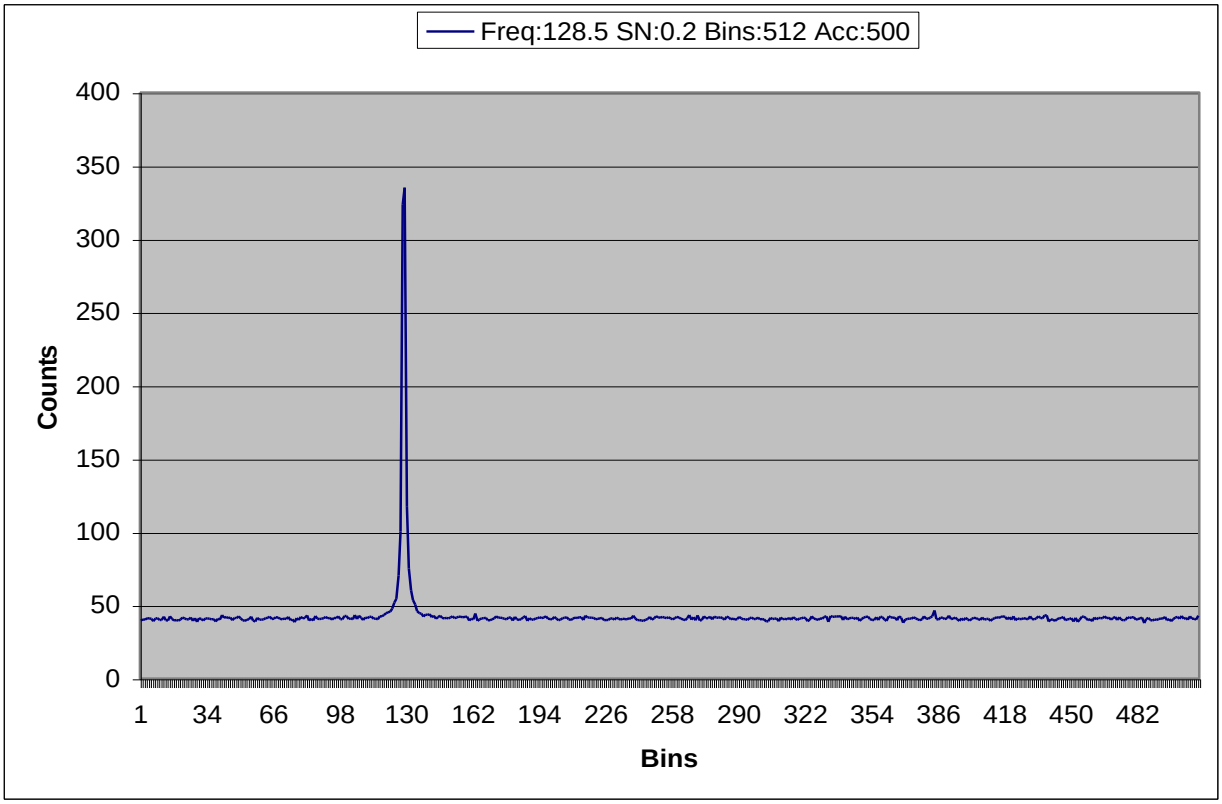


Figura 9 – Spettro d'ampiezza calcolato con il programma di test.

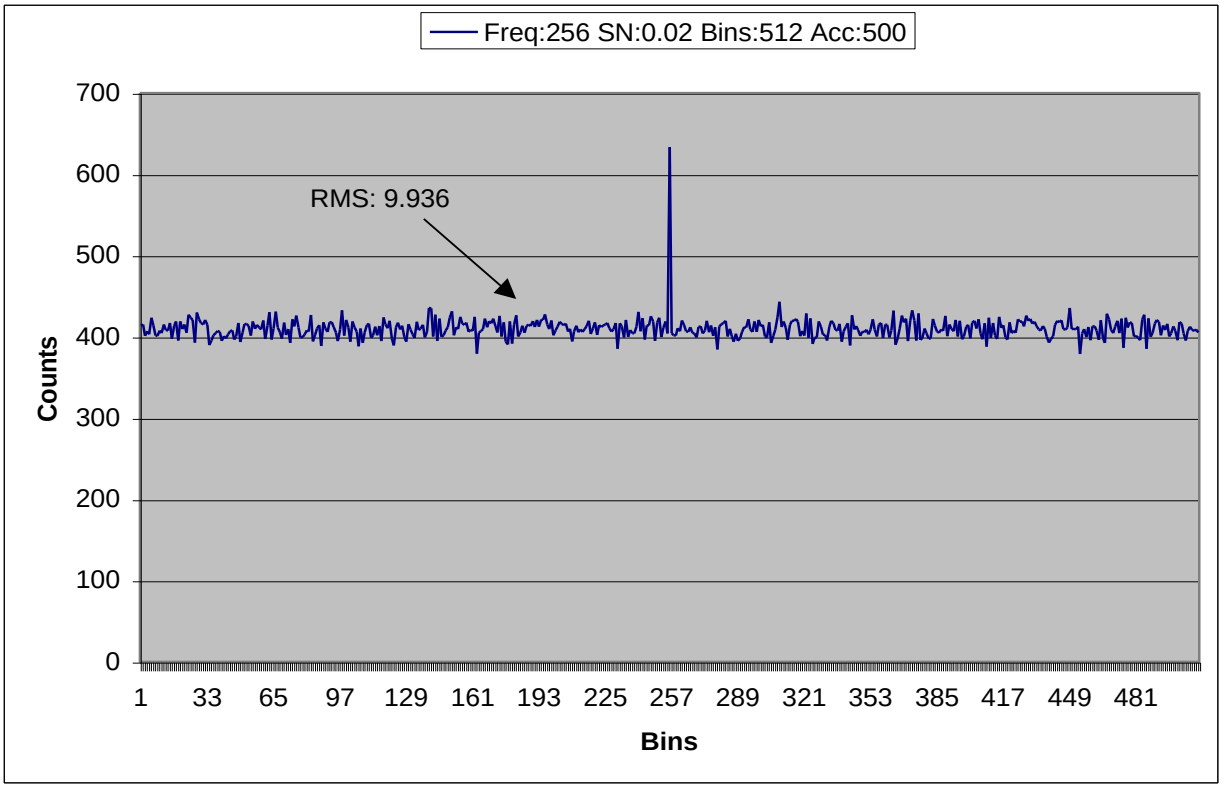


Figura 10 – Spettro d'ampiezza calcolato con il programma di test.

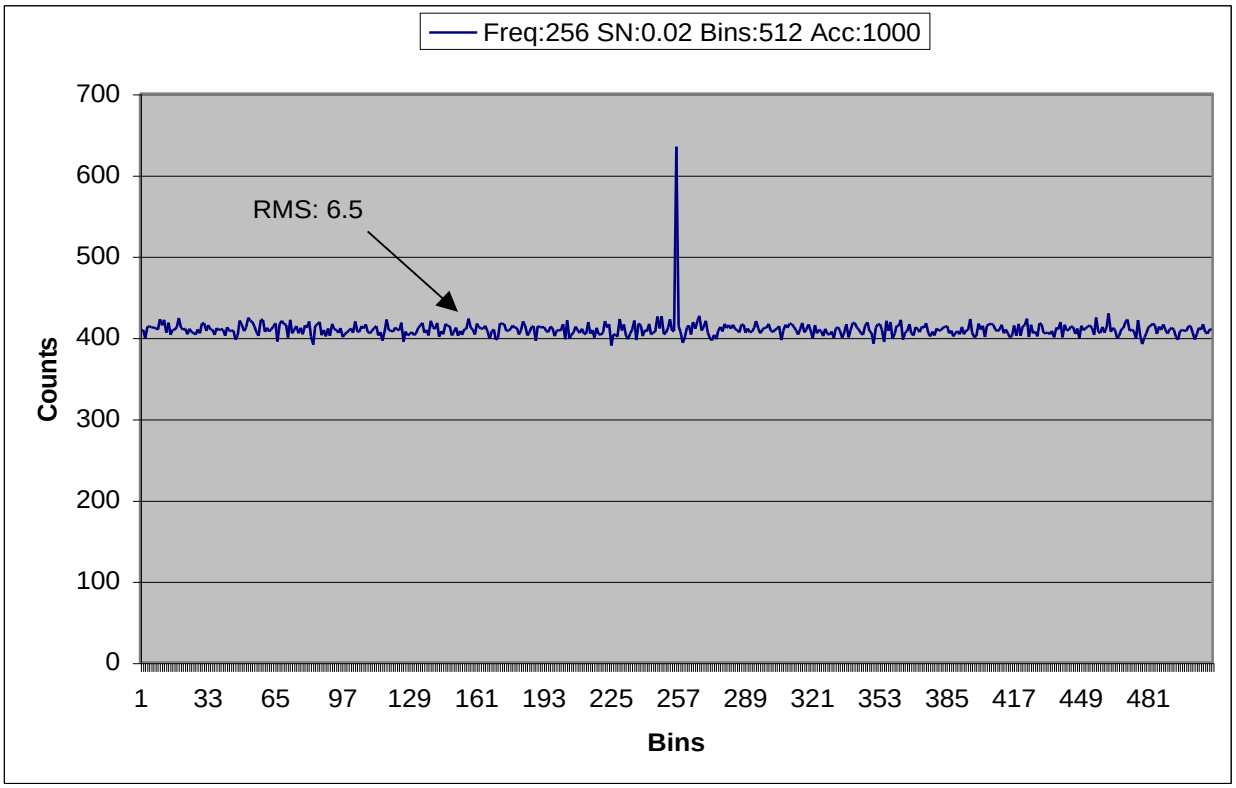
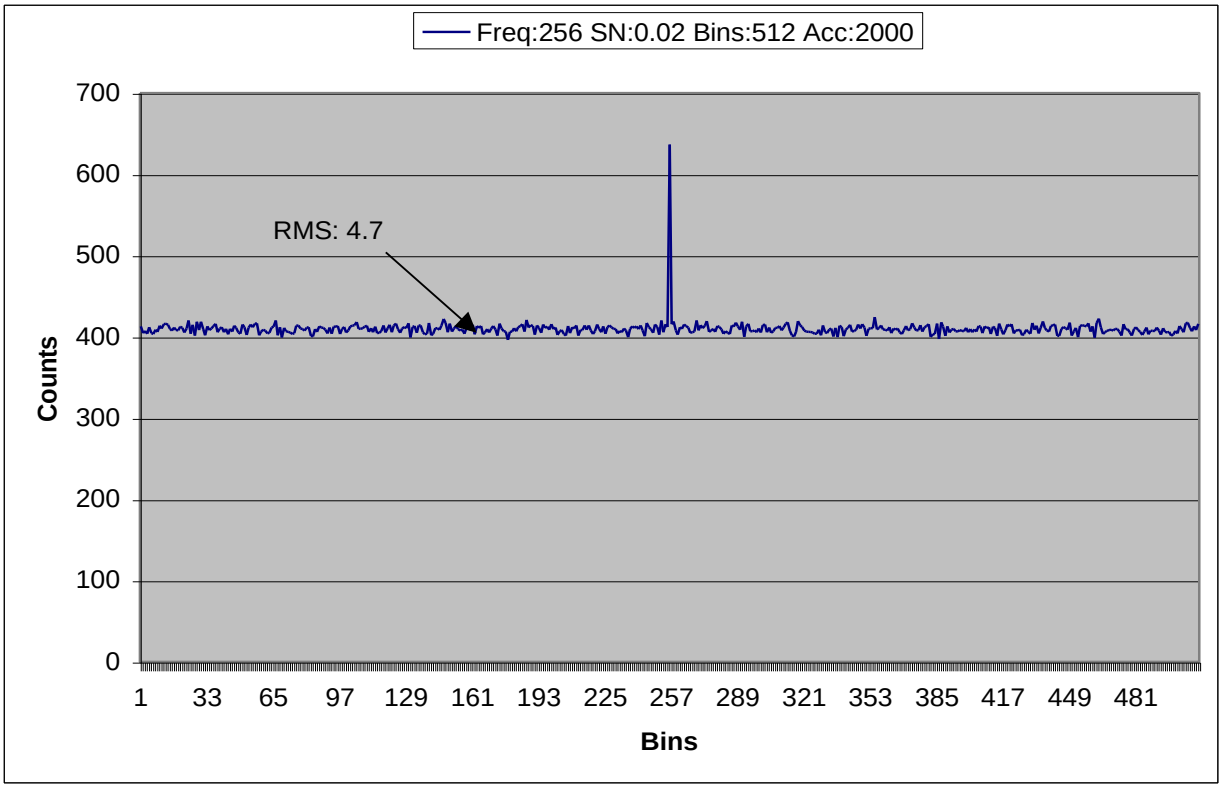


Figura 11 – Spettro d'ampiezza calcolato con il programma di test.



Le Figure 9, 10 ed 11 riportano gli spettri e l'RMS del rumore ottenuti tenendo fissi il numero di canali e il rapporto segnale rumore del segnale nel dominio del tempo ma variando il numero di accumulazioni, ovvero il tempo di integrazione. Come ci si attendeva, aumentando le accumulazioni diminuisce l'RMS proporzionalmente alla radice quadrata del tempo di integrazione. Se prendiamo il caso di Figura 9 (500 accumulazioni) e lo si confronta col caso di Figura 11 (2000 accumulazioni) in cui il tempo di integrazione è quattro volte tanto si verifica che, in questo caso, l'RMS finale è circa la metà del precedente.

4.2 Performance ottenute.

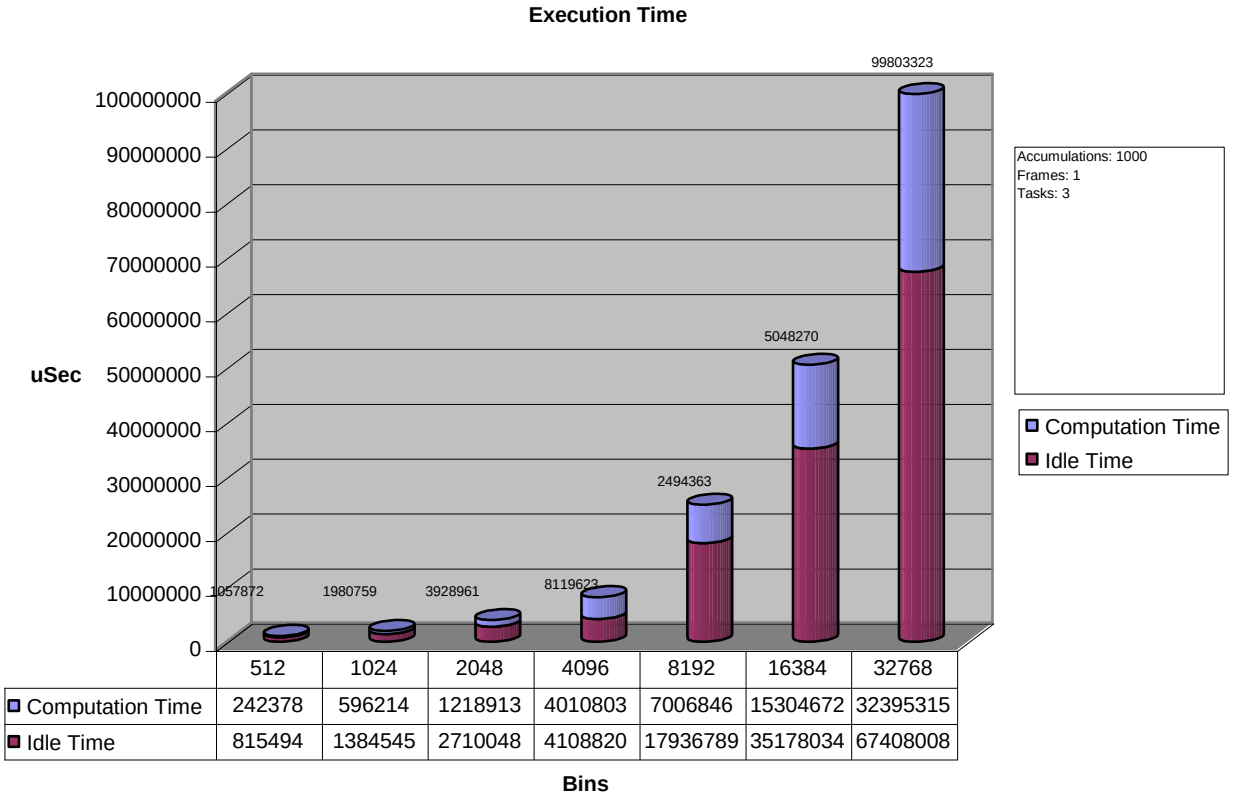
Dare una stima delle performance del cluster, la più vicina possibile alla realtà, non è ovviamente un problema di facile soluzione; questo per l'elevato numero di fattori che possono intervenire, sia a livello del singolo nodo, sia a livello dell'intero cluster: latenze di rete, carico delle CPU variabile e non completamente definibile, parametri di configurazione ecc. Per coprire la maggioranza della casistica sono state ripetuti diversi test, variando di volta in volta alcuni parametri di configurazione. I tempi riportati (tutti in microsecondi) sono frutto delle medie dei tempi ottenuti dalle diverse prove al fine di minimizzare i contributi di fattori non prevedibili.

4.2.1 Test 1: variazione del numero di canali.

Il test più immediato, ma probabilmente più indicativo che può essere fatto è quello di variare il numero di canali con cui si calcola la FFT. La Figura 12 mostra i risultati ottenuti mantenendo fissi il numero d'accumulazioni (1000) e la dimensione dei frame (1). Ogni istogramma riporta il tempo totale di esecuzione (in alto), di questo il tempo impiegato per il calcolo vero e proprio (parte

superiore) e quello impiegato per le altre operazioni (parte inferiore) tra cui la più importante è quella dedicata al trasferimento dei dati.

Figura 12 – Tempi di calcolo della FFT ottenuti variando il numero di canali.

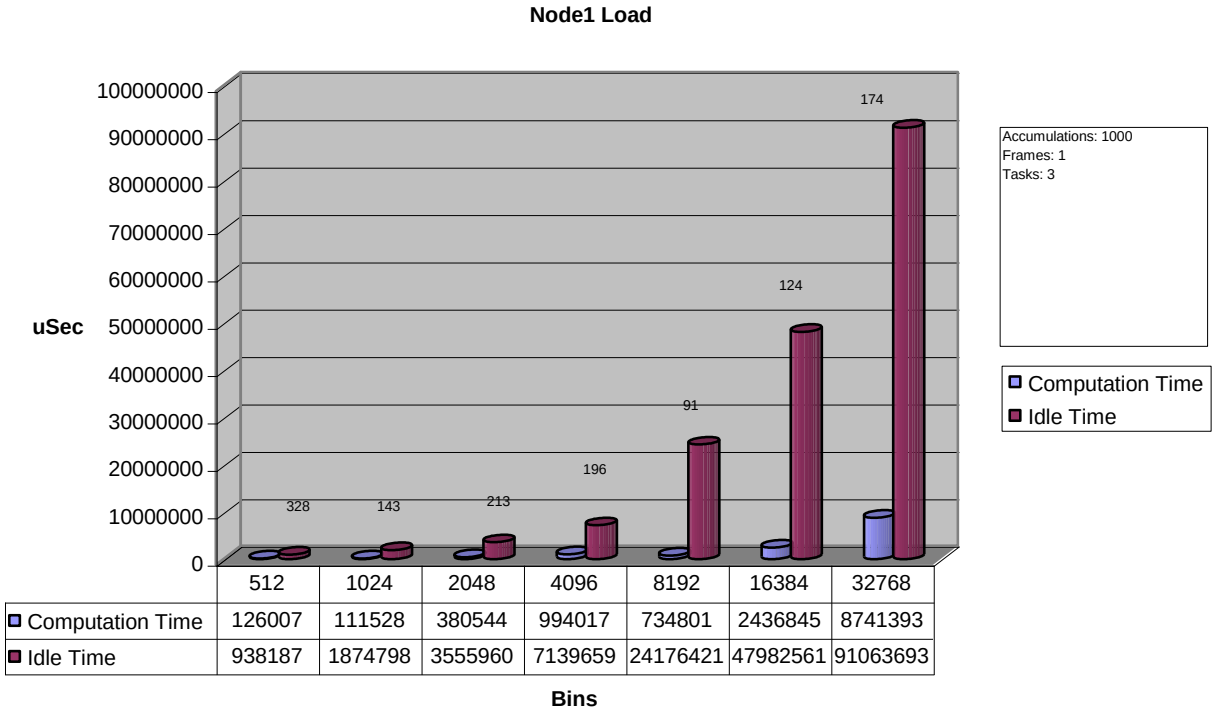


	512	1024	2048	4096	8192	16384	32768
Computation Time	242378	596214	1218913	4010803	7006846	15304672	32395315
Idle Time	815494	1384545	2710048	4108820	17936789	35178034	67408008

La proporzione tra queste due voci rimane quasi sempre inalterata, nel senso che il tempo di calcolo si attesta a circa il 30% del tempo totale a parte il caso dei 512 canali (circa 22%). Quest'ultimo rilievo può essere spiegato col fatto che il frame dati per una FFT da 512 canali è piuttosto piccolo, perciò il tempo del suo trasferimento dal nodo master al nodo di calcolo risente molto di più della latenza iniziale della rete rispetto agli altri casi. In prima approssimazione si può inoltre osservare che il tempo d'esecuzione raddoppia col raddoppiare delle dimensioni della FFT fino a 4096 canali; a questo punto si ha un salto e da lì si ha di nuovo lo stesso andamento quasi lineare. Questa discontinuità è probabilmente da ricercare all'interno dei nodi di calcolo, dove una FFT da 4096 ha dimensioni tali da poter ancora essere "trattata" nella cache del processore, oltre a questo limite la cache non è più sufficiente e le prestazioni hanno un degrado evidente.

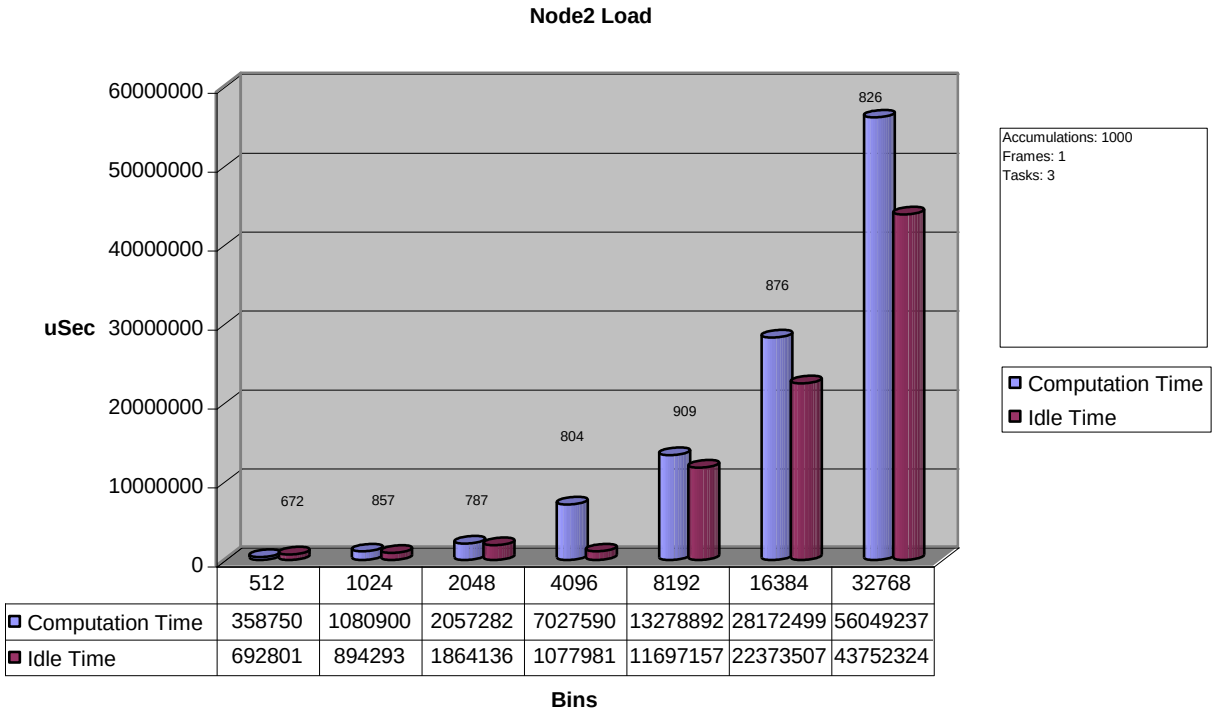
In generale le prestazioni misurate non sono buone; il motivo più evidente salta subito agli occhi confrontando Figura 13 con Figura 14. Le due immagini riportano il carico di lavoro dei due nodi slave suddiviso, anche in questo caso, in tempo di calcolo e tempo di trasferimento dati; il numero riportato sopra è il numero di blocchi processati dal nodo su un totale di 1000 accumulazioni richieste. Come si può facilmente verificare il cluster è fortemente sbilanciato in quanto il secondo nodo (Venerdì) svolge il grosso del lavoro mentre il primo non sfrutta le proprie potenzialità di calcolo e rimane per la maggior parte del tempo in attesa di completare le operazioni di trasferimento dati. La causa quasi certa di questo comportamento è da ricercare nei problemi in ricezione dell'interfaccia di rete di Giovedì così come descritto nel Capitolo 2.1 e raffigurato in Figura 5.

Figura 13 – Carico di lavoro di Giovedi, ottenuto in funzione del numero di canali della trasformata di Fourier.



	512	1024	2048	4096	8192	16384	32768
Computation Time	126007	111528	380544	994017	734801	2436845	8741393
Idle Time	938187	1874798	3555960	7139659	24176421	47982561	91063693

Figura 14 – Carico di lavoro di Venerdi, ottenuto in funzione del numero di canali della trasformata di Fourier.

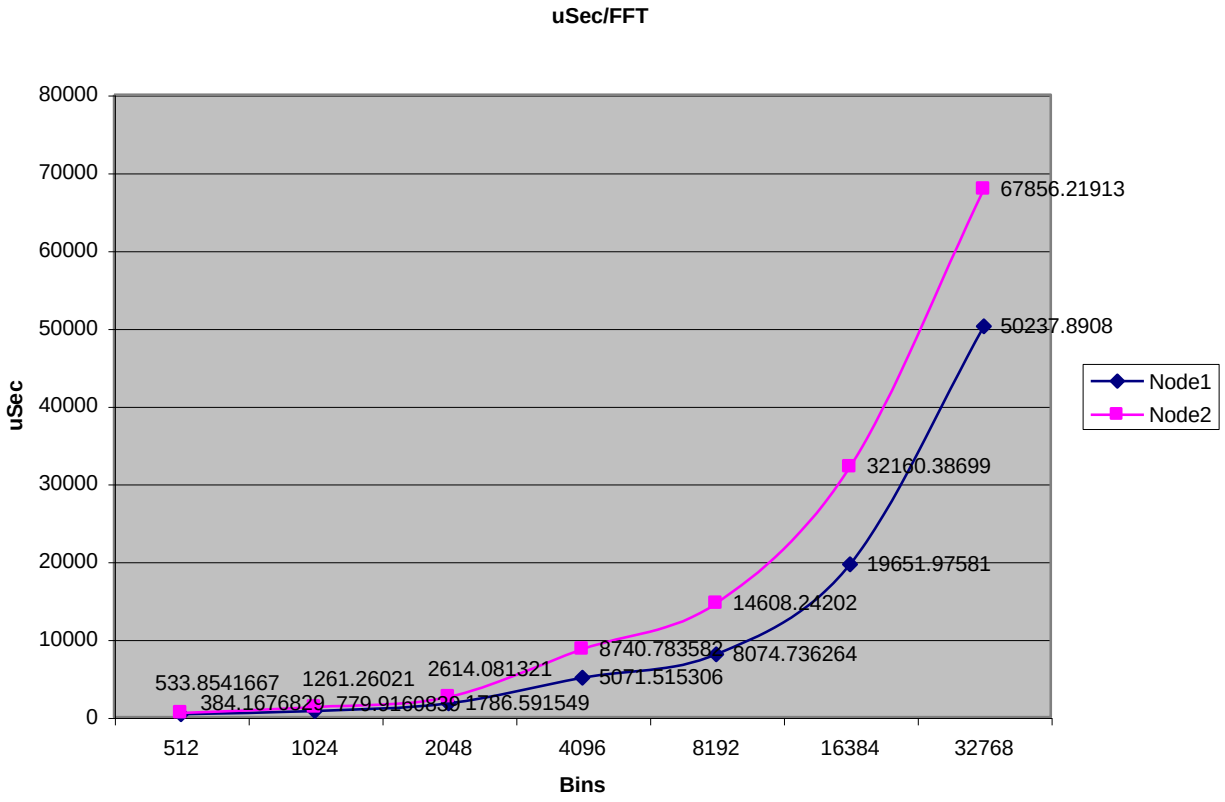


	512	1024	2048	4096	8192	16384	32768
Computation Time	358750	1080900	2057282	7027590	13278892	28172499	56049237
Idle Time	692801	894293	1864136	1077981	11697157	22373507	43752324

Un altro possibile motivo delle scarse prestazioni può essere individuato nel fatto che l'algoritmo di FFT non è stato ottimizzato; la Figura 15 mostra le stime sui tempi (microsecondi a FFT) necessari ai due nodi di calcolo per "produrre" uno spettro d'ampiezza: calcolo della FFT,

calcolo dello spettro d'ampiezza, overhead dovuto al sistema, overhead per la gestione del MPI. I tempi sono, se pur sensibilmente diversi tra le due macchine e a favore di Giovedì che monta un processore più veloce, in entrambe i casi troppo alti per competere coi moderni hardware specialistici.

Figura 15 – Grafico ottenuto stimando i microsencondi impiegati per svolgere una DFT al variare del numero di canali.



4.2.2 Test 2: variazione della dimensione dei frame.

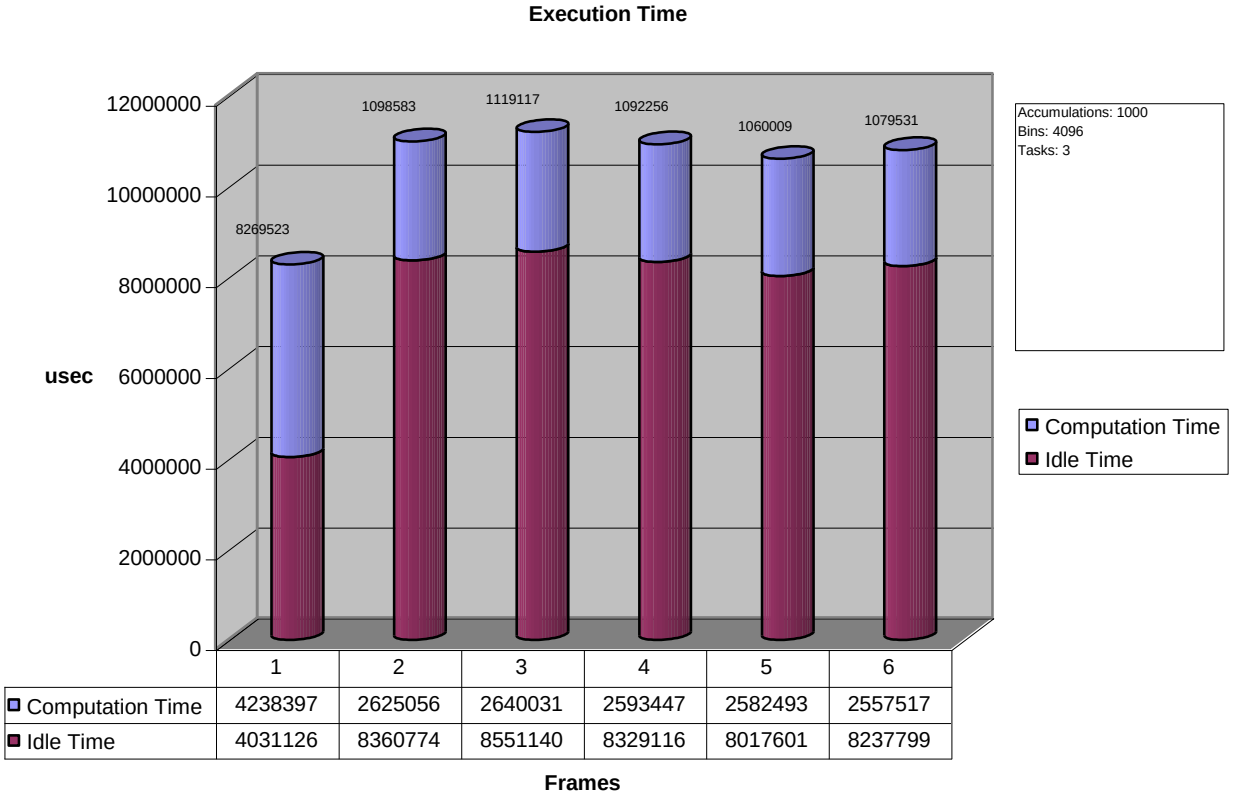
Come spiegato nel Capitolo 3.2 la dimensione del frame è un parametro che stabilisce quanti blocchi dati (necessari a calcolare una singola FFT) il nodo master trasferisce al nodo di calcolo con un solo messaggio, ovvero in una sola volta. Variare questo parametro può essere utile per capire quanto la rete ed in particolare la sua latenza incide sul risultato finale. Sono stati fatti test variando questo parametro dal valore di default 1 fino a 6 e mantenendo fissi il numero di canali e il numero di accumulazioni; i risultati sono riassunti in Figura 16.

L'unico caso che si differenzia è il primo che dimostra prestazioni di circa un venti percento migliori rispetto agli altri, i cui valori sono pressoché identici. Diversi sono i tempi di trasferimento, diversi i tempi di calcolo. L'elemento che maggiormente si distingue è il tempo di idle o trasferimento messaggi che risulta essere circa la metà nel caso migliore; evidentemente per il sistema costituito dalla rete e dal modulo RPI (sysv), la gestione di molti pacchetti dati di dimensioni contenute è meno costosa di quella di pochi pacchetti di dimensioni più elevate. Prima di calcolare la FFT un nodo slave aspetta che tutti i blocchi dati previsti dal parametro frame siano arrivati; questi dati vengono immagazzinati in aree di memoria contigue. Quando la prima FFT è stata completata è probabile che i dati da usare per la seconda siano già nella memoria cache del processore (principio di località spaziale[10]) consentendo un notevole risparmio. Per questo motivo i

[10] La chache di un processore funziona sulla base di due principi: principio di località temporale e principio di località spaziale. Il primo suppone che se un dato è stato utilizzato da poco è probabile che venga nuovamente richiesto per cui

tempi di calcolo quando la dimensione dei frame è maggiore di 1, risultano migliori al caso in cui il blocco dati trasferito è singolo.

Figura 16 – Tempi d'esecuzione ricavati variando la dimensione dei frame, ovvero il numero di blocchi dati trasferiti contemporaneamente dal nodo master ai nodi di calcolo.



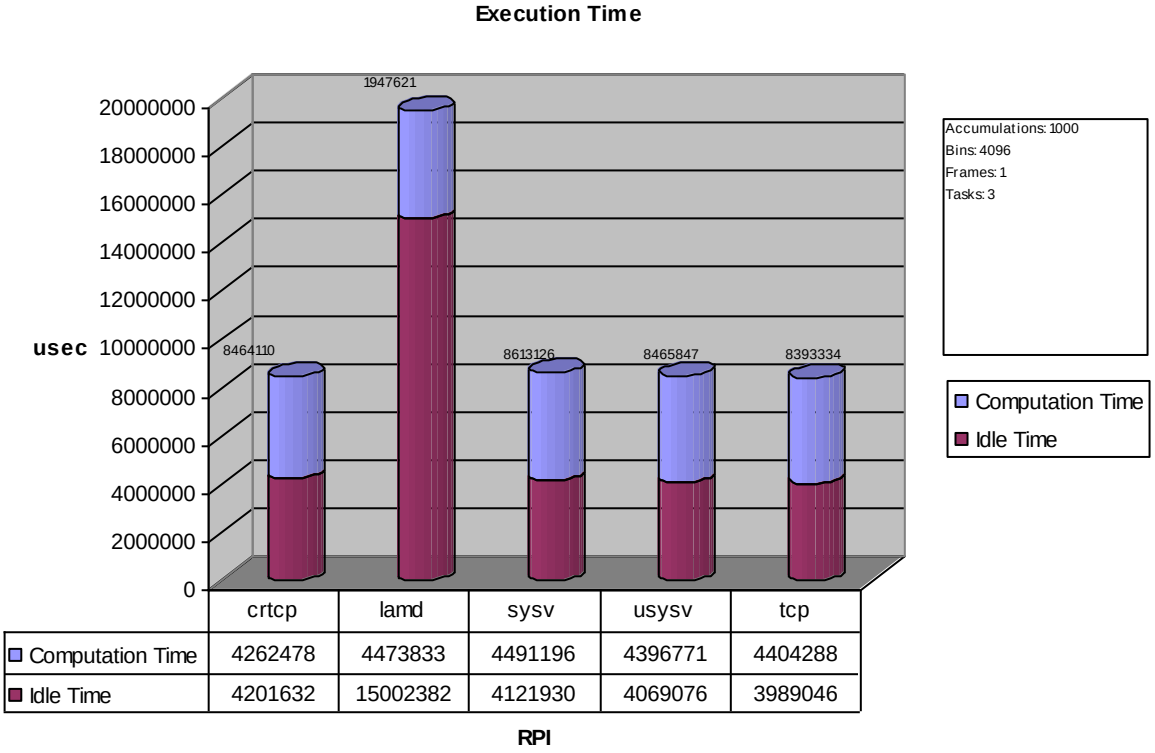
4.2.3 Test 3: variazione dei moduli RPI.

Come descritto nel Capitolo 2.3.1 l'ambiente RTE si basa per la comunicazione punto a punto su diversi moduli di tipo RPI. Uno di questi moduli, elencati in Tabella 6, può essere scelto a run-time. Fissato il numero di canali, il numero d'accumulazioni, la dimensione dei frame, si è provato a variare il modulo utilizzato per capire se uno in particolare fosse più adatto al nostro tipo d'applicazione. I risultati sono illustrati in Figura 17 dove, al solito, i valori sono riportati in milionesimi di secondo. Il valore di questi dati va però pesato considerando che tutti questi moduli hanno la possibilità di essere configurati ed ottimizzati con diversi parametri, ma che per queste misure sono stati lasciati i valori di default.

Dalla figura risulta chiaro che non è emerso nessun modulo particolarmente efficiente: tcp, crtcp, sysv, usysv hanno dato valori simili, nell'ambito dell'errore di misura. E' invece, risultato assolutamente non adatto il modulo lamd in quanto tende a sbilanciare ancora di più il cluster.

è conveniente averlo in cache. Il secondo afferma che se un blocco dati è stato utilizzato è molto probabile che anche il blocco adiacente venga richiesto, per cui quando il blocco usato viene trasferito anche quello successivo viene trasferito.

Figura 17 – Tempi d'esecuzione del cluster ottenuti modificando la scelta del moduli RPI.



4.2.4 Test 4: variazione del numero di processi.

L'ultima prova che pare significativa è variare la configurazione del cluster ed in particolare la distribuzione dei processi sui nodi. Sono state fatte tre misure rispettivamente con 1, 2, 4 processi slave, con FFT da 4096 canali, 1000 accumulazioni e dimensione del frame pari ad 1. Ovviamente il processo master è rimasto uno ed è stato fatto girare su Pcmed14.

Per ottenere questo tipo di configurazione è sufficiente cambiare il file *clusterhosts* che contiene la descrizione e l'organizzazione del cluster durante la fase di boot del sistema fatta col comando *lamboot*. La configurazione con un solo processo di calcolo è stata fatta utilizzando un solo nodo slave (Venerdi), quella con 2 processi è la standard, quella con 4 processi, infine, è stata ottenuta dichiarando che sui nodi erano a disposizione 2 processori e quindi overschedulando con due processi il solo processore realmente installato. Per quest'ultima morfologia si riporta il contenuto del file utilizzato:

pcmed14 cpu=1

giovedi cpu=2

venerdi cpu=2

La Tabella 8 riassume i tre casi ora presentati elencando, per ogni processo i tempi che si sono registrati in microsecondi.

Tabella 8 – Vengono presentati i tempi in microsecondi ottenuti variando il numero di processi di calcolo che girano contemporaneamente sul cluster. Per ogni processo sono riportati il tempo impiegato nella fase di calcolo e quello nella fase di ricezione dei dati (tempo d'attesa).

	Processi Slave: 1			**10957556**
Processo	**Nodo**	**T. Calcolo**	**T. Attesa**	**Blocchi**
1	Venerdi	10052009	913638	1000
	Processi Slave: 2			**8127048**
Processo	**Nodo**	**T. Calcolo**	**T. Attesa**	**Blocchi**
1	Giovedi	994017	7139659	196
2	Venerdi	7027590	1077981	804
	Processi Slave: 4			**8489447**
Processo	**Nodo**	**T. Calcolo**	**T. Attesa**	**Blocchi**
1	Giovedi	514932	7965406	100
2	Giovedì	523828	7583194	104
3	Venerdi	6187970	1895396	367
4	Venerdi	5934065	2164092	429

Dai dati riportati in tabella è possibile fare tre considerazioni. In primo luogo, come era facile attendersi usare un solo processo di calcolo significa far gravare tutto il peso su un solo nodo e un conseguente decadimento delle prestazioni. Secondariamente che l'overschedulazione non porta alcun beneficio apparente, anzi comporta al singolo nodo un overhead per la gestione dei due o più processi. Terzo che lo sbilanciamento del cluster non migliora affatto; basta infatti osservare che il numero totale di blocchi elaborati da Giovedì praticamente non cambia sia che vi siano due o un solo processo.

4.3 Conclusioni.

Generalmente le prestazioni misurate non sono all'altezza né delle previsioni né tanto meno di quelle che possono offrire architetture specializzate. Va comunque sottolineato che i numeri riportati in questo documento sono stati "misurati" senza aver prima fatto un preciso lavoro di ottimizzazione che nella pratica potrebbe portare parecchi benefici. Tra i fattori che caratterizzano il cluster qui descritto su cui varrebbe sicuramente la pena lavorare per ottenere un cluster "ottimizzato", si ricordano:

1. Bilanciamento di carico. Le scarse prestazioni dell'interfaccia di rete di un nodo hanno fatto si che quel nodo non sfruttasse le sue risorse di calcolo per la maggior parte del tempo.
2. Connessione di rete. I nodi sono collegati attraverso una LAN a 100 Mbit; un aggiornamento ad una a 1 Gbit significherebbe già un aumento teorico delle prestazioni di 10 volte.
3. Topografia di rete. Le connessioni di rete di questo cluster passano attraverso uno switch e due hub; l'ideale sarebbe avere una LAN ben delimitata e connessa con un solo switch in cui tra l'altro non passano pacchetti esterni al Cluster che vanno a limitare la banda a disposizione.

4. Ottimizzazione del sistema operativo. L'installazione di partenza del sistema operativo è stata cambiata pochissimo, mentre sarebbe stato utile identificare servizi, demoni e programmi inutili al cluster ma che consumano importanti risorse. Ad esempio i nodi partono col sistema grafico attivo che è notoriamente avido di risorse quali il tempo di calcolo e la memoria.
5. Ottimizzazione dell'ambiente LAM. Come detto l'RTE si basa su diversi moduli, ognuno di questo a parecchi parametri di configurazione. Agire su questi, specialmente quelli relativi al modulo RPI può portare grossi benefici.
6. Ottimizzazione del codice. La routine di FFT non è stata ottimizzata, così come il resto del sorgente. Resta da verificare se il sistema master/slave utilizzato è più performante di altre strategie.

Immaginare quali miglioramenti si potrebbero avere è impossibile, invece è plausibile fare certe supposizioni. Ipotizziamo ora di aver un cluster bilanciato, una connessione di rete ad 1 Gbit (ormai non troppo costose), la possibilità di avere una serie di macchine dalle potenzialità simili a Giovedì, ma senza i problemi di rete. Partendo dalla configurazione qui scelta (paradigma MPI, master/slave, ecc.) e dai numeri riportati è possibile stimare le performance che si potrebbero ottenere?

Fissiamo il caso di FFT da 4096 canali. Dalla Figura 15 un processore simile a Giovedì è in grado di produrre una FFT ogni 5071 microsecondi, il che significa 197.2 FFT al secondo. Il collo di bottiglia rimane comunque la rete. Dalla Figura 5 risulta che una connessione a 100 Mbit può fornire circa 11 MB/sec ovvero circa l'85% della banda teorica di 12.5 MB/sec. Da una connessione ad 1Gbit allora potrebbe essere possibile ricavare una banda passante di circa 106 MB/sec. Analizziamo ora il blocco dati da trasferire. Nel programma di test si sono utilizzati dei sample da 64 bit per ragioni di convenienza ma potrebbero sicuramente esserne sufficienti molto meno, supponiamo 32 (numero di bit per rappresentare un numero intero). Il blocco dati per una FFT reale da 4k punti è 4096x2x8=65536 byte nel programma di test, 32768 byte nel caso che vogliamo ora analizzare. La Figura 14 riporta che Venerdi (macchina senza problemi di rete) per trasferire gli 804 blocchi dati che ha processato del caso 4096 canali ha impiegato nel complesso 10.779 secondi. Questo numero significa 4.89 MB/sec di data rate effettivo (comprensivo di ritardi hardware, software, di sincronizzazione tra i processi ecc.) contro gli 11 MB/sec misurati per un trasferimento puro e semplice, ovvero circa il 56% in meno. Per una rete ad 1 Gbit il data rate diventerebbe allora circa 46,64 Mb/sec che tradotto in blocchi dati sarebbero circa 1424 al secondo. Il numero di macchine necessarie a sostenere questo ritmo sarebbe $1424/197.2 \approx 7$.

Un'ultima considerazione: un cluster, bilanciato, costituito da 7 PC collegati in rete tramite una Gbit LAN potrebbe allora processare in tempo reale con FFT da 4096 canali una banda di 5.8 MHz.

Appendice A: Sorgenti

A.1 Common.h

```
/* COMMON AREA DEFINITION FILE */

extern char error_message[MAX_STRING];
```

A.2 Const.h

```
/* CONSTANT DEFINITION FILE */

#define ERR_PARAM_OUT_RANGE -1
#define MSG_PARAM_OUT_RANGE "Parameter out of range!"
#define ERR_UNKNOW_PARAM -2
#define MSG_UNKNOW_PARAM "Unknown parameter!"

#define MAX_STRING 255
#define MAX_SHORT_STRING 16
#define CFG_BUFFER_LEN 3

#define PI 3,1415929
#define TWOPI 6.2831858

#define TERMINATE_TAG 100
#define STOP_TAG 200
#define INIT_TAG 300
#define DATA_TAG 400
#define SPECTRA_TAG 500
#define INFO_TAG 600
```

A.3 Functions.h

```
/* FUNCTION HEADERS FILE */

int parsecommandline(int argc,char *argv[],T_Config *cfg);
void outmessage(char *format,...);
void outerror(char *format,...);
BOOL IsPowerOfTwo(unsigned x);
void fft_double(unsigned  NumSamples,int InverseTransform,double *RealIn,double
*ImagIn,double *RealOut,double *ImagOut);
unsigned NumberOfBitsNeeded(unsigned PowerOfTwo);
unsigned ReverseBits(unsigned index,unsigned NumBits);
#ifdef __STDC__
void errhandler(MPI_Comm *comm,int *code,...);
#else
void errhandler(MPI_Comm *comm,int *code);
#endif
void slave(MPI_Comm comm);
void master(int argc,char *argv[],MPI_Comm comm,int tasks);
void sendTerminate(MPI_Comm comm,int tasks);
void sendInit(MPI_Comm comm,int tasks,unsigned bins,unsigned accs,unsigned
frames);
void sendStop(MPI_Comm comm,int tasks);
```

```
void acquire(double *buffer,int len,unsigned frames,double freq,double sn);
```

A.4 Types.h

```
/* TYPE DEFINITION FILE */

typedef enum {
      FALSE=0,
      TRUE=1
} BOOL;

typedef struct {
      double freq;
      double snratio;
      unsigned bins;
      unsigned accumulations;
      unsigned group;
      BOOL writefile;
      char filename[MAX_SHORT_STRING];
} T_Config;

typedef struct {
      unsigned executiontime;             //node total execution times in
microseconds
      unsigned idletime;
      unsigned accumulations;
      double *data;
} T_NodeResult;
```

A.5 ClusterFFT.c

```
#include <mpi.h>
#include "const.h"
#include "types.h"
#include "functions.h"
#include <stdio.h>

char error_message[MAX_STRING];

int main (int argc,char *argv[]) {
      int myrank,tasks;
      MPI_Errhandler handler;
      MPI_Comm comm;
      // no error handling...in this case program will automatically abort!!!
      MPI_Init(&argc,&argv);
      // register my error handler....
      MPI_Errhandler_create((MPI_Handler_function*) errhandler,&handler);
      // duplicate communicator....now this communicator will be used.
      MPI_Comm_dup(MPI_COMM_WORLD,&comm);
      // now, use my error handler with just created communicator
      // starting from now every error on MPI will be handled in my finction.
      MPI_Errhandler_set(comm,handler);
      MPI_Comm_rank(comm,&myrank);
      MPI_Comm_size(comm,&tasks);
      if (myrank==0) {
            master(argc,argv,comm,tasks);
      }
      else if (myrank>0) {
            slave(comm);
```

```
	}
	outmessage("Execution halted!");
	MPI_Errhandler_set(comm,MPI_ERRORS_ARE_FATAL);
	MPI_Errhandler_free(&handler);
	MPI_Barrier(comm);
	MPI_Finalize();
	return 0;
}

#ifdef __STDC__
void errhandler(MPI_Comm *comm,int *code,...)
#else
void errhandler(MPI_Comm *comm,int *code)
#endif
{
	char msg[MPI_MAX_ERROR_STRING];
	int len;
	int classcode;
	MPI_Error_class(*code,&classcode);
	MPI_Error_string(classcode,msg,&len);
	outerror("MPI error %d: %s",*code,msg);
	MPI_Abort(*comm,classcode);
	abort();
}
```

A.6 FFT.c

```
/*************************************************************************
***
FFT Sprecta statistical analisis.
/*************************************************************************
****/
#include <stdlib.h>
#include <math.h>
#include "const.h"
#include "types.h"

BOOL IsPowerOfTwo(unsigned x)
{
	if (x<2) return FALSE;
	if (x&(x-1)) return FALSE;
	return TRUE;
}

unsigned NumberOfBitsNeeded(unsigned PowerOfTwo)
{
	unsigned i;
	if (PowerOfTwo<2) {
		return 0;
	}
	for(i=0;;i++) {
		if (PowerOfTwo & (1 << i))
			return i;
	}
}

unsigned ReverseBits(unsigned index,unsigned NumBits)
{
	unsigned i,rev;
	for (i=rev=0;i<NumBits;i++) {
		rev=(rev<<1)|(index & 1);
```

```
            index>>=1;
      }
      return rev;
}

void fft_double(unsigned  NumSamples,int InverseTransform,double *RealIn,double
*ImagIn,double *RealOut,
                                 double *ImagOut)
{
      unsigned NumBits;    /* Number of bits needed to store indices */
      unsigned i,j,k,n;
      unsigned BlockSize,BlockEnd;
      double angle_numerator;
      double tr, ti;     /* temp real, temp imaginary */

      angle_numerator=2.0*PI;
      if (!IsPowerOfTwo(NumSamples)) {
            exit(1);
      }
      if (InverseTransform) angle_numerator=-angle_numerator;
      NumBits=NumberOfBitsNeeded(NumSamples);
   /*
   **   Do simultaneous data copy and bit-reversal ordering into outputs...
   */
      for (i=0;i<NumSamples;i++) {
            j=ReverseBits(i,NumBits);
            RealOut[j]=RealIn[i];
            ImagOut[j]=(ImagIn==NULL)? 0.0:ImagIn[i];
      }
   /*
   **   Do the FFT itself...
   */
      BlockEnd = 1;
      for (BlockSize=2;BlockSize<=NumSamples;BlockSize<<=1) {
            double delta_angle=angle_numerator/(double)BlockSize;
            double sm2=sin(-2*delta_angle);
            double sm1=sin(-delta_angle);
            double cm2=cos(-2 * delta_angle);
            double cm1=cos(-delta_angle);
            double w=2*cm1;
            double ar[3],ai[3];
                  for (i=0;i<NumSamples;i+=BlockSize ) {
                        ar[2]=cm2;
                        ar[1]=cm1;
                        ai[2]=sm2;
                        ai[1]=sm1;
                        for (j=i,n=0;n<BlockEnd;j++,n++) {
                              ar[0]=w*ar[1]-ar[2];
                              ar[2]=ar[1];
                              ar[1]=ar[0];
                              ai[0]=w*ai[1]-ai[2];
                              ai[2]=ai[1];
                              ai[1]=ai[0];
                              k=j+BlockEnd;
                              tr=ar[0]*RealOut[k]-ai[0]*ImagOut[k];
                              ti=ar[0]*ImagOut[k]+ai[0]*RealOut[k];
                              RealOut[k]=RealOut[j]-tr;
                              ImagOut[k]=ImagOut[j]-ti;
                              RealOut[j]+=tr;
                              ImagOut[j]+=ti;
                        }
                  }
                  BlockEnd=BlockSize;
```

```
      }
   /*
   **   Need to normalize if inverse transform...
   */
      if (InverseTransform) {
            double denom=(double)NumSamples;
            for (i=0;i<NumSamples;i++) {
                  RealOut[i]/=denom;
                  ImagOut[i]/=denom;
            }
      }
}
```

A.7 Master.c

```
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <time.h>
#include <mpi.h>
#include "const.h"
#include "types.h"
#include "common.h"
#include "functions.h"

void master(int argc,char *argv[],MPI_Comm comm,int tasks)
{
      int slaves,flg;
      int ierr,j,i,k;
      T_Config conf;
      double *buffers,*outbuff;
      T_NodeResult *nodes;
      double gstart,gstop,gsum;
      unsigned buffersize;
      unsigned doneblocks;
      unsigned totidletime,totexectime;
      BOOL *flags;
      FILE *outfile;
      int cfgbuff[CFG_BUFFER_LEN];
      MPI_Request *req;
      MPI_Status status;
      ierr=parsecommandline(argc,argv,&conf);
      if (ierr==-100) {   // stampa help in linea......
            outmessage("goFFT [-bBins] [-aAccumulations] [-fFrequency] [-
sSNRatio] [-gFrames] [-wFileName] [-h]");
            outmessage("-bBins           Number of bins of the FFT (32-65536),
default 2048");
            outmessage("-bAccumulations  Number of accumulations (1-10000),
default 50");
            outmessage("-fFrequency      Frequency of generated fake signal,
default 512.0");
            outmessage("-sSNRatio        Signal to noise ratio of generated fake
signal, default 0.2");
            outmessage("-gFrames         Number of frames sent to a single nodes
at the same time, default 1");
            outmessage("-w               Write computed spectrum into specified
name");
            outmessage("-h               Shows this help");
            sendTerminate(comm,tasks);
            return;
```

```
      }
      else if (ierr<0) {
            outerror("error %d: %s",ierr,error_message);
            sendTerminate(comm,tasks);
            return;
      }
      else {
            outmessage("number of bins          : %d",conf.bins);
            outmessage("number of accumulations : %d",conf.accumulations);
            outmessage("signal frequency        : %lf",conf.freq);
            outmessage("signal to noise ratio   : %lf",conf.snratio);
            outmessage("number of frames        : %d",conf.group);
            outmessage("number of tasks         : %d",tasks);
      }
      slaves=tasks-1;
      // allocate data buffers.....
      buffersize=(conf.bins*2)*conf.group;
      buffers=(double *)calloc(slaves*buffersize,sizeof(double));
      outbuff=(double *)calloc(conf.bins,sizeof(double));
      flags=(BOOL *)calloc(slaves,sizeof(BOOL));
      req=(MPI_Request *)calloc(slaves,sizeof(MPI_Request));
      nodes=(T_NodeResult *)calloc(slaves,sizeof(T_NodeResult));
      for (i=0;i<slaves;i++) {
            nodes[i].data=(double *)calloc(conf.bins,sizeof(double));
      }
      // randomization seed....
      srand((unsigned)time(NULL));
      // produce spectrum.............
      outmessage("");
      outmessage("Computations begin......");

      //init flags.............
      for(i=0;i<slaves;flags[i]=TRUE,i++);
      for(i=0;i<conf.bins;outbuff[i]=0.0,i++);
      k=i=0;
      gstart=MPI_Wtime();
      // init computation node....wait for the initialization to complete!!!!
      sendInit(comm,tasks,conf.bins,(unsigned)(conf.accumulations/slaves),conf.g
roup);
      while(i<conf.accumulations) {
            // if previous transmission has completed send another buffer!!!
            if (flags[k]) {

      acquire(buffers+(buffersize*k),buffersize,conf.group,conf.freq,conf.snrati
o);

      MPI_Isend(buffers+(buffersize*k),buffersize,MPI_DOUBLE,k+1,DATA_TAG,comm,&
(req[k]));
                  i+=conf.group;
                  // disable write flag....no further messages possible to this
slave.
                  flags[k]=FALSE;
            }
            else {
                  // check if request has completed. In this case re-enable
write flag!
                  MPI_Test(&(req[k]),&flg,&status);
                  if (flg) {
                        flags[k]=TRUE;
                  }
            }
            k++; k%=slaves;
      }
```

```
	sendStop(comm,tasks);
	gstop=MPI_Wtime();
	outmessage("Spectrum completed......");
		//fetch of spectra.....................
	outmessage("...now fetching data");
	doneblocks=0;
	totidletime=totexectime=0;
	for(i=0;i<slaves;i++) {

	MPI_Recv(nodes[i].data,conf.bins,MPI_DOUBLE,i+1,SPECTRA_TAG,comm,&status);

	MPI_Recv(cfgbuff,CFG_BUFFER_LEN,MPI_UNSIGNED,i+1,INFO_TAG,comm,&status);
		nodes[i].idletime=cfgbuff[2];
		nodes[i].executiontime=cfgbuff[1];
		nodes[i].accumulations=cfgbuff[0];
		doneblocks+=nodes[i].accumulations;
		totidletime+=nodes[i].idletime;
		totexectime+=nodes[i].executiontime;
	}
	// reconstract data....
	for (i=0;i<slaves;i++) {
		for (j=0;j<conf.bins;j++) {

	outbuff[j]+=nodes[i].data[j]*((double)nodes[i].accumulations/(double)doneb
locks);
		}
	}
	// write output file.............
	if (conf.writefile) {
		outmessage("Writing output file.....");
		outfile=fopen(conf.filename,"w+");
		if (outfile==NULL) {
			outerror("Error %d while opening output file:
%s",errno,strerror(errno));
		}
		else {
			fprintf(outfile,"#Freqeuncy: %lf\n",conf.freq);
			fprintf(outfile,"#SNRatio: %lf\n",conf.snratio);
			fprintf(outfile,"#Bins: %d\n",conf.bins);
			fprintf(outfile,"#Accumulations: %d\n",conf.accumulations);
			for (i=0;i<conf.bins;i++) {
				fprintf(outfile,"%d %f\n",i,outbuff[i]);
			}
			fclose(outfile);
		}
	}
	// writing results........
	gsum=gstop-gstart;
	outmessage("");
	outmessage("Total execution time (us): %d",(unsigned)(gsum*1000000.0));
	outmessage("Done blocks: %d",doneblocks);
	outmessage("Lost blocks: %d",conf.accumulations-doneblocks);
	outmessage("Mean computation time (us): %d",(unsigned)totexectime/slaves);
	outmessage("Mean idle time (us): %d",(unsigned)totidletime/slaves);
	for (i=0;i<slaves;i++) {
		outmessage("");
		outmessage("Node %d stats:",i+1);
		outmessage("Computation time (us): %d",nodes[i].executiontime);
		outmessage("Idle time (us): %d",nodes[i].idletime);
		outmessage("Done blocks: %d",nodes[i].accumulations);
	}
	outmessage("");
	sendTerminate(comm,tasks);
```

```
	// free buffers.....
	free(buffers);
	free(flags);
	free(req);
	free(outbuff);
	for(i=0;i<slaves;i++) free(nodes[i].data);
	free(nodes);
}

void acquire(double *buffer,int len,unsigned frames,double freq,double sn)
{
	int x,y;
	unsigned framelen;
	framelen=len/frames;
	double omega,arg;
	omega=TWOPI*freq;
	for (y=0;y<frames;y++) {
		for(x=0;x<framelen;x++) {
		arg=omega*x/framelen;
			buffer[x+(y*framelen)]=sin(arg);
			buffer[x+(y*framelen)]+=(1/sn)*(double)rand()/RAND_MAX;
		}
	}
}

// send all processes the TERMINATE message and wait until all received it
void sendTerminate(MPI_Comm comm,int tasks)
{
	unsigned dummy,i;
	MPI_Request *req;
	MPI_Status status;
	req=(MPI_Request *)calloc(tasks-1,sizeof(MPI_Request));
	for(i=1;i<tasks;i++) {
		MPI_Isend(&dummy,1,MPI_UNSIGNED,i,TERMINATE_TAG,comm,&(req[i-1]));
	}
	for(i=1;i<tasks;i++) {
		MPI_Wait(&(req[i-1]),&status);
	}
	free(req);
}

// send initialization buffer.............
void sendInit(MPI_Comm comm,int tasks,unsigned bins,unsigned accs,unsigned
frames)
{
	unsigned buffer[CFG_BUFFER_LEN];
	int i;
	MPI_Request *req;
	MPI_Status status;
	req=(MPI_Request *)calloc(tasks-1,sizeof(MPI_Request));
	buffer[0]=bins;
	buffer[1]=accs;
	buffer[2]=frames;
	for(i=1;i<tasks;i++) {

	MPI_Isend(buffer,CFG_BUFFER_LEN,MPI_UNSIGNED,i,INIT_TAG,comm,&(req[i-1]));
	}
	for(i=1;i<tasks;i++) {
		MPI_Wait(&(req[i-1]),&status);
	}
	free(req);
}
```

```
void sendStop(MPI_Comm comm,int tasks)
{
      unsigned dummy,i;
      MPI_Request *req;
      MPI_Status status;
      req=(MPI_Request *)calloc(tasks-1,sizeof(MPI_Request));
      for(i=1;i<tasks;i++) {
            MPI_Isend(&dummy,1,MPI_UNSIGNED,i,STOP_TAG,comm,&(req[i-1]));
      }
      for(i=1;i<tasks;i++) {
            MPI_Wait(&(req[i-1]),&status);
      }
      free(req);
}
```

A.8 Outerror.c

```
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>

void outerror(char *format,...)
{
      char *rank;
      va_list ap;
      va_start(ap,format);
      rank=getenv("LAMRANK");
      fprintf(stderr,"rank %s ",rank);
      vfprintf(stderr,format,ap);
      fprintf(stderr,"\n");
      fflush(stderr);
      va_end(ap);
}
```

A.9 Outmessage.c

```
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>

void outmessage(char *format,...)
{
      char *rank;
      va_list ap;
      if (strcmp(format,"")==0) {
                  fprintf(stdout,"\n");
      }
      else {
            va_start(ap,format);
            rank=getenv("LAMRANK");
            fprintf(stdout,"rank %s ",rank);
            vfprintf(stdout,format,ap);
            fprintf(stdout,"\n");
            fflush(stdout);
            va_end(ap);
      }
}
```

A.10 Parsecommandline.c

```
#include <stdlib.h>
#include <mpi.h>
#include "const.h"
#include "types.h"
#include "common.h"
#include "functions.h"

int parsecommandline(int argc,char *argv[],T_Config *cfg)
{
      int i;
      char sw[3],value[MAX_SHORT_STRING];
      /* Set defaults values */
      cfg->bins=2048;
      cfg->accumulations=500;
      cfg->writefile=FALSE;
      cfg->freq=512;
      cfg->snratio=0.2;
      cfg->group=1;
      for(i=1;i<argc;i++) {
            strncpy(sw,argv[i],2);
            sw[2]=0;
            strncpy(value,&(argv[i][2]),MAX_SHORT_STRING);
            if (strcmp(sw,"-h")==0) {
                  return -100;
            }
            else if (strcmp(sw,"-b")==0) {
                  cfg->bins=atoi(value);
                  if ((cfg->bins<32) || (cfg->bins>65536)) {
                        strcpy(error_message,MSG_PARAM_OUT_RANGE);
                        return ERR_PARAM_OUT_RANGE;
                  }
                  if (!IsPowerOfTwo(cfg->bins)) {
                        strcpy(error_message,MSG_PARAM_OUT_RANGE);
                        return ERR_PARAM_OUT_RANGE;
                  }
            }
            else if(strcmp(sw,"-a")==0) {
                  cfg->accumulations=atoi(value);
                  if ((cfg->accumulations<1) || (cfg->accumulations>10000)) {
                        strcpy(error_message,MSG_PARAM_OUT_RANGE);
                        return ERR_PARAM_OUT_RANGE;
                  }
            }
            else if (strcmp(sw,"-g")==0) {
                  cfg->group=atoi(value);
                  if (cfg->group>cfg->accumulations) {
                        strcpy(error_message,MSG_PARAM_OUT_RANGE);
                        return ERR_PARAM_OUT_RANGE;
                  }
            }
            else if (strcmp(sw,"-f")==0) {
                  cfg->freq=strtod(value,(char**)NULL);
                  if (cfg->freq<=0) {
                        strcpy(error_message,MSG_PARAM_OUT_RANGE);
                        return ERR_PARAM_OUT_RANGE;
                  }
            }
            else if (strcmp(sw,"-s")==0) {
                  cfg->snratio=strtod(value,(char **)NULL);
                  if (cfg->snratio<=0) {
```

```
                        strcpy(error_message,MSG_PARAM_OUT_RANGE);
                        return ERR_PARAM_OUT_RANGE;
                  }
            }
            else if(strcmp(sw,"-w")==0) {
                  cfg->writefile=TRUE;
                  strcpy(cfg->filename,value);
            }
            else {
                  strcpy(error_message,MSG_UNKNOW_PARAM);
                  return ERR_UNKNOW_PARAM;
            }
      }
      return 0;
}
```

A.11 Slave.c

```
#include <stdlib.h>
#include <math.h>
#include <mpi.h>
#include "const.h"
#include "types.h"
#include "common.h"
#include "functions.h"

void slave(MPI_Comm comm)
{
      MPI_Status cfgstatus,status;
      unsigned cfgbuffer[CFG_BUFFER_LEN],i;
      double *buffer,*realout,*imgout,*pwr,app;
      double start,stop,sum;
      double idlestart,idlestop,idlesum;
      int counter,x;
      BOOL finished=FALSE;
      BOOL ok=FALSE;
      unsigned binsnumber,datasize,frames,framesize;

      double accumulations;
      while (!finished) {
            //receive messages from master, rank=0

      MPI_Recv(cfgbuffer,CFG_BUFFER_LEN,MPI_UNSIGNED,0,MPI_ANY_TAG,comm,&cfgstat
us);
            if (cfgstatus.MPI_TAG==TERMINATE_TAG) {
                  finished=TRUE;
            }
            else if (cfgstatus.MPI_TAG==INIT_TAG) {
                  //inizializzati per il calcolo....
                  binsnumber=cfgbuffer[0];
                  frames=cfgbuffer[2];
                  framesize=binsnumber*2;
                  datasize=framesize*frames;
                  accumulations=(double)cfgbuffer[1];
                  buffer=(double *)calloc(datasize,sizeof(double));
                  realout=(double *)calloc(datasize,sizeof(double));
                  imgout=(double *)calloc(datasize,sizeof(double));
                  pwr=(double *)calloc(binsnumber,sizeof(double));
                  for(i=0;i<binsnumber;pwr[i]=0.0,i++);
                  counter=0;
                  sum=idlesum=0.0;
```

```
                  // begin computations.............
                  while (!ok) {
                        idlestart=MPI_Wtime();

      MPI_Recv(buffer,datasize,MPI_DOUBLE,0,MPI_ANY_TAG,comm,&status);
                        idlestop=MPI_Wtime();
                        idlesum+=(idlestop-idlestart);
                        if (status.MPI_TAG==TERMINATE_TAG) {
                              ok=finished=TRUE;
                        }
                        else if (status.MPI_TAG==STOP_TAG) {
                              // calibrate data for send back....
                              if ((int)accumulations!=counter) {
                                    double ratio;
                                    ratio=accumulations/(double)counter;
                                    for(i=0;i<binsnumber;pwr[i]*=ratio,i++);
                              }

      MPI_Send(pwr,binsnumber,MPI_DOUBLE,0,SPECTRA_TAG,comm);
                              cfgbuffer[0]=counter;
                              cfgbuffer[1]=(unsigned)(sum*1000000);
                              cfgbuffer[2]=(unsigned)(idlesum*1000000);

      MPI_Send(cfgbuffer,CFG_BUFFER_LEN,MPI_UNSIGNED,0,INFO_TAG,comm);
                              ok=TRUE;
                        }
                        else if (status.MPI_TAG==DATA_TAG) {
                              // compute power spectrum of input data.......and
take execution time
                              start=MPI_Wtime();
                              for(x=0;x<frames;x++) {

      fft_double(framesize,0,buffer+(x*framesize),NULL,realout,imgout);
                                    for (i=0;i<binsnumber;i++) {

      app=sqrt(realout[i]*realout[i]+imgout[i]*imgout[i]);
                                          pwr[i]+=app/accumulations;
                                    }
                                    counter++;
                              }
                              stop=MPI_Wtime();
                              sum+=stop-start;
                        }
                  }
                  free(buffer);
                  free(realout);
                  free(imgout);
                  free(pwr);
            }
      }
}
```

Riferimenti

[1] **LAM/MPI Installation Guide Version 7.0.4** Pervasivetechnologylabs, at Indiana University.

[2] **LAM/MPI User's Guide Version 7.0.4** Pervasivetechnologylabs, at Indiana University.

[3] http://www.lam-mpi.org