

ALMA MATER STUDIORUM – UNIVERSITA' DI BOLOGNA
SEDE DI CESENA
FACOLTA' DI SCIENZE MATEMATICHE, FISICHE E NATURALI
CORSO DI LAUREA IN SCIENZE DELL'INFORMAZIONE

TITOLO DELLA TESI

**PROGETTAZIONE ED IMPLEMENTAZIONE DI UN
SISTEMA DI CALCOLO IBRIDO
MULTITHREAD-MULTIPROCESSO PER HPC:
APPLICAZIONE ALL'IMAGING MEDICO**

Tesi di laurea in

Fisica Numerica

Relatore

Chiar.mo Prof. Renato Campanini

Presentata da

Omar Schiaratura

Correlatore

Dott. Matteo Roffilli

Sessione I

Anno Accademico 2002/2003

Alla mia famiglia,
sostegno immancabile e irrinunciabile.

“Zengzi disse: <<Ogni giorno su tre punti esamino me stesso: nell’agire per gli altri sono stato leale? Nei rapporti con gli amici sono stato di parola? mi sono applicato a quel che mi fu trasmesso?>>”

*Confucio
(traduzione)*

Desidero ringraziare tutte le persone che mi hanno aiutato in questo lavoro di tesi, in particolare:

La mia famiglia per avermi consentito l'iscrizione all'Università ed avermi sostenuto finanziariamente e moralmente;

Il professor Campanini per la sua disponibilità e per aver permesso la realizzazione del progetto;

Il dottor Roffilli per avermi proposto il soggetto della tesi ed essersi dimostrato un amico;

La professoressa Lazzaro per avermi iniziato alla disciplina dell'Analisi Numerica grazie alla sua dedizione al lavoro ed alla sua cortesia;

Marco per la sua amicizia e sostegno nei momenti difficili;

Elisa, che ha rappresentato un importante aiuto nel compimento di questo traguardo.

INDICE

Introduzione.....	I
--------------------------	----------

1. Architetture parallele.....	1
---------------------------------------	----------

1.1 HPC e architetture parallele.....	1
1.1.1 SMP.....	2
1.1.2 MPPS.....	4
1.1.3 Cluster.....	8
1.1.4 GRID.....	9
1.2 SMP Intel Vs SMP AMD.....	11
1.3 Beowulf.....	13
1.4 openMosix.....	15
1.4.1 openMosix internals.....	16
1.4.2 File system.....	18
1.4.3 Gestione del cluster e migrazione forzata.....	19
1.5 Beowulf Vs Mosix.....	20

2. Progettazione di un cluster.....	23
--	-----------

2.1 Progettazione.....	23
2.2 Perché Linux.....	24
2.3 Hardware adottato.....	26
2.4 Installazione della rete.....	28
2.5 Configurazione del server DHCP.....	29
2.6 File system.....	33
2.7 Differenze nel FS tra master e nodi.....	37
2.8 Configurazione del kernel.....	37

2.9 Configurazione della rete sui nodi.....	39
2.10 Configurazione openMosix.....	41
2.11 Installazione MPI.....	43
2.12 NYS e autenticazione centralizzata.....	44
2.13 Risoluzione dei nomi.....	45
2.14 Sicurezza.....	46
 3. Metodologie software per l'ottimizzazione di algoritmi...49	
3.1 SIMD.....	50
3.1.1 SSE.....	51
3.1.2 Allineamento dei dati.....	54
3.1.3 Prefetch del codice ed istruzione di prefetch.....	57
3.2 Processi.....	61
3.2.1 IPC.....	63
3.2.2 Signals.....	64
3.2.3 File locking.....	66
3.2.4 Pipe e fifo.....	68
3.3 Thread.....	70
3.3.1 Linux thread.....	72
3.3.2 Semafori.....	74
3.4 MPI.....	76
 4. Prestazioni del cluster e toy-test.....81	
4.1 Valutazione prestazioni generali del cluster.....	82
4.2 Toy test.....	82
 5. Applicazione reale.....89	
5.1 Introduzione all'applicazione.....	89

5.2	Il CAD.....	92
5.3	Mammografi digitali.....	93
5.4	Descrizione del sistema.....	94
5.4.1	Riconoscimento.....	95
5.5	Ottimizzazione multilivello dell'applicazione.....	97
5.6	Test e valutazioni finali.....	105
5.7	Conclusioni e sviluppi futuri.....	112

Conclusioni.....	III
-------------------------	------------

Bibliografia.....	V
--------------------------	----------

Introduzione

Il tumore al seno è statisticamente la prima causa di morte nelle donne tra i 40 ed i 55 anni di età¹.

Il modo più efficace per poter diagnosticare questo tipo di patologie e di conseguenza intervenire tempestivamente è l'esame mammografico effettuato con apparecchiature radiodiagnostiche.

Purtroppo, il 25% di queste patologie sfugge al controllo del radiologo, mentre una percentuale del 17% viene erroneamente diagnosticata da questi.

Per aumentare sensibilmente l'accuratezza delle letture sarebbe necessaria la lettura delle lastre da parte di un secondo radiologo o in alternativa l'utilizzo di un sistema computerizzato specifico per questo tipo di problemi.

Un gruppo di ricerca dell'Università degli Studi di Bologna, costituito da ricercatori del Corso di laurea di Scienze dell'Informazione (sede di Cesena) e del Dipartimento di Fisica ha sviluppato un programma di aiuto al radiologo (CAD, Computer Aided Detection).

In particolare, questo programma CAD è basato sulle più avanzate metodologie di elaborazione di immagini, rivelazioni di oggetti e loro classificazione ed è uno dei pochi CAD al mondo che utilizza come classificatori le Support Vector Machine (da ora SVM), frutto delle più moderne ricerche della Statistical Learning Theory.

Una delle maggiori limitazioni che sorgono quando si ha a che fare con l'elaborazione di immagini in generale ed in particolare con questo tipo di immagini dovuta all'enorme capacità computazionale richiesta dagli algoritmi utilizzati, è il tempo di calcolo elevato che fa venire a meno

¹ Greenlee, Hill-Hammon, Murray, Thun 2001

quella che deve essere una delle più importanti caratteristiche del CAD stesso: la capacità di analizzare le immagini in tempo reale.

Nel corso di questo lavoro, sono state studiate e implementare soluzioni di HPC² che particolarmente si prestano alla soluzione di problemi in cui si devono svolgere una mole consistente di operazioni su matrici.

Verranno quindi presentati metodi di ottimizzazione basate su:

- istruzioni SIMD³*
- macchine multiprocessore*
- cluster per il calcolo parallelo.*

Di quest'ultimo sarà implementata una soluzione su rete ethernet.

Infine, viene presentata l'analisi delle modifiche apportate al programma per avvantaggiarsi di queste tecnologie, e come sia possibile combinarle.

² HPC: High Performance Computation

³ SIMD: Single Instruction Multiple Data

Capitolo 1

Architetture parallele

Vediamo nel seguito cosa si intende per HPC¹, illustreremo una breve rassegna di architetture parallele per comprendere meglio lo stato attuale dell'arte, per poi focalizzarci sui cluster ed in particolar modo i Beowulf² e i SSI³.

1.1 HPC e architetture parallele

La parola HPC indica l'uso di tecnologie e tecniche di calcolo necessarie alla risoluzione di problemi che presentando una complessità elevata, richiedono tempi di elaborazione molto lunghi.

Per risolvere questi problemi ci si è resi conto che la macchina di Von Neumann, su cui ancora si basano i calcolatori seriali ed i PC⁴, non presentava un modello adatto per problemi di tipo complesso, in particolare, alcuni problemi risultano non risolubili su architetture che permettono l'esecuzione seriale delle istruzioni.

I problemi in questione riguardano simulazioni fisiche complesse ed analisi di dati come nell'elaborazione di immagini, nell'analisi

¹ HPC: High Performance Computing

² <http://www.beowulf.org/>

³ SSI: Single System Image

⁴ PC: Personal Computer

spettroscopica, nei calcoli ingegneristici dei voli spaziali, analisi del terreno per la ricerca mineraria o petrolifera, in generale ovunque si ha a che fare con operazioni tra matrici, o dove i dati sono rappresentabili da matrici.

L'intrinseca complessità di questi algoritmi, dovuta alla mole di operazioni da eseguire, mette in risalto la limitazione del modello di Von Newman consistente in un processore ed una memoria collegati mediante un bus, suggerendo un modello esteso consistente in un insieme di processori collegati alla loro memoria privata tramite bus indipendenti dagli altri processori, ed interconnessi tra loro mediante un bus, in genere più lento.

Tali sistemi sono detti multiprocessori.

Oggigiorno è disponibile una gran varietà di HW⁵ di questo tipo dedicato al calcolo HPC, possiamo classificarli nelle seguenti architetture che illustreremo nel prosieguo:

- 1- SMP⁶
- 2- MPPS⁷
- 3- Cluster
- 4- GRID⁸

1.1.1 SMP

I SMP sono macchine con un numero variabile di processori, in genere da 2 a 64, che condividono lo stessa memoria.

⁵ HW: HardWare

⁶ SMP: Simmetric Multi Procesing

⁷ MPPS: Massively Parallel Processing System

⁸ GRID: griglia

Presentano il vantaggio di consentire comunicazioni molto veloci tra di loro, in quanto i dati utilizzati da un processore sono immediatamente disponibili a tutti.

Lo svantaggio è il costo del sistema, esponenziale rispetto al numero di processori e la complessità di progettazione dello stesso che ne riduce la scalabilità.

Il loro costo elevato al crescere del numero di CPU⁹, li rende non più idonei ai problemi di HPC, pertanto il loro utilizzo è principalmente quello di fornire servizi di rete e sono molto diffusi nelle workstation di fascia elevata nella configurazione a 2 o 4 processori.

Nella configurazione più semplice, questi sistemi constano di un bus dati condiviso tra i processori per accedere alla memoria.

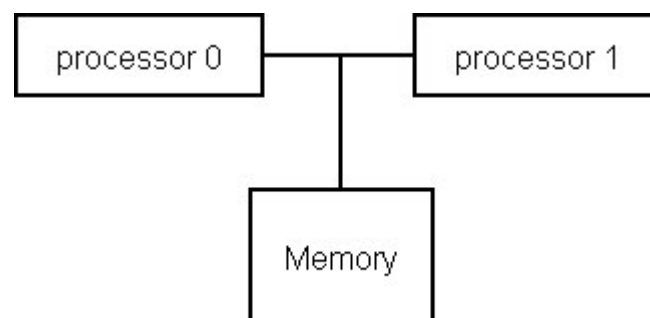


Figura 1 – schema SMP nella sua configurazione più semplice

Sui processori Intel¹⁰ e cloni vi è la consuetudine di integrare la logica per il supporto SMP direttamente all'interno del core della CPU, alleggerendo la progettazione dei chipset, che non devono gestire la logica di sincronizzazione e mutuo accesso al bus nel caso di due processori, mentre consentono una semplificazione di questi circuiti nel caso di più di 2 CPU.

⁹ CPU: Central Processing Unit

¹⁰ Intel è un marchio protetto
<http://www.intel.com/>

Sui sistemi Intel iA32¹¹, il bus implementato permette di utilizzare in modo accettabile un massimo di 8 processori, dopodiché le prestazioni iniziano a degradarsi in modo da non rendere più competitivo il loro rapporto prezzo/prestazioni; sui sistemi RISC¹² il discorso è diverso; in genere esistono sistemi a 64 processori con prestazioni ancora accettabili, come i sistemi SUN¹³, che permettono di utilizzarne oltre 100, grazie ad un bus di sistema progettato in modo migliore.

Nel paragrafo 1.3 analizzeremo più dettagliatamente le architetture SMP di Intel e AMD¹⁴ e le confronteremo.

1.1.2 MPPS

I MPPS sono sistemi con un gran numero di processori identici, in genere ognuno con una memoria locale ed interconnessi tramite un bus proprietario ad alta velocità, ma possono essere costituiti anche da sistemi SMP, dai costi però proibitivi.

Ne sono esempi celebri il Cray di SGI¹⁵, l'IBM SP¹⁶, l'SGI Origin, il SUN Fire 15k.

Le macchine meno recenti utilizzano una struttura a toro 3d e sono macchine a memoria distribuita, in cui cioè ogni processore possiede una memoria propria non accessibile direttamente agli altri.

¹¹ iA32: Intel Architecture 32 bit

¹² RISC: Restricted Instruction Set CPU

¹³ <http://www.sun.com>

¹⁴ AMD: Advanced Micro Device è un marchio protetto
<http://www.amd.com>

¹⁵ SGI è un marchio protetto <http://www.sgi.com>

¹⁶ IBM è un marchio protetto <http://www.ibm.com>

Ad esempio il Cray T3, dove ogni processore dialoga direttamente con i 6 processori adiacenti ad esso, anche se è possibile utilizzare porzioni di memoria condivisa con altri processori.



Figura 2 – Cray X1



Figura 3 – Cray T3E

Ogni processore ha una memoria a cui accede direttamente e dialoga con gli altri processori mediante RPC¹⁷ attraverso un'interfaccia a message passing.

Macchine come l'IBM SP hanno un approccio diverso: esse sono caratterizzate da nodi SMP da 16 processori ciascuno: ogni nodo può essere collegato agli altri mediante una rete ad alte prestazioni e bassa latenza.

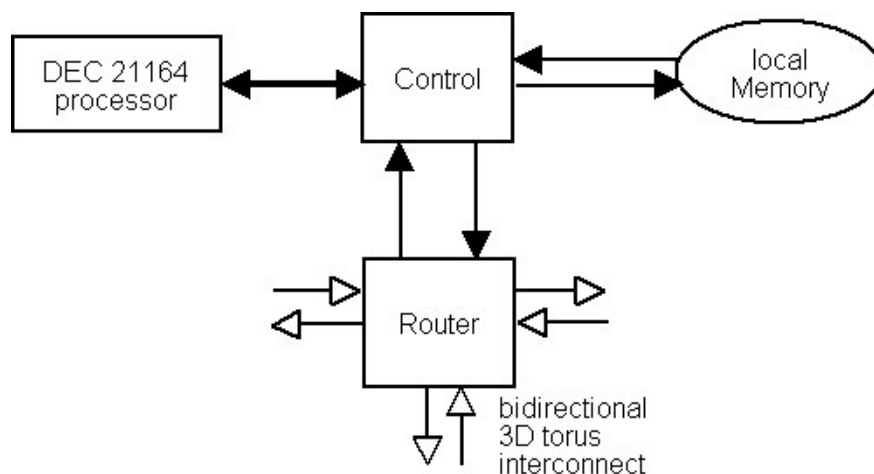


Figura 4 - Schema d interconnessione del Cray T3E

¹⁷ RPC: Remote Procedure Call

Queste macchine utilizzano un sistema ibrido di funzionamento che permette loro di funzionare sia come sistemi SMP che come sistemi distribuiti.

Una tecnologia molto utilizzata su sistemi MPPS è NUMA¹⁸, in contrapposizione a UMA¹⁹ in cui l'accesso alla memoria risulta dato dagli stessi tempi e con stessa latenza.

I sistemi che utilizzano tale tecnologia possono funzionare sia come sistemi a memoria distribuita (multicomputer), che come macchine con un'unica immagine della memoria (max 1 Terabyte su SGI Origin 3000).

In particolare, NUMAflex© è una tecnologia brevettata da SGI ed utilizzata sui suoi supercomputer.

Il sistema è, in questo caso, diviso in moduli nominati C-bricks da 2 o 4 CPU, ogni modulo fa riferimento ad una memoria propria con un massimo di 8 GB.

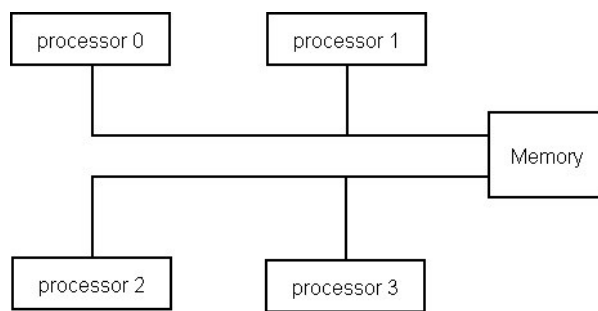
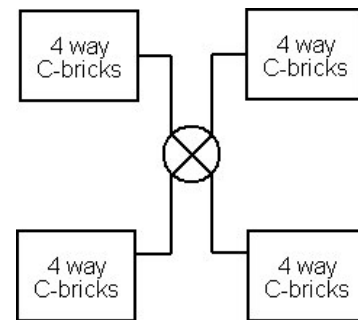
I C-bricks sono collegati tra loro in numero massimo di 4 attraverso un router a 6 od 8 porte NUMA e vengono chiamati in questa configurazione R-bricks.

Di questi router, 4 porte vengono usate per collegare i C-bricks, mentre le altre per connettersi ad altri R-bricks per un massimo di 8 moduli R-bricks da 16 CPU interconnessi tra di loro.

Attraverso un metarouter è anche possibile espandere il sistema fino ad un massimo di 512 CPU unendo 4 sistemi ad ipercubo assieme.

¹⁸ NUMA: Non Uniform Memory Access

¹⁹ UMA: Uniform Memory Access

**Figura 5 – C-bricks****Figura 6 – R-bricks**

I vantaggi di queste architetture sono le prestazioni, ma anche il costo più contenuto rispetto a supercomputer tipo i Cray.

Tutti i componenti utilizzati sono studiati appositamente per l'HPC e la comunicazione CPU-CPU e CPU-memoria sfrutta reti di interconnessione dedicate ad alta velocità.

L'HW delle macchine è inoltre ridondante e fault-tolerance, in questo modo ogni nodo del supercomputer è indipendente da un altro e permette il funzionamento della macchina anche se una sola CPU rimane funzionante.

I produttori di questi sistemi inoltre, non curano solo l'HW, ma dotano i loro sistemi di software di gestione della macchina semplificato e forniscono soluzioni chiavi in mano compresi di assistenza e montaggio.

Il rovescio della medaglia è dato dai costi ancora molto elevati, sia per quanto riguarda l'acquisto iniziale che per quanto riguarda la manutenzione, in quanto tali sistemi necessitano di ambienti a loro dedicati con temperatura, polvere ed umidità controllate.

La velocità del mercato nel fornire soluzioni sempre più performanti rende inoltre anche questi “mostri macinanumeri” obsoleti in poco tempo.

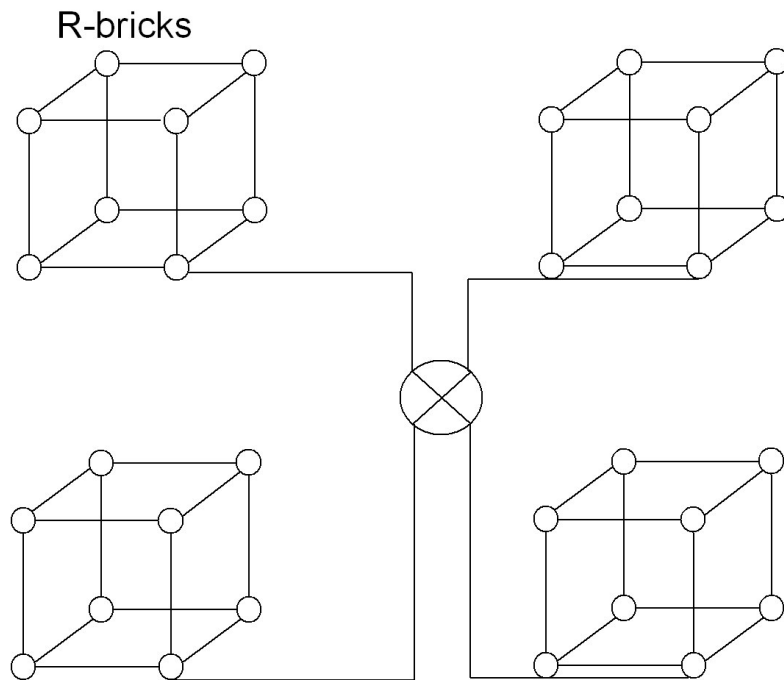


Figura 7 – configurazione a 512 CPU connettendo 4 R-brick da 128 CPU l'uno ad ipercubo

1.1.3 CLUSTER

I cluster sono macchine distribuite basate su workstation o PC, il loro vantaggio è dato dall'economicità dei componenti utilizzati per la loro costruzione dovuto sia all'utilizzo di materiali non specificatamente progettati per l'utilizzo di macchine per il calcolo parallelo sia per la reperibilità sul mercato della componentistica.

La tendenza di mercato è quella di sostituire o costruire i sistemi MPPS con tale tecnologia dato che il rapporto prezzo/prestazioni risulta molto competitivo.

I cluster sono modelli di calcolatori a memoria distribuita, in cui ogni nodo dello stesso, che in questo caso è costituito da un computer con un proprio HW ed un proprio sistema operativo, è collegato agli altri mediante comuni interfacce di rete, siano esse delle gigabit ethernet, myrinet, dolphin o fast ethernet e mediante una qualsiasi tipologia di rete.

Il sistema è visto dall'esterno come un unico computer grazie a tecnologie sia HW che SW.

Anche se i sistemi proposti dai grandi produttori sono basati su HW omogeneo e su soluzioni di interconnessione dei nodi ad hoc, nulla vieta di costruire il proprio cluster all'insegna del risparmio, magari utilizzando vecchi computer.

La flessibilità di questa soluzione è molto elevata, in quanto permette l'utilizzo di macchine eterogenee basate ad esempio su architetture e con sistemi operativi differenti nello stesso cluster; addirittura è possibile utilizzare reti con velocità diverse.

I cluster maggiormente utilizzati sono:

- 1- Beowulf
- 2- Mosix o openMosix.

L'utilizzo dei primi avviene prettamente attraverso l'uso di interfacce di message passing, e la virtualizzazione del sistema avviene grazie ad un layer software, che consiste in delle API per la programmazione del message passing e la rete può essere di qualsiasi tipo come discusso precedentemente.

Il secondo tipo di cluster, disponibile esclusivamente su piattaforma Linux, ad eccezione di una versione di Mosix per FreeBSD, introduce vincoli più restrittivi come l'utilizzo all'interno del cluster di architetture di macchine dello stesso tipo.

Introduce però un vantaggio non indifferente, che è quello di vedere la macchina come un unico computer multiprocessore, o SSI²⁰ migrando i processi automaticamente sul nodo più scarico.

Nel prosieguo ci occuperemo di un cluster ibrido Beowulf/openMosix con nodi SMP a 2 vie (2 CPU).

²⁰ SSI: Single System Image

Il vantaggio nell'utilizzare questo tipo di soluzioni per HPC risiede nel costo molto contenuto per l'acquisto iniziale dei componenti (almeno per alcune tipologie di essi), a scapito di una difficoltà maggiore nel creare inizialmente una soluzione funzionante, anche se molti rivenditori, tra i quali IBM, HP e Compaq forniscono le loro soluzioni di cluster già configurate e con una semplicità di gestione simile a quella dei sistemi MPPS.

1.1.4 GRID

Non possiamo infine escludere i GRID, che più che un'architettura parallela, rappresenta un modello di calcolo distribuito su più sistemi eterogenei (SMP, MPPS, PC, workstation, cluster) interconnessi tramite reti di tipo eterogeneo, sia in LAN che in WAN; in questi casi si parla di DHPC²¹.

L'utilizzo di tali sistemi è indicato per quelle applicazioni che richiedono:

- 1- una potenza di calcolo elevata;
- 2- un quantitativo di memoria elevato;
- 3- una capacità di memorizzazione elevate.

Questi sistemi sono concettualmente simili ai cluster, tuttavia per la loro peculiarità di essere distribuiti su reti geografiche non sempre dedicate, necessitano per il loro funzionamento di una più accurata gestione della sicurezza, quindi ogni sistema dovrà autenticarsi in modo "fidato" per poter avere accesso a determinate risorse sia di calcolo che di archiviazione.

²¹ DHPC: Distributed High Performance Computing

La scrittura di applicazioni per il GRID deve tenere conto della non necessaria località delle risorse e della eterogeneità delle stesse; inoltre questi algoritmi devono:

- 1- utilizzare tecniche per il fault-tolerance, in quanto da un momento all'altro un nodo può cadere in maniera inaspettata e l'applicazione deve ripartire senza problemi escludendo i nodi irraggiungibili;
- 2- ripartire il carico del lavoro, essendo possibile che dei nodi siano nettamente più veloci di altri, cosa che in genere in un cluster non succede.

La gestione di questi sistemi, viene semplificata attraverso l'utilizzo di alcuni tool liberamente utilizzabili come ad esempio CONDOR, UNICORE e Globus.

Questi software forniscono servizi per la gestione del GRID quali:

- 1- autenticazione degli accessi;
- 2- allocazione delle risorse;
- 3- gestione e monitoraggio dei job;
- 4- fault-tolerance;
- 5- monitoraggio dello stato del GRID;
- 6- accesso alle risorse remote (storage);
- 7- API²² standard per la gestione e l'accesso al sistema.

Per le comunicazioni e l'autenticazione, si utilizzano protocolli standard e consolidati, come lo standard X.509 per i certificati a chiave asimmetrica, più algoritmi di criptazione a chiave simmetrica per la sicurezza ed integrità dei dati (tunnel SSL²³, SSH²⁴) e tunnel di comunicazione criptati tra nodi.

²² API: Application Program Interface

²³ SSL: Secure Socket Layer

²⁴ SSH: Secure SHell

1.2 SMP Intel Vs SMP AMD²⁵

La diffusione dei componenti per calcolatori e il miglioramento della qualità produttiva degli stessi, ha reso possibile la creazione e la vendita di sistemi multiprocessore economici basati su architettura iA32 e destinati al grande pubblico ed ai professionisti.

Questi sistemi, data la loro economicità sono ottimi per la realizzazione di cluster, unendo così i vantaggi dei cluster con quelli relativi ad un bus a memoria condivisa.

Le due architetture analizzate di seguito riguardano i sistemi biprocessori Intel ed AMD.

I processori basati sull'architettura Intel X86 a partire dal PentiumPRO, sono dotati di logica di controllo per sistemi biprocessore all'interno del chip contenete il core della CPU.

Questo permette di realizzare facilmente macchine biprocessore con chipset comuni o comunque relativamente semplici e quindi economici.

Vediamo quindi in cosa si differenzia l'implementazione dell'una rispetto all'altra piattaforma partendo da AMD.

I processori della famiglia Athlon MP vengono certificati da AMD per l'utilizzo in sistemi multiprocessore, di cui la stessa AMD ne sviluppa un chipset per sistemi due CPU: l'AMD 760, di cui considereremo solo il north bridge, in quanto responsabile del trasferimento dei dati dalla memoria alla CPU e viceversa.

A differenza dei tradizionali sistemi SMP, in configurazione biprocessore, l'Athlon dispone di un bus separato per ogni CPU da 2,6 GB/s l'uno.

²⁵ Intel© e AMD© sono marchi registrati

Questi bus sono messi in comunicazione attraverso il north bridge, che garantisce l'accesso alla memoria attraverso un terzo bus sempre alla stessa velocità.

Come si può notare anche dalla figura 6, in realtà l'accesso alla memoria viene gestito in mutua esclusione tra i due bus per le CPU e quello PCI²⁶ a 64 bit, rendendo di fatto il doppio bus equivalente ad un bus condiviso. La configurazione Intel invece risulta la classica implementazione di un biprocessore con bus per l'accesso alla memoria condiviso.

In questo caso, essendo gran parte della logica di controllo integrata nei microprocessori, sono gli stessi che gestiscono l'accesso esclusivo al bus, che attraverso il north bridge, permette l'accesso alla memoria in mutua esclusione con un bus generico a cui sono attaccate sia le periferiche che il bus PCI.

In AMD le periferiche non PCI sono collegate al bus PCI tramite south bridge, mentre in Intel vi è un bus dedicato che gestisce la comunicazione anche del bus PCI, sempre a 64 bit.

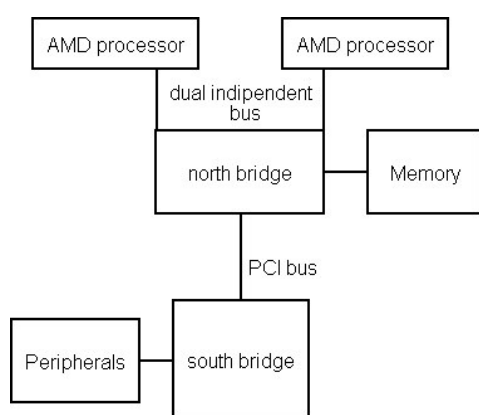


Figura 7 – SMP versione Intel

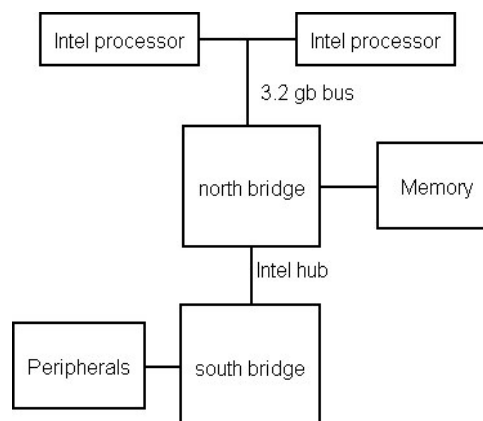


Figura 7– SMP versione AMD

²⁶ Peripheral Component Interface

1.3 Beowulf

Su questa classe di cluster non c'è molto da dire, nascono da un progetto del Dipartimento della Difesa Americano per la costruzione di sistemi di calcolo ad alte prestazioni utilizzando comune HW.

Questi sistemi sono utilizzabili come un unico computer a patto di scrivere le applicazioni servendosi di particolari librerie, le più famose sono MPI²⁷ e PVM²⁸, ma ne esistono altre.

In particolare MPI è quella più utilizzata e che ha avuto un seguito maggiore, in quanto tutte le piattaforme Unix per uso HPC ne possiedono una implementazione particolare, oltre all'esistenza di diverse implementazioni gratuite disponibili in rete.

Queste librerie forniscono un'interfaccia standardizzata al MP²⁹ e oltre a ciò viene eseguito un demone che gestisce gli scambi di messaggi tra le varie parti del programma.

In realtà il meccanismo che sta alla base usa massicciamente i protocolli rexec³⁰ ed rsh³¹; ogni volta che si esegue un programma, in realtà questo sarà lanciato su tutti i nodi del cluster o su un prefissato numero di nodi.

Ogni programma lavorerà su una parte di dati e quando deve trasmettere, ricevere o collezionare i dati con gli altri programmi eseguiti sugli altri nodi, utilizzerà i protocolli di rete TCP³² o UDP³³ per dialogare con gli altri processi remoti analogamente come un'applicazione client/server.

²⁷ MPI: Message Passing Interface

²⁸ PVM: Parallel Virtual Machine

²⁹ MP: Message Passing

³⁰ rexec: remote execution

³¹ rsh: remote shell

³² TCP: Transmission Control Protocol

³³ UDP: User Datagram Protocol

Congiuntamente a queste librerie, è possibile aggiungere al cluster un software per l'esecuzione BATCH dei programmi sui diversi nodi, quali ad esempio PBS³⁴, oppure una patch per il kernel per eseguire la migrazione dei processi come BProc³⁵.

Per quel che riguarda l'HW, ve ne è una varietà eterogenea impiegabile, si va dai comuni PC, magari datati e non più idonei ad un uso come workstation a costosi sistemi venduti appositamente per uso in cluster Beowulf.

In contrasto all'eterogeneità di HW, I programmi (seppur scritti appositamente per questo tipo di cluster) gireranno senza problemi su qualsiasi sistema Beowulf senza modifiche, come invece avviene quando si passa da un'architettura di supercomputer ad un'altra per poterne sfruttare le peculiarità.

1.4 openMosix

openMosix nasce da un fork del progetto Mosix, sviluppato dal Prof. Barak e dal Dr. Moshe Bar dell'Università Israeliana di Tel Aviv. Leggi restrittive sull'importazione di supercomputer in Israele portarono allo sviluppo di Mosix su PDP101 e ne fu fatto un porting per piattaforma Linux e BSD.

Nel 1999 il supporto di Mosix per Linux divenne stabile ma, per problemi di licenza (openMosix è rilasciato sotto GPL, Mosix ha una licenza proprietaria più restrittiva) venne creato un nuovo progetto dal coautore di Mosix Moshe Bar: il suo nome era openMosix.

³⁴ PBS: Portable Batch Sistem

³⁵ BProc: Beowulf distribuitied Process space

Attualmente, viene sviluppato sia dalla comunità opensource sia dalla società Qlusters Inc. fondata dallo stesso Dr. Moshe Bar che ne è il CTO³⁶.

openMosix rientra in quella serie di cluster di tipo SSI; questi cluster consentono di vedere i nodi del cluster come se fossero un'unica macchina.

In pratica, l'esecuzione di un programma lanciato da un qualsiasi nodo può essere eseguito su qualsiasi altro nodo del sistema, esattamente come succede con una macchina SMP, senza che l'utente si possa effettivamente rendere conto di questo.

Vi sono state sul mercato varie implementazioni di sistemi di questo tipo, ma una peculiarità di openMosix è quella di non essere un'implementazione completa di un sistema operativo distribuito, ma di appoggiarsi sul kernel di Linux, in modo che tutti i programmi già esistenti per questo sistema continuino a girare e possano persino trarre beneficio dalla nuova architettura.

D'altro canto in questo modo, lo stesso openMosix si potrà avvantaggiare di un sistema già consolidato sul mercato e basato su un codice molto veloce e dotato di uno stack TCP/IP³⁷ efficiente e robusto.

1.4.1 openMosix internals

openMosix consiste in due parti:

- 1- un PPM³⁸;
- 2- un insieme di algoritmi adattivi per la condivisione delle risorse.

³⁶ CTO: Chief Technical Officer

³⁷ IP: Internet Protocol

³⁸ PPM: Preemptive Process Migration

Il primo consente di migrare fisicamente un processo da un nodo ad un altro; il secondo colleziona delle statistiche sulle risorse dei vari nodi.

Ogni processo eseguito possiede un UHN³⁹, che corrisponde al nodo in cui il processo viene creato e generalmente corrisponde allo stesso nodo a cui l'utente possessore del processo ha effettuato la login al sistema.

Inizialmente, il processo viene eseguito nel suo UHN, ma se vengono a meno i requisiti di memoria o di cicli macchina necessari, migrerà su un nodo più prestante, in modo da mantenere le risorse distribuite nel modo più uniforme possibile sui nodi.

Questo può avvenire più volte durante la vita dello stesso processo.

I principali algoritmi per la condivisione di risorse sono basati su:

- 1- disponibilità di memoria;
- 2- disponibilità di CPU.

Il meccanismo di migrazione fa in modo di minimizzare lo swap migrando processi da un nodo in cui ve ne sono in eccesso dovuti al diminuire della memoria ed allo stesso tempo cerca il nodo più scarico dal punto di vista computazionale.

Attraverso un modello derivato dall'economia, vengono collezionati a determinate scadenze e ad ogni system-call delle statistiche sull'utilizzo di memoria, cpu, rete su ogni nodo.

A questi valori viene assegnato un costo; il nodo con il costo inferiore è quello più indicato ad eseguire un processo.

Per rendere il processo il più leggero possibile, ogni nodo invia i dati ad un subset arbitrario di altri nodi anziché a tutti e per la trasmissione dei dati viene utilizzato UDP come protocollo di trasporto, inoltre, se la migrazione del processo risulta troppo onerosa per via di un eccessivo

³⁹ UHN: Unique Home Node

numero di dati da trasferire in relazione alla velocità della rete, il processo non migrerà.

Il processo migrato possiede due parti: una residente su un UHN e corrispondente alla parte di programma in kernel mode, e l'altra contenete la parte del processo eseguita in user space e residente su un altro nodo, la prima chiamata deputy, la seconda remote.

Il remote contiene:

- 1- la memoria del programma;
- 2- lo stack;
- 3- l'immagine del processore.

Il deputy contiene la parte di programma eseguito in kernel mode.

Ogni volta che il processo eseguirà una system-call, il remote contatterà il deputy che la eseguirà, al termine la deputy invierà il risultato alla remote, che continuerà a lavorare.

Il remote potrà migrare più volte su diversi nodi, quando il nodo che esegue il remote non soddisferà più i vincoli richiesti, l'esecuzione dell'intero processo sarà riportata sull'UHN per poi essere migrata su un altro nodo se necessario.

Per non appesantire troppo il processo di migrazione, solamente le dirty page vengono migrate al momento della localizzazione di un nodo migliore, per poi migrare le altre al momento di un page fault.

Il processo viene visto sempre dal nodo UHN come se si trattasse di un task locale e non nel nodo che realmente lo sta eseguendo.

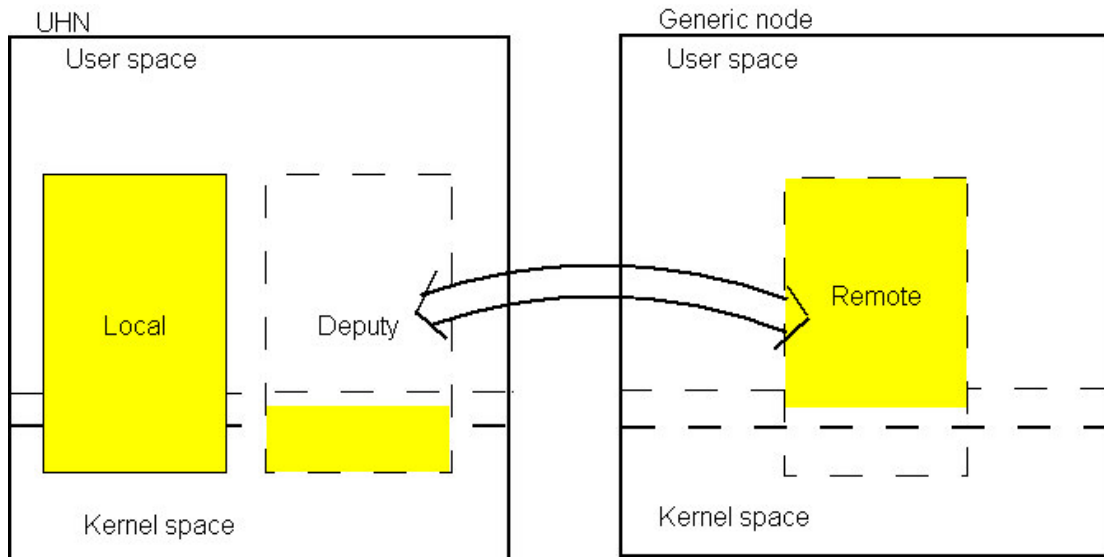


Figura 8 – Interazione tra Deputy e Remote

1.4.2 File system

Un'altra questione da risolvere riguarda la gestione del FS⁴⁰ all'interno di un cluster openMosix.

Quando un processo accede ad un file, e poi migra, deve continuare ad avere accesso al file originale che si trova sul UHN.

Vi sono due possibilità per risolvere il problema, la prima consiste nel fare in modo che tutti i nodi accedano a tutti i file degli altri, il secondo è far sì che openMosix intercetti tutte le chiamate di I/O e le inoltri al UHN.

Oltre a gestire la seconda ipotesi, openMosix implementa anche un FS studiato appositamente per garantire la prima, tale FS si chiama OMFS⁴¹.

Questo FS garantisce la coerenza delle immagini dei FS su tutti i nodi.

La migrazione dei processi avviene privilegiando l'uso del FS su determinati nodi utilizzando OMFS.

⁴⁰ FS: File System

⁴¹ OMFS: OpenMosix File System

Se si scrive un file attraverso quest'ultimo che risiede su un nodo *n*, tale nodo sarà un candidato privilegiato per l'esecuzione del remote.

1.4.3 Gestione del cluster e migrazione forzata

Per gestire in modo trasparente il cluster e garantirne al contempo la compatibilità con le applicazioni esistenti, il controllo di openMosix avviene attraverso una interfaccia standard al FS virtuale */proc*, analogamente a tutte le componenti del kernel di Linux e di Unix in generale.

Questo approccio consente un tuning del sistema, monitorandone le risorse, i processi migrati, quelli in esecuzione e forzando la migrazione degli stessi e viceversa allocandoli su un determinato nodo, in modalità runtime.

L'interfaccia è raggiungibile tramite */proc/openMosix* e */proc/openMosix/admin*.

1.5 Beowulf Vs Mosix

Lo svantaggio principale di un cluster di tipo Mosix risiede nell'overhead incorrente ad ogni system-call di un processo migrato, questo è tanto maggiore quanto la latenza della rete è elevata.

Il problema viene in parte risolto preallocando le risorse anche attraverso l'utilizzo di librerie quali MPI, come avviene ad esempio su cluster di classe Beowulf.

In questo modo tutte le risorse necessarie inizialmente alla parte di programma in esecuzione su un nodo sono all'istante disponibili su di esso, in quanto il programma viene fisicamente lanciato su più nodi contemporaneamente.

Un altro svantaggio consiste nella granularità del cluster, rappresentata dai processi: allo stato attuale, la migrazione dei thread non è possibile, ed in generale, qualsiasi processo o job che usa memoria condivisa con altri processi/job come nel caso di thread non sarà migrato se non in blocco assieme a tutti i processi che usano la stessa porzione di memoria. Su Beowulf viene lanciato lo stesso programma su ogni macchina ma opererà su dati differenti e non avrà memoria condivisa con altri processi. Al momento si sta lavorando ad una versione che riesca a migrare anche i thread.

A fronte di questi svantaggi, vi è il non trascurabile vantaggio di non dover riscrivere le proprie applicazioni o di utilizzare particolari librerie per parallelizzare i propri programmi, come avviene per Beowulf.

La gestione del sistema in generale risulta comunque semplificata, permettendo ad un normale amministratore di rete di gestire il cluster sia in un caso che nell'altro senza che necessiti di una particolare preparazione.

A differenza di Beowulf, openMosix gestisce il fault-tolerance: ogni nodo di Mosix è indipendente, non esistendo un concetto di master/slave e quando un nodo cade gli altri nodi non riescono più a comunicare con esso e lo escludono dal resto del sistema.

In Beowulf questo controllo non c'è utilizzando le librerie standard ed occorre implementarlo con altri programmi o sistemi; in caso contrario, se il programma cerca di lanciare un processo su un nodo off-line, terminerà

dando un errore dopo lo scadere di un timeout che dipende dall'implementazione dello stack TCP/IP.

Il bilanciamento del carico, inoltre viene controllato esclusivamente dal programma nel secondo tipo di cluster, mentre nel primo è il sistema che “decide” dove eseguire che cosa in base al carico dei nodi.

<i>Caratteristica</i>	<i>BEOWULF</i>	<i>OpenMOSIX</i>
FS distribuito	NO	SI
Migrazione processi in esecuzione	Solo con software aggiuntivo	SI
Esecuzione BATCH su più nodi	Solo con software aggiuntivo	Limitatamente alle risorse di rete
Fault tollerance	Solo con software aggiuntivo	SI
Assegnazione statica delle risorse	SI, ogni porzione di programma viene eseguita su un processore diverso	NO
Overhead	Nelle trasmissioni	Trasmissioni, system call e migrazione
Controllo selettivo dei nodi	SI	In parte

Tabella 1 Confronto tra cluster openMosix e Beowul

Capitolo 2

Progettazione di un cluster

Vediamo in questo capitolo la parte inerente la progettazione di un sistema per il calcolo parallelo basato su cluster ed i suoi vantaggi rispetto ad una soluzione proprietaria basata su supercomputer.

L'implementazione viene analizzata nel dettaglio, assieme alle motivazioni e le problematiche che ha comportato e che hanno portato a questa scelta, i vantaggi della soluzione rispetto ad altri tipi di cluster e/o macchine parallele.

2.1 Progettazione

La tipologia di HW adottato, consiste in un cluster ibrido di tipo openMosix/Beowulf.

La scelta è stata fatta per poter usufruire delle caratteristiche migliori dei due sistemi, in questo modo il cluster può funzionare sia come Beowulf, sia come SSI, sia ibrido SSI/Beowulf.

La configurazione del cluster è stata effettuata su SO Linux, inizialmente su Slackware 8.0, con una configurazione BSD-like, poi nella versione definitiva su una Debian 3.0 con una configurazione alla sysV, scelta dettata dalla migliore gestione degli aggiornamenti di quest'ultima e su una migliore manutenzione del sistema installato.

Entrambe le prove sono state effettuate con un kernel v2.4.

La soluzione finale consiste in un server contenente i servizi di rete e 4 client diskless che accedono al server.

I servizi di rete forniti dal server sono:

- 1- NFS per il file system condiviso;
- 2- DHCP per assegnare indirizzi IP;
- 3- TFTP per il boot.

Il server ha la funzione di nodo frontend per l'interfacciamento al cluster e di nodo master per Beowulf, assegna indirizzi IP ai client, fornisce l'immagine di boot agli stessi e fornisce anche il file system di root (/).

I client sono i nodi slave del cluster Beowulf ed inoltre sono anche nodi openMosix; eseguono le seguenti operazioni di inizializzazione:

- 1- ottengono un indirizzo IP dal server;
- 2- ottengono l'immagine del kernel di boot dal server;
- 3- montano il file system remoto via NFS (sul server) come root (/);
- 4- caricano il resto del sistema operativo ed i servizi dal FS del punto 3.

2.2 Linux

La principale motivazione della scelta di Linux come sistema operativo del cluster è di tipo economico, infatti Linux è completamente gratuito.

Un altro aspetto da non sottovalutare è la disponibilità del codice sorgente; questo implica poter apportare delle modifiche su misura delle esigenze che possono verificarsi in base all'utilizzo che se ne vuole fare.

Quest'ultimo motivo ha spinto molte grandi multinazionali a sviluppare software per il clustering o a presentare soluzioni personalizzate dello stesso sistema operativo, disponibile per la quasi totalità delle piattaforme presenti sul mercato.

Inoltre la licenza su cui si basa il codice, la GPL¹, favorisce la modifica del codice a proprio piacimento senza imposizioni restrittive o limitazioni di alcun genere.

L'utilizzo di sistemi operativi proprietari, per contro, implicherebbe l'utilizzo di costoso HW dedicato ed anche rivolgersi a sempre costose soluzioni SW proposte dal rivenditore, non essendo consentito l'utilizzo del codice.

Il modello comunitario di sviluppo di Linux, consente inoltre di possedere un sistema sempre aggiornato con conseguente correzione dei bug riscontrati in tempi brevissimi, non dovendo aspettare il rilascio di patch da parte di alcuna società.

Oltre ad essere una reale alternativa ai sistemi Unix, Linux risulta anche una comoda alternativa ai sistemi Microsoft®, data la sua disponibilità di software in quantità ed in qualità sia per uso personale, per office-automation e in qualsiasi ambito lavorativo.

Un'altra caratteristica di Linux ed in generale delle distribuzioni basate su di esso, è la loro conformità agli standard di mercato, diversamente da quanto accade per i sistemi proprietari legati a protocolli di comunicazione, anch'essi proprietari, dei quali vengono gelosamente custodite le specifiche.

¹ GPL: General Public License

2.3 Hardware adottato

La scelta dell'HW è ricaduta su 4 macchine biprocessore identiche più una macchina che funge da server per l'accesso al cluster già disponibile.

Per motivi di gestione centralizzata si è optato per una soluzione master/slave.

Sono state utilizzate 4 macchine dual AthlonMP 1800+ che sono adibite al calcolo sia Beowulf che openMosix ed una macchina dual PIII 1000 per fornire servizi di file server centralizzato e master per l'utilizzo del cluster Beowulf.

Inoltre, l'utilizzo di openMosix permette, congiuntamente ad MPI di ridistribuire il calcolo sui nodi più scarichi in caso di bilanciamento dei dati non ottimale durante l'esecuzione.

Per semplificarne la manutenzione si è inoltre optato per delle macchine diskless, con il sistema operativo risiedente sulla macchina PIII che contiene anche i servizi di rete e di frontend al cluster vero e proprio.

Questa soluzione ha inoltre il vantaggio di essere la più economica possibile.

Le macchine SMP permettono inoltre di utilizzare un modello ibrido di programmazione multiprocesso/multithread per minimizzare i tempi di accesso ai dati tra processori sullo stesso segmento di memoria.

La scelta delle macchine Athlon è stata preferita in base al miglior rapporto prezzo/prestazione esistente al momento dell'acquisto, in quanto l' Athlon è risultata la piattaforma più performante nell'utilizzo di operazioni in floating point e la piattaforma biprocessore più economica.

Come bus di interconnessione sono state scelte due reti fast ethernet connesse a due switch 10/100 base TX grazie al loro costo contenuto.

La prima rete viene utilizzata esclusivamente per la lettura dei nodi sul FS attraverso NFS², mentre la seconda viene utilizzata per lo scambio di messaggi e la migrazione dei processi.

Ogni nodo del cluster è equipaggiato con una scheda di rete 3com ed una Intel pro1000; la macchina master possiede due schede di rete 3com più una terza per l'interfacciamento con l'esterno del cluster.

La scelta di una rete a soli 100 Mbit non influenza in modo eccessivo le performance del cluster, in quanto l'utilizzo dello stesso è stato previsto per algoritmi altamente parallelizzabili con una mole di calcoli molto elevata rispetto ai dati stessi.

La spesa di una rete Gigabit Ethernet non è invece giustificata per un numero così basso di nodi, che permette ancora delle prestazioni accettabili a 100 Mbit.

CPU	Dual PIII 1Ghz
Memoria	2 x 512 Mb DIMM pc133 ECC ³ registered
Scheda madre	Supermicro Super 370DE6-G
Schede di rete	3 X 3Com 590c
Sistema operativo	GNU Debian Linux i386 3.0r0
Kernel	2.4.20
Controller dischi aggiuntivo	Promise RAID ⁴ 0-1 Fasttrack 100
Dischi	4 x IBM Deskstar 60 Gb 1 x Maxtor Diamondmax 5400 20 Gb
Partizioni	/home su 2 IBM Deskstar in RAID 1 su ctrl Promise /etc su IBM Deskstar in RAID 1 su ctrl promise /tmp in RAID 0 su 2 IBM Deskstar /var su RAID 0 / su RAID 1

Tabella 2 - Mappa dell'HW del master

² NFS: Network File System

³ ECC: Error Code Correction

⁴ RAID: Rendundant Array of Indipendent Disk

CPU	Dual Athlon MP 1800+ 1533 Mhz
Memoria	2 x 512 Mb DIMM DDR pc266
Scheda madre	Asus A7M266-D
Schede di rete FS	3Com 590c
Scheda di rete dati	Inter pro1000
Dischi	No

Tabella 3 - Mappa dell'HW dei nodi

2.4 Installazione della rete

L'unico modo per accedere al cluster è attraverso la macchina master (PIII), tramite console o mediante protocollo SSH via rete pubblica.

La prima scheda di rete connessa al master offre i servizi di boot e di NFS per i nodi slave, la seconda viene usata come bus per il MP o la migrazione.

Analogamente, sui nodi per il calcolo la seconda scheda di rete (Intel) viene usata come bus per il MP e la prima per il boot ed il FS (File System).

La rete per il caricamento del FS è 10.10.10.0/24, quella per il MP è 100.100.100.0/24; il master ha rispettivamente l'indirizzo 11 e 101, agli altri nodi sono riservati i successivi 4 indirizzi.

Gli hostname sono risolti per mezzo del file */etc/hosts* e sono assegnati seguendo il nome beoxfs.beowulf-fs sulla rete per i servizi di NFS, con x il numero progressivo del client, e come beox.beowulf-dati per la rete per il MP.

Ci occuperemo ora della configurazione del server/master in dettaglio vedendo come installare tutti i servizi di base necessari al buon funzionamento del nostro cluster per poi entrare nel dettaglio dei client.

La rete esterna viene utilizzata per accedere al cluster attraverso il master ed utilizza la connessione fornita dalla LAN⁵ universitaria; i parametri sono in figura 9.

rete:	137.204.72.0
dns:	137.204.72.1
gw:	137.204.72.254
ip:	137.204.72.45
netmask:	255.255.255.0
hostname:	maximus
domain:	csr.unibo.it

Figura 9 – Configurazione rete LAN

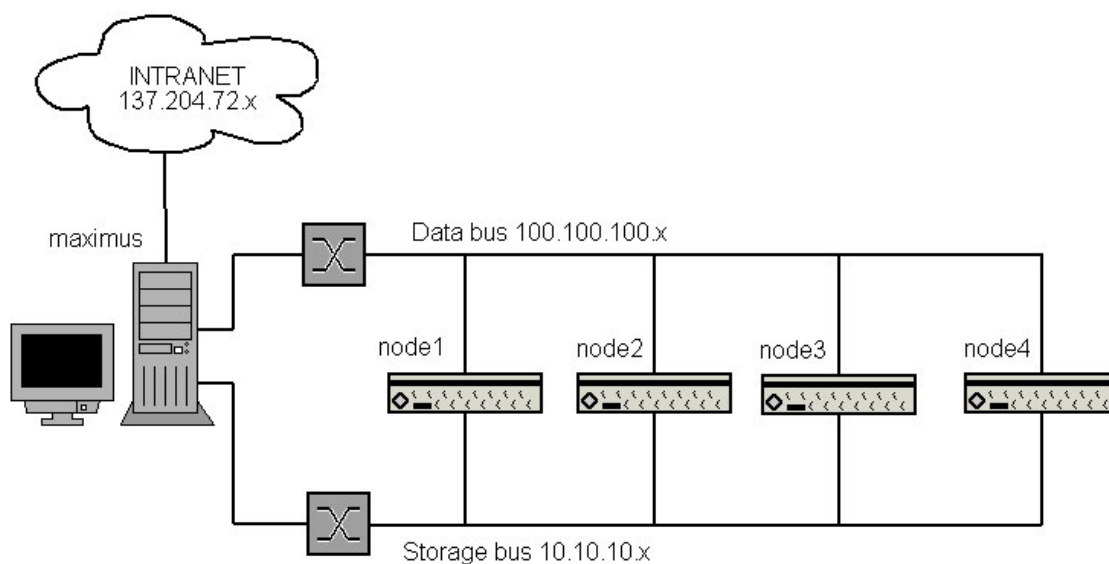


Figura 10 – Schema del cluster

⁵ LAN: Local Area Network

2.5 Configurazione del server DHCP

Per consentire ai client di usufruire dei servizi sopra richiesti, occorre assegnare ad ognuno di essi, un indirizzo IP valido.

Per ogni scheda di rete di ogni client, viene assegnato un indirizzo IP fisso controllando il MAC⁶ della scheda stessa, in totale vengono assegnati due indirizzi, il primo per la rete relativa al FS ed il secondo per la rete dati.

Il servizio che si occupa di questo, è il DHCP⁷, che oltre a ciò indica anche quale sia il *next-server*, cioè l'indirizzo del server tftp che nel caso corrisponde allo stesso del DHCP ed al nome del file da caricare come kernel, essendo le nostre macchine diskless.

Occorre quindi configurare due reti distinte gestite dallo stesso server e modificare il file `/etc/dhcpd.conf` secondo le esigenze.

Le opzioni generali sono in figura 11.

```
ddns-update-style inerim;  
default-lease-time 600;  
max-lease-time 7200;  
authoritative;
```

Figura 11 – Opzioni base DHCP

Queste opzioni non sono interessanti dal punto di vista della progettazione del cluster, servono solo a specificare i timeout della richiesta del client ed imposta il DHCP come server autoritativo per la risoluzione dei nomi.

Per ogni rete occorre inserire poi la direttiva `subnet` per creare una sezione ad essa riservata:

⁶ MAC: Medium Access Control

⁷ DHCP: Dinamic Host Configuration Protocol

```
subnet 10.10.10.0 netmask 255.255.255.0
{
    ...
    ...
    ...
}
```

Figura 12 – Schema configurazione rete

all'interno della sezione si possono inserire delle direttive specifiche per il tipo di rete non specificate nelle direttive sopra, quelle che possono interessare sono:

```
use-host-decl-names on;
option subnet-mask 255.255.255.0;
option broadcast-address 10.10.10.255;
range 10.10.10.11 10.10.10.253;
option domain-name "Beowulf-fs";
```

Figura 13 – Opzioni specifiche per una rete

Analizziamo queste righe nello specifico:

- 1- prende l'hostname della macchina dal suo indirizzo completo;
- 2- specificata la maschera di rete;
- 3- indica l'indirizzo di broadcast;
- 4- range di indirizzi utilizzabili nell'assegnazione degli IP ed il nome del dominio assegnato alla rete.

Nel cluster sarà inoltre presente un'altra sezione relativa al traffico dei dati sulla rete 100.100.100.0/24 con nome di dominio *beowulf-dati*.

Arriviamo quindi alla dichiarazione dei dati da assegnare alla macchina che ne ha fatto richiesta.

La direttiva, all'interno della sezione precedente, è *host* che crea una sottosezione e deve essere specificata per ogni indirizzo da assegnare.

```
host beo1fs
{
    hardware ethernet 00:0a:30:10:11:6b;
    fixed-address 10.10.10.11;
    next-server 10.10.10.10;
    filename "vmlinuz-athlon";
    option root-path "/";
}
```

Figura 14 – Configurazione di un Host

Per semplicità è stato preso ad esempio un nodo reale della macchina configurata: *beo1fs* è l'hostname della macchina con i requisiti inseriti tra parentesi {}:

- 1- il numero a 48 bit inserito dopo *hardware ethernet* va sostituito con l'indirizzo MAC della scheda di rete della macchina;
- 2- il valore dopo *fixed-address* va sostituito con l'IP da assegnare alla macchina;
- 3- *nextserver* è l'indirizzo del server tftp;
- 4- *filename* indica il percorso relativo rispetto alla root del server tftp, del file da spedire al client dopo l'assegnazione dell'IP;
- 5- *root-path* è la root del server tftp, deve essere la stessa specificata nel file */etc/inetd.conf*, ma si può anche omettere.

La sezione corrispondente all'host per la rete 100.100.100.0/24 non necessita delle parti relative al server tftp, avendo la macchina già fatto il boot durante l'assegnazione degli indirizzi della classe 10.10.10.0/24, quindi le uniche direttive da specificare sono:

- 1- *hardware ethernet*;
- 2- *fixed-address*.

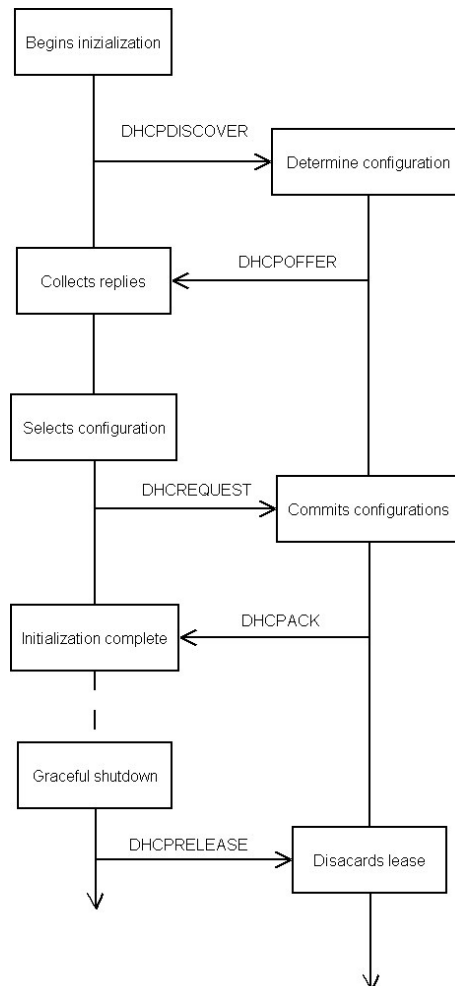


Figura 15 – Schema di funzionamento DHCP

2.6 File System

La root del server è condivisa per intero con i client diskless e le differenze di configurazione sono possibili grazie all'utilizzo di ClusterNFS⁸, un server NFS modificato che permette di fornire file

⁸ <http://clusternfs.sourceforge.net/>

diversi su richieste dello stesso nome del file, discriminandolo dal nome dell'host.

Inserendo il suffisso *\$\$discriminante\$\$* dopo il nome del file, dove discriminante è uno dei seguenti:

\$\$IP=IP_MACCHINA\$\$

\$\$USER=NOME_UTENTE\$\$

\$\$GROUP=NOME_GRUPPO\$\$

Se sono presenti più file di questo tipo, l'ordine di risoluzione è quello proposto nell'elencazione, se non vi sono file di quel tipo viene fornito il file senza suffisso, il file viene visto dal client come *nomefile*.

ClusterNFS è trasparente ai client, che utilizzano un semplice client NFSv2 o v3, mentre sul server va installata una versione particolare del demone in user space, senza necessità di applicare patch al kernel.

Il demone *nfsd* verrà installato nella directory */usr/sbin* e per abilitare la risoluzione dei nomi alternativa va richiamato dai file di configurazione come:

rpc.nfsd -T

Nel nostro caso il file da modificare è */etc/init.d/nfs-kernel-server*.

Nel caso lo si voglia far partire a mano, i comandi sono:

rpc.portmap

rpc.nfsd -T

rpc.mountd

Il primo abilita il demone *portmap*, che permette di gestire servizi RPC mappando, ad ogni specifico servizio una porta di rete specifica; il secondo abilita il server NFS lanciando diversi thread in kernel space; il terzo permette di soddisfare le richieste di mount dei client, controllando

le directory esportate ed i permessi sull'esportazione della stessa da parte del client che ne ha fatto richiesta.

Occorre anche editare il file `/etc/exports`, che contiene l'elenco delle directory esportabili sui client; la semplice riga di comando utilizzata consente di montare sui client l'intero FS:

```
/      *.beowulf-fs(rw,no_root_squash)
```

Come si può facilmente intuire, l'accesso al disco è consentito solamente alle macchine sulla rete dedicata al FS in lettura e scrittura, mentre l'opzione `no_root_squash`, indica al server di non rimappare sull'utente non privilegiato `nobody` le richieste fatte da root.

Essendo il boot della macchina eseguito sempre come tale utente (root), l'esecuzione da parte di `nobody` creerebbe dei malfunzionamenti, oltre a non consentire operazioni basilari come l'avvio dei servizi necessari alla rete e a montare i FS.

```
tar xzf clusternfsxx.tar.gz
cd clusternfsxx
./configure
make
make install
```

Figura 16 –Installazione ClusterNFS

Per poter montare la root via FS remoto, occorre abilitare un altro servizio che permetta di scaricare l'immagine del kernel al boot, questo servizio è `tftp`⁹.

In pratica, si tratta di un servizio ftp molto ridotto, senza la parte di autenticazione degli utenti e studiato apposta per trasferire file su macchine “sicure”, l'unica cosa che occorre per utilizzarlo è uno stack TCP/IP minimale con un indirizzo IP valido.

⁹ Tftp: trivial file transfer protocol

Per abilitare il servizio attraverso l'uso di `inetd` occorre aggiungere la seguente riga al file `/etc/inetd.conf`:

```
tftp dgram udp wait root /usr/sbin/tcpd in.tftpd -s /
```

I campi indicano, nell'ordine:

- 1- il nome di riferimento del servizio da abilitare;
- 2- il tipo di socket da utilizzare(datagramma, stream, raw, seqpacket);
- 3- il protocollo di rete usato;
- 4- se il protocollo è di tipo multithread (nowait, cioè vengono accettate connessioni di più client contemporaneamente) o di tipo singletread (wait);
- 5- l'utente con cui viene lanciato il demone;
- 6- il nome dell'eseguibile con il percorso completo;
- 7- i parametri dell'eseguibile.

Il quarto campo è applicato solo ai pacchetti di tipo datagramma, ma nel caso di `tftp`, che è un servizio multithread, l'opzione `wait` viene specificata ugualmente per evitare race condition sulle richieste.

Il sesto campo invece di essere il percorso del servizio che ci interessa è in questo caso `tcpd`, un demone che monitorizza le richieste del demone passatogli come parametro per consentire un controllo d'accesso più accurato.

Il metodo più utilizzato per garantire questo è basato su `tcpwrapper` di Wietse Venema, il parametro risulta così dato dal nome del server reale da eseguire e dai suoi parametri, in questo caso solo `-s` che indica la directory root virtuale del server, che equivale alla root reale nel nostro caso.

2.7 Differenze nel FS tra master e nodi

Ci sono alcuni file e directory che devono, per necessità amministrative, essere diverse su ogni nodo del client.

Per poterle differenziare occorre identificare tutte le directory contenenti questi file ed in alcuni casi i file stessi e crearne una copia distinta per ogni macchina.

Le directory sono:

/etc

/var

/tmp

In questo, ci viene in aiuto ClusterNFS, la sintassi utilizzata è quella con suffisso *\$\$IP=IP_NUMBER\$\$*.

2.8 Configurazione del kernel

Tutte le opzioni sopra elencate necessitano che il kernel sia compilato ad-hoc per supportare il protocollo NFS come servizio sul master e sui nodi come client, inoltre il kernel per i nodi deve avere abilitate le funzioni per i servizi di DHCP e root via NFS.

Partiamo dal kernel del master, dalle opzioni di compilazione occorre selezionare le seguenti voci:

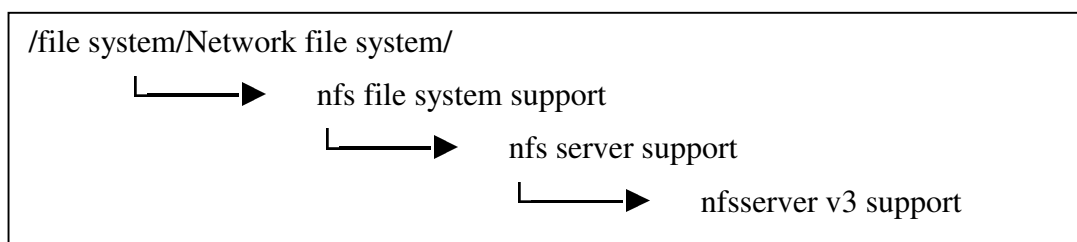


Figura 17 – Configurazione server NFS

Le opzioni del kernel dei nodi invece:

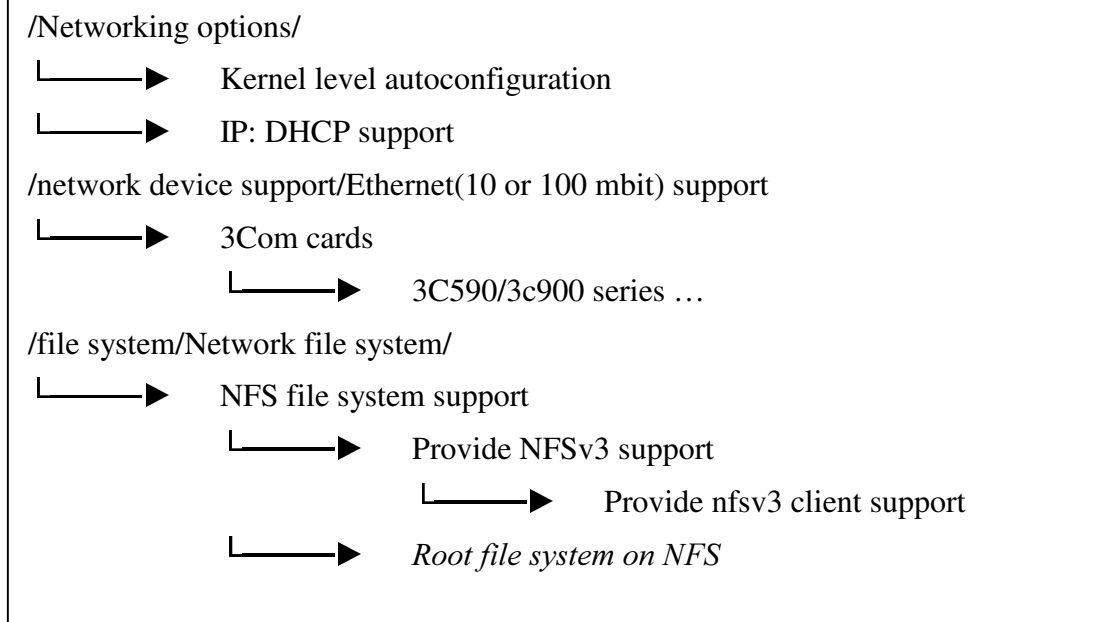


Figura 18 – Configurazione schede di rete

Le opzioni per abilitare openMosix (solo per i nodi):

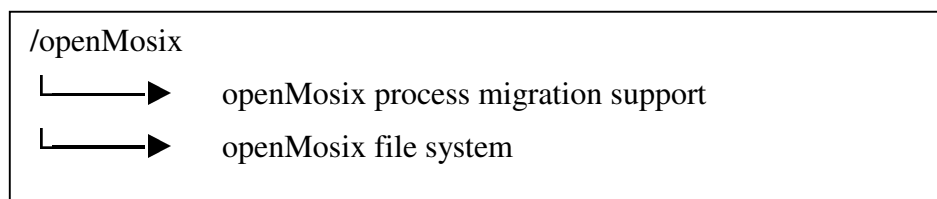


Figura 20 – Configurazione nodo openMosix

Una opzione da considerare, è la possibilità di avere una console sulla seriale di ogni macchina:

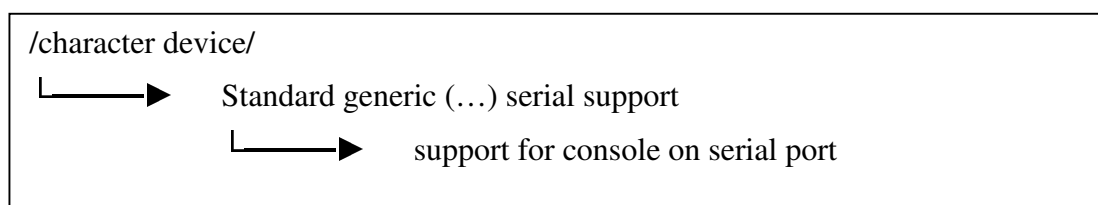


Figura 19 – Configurazione console via seriale

Oltre a questo occorre abilitarne l'uso da Linux inserendo la linea seguente nel file */etc/inittab*:

```
s1:12345:respawn:/sbin/agetty -L ttyS0 9600 vt100
```

che indica di aprire una console virtuale sulla prima porta seriale, con comunicazione a 9600 baud ed emulazione terminale vt100.

Il kernel dei client va compilato sul master e reso bootable attraverso l'utility *mknbi*¹⁰.

L'utilizzo è semplice, una volta compilato il kernel, si ottiene un secondo kernel che può essere eseguito via rete specificando di caricare il FS via NFS, ottenere l'IP via DHCP e con una console via seriale attiva, il comando è:

```
mknbi-linux -output=/tftpboot/vmlinuz-athlon immagine-kernel \  
--append="root=/dev/nfs ip=dhcp console=ttyS0,9600n8"
```

Poiché il kernel viene caricato via rete dai client attraverso il protocollo PXE¹¹, nel nostro caso, in cui le schede di rete erano sprovviste del firmware, è stato necessario utilizzare un FD¹² di boot con il firmware specifico per la scheda di rete 3Com¹³, una volta ottenuta l'immagine si usa il programma *dd* per riversarlo su disco:

```
dd if=3c500b.img of=/dev/fd0
```

2.9 Configurazione della rete sui nodi

L'indirizzo IP per il FS (prima scheda), viene configurato al boot della macchina, ma occorre configurare l'indirizzo della seconda scheda di rete.

¹⁰ <http://etherboot.sourceforge.net/>

¹¹ PXE: Pre boot EXecute

¹² FD: Floppy Disk

Nel caso si utilizzi una scheda Gigabit ethernet su una versione di Linux 2.4.19 o precedente, come nel nostro caso, o si utilizzi una scheda di rete i cui driver non siano inclusi nei sorgenti del kernel, occorre compilare un apposito modulo.

Per compilarli, dopo averli scaricati, basta eseguire:

```
tar xzf e1000.tgz
cd e1000
make
make install
```

Figura 21 – Installazione driver scheda Intel pro 1000

Per abilitarla, occorre inserire in un file di boot tipo */etc/init.d/boot.local*:

```
modprobe e1000
```

oppure in */etc/modules*:

```
e1000
```

Per usare il dhcp anche per questa scheda si modifica il file */etc/network/interface* aggiungendo queste righe:

```
auto eth0 eth1
iface eth0 inet dhcp
iface eth1 inet dhcp
```

Figura 22 – Configurazione dinamica rete

Si può anche optare per una configurazione statica, inserendo, al posto dell'ultima riga i dati della scheda eth1 già visti nella configurazione del DHCP:

¹³ www.etherboot.com

```
iface eth1 inet static
    address 100.100.100.101
    netmask 255.255.255.0
    network 100.100.100.0
    broadcast 100.100.100.255
```

Figura 23 – Configurazione statica rete

Questa è la configurazione della prima macchina.

2.10 Configurazione openMosix

A questo punto i nodi del cluster sono in grado di fungere da client DHCP e la seconda scheda di rete dovrebbe essere già configurata.

Manca la configurazione del software per utilizzare la macchina come un cluster, partiamo quindi dalla configurazione di openMosix.

In realtà l'installazione di openMosix è stata effettuata alla ricompilazione del kernel, occorrono solo alcune accortezze per avviare il servizio di openMosix.

Dopo avere applicato la patch al kernel scaricata dal sito <http://www.openmosix.org/> assieme agli user tools e configurato il kernel adeguatamente, si può procedere all'installazione di questi ultimi.

Nel caso di un sistema sysV sarà presente uno script per lanciarlo in */etc/init.d/openmosix* ed all'avvio sarà già possibile avere openMosix funzionante.

Occorre indicare però le macchine del cluster all'interno del file */etc/mosix*, nel nostro caso:

```
1      100.100.100.101  4
```

I numeri indicano:

- 1- nome del primo nodo openMosix;
- 2- IP della prima macchina;
- 3- numero di macchine appartenenti al cluster con indirizzi successivi al secondo numero.

Il file può contenere diverse righe di questo tipo se gli indirizzi IP non sono tutti contigui.

Una cosa da fare è impostare l'hostname sul nodo rispetto alla seconda interfaccia di rete con il comando `host` oppure scrivendolo direttamente nel file `/etc/hostname`.

I nomi utilizzati sono `beox`, dove *x* è il numero del nodo openMosix.

Nel file `/etc/fstab` si deve inserire la riga relativa al MFS:

```
cluster    /mfs  mfs    auto  0 0
```

A questo punto si può far partire openMosix con il comando:

```
/etc/inet.d/openmosix start
```

Al reboot, ogni macchina sarà in grado di effettuare automaticamente questa procedura; ogni nodo aggiuntivo necessiterà a questo punto sola la copiatura delle directory precedentemente viste e della personalizzazione dei file `hostname` e `network/interface` sotto la directory `/etc`.

Questa procedura evita la configurazione di un servizio aggiuntivo quale NYS.

I programmi lanciati sui nodi del cluster potranno essere eseguiti su una qualsiasi delle macchine secondo l'algoritmo di balancing usato da openMosix.

E' possibile anche indicare il nodo su cui eseguire il programma o in alternativa un insieme di nodi preferiti con il comando `mosrun`:

```
mosrun -jn[-ln[,n2[-n3]]] <comando> [parametri comando]
```

Dove con *nx* sono indicati i nodi preferiti che possono essere in una lista separati da virgole o un range contiguo separando il primo dall'ultimo con il -. A tal proposito, se gli openMosixtool non sono stati installati attraverso il loro pacchetto rpm precompilato, su alcune distribuzioni la migrazione non sarà effettuata in modo automatico senza l'utilizzo di *mosrun*.

Per avere una migrazione automatica su tali sistemi, occorre inserire la seguente porzione di codice all'interno del file */etc/inittab*:

```
/bin/mosrun -h
```

questo in ogni riga contenente un comando del tipo:

```
/etc/rc.d/rc*
```

prima del comando stesso.

2.11 Installazione di MPI

La versione di MPI al momento disponibile è la 1.2.5, I passi i per la compilazione sono:

```
cd /usr/local
tar xzf mpi-1.2.5.tar.gz
mv mpi-1.2.5 mpich
./configure --arch=LINUX --device=ch_p4:-listenersig=SIGUSR2 \
--mpe --comm=shared --enable-c++ --opt=-O --disable-dev-debug \
--prefix=/usr/local/mpchi --exec-prefix=/usr/local/mpich
make serv_p4
make
make install
```

Figura 24 – Compilazione ed Installazione MPI

Le opzioni di configurazione indicano:

- 1- installare librerie ed eseguibili sotto la root
/usr/local/mpich;
- 2- usare il device *ch_p4*, utilizzato per la comunicazione in un cluster;
- 3- Utilizzo del SIGUSR 2, per poter essere utilizzata con i thread (altrimenti genera un conflitto di segnali)
- 4- compilare il supporto per l'interfaccia di debug MPE;
- 5- compilare le librerie in versione dinamica;
- 6- disabilitare il debug del codice della libreria.

Nel file */usr/local/mpich/var/machine.LINUX*, vanno inserite, una per ogni riga, i nomi delle macchine facenti parte del nostro cluster con il relativo numero di processori che si vuole utilizzare sulla macchina separati da ':':

```
hostname:2
```

Nel file */etc/skel/.bashrc*, va inserita la seguente riga:

```
export P4_GLOBSIZE=200000000
export PATH=$PATH:/usr/local/mpich/bin
```

Questa indica di utilizzare una dimensione massima di circa 200 MB per i pacchetti di MPI; saranno poi le stesse librerie che li convertiranno in più pacchetti TCP validi di dimensione adeguata.

Questo valore è idoneo al nostro cluster, ma va modificato secondo le relative necessità.

In questo modo tutti i nuovi utenti aggiunti utilizzeranno questo valore, la seconda riga aggiunge la directory con gli eseguibili per mpi nel path.

Infine, i nomi di tutti gli host, o in alternativa gli indirizzi IP, vanno scritti nel file */etc/host.equiv* uno per riga.

Questo passo si rende necessario per utilizzare correttamente i comandi `rexec` e `rsh` senza dover inserire di volta in volta la password dell'utente che li ha invocati; MPI adopera pesantemente questi comandi e saltarlo renderebbe inutilizzabile quest'ultimo.

Il file delle password viene sincronizzato attraverso uno script che copia i file *passwd*, *shadow*, *group* e *sgroup* nelle directory */etc* dei nodi; lo script va lanciato ad ogni creazione di nuovi utenti.

Ora i programmi potranno essere eseguiti attraverso il comando *mpirun*.

2.12 NYS e autenticazione centralizzata

Come è stato anticipato, l'autenticazione degli utenti, è stata effettuata tramite replica dei file delle password su tutti i nodi del cluster attraverso l'utilizzo di uno script; è possibile utilizzare, in alternativa, un servizio di autenticazione come NIS, LDAP o altri.

La scelta migliore può essere l'utilizzo di NYS, una versione avanzata del protocollo NIS¹⁴.

L'utilizzo di questo protocollo permette di gestire in modo semplice (almeno se non necessitano politiche di sicurezza molto restrittive) e centralizzato gli account sui nodi del cluster, creandoli una volta per tutte e senza doverli sincronizzare.

Inoltre è possibile effettuare il caching delle password con lo stesso meccanismo del DNS attraverso il `nsd`¹⁵ in modo da non dovere interrogare il demone NYS ad ogni `rsh` o `rexec`.

Il meccanismo a cui si aggancia NYS, è lo stesso di NFS: il `portmap`.

La configurazione avviene manualmente attraverso i seguenti comandi:

¹⁴ NIS: Network Information Protocol

```
ypdomainname beowulf-data
ypserv
rpc.yppasswdd
ypbind
```

Figura 25 – Configurazione dominio NYS

Come si può notare, occorre definire un dominio NYS, lanciare il servizio principale e lanciare il servizio per la modifica, da parte dei client, della password attraverso il comando `yppasswd`; infine il servizio per la risoluzione dei nomi NYS.

Sul client occorrerà solamente specificare il dominio NYS nel file `/etc/yp.conf`:

```
domain beowulf-dati broadcast
```

Broadcast indica al client di cercare il server NYS in rete facendo una richiesta in broadcast.

Sui client, inoltre è necessario che l'autenticazione degli utenti possa utilizzare NYS (configurazione di PAM).

2.13 Risoluzione dei nomi

Le opzioni scelte per la risoluzione dei nomi sono state l'utilizzo del file `/etc/hosts`, oppure tramite DHCP.

Nel caso sia stata utilizzata la configurazione statica della seconda interfaccia, occorre almeno inserire i client della rete 100.100.100.0/24, in caso contrario, il file `/etc/hosts` risulta del tutto superfluo.

¹⁵ nscd: name service cache daemon

Nel caso il cluster sia composto da un numero elevato di nodi, conviene optare per una soluzione basata su DHCP, se non altro per il non dovere replicare il file per ogni client.

L'utilizzo del DNS si rende superfluo dal momento che può essere comodamente utilizzato DHCP, tranne in caso di particolare appesantimento della rete del cluster.

2.14 Sicurezza

Come indicato all'inizio, l'unica macchina visibile dall'esterno è il server che, possedendo un indirizzo IP pubblico, è suscettibile ad attacchi dall'esterno.

Gli altri nodi del cluster appartengono invece ad una rete interna privata ed i pacchetti delle macchine non possono in teoria uscire dalla rete a loro dedicata e non sono instradabili direttamente dai router.

Per questo motivo, si è deciso di separare il più possibile il master dagli altri nodi, cercando di non farlo intervenire nel calcolo, inoltre è stato configurato un firewall basato su *ipfilter* che vieta il transito di pacchetti alle due reti interne private e tutto il traffico dall'esterno verso il master, lasciando aperte solo le porte in uscita e la porta 22 per il SSH ed il SFTP.

È stato anche configurato il *tcpwrapper* per impedire le connessioni *tftp*, *mountd* ed *nfsd* ad eccezione del dominio ***beowulf-fs*** e le home degli utenti e le directory *tmp* sono state montate con opzione *nosuid*, in modo da evitare che vengano eseguiti programmi con il bit *suid* attivo di proprietà di altri utenti.



Figura 26 – Il cluster realizzato: in basso da sinistra il nodo master ed a destra i 4 nodi dual Athlon; in alto da sinistra i due monitor per la console, il firewall con il suo monitor ed una workstation

Capitolo 3

Metodologie software per l'ottimizzazione di algoritmi

Nei precedenti capitoli, sono state classificate diverse architetture parallele disponibili attualmente e l'implementazione di un cluster, ora verranno presentate diverse tecniche software utilizzate sia per trarre vantaggio dalle architetture presentate, sia per ottenere un'implementazione efficiente ed il più veloce possibile di algoritmi. Analogamente al capitolo 1, verrà ora focalizzata l'attenzione su alcune tecniche di programmazione che vedremo all'opera nei successivi capitoli, applicate sia a toy test, sia ad una reale applicazione.

Nell'ordine sono:

- 1- SIMD¹
- 2- IPC²
- 3- Thread
- 4- MPI

¹ SIMD: Single Instruction Multiple Data stream

² IPC: InterProcess Communication

3.1 SIMD

Le SIMD sono istruzioni che permettono di eseguire la stessa operazione elementare (addizione, moltiplicazione ...) su più dati contemporaneamente risparmiando così preziosi cicli macchina, tradizionalmente sono le istruzioni dei comuni DSP³ per HI-FI⁴, dedicati all'elaborazione di dati digitali quali l'audio.

L'utilizzo di queste istruzioni è particolarmente indicato in quegli algoritmi che fanno un uso intensivo di prodotti matriciali, in quanto permettono di eseguire in modo atomico espressioni tipo:

$$\mathbf{C}_i = \alpha \mathbf{A}_i \times \mathbf{B}_i + \beta \mathbf{C}_i$$

Dove \mathbf{A}_i , \mathbf{B}_i e \mathbf{C}_i sono sottomatrici di $\mathbf{A}_{n \times k}$, $\mathbf{B}_{k \times m}$, $\mathbf{C}_{n \times m}$ ed α , β coefficienti scalari.

$$\left[\begin{bmatrix} \mathbf{C}_i \end{bmatrix} \right] = \alpha \left[\begin{bmatrix} \mathbf{A}_i \end{bmatrix} \right] \times \left[\begin{bmatrix} \mathbf{B}_i \end{bmatrix} \right] + \beta \left[\begin{bmatrix} \mathbf{C}_i \end{bmatrix} \right]$$

Figura 27 – Tipica operazione eseguita dalle istruzioni SIMD

La sempre maggiore disponibilità di risorse multimediali per PC e la rispettiva maggiore esigenza computazionale delle stesse, ha reso disponibile questa tecnologia sui tradizionali calcolatori e workstation, inizialmente attraverso l'ausilio di DSP (il NeXT⁵ ne fu un precursore con il DSP MC56001 della Motorola⁶) e successivamente integrandola all'interno delle CPU general purpose stesse come set di istruzioni assembler estese.

³ DSP: Digital Signal Processor

⁴ HI-FI: HIgh-FIdelity

⁵ www.next.com

⁶ www.motorola.com

Di queste ultime, le più diffuse sono Intel iSSE⁷ o SSE, Motorola Altivec e SUN VIS⁸, di cui tratteremo approfonditamente solo la prima per via della sua disponibilità elevata (tutti i PC di ultima generazione ne sono provvisti).

Differentemente dalle altre metodologie e tecnologie che vedremo, questa è strettamente legata all'architettura della macchina (CPU o DSP utilizzata/o), il che rende il codice scritto utilizzando queste istruzioni non portabile, a meno di utilizzare delle librerie multiplatforma che forniscano un'astrazione della particolare architettura utilizzata.

3.1.1 SSE

Intel introdusse istruzioni SIMD sin dal 1997 con il nome di MMX⁹; queste consistevano in 57 nuove istruzioni che permettevano di manipolare in vario modo interi a singola precisione (16 bit).

L'approccio Intel si basa sull'inserimento di più dati dello stesso tipo all'interno di registri speciali ed all'utilizzo di questi per le relative operazioni (si parla di SWAR¹⁰); con la tecnologia MMX vennero utilizzati gli 8 registri floating point, ed ogni volta che si passava da un calcolo SIMD ad uno floating point occorreva fare un cambio di contesto e reinizializzare i registri tramite l'apposita istruzione *EMMS*.

Con il PIII, furono poi introdotte 50 nuove istruzioni SIMD per l'utilizzo di operatori in virgola mobile a singola precisione (32 bit), appunto le SSE.

Questa seconda generazione di istruzioni utilizza 8 nuovi registri (xmm0-

⁷ iSSE: intel Streaming SIMD Extension

⁸ VIS: Vector Instruction Set

⁹ A seconda delle fonti, MMX - MultiMedia eXtension e MMX - Matrix Manipulation eXtension

¹⁰ SWAR: SIMD Within A Register

xmm7) a 128 bit e non necessita il ripristino dello stato per passare dall'esecuzione di istruzioni SSE a floating point.

Le nuove istruzioni comprendono:

- 1- spostamenti di blocchi da 64 e 128 bit dalla memoria ai registri e viceversa;
- 2- operazioni aritmetiche;
- 3- operazioni logiche/comparative;
- 4- operazioni di conversione dati;
- 5- operazioni di prefetch delle istruzioni in cache.

Le operazioni simultanee permesse, grazie all'utilizzo dei nuovi registri, sono un massimo di 4 operazioni elementari con 4 operandi per registro.

I dati sono memorizzati in memoria ed utilizzati nei registri in modalità packed.

Questa modalità permette di elaborare 128 bit alla volta in floating point, trattando 4 dati come se fossero un unico dato.

Lo stato attuale permette di eseguire 4 moltiplicazioni e 2 addizioni contemporaneamente utilizzando dati interi.

Inoltre è possibile eseguire operazioni scalari utilizzando solo la parte più bassa dei registri (16 bit e 32 bit rispettivamente per interi e floating point) riportando gli altri dati inalterati.

Non essendo possibile settare il registro dei flag per tutti i dati, vi è la possibilità di saturare il valore finale; in questo caso, se il valore risultante dall'operazione eseguita supera la precisione degli operatori, essi vengono posti al valore più alto rappresentabile, altrimenti vi è un overflow ed i dati rappresentano solamente la parte più bassa del numero rappresentabile con la precisione dei dati (sempre rappresentati in modo packed).

Queste istruzioni permettono di ottenere un incremento notevole di prestazioni, in tabella 4 si possono vedere i cicli di clock per eseguire alcune operazioni SIMD su Pentium, inoltre queste istruzioni sono accoppiate ad altre nelle pipeline del processore, quindi terminata l'esecuzione della prima istruzione, le successive possono essere eseguite alla velocità di 1 ciclo di clock l'una. Nei paragrafi successivi vedremo perché questo non è sempre vero e come fare in modo che succeda.

ISTRUZIONE	DETTAGLI	PIII	P4	ATHLON
MOVD MEM,REG32	Copia 32 bit da un registro in memoria. Istruzione MMX	1	2	3
MOVQ MEM,REG	Copia 64 bit da un registro alla memoria. Istruzione MMX	1	6	2
MOV MEM,REG16	Copia un valore a 16 bit da un registro in memoria. Istruzione x86	4	4	4
PADDSD	Esegue 4 addizioni di word contemporaneamente. Istruzione MMX	1	2	3
PMULHW	Esegue 4 moltiplicazioni contemporaneamente su word e salva la parte più alta del numero. Istruzione MMX	3	8	3
MOVAPS MEM,REG	Sposta dati allineati a 128 bit in memoria. Istruzione SSE	1	1	1
MOVUPS MEM,REG	Sposta dati non allineati da registro a memoria. Istruzione SSE	1	1	1
MULPS	Esegue 4 moltiplicazioni di float single precision. Istruzione SSE	1	1	1
ADDPS	Esegue 4 addizioni di float. Istruzione SSE	1	1	1
ADD REG,REG	Somma 2 valori. Istruzione x86	1	1	1
MUL REG16	Moltiplica valori interi a 16 bit. Istruzione x86	11(1)	11(1)	11(1)
MUL REG32	Moltiplica valori interi a 32 bit. Istruzione x86	10(1)	10(1)	10(1)
FADD STi,ST	Addizione floating point Istruzione FPU	1	1	1
FMULL STi,ST	Moltiplicazione floating point Istruzione FPU	1	1	1

Tabella 4 – Cicli di clock di alcune istruzioni SSE/MMX e x86

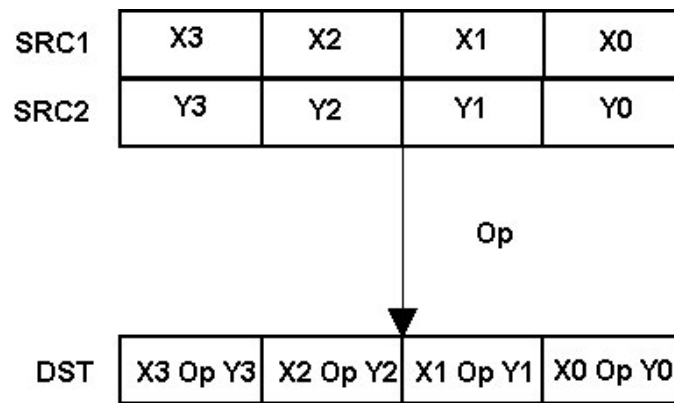


Figura 18 – Struttura istruzione aritmetica SSE

3.1.2 Allineamento dei dati

In generale l'allineamento dei dati nei programmi migliora sensibilmente le performance degli stessi e non ne sono un'eccezione i programmi basati su codice SSE, che mette a disposizione due nuove istruzioni per lo spostamento di dati packed da registro a registro, da registro a memoria e viceversa:

1- *movups*

2- *movaps*

La prima viene utilizzata per copiare dati in cui la memoria non è allineata o non è detto che lo sia; la seconda per copiare dati in cui la memoria risulta allineata a 16 byte (128 bit o paragrafo).

Nel caso vi sia una copia da registro a registro, le istruzioni sono del tutto equivalenti (un registro MMX equivale ad un'area di memoria già allineata alla dimensione minima utile per questo tipo di istruzioni).

La memoria viene allineata da alcuni compilatori ed assembler a word, ovvero 2 byte, ma la tendenza attuale è quella di avere un allineamento almeno a 8 byte, come nel compilatore gcc o nello stesso compilatore Microsoft Visual C++.

Questi valori sono ancora insufficienti per le applicazioni SSE e ne consegue l'impossibilità nell'utilizzare la seconda istruzione di spostamento, che genererà un segmentation fault non appena il programma accederà ad un blocco di memoria non allineata alla dimensione dei dati SSE.

Sulle architetture iA32¹¹ dal Pentium in poi, i compilatori che generano codice ottimizzato effettuano un allineamento a 8 byte, e la procedura *malloc()* alloca porzioni di memoria allineate a tale valore essendo l'architettura in grado di accedere alla memoria 64 bit alla volta.

Per allineare i dati alla dimensione voluta, si utilizzano le seguenti procedure alternative alla classica *malloc()*:

- 1- *valign()*
- 2- *posix_memalign()*

La prima si usa analogamente alla *malloc()* e cerca di allocare una porzione di memoria allineata alla dimensione della pagina; questo approccio non è più utilizzato sui nuovi sistemi operativi, in quanto presenta diversi svantaggi, tra i quali:

- 1- dipendenza della dimensione delle pagine dall'architettura;
- 2- allocazione della memoria sulla disponibilità di pagine completamente libere;
- 3- spreco di memoria maggiore.

Il primo punto non ci interessa direttamente, in quanto l'ottimizzazione proposta è comunque platform-dependent e la portabilità del codice non è un problema al momento, mentre il secondo punto è più interessante, in quanto, se la disponibilità di pagine di memoria completamente libere finisce prima dell'allocazione dei miei dati, si avrà anche in questo caso un segmentation fault; infine (terzo punto), con queste dichiarazioni le

¹¹ iA32: inter Architecture 32 bit

pagine sono allocate per intero, quindi se si dichiara anche solo una stringa di 10 caratteri, la memoria allocata sarà come minimo una pagina. Anche in questo caso però, se si devono allocare delle matrici di grandi dimensioni la parte sprecata di memoria sarà esigua rispetto ai dati trattati, e per la dichiarazione degli altri dati sarà sempre possibile utilizzare la classica `malloc()`.

Un appunto a favore dell'utilizzo di questa funzione risulta nella portabilità, tutte le distribuzioni di Unix (e quindi anche Linux e FreeBSD) dispongono di un'implementazione della stessa all'interno delle loro libc e viene supportata anche da compilatori gcc della serie 2.9. La seconda funzione risolve tutti i problemi, permettendo di allocare memoria allineata ad un numero arbitrario di byte, purché siano delle potenze di 2, pena un `segmentation error`.

L'unico appunto risulta nella disponibilità solo a partire dalla versione 2.3 delle glibc per Linux e supportata solo dal gcc versione 3.0 o successivo.

Un altro problema per l'allineamento dei dati, risulta nello stack.

Quando viene richiamata una funzione, le variabili passate per valore sono copiate nello stack (nel caso del linguaggio C) e non necessariamente lo stack è già allineato o correttamente allineato.

Una soluzione consiste nel passare le variabili per puntatore, cosa sicuramente più corretta in caso di dati di grandi dimensioni (generalmente strutture molto complesse, in quanto i vettori sono comunque passati per indirizzo come per le matrici).

Una possibilità da non scartare nemmeno nel caso precedente, consiste nel definire l'allineamento dello stack attraverso direttive al compilatore come:

`--with-stack-alignment 16`

che allinea lo stack a 16 byte come serve a noi.

Nel caso il dato non coincida con la dimensione od un multiplo della stessa, come una push di un registro a 32 bit, I restanti bit vengono impostati a zero fino al riempimento dei 16 byte (nel nostro caso).

Sebbene generalmente non sia necessario allineare i dati sullo stack, a volte è consigliato farlo per aumentare le prestazioni, a prezzo di uno spreco maggiore di memoria.

Anche in questo caso va fatto un'appunto: con versioni vecchie del compilatore gcc non è possibile allineare la memoria sullo stack a più di 12 byte su architettura iA32.

È possibile allineare anche i salti (jump), le procedure (call) ed il segmento del codice con le relative direttive al compilatore (gcc 3.x):

--falign-jump=8

--falign-functions=8

--malign-double

queste però non inficiano le prestazioni come nel caso dei dati, e non a caso le prime 2 sono impostate ad 8, non essendo possibile allineare ad un valore maggiore il codice sulle architetture iA32 come avviene invece con i processori RISC a 64 bit e con alcuni a 32 bit con il compilatore gcc della GNU.

3.1.3 Prefetch del codice e dei dati

La velocità dei processori attuali, supera di molto quella della memoria, in alcuni casi addirittura di un fattore 30, il che rende infinitamente lento e poco produttivo l'accesso alla memoria centrale.

Per limitare il fenomeno, occorre utilizzare in modo intensivo e mirato le varie cache dei processori divise in livelli via via più lenti:

- 1- L0: i registri stessi;
- 2- L1: interna al core del processore, 64k per l'Intel PIII;
- 3- L2: esterna al processore, con la tendenza ad integrarla nello stesso die assieme al core della CPU in modo che possa lavorare alla stessa frequenza di quest'ultima.

Programmando in assembler, occorre quindi fare in modo di sfruttare al massimo tutti i registri disponibili, il che non è molto difficile su un processore CISC¹² come l'iX86, in quanto dispone di pochi registri: 16 general purpose contro i 20/30 dei processori RISC¹³ attuali.

Occorre poi fare in modo che le istruzioni e soprattutto i dati, siano presenti sempre in cache, almeno L2 e quindi ridurre al minimo i cache missing più che in passato, dato che è inutile eseguire potenzialmente 4 operazioni in un ciclo di clock se poi bisogna aspettarne altri 30 per accedere ai dati da elaborare.

In genere, questo lavoro viene già svolto egregiamente dal processore, ma quando non basta, è possibile modificare la scelta dello stesso su come inserire e mantenere i dati in cache.

Occorre a questo punto fare una distinzione tra i vari tipi di dati a cui un programma accede in riferimento alla loro località:

- 1- temporale (temporal data);
- 2- non temporale (non-temporal data);
- 3- spaziale (spatial location data).

I dati del primo tipo sono quelli che, per un motivo o per un altro vengono acceduti più volte durante l'esecuzione del codice, e presumibilmente dovranno essere preservati in cache per più tempo possibile.

Il secondo tipo appartiene a quei dati che una volta utilizzati, non

¹² CISC: Complex Instruction Set

¹³ RISC: Restricted Instruction Set

necessitano più di essere acceduti.

Il terzo tipo di dati può essere sia del primo che del secondo, e si riferisce a quei dati collocati in locazioni adiacenti di memoria; molti algoritmi di ottimizzazione sfruttano la spazialità dei dati, in quanto se una locazione viene acceduta, la probabilità che quella successiva sia acceduta a sua volta è molto alta, basti pensare alla gestione della cache del disco che si basa su questo principio per memorizzare i dati dei blocchi adiacenti a quello letto prima che essi siano acceduti, oppure alle operazioni su vettori o matrici; se si accede ad un elemento, quasi sicuramente si dovrà accedere all'elemento successivo nella stessa riga di appartenenza.

Da un lato occorre quindi evitare che i **non-temporal data** “inquinino” la cache, ma anche che i **temporal-data** siano presenti in cache.

Esistono al proposito istruzioni di caricamento apposite per i **non-temporal data**, che non utilizzano la cache, ma eseguono una scrittura write-through in memoria evitando così di sporcare la cache a qualsiasi livello.

Per il nostro tipo di applicazione, il problema però non si risolve e la nostra necessità sarebbe quella di avere in cache i dati relativi alla porzione di matrice al momento del suo utilizzo, indipendentemente se siano riutilizzati in seguito o meno.

Il caricamento in cache dei dati prima che siano effettivamente utilizzati è detto **prefetch**.

Le CPU della classe Pentium eseguono già il prefetch dei dati congiuntamente ad un meccanismo di predizione dell'ordine di utilizzo; nelle moderne CPU e nell'architettura SSE vi è la possibilità di forzare in qualche modo questo meccanismo.

In SSE vi è una istruzione apposita:

prefetchN

Questa istruzione permette di caricare i dati ad un livello di cache suggerito, in modo che siano disponibili al processore immediatamente quando necessario e la N equivale al tipo di prefetch da eseguire.

ISTRUZIONE	Azione	Criterio
Prefetch0	Inserisce i dati in tutti i livelli di cache: PIII: L2 o L1 P4 e Xeon: L2 Athlon: L0, L1 o L2	Temporal data
Prefetch1	Inserisce i dati nella cache L2 o superiore PIII P4 e Xeon: L2 Athlon: L1 o L2	Temporal data
Prefetch2	Attualmente come prefetch1 Su Athlon: solo L2	Temporal data
Prefetchinta	Inserisce i dati in cache solo se disponibili delle aree inutilizzate minimizzando l'inquinamento della cache PIII: L1 P4 e Xeon: L2 Athlon: L1 o L2	Non-temporal data

Tabella 5 – Istruzioni di prefetch dei dati

L'utilizzo di questa istruzione consente di ottimizzare la velocità di esecuzione del codice, ma utilizzandola spesso si rischia di consumare la banda della memoria, quindi va utilizzata solamente in sezioni critiche di codice.

3.2 Processi

Il prossimo argomento trattato riguarda la comunicazione tra processi o IPC.

Prima di entrare nel vivo della discussione, occorre fare una premessa illustrando cosa si intenda per processo.

In generale, il termine processo viene utilizzato per indicare un programma in esecuzione.

Il processo è l'astrazione di un programma in esecuzione utilizzata dal sistema operativo.

Ogni processo ha una sua memoria, un suo stack, un suo PC¹⁴ relativo all'istruzione eseguita attualmente, un'immagine della CPU comprensiva dei registri e flag, che identificano lo stato della macchina e del processo. Nei sistemi multiprogrammati, possono esistere diversi processi attivi che verranno eseguiti per un certo periodo di tempo secondo le politiche di schedulazione del OS¹⁵ dando all'utente l'illusione di essere eseguiti contemporaneamente.

Ogni volta che il calcolatore passa dall'esecuzione di un processo ad un altro, l'OS esegue quello che si dice “**context switching**”, cioè fa cambiare lo stato della macchina in modo che possa eseguire il processo nel punto in cui era stato sospeso precedentemente.

Per creare un processo, viene utilizzata la system-call `fork()` in sistemi operativi Unix, che equivale alla `clone()` su sistemi Linux.

Questa system-call permette di creare un processo figlio partendo da un processo padre dal quale viene richiamata.

¹⁴ PC: Program Counter; IP: Instruction Pointer su architetture 80x86

¹⁵ OS: Operating System

Il processo figlio sarà un clone perfetto del processo padre al momento della sua creazione, cioè avrà un'immagine di memoria duplicata e lo stato dei registri della CPU saranno gli stessi.

Il programma figlio a questo punto continuerà la sua esistenza eseguendo un compito generalmente differente da quello del padre che continuerà anch'esso il suo lavoro.

Per discriminare il padre dal figlio viene fatto un controllo sulla chiamata `fork()`, se il processo è il padre verrà riportato il PID¹⁶ del figlio, altrimenti 0.

Un simile meccanismo necessita sempre la presenza di un processo per poterne lanciare un altro, in genere l'utente crea i processi ogni qual volta lancia un programma da shell, quindi il processo padre risulta essere la shell stessa; all'avvio della macchina viene creato un processo con PID 0, detto di IDLE o processo inoperoso utilizzato come padre per creare nuovi processi.

A volte occorre poter controllare processi da altri processi, ed è qui che entrano in gioco le IPC.

Queste funzioni forniscono un supporto per la programmazione concorrente, mettendo a disposizioni monitor, semafori, sezioni critiche, code di messaggi, pipe e memoria condivisa.

Noi utilizzeremo queste strutture per modellare i nostri algoritmi paralleli, utilizzando la funzione `fork()` o `clone()` per far eseguire porzioni di calcolo differenti alle CPU dei nostri computer o ai nodi del nostro cluster.

Su macchine multiprocessore, si possono effettivamente eseguire contemporaneamente tanti programmi quante sono le CPU, e lo stesso vale per cluster di tipo openMosix.

¹⁶ PID: Process IDentificator

L'implementazione più naturale di algoritmi paralleli su cluster openMosix, risulta essere l'implementazione di più processi identici, che eseguono gli stessi calcoli su porzioni di dati differenti e che dialogano tra di loro attraverso l'IPC.

3.2.1 IPC

Come accennato, il meccanismo standard, per sincronizzare i processi, risulta essere l'IPC.

Qui di seguito ci riferiremo alle IPC di Unix/Linux.

Quelle introdotte nello standard Posix e presenti nelle implementazioni ANSI di compilatori C sono:

- 1- signals;
- 2- file locking;
- 3- pipe;
- 4- FIFO¹⁷.

A cui vanno aggiunte alcune estensioni introdotte su system V ed anch'esse Posix compliant:

- 5- code di messaggi;
- 6- semafori;
- 7- memoria condivisa.

Su questi ultimi non ci soffermeremo più di tanto, approfondendoli meglio durante la trattazione dei thread più avanti in questo capitolo.

In genere, per utilizzare questi meccanismi vengono allocate risorse identificate da descrittori di file, per controllare queste risorse si può ricorrere a programmi quali:

¹⁷ FIFO: First In First Out

- 1- *ipcs*;
- 2- *ipcrm*.

Il primo controlla le risorse allocate, il secondo consente di rimuoverle.

3.2.2 Signals

Sono i segnali standard che possono essere inviati ai processi.

Attraverso la system call:

signal(SIGNUM,proc_handler)

è possibile ridefinire il comportamento del programma in modo che quando riceve un segnale di tipo *SIGNUM* esegua la procedura *proc_handler* prima di eseguire la signal vera e propria.

```
#include <stdlib.h>
#include <signal.h>
int main(void)
{
    void sigint_handler(int sig); /* prototipo */
    /* set up handler */
    if (signal(SIGINT, sigint_handler) == SIG_ERR)
    {
        exit(1);
    }
    return 0;
}

/* handler */
void sigint_handler(int sig)
{
    printf("Test");
}
```

Figura 29 – Premendo CTRL+C viene eseguito sigint_handler

Signal	Significato
SIGABRT	Process abort signal.
SIGALRM	Alarm clock.
SIGFPE	Erroneous arithmetic operation.
SIGHUP	Hangup.
SIGILL	Illegal instruction.
SIGINT	Terminal interrupt signal.
SIGKILL	Kill (non può essere sospesa o ignorata).
SIGPIPE	Write su a pipe su cui non è stata fatta nessuna read.
SIGQUIT	Terminal quit signal.
SIGSEGV	Invalid memory reference.
SIGTERM	Termination signal.
SIGUSR1	User-defined signal 1.
SIGUSR2	User-defined signal 2.
SIGCHLD	Un processo appena creato è terminato o stoppato
SIGCONT	Continua l'esecuzione
SIGSTOP	Stop executing (non può essere ignorata).
SIGTSTP	Terminal stop signal.
SIGTTIN	Background process attempting read.
SIGTTOU	Background process attempting write.
SIGBUS	Bus error.
SIGPOLL	Pollable event.
SIGPROF	Profiling timer expired.
SIGSYS	Bad system call.
SIGTRAP	Trace/breakpoint trap.
SIGURG	High bandwidth data is available at a socket.
SIGVTALRM	Virtual timer expired.
SIGXCPU	CPU time limit exceeded.
SIGXFSZ	File size limit exceeded.

Tabella 6 – Segnali standard

I segnali sono inviati ai processi attraverso la shell con il comando *kill* o con l'utilizzo delle combinazioni speciali di tasti come *ctrl-c* o *ctrl-z*, la prima interrompe l'esecuzione del programma (SIGINT), la seconda la sospende (SIGSTOP).

Per poter inviare segnali ad un processo da shell occorrono i seguenti requisiti:

- 1- occorre conoscere il PID del processo;
- 2- bisogna avere i permessi adeguati.

Il PID del processo viene semplicemente trovato utilizzando il comando

ps, oppure utilizzando il comando *killall* seguito dal nome del programma da controllare.

Per quanto riguarda i permessi, di solito il proprietario del processo è lo stesso che vuole controllarne l'esecuzione.

3.2.3 File locking

Questo meccanismo permette di sincronizzare processi in modo semplice quando questi accedono ad uno stesso file.

I lock possono essere di 2 modelli:

- 1- mandatory;
- 2- advisory.

Di ognuno poi ne esistono a loro volta di 2 tipi:

- 1- read locking;
- 2- write locking.

Il primo modello (mandatory) permette di fare un lock esclusivo, se è presente un lock in lettura, un altro processo non potrà accedere al file, se presente un lock in scrittura sarà possibile solo leggere il file.

Quando un processo fa un lock di questo tipo ed un secondo processo tenta un'operazione non permessa da questo tipo di lock, finché il primo processo non termina le operazioni sul file (può anche non usare il file) e non libera il lock, il secondo rimarrà in attesa.

Il modello advisory invece, permette di accedere al file indipendentemente dal tipo di locking, mettendo a disposizione un meccanismo per controllare se il file è in lock da un altro processo.

Per la sincronizzazione dei processi è sufficiente poter controllare se il file è in lock oppure no, quindi vedremo solo l'advisory.

Per settare un lock su di un file si può utilizzare la funzione *flock()*, oppure la funzione:

fcntl(fd, FLAG, &fl)

Questa seconda funzione permette di:

- 1- settare i lock;
- 2- resettare i lock;
- 3- leggere lo stato dei lock.

I parametri sono nell'ordine:

- 1- il descrittore del file precedentemente aperto;
- 2- il tipo di operazione da eseguire sul file;
- 3- una struttura che indica vari parametri.

Il tipo di operazione da eseguire può essere:

- 1- F_GETLK: ritorna lo stato del lock sul file scrivendolo nella struttura *fl*;
- 2- F_SETLK: cerca di ottenere il tipo di lock indicato nella struttura *fl* e se non lo ottiene *fcntl()* ritorna -1;
- 3- F_SETLKW: esegue la stessa operazione di F_SETLK ma in modo bloccante e finché non ottiene il lock attende.

L'ultimo parametro è una struttura di tipo *flock*, di cui sono riportati i record in tabella 7.

```
struct flock fl;
int fd;
fl.l_type = F_WRLCK;           /* setta lock in scrittura */
fl.l_whence = SEEK_SET;       /* posizione all'inizio del file */
fl.l_start = 0;               /* Offset da l_whence */
fl.l_len = 0;                 /* 0 = lock fino alla fine del file */
fl.l_pid = getpid();          /* il PID del processo */
fd = open("filename", O_WRONLY); /* apre il file */
fcntl(fd, F_SETLKW, &fl);     /* setta il lock */
```

Figura 30 - Settaggio di un lock

<i>Record</i>	<i>significato</i>
<i>l_type</i>	Il tipo di lock da applicare, può essere: F_RDLCK per lock in lettura; F_WRLCK per lock in scrittura; F_UNLCK per unlock.
<i>l_whence</i>	Il punto da cui effettuare il lock: SEEK_SET dall'inizio del file; SEEK_CUR dalla posizione corrente del file; SEEK_END dalla fine del file.
<i>l_start</i>	Posizione relativa a <i>l_whence</i> da dove effettuare il lock
<i>l_len</i>	Lunghezza assoluta del lock
<i>l_pid</i>	Il pid del processo che possiede il lock Generalmente si richiama <i>getpid()</i> .

Tabella 7 – record della struttura *flock*

3.2.4 Pipe e FIFO

Le pipe sono il metodo più semplice per far comunicare più processi correlati fra di loro e concorrenti.

Il funzionamento delle pipe è lo stesso del '|' tra due o più comandi: in pratica la pipe viene vista come una coppia di descrittori di file, uno per la lettura ed uno per la scrittura, ma in realtà si tratta di un'area di memoria e non di un file fisicamente presenti su disco.

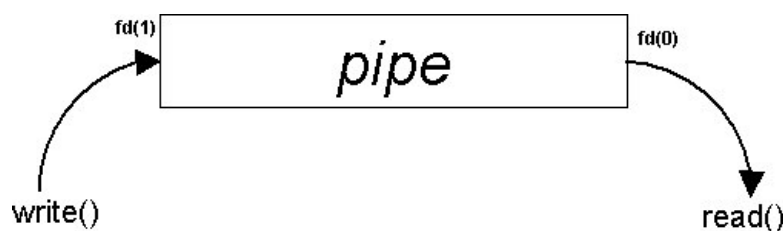


Figura 31 – Funzionamento pipe

Per creare una pipe si utilizza la system-call *pipe()*.

L'unico parametro della pipe è un array di due elementi di tipo interi che ritornano il descrittore in ingresso (scrittura) alla posizione 1 e quello di uscita (lettura) in posizione 0, come indicato in figura 31.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    int pfd[2];          /* descrittore pipe */
    char buf[30];        /* buffer */
    pipe(pfd);           /* crea le pipe */
    if (!fork())         /* fork del processo */
    {
        write(pfd[1], "test", 5); /* il figlio scrive nella pipe */
        exit(0);
    }
    else
    {
        read(pfd[0], buf, 5);      /* il padre legge dalla pipe */
        wait(NULL);
    }
}
```

Figura 32 – Utilizzo delle pipe

Il meccanismo permette ad un processo figlio creato dopo la pipe di poter usufruire di questa per avere un meccanismo simile allo scambio di messaggi tra il padre ed il figlio o tra altri processi figli.

Nel caso i processi siano indipendenti non sarà invece possibile utilizzare questo meccanismo.

Per far dialogare due processi indipendenti con lo stesso meccanismo, sono state create le *named pipe* o FIFO.

A questo tipo di pipe viene assegnato un nome esattamente come se fosse un file, a cui accedere da processi diversi.

La system-call per creare una FIFO è *mknod()*, richiamata come segue:

mknod(nomefifo, S_FIFO | fileperms , 0);

I parametri sono:

- 1- il nome della pipe;
- 2- il tipo di descrittore da creare (una FIFO in questo caso), ed i permessi da assegnargli;
- 3- Il numero del device o *minor number*, ignorato nel caso di una FIFO.

Per leggere o scrivere la FIFO dal processo padre o da un altro processo, si utilizzano le funzioni per i file.

3.3 Thread

L'utilizzo di macchine SMP consente l'utilizzo contemporaneo di 2 o più processori che condividono lo stesso spazio di memoria.

Poiché i processi sono eseguiti uno alla volta ed uno per CPU allo stesso istante di tempo, viene naturale pensare di poter sfruttare questo parallelismo per i nostri algoritmi.

Poiché la memoria è condivisa dalle due CPU, non occorre nemmeno che i processi eseguiti su una CPU dialoghino tra di loro per ottenere porzioni di dati elaborate da un altro processo in esecuzione su un'altra CPU.

Sebbene in System V sia possibile utilizzare porzioni di memoria condivise tra più processi correlati fra di loro, spesso risulta più semplice ed efficiente utilizzare un altro tipo di struttura diversa dai processi: i **thread**.

I thread possono essere visti come dei processi ridotti e non a caso sono detti anche *lightweight process*.

Il processo può essere visto come un'insieme di thread che condividono uno stesso spazio di memoria.

In questo modo è possibile utilizzare un processo con più flussi di controllo, detti semplicemente thread, tanti quante sono le CPU del sistema.

La creazione di un thread è molto più veloce della relativa creazione di un processo, in quanto, diversamente dalla *fork()*, il processo non deve essere duplicato per intero, avendo il thread immediatamente disponibile tutti i dati del programma.

Gli algoritmi paralleli, anche se ben progettati sono soggetti a tempi di esecuzione maggiori della semplice divisione del tempo totale di esecuzione su una CPU diviso il numero di CPU.

Questi ritardi sono dati in genere dai protocolli di comunicazione tra le varie parti del programma o dal tempo di startup maggiore per creare un numero maggiore di processi o di thread.

Questo concetto, noto come legge di Amdahl è indicato dalla formula:

$$T = T_p / N + T_s$$

T indica il tempo totale di esecuzione; T_s è il tempo seriale dovuto allo startup dell'applicazione ed a quelle parti di codice che per motivi intrinseci dell'algoritmo devono essere eseguiti serialmente; T_p è il tempo di esecuzione delle parti di algoritmo eseguibili in parallelo eseguito su una sola CPU e N il numero di CPU.

L'utilizzo di thread rende il processo seriale il più breve possibile.

Per sincronizzare i vari thread, si possono utilizzare in questo caso i semafori, in quanto risultano il metodo più veloce per l'accesso in mutua esclusione alla memoria e per gestire le priorità tra le varie operazioni eseguite dai vari thread.

L'uso delle pipe è sempre possibile, ma nel nostro caso, essendo la memoria condivisa, comporta un overhead dovuto alla creazione, lettura e scrittura delle pipe di dati già in memoria.

3.3.1 Linux thread

Il sistema operativo Linux fornisce la libreria *libpthread*¹⁸ per l'utilizzo dei thread.

Si tratta di una libreria *Posix compliant*¹⁹, quindi dispone di tutte le funzioni di gestione dei thread presenti sulle altre piattaforme Unix, favorendo la portabilità del codice.

Tale libreria è ben integrata nelle librerie di sistema *libc*, che forniscono l'interfaccia alle system-call per il funzionamento corretto del sistema e dei programmi in modalità kernel dei sistemi Unix.

L'implementazione dei thread che fornisce la libreria è basata sulla chiamata di sistema *clone()* di Linux, la stessa usata per creare i processi e sulla condivisione dell'area globale di memoria del processo padre con i figli.

Quando un processo viene creato, non contiene thread ed ha un'unica immagine del sistema e della memoria; quando viene inizializzato un primo thread dal processo attraverso la chiamata *pthread_create()* come risultato si avranno tre processi in memoria in luogo del vecchio processo:

- 1- Thread 1;
- 2- Thread Helper;
- 3- Thread 2.

¹⁸ Porting in Win32 a <http://source.redhat.com/pthreads-win32>

¹⁹ Posix versione 1003.1C

In realtà su sistemi Linux, sia che venga fatta una *fork()*, sia che venga fatta una *pthread_create()*, viene eseguita una chiamata alla funzione:

*clone(*function,**stack,flags,argc,argv)*

In tabella 8 sono descritti i parametri della system-call, mentre in tabella 9 sono descritti i vari flag per la condivisione dei dati.

<i>Parametro</i>	<i>Descrizione</i>
Function	Puntatore alla routine da eseguire per il nuovo thread/processo
stack	Lo stack del nuovo thread/processo
flags	Flag
argc	Numero di parametri di function
Argv	I parametri della funzione da eseguire

Tabella 8 – Parametri system call *clone()*

<i>Flag</i>	<i>Descrizione</i>
CLONE_VM	Condivide memoria dati e stack
CLONE_FS	Condivide le informazioni del filesystem
CLONE_FILES	Condivide i file aperti
CLONE_SIGHAND	L'invio di un segnale al padre o al figli viene automaticamente inviato a tutti e due
CLONE_PID	Condivide il PID con quello del padre

Tabella 9 – flag della system call *clone()*

Nel caso la system-call venga richiamata da una *fork()*, la funzione richiamata sarà la *main()*.

I flag possono essere combinati attraverso l'uso dell'operatore | (or binario) e nel caso della *fork()* saranno usati il secondo, il terzo ed il quarto.

Tornando ai thread, essi hanno codice e dati condivisi e verrà eseguita la funzione indicata da *pthread_create()*.

Veniamo quindi al significato dei 3 processi iniziali, il primo processo è il padre, che verrà visto come un thread e continuerà ad eseguire le sue funzioni, il secondo processo, è un processo che condivide il PID del padre e funziona da dispatcher, l'ultimo processo è il nuovo thread creato. Da adesso in avanti, sarà il dispatcher che gestirà i nuovi thread ed assegnerà l'esecuzione dei thread a diverse CPU.

```
#include <pthread.h>
void main()
{
    pthread_t tread_handler;
    char buffer[30];
    void* function_handler(void* argt);
    (void)pthread_create(&tread_handle,NULL,function_handler,(void*)buffer);
}

void* function_handler(void* argt)
{
    printf("%s",(char*)argt);
    pthread_exit(argt);
}
```

Figura 33 - Creazione di un thread

3.3.2 Semafori

I semafori sono particolarmente utili nel gestire la concorrenza e la sincronizzazione nei thread.

Essi consistono in una variabile condivisa gestita come se fosse un descrittore di file.

Inizialmente si inizializza al numero di risorse disponibili con la system-call:

sem_init()

È possibile utilizzare i semafori per controllare l'utilizzo di risorse binarie (disponibili in numero massimo di 1) attraverso tre semplici operazioni:

- 1- set;
- 2- reset;
- 3- wait.

La prima consente di impostare il semaforo a 1 in modo da rendere la risorsa accessibile; la seconda lo imposta a 0 in modo da rendere indisponibile la risorsa; l'ultima aspetta indefinitamente finché la risorsa non è disponibile.

Le tre operazioni vengono eseguite dalle system-call:

- 1- *sem_post()*;
- 2- *sem_wait()*.

La prima incrementa il contatore del semaforo, l'ultima serve sia a decrementare il contatore del semaforo che ad attendere che si liberi.

Nei processi abbiamo utilizzato le pipe per la comunicazione di dati; nei thread la memoria globale è condivisa, quindi i dati sono già disponibili a tutti i thread quando lo desiderano, l'unica accortezza è quella di accedere solo a dati "puliti".

I semafori permettono questo.

Per sincronizzare l'accesso ai dati si controlla prima la disponibilità del dato leggendo un semaforo o aspettando che sia disponibile settando un semaforo dal thread che deve elaborare il dato.

Il listato di figura 33 crea un thread senza porsi troppi problemi; in figura 34 estendiamo il listato con l'utilizzo dei semafori.

```
#include <string.h>
#include <pthread.h>
#include <semaphore.h>
sem_t semfd;
void main()
{
    pthread_t tread_handler;
    char buffer[30];
    void* function_handler(void* argt);
    sem_init(&semfd,0,0); /* inizializza il semaforo come non disponibile */
    (void)pthread_create(&tread_handle,NULL,function_handler,(void*)buffer);
    strcpy("prova",buffer);
    sem_post(&semfd); /* ora il semaforo è utilizzabile (risorsa libera) */
}

void* function_handler(void* argt)
{
    sem_wait(&semfd); /* aspetta che venga liberata la risorsa */
    printf("%s",(char*)argt);
    sem_destroy(semfd); /* dealloca il semaforo */
    pthread_exit(argt);
}
```

Figura 34 - Thread + semafori

3.4 MPI

MPI è uno standard per il message passing molto diffusa per l'implementazione di algoritmi paralleli.

Ne esistono varie versioni, tra cui quelle implementate dai grandi distributori di sistemi paralleli: le versioni open più diffuse ed utilizzate sono sicuramente MPICH²⁰ e MPI/LAM²¹.

Queste implementazioni della libreria supportano la versione 1 di MPI ed in parte la 2; è comunque in fase beta una versione di MPICH che ne implementa le API²² in versione 2.

²⁰ <http://www-unix.mcs.anl.gov/mpi/mpich/>

²¹ <http://www.lam-mpi.org/>

²² API: Application Program Interface

La libreria in questione permette di gestire la comunicazione tra processi attraverso scambi di messaggi e di gestire la sincronizzazione degli stessi e la concorrenza, sia che i processi siano in locale sulla macchina che distribuiti su più computer, facendo funzionare di fatto i nodi di un cluster come se si trattasse di un'unica macchina.

Il message passing (da ora MP) è realizzato attraverso l'utilizzo di socket BSD, e di fatto fornisce uno strato semplificato di utilizzo di queste attraverso primitive di MP.

Per poter eseguire porzioni del programma su tutti i nodi del cluster, il programma viene lanciato su ognuno di essi utilizzando il comando *mpirun*.

mpirun è uno script shell (bash²³) che consente l'esecuzione del programma sul numero indicato di nodi:

```
mpirun -np 4 programmatest
```

In questo esempio viene eseguito *programmatest* su 4 nodi (se disponibili).

Il meccanismo di esecuzione fa uso del comando *rsh*²⁴.

Esiste anche un demone preposto al controllo ed alla collezione dell'input ed output, il server *ch_p4*.

Questo servizio intercetta l'input e l'output di qualsiasi nodo e lo redirige al nodo master, che solitamente coincide con la macchina master del cluster, essendo quest'ultima dotata di console.

Il funzionamento di questo servizio si basa su tre agenti che chiameremo:

- 1- **Console;**
- 2- **Manager;**
- 3- **Client.**

²³ Bash: bourne again shell

²⁴ rsh: remote shell

Console e' un processo creato da un comando utilizzato per gestire il cluster, questi comandi servono ad inviare segnali a tutti i processi remoti, ad eseguirli ed a controllare lo stato dei nodi o processi.

Il comando *mpirun* visto precedentemente fa parte di questi programmi.

Il demone *ch_p4mpd*²⁵ utilizza anche altri programmi di questo tipo quali:

1- *mpdtrace*;

2- *mpdallexit*.

Il primo controlla lo stato dei nodi ed il secondo termina tutti i processi distribuiti.

Questi programmi non sono utilizzati dal server *ch_p4* standard.

Il server *ch_p4mpd* consente è una riscrittura del vecchio *ch_p4* appositamente studiata per cluster composti sia da macchine omogenee che eterogenee.

Il vantaggio di utilizzare il vecchio servizio sta nella gestione di nodi multiprocessore, in quanto riesce a gestire le comunicazioni tra processori dello stesso nodo attraverso la memoria condivisa.

Il **manager** è il processo creato dallo stesso servizio *ch_p4*; ha lo scopo, oltre che di fare la collect degli input/output, di permettere l'autenticazione veloce tra i vari nodi una volta lanciato il processo, grazie ad una cache.

Il **client** è il processo che esegue il programma vero e proprio.

Il processo console viene lanciato sul nodo master e viene avviato su tutti i nodi necessari tramite *rsh*.

Il **manager** di ogni nodo intercetta i parametri passati da **console** e genera un processo figlio, il **client**, che eseguirà il programma vero e proprio.

²⁵ *ch_p4mpd*: *ch_p4* multi purpose daemon

Ora, tutto l'I/O del client viene intercettato dal **manager** del nodo ed inviato al **manager** opportuno, che in genere coincide con il nodo 0 o nodo master.

In quest'ultima fase il processo console non viene più utilizzato, finché a fine esecuzione del client viene terminato e tutte le risorse allocate vengono rilasciate.

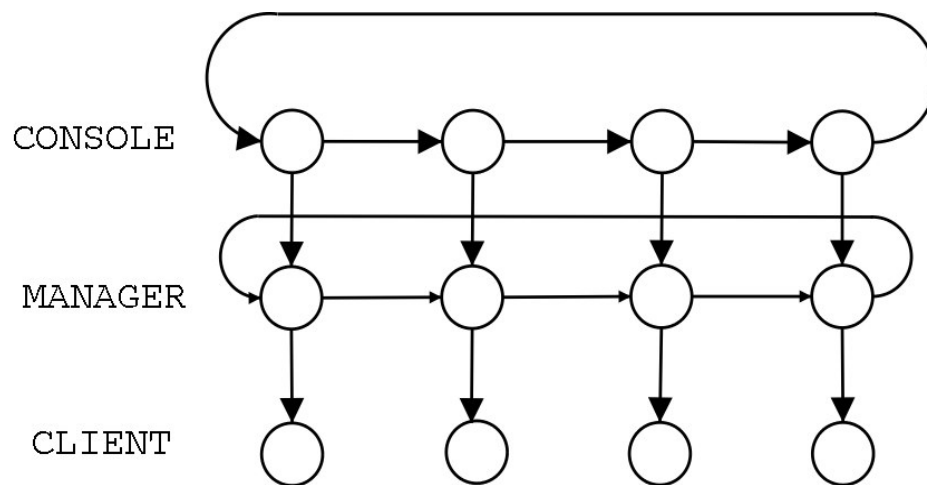


Figura 35 – Struttura esecuzione processo

Capitolo 4

Prestazioni del cluster e toy-test

Il corretto funzionamento del cluster è stato verificato mediante la suite di test mpich-test¹.

In questo capitolo invece faremo alcune considerazioni sulle prestazioni del cluster.

Per valutare le prestazioni del cluster, sono stati effettuati una serie di test su problemi giocattolo (toy test).

Questi test sono serviti per controllare la latenza del cluster nelle comunicazioni e la sua scalabilità sia all'aumentare dei processori che all'aumentare dei dati in ingresso.

La trasmissione dei dati e la divisione delle matrici risultano simili all'implementazione dell'applicazione che illustreremo nel capitolo successivo.

¹ <http://www.mcs.anl.gov/mpi/mpich/>

4.1 Valutazione prestazioni generali del cluster

Consideriamo i nodi del cluster composti da PC con processore AMD Athlon.

Queste CPU consentono di eseguire le istruzioni in pipeline (9 vie), permettendo, a regime di eseguire una istruzione per ciclo di clock.

Essendo le CPU del cluster a 1533 Mhz, esse permettono una potenza di calcolo di picco teorica di 1533 Mflops².

In totale sono presenti 8 CPU, quindi la potenza di calcolo teorica del sistema è di circa 12 Gflops.

In realtà, questo picco di prestazioni non è mai raggiunto, in quanto il sistema nel suo complesso può essere visto come un'insieme di parti legate tra loro attraverso sistemi di comunicazione che, per loro natura hanno una bandwidth limitata.

In primo luogo, il bus di comunicazione tra i processori e la memoria funziona alla velocità di 266 Mhz, mentre il bus di comunicazione con le periferiche solamente a 33 Mhz.

A tale sistema sono collegate le interfacce di rete, che sono di tipo fast ethernet nella nostra configurazione, quindi permettono di scambiare dati a circa 12 Mbyte/s tra un nodo e l'altro.

4.2 Toy test

Questi test servono per avere un'idea delle prestazioni reali del nostro sistema.

² Mflops: Milion floating points operation per second

L'utilizzo di dati per i test dello stesso ordine dei dati che dovrà elaborare il sistema reale, può dare una stima corretta delle sue potenzialità.

I toy-test forniscono inoltre le prestazioni ideali del sistema, in quanto si tratta dell'implementazione del prodotto matriciale versione ijk fortemente parallelizzabile.

Una volta che i processori hanno ricevuto i dati da elaborare (prima matrice per parte della seconda trasposta), non necessitano più di comunicazione.

Al termine dell'elaborazione tutti i processori hanno una parte di risultato che spediscono al nodo principale.

Le prestazioni sono state raggruppate per tipologia di ottimizzazione, per ognuna delle quali viene calcolata la speed-up: T_s/T_p

La dimensioni delle due matrici usate nei test sull'ottimizzazione sono rispettivamente 4000×300 float (4 byte) e 300×10000 float, indicati con $m \times n$ ed $n \times k$, per un totale di 4687 KB per la prima e 11718 KB la seconda.

Per valutare la scalabilità del cluster sono stati effettuati sia test al variare dei processori che al variare delle dimensioni delle matrici.

La scalabilità all'aumentare del numero di processori sembra abbastanza lineare sul cluster, ma si discosta dalla retta ideale a causa del tempo di trasmissione dati elevato per le dimensioni del problema.

Con i 4 nodi del cluster, l'algoritmo di prova scala ancora bene.

All'aumentare dei dati in ingresso, la scalabilità del cluster è ottima, e la speed-up tende idealmente a 8.

L'ottimizzazione dell'algoritmo mediante codice SSE su PIII raggiunge prestazioni molto vicine a quelle teoriche, con uno speed-up di 3.77, molto vicino a quello ideale di 4.

L'Athlon ha invece dei miglioramenti inferiori utilizzando lo stesso codice, ma può vantare su un'unità in virgola mobile più performante.

L'utilizzo di openMosix sul cluster invece non apporta vantaggi alle applicazioni scientifiche in generale.

Utilizzando l'IPC, i processi vengono migrati solamente quando non vi è comunicazione tra di loro e non ci sono letture/scritture su file system.

È stata testata l'esecuzione di varie applicazioni, ma le uniche che ne traggono un reale vantaggio sono quelle in cui, una volta completata la fase di spedizione dei dati ai vari processi non vi è più comunicazione.

Per le applicazioni scientifiche, in generale vi è necessità di comunicazione tra i processi durante l'esecuzione degli stessi.

Questo genera una migrazione continua dei processi tra nodo HUN ed un remote.

La mancanza di meccanismi di broadcast analogo ad MPI per le trasmissioni degli stessi dati a più nodi, rende il tutto molto lento e laborioso ed il tempo di startup ne risulta molto elevato.

<i>Architettura</i>	<i>Ottimizzazioni</i>	<i>Tempo(sec.)</i>	<i>Speed-up</i>
Intel PIII	Seriale	373	1
Intel PIII	SSE	99	3.77

Tabella 10 – Valutazione performance SSE su architettura Intel

<i>Architettura</i>	<i>Ottimizzazione</i>	<i>Tempo(sec.)</i>	<i>Speed-up</i>
AMD Athlon	Seriale	163	1
AMD Athlon	SSE	59	2.76

Tabella 11 – Valutazione performance SSE su architettura AMD

<i>Architettura</i>	<i>Ottimizzazione</i>	<i>Tempo(sec.)</i>	<i>Speed-up</i>	<i>Intel/AMD</i>
Intel PIII	Seriale	373	1	2.29
AMD Athlon	Seriale	163	1	
Intel PIII	SSE	99	3.77	1.68
AMD Athlon	SSE	59	2.76	

Tabella 12 – Valutazione performance SSE su architettura Intel e AMD

<i>Architettura</i>	<i>Ottimizzazione</i>	<i>Tempo(sec.)</i>	<i>Speed-up</i>
AMD Athlon	Seriale	163	1
AMD Athlon	SSE	59	2.76
AMD Athlon	Thread	82	1.99
Cluster	MPI (8 CPU)	43	4.94
Cluster + Mosix	IPC (8 CPU)	49	3.33

Tabella 13 – Valutazione performance SSE/SMP/MPI su architettura AMD

<i>#CPU</i>	<i>#nodi</i>	<i>Tempo spedizione(s.)</i>	<i>Tempo calcolo(s.)</i>	<i>Speed-up</i>	<i>Speed-up teorica</i>
2	1	0	82	1.99	2
4	2	8	43	3.20	4
6	3	11	30	3.97	6
8	4	12	21	4.94	8

Tabella 14 - scalabilità numero processori con cluster no master

<i>Dimensioni ($m \times n \times k$)</i>	<i>Tmpo spedizione(s.)</i>	<i>Tmpo calcolo(s.)</i>	<i>Tmpo seriale(s.)</i>	<i>Speed- up</i>	<i>Speed- up ideale</i>
$4 \cdot 10^3 \times 3 \cdot 10^2 \times 1 \cdot 10^4$	12	21	163	4.94	8.0
$4 \cdot 10^3 \times 1.5 \cdot 10^3 \times 1 \cdot 10^4$	18	102	815	6.79	8.0
$4 \cdot 10^3 \times 3 \cdot 10^3 \times 1 \cdot 10^4$	24	205	1632	7.13	8.0
$4 \cdot 10^3 \times 1.5 \cdot 10^4 \times 1 \cdot 10^4$	80	1010	8150	7.48	8.0

Tabella 15 - Scalabilità dimensioni problema utilizzando 8 CPU con cluster no master

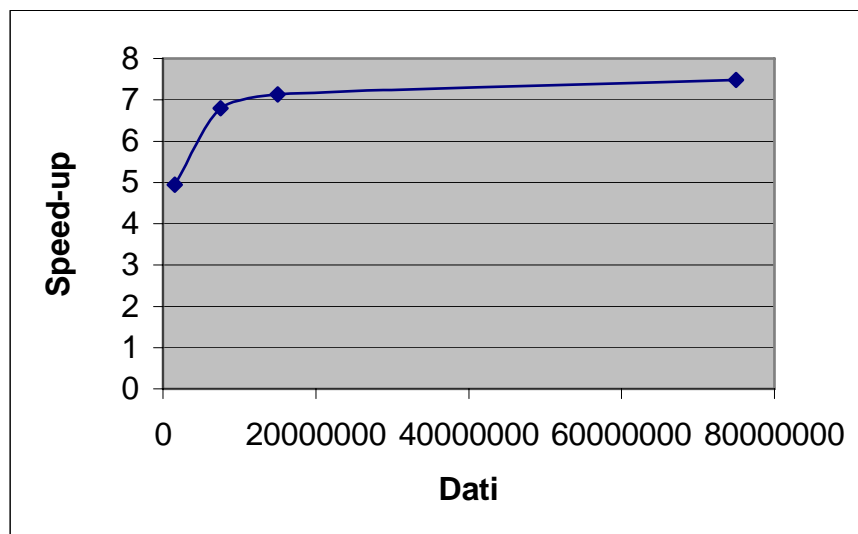


Figura 36 - Scalabilità all'aumentare delle dimensioni problema

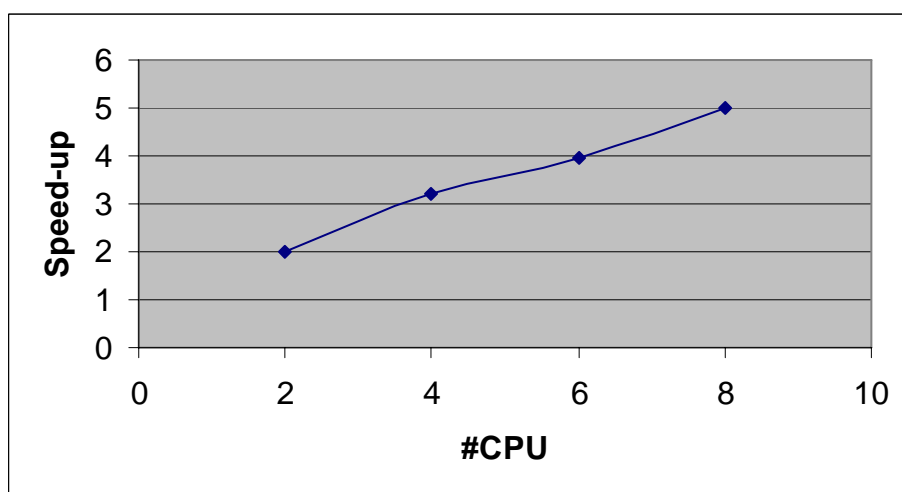


Figura 37 - Scalabilità all'aumentare delle CPU

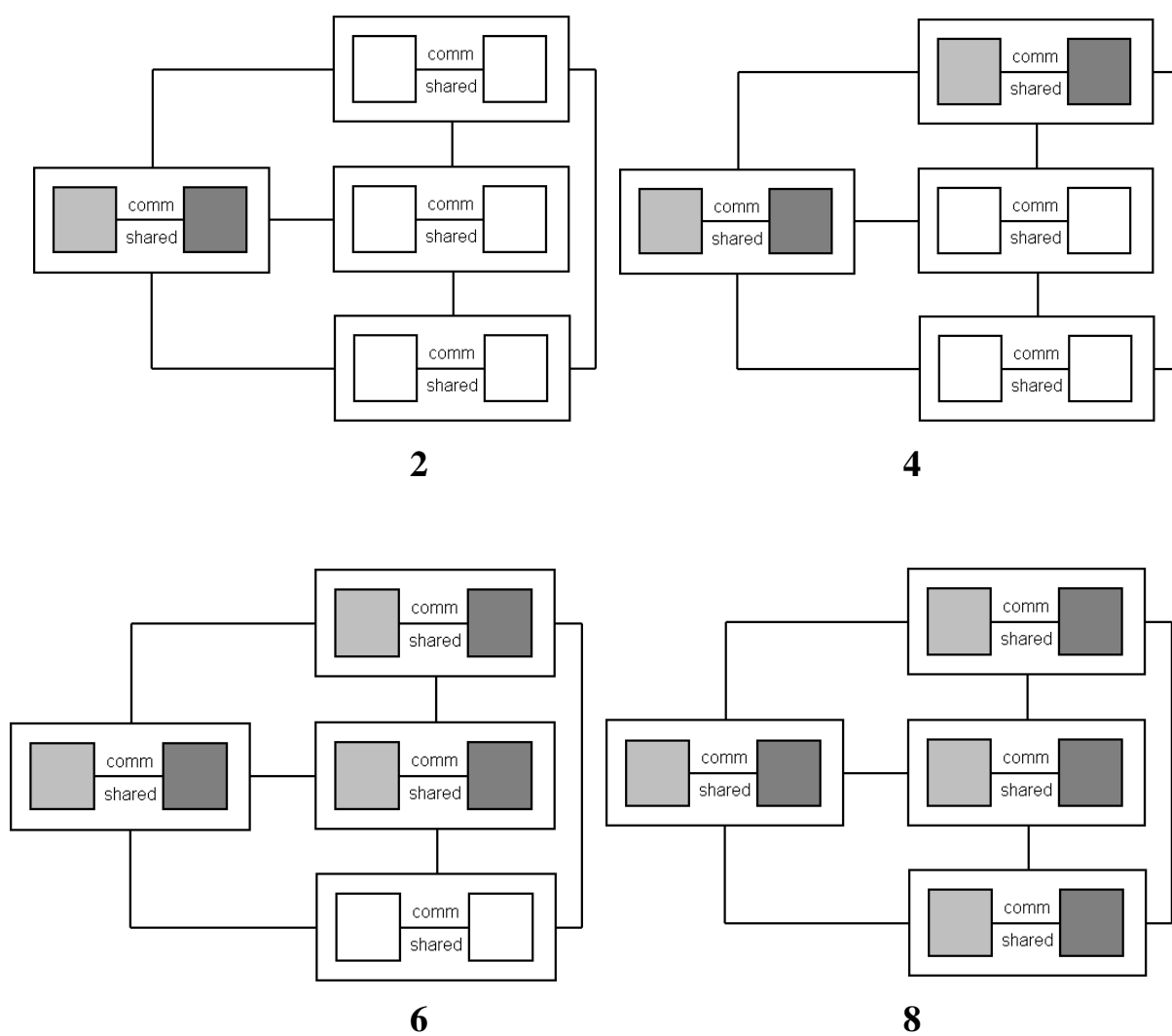


Figura 36 – Cluster no master: configurazione cluster tipo no master con in grigio le CPU attive

Capitolo 5

Applicazione reale

Viene presentata in questo capitolo una introduzione al problema che ha portato allo sviluppo dell'applicazione iniziale per poi addentrarci nei dettagli implementativi degli algoritmi utilizzati.

È analizzato il processo di ottimizzazione che ha portato all'attuale versione del programma durante il lavoro di tesi, il tutto correlato da test sulle prestazioni.

5.1 Introduzione all'applicazione

Il carcinoma mammario si colloca al primo posto in ordine di incidenza e di mortalità fra i tumori maligni che colpiscono la popolazione femminile dei paesi industrializzati.

È stato stimato che una donna, nel corso della sua vita, ha il 12% di probabilità di soffrire di tale disturbo che, se non diagnosticato e curato tempestivamente, può essere mortale [HAR92].

La malattia si può presentare in quattro diverse fasi che andremo brevemente a discutere.

La fase 1 corrisponde alla manifestazione iniziale del tumore, che solitamente presenta una lesione singola e localizzata (early o minimal breast cancer) di dimensioni che non superano il centimetro.

La fase 2 implica lesioni di dimensioni leggermente più grandi rispetto alle precedenti (solitamente dai 2 ai 5 cm).

La fase 3 si riferisce a tumori sempre circoscritti, ma ad uno stadio di sviluppo ancora più avanzato.

Infine, la fase 4 presenta metastasi a distanza dal tumore primitivo.

Appare evidente che le probabilità di successo nella lotta alla rivelazione di questi disturbi, sono strettamente legate al grado di sviluppo della lesione nel momento in cui questa viene diagnosticata.

Il metodo più semplice per esaminare il seno è la palpazione, che può essere eseguita dalla paziente stessa.

Il problema principale di questo tipo di tecnica è la difficile individuazione di patologie anche se eseguita da esperti.

Per permettere un'analisi approfondita della struttura interna della mammella sono teoricamente disponibili diverse tecniche, tra le quali ricordiamo: sonomammografia (eseguita tramite ultrasuoni), tomografia computerizzata e risonanza magnetica.

Queste tecniche presentano diversi svantaggi, tra i quali: costi elevati, alti dosaggi di radiazioni, bassa risoluzione spaziale.

La soluzione preferita rimane quella di compiere l'analisi mediante raggi-X (mammografia).

Questa tecnica produce immagini ad alto contrasto, alta risoluzione spaziale, costi relativamente contenuti e basse dosi di radiazioni.

Sono eseguite, di base, due proiezioni per ogni mammella: una lungo l'asse cranio-caudale (CC) ed una obliqua medio-laterale (MLO) per un totale di quattro radiografie per paziente.

Altre proiezioni possono essere eseguite su necessità specifiche.

Le masse tumorali si presentano come addensamenti chiari a dimensioni lineari variabili dai 3 ai 30-50 mm.

Esse sono a bordi sfumati e variabili in tipo e definizione.

Questo tipo di lesioni possono variare notevolmente in densità ottica, da casi radiolucenti (scuri nella mammografia) associati generalmente a lesioni benigne a lesioni dense con forte radiopacità, spesso maligne.

Le masse sono inoltre distinte per forma, posizione, dimensione, caratteristiche al bordo, attenuazione ai raggi X (indice di densità) ed effetti sul tessuto circostante[FRO98].

Una prima classificazione tiene conto solo di forma e di bordo ma varie combinazioni di forma e bordo sono riscontrabili.

<i>FORMA</i>	<i>BORDO</i>
Circolare	Circoscritto
Ovale	Oscurato
Irregolare	Microlobulato
Distorsione architetturale	Non definito
Asimmetrie di tessuto	spicolato

Tabella 18- Classificazione masse tumorali

La mammografia pur essendo l'esame più importante per la diagnosi del carcinoma alla mammella, non rileva tuttavia la totalità dei casi.

In media, i casi di patologia sfuggiti all'esame variano da una percentuale del 10% ad una del 30%.

La causa della mancata diagnosi può dipendere dalla particolare patologia tumorale (troppo basso contrasto intrinseco rispetto ai tessuti circostanti), dalla scarsa qualità della mammografia, dal mancato riconoscimento da parte del radiologo, o da più combinazioni di queste.

I limiti della mammografia sono particolarmente evidenti per i casi di giovani pazienti con seno denso, per le quali la presenza di numerose strutture fibrose ad alta radiopacità rende poco evidenti i segnali radiologici del tumore.

5.2 II CAD¹

Data l'importanza di una diagnosi precoce e le difficoltà di diagnosi evidenziate, è evidente l'utilità di sistemi di rivelazione automatica di patologie tumorali in immagini mammografiche che coadiuvino l'attività diagnostica del radiologo.

Tali sistemi sono riferiti come CAD.

L'obiettivo del CAD è quello di fornire un sistema di rivelazione automatica che si affianchi al radiologo nella fase di diagnosi.

Tali sistemi hanno lo specifico compito di portare all'attenzione del radiologo quelle zone dell'immagine mammografica individuate dall'algoritmo di rivelazione come sospette ovvero di potenziale anormalità.

Il programma CAD deve soddisfare alcuni vincoli fondamentali:

- 1- efficienza di rivelazione pari o superiore al 70%;
- 2- numero di falsi positivi per immagine limitato, in media meno di uno;
- 3- tempo di elaborazione comparabile con il tempo di analisi del radiologo.

Il primo punto equivale all'efficienza media del radiologo in condizioni di attenzione minimi; il secondo riguarda sempre l'affidabilità del

¹ CAD: Computer Aided Detection

sistema, in quanto un numero elevato di falsi positivi non fa considerare affidabile il sistema dal radiologo; infine l'elaborazione delle immagini deve richiedere un tempo di circa 30", dell'ordine di tempo che grossomodo impiegherebbe il radiologo.

5.3 Mammografi digitali

L'uso di un sistema CAD richiede l'utilizzo di un formato digitale dell'immagine.

Tale formato può essere ricavato in seguito alla digitalizzazione della lastra radiofotografica.

Questo passaggio intermedio, tuttavia genera rumore che dipende dalla sensibilità dello scanner, dalla sua risoluzione, dalla curva di trasferimento ecc.

Questi limiti sono stati superati dall'introduzione di mammografi digitali. Tali strumenti permettono di ottenere direttamente un'immagine digitale. Tali immagini possono essere elaborate direttamente al computer per modificarne la resa e facilitarne la diagnosi.

I sensori digitali che forniscono le immagini richiedono meno radiazioni rispetto alla lastra radiofotografica e consentono di utilizzare anche immagini non perfette rielaborandole al computer prima dell'analisi del radiologo e del CAD.

5.4 Descrizione del sistema

Lo scopo del sistema sviluppato è quello di riconoscere la presenza di oggetti di una determinata categoria (positivi) all'interno di una serie di immagini digitali fornitegli.

Il programma in questione è composto da due moduli distinti:

- 1- Estrazione delle features per la generazione dei vettori di training e apprendimento;
- 2- Riconoscimento.

I due moduli si basano su trasformate di Haar e SVM.

Il primo punto è inerente all'apprendimento della rete neurale.

Questo passo consiste nel fornire al programma di apprendimento varie immagini contenenti oggetti appartenenti alle seguenti classi:

- 1- oggetti da riconoscere (positivi);
- 2- oggetti qualunque (negativi).

Tali immagini costituiscono il training set per il motore SVM.

Dalle immagini si estraggono le features che descrivono la natura degli oggetti positivi e negativi.

Una volta istruita la rete e ottenuto il modello di interesse, lo si può utilizzare per confrontare altre immagini (di test) fornite al modulo di riconoscimento.

Per quanto possa essere lunga la fase di apprendimento, questa consiste in una procedura finita e supervisionata dall'operatore che deve controllare le immagini fornite e scegliere dei campioni significativi allo scopo del programma.

In definitiva questa operazione viene eseguita una volta soltanto, mentre la fase di riconoscimento verrà eseguita successivamente per ogni analisi delle immagini.

5.4.1 Riconoscimento

La fase di riconoscimento consiste nel mostrare al classificatore un'immagine qualsiasi che può contenere o meno oggetti della classe dei positivi.

Nell'eventualità che l'immagine contenga uno o più di questi oggetti, il sistema deve rilevarli e localizzarli a livello spaziale.

Tale fase consiste nei seguenti passaggi:

- 1- Una maschera di dimensione $n \times n$ viene fatta passare su tutta l'immagine di test. Di tale maschera viene calcolata ad ogni passo (scan) la trasformata di Haar (normale o overcomplete a seconda di quella utilizzata nel training set).
- 2- I coefficienti wavelet orizzontali, diagonali e verticali formano il vettore che SVM classifica.
- 3- L'immagine di test viene ridimensionata e si ripetono i punti 1 e 2.

Attraverso i punti 1 e 2 gli oggetti positivi vengono rilevati e localizzati spazialmente.

Il punto 3 è fondamentale per riconoscere oggetti di diverse dimensione. Anche in questo caso le immagini sono normalizzate coerentemente con il modello.

Nella fase di apprendimento gli oggetti positivi sono tutti centrati all'interno delle immagini normalizzate (di dimensione $n \times n$).

Si supponga che l'immagine contenga un positivo di dimensione più piccola o più grande di $n \times n$: ridimensionando iterativamente l'immagine di test e mantenendo invariata la dimensione della maschera è possibile rieseguire i punti 1 e 2 per rilevare a scale diverse tali oggetti.

Una porzione di programma riguarda inoltre la segmentazione delle immagini.

La segmentazione non è necessaria, ma presenta dei vantaggi non indifferenti.

Segmentando l'immagine, si estrae la regione di interesse contenente il seno, eliminando tutto il rumore esterno ad esso.

Le immagini contenute nel database di lavoro contengono una serie di etichette che riportano alcune informazioni sulla natura della radiografia.

Tali etichette non appartengono all'immagine da analizzare, ma vengono comunque utilizzate se non viene effettuata la segmentazione.

Le immagini analogiche sono digitalizzate ad altissima risoluzione.

La scansione dell'immagine comporta tempi di calcolo molto elevati e limitarsi a controllare sotto scansione solo la parte di interesse dimezza o più i tempi di scansione.

La segmentazione genera, a partire da un'immagine, un file che contiene le coordinate della regione di interesse.

Questo file è utilizzato sia dal modulo di riconoscimento che da quello di apprendimento.

L'algoritmo sopra descritto viene quindi eseguito solo sulla parte di immagine segmentata.

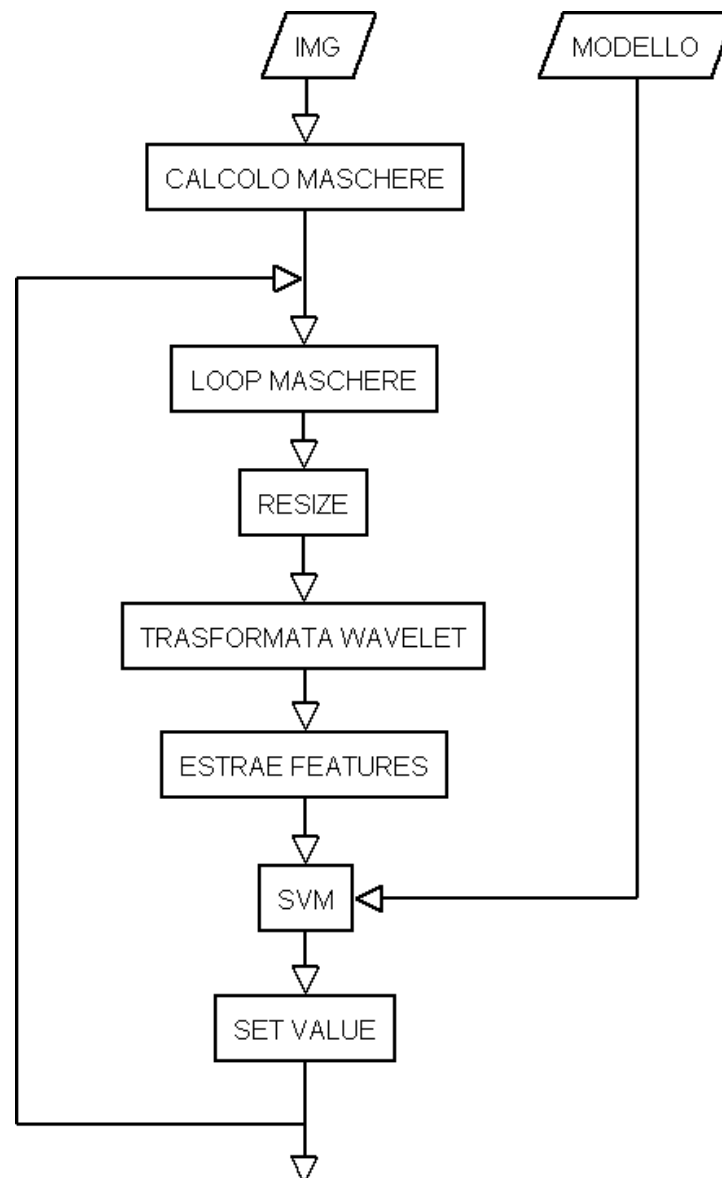


Figura 38 - Schema generale dell'applicazione

5.5 Ottimizzazione multilivello dell'applicazione

Analizziamo nel dettaglio le ottimizzazioni apportate al progetto iniziale.

Tali ottimizzazioni hanno portato l'elaborazione media di un'immagine dai circa 117" iniziali a 14" di tempo, un miglioramento delle prestazioni di un ordine di grandezza.

.

Queste ottimizzazioni comprendono:

- 1- MPI;
- 2- Thread;
- 3- SSE.

Poiché i cluster di tipo Beowulf si basano su una struttura master/slave, si è deciso di utilizzare un processo master risiedente sulla macchina master del cluster a cui sono assegnati i calcoli seriali, come la divisione dell'immagine per l'assegnazione ai nodi di calcolo e distribuzione dei dati.

Inoltre questo processo raccoglie i dati degli altri nodi e li ricompone.

In questo modo il master, che gestisce tutti i servizi del cluster, non vengono eseguiti calcoli, lasciando questo compito agli altri nodi che sono del tutto scarichi e dedicati allo scopo; inoltre eliminando il master dal calcolo si evita di creare un collo di bottiglia per il calcolo, in quanto dovendo gestire anche la logica di controllo, il master potrebbe non essere allineato con il resto dei nodi per fornire la sua soluzione.

Il modello viene spedito a tutti i nodi di calcolo, mentre la matrice delle features viene divisa per il numero di nodi di calcolo in righe uguali.

Le righe rimanenti, se necessario, vengono ridistribuite fra i nodi o semplicemente assegnate ad un unico nodo.

Lo scheduler utilizzato per il bilanciamento del calcolo sui nodi è statico e deciso a priori.

La non ridistribuzione tra i nodi delle righe rimanenti della matrice non incide in modo apprezzabile sull'algoritmo, soprattutto quando il numero di nodi è basso, ed inoltre il calcolo dovuto alle righe in eccesso è molto piccolo rispetto al calcolo complessivo.

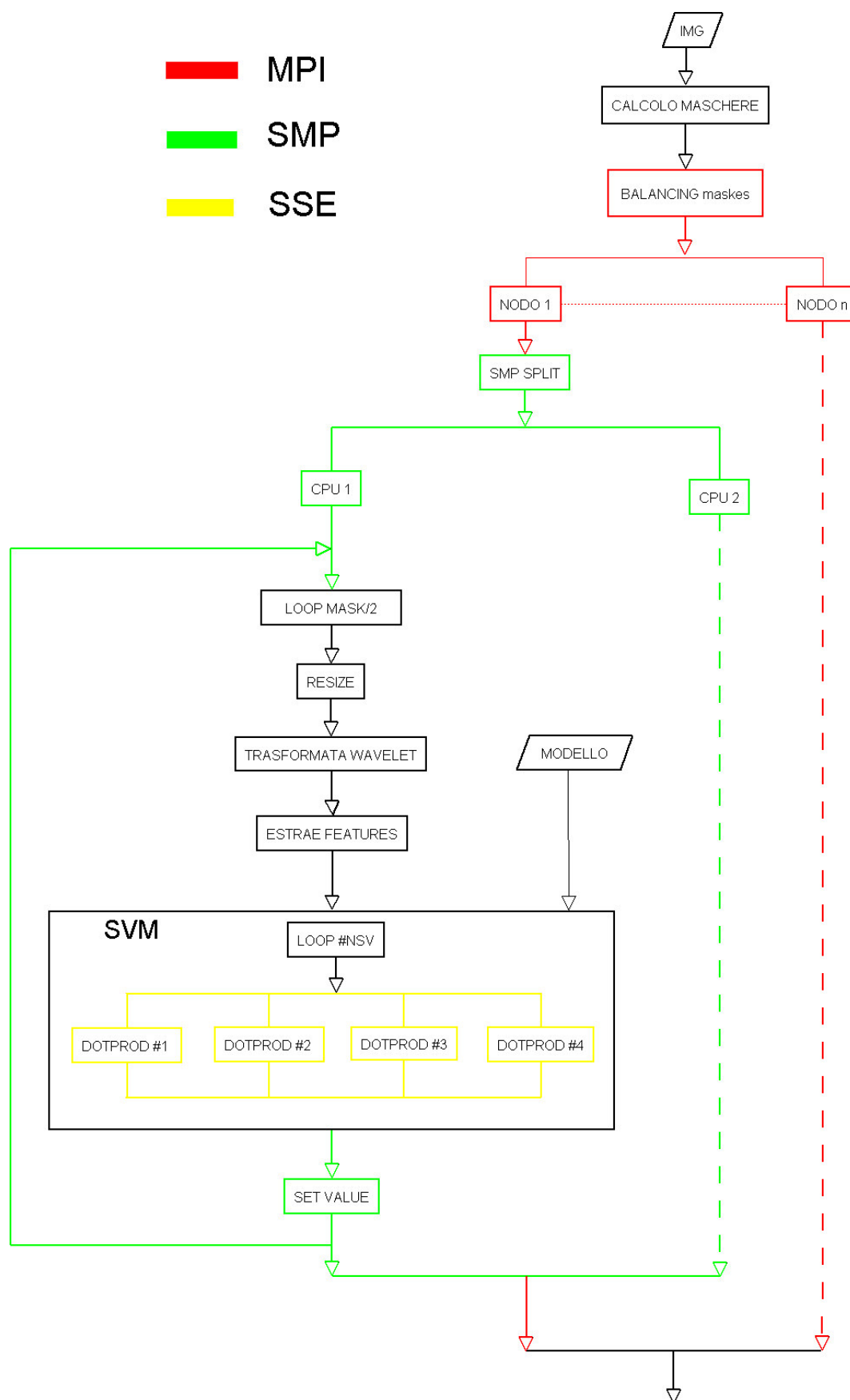


Figura 39- Schema dell'applicazione con tutti i livelli di ottimizzazione

Per quanto riguarda la divisione in thread, questa viene effettuata sul nodo di calcolo, una volta ricevuti tutti i dati necessari e prima di eseguire il calcolo vero e proprio.

Essendo dotato di 2 CPU, ogni nodo trae beneficio dall'esecuzione in due thread paralleli distinti, il tempo di startup è molto basso per i thread avendo già tutti i dati in memoria locale.

Se fosse stata scelta l'opzione inversa, sarebbero state necessarie il doppio delle comunicazioni.

Se l'applicazione funziona in modalità singola CPU e non in cluster, la parte di startup descritta viene soppressa e tutto il calcolo viene effettuato dal nodo master nel nostro caso.

Va comunque ricordato, che se l'applicazione viene eseguita in questa modalità su un nodo preposto al calcolo, invece del nodo master, l'applicazione può girare virtualmente su qualsiasi dei nodi grazie all'implementazione di openMosix.

In genere l'applicazione sarà eseguita sul nodo più scarico, diverso dal nodo sul quale viene lanciata l'applicazione se il costo della migrazione è conveniente.

Nel caso in cui si volesse utilizzare solo l'ottimizzazione a thread, viene eliminata la parte iniziale di setup per l'allocazione statica delle risorse mediante MPI, ma il procedimento di parallelizzazione dell'algoritmo rimane sostanzialmente lo stesso.

È anche possibile non utilizzare l'ottimizzazione del cluster con MPI senza i thread non eseguendo la creazione di un thread secondario per la seconda CPU.

In quest'ultima ipotesi si potrebbero verificare 2 situazioni:

- 1- Il cluster utilizza tutte le CPU;
- 2- Il cluster utilizza solo una CPU per nodo.

Nella prima situazione si ottiene un risultato simile allo splittamento in thread del processo prima della ricezione dei dati, con il raddoppio delle comunicazioni ed un overhead dovuto in maggior parte alla spedizione di due matrici del modello per nodo invece di una unica ed al maggior numero di socket aperte per le comunicazioni.

È comunque possibile compilare MPI utilizzando per il message passing tra processori di uno stesso cluster memoria condivisa.

L'ultima ottimizzazione apportata è quella SSE.

Questa viene applicata alla parte relativa alla SVM.

In pratica questa parte consiste in una moltiplicazione tra matrici, nel dettaglio tra la matrice delle features e dell'immagine.

La scomposizione in pseudocodice della procedura relativa a SVM:

```

for i=0 to #SV-1
  for j=0 to #features-1
    result[i*#SV+j*#nfeatures]=0;
    for k=0 to #masks-1
      result[i*#SV+j*#nfeatures]+= \
model[i*#SV+k*#masks]*image[j*#features+k*#masks];

```

Figura 40 – Scomposizione classificatore SVM

diventa:

```

for i=0 to #SV-1
  for j=0 to #features-1
    result[i*#SV+j*#nfeatures]=0;
    < sse dot prod >

```

Figura 41 – Scomposizione classificatore SVM modificato con SSE

La matrice dell'immagine è memorizzata per colonne, mentre quella del modello per righe.

Attraverso SSE è possibile implementare il prodotto matriciale in modo molto ottimizzato; in particolare, viene spezzato il ciclo relativo al prodotto scalare tra le righe della prima matrice per le righe della seconda ed implementato interamente in assembler con istruzioni SSE.

Per sfruttare tutti i registri disponibili (8) viene diviso il ciclo interno in altri cicli che comprendono il prodotto scalare tra sottovettori di dimensione 32, in totale sono $\lceil \#masks/32 \rceil$.

All'interno di ognuno di questi cicli, vengono eseguite 4 moltiplicazioni per volta, in totale 8 operazioni invece di 32.

Questi valori poi vengono sommati, ancora una volta eseguendo 4 addizioni alla volta, in totale altre 7 operazioni.

Alla fine rimangono 4 soluzioni parziali da sommare serialmente tra di loro ed alla somma totale.

Le addizioni totali all'interno del ciclo diventano così 11 invece di 31, il fattore di risparmio diventa quindi:

$$f = \lceil 32 + 31/8 \rceil + 11$$

Considerando l'architettura Pentium, che permette l'esecuzione di una istruzione per ciclo di clock, si ottiene un incremento di prestazioni ottimali di più di un fattore 3.

Se si utilizza quest'ultima ottimizzazione è necessario riempire gli ultimi valori della matrice con degli '0' (zero padding).

Il motivo è quello di mantenere allineata la memoria per righe e di avere un multiplo del numero di elementi massimi da poter moltiplicare contemporaneamente.

Se non venisse effettuata l'operazione di “padding”, otterremmo dei valori errati, in quanto, l'ultima moltiplicazione parallela verrebbe fatta tra gli ultimi valori delle matrici e dei valori in memoria sporchi.

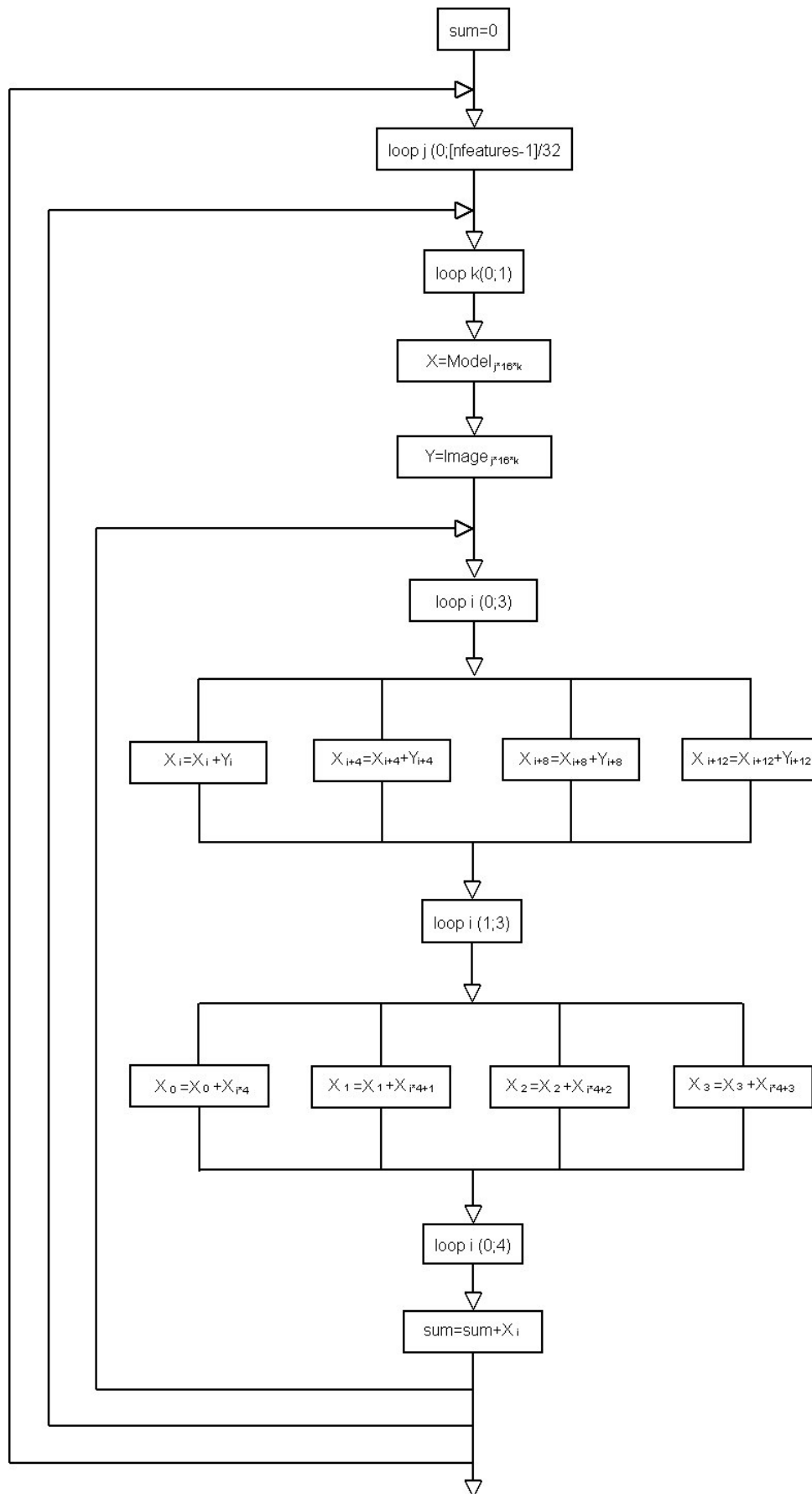


Figura 42 - Schema della parte di calcolo SSE

5.6 Test e valutazioni finali

I test che verranno presentati sono stati effettuati su un campione casuale di 100 immagini.

Tutti i test sono stati ripetuti tre volte per ogni combinazione di ottimizzazione.

Il risultato considerato è la media delle tre prove.

L'algoritmo ottimizzato scala molto bene sul cluster.

Fino a 8 processori si ottiene ancora un incremento di performance apprezzabile, anche se la speed-up risulta essere quasi la metà di quella ideale.

Il risultato peggiora se si considera come tempo seriale quello ottenuto con l'utilizzo di SSE.

Nel complesso si può comunque dire che si ha un buon parallelismo, come infatti si può notare dai grafici di figura 46 e 47, si ha un aumento quasi lineare della speed-up all'aumentare dei processori.

L'utilizzo dei thread raddoppia praticamente le prestazioni, in accordo con le aspettative.

Purtroppo, non è possibile utilizzare più di 2 processori per nodo, quindi non si ha un'idea della scalabilità dell'algoritmo in sistemi SMP con più di 2 processori.

Confrontando il cluster in configurazione B (ogni nodo viene visto da MPI come un monoprocesso ed utilizzando i thread si sfrutta il SMP) con il cluster in configurazione C (MPI vede tutti i processori presenti sul nodo ed utilizza una porzione di memoria condivisa per dialogare tra processori in SMP), si ottengono risultati sovrapponibili.

Questo si può imputare all'implementazione ottimizzata di MPI per nodi che utilizzano memoria condivisa (**ch_p4** in configurazione **comm shared** per SMP).

Sembrerebbe che all'aumentare dei nodi le performance migliorino nella configurazione B, ma il numero di processori non è tale da poter avvalorare l'ipotesi.

Discorso a parte merita SSE.

La parte di codice è stata scritta in assembler per architettura Intel e riguarda solo una parte dell'algoritmo, quello del classificatore SVM.

Rispetto ai toy-test si ha un incremento di prestazioni più esiguo, dell'ordine del 40% su Intel, contro il 33% su Athlon.

Questo gap tra Intel e AMD, che è stato messo in risalto anche nei toy-test, è dovuto a:

- 1- migliori prestazioni dell'unità floating point dell'Athlon rispetto al PIII;
- 2- miglior sfruttamento di codice SSE da parte di processori Intel.

Confrontando i valori ottenuti con l'ottimizzazione SSE attiva, si nota un incremento di prestazioni costante sul tempo di calcolo totale all'aumentare dei processori.

Nelle seguenti tabelle viene presentato, oltre al tempo di calcolo, anche la speed-up dell'algoritmo rispetto all'esecuzione seriale dello stesso.

Con +1 viene indicato il nodo master, che esegue meno calcoli degli altri nodi ma esegue gran parte dello start-up dell'algoritmo e la divisione equa dei dati, quindi della distribuzione di lavoro tra nodi (load-balancing).

I processori Athlon hanno una frequenza di 1533 Mhz, mentre i PIII hanno una frequenza di 1000 Mhz.

Il calcolo della speed-up è stato effettuato sia confrontando l'algoritmo con la versione seriale che confrontandolo con la versione ottimizzata con SSE eseguito sempre su un processore (colonna speed-up SSE).

<i>Architettura</i>	<i>SSE</i>	<i>#CPU</i>	<i>Tempo (sec.)</i>	<i>Speed-up Ts/Tp</i>
Intel PIII		1	11706	1
Intel PIII	X	1	8304	1.41

Tabella 19 – Valutazione performance SSE su architettura Intel

<i>Architettura</i>	<i>SSE</i>	<i>#CPU</i>	<i>Tempo (sec.)</i>	<i>Speed-up</i>
AMD Athlon		1	6509	1
AMD Athlon	X	1	4905	1.33

Tabella 20 – Valutazione performance SSE su architettura AMD

<i>Architettura</i>	<i>SSE</i>	<i>#CPU</i>	<i>Tempo (sec.)</i>	<i>Speed-up</i>	<i>Intel/AMD</i>
Intel PIII		1	11706	1	1.80
AMD Athlon		1	6509	1	
Intel PIII	X	1	8304	1.41	1.06
AMD Athlon	X	1	4905	1.33	

Tabella 21 – Valutazione performance SSE su architettura Intel e AMD

<i>#nodi</i>	<i>#CPU</i>	<i>Tempo (s.)</i>	<i>Speed-up SSE</i>
1	2+1	3628	1.79
2	4+1	2114	2.65
3	6+1	1613	3.04
4	8+1	1379	3.56

Tabella 22 – Valutazione performance SSE su cluster B al variare delle CPU

<i>Architettura</i>	<i>SSE</i>	<i>SMP (Thread)</i>	<i>MPI (4+1 nodi)</i>	<i>#CPU</i>	<i>Tempo (sec.)</i>	<i>Speed -up</i>	<i>Speed- up SSE</i>
AMD Athlon				1	6509	1	
AMD Athlon	X			1	4905	1.33	1
AMD Athlon		X		2	3540	1.84	
AMD Athlon	X	X		2	2740	2.37	1.79
Cluster A			X	4+1	2256	2.88	
Cluster A	X		X	4+1	1850	3.52	2.65
Cluster B		X	X	8+1	1582	4.11	
Cluster B	X	X	X	8+1	1379	4.72	3.56

Tabella 23 – Valutazione performance SSE/SMP/MPI su architettura AMD

<i>#nodi</i>	<i>#CPU</i>	<i>Tempo (s.)</i>	<i>Speed-up</i>
1	2+1	3628	1.79
2	4+1	2114	3.08
3	6+1	1613	4.03
4	8+1	1379	4.72

Tabella 24 - Prestazioni cluster tipo B

<i>#nodi</i>	<i>#CPU</i>	<i>Tempo (s.)</i>	<i>Speed-up</i>
1	2+1	3575	1.82
2	4+1	2114	3.08
3	6+1	1650	3.94
4	8+1	1435	4.53

Tabella 25 - Prestazioni cluster tipo C

<i>#nodi</i>	<i>#CPU</i>	<i>Speed-up ideale</i>	<i>Speed-up B</i>	<i>Speed-up C</i>	<i>SP(B)- SP(C)</i>
1	2+1	2.00	1.79	1.82	-0.03
2	4+1	4.00	3.08	3.08	0
3	6+1	6.00	4.03	3.94	0.07
4	8+1	8.00	4.72	4.53	0.19

Tabella 26 – Valutazione speed-up al variare dell'architettura del cluster

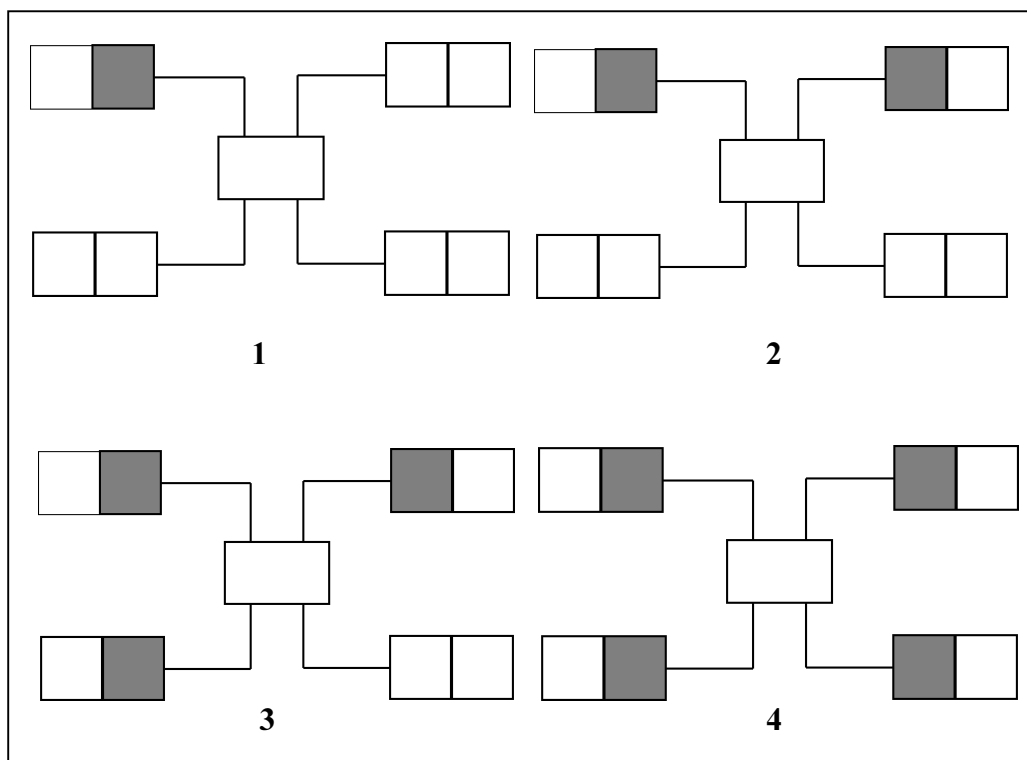


Figura 43 – Cluster A: configurazione cluster tipo A con in grigio le CPU attive

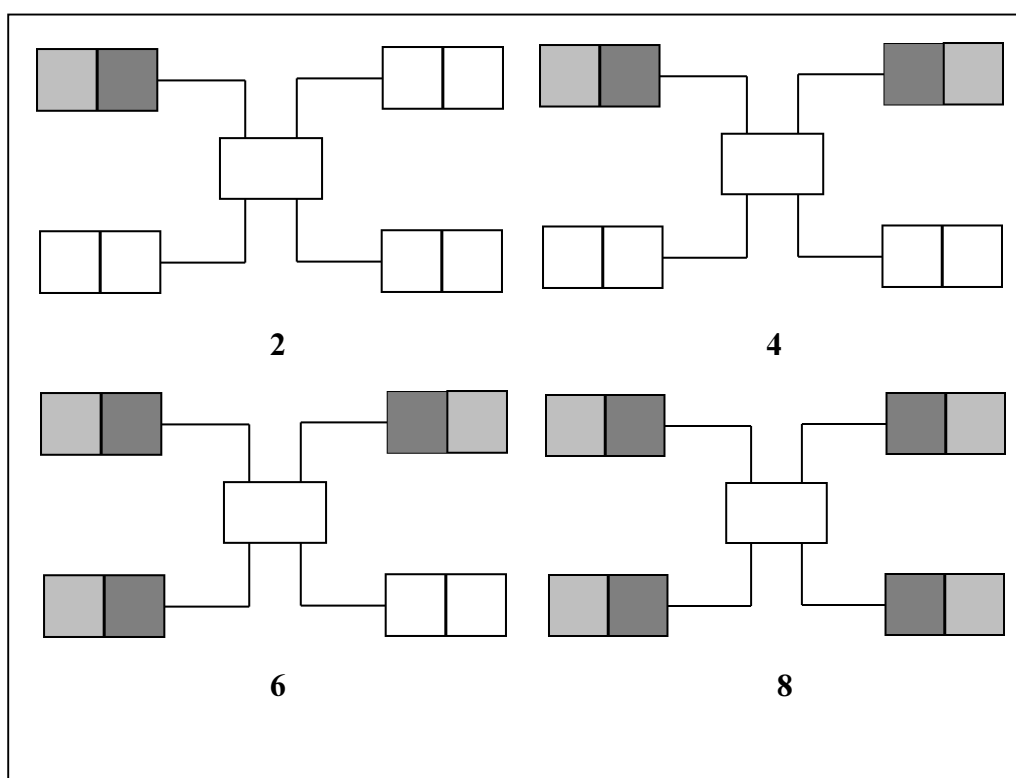


Figura 44 – Cluster B: configurazione cluster tipo B con in grigio le CPU attive

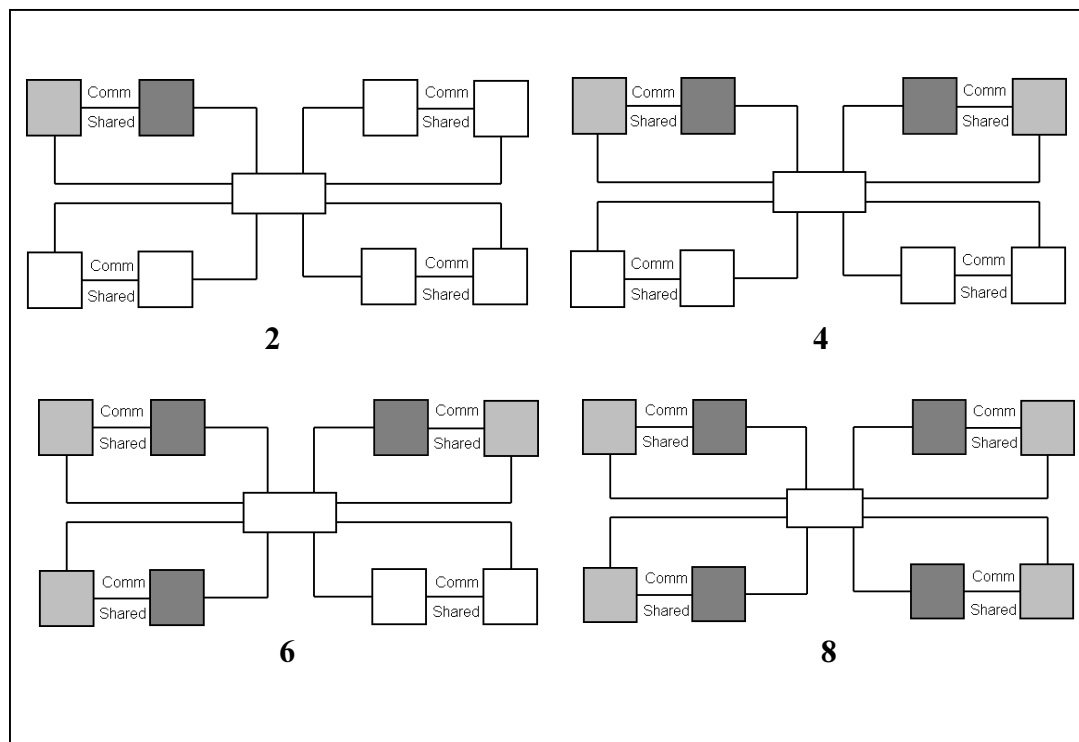


Figura 45 – Cluster C: configurazione cluster tipo C con in grigio le CPU attive

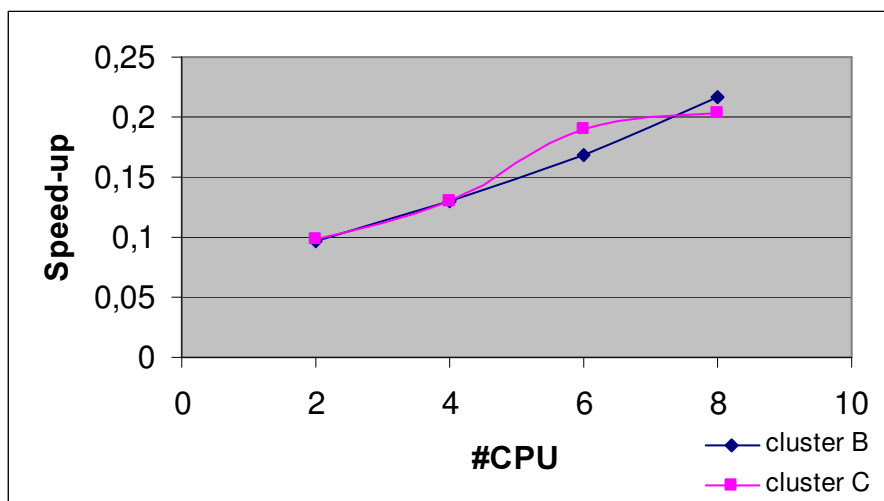


Figura 46 - Differenze di performance tra cluster B e cluster C

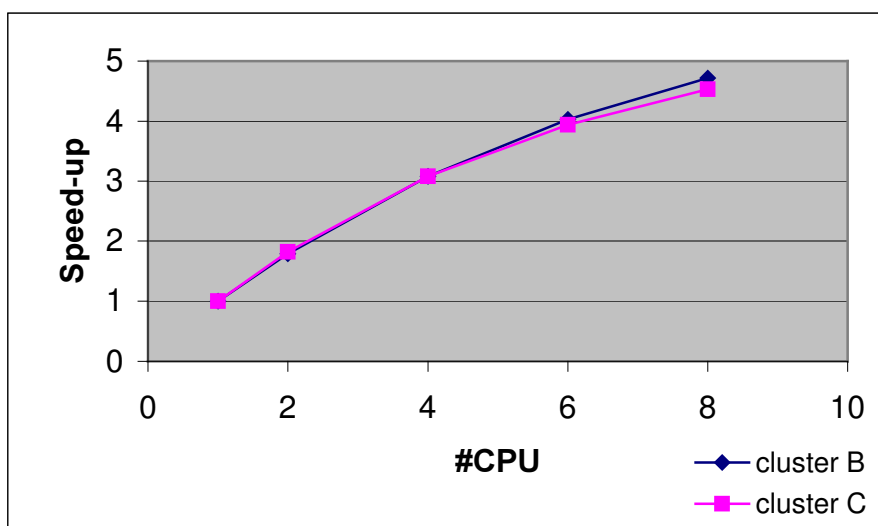


Figura 47 – Speed up relativa a SSE su Cluster B al variare delle CPU

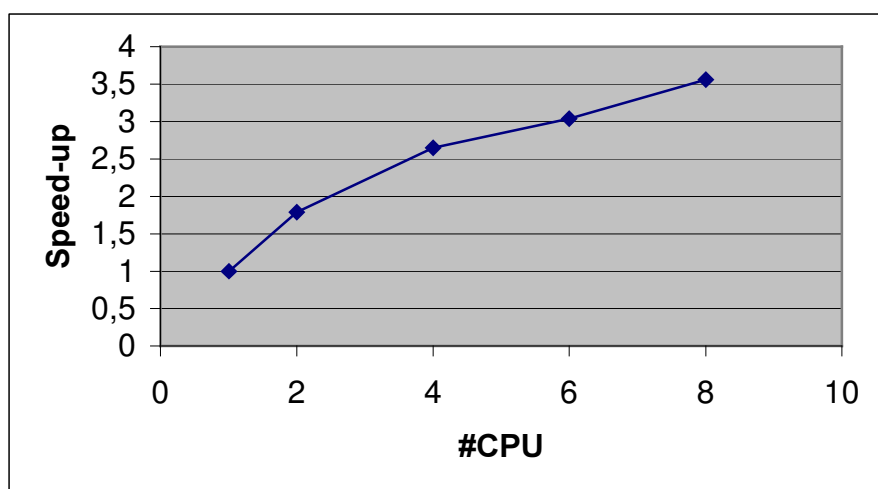


Figura 48 – Speed up relativa a Cluster B al variare delle CPU

5.7 Sviluppi futuri

Il lavoro di questa tesi è stato incentrato sull'ottimizzazione di algoritmi e sull'implementazione di soluzioni HW e SW per il calcolo ad alte prestazioni.

L'algoritmo che è stato modificato, ha raggiunto un livello di ottimizzazione molto elevato grazie alle tecniche utilizzate ed alla sua esecuzione distribuita.

Per migliorare ancora il suo sviluppo si possono seguire 3 strade:

- 1- ottimizzazione ulteriore del cluster;
- 2- porting dell'applicazione su HW embedded;
- 3- ottimizzazione ulteriore del codice.

Naturalmente una strada non necessariamente esclude un'altra.

Per l'ottimizzazione del cluster è possibile modificare la priorità dello scheduler del SO, cambiarne il tipo ed aumentare il timeout per la chiamata allo scheduler round-robin, in modo da preferire alla responsività dei nodi, prestazioni più brillanti sul calcolo.

Il porting dell'applicazione su HW embedded, permetterebbe di risparmiare denaro in quanto si potrebbero utilizzare soluzioni altamente integrate e dedicate ed inoltre si otterrebbe anche un miglioramento della gestione dell'unità di calcolo.

Mi riferisco in particolare all'implementazione su HW DSP, molto simile come programmazione alle istruzioni SWAR utilizzate.

Il vantaggio di utilizzare tali strumenti risiede nella loro ampia disponibilità sul mercato ed al loro costo molto contenuto.

Utilizzare invece un'altra architettura simile, quale Motorola Altivec congiuntamente a processori PowerPC, permetterebbe di implementare cluster, anche con HW embedded con una dissipazione di calore inferiore

e prestazione altrettanto brillanti ad un costo comunque superiore sia alle soluzioni basate su DSP che all'attuale soluzione di cluster iA32.

Infine si potrebbe raggiungere un incremento di prestazioni teorico del 100% implementando con codice SSE anche la parte relativa alla trasformate wavelet con funzione base di Haar.

Questa parte di codice è computazionalmente molto onerosa, ma beneficia al momento delle ottimizzazioni per SMP e per i cluster.

In alternativa ad SSE si potrebbe effettuare il porting di tutti i calcoli eseguiti, sia floating point che SSE su SSE2, presente su AMD Athlon ed Intel P4 di nuova generazione.

Queste istruzioni hanno tutti i vantaggi delle istruzioni SSE, ma permettono di utilizzare la stessa precisione dei calcoli in virgola mobile dell'unità floating point.

Conclusioni

L'ottimizzazione di algoritmi è un lavoro meticoloso e metodico.

Un codice, come si è visto, può essere ottimizzato a vari livelli.

L'ottimizzazione è un lavoro non portabile, ogni architettura ha una sua particolare peculiarità.

Ottimizzare un codice per cluster non serve a nulla se poi il codice verrà eseguito su un'unica macchina o su macchine SMP, allo stesso modo le istruzioni SIMD non sono portabili su tutte le architetture, ma il codice va riscritto per ogni architettura.

Lo stesso sistema operativo e compilatore danno luogo a percorsi di ottimizzazione diversi.

L'utilizzo di tecnologie GNU e open source permette di evitare in parte i problemi dovuti alle diverse architetture, fornendo strumenti di programmazione omogenei.

La disponibilità di tali strumenti in aggiunta a software per il clustering ed il calcolo parallelo permettono di ottenere soluzioni performanti e scalabili su piattaforme Unix/Linux; scalabili sia sotto il punto di vista delle performance che sotto il punto di vista della compatibilità.

Gli strumenti analizzati in questo lavoro di tesi si possono, e sono stati utilizzati, sia su mainframe che su PC.

L'unica necessità imposta dal mercato è quella di garantire la compatibilità del codice su sistemi Microsoft, data la loro enorme diffusione.

In parte questa limitazione è risolta dalla disponibilità di software gratuito che riguarda la maggior parte dei tool di programmazione utilizzati in questo lavoro di tesi.

L'ottimizzazione estrema di codice, congiuntamente all'utilizzo di cluster Linux basati su tecnologia Intel, consente l'utilizzo di tecnologie a basso costo e ad alte prestazioni idonee all'utilizzo in ambito medico, dove i tempi di elaborazione, dovuti al processing di immagini di grandi dimensioni, risulterebbe proibitivo per i sistemi basati su PC, e l'utilizzo di supercomputer dedicati ne renderebbe proibitivo il costo.

Per quel che concerne la nostra applicazione, le stesse parti di codice implementate con l'utilizzo di SSE, possono essere portate facilmente su processori DSP, ottenendo un risparmio in denaro ancora maggiore dell'utilizzo di comuni PC.

La parte di codice per lo startup può essere agilmente gestita da un PC, mentre il porting su SWAR della parte di calcolo relativa alla funzione di Haar ed alle wavelet permetterebbe di ottenere un sistema più integrato (di tipo embedded) e ad un costo inferiore.

In generale, quasi tutte le applicazioni per il processing di segnali digitali, immagini o suono che siano possono essere implementate da un DSP.

Uno sviluppo interessante potrebbe essere quello di un porting dell'intero codice su architettura SSE/SSE2 o simili.

Data la complessità di scrivere codice assembler, e la complessità stessa dei programmi attuali, quest'ultima scelta sarebbe realizzabile solamente se il compilatore stesso eseguisse un'ottimizzazione in questa direzione, come il compilatore Intel, disponibile per Windows e Linux.

Infine, per ottimizzare le prestazioni di un cluster Linux-based, può essere fatto un tuning molto approfondito della compilazione del kernel del sistema.

Bibliografia

- [1] AMD “*AMD Athlon Processor x86 Code Optimization Guide*”
<http://www.amd.com/>
- [2] AMD “*AMD Athlon Processor Model 4 Data Sheet*”
<http://www.amd.com/>
- [3] AMD “*AMD Athlon Processor Processor Module Data Sheet*”
<http://www.amd.com/>
- [4] AMD “*AMD Extension to the 3dNow! And MMX Instruction Sets Manual*”
<http://www.amd.com/>
- [5] Moshe Bar, Stefano Cozzini, Maurizio Davini, Alberto Marmodoro
“*openMosix vs Beowulf: a case study*”
INFN Democritos, openMosix Project, Department of physics
University of Pisa
- [6] Moshe Bar “*Linux file system*”
Mc Graw Hill
- [7] C. M. Bishop “*Neural networks for pattern recognition*”
Oxford University press

- [8] David Bond “*Compile/Run-time Support for Thraded MPI Execution on Multiprogrammed Shared Memory Machines*”
University of California, Santa Barbara
- [9] Andrea Dell’Amico, Mitchum Dsouza, Erwin Embsen, Peter Eriksson “*Linux NIS(YP)/NIS+/NYS HOWTO*”
<http://www.linuxdocs.org/>
- [10] A. Doria, R. Esposito, P. Mastroserio, F. M. Taurino, G. Tortone
“*Configurazione di una farm Linux con Mosix e ClusterNFS*”
- [11] R. Droms “*Dynamic Host Configuration Protocol*”
RFC 2131 - Network Working Group
Prentice Hall International edition
- [12] L. Fereira, G. Kettmann, A. Thomasch, E. Silcocks, J. Chen,JC. Daunois, J. Ihamo, M. Harada,S. Hill,W.Bernocchi,E Ford “*Linux HPC Cluster Installation*”
IBM Redbooks <http://www.ibm.com/redbooks/>
- [5] George Gousios “*Root over NFS – Another Approach*”
University of Aegean, Greece
- [13] Willan Gropp, Ewing Lusk “*Installation guid and user’s guide to MPICH a Portable implementation of MPI*”
<http://www-unix.mcs.anl.gov/mpi/mpich/>

-
- [14] Intel “*IA-32 Intel Architecture Software Developer’s Manual volume 1: Basic Architecture*”
<http://www.intel.com/>
- [15] Intel “*Intel Architecture Optimization Manual*”
<http://www.intel.com/>
- [16] Nicolai Langfeldt “*NFS HOWTO*”
<http://www.linuxdocs.org/>
- [17] Paolo Mastroserio, Francesco Maria Taurino, Gennaro Tortone
“*MOSIX: High performance Linux farm*”
INFN Italy
- [18] James McQuillan “*LTSP – Linux Terminal Server Project – v3.0*”
<http://www.ltsp.org/>
- [19] Corrado Mencar “*Analisi statistica dei processi di apprendimento*”
Università degli studi di Bari
- [20] N. Petrick, B. Petrick, B. Sahiner, H.-P. Chan, M. A. Helvie, S. Paquerault, L. M. Hadjiiski “*Breast Cancer Detection: Evaluation of a Mass-Detection Algorithm for Computer-aided Diagnosis – Experience in 263 Patients*”
Radiology – July 2002, Ed. Petrick et al

- [21] Quality Assurance Coordinating Group “*Computer aided detection in mammography*”
NHSBSP Publication No 48 – January 2001

- [22] Daniel Robbins “*Linux clustering with Mosix*”
<http://www.ibm.com/>

- [23] Michael L Schmit “*Il Manuale Pentium*”
Mc Graw Hill

- [24] Kai Shen, Hong Tang, Tao Yang “*Adaptive Two-level Thread Management for Fast MPI Execution on Shared Memory Machines*”
University of California, Santa Barbara

- [25] Kai Shen, Hong Tang, Tao Yang “*More than you ever wanted to know about GCC, GAS and ELF*”
Share 98 Session 8131

- [26] B. A. Shirazi, A. R. Hurson, K. M. Kavi “*Scheduling and balancing in parallel and distributed system*”
IEEE Computer Society Press

- [27] Silberschatz, Galvin, Gagne “*Applied operating system concepts*”
John Wiley & sons, INC

- [28] Norm Snyder “*IBM Linux Cluster*”
IBM@server White Paper <http://www.ibm.com>

- [29] Hal Stern “*Managing NFS and NIS*”
O’Reilly Assodiated INC

- [30] Hong Tang, Tao Yang “*Optimizing Threaded MPI Execution on SMP Clusters*”
University of California, Santa Barbara

- [31] A. Tanenbaum “*Computer networks*”
Pretice Hall International edition

- [32] A. Tanenbaum “*Modern operating system*”

- [33] A. Tanenbaum “*Distribuited operating system*”
Pretice Hall International edition

- [34] L. Tarassenko “*A guide to neural computing application*”
Arnold

- [35] Chtistopher Torrence, Gilbert P. Compo “*A Pratical Guide to Wavelet Analisis*”
University of Colorado

- [36] *ClusterNFS homepageI*
<http://clusternfs.sourceforge.net/>

- [37] “*FreeBSD Handbook*”
<http://www.freebsd.org/>

- [38] *MPICH homepage*
<http://www-unix.mcs.anl.gov/mpi/mpich/>

- [39] *openMosix Project*
<http://www.openmosix.org/>

- [40] *TMPI homepage*
<http://www.cs.ucsb.edu/projects/tmpi/>