I.T.I.S Vito Volterra San Doná Di Piave (VE)

Gobbo Nicola
AS: 2006/2007

UN MOTORE DI INDICIZZAZIONE PARALLELO

1 Cos'é il calcolo parallelo?

Il calcolo parallelo é un sistema di programmazione e progettazione di algoritmi che, supportato da un opportuno hardware, permette l'esecuzione di codice su piú processori al fine di aumentare le dimensioni dei problemi affrontabili e riducendo i tempi richiesti per risolverli. Ho parlato di opportuno hardware, questo non significa necessariamente costoso, anche noi, nel nostro piccolo laboratorio abbiamo costruito un modesto cluster (cosí vengono definiti i calcolatori paralleli) di tipo Beowulf, dove normali pc, collegati in LAN, collaborano tra di loro sincronizzati da una suite di librerie, le MPICH2. Le MPICH2 sono un'implementazione Open Source dello standard Message Passing Interface (MPI) dove vengono definite una serie di funzioni e procedure che permettono di inviare messaggi tra i vari processi di un programma parallelo, per permettere la sincronizzazione, lo scambio di informazioni ecc.

2 Il programma Geegle

In questo ambiente ho progettato e sviluppato un programma chiamato Geegle, l'assonanza con il famoso motore di ricerca non é casuale, anche se il progetto é piú modesto, le somiglianze non mancano. Usando Google avrete senz'altro notato che il tempo con cui risponde a qualsiasi vostra richiesta é sempre inferiore al decimo di secondo. Un risultato insperabile se, per ogni interrogazione, dovesse scorrere tutti i documenti del web alla ricerca delle vostre parole chiave, tanto piú se a farlo fosse una macchina sola. Il successo di questo motore sta nell'efficiente utilizzo di un numero elevatissimo di macchine che costantemente scaricano i nuovi documenti dal web e aggiornano quelli vecchi e nell'utilizzo di una serie di indici che contengono particolari profili per ogni file scaricato. Ovviamente scorrere un indice, per quanto grande, sará molto piú veloce che scorrere tutti i singoli file, il nodo del problema quindi, sta nel costruire questo indice ed é qui che Geegle si innesta. Dato un "calderone" riempito con un gran numero di documenti e un file che chiameremo "dizionario" contenente le parole da ricercare, il programma si incaricherá di scovare tutti i file presenti, analizzarli uno ad uno, confrontando ogni parola al loro interno con quelle presenti nel dizionario e, se c'é una corrispondenza, incrementare un opportuno contatore di ripetizione per quella parola. Terminata l'analisi, scriverá l'indice contenente i profili di tutti i file su disco, in modo da poter essere consultato.

3 Primi passi...

Capito il compito da svolgere e valutato il potenziale incremento di prestazioni offerto da un'implementazione parallela é venuto il momento di decidere il metodo di programmazione piú adatto. Quando si programma "in parallelo" ci sono diversi metodi per affrontare un problema. Il piú semplice e usato é il *Domain Decomposition* dove tutti i computer sono istruite per eseguire le stesse operazioni ma, su zone diverse del problema, assegnate in base a un numero univoco dato dall'ambiente MPICH2 a ogni processo. Un esempio efficace é la classica squadra di imbianchini che devono dipingere un muro: una volta accordati su che parte del muro dovrá imbiancare ognuno di loro, si dedicano tutti alla pittura finché non hanno finito. Questa tecnica peró, necessita che la dimensione del dominio (nell'esempio, il muro) sia conosciuta a priori e dovrebbe essere il piú omogeneo possibile. Entrambe queste due condizioni peró, nel nostro caso sono violate in quanto il numero di file puó variare e possono avere dimensioni molto diverse. Ho deciso allora di utilizzare il paradigma del *Manager/Worker*. Con questo metodo, fin dall'inizio c'é una distinzione tra processi che comandano e sincronizzano (in Geegle uno) e processi che eseguono il lavoro. D'altro canto peró, viene ad aumentare notevolmente il numero

di comunicazioni necessarie alla sincronizzazione e ogni comunicazione, soprattutto se fatta in una LAN "artigianale", ruba tempo prezioso al calcolo abbassando le prestazioni.

4 Un'occhiata piú ravvicinata

Il programma si basa su un algoritmo di fondo molto semplice. Il processo manager cerca tutti i file che puó elaborare all'interno di una directory che funge da cache e ne crea un elenco. Negli stessi istanti un worker si sta incaricando di leggere il dizionario contenente le parole da cercare e lo sta condividendo con gli altri. Una volta che entrambi hanno le informazioni necessarie per l'esecuzione, si entra in un ciclo in cui i worker disoccupati richiedono lavoro al manager il quale, legge la lista dei file da analizzare e, se ve ne sono ancora di non esaminati, invia il loro nome al worker disoccupato. Questo prende in consegna il file e ne crea un profilo da rinviare al manager che lo scriverá su disco assieme agli altri, andando a formare l'indice. Quando la lista dei file da profilare é finita il manager invia al worker un segnale particolare per informarlo che puó terminare.

4.1 Manager

Questo é l'algoritmo base usato dal *Manager* per coordinare i lavori

```
//spiegazione variabili utilizzate
cahe_dir         //cartella dove sono salvati i file da profilare
elenco_file      //vettore contenente i nomi dei file da profilare
messaggio        //serve a scambiare informazioni tra worker e manager
file_profilati   //numero dei file analizzati
file_totali      //numero dei file trovati in cache_dir
worker_num       //ID del worker che ha inviato il messaggio
worker_terminati //numero di worker terminati
processi_totali  //numero di processi totali

file_profilati = 0
worker_terminati = 0
identifica_file(cache_dir,elenco_file,file_totali)
do
  ricevi_messaggio_da_worker(messaggio,worker_num)
if messaggio_e'_profilo(messaggio)
  then
    scrivi_profilo_su_hd(messaggio)
  else
    //il messaggio inviato e' la prima richiesta di lavoro
endif
if file_profilati < file_totali
  then
    invia_file_al_worker(elenco_file[file_profilati],worker_num)
    file_profilati = file_profilati + 1
  else
    invia_segnale_termine(messaggio,worker_num)
    worker_terminati = worker_terminati + 1
endif
while worker_terminati < processi_totali - 1
```

4.2 Worker

Questo é l'algoritmo base usato dal *Worker* per eseguire i lavori

```
//spiegazione variabili utilizzate
dizionario      //file dove sono salvate le parole
elenco_parole   //vattore contenente le parole lette dal file
file_name       //nome del file da profilare
profilo         //vettore contenente il contatore di ripetizioni
worker_num      //ID del worker
manager_num     //ID del manager

invia_prima_riciesta_di_lavoro(messaggio,manager_num)
if worker_num = 0
then
  leggi_dizionario(dizionario,elenco_parole)
endif
condividi_dizionario(elenco_parole)
do
ricevi_nome_file(file_name,manager_num)
if file_name <> ""
  then
    carica_documento(file_name)
    profila_documento(file_name,elenco_parole,profilo)
    invia_profilo(profilo,manager_num)
  else
      //il worker deve terminare
  endif
while file_name <> ""
```

4.3 Funzioni MPI utilizzate

Ogni volta che nel programma é necessario trasmettere o ricevere informazioni dagli altri processi si devono usare delle funzioni MPI. Nello standard ne sono definite a bizzeffe che servono agli usi piú disparati e specifici ma in questo caso ne vengono usate un numero molto limitato:

`MPI_Init (int *argc, char **argv[]);`
Informa il sistema operativo che si sta eseguendo un programma MPI e gli consente di eseguire tutte le inizializzazioni necessarie.

`MPI_Comm_rank (MPI_Comm comm, int *rank);`
Restituisce in *rank* l'ID del processo; c'é un ID diverso per ogni processo in esecuzione. *comm* é detto comunicatore e sostanzialmente definisce un insieme di processi che possono comunicare tra di loro.

`MPI_Comm_size (MPI_Comm comm, int *size);`
Restituisce in *size* il numero di processi all'interno del comunicatore.

`MPI_Comm_split (MPI_comm comm, int tag, int rank, MPI_Comm *new_comm);`
Divide il comunicatore *comm* in un altro comunicatore *new_comm*. Tutti i processi contenuti in *comm* devono eseguire questa istruzione.

`MPI_Send(void *message, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm);`
Invia un messaggio lungo *count* elementi di tipo *datatype* alla destinazione identificata da *dest* e contenuta nel comunicatore *comm*. Il parametro *tag* serve per distinguere comunicazioni diverse aventi i parametri precedenti uguali.

`MPI_Recv(void *message, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status);`
Riceve un messaggio lungo *count* elementi di tipo *datatype* dal processo identificato da *source* e contenuto nel comunicatore *comm*. Il parametro *tag* serve per distinguere comunicazioni diverse aventi i parametri precedenti uguali, *status* invece fornisce informazioni aggiuntive sulla trasmissione del messaggio.

`MPI_Bcast (void *message, int count, MPI_Datatype datatype, int source, MPI_Comm comm);`
Condivide un messaggio lungo *count* elementi di tipo *datatype* con tutti i processi di un comunicatore *comm*. Il parametro *source* individua chi ha i dati da condividere. Tutti i processi contenuti nel comunicatore devono eseguire questa istruzione.

`MPI_Finalize();`
Informa il sistema operativo che si sta terminando un programma MPI.

5 Improvements... Perché migliorare é sempre possibile

Una volta stilata la versione base del programma siamo passati a complicarlo un po' per renderlo piú efficente e arricchirlo con nuove funzionalitá.

5.1 Hash table

La prima delle varie ottimizzazioni apportate, vede l'utilizzo di una hash table per contenere le parole lette dal dizionario. Un algoritmo cosiddetto di hashing, converte una qualsiasi stringa di caratteri in un'altra di dimensione fissa. La particolaritá é che ogni stringa di partenza ne produce una diversa di arrivo (non propriamente vero, in quanto possono verificarsi collisioni, comunque molto rare con i recenti algoritmi). Una hash table quindi, é un vettore in cui, ogni campo é puntato dall'hash di una determinata stringa, nel nostro caso, una delle parole del dizionario. Questo ci consente di risparmiare il dover scorrere, nel caso peggiore, l'intero array dizionario per ogni parola che compone ogni file.

5.2 Supporto database

Il secondo miglioramento é venuto spontaneo al primo avvio del programma. I risultati erano scritti in un file che una volta aperto conteneva una miriade di numeri e nomi file difficilmente comprensibile. Ho deciso allora di portare tutto l'ambiente su un database. Le performance del programma sarebbero state leggermente peggiori in quanto scrivere dati su un file é piú veloce che in un database ma, attraverso opportune interrogazioni, il database avrebbe risposto meglio a qualsiasi esigenza, inoltre sarebbe stato fruibile anche a macchine esterne al nostro cluster.

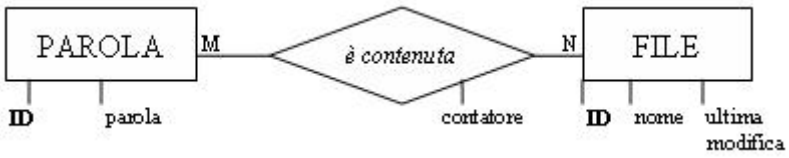

Figure 1: Schema concettuale del database

5.3 Aggiornamento dei dati ("Daemoning")

Fino a questo punto il programma cerca i file in una directory, li analizza e ne crea un profilo in base alle parole presenti nel dizionario, stampa i risultati e poi termina. Ma cosa succede se aggiungo una parola al dizionario o un altro file? Niente, finché non si lancia nuovamente il programma. Ho deciso allora di modificarlo in modo che, finito un ciclo riprendesse dall'inizio aggiornando i dati se vi erano state delle modifiche. Il programma aveva preso la forma di una sorta di "demone" (termine noto in ambiente linux che identifica un servizio che si avvia con l'accensione del PC e termina al suo spegnimento).

5.4 Reperimento autonomo dei file

Per avvicinarci un po' a quello che fa Google nella realtá é stata implementata anche la capacitá di reperire file da fonti diverse alla directory di cache selezionata, ad esempio siti internet, pagine web singole o perfino file conservati nel computer locale. Siccome non potevamo costruire un vero e proprio *spider*(gli agenti di Google incaricati di monitorare il WEB), abbiamo aggiunto una tabella al database nella quale saranno contenuti i lavori da svolgere. Il processo manager si incaricherá periodicamente di controllare questa tabella, andando a scaricare/copiare nella sua cache i file indicati, per poi passare alla loro analisi.

5.5 Interfaccia grafica

Quarto ma non ultimo sviluppo é stato l'introduzione di un'interfaccia web, scritta in PHP, accessibile da qualsiasi PC dotato di un browser e utilizzabile anche da persone a digiuno di SQL o Linux a riga di comando. Questa permette, oltre all'aggiunta di parole al dizionario e file alla cache, anche la possibilitá di visualizzare i risultati stilati dal programma in base, magari, al numero di ripetizioni di una parola o a un file specifico ecc.

6 Conclusione

Il programma é lungi da essere competitivo come il suo fratello maggiore e presenta ancora qualche difettuccio di aggiornamento ma, lo scopo per il quale é stato ideato, puó essere considerato piú che raggiunto.

7 Bibliografia

Testo di riferimento:
Michael J. Quinn, *Parallel Programming in C with MPI and OpenMP*,
Mc Graw Hill 2003

8 Installazione & Configurazione

In allegato a questo documento troverete anche tutto l'occorrente per far funzionare *Geegle*.

8.1 Requisiti

Ció che dovreste avere:

- Ambiente MPI con librerie MPICH2, su un sistema Linux.
- Server MySQL 5.0 o superiore, con le librerie *mysqlclient* installate su ogni macchina del cluster.
  (N.B.: Se si installa il pacchetto MySQL server sono giá incluse).
- Facoltativo: Server Apache con estensione PHP e MySQL per usare l'interfaccia WEB
- Un minimo di dimestichezza con il server MySQL (altrimenti provvedere a installare il phpMyAdmin: www.phpmyadmin.net).
  Un minimo di dimestichezza con l'ambiente Linux.

8.2 Predisporre il database

Attraverso mysql create un nuovo database che andrá a contenere le tabelle necessarie a Geegle, ad esempio:

```
shell> myslqadmin -u root -p -h host_name create geegle_db
```

Dentro la cartella *sql_script* troverete i file necessari alla predisposizione dell'ambiente. I nomi sono sufficentemente significativi, comunque ecco un piccolo riassunto:

`geegle_creation_db.sql`
Crea tutte le tabelle e le relazioni. É sufficente per far funzionare Geegle.
`geegle_fill_dict_base.sql`
Inserisce alcune parole nel dizionario.
`geegle_fill_dict_english.sql`
Inserisce un vocabolario limitato di parole inglesi.
`geegle_fill_dict_italian.sql`
Inserisce un vocabolario completo di parole italiane.
Attenzione: sono piú di 60000 parole, l'elaborazione potrebbe metterci un bel po'.
`geegle_empty_tables.sql`
Svuota le tabelle del Database.
Attenzione: non farlo quando il programma é in esecuzione, potrebbe crashare.
`geegle_elimination_db.sql`
Elimina le tabelle create, ma non il database.
`inserisci_permessi.sql`
Necessario solo se é presente anche il database SCP del progetto *Scheduler Layer*.

8.3 Compilare Geegle

Questi i passi da seguire:

8.3.1 Configurare il programma

La prima cosa da fare é configurare il programma con i vostri parametri di connessione al database. Per farlo cercate ed editate le seguenti righe di codice

```
/* DataBase connection parameters */
#define DBASE_HOST         "localhost"    //l'host su cui risiede il database
#define DBASE_USER         "studente"     //l'utente con cui accedere
#define DBASE_PASS         "studente"     //la password dell'utente
#define DBASE_NAME         "geegle_db"    //il database contenente tutto l'ambiente
```

Prestate attenzione al fatto che l'utente deve avere tutti i permessi necessari: creazione\eliminazione tabelle, inserimento\cancellazione dati, creazione tabelle temporanee, ecc.

8.3.2 Configurare lo script di avvio

Una volta configurato il programma si passa allo script di avvio, queste sono le varibili interessate:

```
#Variables
work_table="geegle_work"       #nome tabella lavori
file_table="geelge_file"       #nome tabella file
dict_table="geegle_dict"       #nome tabella parole
result_table="geegle_result"   #nome tabella risultati
cache_dir="cache_dir"          #nome directory di cache
```

Se non vengono cambiati i nomi delle tabelle negli script SQL che generano tutto l'ambiente, vi basterá modificare il nome della directory di cache, facendola puntare, magari, alla cache del vostro proxy o altro.

8.3.3 Avviare Geegle

Per far partire il motore di indicizzazione basterá dare un comando simile a:

```
shell> ./compile2.sh 5
```

dove *5* é il numero di processi su cui far girare Geegle. La shell resterá bloccata finché il programma non verrá terminato (vedere sez. successiva).
L'output generato dal programma viene scritto in un file chiamato *geegle.log* e puó essere consultato in tempo reale attraverso un comando del tipo:

```
shell> watch tail -n 15 geegle.log
```

8.3.4 Terminare Geegle

Essendo un demone, per terminarne l'esecuzione, é necessario inserire nella tabella lavori un job particolare, che abbia come `work_type` il valore `'die'`. La query che assolve a questo compito é la seguente:

```
mysql> INSERT INTO geegle_work (work_type) VALUES ('die');
```

P.S.: Se siete su una singola macchina potete terminare l'esecuzione anche digitando CTRL+C nella shell di lancio ma, citando un caro professore, "Non é sua madre".

8.4 Installare l'interfaccia grafica

Una volta che il programma funziona e vi siete stancati di arzigogolarvi con le query sql per vedere qualche risultato, é giunto il momento di installare *geegleAdmin*, l'interfaccia in PHP che semplificherá di gran lunga le cose.
Per farlo vi basterá semplicemente copiare tutta la cartella *geegleAdmin* nella root o in una qualsiasi cartella reperibile dal vostro server HTTP. Poi, a seconda che abbiate installato anche l'ambiente del progetto *Scheduler Layer* o meno, dovete avviare l'interfaccia dal file *index_sl.html* (se avete anche l'ambiente Scheduler Layer) oppure dal file *index_nosl.html* (se non avete l'ambiente).

8.4.1 Configurazione

Qualunque index scegliate, bisogna comunque configurare i parametri con cui l'interfaccia si collegherá al database. I valori da modificare si trovano nel file *login.php* e sono i seguenti:

```
/* setting up session variables */
$_SESSION['mysql_host'] = "localhost";         //l'host su cui risiede il database
$_SESSION['mysql_user'] = "studente";          //l'utente con cui accedere
$_SESSION['mysql_passwd'] = "studente";        //la sua password
$_SESSION['mysql_db_geegle'] = "geegle_db";    //il nome del database

/* solo se si utilizza il file index_sl.html */
$_SESSION['mysql_db_login'] = "scp";
/* questa variabile indica il nome del database del progetto scheduler da cui
   andare a prendere i dati relativi agli utenti */
```

Questi dati dovrebbero essere gli stessi inseriti anche nel programma Geegle. Una volta impostati siete pronti per usare il geegleAdmin. P.S.: L'interfaccia fornisce anche un feedback se il programma é in esecuzione o meno. Perché funzioni peró, il server HTTP dovrebbe essere su una delle macchine dove gira anche Geegle.

Contents