Beowulf HOWTO

Jacek Radajewski and Douglas Eadline v1.1.1, 22 Novembre 1998

Questo documento introduce l'architettura Beowulf per Supercomputer e fornisce informazioni di base sulla programmazione parallela, insieme a link ad altri documenti più specifici e pagine web. Documentazione tradotta dall'HackLab di Firenze (*hacklab@firenze.linux.it*).

Contents

1 Introduzione

1.1 Liberatoria

Non ci possiamo assumere alcuna responsabilità per qualsiasi informazione sbagliata contenuta in questo documento, né per qualsiasi danno provocato dal suo uso.

1.2 Copyright

1.3 Note su questo HOWTO

Jacek Radajewski iniziò a lavorare su questo documento nel Novembre del 1997 e fu subito aiutato da Douglas Eadline. Dopo pochi mesi il Beowulf HOWTO divenne troppo grande e nell'Agosto 1998 venne diviso in tre documenti: il Beowulf HOWTO, il Beowulf Architecture Design HOWTO, and the Beowulf Installation and Administration HOWTO. La versione 1.0.0 del Beowulf HOWTO venne rilasciato al Linux Documentation Project l'11 Novembre 1998. Noi speriamo che questo sia solo l'inizio di ciò che dovrebbe diventare un completo Beowulf Documentation Project.

1.4 Note sugli autori

- Jacek Radajewski lavora come Amministratore di Rete e sta studiando per la laurea in informatica all'Università di Southern Queensland, in Australia. Il suo primo approccio con Linux avvenne nel 1995 e fu subito amore a prima vista. Jacek ha costruito il suo primo cluster Beowulf nel Maggio del

1997 e sta giocando con questa tecnica da allora, cercando continuamente di trovare nuovi e migliori modi per impostare il tutto. Può essere contattato all'indirizzo *jacek@usq.edu.au*

- Douglas Eadline, Ph.D. è il Presidente e Direttore della Ricerca a Paralogic, Inc., Bethlehem, PA, USA. Ha studiato come Chimico Fisico/Analitico e si è interessato ai computer fin dal 1978 quando costruì il suo primo computer per usarlo con le strumentazioni chimiche. I suoi attuali interessi sono Linux, i cluster Beowulf e gli algoritmi paralleli. Può essere raggiunto all'indirizzo *deadline@plogic.com*

1.5 Riconoscimenti

La stesura del Beowulf HOWTO è stato un processo lungo ed è finalmente completo, grazie a molte persone. Voglio ringraziare le seguenti persone per il loro aiuto e i loro contributi a questo HOWTO.

- Becky per il suo amore, supporto e comprensione.
- Tom Sterling, Don Becker e altre persone alla NASA che hanno iniziato il progetto Beowulf.
- Thanh Tran-Cong e la Facoltà di Ingegneria e Controllo per aver reso disponibile per gli esperimenti la macchina Beowulf *topcat*.
- Il mio tutor Christopher Vance per molte grandi idee.
- Il mio amico Russell Waldron per le grandi idee, il suo interesse generale nel progetto e l'aiuto.
- Il mio amico David Smith per aver controllato questo documento.
- Le molte persone della lista Beowulf che mi hanno dato feedback e idee.
- Tutte le persone curatrici del sistema operativo Linux e di tutti i programmi di software libero usati su *topcat* e altre macchine Beowulf.

2 Introduzione

Via via che le prestazioni dei computer comuni e dell'hardware di rete crescono, e il loro prezzo cala, diventa via via sempre più facile costruire sistemi di calcolo parallelo dai componenti di facile reperibilità invece che comprare tempi CPU sui Supercomputer molto costosi. In effetti il rapporto costo e prestazioni di una macchina di tipo Beowulf è dalle tre alle dieci volte migliore che nei tradizionali Supercomputer. L'architettura Beowulf è scalabile, è facile da realizzare e si deve pagare solo l'hardware in quanto gran parte del software è gratuito.

2.1 A chi è indirizzato questo HOWTO?

Questo HOWTO è adatto per persone che abbiano un po' di esperienza con il sistema operativo Linux. La conoscenza della tecnologia Beowulf o la comprensione di sistemi operativi più complessi e concetti di networking non è essenziale, ma qualche esperienza con il calcolo parallelo potrebbe essere utile (d'altronde dovresti avere qualche ragione per leggere questo documento). Questo HOWTO non risponde a tutte le domande che potrebbero sorgere su Beowulf, ma noi speriamo che possa darti buone idee e guidarti nella giusta direzione. Lo scopo di questo HOWTO è di dare informazioni iniziali, link e riferimenti a documenti più avanzati.

2.2 Cos'è un Beowulf?

Famed was this Beowulf: far flew the boast of him, son of Scyld, in the Scandian lands. So becomes it a youth to quit him well with his father's friends, by fee and gift, that to aid him, aged, in after days, come warriors willing, should war draw nigh, liegemen loyal: by lauded deeds shall an earl have honor in every clan.

Beowulf è il più antico poema epico scritto in inglese arrivato fino a noi. È la storia di un eroe di grande forza e coraggio che sconfisse un mostro chiamato Grendel. Vai a 5.7 (Storia) per saperne di più dell'eroe Beowulf.

Probabilmente ci sono tante definizioni di Beowulf quante sono le persone che realizzano o usano le caratteristiche dei Supercomputer Beowulf. Alcuni dicono che possono essere chiamati Beowulf solo i computer realizzati allo stesso modo della macchina originale della NASA. Altri invece vanno all'estremo e chiamano Beowulf tutti i sistemi di workstation che fanno girare codice parallelo. La mia definizione di Beowulf sta da qualche parte nel mezzo fra questi due modi di vedere ed è basata su molti messaggi della lista Beowulf:

Beowulf è un'architettura a multicomputer che può essere usata per calcoli paralleli. È un sistema che normalmente consiste di un nodo server e uno o più nodi client connessi via Ethernet o altri tipi di rete. È un sistema costruito usando componenti hardware comuni, come qualunque PC che può far girare Linux, normali adattatori Ethernet e switch. Non contiene alcun componente hardware speciale ed è facilmente realizzabile. Beowulf utilizza inoltre software comune come il sistema operativo Linux, Parallel Virtual Machine (PVM) e Message Passing Interface (MPI). Il nodo server controlla tutto il cluster e fornisce i file ai nodi client. È anche la console del cluster e il gateway con il mondo circostante. Grandi macchine Beowulf possono avere più di un nodo server e possibilmente altri nodi dedicati a compiti particolari, come per esempio console o stazioni di monitoraggio. In molti casi i nodi client in un sistema Beowulf sono dedicati al lavoro per il Beowulf, più sono dedicati meglio è. I nodi sono configurati e controllati dal nodo server, e fanno solo ciò che gli viene detto. In una configurazione disk-less (senza dischi), i nodi client non conoscono nemmeno il loro indirizzo IP né il nome finché il server glielo comunica. Una delle differenze principali fra il Beowulf e il Cluster di Workstation (COW) è il fatto che Beowulf si comporta più come una macchina singola che le molte workstation. In molti casi i client non hanno tastiere o monitor, e vi si accede solo via login remoto o possibilmente terminali seriali. Un nodo Beowulf può essere pensato come un pacchetto composto da CPU e memoria che può essere inserito nel cluster, proprio come una CPU o un modulo di memoria possono essere infilati in una scheda madre.

Beowulf non è un nuovo pacchetto software, un nuovo tipo di rete o l'ultima versione di sviluppo del kernel. Beowulf è un tecnica di clustering di computer con Linux per formare un supercomputer parallelo virtuale. Inoltre ci sono molti pacchetti software come modifiche al kernel, le librerie PVM e MPI e programmi di configurazione che rendono l'architettura Beowulf più veloce, più facile da configurare e molto più usabile; si può creare una macchina di classe Beowulf usando distribuzioni Linux standard senza alcun software addizionale. Se avete due computer Linux in rete che condividono al limite il file system `/home` via NFS e permettono di eseguire shell remote (rsh), allora si può pensare di avere una semplice macchina Beowulf composta da due nodi.

2.3 Classificazione

I sistemi Beowulf sono stati realizzati da una gran varietà di parti. Per favorire le prestazioni sono stati usati alcuni componenti non comuni (cioè prodotti da una singola ditta). Per contare i differenti sistemi e per rendere la spiegazione delle macchine un po' più semplice, noi proponiamo questo semplice schema di classificazione:

BEOWULF DI CLASSE I:

Questa classe di macchine costruita interamente da componenti comuni di facile reperibilità. Noi useremo il test di certificazione di "Computer Shopper" per definire i componenti comuni di facile reperibilità (Computer Shopper è una rivista/catalogo mensile spessa un pollice di sistemi e componenti per PC). Il test è questo:

Un Beowulf di CLASSE I è una macchina che può essere realizzata da componenti trovabili in almeno 3 cataloghi pubblicitari a carattere nazionale o globale.

I vantaggi di un sistema di CLASSE I sono:

- l'hardware è reperibile da molte fonti (bassi prezzi e facile manutenzione)
- non c'è dipendenza da un singolo rivenditore di hardware
- supporto dei driver dei prodotti Linux
- è basato normalmente su standard (SCSI, Ethernet, ecc.)

Gli svantaggi di un sistema di CLASSE I sono:

- maggiori prestazioni possono richiedere un hardware di CLASSE II

Beowulf di CLASSE II

Un Beowulf di CLASSE II è semplicemente qualsiasi macchina che non soddisfa il test di certificazione di Computer Shopper. Questa non è una cosa brutta. È semplicemente una classificazione della macchina.

I vantaggi di un sistema di CLASSE II sono:

- Le prestazioni possono essere abbastanza buone!

Gli svantaggi di un sistema di CLASSE II sono:

- il supporto dei driver può variare
- dipendenza da un singolo rivenditore hardware
- possono essere molto più costosi di un sistema di CLASSE I.

Una CLASSE non è necessariamente migliore dell'altra, Tutto dipende dalle tue necessità e dal tuo budget. Questa classificazione serve solo a rendere la spiegazione dei sistemi Beowulf un po' più breve. La sezione 4 (Progetto del sistema) ti può aiutare a determinare che tipo di sistema si adatta alle tue necessità.

3 Uno sguardo all'architettura

3.1 A cosa assomiglia?

Io credo che il miglior modo per descrivere l'architettura del supercomputer Beowulf è quello di usare un esempio molto simile al Beowulf reale ma molto familiare a molti amministratori di sistema. L'esempio più vicino ad una macchina Beowulf è un laboratorio Unix con un server e un certo numero di client. Per essere ancora più precisi userò come esempio il laboratorio di computer DEC Alpha per laureandi della facoltà di Scienze dell'USQ. Il computer server si chiama *beldin* e le macchine client sono chiamate *scilab01*, *scilab02*, *scilab03*, fino a *scilab20*. Tutti i client hanno installata una copia locale del sistema operativo Digital Unix 4.0 ma condividono la directory degli utenti `/home` e la directory `/usr/local` dal server via NFS (Network File System). Ogni client ha una voce per il server e una per ognuno degli altri client nel suo file `/etc/hosts.equiv`, così che ogni client può eseguire una shell remota (rsh) in tutti gli altri. La macchina server è anche server NIS per tutto il laboratorio, così che tutti gli account sono comuni in tutte le macchine. Una persona può sedersi davanti alla console di *scilab02* ed avere la stessa configurazione come se si fosse fatto il login nel server o nel client *scilab15*. La ragione per la quale tutti i client hanno lo stesso look and feel è che il sistema operativo è installato e configurato allo stesso modo su tutte le macchine e che sia `/home` che `/usr/local` sono fisicamente sul server e condivise da tutti i client via NFS. Per maggiori informazioni a proposito di NIS e NFS guardate i relativi HOWTO a *NIS* e *NFS* .

3.2 Come usare gli altri nodi?

Ora che abbiamo un'idea circa l'architettura del sistema, diamo un'occhiata a come possiamo utilizzare i cicli di CPU disponibili nei computer del laboratorio. Ogni persona può fare il login in ogni macchina, e far girare un programma nella sua directory home, ma può anche distribuire il lavoro su macchine differenti semplicemente eseguendo shell remote. Per esempio, mettiamo che vogliamo calcolare la somma delle radici quadrate di tutti gli interi compresi fra 1 e 10. Scriviamo un programma chiamandolo `sigmassqrt` (vedi 6.2 (Codice sorgente)) che fa proprio questo. Per calcolare la somma delle radici quadrate dei numeri da 1 a 10 noi eseguiamo:

```
[jacek@beldin sigmasqrt]$ time ./sigmasqrt 1 10 22.468278

real    0m0.029s
user    0m0.001s
sys     0m0.024s
```

Il comando `time` ci permette di controllare il tempo trascorso per completare l'altro comando. Come possiamo vedere, questo esempio ha necessitato solo una frazione di secondo (0,029 sec), ma cosa succederebbe se io volessi conoscere la somma delle radici quadrate degli interi compresi fra 1 e 1 000 000 000 (un miliardo)? Riproviamo e ricontrolliamo il tempo necessario.

```
[jacek@beldin sigmasqrt]$ time ./sigmasqrt 1 1000000000
21081851083600.559000

real    16m45.937s
user    16m43.527s
sys     0m0.108s
```

Questa volta il tempo di esecuzione è decisamente maggiore. L'ovvia domanda che sorge è cosa possiamo fare per abbreviare il tempo di esecuzione di questo programma? Come possiamo cambiare il modo in cui si esegue questo programma per ridurne il tempo di esecuzione? L'ovvia risposta è quella di dividere il lavoro in una serie di sotto-compiti e di farli girare in parallelo su tutti i computer. Noi possiamo dividere il calcolo di una grossa addizione in 20 parti, calcolare una serie di radici quadrate e farli girare su ogni nodo. Quando tutti i nodi finiscono il calcolo e restituiscono i loro risultati, i 20 numeri possono essere sommati per ottenere il risultato finale. Prima di far girare questo programma noi faremo una named pipe che verrà usata da tutti i processi per scrivere il loro risultato.

```
[jacek@beldin sigmasqrt]$ mkfifo output
[jacek@beldin sigmasqrt]$ ./prun.sh & time cat output | ./sum
[1] 5085
21081851083600.941000
[1]+  Done                    ./prun.sh

real    0m58.539s
```

```
user    0m0.061s
sys     0m0.206s
```

Il tempo che otteniamo è di circa 58,5 secondi. Questo è il tempo trascorso dall'inizio del programma fino alla restituzione all'interno della pipe dei risultati di tutti i nodi. Il tempo così calcolato non include la somma finale dei venti numeri, ma questo tempo è molto breve e può essere trascurato. Possiamo così vedere che c'è un notevole miglioramento eseguendo questo programma in parallelo. In effetti il programma parallelo ha girato 17 volte più velocemente, che è un buon risultato avendo incrementato di 20 volte il numero di CPU coinvolte. Lo scopo di questo esempio è quello di far vedere il metodo più semplice di realizzare codice parallelo concorrente. In pratica però esempi così semplici sono rari ma vengono usate altre tecniche (le API PVM e PMI) per ottenere il parallelismo.

3.3 In cosa Beowulf è differente da un COW ?

Il laboratorio di computer descritto sopra è un perfetto esempio di un Cluster di Workstation (COW). E allora cosa ha tanto di speciale Beowulf e in cosa è diverso da un COW? La verità è che non c'è una grande differenza, ma Beowulf ha alcune caratteristiche peculiari. Innanzitutto in molti casi i nodi client di un cluster Beowulf non hanno tastiere, mouse, schede video né monitor. Tutti gli accessi ai nodi client sono realizzati in remoto dal nodo server, nodi che fungono solo da console o da un terminale seriale. Poiché non c'è bisogno per i nodi client di accedere a computer al di fuori del cluster, né per computer esterni al cluster di accedere direttamente ai nc, è pratica comune dotare i nc di IP privati come gli indirizzi compresi in 10.0.0.0/8 o 192.168.0.0/16 (vedi RFC 1918 *http://www.alternic.net/rfcs/1900/rfc1918.txt.html*). Normalmente l'unica macchina connessa all'esterno con una seconda scheda di rete è il nodo server. Il modo più comune per usare un sistema di questo tipo è di usare la console del ns, oppure di entrare in telnet o shell remota dal computer personale. Una volta sul ns, gli utenti possono modificare e compilare il loro codice e anche suddividere i loro programmi su tutti i nodi del cluster. In molti casi i COW sono usati per calcoli paralleli la notte e durante i fine settimana quando gli utenti non usano effettivamente i computer per i compiti di tutti i giorni, utilizzando così cicli idle di CPU. Beowulf d'altra parte è una macchina dedicata normalmente solo al calcolo parallelo e ottimizzato per questo scopo. Beowulf inoltre ha un miglior rapporto fra prezzo e prestazioni poiché è realizzato da componenti di facile reperibilità e usa normalmente software gratuito. Beowulf ha inoltre un'immagine più da sistema singolo cosa che aiuta gli utenti a vedere un cluster Beowulf come una singola workstation

4 Progetto del sistema

Prima di acquistare dell'hardware, è sempre una buona idea prendere in considerazione il progetto del sistema che si intende realizzare. Fondamentalmente ci sono due questioni relative al progetto di un sistema Beowulf: il tipo di nodi (cioè di computer) da usare e il modo in cui tali nodi sono connessi. C'è una questione relativa al software, che può influire sulle decisioni che riguardano l'hardware: la libreria di comunicazione (API). Nel seguito di questo documento verrà fatta una discussione più dettagliata riguardo all'hardware e al software di comunicazione.

Anche se il numero di possibilità non è elevato, quando si costruisce un Beowulf, ci sono comunque alcune importanti decisioni di progetto da prendere. Poiché la scienza (o l'arte) della "computazione parallela" ha molte interpretazioni differenti, qui di seguito ne viene data un'introduzione. Se non ti piace leggere materiale di rassegna, puoi saltare questa sezione, ma sei vivamente invitato a leggere la sezione 4.4 (Fattibilità) prima di prendere le decisioni finali relative all'hardware.

4.1 Una breve rassegna sulla computazione parallela

Questa sezione fornisce una rassegna sui concetti della computazione parallela. NON è una descrizione completa né esauriente della scienza e tecnologia della computazione parallela. È una breve descrizione delle questioni che possono essere importanti per un progettista o per un utente Beowulf.

Mentre progetti e costruisci il tuo Beowulf, molte delle questioni che vengono descritte sotto diventeranno importanti nel processo delle tue decisioni. A causa della natura dei suoi componenti, un supercomputer Beowulf richiede che si prendano attentamente in considerazione molti fattori che adesso vengono a essere sotto il nostro controllo. In generale, non è così difficile comprendere le questioni relative alla computazione parallela. In realtà, una volta che le questioni sono comprese, le tue aspettative saranno più realistiche e avrai una maggiore probabilità di successo. A differenza del "mondo sequenziale", in cui la velocità del processore è considerato il solo fattore che ha la massima importanza, la velocità dei processori nel "mondo parallelo" è solo uno dei vari fattori che determineranno le prestazioni e l'efficienza del sistema complessivo.

4.2 I metodi della computazione parallela

La computazione parallela può assumere molte forme. Dal punto di vista dell'utente è importante considerare vantaggi e svantaggi di ciascuna metodologia. La sezione seguente tenta di dare diverse prospettive sui metodi della computazione parallela e indica dove va a cadere una macchina Beowulf all'interno di questo continuum.

4.2.1 Perché più di una CPU?

È importante rispondere a questa domanda. Usare 8 CPU per eseguire un word processor suona un po' "over-kill" – e in effetti lo è. Ma cosa dire di un server web, una base di dati, un programma di rendering o uno schedulatore di progetti? Forse più CPU potrebbero essere d'aiuto. E cosa dire di una simulazione complessa, un codice che studia la dinamica dei fluidi, o un'applicazione di estrazione di dati? In queste situazioni, più CPU sono sicuramente d'aiuto. In effetti, sistemi con CPU multiple vengono usati per risolvere sempre più problemi.

Generalmente la successiva domanda è: "Perché ho bisogno di due o quattro CPU? Aspetterò semplicemente il chip iper-turbo 986." Ci sono varie ragioni:

1. Grazie all'uso di sistemi operativi multi-tasking, è possibile fare più cose alla volta. Questo è un "parallelismo" naturale che è facilmente sfruttato da più CPU a basso costo.
2. La velocità dei processori va raddoppiando ogni 18 mesi, ma cosa dire delle velocità delle RAM e degli hard disk? Purtroppo, queste velocità non stanno aumentando velocemente così come quelle delle CPU. Si tenga presente che la maggior parte delle applicazioni richiedono "accessi in memoria fuori cache" e accessi agli hard disk. Un modo per aggirare alcune di queste limitazioni consiste nel fare più cose in parallelo.
3. Le previsioni indicano che le velocità dei processori non continueranno a raddoppiare ogni 18 mesi dopo l'anno 2005. Per poter mantenere questa tendenza ci sono alcuni ostacoli molto ardui da superare.
4. A seconda dell'applicazione, la computazione parallela può velocizzare l'esecuzione di un fattore da 2 a 500 volte (in qualche caso anche di più). Tali performance non sono disponibili usando un singolo processore. Anche i supercomputer che un tempo usavano processori molto veloci, adesso sono costruiti utilizzando più CPU reperibili comunemente.

Se hai bisogno di velocità - sia a causa di un problema di limiti della computazione che di un problema di limiti nell'I/O, vale la pena considerare il parallelismo. Poiché la computazione parallela è implementata in vari modi, per risolvere il tuo problema con il parallelismo dovranno essere prese alcune decisioni molto importanti. Queste decisioni possono avere effetti decisivi sulla portabilità, la performance e il costo della tua applicazione.

Prima di entrare nei dettagli tecnici, diamo uno sguardo a un "problema di computazione parallela" reale, usando un esempio con il quale abbiamo familiarità: l'attesa in lunghe code a un negozio.

4.2.2 Il negozio della computazione parallela

Immaginiamo un grande negozio con 8 registratori di cassa raggruppati insieme nella parte anteriore del negozio. Assumiamo che ogni cassiere/registratore corrisponda a una CPU e ogni cliente a un programma di computer. La dimensione del programma di computer (il carico di lavoro) corrisponde alla dimensione della spesa del cliente. Per illustrare i concetti della computazione parallela possono essere usate le seguenti analogie.

Sistema operativo Single-tasking Un solo registratore di cassa è aperto (in uso) e deve servire ogni cliente, uno alla volta.

Esempio relativo ai computer: MS DOS

Sistemi operativi Multi-tasking: Un solo registratore di cassa è aperto, ma adesso viene servita solo una parte di una spesa per volta, si passa al prossimo cliente e si serve una parte della sua spesa. I clienti "sembrano" muoversi lungo la coda insieme, ma se oltre a te non ci sono altri clienti, verrai servito più velocemente.

Esempio relativo ai computer: UNIX, NT utilizzante una singola CPU

Sistemi operativi Multitasking con più CPU: Adesso apriamo più registratori di cassa nel negozio. Ogni cliente può essere servito da un registratore di cassa separato e la coda può muoversi più velocemente. Questa viene chiamata SMP - Multi-elaborazione simmetrica (Symmetric Multi-processing). Sebbene ci siano più registratori di cassa aperti, continuerai a non muoverti lungo la coda più velocemente del caso in cui ci sei solo tu e un solo registratore di cassa.

Esempio relativo ai computer: UNIX e NT con più CPU

Thread su un sistema operativo multitasking con più CPU: Se "spezzi" gli articoli della tua spesa, puoi muoverti più velocemente sulla coda usando più registratori di cassa allo stesso tempo. Innanzitutto, dobbiamo ipotizzare un gran guadagno, perché il tempo investito nello "spezzare la spesa" deve essere riguadagnato usando più registratori di cassa. In teoria, dovresti muoverti lungo la coda "n" volte più veloce di prima, dove "n" è il numero di registratori di cassa. Quando i cassieri hanno bisogno dei totali parziali, possono scambiarsi velocemente informazioni guardando e parlando a ogni altro registratore di cassa "locale". Possono anche curiosare ai registratori di cassa vicini per cercare informazioni di cui hanno bisogno per lavorare più velocemente. Comunque, c'è un limite al numero di registratori di cassa che possono trovarsi in un punto del negozio.

La legge di Amdal, inoltre, limiterà la velocità dell'applicazione alla porzione sequenziale del programma più lenta.

Esempio relativo ai computer: UNIX oppure NT con extra CPU sulla stessa scheda madre che eseguono programmi multi-threaded.

Invio di messaggi su sistemi operativi multitasking con più CPU: Al fine di migliorare la performance, il negozio aggiunge 8 registratori di cassa sul retro del negozio. Poiché i nuovi registratori di cassa sono lontani da quelli che si trovano sul davanti, i cassieri devono chiamare questi al telefono per comunicare i loro totali parziali. Tale distanza aggiunge un sovraccarico (di tempo) alla comunicazione tra cassieri, ma se le comunicazioni sono minimizzate, ciò non costituisce un problema. Se fai una spesa davvero molto grande, tale da richiedere tutti i registratori di cassa, allora, come avveniva prima, la velocità può essere aumentata usando tutti i registratori di cassa allo stesso tempo, e il sovraccarico va riconsiderato. In qualche caso, il negozio può avere singoli registratori di cassa (o gruppi isolati di registratori di cassa) sparsi per il negozio: ogni registratore di cassa (o isola) deve comunicare via telefono. Poiché tutti i cassieri possono comunicare l'un l'altro attraverso il telefono, non è molto importante dove si trovano.

Esempio relativo ai computer: una o più copie di UNIX o NT con più CPU sulla stessa o su differenti schede madri, che comunicano attraverso messaggi.

Gli scenari descritti sopra, sebbene non esatti, sono una buona rappresentazione dei vincoli posti sui sistemi paralleli. A differenza di sistemi con singola CPU (o registratore di cassa), qui va presa in considerazione la possibilità di comunicazione tra diverse CPU.

4.3 Architetture per la computazione parallela

Di seguito vengono presentati i metodi e le architetture comuni della computazione parallela. Sebbene la presente trattazione non sia sicuramente esauriente, è sufficiente per comprendere le questioni fondamentali relative a un progetto Beowulf.

4.3.1 Architetture Hardware

Ci sono fondamentalmente due modi in cui viene messo insieme l'hardware dei computer paralleli:

1. Macchine con memoria locale che comunicano mediante messaggi (clusters Beowulf)
2. Macchine con memoria condivisa che comunicano attraverso la memoria (macchine SMP)

Un tipico Beowulf è un insieme di macchine con singola CPU, connesse usando fast Ethernet ed è, pertanto, una macchina a memoria locale. Una macchina SMP a 4 vie è una macchina a memoria condivisa e può essere usata per fare computazioni parallele - applicazioni parallele che comunicano usando la memoria condivisa. Proprio come nell'esempio dell'analogia negozio-computer, le macchine a memoria locale (registratori di cassa individuali) possono essere scalati a grandi numeri di CPU, mentre il numero di CPU in macchine a memoria condivisa (il numero di registratori di cassa che si possono mettere in un punto) può essere limitato a causa di conflitti nell'accesso alla memoria.

È possibile, comunque, connettere molte macchine a memoria condivisa per creare una macchina a memoria condivisa "ibrida". Queste macchine ibride "appaiono" all'utente come una grande macchina SMP singola e sono spesso chiamate macchine NUMA (accesso in memoria non uniforme), perché la memoria globale vista dal programmatore e condivisa da tutte le CPU può avere differenti ritardi. A qualche livello, comunque, una macchina NUMA deve "inviare messaggi" tra gruppi di memorie localmente condivise.

È anche possibile connettere macchine SMP come nodi di computazione a memoria locale. Le tipiche schede madri della CLASSE I hanno 2 o 4 CPU e sono usate spesso come un mezzo per ridurre il costo del sistema complessivo. Lo schedulatore interno di Linux determina come queste CPU ottengono le risorse condivise. L'utente non può (a questo punto) assegnare uno specifico task a uno specifico processore SMP. L'utente può, comunque, iniziare due processi indipendenti oppure un processo multithreaded e aspettarsi un aumento di performance rispetto a un sistema avente una singola CPU.

4.3.2 Architetture API software

Ci sono fondamentalmente due modi per "esprimere" la concorrenza in un programma:

1. Usare l'invio di messaggi tra processori
2. Usare i thread del sistema operativo

Esistono altri metodi, ma questi due sono quelli più largamente usati. È importante ricordare che l'espressione della concorrenza non è necessariamente controllata dall'hardware sottostante. Sia lo scambio di messaggi che i thread possono essere implementati su SMP, NUMA-SMP e cluster - sebbene, come spiegato sotto, l'efficienza e la portabilità sono questioni importanti.

Messaggi Storicamente, la tecnologia dello scambio di messaggi rifletteva il modello dei primi computer paralleli a memoria locale. I messaggi richiedono un'operazione di copia dei dati, mentre i thread, corrispondentemente, usano dati. I ritardi e le velocità alle quali i messaggi possono essere copiati sono i fattori limitanti dei modelli a scambio di messaggi. Un messaggio è piuttosto semplice: qualche dato e un processore destinatario. Le API comuni per lo scambio di messaggi sono *PVM* oppure *MPI* . Lo scambio di messaggi può essere implementato efficientemente usando i thread, quindi i messaggi funzionano bene sia su macchine SMP che tra cluster di macchine. Il vantaggio nell'uso dei messaggi su una macchina SMP, invece del normale uso dei thread, è che se decidi in futuro di usare cluster, è facile aggiungere macchine o scalare le tue applicazioni.

Thread I thread del sistema operativo sono stati sviluppati poiché i progetti di SMP (multi-elaborazione simmetrica) a memoria condivisa consentivano una comunicazione e sincronizzazione molto veloci, tra parti concorrenti di un programma. I thread funzionano bene su sistemi SMP perché la comunicazione avviene attraverso la memoria condivisa. Per questa ragione l'utente deve isolare i dati locali da quelli globali, altrimenti i programmi non gireranno nel modo corretto. A differenza dei messaggi, con i thread una buona parte di copiatura può essere evitata, perché i dati sono condivisi dai processi (thread). Linux supporta i thread POSIX. Il problema con i thread è che è difficile estenderli oltre una macchina SMP e poiché i dati sono condivisi dalle CPU, le questioni relative alla coerenza delle cache può contribuire al sovraccarico. Estendere i thread oltre i limiti della SMP in modo efficiente richiede la tecnologia NUMA che è costosa e non supportata da Linux in forma nativa. I thread sono stati implementati sopra i messaggi (*(http://syntron.com/ptools/ptools_pg.htm)*), ma i thread implementati usando i messaggi sono spesso inefficienti.

Riguardo alla performance si può affermare quanto segue:

```
                performance di      performance di un        scalabilità
                una macchina SMP    cluster di macchine
                ----------------    -------------------      -----------
messaggi        buona               ottima                   ottima

thread          ottima              scarsa*                  scarsa*

* richiede una tecnologia NUMA costosa.
```

4.3.3 Architettura dell'applicazione

Per poter eseguire un'applicazione in parallelo su più CPU, occorre suddividerla in parti concorrenti. Un'applicazione progettata per una singola CPU, non verrà eseguita più velocemente di un'applicazione per singola CPU su una macchina multiprocessore.

Ci sono alcuni strumenti e compilatori che possono suddividere i programmi, ma parallelizzare il codice, non è un'operazione "plug'n'play". A seconda del tipo di applicazione, parallelizzare il codice può essere facile, estremamente difficile o addirittura impossibile, in base alle dipendenze dell'algoritmo.

Prima di affrontare la questione del software, occorre introdurre il concetto di Fattibilità.

4.4 Fattibilità

Molte domande relative all'elaborazione parallela, hanno la stessa risposta:

"Dipende dall'applicazione"

Prima di addentrarsi nel problema, c'è un'importante distinzione da fare - la differenza tra CONCORRENTE e PARALLELO. Per il gusto della discussione definiremo così questi due concetti:

CONCORRENTI: parti di un programma che possono essere eseguite indipendentemente.

PARALLELE: parti CONCORRENTI di un programma eseguite da processori diversi nello stesso momento.

La distinzione è molto importante, poiché la CONCORRENZA è una proprietà del programma, mentre il PARALLELISMO è una proprietà della macchina. Idealmente un'esecuzione parallela dovrebbe risultare in prestazioni più veloci. Il fattore limitante nelle prestazioni parallele è la velocità di comunicazione e la latenza fra i nodi (la latenza esiste anche con applicazioni i cui thread sono eseguiti su CPU diverse, a causa della necessità di controllare la coerenza della cache - cache coherency). Molti dei comuni benchmark paralleli sono altamente paralleli, quindi latenza e comunicazione non costituiscono colli di bottiglia. Questo tipo di problema può essere chiamato "ovviamente parallelo". Altre applicazioni non sono così semplici ed eseguire parti CONCORRENTI del programma in PARALLELO potrebbe causare un rallentamento del programma, vanificando così ogni miglioramento di prestazioni ottenuto nelle parti CONCORRENTI del programma. In altre parole, il tempo di comunicazione impiegato deve essere proporzionato con quello impiegato nell'elaborazione, altrimenti l'esecuzione PARALLELA di parti CONCORRENTI è inefficiente.

È compito del programmatore determinare quali parti CONCORRENTI del programma devono essere eseguite in PARALLELO e quali NO. La risposta a ciò determinerà l'EFFICIENZA dell'applicazione. Il grafico che segue, riassume la situazione per il programmatore:

```
        | *
        | *
        | *
% di    | *
appli-  |  *
cazione |  *
        |  *
        |  *
        |    *
        |     *
```

```
|       *
|         ****
|             ****
|                 ********************
+------------------------------------
 tempo di comunicazione/tempo di elaborazione
```

In un perfetto computer parallelo, il rapporto fra comunicazione e elaborazione sarebbe uguale, e tutto ciò che fosse CONCORRENTE potrebbe essere implementato in PARALLELO. Sfortunatamente computer paralleli reali, incluse macchine con memoria condivisa, sono soggette agli effetti descritti nel grafico. Chi progetta un Beowulf, tenga in mente questo grafico, perché l'efficienza dell'elaborazione parallela dipende dal rapporto fra tempo di comunicazione e tempo di elaborazione per UNO SPECIFICO COMPUTER PARALLELO. Le applicazioni possono essere portabili fra computer paralleli, ma non c'è garanzia che saranno efficienti su una piattaforma differente.

IN GENERALE, NON ESISTE UN PROGRAMMA PARALLELO PORTABILE ED EFFICIENTE.

C'è ancora un'altra conseguenza del grafico di cui sopra. Poiché l'efficienza dipende dal rapporto comunic./elab., cambiare un solo componente del rapporto non significa, necessariamente, che un'applicazione sarà più veloce. Un processore più veloce, mantenendo la stessa velocità di comunicazione, potrebbe non avere effetti visibili sul programma. Per esempio, raddoppiando o triplicando la velocità della CPU, mantenendo la stessa velocità di comunicazione, potrebbe far sì che alcune porzioni che prima erano eseguite in PARALLELO, adesso siano più efficienti se eseguite SEQUENZIALMENTE. Quindi, adesso potrebbe essere più veloce eseguire le parti che prima erano PARALLELE, come SEQUENZIALI. Inoltre, una inefficiente esecuzione di porzioni parallele, impedirà all'applicazione di raggiungere la sua massima velocità. Quindi l'aggiunta di un processore più veloce potrebbe rallentare l'applicazione (impedendo alla nuova CPU di raggiungere la sua massima velocità, per quell'applicazione).

SOSTITUIRE LA CPU CON UNA PIÙ VELOCE, POTREBBE RALLENTARE L'APPLICAZIONE.

In conclusione, quindi, per sapere se usare o no un'architettura parallela, occorre avere un po' di intuito circa l'adeguatezza di una particolare macchina per un'applicazione. Occorre tener presente svariati fattori, incluso la velocità della CPU, il compilatore, le "message passing API", la rete, ecc. Inoltre occorre tener presente che l'aver tracciato lo schema di un'applicazione non è tutto.

È possibile identificare una porzione del programma in cui è richiesta una pesante elaborazione, ma non è possibile conoscerne il costo in termini di comunicazione. Potrebbe essere che, per un certo sistema, i costi di comunicazione rendano inefficiente l'esecuzione in parallelo del codice.

Una nota finale su un malinteso comune. Spesso viene detto che un programma è PARALLELIZZATO, ma in realtà solo le parti CONCORRENTI lo sono. Per tutte le ragioni dette sopra, il programma non è PARALLELIZZATO. Un'efficiente PARALLELIZZAZIONE è una proprietà della macchina.

4.5 Scrittura e porting di software parallelo

Una volta stabilita la necessità della computazione parallela e quindi di costruire un Beowulf, potrebbe essere una buona idea ripensare all'applicazione in base a quanto detto precedentemente.

In generale ci sono due cose che possono essere fatte:

1. Andare avanti e costruire un Beowulf di classe I e quindi adattare ad esso l'applicazione. O eseguire applicazioni parallele che sappiamo funzioneranno su tale Beowulf (ma attenzione alla portabilità e all'efficienza di cui sopra).

2. Studiare le applicazioni che verranno eseguite sul Beowulf e fare alcune stime sul tipo di hardware e software di cui c'è bisogno.

In altri casi, in qualche momento, occorrerà prendere in considerazione i fattori relativi all'efficienza.

In generale possono essere fatte tre cose:

1. Determinare le parti concorrenti di un programma
2. Stimare efficientemente le porzioni parallele
3. Descrivere le parti concorrenti

Esaminiamole una per una

4.5.1 Determinare le parti concorrenti di un programma

Questo passo è spesso considerato come "parallelizzazione del programma". Le decisioni sulle parallelizzazioni, verranno prese nel passo 2. In questo passo occorre determinare le data dependencies.

Da un punto di vista pratico, le applicazioni possono esporre due tipi di concorrenza: pesanti elaborazioni (macina numeri, "number crunching") e I/O (database). Sebbene in molti casi la concorrenza di elaborazione e I/O sono ortogonali, ci sono applicazioni che li richiedono entrambi. Esistono alcuni strumenti che possono effettuare una analisi di concorrenza sulle applicazioni esistenti. La maggior parte di questi strumenti sono progettati per il FORTRAN. Due sono le ragioni per cui viene utilizzato il FORTRAN: storicamente la maggior parte di applicazioni che effettuano numerosi calcoli sono state scritte in FORTRAN ed è più facile da analizzare. Se non ci sono strumenti disponibili, questo passo può essere complicato, per le applicazioni esistenti.

4.5.2 Stima dell'efficienza parallela

Senza l'aiuto di strumenti, questo passo può richiedere prove ed errori, o solo provare formulare delle ipotesi. Per ogni specifica applicazione occorre provare a determinare se ha dei limiti di CPU (legati all'elaborazione) o di hard disk (legati all'I/O). Le esigenze di Beowulf possono differire in base alle necessità. Per esempio un problema legato all'elaborazione può aver bisogno di poche CPU molto veloci e una rete molto veloce, mentre un problema legato all'I/O può funzionare meglio con più CPU più lente e una fast Ethernet.

Questa raccomandazione, in genere, è una sorpresa per molte persone, perché è un luogo comune che i processori più veloci sono sempre meglio. Questo è vero se si dispone di un budget illimitato, i sistemi reali possono avere vincoli di costo che occorre valutare. Per i problemi legati all'I/O, c'è una regola poco conosciuta (chiamata Legge Eadline-Dedkov) che è abbastanza utile:

Poiché due computer paralleli con lo stesso indice di prestazioni per la somma delle CPU, quello che ha i processori più lenti (e probabilmente una corrispondente rete di comunicazione più lenta fra i processori) avrà prestazioni migliori per quelle applicazioni che fanno largo uso di I/O.

Mentre la profondità di questa regola va oltre lo scopo di questo documento, potrebbe essere interessante scaricare il documento *Performance Considerations for I/O-Dominant Applications on Parallel Computers* (Postscript format 109K) *(ftp://www.plogic.com/pub/papers/exs-pap6.ps)*

Una volta determinato il tipo di concorrenza dell'applicazione, occorre stimare quanto possa essere efficiente in parallelo. Vedere la sezione 5.4 (Software) per una descrizione degli strumenti Software.

In assenza di strumenti, occorre provare ad ipotizzare la propria strada, in questa fase. Se un'elaborazione che richiede molta cpu può essere misurata in minuti e i dati possono essere trasferiti in secondi, allora potrebbe essere una buona candidata per la parallelizzazione. Occorre ricordare, però, che se un loop di 16 minuti viene suddiviso in 32 parti, il trasferimento dei dati richiede diversi secondi per parte, allora le cose potranno essere difficili. Si raggiungerà un punto in cui i ritorni sono ridotti.

4.5.3 Descrizione delle parti concorrenti di un programma

Ci sono diversi modi per descrivere le parti concorrenti di un programma:

1. Esecuzione parallela esplicita
2. Esecuzione parallela implicita

La maggiore differenza fra le due è che l'esplicita è determinata dall'utente, mentre l'implicita è determinata dal compilatore.

Metodi espliciti In pratica l'utente deve specificatamente modificare il codice sorgente per un computer parallelo. L'utente deve aggiungere messaggi usando *PVM* o *MPI* o aggiungere thread usando i thread POSIX (occorre ricordare, però, che i thread non si possono muovere fra piastre madri SMP). I metodi espliciti sono i più difficili da implementare e da mettere sotto debug. Gli utenti, in genere, includono chiamate a funzioni esplicite in sorgenti standard FORTRAN 77 o C/C++. Alla libreria MPI sono state aggiunte alcune funzioni per rendere più facili da implementare alcuni metodi paralleli standard (ad es. le funzioni scatter/gather). Inoltre è possibile usare librerie standard scritte per computer paralleli. Occorre considerare sempre, però il rapporto fra portabilità ed efficienza).

Per ragioni storiche, la maggior parte dei sorgenti di programmi che effettuano pesanti calcoli ("number crunching") sono scritti in FORTRAN. Per questo motivo, il FORTRAN ha il supporto maggiore (strumenti, librerie, ecc.) per l'elaborazione parallela. Molti programmatori, adesso, utilizzano il C o riscrivono in C vecchie applicazioni scritte in FORTRAN con l'idea che il C permetterà un'esecuzione più veloce. Questo è vero, dal momento che il C è quanto di più vicino ad un linguaggio macchina universale, ma ha anche alcuni svantaggi. L'utilizzo di puntatori, in C, rende estremamente difficile determinare le dipendenze fra i dati. Se avete un programma FORTRAN e pensate di parallelizzarlo, NON CONVERTITELO IN C!

Metodi Impliciti Sono quelli in cui l'utente lascia alcune (o tutte) decisioni di parallelizzazione al compilatore. Esempi sono FORTRAN 90, High Performance FORTRAN (HPF), Bulk Synchronous Parallel (BSP) e un'intera collezione di altri metodi che sono sotto sviluppo.

I metodi impliciti richiedono che l'utente fornisca alcune informazioni circa la natura concorrente dell'applicazione, ma il compilatore, poi, prenderà la decisione di come eseguire, questa concorrenza, in parallelo. Questi metodi forniscono alcuni livelli di portabilità ed efficienza, ma non c'è ancora alcun "modo perfetto" per descrivere un problema concorrente per un'elaborazione parallela.

5 Risorse per Beowulf

5.1 Per cominciare

- La mailing list di Beowulf. Per iscriversi mandare un messaggio a *beowulf-request@cesdis.gsfc.nasa.gov* con la parola *subscribe* nel corpo del messaggio.
- La homepage di Beowulf *http://www.beowulf.org*
- Extreme Linux *http://www.extremelinux.org*

- Extreme Linux Software di Red Hat *http://www.redhat.com/extreme*

5.2 Documentazione

- L'ultima versione del Beowulf HOWTO *http://www.sci.usq.edu.au/staff/jacek/beowulf* .
- Costruire un sistema Beowulf *http://www.cacr.caltech.edu/beowulf/tutorial/building.html*
- I link di Jacek su Beowulf *http://www.sci.usq.edu.au/staff/jacek/beowulf* .
- Beowulf Installation and Administration HOWTO (DRAFT) *http://www.sci.usq.edu.au/staff/jacek/beowulf* .
- Linux Parallel Processing HOWTO *http://yara.ecn.purdue.edu/~pplinux/PPHOWTO/pphowto.html*

5.3 Documenti cartacei

- Chance Reschke, Thomas Sterling, Daniel Ridge, Daniel Savarese, Donald Becker, and Phillip Merkey *A Design Study of Alternative Network Topologies for the Beowulf Parallel Workstation.* Proceedings Fifth IEEE International Symposium on High Performance Distributed Computing, 1996. *http://www.beowulf.org/papers/HPDC96/hpdc96.html*
- Daniel Ridge, Donald Becker, Phillip Merkey, Thomas Sterling Becker, and Phillip Merkey. *Harnessing the Power of Parallelism in a Pile-of-PCs.* Proceedings, IEEE Aerospace, 1997. *http://www.beowulf.org/papers/AA97/aa97.ps*
- Thomas Sterling, Donald J. Becker, Daniel Savarese, Michael R. Berry, and Chance Res. *Achieving a Balanced Low-Cost Architecture for Mass Storage Management through Multiple Fast Ethernet Channels on the Beowulf Parallel Workstation.* Proceedings, International Parallel Processing Symposium, 1996. *http://www.beowulf.org/papers/IPPS96/ipps96.html*
- Donald J. Becker, Thomas Sterling, Daniel Savarese, Bruce Fryxell, Kevin Olson. *Communication Overhead for Space Science Applications on the Beowulf Parallel Workstation.* Proceedings,High Performance and Distributed Computing, 1995. *http://www.beowulf.org/papers/HPDC95/hpdc95.html*
- Donald J. Becker, Thomas Sterling, Daniel Savarese, John E. Dorband, Udaya A. Ranawak, Charles V. Packer. *BEOWULF: A PARALLEL WORKSTATION FOR SCIENTIFIC COMPUTATION*. Proceedings, International Conference on Parallel Processing, 95. *http://www.beowulf.org/papers/ICPP95/icpp95.html*
- Papers at the Beowulf site *http://www.beowulf.org/papers/papers.html*

5.4 Software

- PVM - Parallel Virtual Machine *http://www.epm.ornl.gov/pvm/pvm_home.html*
- LAM/MPI (Local Area Multicomputer / Message Passing Interface) *http://www.mpi.nd.edu/lam*
- BERT77 - FORTRAN conversion tool *http://www.plogic.com/bert.html*
- Beowulf software from Beowulf Project Page *http://beowulf.gsfc.nasa.gov/software/software.html*
- Jacek's Beowulf-utils *ftp://ftp.sci.usq.edu.au/pub/jacek/beowulf-utils*
- Beowulfatch - cluster monitoring tool *http://www.sci.usq.edu.au/staff/jacek/Beowulfatch*

5.5 Macchine Beowulf

- Avalon è composto da 140 processori Alpha, 36 GB di RAM ed è probabilmente la più veloce macchina Beowulf, viaggiando a 47,7 Gigaflop e si trova al 114° posto della lista dei 500 computer più veloci. *http://swift.lanl.gov/avalon/*
- Megalon-A Massively PArallel CompuTer Resource (MPACTR) è composto da 14 nodi CPU Pentium Pro quadriprocessore 200 e da 14 GB di RAM. *http://megalon.ca.sandia.gov/description.html*
- theHIVE - Highly-parallel Integrated Virtual Environment è un altro veloce Supercomputer Beowulf. theHIVE è composto da 64 nodi, 128 CPU per un totale di 4 GB RAM. *http://newton.gsfc.nasa.gov/thehive/*
- Topcat è una macchina molto più piccola e consiste di 16 CPU e da 1,2 GB di RAM. *http://www.sci.usq.edu.au/staff/jacek/topcat*
- MAGI cluster - questo è un sito molto interessante con molti link interessanti. *http://noel.feld.cvut.cz/magi/*

5.6 Altri siti interessanti

- SMP Linux *http://www.linux.org.uk/SMP/title.html*
- Paralogic - Buy a Beowulf *http://www.plogic.com*

5.7 Storia

- Leggende - Beowulf *http://legends.dm.net/beowulf/index.html*
- Le avventure di Beowulf *http://www.lnstar.com/literature/beowulf/beowulf.html*

6 Codice sorgente

6.1 sum.c

```
/* Jacek Radajewski jacek@usq.edu.au */
/* 21/08/1998 */

#include <stdio.h>
#include <math.h>

int main (void) {

  double result = 0.0;
  double number = 0.0;
  char string[80];


  while (scanf("%s", string) != EOF) {

    number = atof(string);
    result = result + number;
```

```
  }

  printf("%lf\n", result);

  return 0;

}
```

6.2 sigmasqrt.c

```
/* Jacek Radajewski jacek@usq.edu.au */
/* 21/08/1998 */

#include <stdio.h>
#include <math.h>

int main (int argc, char** argv) {

  long number1, number2, counter;
  double result;

  if (argc < 3) {
    printf ("usage : %s number1 number2\n",argv[0]);
    exit(1);
  } else {
    number1 = atol (argv[1]);
    number2 = atol (argv[2]);
    result = 0.0;
  }

  for (counter = number1; counter <= number2; counter++) {
    result = result + sqrt((double)counter);
  }

  printf("%lf\n", result);

  return 0;

}
```

6.3 prun.sh

```
#!/bin/bash
# Jacek Radajewski jacek@usq.edu.au
# 21/08/1998

export SIGMASQRT=/home/staff/jacek/beowulf/HOWTO/example1/sigmasqrt

# $OUTPUT must be a named pipe
# mkfifo output
```

```
export OUTPUT=/home/staff/jacek/beowulf/HOWTO/example1/output

rsh scilab01 $SIGMASQRT         1  50000000 > $OUTPUT < /dev/null&
rsh scilab02 $SIGMASQRT  50000001 100000000 > $OUTPUT < /dev/null&
rsh scilab03 $SIGMASQRT 100000001 150000000 > $OUTPUT < /dev/null&
rsh scilab04 $SIGMASQRT 150000001 200000000 > $OUTPUT < /dev/null&
rsh scilab05 $SIGMASQRT 200000001 250000000 > $OUTPUT < /dev/null&
rsh scilab06 $SIGMASQRT 250000001 300000000 > $OUTPUT < /dev/null&
rsh scilab07 $SIGMASQRT 300000001 350000000 > $OUTPUT < /dev/null&
rsh scilab08 $SIGMASQRT 350000001 400000000 > $OUTPUT < /dev/null&
rsh scilab09 $SIGMASQRT 400000001 450000000 > $OUTPUT < /dev/null&
rsh scilab10 $SIGMASQRT 450000001 500000000 > $OUTPUT < /dev/null&
rsh scilab11 $SIGMASQRT 500000001 550000000 > $OUTPUT < /dev/null&
rsh scilab12 $SIGMASQRT 550000001 600000000 > $OUTPUT < /dev/null&
rsh scilab13 $SIGMASQRT 600000001 650000000 > $OUTPUT < /dev/null&
rsh scilab14 $SIGMASQRT 650000001 700000000 > $OUTPUT < /dev/null&
rsh scilab15 $SIGMASQRT 700000001 750000000 > $OUTPUT < /dev/null&
rsh scilab16 $SIGMASQRT 750000001 800000000 > $OUTPUT < /dev/null&
rsh scilab17 $SIGMASQRT 800000001 850000000 > $OUTPUT < /dev/null&
rsh scilab18 $SIGMASQRT 850000001 900000000 > $OUTPUT < /dev/null&
rsh scilab19 $SIGMASQRT 900000001 950000000 > $OUTPUT < /dev/null&
rsh scilab20 $SIGMASQRT 950000001 1000000000 > $OUTPUT < /dev/null&
```