Un'architettura distribuita per la visita di alberi di gioco

Bruno Castellani

July 17, 2002

Contents

List of Tables

List of Figures

Chapter 1

Introduzione

Ha sempre destato un grande interesse la possibilità di applicare l'utilizzo di calcolatori a quella categoria di problemi che non possono essere ricondotti a delle formule matematiche. L'ambiente cui appartengono molti problemi di questo genere è quello dei giochi, cui tipici rappresentanti sono gli scacchi e la dama.

Infatti in tali contesti il calcolo di una mossa non è certamente riconducibile ad una semplice valutazione di carattere matematico, ma richiede la considerazione di più componenti. Queste devono condurre non tanto ad un traguardo immediato quanto al raggiungimento di un obiettivo più ampio come la vittoria di una partita. Il fascino di tale tipo di problemi viene accresciuto dalla possibilità di avere a disposizione elaboratori sempre più potenti e veloci, o addirittura di poter utilizzare reti di workstation. Infatti in questo progetto si intende costruire una architettura distribui-ta per la soluzione di una particolare categoria di problemi decisionali alla quale appartiene anche il gioco degli scacchi. In particolare vengono considerati quei problemi in cui:

- vi sono solamente due partecipanti;
- quanto perso da un partecipante equivale al guadagno dell'altro (0_sum);
- in qualsiasi momento ogni partecipante conosce tutte le soluzioni possibili (informazione perfetta).

Questo genere di problemi è ben rappresentabile mediante una struttura ad albero, detta albero di gioco, in cui i nodi identificano gli stati del gioco e

gli archi indicano quelle azioni che consentono di transitare da un nodo padre ad un nodo figlio. L'analisi dei problemi presentati consta di due componenti principali: la conoscenza e la ricerca. La conoscenza è costituita da un insieme di criteri che permettono di giudicare una determinata situazione di gioco. Tali parametri sono individuabili in ogni gioco e devono essere presi in considerazione tutti per poter esprimere una corretta valutazione. Ad esempio nel gioco degli scacchi sono stati individuati otto parametri principali ([Sch84]). La ricerca, invece, permette di analizzare le azioni possibili a partire da un nodo dell'albero e consente di sviluppare una strategia per il raggiungimento di un obiettivo prefissato.

Tali componenti sono assolutamente complementari. Infatti una conoscenza perfetta, che consentisse di valutare esattamente una situazione di gioco solamente sulla base di elementi contingenti, permetterebbe di non effettuare alcun tipo di ricerca sull'albero di gioco. Mentre, se fosse possibile analizzare l'intero albero di gioco fino alle foglie relative alle situazioni finali, si potrebbe rinunciare alla fase di valutazione sulla base della conoscenza.
Non essendo possibile affrontare il problema in alcuno dei due modi appena descritti, sia per la profondità dell'albero di gioco che per l'impossibilità di trovare una conoscenza in grado di valutare perfettamente una situazione, le componenti vengono utilizzate entrambe.

Per migliorare l'analisi di questa categoria di problemi si può sviluppare una architettura che permetta di rendere distribuita la fase della ricerca e, contemporaneamente, la conoscenza.
Con distribuzione della conoscenza si intende che un processo può valutare una situazione di gioco secondo criteri differenti da quelli di altri processi. In sostanza, ogni processo è dotato di una propria istanza di ricerca. Tale istanza costituisce sempre un sottoinsieme di tutti i parametri di giudizio relativi ad un determinato problema.
In [Sch84] è possibile trovare una serie di valutazioni sull'importanza dei diversi criteri di conoscenza nel gioco degli scacchi.
Tale tipo di parallelismo risulta comunque meno esplorato, mentre sono numerosi gli algoritmi già sperimentati che rendono parallela la visita di un albero di gioco ([Bro96]).

In [Toz93] e [San93] sono stati sviluppati dei sistemi che rendevano distribuite rispettivamente la ricerca e la conoscenza. Nel primo veniva sviluppato un progetto che prevedeva una visita dell'albero di gioco con distribuzione della

ricerca utilizzando i metodi più classici di decomposizione dell'albero. Venivano inoltre impostate le basi per degli sviluppi relativi alla distribuzione della conoscenza.
Nel secondo veniva raccolto questo spunto e realizzata un'architettura a conoscenza distribuita specifica per il gioco degli scacchi. In essa ogni processo analizzava l'albero utilizzando un proprio criterio di valutazione costituito da un sottoinsieme dei parametri individuati per il gioco.
Ogni processo coinvolto, proponeva la mossa giudicata migliore secondo la propria istanza e fra le mosse candidate veniva di seguito scelta la migliore. Per il raggiungimento della decisione finale venivano proposti una serie di criteri di selezione. Fra questi aveva mostrato particolare efficacia un criterio che prevedeva una nuova analisi dell'albero di gioco ristretto ai sottoalberi aventi come radice le mosse candidate. In ognuna di queste fasi ogni processo analizzava l'albero di gioco sequenzialmente. Per l'analisi dell'albero "ridotto" veniva quindi utilizzato un solo processo mentre tutti gli altri rimanevano inattivi.

1.1 Obiettivo del lavoro

Con questo lavoro si intende procedere in linea con i progetti descritti sopra, creando una opportuna architettura che concili la distribuzione della ricerca e della conoscenza. Anche in questo caso si utilizza il gioco degli scacchi come dominio di applicazione per lo sviluppo di tale architettura. Partendo infatti dalla versione sequenziale di un programma di dominio pubblico (Gnuchess 4.00 pl 77), si procede ad effettuare una serie di modifiche aventi lo scopo di renderlo distribuito. Intendiamo costruire un'architettura che preveda una prima fase di analisi dell'albero di gioco distribuendo la conoscenza, ed una seconda in cui viene rivalutato l'albero, ristretto alle sole mosse proposte.

A differenza di quanto avveniva in [San93], in cui ogni processo analizzava l'albero autonomamente, in questo lavoro si è tentato di sfruttare al meglio le potenzialità offerte dal sistema. Oltre alla conoscenza viene infatti resa parallela anche la ricerca.
In sostanza, viene creato un programma (denominato ChessPar) che associa ad ogni istanza un gruppo di processi anziché uno solo, in modo da poter applicare degli opportuni algoritmi di ricerca distribuita anche sulla prima fase di analisi.

Ogni gruppo viene coordinato da un processo supervisore, ed a loro volta i supervisori sono gestiti da un altro processo principale che si preoccupa di ridurre l'albero ai soli rami aventi come radice una delle mosse candidate. Successivamente tale processo diviene l'unico supervisore per la seconda fase di ricerca che viene resa distribuita. Infatti anche in questa fase tutti i processi devono collaborare analizzando ognuno la propria porzione di albero.
In tale fase il denominatore comune ai processi è la conoscenza che è completa per tutti.
L'architettura finale che ne deriva combina quindi i diversi tipi di parallelismo proposti in [San93] e [Toz93].
Lo strumento utilizzato per rendere parallela la versione sequenziale del programma, è il linguaggio di coordinazione Network C_Linda.
La sperimentazione verrà eseguita su un'architettura parallela composta da 20 Sun SparcStation.
Per la verifica dei risultati ottenuti sono stati disputati tornei fra le varie versioni di ChessPar ottenute, ed il programma di base Gnuchess4.00.

1.2 Struttura della tesi

Il percorso che viene proposto in questa tesi prende avvio dal programma sequenziale di gioco e, proponendo una serie di architetture intermedie, giunge fino alla descrizione del programma definitivo. Pertanto nel prossimo capitolo verrà analizzato il programma di base Gnuchess4.00 in quelle parti che serviranno in seguito per lo sviluppo di altre architetture. Si analizzeranno in modo particolare la funzione di ricerca e le strutture dati che servono per rappresentare l'albero di gioco.

Nel capitolo successivo verrà analizzato lo strumento utilizzato per la creazione dei processi coordinati. Vengono prese in considerazione le caratteristiche principali del linguaggio Network C_Linda (versione 3.2), con degli accenni alle sue successive evoluzioni (Piranha e Paradise).
A partire dal capitolo 4 verranno invece presentate le varie architetture distribuite che sono state sviluppate. Più specificatamente, in questo capitolo, verrà descritta la struttura del programma ottenuto con la sola distribuzione della conoscenza e solo nel capitolo successivo si procederà a combinarla con la distribuzione della ricerca.
Mentre in questi capitoli sarà condotta una valutazione prettamente quanti-

tativa dei giocatori ottenuti, nel capitolo 6 verranno effettuate considerazioni di carattere qualitativo con la discussione dei risultati dei tornei fra ChessPar, nelle sue diverse versioni, e Gnuchess4.00.

Chapter 2

Gnuchess4.00

2.1 Introduzione

Il programma sequenziale di gioco di scacchi che viene utilizzato all'interno di questo progetto è Gnuchess4.00 pl77.
Questo è un programma di pubblico dominio messo a disposizione dalla Free Software Foundation, alla cui realizzazione e miglioramento hanno contribuito e continuano a collaborare molti programmatori. A causa di queste frequenti modifiche è stata sacrificata la modularità del codice C e la documentazione.
Gnuchess permette la disputa di partite di scacchi contro altri giocatori o contro se stesso ed è in grado di combinarsi con l'interfaccia grafico xboard.
L'architettura generale di questo programma consta di un ciclo principale che prevede in sequenza, l'inserimento del comando da parte dell'utente e la selezione della mossa da parte dell'elaboratore. Inoltre ogni partita termina con la vittoria di uno dei giocatori, o con una patta o per la fine del tempo concesso.

2.2 Strutture dati principali

La struttura principale di un programma di scacchi è certamente l'albero di gioco. Considerate le ampie dimensioni dell'albero conviene rappresentarne solamente la porzione che interessa in un determinato momento. Gnuchess utilizza delle strutture idonee ad effettuare questo tipo di gestione.

La posizione da valutare viene collocata alla radice dell'albero e vengono quindi sviluppati solamente quei cammini che portano al nodo in esame.
La struttura dati che viene utilizzata per la memorizzazione di una mossa è :

```
struct leaf {
  short  f;
  short t;
  short score;
  short reply;
  short width;
  unsigned short flags;
 }
```

All'interno di "leaf" vi sono tutti quei valori necessari per caratterizzare una mossa: f (from) indica la casella di provenienza della pedina; t (to) indica quella di arrivo; score mantiene il punteggio relativo alla mossa effettuata e quindi una serie di altri campi comunque necessari.
L'albero è rappresentato da due array:

struct leaf Tree[1500];
short TrPnt[15].

Il vettore "Tree" mantiene la rappresentazione di tutte le mosse che compongono il cammino dalla radice al nodo in esame. TrPnt indica da quale elemento di "Tree" iniziano le mosse associate ad un nodo.
Se un nodo appartenente al cammino è situato al livello "ply", si ha che le mosse possibili a partire da questo nodo sono quelle contenute nell'intervallo di albero compreso fra $TrPnt[ply]$ e $TrPnt[ply+1]-1$.

2.3 Selezione della mossa

La selezione della mossa da parte di Gnuchess avviene secondo l'algoritmo Aspiration Search in combinazione con l'algoritmo alpha_beta in versione Fail Soft ([Cia92]).

Molto brevemente, la scelta del cammino migliore lungo l'albero di gioco, si basa sul principio che la mossa migliore per un giocatore è la mossa peggiore per l'avversario (0_sum). L'algoritmo di visita alpha_beta costruisce il proprio cammino lungo l'albero di gioco supponendo che al proprio turno l'avversario giochi la propria mossa migliore. Gnuchess utilizza questa tecnica per visitare l'albero.
Per poter ogni volta realizzare quale sia la propria mossa migliore, bisognerebbe analizzare tutto l'albero di gioco. È possibile stabilire un range iniziale per la ricerca ($\alpha\beta\_window\_search$), in modo che possano essere esclusi a priori dalla visita alcuni sottoalberi che certamente non porterebbero ad alcun miglioramento del punteggio globale.
Ovviamente più è ristretto questo intervallo di ricerca, minore è la porzione di albero che deve essere analizzata. Lo score finale della mossa che viene selezionata dovrà rientrare in questa finestra di ricerca.
Per poter applicare questi tagli è necessario settare questa finestra su dei valori adatti alla situazione. Gli estremi α e β devono appartenere ad un intorno dello score che si prevede per la mossa che verrà selezionata.
Prima di iniziare la ricerca avviene una sommaria valutazione dello score in modo da poter fissare gli estremi dell'intervallo. Questo verrà aggiornato nel corso della valutazione in modo da poter applicare tagli sempre più significativi ai rami dell'albero.
Se al termine della ricerca non dovesse trovarsi alcuna mossa all'interno della prevista "$\alpha\beta\_window\_search$", l'algoritmo (variante Fail Soft) prevede che debba essere ripetuta la ricerca, settando però, la finestra iniziale a dei valori più ampi $\alpha = -\infty$ e $\beta = +\infty$.
Come nel caso precedente l'intervallo si restringe proseguendo nell'analisi dell'albero.
Entrando più in dettaglio, Gnuchess contiene una funzione di ricerca principale "search(...)" che, invocata ricorsivamente a profondità successive, seleziona alternativamente la mossa migliore o peggiore.
Qui di seguito vengono illustrati i passi principali di questa funzione in quanto, nel corso della realizzazione di questo lavoro, sarà oggetto di cambiamenti.

```
int search (side, ply, depth, ext,
     alpha, beta, bstline, rpt,
```

```
     quiescent, didnull)
{
/* Sommaria valutazione dello score */
score = evaluate (side, ply, depth, ext,
               alpha, beta, &terminal, &InChk);

/* Aggiunta del livello successivo dell'albero   */
/* L'albero viene costruito mano a mano che si analizza */
if (ply >1)
      MoveList (side, ply);

/* Si aggiustano gli estremi della finestra di ricerca */
if (nmscore > alpha)
     alpha= nmscore;
...
  if (best > alpha)
      alpha = best;
/****** main loop ************************/
  /* look at each move until no more or beta cutoff */
  for (pnt = pbst = TrPnt[ply]; pnt < TrPnt[ply + 1]
            && best <= beta && best != 9999-ply; pnt++)
    {
/* find the most interesting looking of the
                                        remaining moves */
      if (ply > 1)
pick (pnt, TrPnt[ply + 1] - 1);
      mv = &Tree[pnt];
      if (!(node->flags & exact)) {
     /* Esegue la mossa e prosegue in profondita' */
        MakeMove (mv);
        ...
        if (flag.pvs && depth > 0) {
           if (pbst == pnt) {
           mv->score= -search (xside, ply + 1,
               depth > 0 ? depth - 1 : 0, ext,
                -beta, -alpha,
                nxtline, &rcnt, quiescent, 0);
```

```
            } else {
           mv->score= -search(xside, ply + 1,
              depth > 0 ? depth - 1 : 0, ext,
              -alpha-1, -alpha,
               nxtline, &rcnt, quiescent, 0);
           if (mv->score >= best && alpha <= mv->score
              && mv->score <= beta)
            mv->score = -search (xside, ply + 1,
                   depth > 0 ? depth - 1 : 0, ext,
                    -beta, -mv->score,
                    nxtline, &rcnt, quiescent, 0);
         }
       } else
          mv->score = -search (xside, ply + 1,
              (depth > 0) ? depth - 1 : 0, ext,
               -beta, -alpha,
                nxtline, &rcnt, quiescent, false);
             ....
/* Viene disfatta la mossa che appariene al cammino */
UnmakeMove (mv);
}
/* Aggioranamento alpha prima di rifare l'analisi */
      if (best > alpha) { alpha = best; }
 }
/********** out of main loop **************/
...
}
```

La funzione "search(...)" restituisce il punteggio della mossa selezionata.
La funzione MakeMove(mv) aggiunge la mossa "mv" al cammino principale per poi proseguire in profondità nell'albero.
Ad ogni invocazione della "search" viene aggiornato l'albero costruendo il nuovo livello con tutte le mosse possibili a partire da 'mv'.
La prima invocazione della funzione "search(..)" avviene da una procedura superiore che si preoccupa di cambiare la finestra di ricerca se la prima analisi non è andata a buon fine. Questa funzione è "SelectMove(...)" il cui nucleo

è costituito da un ciclo che prosegue fino a quando non termina il tempo a disposizione.
Viene fissata una profondità massima iniziale fino alla quale l'albero verrà certamente analizzato. Qualora al termine di questo primo ciclo di valutazione dovesse esserci ancora tempo allora si tenterebbe di analizzare l'albero ad una profondità maggiore sfruttando i risultati del lavoro appena compiuto (Iterative deepening).

```
void SelectMove (side,  iop)
{
/* Esamina la posizione e setta un valore
   iniziale per lo score                              */
ExaminePosition ();
start_score= Tscore[0]= Tscore[1]= score=
    evaluate (side, 1, 1, 0, -9999, 9999,
          &terminal, &InChkDummy);

/* setta la finestra iniziale di ricerca */
  alpha = score - ((computer == black) ?
      BAwindow : WAwindow);
  beta = score + ((computer == black) ?
      BBwindow : WBwindow);

/* Crea l'albero */
  VMoveList (side, 1);

/* fissa una profondita' iniziale */
Sdepth = (MaxSearchDepth < (MINDEPTH - 1)) ?
     MaxSearchDepth : (MINDEPTH - 1);

/****** main loop ************/
while (!flag.timeout)
  {
   Sdepth++;

/* Prima ricerca */
```

```
   score = search (side, 1, Sdepth, 0,
    alpha, beta, PrVar, &rpt, QBLOCK, false);

/* se il punteggio trovato esce dalla finestra
 prevista si fa una nuova ricerca */
 if (score < alpha)
/* I nuovi estremi sono -e+ INFINITO */
   score = search (side, 1, Sdepth, 0,
     -9999, 9999, PrVar, &rpt,QBLOCK,false);

else if (score > beta && score != 9998)
   score = search (side, 1, Sdepth, 0,
    -9999, 9999, PrVar, &rpt,QBLOCK,false);

/* Ricalcola la finestra di ricerca */
  beta = score + ((computer == black) ?
     BBwindow : WBwindow);
  alpha = score - ((computer == black) ?
     BAwindow : WAwindow);
  }
  /***** end of main loop ********/
 ....
}
```

Queste funzioni sono la base da cui si prendono le mosse per tutte le revisioni nel corso del progetto.

Chapter 3

Introduzione alla programmazione coordinata

I modelli classici di programmazione distribuita sono due:

- il modello a scambio di messaggi;
- il modello con strutture dati condivise.

Nel primo caso i processi possono comunicare tra loro solamente scambiandosi messaggi, dove ogni messaggio deve essere costituito da un mittente, un destinatario, ed un corpo contenente tutti i dati che devono essere resi noti a tutti. Nel secondo caso esistono delle variabili comuni a tutti i processi, alle quali tutti possono accedere, ma lasciando al programmatore il gravoso compito di gestirne le operazioni di lettura e scrittura.
Il modello di Linda deve essere collocato al di fuori degli schemi appena illustrati, in quanto non utilizza né il sistema a scambio messaggi, né quello con strutture dati distribuite, ma mette a disposizione dei processi un ambiente comune detto spazio delle tuple. Qui ogni processo può depositare, leggere e modificare gli elementi che sono presenti. Questi elementi vengono definiti tuple.
Il modello di Linda prende il nome di comunicazione generativa, in quanto per poter comunicare vengono generate delle tuple che potranno poi essere maneggiate dagli altri processi. Esso non è però un linguaggio di programmazione in quanto, per poter essere utilizzato necessita di un linguaggio di base che viene arricchito dalle operazioni che Linda mette a disposizione.

Linda è in grado di combinarsi con il C e con il Fortran 77.
Pertanto Fortran_Linda e C_Linda, utilizzato in questo progetto, funzionano come linguaggi di coordinazione e forniscono a processi distinti gli strumenti necessari per comporsi in un programma parallelo.

3.1 Spazio delle tuple

Lo spazio delle tuple è l'ambiente di memoria condiviso logicamente dai processi che lo devono utilizzare per poter comunicare. Esso non ha una collocazione precisa all'interno della rete di elaboratori utilizzati, ma risiede parzialmente su ognuno di essi. La gestione di questo ambiente è trasparente all'utente ed è logicamente indipendente dall'architettura del sistema.
Un processo può collocare in tale spazio un numero teoricamente infinito di elementi, che, una volta depositati, perdono ogni riferimento al processo che li ha generati e sono a disposizione di tutti gli altri indistintamente. Non esiste inoltre un ordinamento fra le tuple presenti.
Lo spazio delle tuple è in sostanza, un multinsieme in cui possono coesistere elementi diversi o anche più elementi assolutamente identici.
Le tuple sono le strutture dati dello spazio delle tuple e sono queste ad essere elaborate dai processi. Le tuple sono delle sequenze ordinate di campi tipati, al massimo 16, e non hanno alcun indirizzo all'interno dello spazio di memoria condiviso. Ogni elemento può essere distinto solamente in base al contenuto. Ad esempio la tupla:

("job", A, 2)

può essere indicata come la tupla di tre campi che ha

- come primo elemento "job";
- come secondo elemento la variabile A, il cui valore è quello corrente;
- come terzo elemento il valore 2;

oppure combinandone più caratteristiche.
Ogni riferimento si basa quindi sulla corrispondenza dei contenuti fra i rispettivi parametri. Ogni tupla può contenere i seguenti elementi:

```
int, long, short, char (signed o unsigned);
float e double;
struct o union;
array di qualsiasi dimensione;
```

Ogni parametro può essere attuale o formale. Con attuale si indica un elemento che ha un valore preciso sia esso una variabile oppure esplicitamente un numero o una stringa, mentre con formale si intende una variabile che non ha un valore ma che attende di riceverne uno. Tali variabili devono essere precedute da un '?'. Ad esempio la tupla

("parameters", A, 2, ?int, ?var)

contiene tre parametri attuali e due formali.
Tuple contenenti parametri formali vengono definite 'antituple' in quanto vengono utilizzate in fase di lettura (v. operazioni *rd* e *in*), per creare una correlazione con le tuple già presenti in memoria.
Ai parametri formali vengono assegnati i valori corrispondenti della tupla con la quale è avvenuto il match.
La corrispondenza fra una antitupla 'tmpl' ed una tupla 't' si ha quando:

- entrambe hanno lo stesso numero di campi;
- il tipo, i valori e la lunghezza dei parametri attuali dell'antitupla 'tmpl' sono gli stessi dei corrispondenti campi della tupla 't';
- il tipo e la lunghezza dei parametri formali di 'tmpl' corrispondono a quelli dei rispettivi campi di 't'.

Esiste anche un parametro formale, detto anonimo, che si utilizza quando si vuole che vi sia corrispondenza fra due campi di una tupla, senza che però ne derivi alcun assegnamento di valore all'elemento dell'antitupla. Nell'antitupla precedente il parametro formale '?int' è anonimo in quanto non si utilizza una variabile ma solamente il tipo.
Si noti che il primo elemento di una tupla è costituito da una stringa costante. Ciò non costituisce una regola, ma è solo una consuetudine di buona programmazione in C_Linda, affinché ogni tipo di tupla possa essere immediatamente caratterizzato dal primo elemento, con ovvi miglioramenti sulle prestazioni del sistema.

3.2 Operazioni

Le operazioni con le quali i processi possono interagire con lo spazio delle tuple sono quattro.

out(tpl) : la tupla tpl viene aggiunta nello spazio delle tuple ed il processo che l'ha invocata prosegue immediatamente. 'tpl' può anche contenere delle chiamate ad altre funzioni, come ad esempio

out("calcolo", i, j, w, f(i,j,w)) ;

in questo caso il processo che sta chiamando "out" valuta f(i,j,w) prima di depositare la tupla.

eval(tpl) : la tupla 'tpl' viene anche in questo caso aggiunta allo spazio delle tuple, ma essa deve contenere una chiamata ad una funzione. In questo caso, però, la valutazione viene demandata ad un altro processo, costringendolo a mettersi al lavoro per il calcolo della funzione.
Il processo che si occupa di questa computazione viene creato al momento della chiamata ad 'eval', viene cioè generato appositamente e terminerà una volta conclusa la propria elaborazione. Il processo che ha invocato "eval" prosegue immediatamente senza attendere la terminazione delle computazioni che ha generato. La tupla 'tpl' prima della sua valutazione viene definita attiva, in quanto, come già detto, costringe altri processi ad interagire con essa. Una volta terminata la valutazione, il risultato andrà a sostituire la chiamata all'interno della tupla, che così diventerà passiva. Riprendendo l'esempio di prima, si ha che la chiamata

eval("calcolo", i, j, w, f(i, j, w))

costringe C_Linda a creare un nuovo processo che esegua 'f(i, j, w)' e che termini alla fine di questo calcolo.
Supponendo che 'f(i, j, w)' restituisca valore 4, la tupla attiva ("calcolo", i, j, w, f(i,j,w)), precedentemente depositata, diventerà la tupla passiva ("calcolo", i, j, w, 4). Le variabili passabili con 'eval' possono essere solamente dei tipi base e non strutture.

L'ambiente noto al processo appena creato è limitato alle variabili passate esplicitamente come argomenti all'interno di eval(). Qualora fossero necessari altri elementi, potrebbero essere comunicati con l'ausilio dell'operazione 'out()'.

in(tmpl) : tenta di rimuovere dallo spazio delle tuple una tupla che faccia match con l'anti-tupla 'tmpl', secondo le regole esposte nel paragrafo precedente.
L'operazione 'in(...)' è bloccante, il processo che l'ha invocata rimane sospeso finché non trova una tupla con le caratteristiche cercate. Una volta trovata, essa viene rimossa dallo spazio in memoria ed ogni parametro formale viene settato al valore del corrispondente campo nella tupla. Ad esempio:

in("valori", ?val_1, ?val_2)

significa che il processo sta aspettando che sopraggiunga una tupla ("valori", num1, num2) per poi rimuoverla e settare rispettivamente

val_1 = num1 e val_2 = num2;

invece

in("valori", ?val_1, 3)

indica che il processo attende una tupla dello stesso tipo della precedente, ma il cui terzo elemento deve valere 3.

rd(tmpl) : analogo a 'in' con la differenza che questa operazione non rimuove la tupla dalla memoria condivisa.

Esistono poi delle chiamate simili alle precedenti, ma che, al contrario di queste, non sono bloccanti, in particolare:

inp(tmpl) : come 'in' rimuove la tupla, però, al contrario, non è bloccante infatti restituisce 1 se è riuscita a fare match, 0 altrimenti;

rdp(tmpl) : analogo a 'rd' e con gli stessi output di 'inp'.

Qualora esistano più tuple con le quali 'tmpl' può fare match, ne verrà scelta una secondo un criterio non deterministico.
L'ambiente in cui si trovano a lavorare i processi creati tramite le chiamate 'eval' è ben definito solamente per quanto riguarda le variabili che vengono passate esplicitamente attraverso la chiamata, mentre per quanto riguarda le altre variabili, anche globali, il comportamento non è definibile. Questo non crea però problemi in quanto tutte quelle variabili di cui un processo necessita e che non sono passabili con 'eval', possono essere trasmesse con una o più tuple tramite l'operazione 'out'.
Network C_Linda prevede una gestione completamente autonoma e quindi trasparente al programmatore, dell'accesso all'area di memoria condivisa che altrimenti comporterebbe un lungo e difficoltoso utilizzo dei semafori.

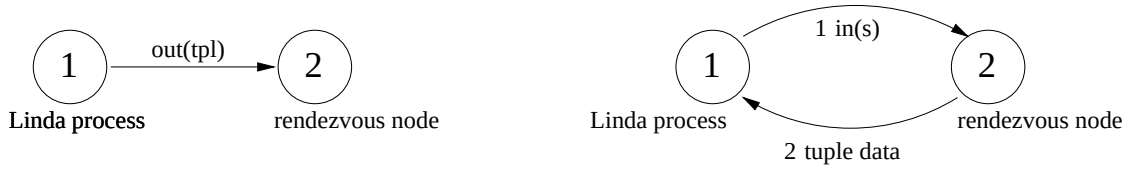
3.3 Implementazione di Network C_Linda

L'implementazione di Linda consta di tre componenti principali (v. [AB89]):

1. un precompilatore;
2. un ottimizzatore a link-time;
3. una serie di librerie run-time.

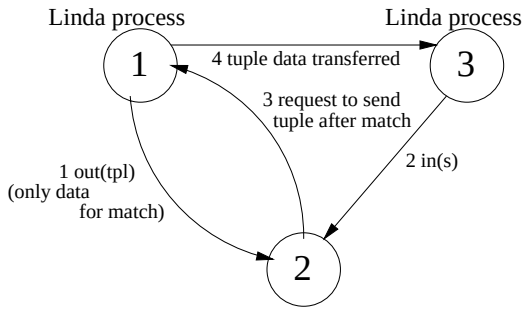
Il precompilatore si preoccupa di trasformare le istruzioni di Linda, in codice del linguaggio base(C o Fortran), nel quale le operazioni sullo spazio delle tuple vengono sostituite da chiamate a funzioni delle run-time library.
Il prelinker analizza completamente quello che dovrà essere lo spazio delle tuple e ne costruisce una prima ottimizzazione.
Partendo infatti dalla semplice osservazione che tuple di 'n' elementi potranno trovare una corrispondenza solo con tuple di altrettanti elementi dello stesso tipo, il prelinker crea una classe di equivalenza per ogni tipo di tupla.
In questo modo ogni operazione sullo spazio delle tuple può essere ridotta ad una invocazione ad una funzione che maneggia solamente quella specifica classe.
A tempo di esecuzione ogni classe viene assegnata ad un nodo, che dovrà

gestire tutte le chiamate a quella porzione di spazio di tuple e memorizzarne le tuple ed antituple(v. fig.3.1).
L'attuale versione di C_Linda(versione 3.1) prevede che ad ogni nodo vengano attribuite una o più determinate classi C[t] di tuple. Da questo deriva che, quando un processo fa un out(tpl) dove tpl $\in$ C[t], 'tpl' viene memorizzata nel nodo N(C[t]) che ha in gestione quella classe.



a. out for a small tuple

b. in for a small tuple



c. out-in pair for a large tuple

Figure 3.1: Gestione delle Tuple

Quando un processo fa invece un in(s) con s $\in$ C[t], la domanda viene inoltrata al nodo N(C[t]), che restituisce la tupla cercata al processo richiedente. Questo è valido, però, solamente nel caso di tuple di piccole dimensioni. Per tuple 's' più grandi, che comportano costi di trasmissione da un nodo all'altro piuttosto onerosi, non avviene la memorizzazione nel nodo dedicato N(C[t]). Esse infatti rimangono memorizzate nel nodo su cui è attivo il processo 'p' che le ha generate, mentre al nodo N(C[t]) vengono comunicati solo gli estremi di queste tuple.
Qualora un processo voglia ricevere in input la tupla 'tpl', invierà comunque

la domanda al nodo N(C[t]) che, confrontata la richiesta con gli estremi di 's' che gli sono noti, demanderà a 'p' il compito di trasmettere la tupla 's' al nodo che ne necessita. A tempo di esecuzione, qualora Linda si renda conto che un processo, fa più richiesta di un certo tipo di tupla piuttosto di altri, si preoccuperà di trasferire la gestione di questa porzione di spazio di tuple proprio al nodo su cui è attivo questo processo.
guadagno Le classi di tuple più richieste, se ciò non richiede troppo spreco di memoria, vengono memorizzate in più nodi, in modo che si possa accedervi immediatamente.

dello spazio equivalenza, la tupla su cui è attivo il di tipo globale, i nodi, fra ottimizzazione come

3.4 Paradigma MasterWorker

Un'architettura tipica di programmazione coordinata è quella definita Master-worker ed è questa che verrà utilizzata nel corso di questo progetto.
Essa consta di un processo principale (Master), che ha il ruolo di creare, coordinare e raccogliere i risultati di tutta una serie di altri processi (worker), il cui compito è quello di prendersi il carico della computazione.
Di seguito verrà analizzato questo modello di programmazione utilizzando frammenti di codice scritti in pseudo C_Linda.
Tipicamente il processo master è attivo sulla macchina dalla quale viene fatto partire il programma, mentre gli altri worker girano sulle altre macchine della rete.
Il paradigma tipico è il seguente:
nella prima fase il processo master crea tutti i processi worker che sono ritenuti necessari tramite una serie di chiamate alla funzione 'eval' di C_Linda,

```
void real_main(argc, argv)
{
     ...
     /* Generazione dei worker */
     for(i=0; i<nworkers; i++)
            eval("worker", func(i));
     ...
```

quindi si occupa della raccolta dei risultati:

```
      ...
      /* Raccolta dei risultati */
      for(i=0; i<nworkers; i++)
          in("worker", result);
     ...
}
```

Si noti che i processi worker possono anche essere differenti tra loro.
Con una struttura di questo tipo, per il master non è possibile contraddistinguere il risultato di un processo dal risultato di un altro, a tale scopo sarebbe necessario dotare ogni worker di un proprio identificatore così che ogni processo sia immediatamente distinguibile. Lo schema precedente diventerebbe perciò

```
void real_main(argc, argv)
{
        ...
        /* Generazione */
       for(i=0; i<nworkers; i++)
            eval("worker", ident[i], func(i));
       ...
       /* Raccolta dei risultati */
       for(i=0; i<nworkers; i++)
          in("worker", ident[i], ?result);
       ...
}
```

dove abbiamo creato una sistema di scambio di comunicazioni da 1 a 'n' (creazione dei processi) e da 'n' a 1 (fase di raccolta dei risultati).

Creare un processo per ogni fase di calcolo non è ovviamente la soluzione migliore, in quanto tale generazione può risultare molto costosa in termini di tempo con conseguenze disastrose sull'intera esecuzione.
Conviene quindi creare dei processi polivalenti in grado di eseguire più tipi di valutazione restituendo ogni volta il risultato della propria computazione senza terminare essi stessi, se non al termine di tutta l'esecuzione.

A questo tipo di architettura fa riferimento anche il progetto di visita distribuita di alberi di gioco.
Terminata la generazione può essere conveniente che lo stesso master diventi un worker così da poter contribuire al calcolo ed utilizzando il tempo di macchina che altrimenti potrebbe venire sprecato nell'attesa dei risultati dai worker.

3.5 Piranha

Per completezza in questa breve trattazione del modello Linda, ne analizziamo la recente evoluzione che è costituita da Piranha, che ha le stesse caratteristiche del suo predecessore.
Sfrutta anch'esso lo spazio delle tuple per la comunicazione fra i processi ed ha le medesime chiamate, costituisce però un sistema più flessibile rispetto a Network C_Linda.
Infatti lo scopo del modello Piranha è quello di sfruttare i tempi di CPU liberi delle varie macchine che compongono la LAN. Il numero di processi coinvolti può quindi variare nel corso della computazione a seconda del livello di utilizzo di ogni macchina.
Un processo può quindi iniziare il proprio ciclo su una macchina, ma, qualora il carico su questo nodo diventi eccessivo, il lavoro viene sospeso e ripreso su una elaboratore più libero.
A differenza di C_Linda, Piranha non prevede l'operazione 'eval' per la creazione dei processi, in quanto la sua architettura si basa sul concetto di cooperazione secondo il paradigma Master-Worker appena illustrato.
I processi worker non vengono perciò creati tramite la chiamata 'eval' come in precedenza, ma essi sono già previsti all'inizio dell'elaborazione.
L'architettura di questo sistema consta di tre funzioni principali:

feeder è la funzione utilizzata dal processo master che è attivo sulla macchina dalla quale viene dato inizio al programma, ed è suo compito distribuire

i lavori e raccogliere i risultati.
Questo è il solo processo che non può essere sospeso.

piranha è quello che prima è stato definito come processo worker;

retreat è la funzione che viene invocata quando un processo Piranha deve essere sospeso. Sarà sua preoccupazione depositare nello spazio delle tuple i dati necessari affinché possa essere ripresa l'esecuzione del processo su un altro nodo.

Naturalmente in un programma possono esistere delle sezioni critiche al cui interno non è possibile sospendere il programma. Per questo tipo di situazioni il sistema Piranha mette a disposizione due ulteriori funzioni:

disable_retreat indica l'inizio di una sezione di programma al cui interno non possono avvenire sospensioni(retreat);

enable_retreat analogamente indica il termine della sezione critica.

Ovviamente l'utilizzo di Piranha comporta la costruzione di una differente architettura del programma. Mentre infatti con C_Linda è possibile in ogni momento creare nuovi processi, ognuno con competenze anche molto diverse a seconda delle necessità, con Piranha è invece necessario creare dei processi che siano molto flessibili in grado di eseguire molti lavori.
Si consideri ad esempio un processo principale (master, feeder), che necessiti di parallelizzare due fasi di calcolo:
- con C_Linda sarebbe sufficiente inserire nel codice due chiamate a 'eval'

```
void real_main(argc, argv)
       ...
       eval("calcolo1", var_1, var_2,...var_n,
            funct1(var_1, var_2,...var_n));
       ...
       eval("calcolo2", var_1, var_2,...var_n,
             funct2(var_1, var_2,...var_n));
       ...
```

in modo tale da creare due processi dedicati esclusivamente alla valutazione delle relative funzioni;
- con Piranha invece bisognerebbe progettare un sistema diverso, costruendo un processo worker(piranha) capace di calcolare sia 'funct1' che 'funct2'. Tali processi dovrebbero comunque essere avviati dal master (feeder).
Uno schema plausibile potrebbe essere:

```
void feeder(argc, argv)
{
       ...
      out("task", wrk1);
      ...
      out("task", wrk2);
      out("parameters", var_1, var_2,...var_n);
      ...
      in("result", wrk1, ?result);
      in("result", wrk2, ?result);
}

void piranha()
{
      ...
     while(1)                /* il worker deve essere
    {                          continuamente in esecuzione */
         in("task", ?job)
         switch(job)
            case wrk1:
                   in("parameters1", ?var_1, ?var_2,...?var_n);
                   enable_retreat();
                   result = funct1(var_1, var_2,...var_n);
                   disable_retreat();
                   ...
                   break;
           case wrk2:
```

```
                in("parameters2", ?var_1, ?var_2);
                enable_retreat();
                result = funct2(var_1, var_2,...var_n);
                disable_retreat();
                ...
                break;
          default:
                ...
         /* Viene ora restituito
          il risultato della computazione *\
       out("result", job, result);

    }
}

retreat()
{
      out("task", job);
      switch(job)
            case wrk1:
                out("parameters1", var_1, var_2,...var_n);
            case wrk2:
                out("parameters2", var_1, var_2);
}
```

La struttura che è stata costruita con il modello Piranha, esige una gestione più complessa rispetto al modello C_Linda, ma essa risulta analoga a quella del paradigma master-worker illustrato in precedenza.
Molto brevemente facciamo alcune considerazioni sull'utilizzo delle funzioni di abilitazione e disabilitazione delle interruzioni.
Enable_retreat viene collocata prima dell'inizio della fase di calcolo che quindi diventa la sola fase realmente sospendibile. Non appena terminata la computazione, la funzione 'retreat' viene disabilitata per evitare che debba intercorrere una interruzione a calcolo già avvenuto ma prima dell'output del risultato. Ciò comporterebbe la ripetizione dell'intero lavoro.
La sospensione rimane disabilitata fino a quando il processo non ha letto i

dati necessari per iniziare una nuova fase. Questo è necessario in quanto l'intervento di un 'retreat' in questa fase potrebbe creare dei problemi di inconsistenza. ancora da fare. Il ruolo del retreat è semplicemente quello di ripristinare nello spazio delle tuple una tupla identica a quella prelevata dal processo piranha ora sospeso. Si avrebbero problemi anche se intervenisse una interruzione fra l'istruzione

in("task, ?job);

e

in("parameters1", ?var_1, ?var_2,...?var_n).

La funzione 'retreat' tenterebbe di ripristinare le tuple

("task", job) e ("parameters1", var_1, var_2,...var_n);

senza che quest'ultima, però, sia stata ancora letta con conseguenze così indefinite.
Naturalmente se si creassero sezioni critiche molto lunghe, si verrebbe a perdere l'utilità di un sistema come quello di Piranha; è quindi buona regola creare sezioni critiche brevi, ma con salvataggio dei risultati parziali in modo tale da poter ripristinare la computazione ad una fase il più avanzata possibile.

3.6 Paradise

Nel contesto della programmazione coordinata accenniamo ad una ulteriore evoluzione del modello di Linda: Paradise.
Paradise consente di creare e gestire più spazi delle tuple contemporaneamente ed ogni processo può maneggiarne più di uno. In questo modo un gruppo di processi può avere il proprio spazio di memoria dedicato, senza dover utilizzare un unico spazio delle tuple.
Questi spazi di memoria condivisa (VSM Virtual Shared Memory), hanno inoltre la peculiarità di poter rimanere allocati anche per lungo tempo senza che alcun processo vi stia agendo. Paradise consente di coordinare più processi che vengono eseguiti anche a distanza di mesi l'uno dall'altro.

Questo non era assolutamente possibile in Linda dove lo spazio delle tuple era unico ed i processi dovevano essere eseguiti contemporaneamente.
I programmi coordinati possono essere molto diversi ed assolutamente indipendenti. Non accade, come nel modello Linda, che ci sia un Master che deve generare gli altri processi worker.
Con Paradise i programmi coordinati possono essere molto diversi e coesistono in modo assolutamente indipendenti dagli altri. Il solo legame è costituito dall'accesso al medesimo spazio delle tuple.

3.6.1 Spazio delle tuple in Paradise

Lo spazio delle tuple può essere di due tipi : *persistente* oppure *non persistente*. Il primo è quello che consente ai processi di collaborare anche a distanza di tempo, esso infatti permane allocato fino a quando uno dei programmi non lo distrugge esplicitamente.
Il secondo viene disallocato quando termina l'ultimo dei programmi che poteva accedervi. Questo tipo di spazio delle tuple viene utilizzato per processi che sono attivi simultaneamente ed hanno necessità di condividere immediatamente dei dati.
Ogni spazio delle tuple deve essere creato. Per generarne uno, un programma deve innanzitutto aprire lo spazio delle tuple "radice", con l'operazione

open@roots()

e quindi creare il nuovo spazio mediante l'invocazione alla funzione "create".

create@rootts(type, label) : genera un nuovo spazio delle tuple del tipo "type" (PERSISTENT oppure NONPERSISTENT), associabile ad una label.
Questa operazione restituisce un identificatore dello spazio delle tuple. Questo valore viene utilizzato in tutte le operazioni per individuare univocamente lo spazio su cui operare.

I programmi che accedono solamente ad uno spazio senza esserselo creato, prima di poterlo manipolare devono ottenere il permesso di maneggiarlo e quindi aprirlo. Possono infatti esistere dei programmi che possono lavorare solo su alcuni spazi e non su altri.

Le operazioni necessarie sono le seguenti:

lookup_handle(string1, string2, myts): permette ad un programma di ottenere l'identificatore (myts) di uno spazio delle tuple individuabile dalle due stringhe passate come argomento (string1, string2);

open@myts: apre lo spazio delle tuple che ha come identificatore "myts";

close@myts: chiude lo spazio delle tuple una volta terminato il lavoro su di esso. Se lo spazio delle tuple è non_persistente e il processo che esegue la close è l'ultimo che vi doveva lavorare, allora lo spazio "myts" viene disallocato.

3.6.2 Operazioni sullo spazio delle tuple

Le operazioni sono analoghe a quelle di C_Linda con la differenza che ad ognuna di esse deve essere aggiunto l'identificatore dello spazio su cui lavorare. I rimanenti parametri rimangono invariati:

rd@myts(..): esegue la lettura di una tupla dallo spazio "myts";

in@myts(..): preleva una tupla da "myts";

out@myts(..): deposita una tupla.

Non compare l'operazione "eval()" che invece era fondamentale in C_Linda. Questo avviene in quanto i programmi Paradise esistono senza che un processo supervisore li debba creare.

Chapter 4

Architettura del programma ChessPar con distribuzione della conoscenza

4.1 Introduzione

In questo progetto viene costruita un'architettura per l'analisi di alberi di gioco, che combina una prima di fase con distribuzione della conoscenza ad una seconda di distribuzione della ricerca. L'approccio ad una ripartizione della conoscenza venne ispirato da Schaeffer [Sch84] e già in [San93] è possibile trovare un'applicazione di queste ricerche ad un programma che prevede la visita di alberi. Questo era un programma sequenziale di gioco di scacchi di pubblico dominio Gnuchess (versione 4.00 pl 60).
Questo lavoro prende le mosse proprio da questa prima applicazione, aggiornando le considerazioni fatte in [San93] su una versione aggiornata dello stesso programma (Gnuchess 4.00 pl77) di due anni successiva alla precedente.
Nel capitolo successivo verranno applicate delle tecniche di distribuzione della ricerca a questa architettura di base.

4.2 Conoscenza negli scacchi

I programmi per il gioco degli scacchi rientrano nella categoria di programmi basati su conoscenza in cui i criteri euristici di selezione di una mossa sono molteplici.

In particolare per valutare la bontà di una mossa si sono individuate otto euristiche principali che sono le seguenti:

- Materiale (m): questo è il criterio principale per giudicare una mossa, è infatti risaputo che ogni pezzo ha un diverso valore, è quindi di fondamentale importanza tenere sempre in gioco le pedine più importanti. Tale criterio prescinde dalla collocazione delle pedine sulla scacchiera, si tratta solamente di una valutazione quantitativa e non qualitativa;

- Sistemazione dei pezzi(b): questa euristica prende in considerazione la sicurezza dei pezzi, vengono infatti valutati positivamente le proprie pedine che sono protette, mentre viene attribuito un punteggio negativo a quelle che sono soggette all'attacco degli avversari;

- Collocazione del Re(k): questo criterio è il più ovvio ed indispensabile, esso infatti concerne la sicurezza del re. Viene attribuito un valore, positivo o negativo, alla posizione di questa pedina. Il punteggio sarà tanto più alto quanto meglio sarà difeso il re;

- Spazio e Mobilità(x): viene assegnato un punteggio in base alla libertà di movimento ed alla posizione occupata da ogni singola pedina, è infatti importante che ogni pezzo sia il più libero possibile;

- Controllo del centro(c): riveste un'importanza fondamentale per l'evoluzione di una partita, il controllo del centro della scacchiera, viene perciò attribuito un punteggio anche per tale attributo;

- Pedoni (p): i pedoni sono elementi sovente decisivi sia in difesa che nei finali di partita, per questo deve essere tenuta in considerazione anche la struttura pedonale;

- Propensione all'attacco (a): questa euristica valuta la collocazione delle pedine in funzione di piani di attacco, che puntano a raggiungere obiettivi quali per esempio la promozione di un pedone;

- Relazione tra i pezzi (r): può spesso rivelarsi importante la relazione esistente fra i pezzi durante alcune fasi della partita (es. la coppia di alfieri).

Nella versione sequenziale del programma Gnuchess questi criteri sono considerati tutti contemporaneament; bisogna però sottolineare che spesso il valutare una delle suddette caratteristiche può risultare più dannoso che vantaggioso, in quanto può disturbare il giudizio complessivo di una situazione di gioco.
A questo proposito vengono riportati di seguito (tabella 4.1) i risultati di uno studio condotto in [San93]su Gnuchess 4.00.
Tale statistica è stata estrapolata da un test condotto su 500 posizioni di gioco, nel quale la conoscenza del giocatore viene gradualmente incrementata fino ad arrivare ad una conoscenza completa, cioè tutte le 8 euristiche, e per ogni gruppo di criteri (istanza), si verifica quante volte viene selezionata la mossa migliore relativamente a quelle date posizioni.

	p	a	c	b	r	k	x
m	85	83	72	78	64	73	68
m	p	95	90	91	92	85	88
m	p	a	101	96	92	94	92
m	p	a	c	104	98	94	91
m	p	a	c	b	106	105	96
m	p	a	c	b	r	107	100
m	p	a	c	b	r	k	100

Table 4.1: Valutazione delle istanze

Come si può vedere non è certamente l'istanza a conoscenza completa a dare i risultati migliori, bensì un gruppo di istanze a conoscenza parziale che sembrano equivalersi a meno di una mossa.
Questo esperimento è sufficiente a giustificare il primo approccio alla parallelizzazione di Gnuchess 4.00.
Da questa prima analisi l'ordine di importanza delle varie euristiche risulta m p a c b r k x.

4.3 Prima fase di parallelismo

Questa prima fase di parallelizzazione era già stata introdotta in [San93], ma si rende ora necessario recuperarla poichè si riprenderà lo studio proprio da questa architettura di base.
Tale architettura è implementata secondo il modello Master-Worker descritto in precedenza. Ad ogni processo viene assegnata una istanza, cioè una porzione di conoscenza, sulla base della quale ognuno conduce la propria ricerca della mossa migliore a partire dalla situazione corrente.
Terminata questa prima fase di ricerca nel corso della quale ogni processo lavora in maniera assolutamente autonoma, il processo master si preoccupa di raccogliere tutte le mosse selezionate e di scegliere quella che dovrebbe essere la migliore.
Si pongono due problemi:

1. come conviene distribuire le euristiche, considerando che i risultati sono molto condizionati dai criteri scelti?

2. come viene scelta la mossa migliore tra quelle selezionate nella prima fase?

4.3.1 Scelta dei criteri di conoscenza

Si utilizzano in proposito, i risultati di un analisi già svolta in precedenza, in [San93], nella quale vengono presi in considerazione i seguenti sistemi di ripartizione delle euristiche:

distribuzione con nucleo comune : tutte le istanze contengono un sottoinsieme comune costituito dalle euristiche principali, alle quali vengono poi aggiunti alcuni dei rimanenti criteri distribuendoli in modo tale che alla fine ogni istanza risulti differente dalle altre;

distribuzione bilanciata : ogni istanza contiene lo stesso numero di euristiche, anche in questo caso ognuna di esse deve essere differente dalle altre pur contenendo tutte il criterio principale ed immancabile riguardante il materiale (m);

distribuzione sbilanciata : una sola istanza contiene tutta la conoscenza. Metà delle rimanenti è costituita dal criterio più importante (m) in

aggiunta ad un altro fra i principali mentre le altre contengono tutte le euristiche ad esclusione di una fra le meno importanti;

distribuzione incrementale : si inizia da una istanza che contenga alcuni dei criteri principali, poi ne viene aggiunto uno, in ordine di importanza, ad ogni successiva istanza fino ad avere la conoscenza completa nell'ultimo caso.

Si analizzi ora la tabella 4.2.

	ist	union	inters
Common kernel	4	140	56
	5	151	53
	6	158	41
	7	172	20
Balanced distribution	4	164	34
	5	176	26
	6	183	25
	7	186	24
Unbalanced distribution	4	159	43
	5	172	41
	6	185	31
	7	191	30
Incremental distribution	4	136	72
	5	141	65
	6	152	61
	7	161	53

Table 4.2: Valutazione delle distribuzioni

Viene indicato quante volte una determinata distribuzione è stata in grado di trovare in almeno un caso la mossa migliore su un test eseguito su 500 posizioni di mediogioco. Questo risultato viene indicato alla voce 'unione'.
Con intersezione si vuole invece indicare per quante volte tutte le istanze si sono trovate in accordo nell'indicare la mossa migliore da fare, nonostante vada detto che questa indicazione "certa" delle istanze non coincida nella

realtà con quella che è realmente la mossa migliore.
Bisogna comunque considerare con più attenzione il valore indicato dall'unione. Innanzitutto è necessario che fra tutte le potenziali mosse migliori, ci sia veramente la migliore, poichè in questo caso sarà poi compito della seconda fase selezionarla fra tutte quelle indicate.
Se le varie istanze non fossero in grado di selezionare, fra le altre, anche la mossa migliore, il principio utilizzato nella seconda parte di selezione potrebbe essere anche infallibile ma sarebbe assolutamente inutile.
Alla luce di queste considerazioni, le distribuzioni di conoscenza reputate più adatte si sono rivelate quella bilanciata e quella sbilanciata, per cui saranno queste ad essere utilizzate nel prosieguo di questo progetto.

4.3.2 Criterio di selezione

Possono essere utilizzati diversi criteri per selezionare la mossa piú adatta fra quelle proposte.

1. Maggioranza generalizzata con pesi costanti (Pesi). Ad ogni istanza viene associato un peso costante ottenuto dalla somma dei pesi delle sue euristiche. Viene perciò scelta la mossa proposta dall'istanza con maggiore peso.

2. maggioranza generalizzata con pesi variabili. Ad ogni istanza viene, come prima, associato un peso, ma questo può variare nel corso del match secondo le seguenti caratteristiche:

   - il peso dipende dalla profondità raggiunta (Criterio depth);
   - viene applicato un criterio democratico, cioè viene scelta la mossa che ha più sostenitori, in caso di parità si utilizza il criterio precedente (Criterio Democratico);

3. visita selettiva a posteriori dell'albero di gioco (Research). Viene eseguita un'ulteriore visita dell'albero di gioco, limitando però il primo livello alle sole mosse proposte nel corso della prima fase. Questa ulteriore analisi avviene sequenzialmente e utilizzando la conoscenza completa costituita dalle otto euristiche.

Criterium	best		worst	
	Bal	Unb	Bal	Unb
weight	88	100	13	10
depth	91	102	13	11
democratic	88	100	13	11
research	109	102	12	10

Table 4.3: Valutazione dei criteri

Si analizzi anche in questo caso la tabella 4.3.

Come nel caso precedente si è condotta la ricerca su un insieme di 500 posizioni ed il risultato che ne è derivato è che i criteri 'depth', 'pesi' e 'research' sono i migliori in abbinamento con l'utilizzo della distribuzione sbilanciata o bilanciata già descritte.
In conclusione, nella prima fase vengono scelte una serie di mosse potenzialmente migliori, secondo il sistema della distribuzione bilanciata o sbilanciata delle euristiche.
Nella seconda fase dalle mosse proposte, viene estrapolata quella che viene ritenuta essere la migliore secondo il criterio di selezione 'pesi', 'depth' o 'research'.
In particolare su quest'ultimo metodo verranno costruite delle nuove architetture.

4.4 Descrizione dell'architettura di base

Per la realizzazione di questo progetto si utilizza un'architettura distribuita che prevede che ad ogni istanza di conoscenza venga associato un processo, che parallelamente ai processi relativi alle altre istanze, concorre per definire il primo gruppo di mosse candidate.
Una volta terminata la ricerca, il processo Master si fa carico di raccogliere tutte le mosse e di selezionare la migliore.
Questo tipo di architettura si presta perfettamente ad essere implementata secondo il modello Master-Worker tramite Linda.

Si analizzi ora nel dettaglio l'intera struttura del programma.

4.4.1 Struttura del Master

Il master è il processo che si preoccupa di ricevere gli input dall'utente, di creare i worker e di distribuire loro i lavori da svolgere. In particolare il processo principale legge dalla linea di comando quante e quali sono le istanze che dovranno cooperare alla ricerca della mossa. Della prima di queste istanze si farà carico lo stesso processo master. Per le altre verranno creati tanti processi quante sono le istanze rimanenti e verranno distribuiti sulle macchine della rete locale.
Di seguito viene riportato il codice del processo Master nei suoi passaggi principali:

```
void real_main(argc, argv)
int argc;
char **argv;
{
      /* Ricezione da linea di comando delle istanze */
      ninstances = Init_fuct();
      nworkers = ninstances -1;

      /* Inizializzazione dei worker */
      Init_worker(nworkers);

      /* Ciclo principale */
     Main_loop();

     /* Terminazione dei worker */
    Terminate_workers();
}

void main_loop()
{
/* Ciclo principale */
```

```
    while (!quit)
    {
    /* Input comandi dell'utente (anche la mossa)*/
       Par_Input_Command();

    /* Selezione dela mossa da parte di ChessPar */
       Par_SelectMove();
    }
}
```

Da questi frammenti di codice risultano evidenti i compiti del processo Master.

Esso stabilisce quanti processi devono essere utilizzati (num_istanze -1), poi li crea mediante la chiamata ad 'InitWorker' e quindi entra nel ciclo principale che permette di giocare l'intera partita.
La funzione Init_workers è di particolare importanza in quanto, come si può vedere dal seguente codice, crea nworkers processi, ma ognuno con una caratteristica differente dagli altri. Ogni processo sarà caratterizzato e, quindi, anche univocamente identificato, dalla propria istanza che verrà utilizzata nel corso di tutta la partita per la scelta delle mosse nella prima fase di ricerca.

```
void Init_workers(nworkers)
int nworkers;
{
     ...
     /* Creazione dei processi */
     for(i=1; i<=nworkers; i++)
         eval("workers", func[i], workers(func[i]));

     /* func[i] contiene l'istanza di valutazione
          dell'i-esimo processo */
     ...
}
```

Il vettore func[i] contiene le istanze di ricerca dell' i-esimo giocatore. L'istanza

func[0] è sempre quella del processo Master che infatti non viene creato in Init_workers.
All'interno di 'main_loop' vi è la chiamata alla funzione 'Par_SelectMove' che costituisce il nucleo del Master. Essa consta di quattro funzioni principali.

Master_SelectMove: il compito di questa funzione è quello di attribuire ad ogni worker il lavoro da compiere e di far contribuire lo stesso master alla ricerca secondo il proprio criterio di valutazione.
Durante questa fase il Master diviene anch'esso un processo con le stesse competenze dei worker;

Get_results: gestisce la raccolta delle mosse selezionate nella prima parte sia dai worker che dal master stesso;

Det_bestmove: costituisce la seconda fase della ricerca, durante la quale il processo principale seleziona la mossa migliore fra quelle proposte.
In questo momento il solo processo attivo è il master, in quanto gli altri hanno per il momento terminato il loro lavoro ed attendono la comunicazione della mossa scelta;

Move_Communication: la mossa scelta dal Master deve essere comunicata ai worker che ora riprendono ad essere attivi, poichè devono aggiornare la propria copia locale degli estremi della partita.

```
void
Par_SelectMove(side, iop)
short int side, iop;
{
   struct leaf *moveOK;
         /* Prima fase di selezione della mossa  */
         Master_SelectMove(side,1);
         /* Raccolta dei risultati  degli altri worker */
         Get_results();
         /* Determinazione mossa migliore fra le selezionate*/
         moveOK = det_best_move(side);
```

```
        /* Comunicazione della mossa agli altri worker    */
        Move_Communication();

/* Aggiornamento copie locali delle variabili */
       ...
/*Aggiornamento tempi */
       ...
}
```

La funzione Master_SelectMove si dedica alla prima fase di ricerca. Essa comunica con i worker attraverso lo spazio delle tuple, indicando che il loro prossimo lavoro è 'SELECT_MOVE'. Dopo aver lanciato questa comunicazione, costringe il Master a dedicarsi allo stesso lavoro (Select_my_move). Il processo Master non preleva i suoi lavori dallo spazio delle tuple, come se dovesse preoccuparsi di svolgere quelle valutazioni che nessun worker ha svolto. Il suo lavoro è già definito a priori. Questo avviene poiché ogni processo è ben definito ed identificato dalla propria istanza, quindi un worker non è equivalente all'altro e allo stesso modo il master.
Prima di iniziare la ricerca, che avviene secondo l'algoritmo Alpha_beta, ([Pla96]) vengono settati gli estremi alfa e beta della finestra di ricerca. Qualora la ricerca non sortisse gli effetti sperati, cioè se non venisse trovata alcuna mossa plausibile all'interno di quel range, verrebbe ripetuta la ricerca ma su un intervallo più ampio.

```
void
Master_SelectMove(side, iop)
short int side;
short int iop;
{
  static short int i, tempb, tempc, tempsf, tempst, xside, rpt;
  static short int alpha, beta, score;

   /* Gestione  dei tempi */
   Get_Time();
```

```
 /* Viene calcolato lo score relativo all'attuale */
 /* disposizione delle pedine sulla scacchiera    */
 score = evaluate(...);

 /* Viene settata la finestra iniziale di ricerca */
alpha = score - ((computer == black) ? BAwindow : WAwindow);
beta = score + ((computer == black) ? BBwindow : WBwindow);

/* La root viene settata sul primo nodo dell'albero */
TrPnt[1] = 0;
root = &Tree[0];

/* Vengono aggiunte all'albero tutte le mosse possibili */
MoveList (side, 1);
/* poi ordinate */
 for (i = TrPnt[1]; i < TrPnt[2]; i++)
        if (!pick (i, TrPnt[2] - 1))

/* Prima di iniziare la ricerca, si verfifica che la mossa */
/* non esista gia' nel libro delle aperture                */
  if (flag.regularstart && Book)
  {
       flag.timeout = bookflag = OpeningBook (&hint, side);
        if (bookflag)
             root->score = score;
  }


  if (nworkers>=1)
 {
        if (bookflag)
        {
             flag.timeout = 1;
             ElapsedTime(0);
        }
        else
             send_job (side, SELECT_MOVE);
```

```
    }

    /* Il processo Master diviene un worker */
    /* ed inizia la propria ricerca         */
    Select_my_move();

}
```

La funzione 'Get_result' ha il compito di raccogliere i risultati di tutte le istanze, quindi sia dei "worker" che del "master". Essa tramite le chiamate alle funzioni 'in' di C_Linda, obbliga il processo che la sta eseguendo, cioè il master, a sospendersi in attesa dei risultati di tutti i workers.

```
in("result", ?instances[i], ?selected_moves[i], ?nodes[i], ?depths[i],
             ?ets[i], ?PrVars[i], ?DRAWs[i]);
```

Questa antitupla permette di raccogliere i risultati mano a mano che arrivano, senza un ordine prestabilito memorizzando nel vettore

?instances[i]

l'istanza che ha restituito quel risultato.
Si sarebbe ottenuto lo stesso effetto se il codice fosse stato

```
for(i=1;i<nworkers; i++)
        in("result", func[i], ?selected_moves[i], ?nodes[i],
                     ?depths[i], ?ets[i], ?PrVars[i], ?DRAWs[i]);
```

dove compare il parametro attuale 'func[i]' al posto del parametro formale '?instances[i]'. In questa maniera il master verrebbe obbligato ad attendere innanzitutto il risultato della prima istanza, poi quello della seconda e così via, con conseguenti lunghi tempi di attesa per tutti qualora uno dei processi tardasse a fornire i risultati.

Nella tupla "results" compaiono, oltre al vettore 'instances', anche altri parametri formali:

- selected_moves[i] contiene la mossa selezionata dall'i-esima istanza;
- nodes[i] indica quanti nodi sono stati visitati da quel processo;
- depths[i] contiene la profondità raggiunta nel corso della valutazione;
- ets[i] il tempo impiegato per la visita;
- PrVar[i] la variante principale individuata da quella istanza;
- DRAWs[i] indica se nel corso della ricerca sono state trovate mosse che possono condurre ad una patta.

```
void Get_Results()
{
     /* determina gli array relativi a tutte le */
     /* istanze del giocatore parallelo         */
     for (i=1; i<=nworkers; i++)
      in("result", ?instances[i], ?selected_moves[i],
       ?nodes[i], ?depths[i], ?ets[i], ?PrVars[i], ?DRAWs[i]);

     /* Raccolta del risultato del Master */
     instances[0]=e_flags;
     selected_moves[0]=*root;
     nodes[0]=NodeCnt;
     depths[0]=Sdepth;
     ets[0]=(et+50)/100;
     for (i=0; i<MAXDEPTH; i++)
            PrVars[0][i] = PrVar[i];
     DRAWs[0] = draw_type;
     NodeTot = NodeCnt;
}
```

La funzione Det_best_move riguarda solo il Master e costituisce da sola tutta la seconda fase della ricerca.
Essa varia a seconda di quale criterio si è deciso di utilizzare, ma di seguito ci si limiterà a valutare il criterio 'research'.
Come prima cosa vengono contate quante mosse diverse sono state selezionate e da quali istanze sono state proposte (questo serve se si utilizza il criterio 'pesi'), quindi avviene l'estrapolazione della mossa migliore.
Se il criterio utilizzato è il 'research', si ha che viene costruito un nuovo albero di gioco (v. ResearchList), che però al primo livello contiene solamente le mosse proposte dalle varie istanze. Come già detto questa nuova ricerca avviene con una conoscenza completa.

```
struct leaf *
det_best_move (side)
short int side; /* per la research */
{
struct leaf *mvtoret;

   /* determina gli array 'mosse_diverse' e 'prima istanza con
   quella mossa'; determina inoltre il limite degli array*/
      c = how_many_moves();
     if (c == 1)
        ind_max = 0;
     else
    {
         switch (criterium)
          {
            case SIMWEIGHT:
            case GENWEIGHT:
            case DEPTH:
/* determina il peso di ogni istanza; somma i pesi
delle istanze diverse che hanno scelto la stessa mossa;
sceglie la mossa col peso maggiore */
           for (i=0; i<c; i++)
```

```
          {
            num[i] = 0;
            for (j=0; j<ninstances; j++)
            if ((selected_moves[j].f<<8 |
selected_moves[j].t)==mvs[i])
                num[i] += (criterium==DEPTH? depth_weight(j) :
                      inst_weight(instances[j], criterium));
           }
           ind_max = 0;
           for (i=1; i<c; i++)
           {
              if (num[i]>num[ind_max] ||
            /* scelta casuale tra due mosse con lo stesso valore */
              (num[i] == num[ind_max] && urand() < 0x7FFF))
              ind_max = i;
           }
           break;
          case RESEARCH:
             re_search = true;
            /* La conoscenza diventa completa */
             e_flags = e_gnuchess;
            /* viene ricostruito l'albero */
            /* con le sole mosse selezionate */
             ResearchList();
             /* Nuova ricerca */
             SelectMove (side, 1);

             re_search = false;
             /* Viene ripristinata l'istanza iniziale */
             e_flags = func[0];
             break;
          default:
            /* impossibile */
            break;
        }
  }
```

```
      if (criterium == RESEARCH && c > 1)
             mvtoret = root;
      else
      {
            int indice;
            indice = first_inst[ind_max];
            NodeCnt = nodes[indice];
            Sdepth = depths[indice];
            ElapsedTime (0);
            for (i=0; i<MAXDEPTH; i++)
                        PrVar[i] = PrVars[indice][i];
             if (DRAWs[indice]!=0)
                        DRAW = CP[DRAWs[indice]];
            mvtoret = &selected_moves[indice];
        }
        return (mvtoret);
}
```

Dalla funzione det_best_move esce quindi la mossa migliore che però deve essere comunicata ai worker, affinché questi possano aggiornare le loro copie locali degli estremi della partita.
Di questo si occupa la funzione Move_Communication che segnala ai worker che il loro prossimo lavoro è 'MAKE_MOVE' e quindi comunica loro la mossa. Si noti che la tupla contenente la mossa non ha un destinatario preciso ma è un messaggio in broadcasting a tutti i processi.

```
void  Move_Communication()
{
       /* comunica la mossa ai workers */
       if (nworkers>=1)
      {
          send_job(side, MAKE_MOVE);
          send_move(side, moveOK);
       }
}
```

4.4.2 Scambio di messaggi fra Master e Worker

E' necessario soffermarsi brevemente sul sistema utilizzato dal Master per comunicare con i worker.
Nelle funzioni precedenti si è sempre utilizzato un intuitivo

send_job(side, msg)

dove 'msg' era di volta in volta uguale a 'SELECT_MOVE' oppure a 'MAKE_MOVE'.
Il codice di questa funzione e' il seguente:

```
void
send_job (side, job_type)
short int side, job_type;
{
short int i;

       for (i=1; i<=nworkers; i++)
            out ("job", func[i], side, job_type);
}
```

Per le comunicazioni dei lavori, il processo Master, pur utilizzando le chiamate di Linda e passando attraverso lo spazio delle tuple, realizza logicamente un sistema analogo allo scambio di messaggi.
Inserendo nella tupla anche il parametro attuale func[i], dota la tupla di un identificatore del destinatario, che potrà quindi ricevere solo la comunicazione riservata a se stesso.

4.4.3 Struttura del worker

Lo scheletro del worker è assolutamente analogo a quello già descritto nella parte iniziale a proposito del paradigma Master_Worker.
Il worker costituisce infatti un processo continuo che termina solamente con la fine dell'esecuzione, quando cioè gli viene passato il lavoro 'Quit' da svolgere.
Questo processo tiene traccia di quelle che sono le variabili principali della partita onde evitare ogni volta al Master di trasmettere tutti i dati necessari al worker al fine di svolgere l'analisi della sua parte di albero di gioco.

Poichè la lettura di valori dallo spazio delle tuple è comunque un'operazione molto costosa, conviene limitare la trasmissione dei dati sia nella frequenza che nella dimensione.
Il ciclo principale della vita del worker viene scandito da tre fasi principali.

creazione dei processi: innanzitutto il processo viene generato dal Master e viene contraddistinto da un valore caratteristico costituito dalla propria istanza di ricerca;

ricezione degli estremi del match: il worker deve recepire gli estremi necessari per condurre le proprie ricerche nel corso del match.
Tali parametri sono costituiti dal tempo di gioco e da alcuni flag indispensabili che il master non ha altro modo di comunicare ai processi che cooperano se non attraverso lo spazio delle tuple.
All'atto della creazione le sole variabili consistenti nell'ambiente del worker sono quelle che sono state esplicitamente passate mediante 'eval'.
Per quanto riguarda tutte le variabili rimanenti, seppur globali, il solo modo per settarle a dei valori consistenti è quello di riceverli attraverso lo spazio delle tuple.
Questo avviene attraverso la tupla

rd("parameters", ?rehash, ?TCflag, ?MaxResponseTime,
?tempxwndw, ?hash, ?ahead).

Terminata questa fase preliminare assolutamente necessaria, il worker può finalmente iniziare a collaborare attivamente.

ricezione del lavoro: la ricezione del lavoro avviene mediante l'operazione

in ("job", evalflags, ?side, ?job_type);

che contiene il parametro attuale 'evalflags' coerentemente con quanto esposto finora relativamente allo scambio di tuple fra processi.

Identificare ogni processo con la propria istanza può apparire una precauzione inutile. Si sarebbe portati a pensare che l'istanza di ricerca possa essere anch'essa passata ad un generico worker comunicandogli così quale criterio usare per questa sessione di ricerca; purtroppo non è così.
Si supponga infatti di non utilizzare alcun identificatore per il worker; sorgerebbero alcuni problemi di sincronizzazione.
Per esempio quando il master deve comunicare ai worker la mossa scelta per permettere loro di aggiornare le copie locali delle variabili. Se l'operazione di ricezione fosse semplicemente

in("job", ?side, ?job_type)

potrebbe accadere che un processo particolarmente rapido vada a ricevere due volte la stessa istruzione con conseguenze impredicibili per la propria consistenza dei dati, e lasciando inoltre un processo privo della tupla da rilevare.
Lo stesso problema si porrebbe se venisse aggiunto un qualsiasi altro tipo di sincronizzazione privo degli identificatori.
L'identificatore di un processo poteva essere costituito da un qualsiasi valore e non necessariamente dall'istanza di ricerca, ma essendo que-st'ultima assolutamente individuale rappresentava già un perfetto particolare di distinzione fra i processi. Ci si è soffermati molto su questo punto in quanto nelle prossime revisioni che descriveremo verranno modificate queste peculiarità dei processi.

```
int
worker (evalflags)
short int evalflags;
{
        /* Lettura degli estremi del match */
        rd("parameters", ?rehash, ?TCflag, ?MaxResponseTime,
               ?tempxwndw, ?hash, ?ahead);

       /* fase di sincronizzazione */
       synchr_receiver ();
       ...
      e_flags = evalflags;
```

```
 quit = false;
 /* Ciclo per la ricezione dei lavori */
 while (!quit)
{

     in ("job", evalflags, ?side, ?job_type);
     switch (job_type)
    {
          case SELECT_MOVE:
                 ElapsedTime(0);
                 Sdepth = 0;
                 select_my_move (side, 1, evalflags);
                 out ("result", evalflags, *root, NodeCnt,
                      Sdepth, time, PrVar, draw_type);

          case MAKE_MOVE:
                 rd ("mossa", ?node);
                 synchr_receiver ();

                if (node.f != node.t)
                       MakeMove (side, &node, &tempc, &tempb,
                             &tempf, &tempst, &INCscore);

              break;
           case NEW_GAME:
                synchr_receiver ();
                Local_NewGame ();
                break;
           case UNDO:
                synchr_receiver ();
                Undo ();
                break;
           case REMOVE:
                synchr_receiver ();
                Undo ();
                Undo ();
                break;
```

```
                case QUIT:
                     quit = true;
                     break;
                default:
                     break;
           }
     }

return (0);
}
```

Il parametro 'job_type' scandisce quale deve essere il successivo impegno del worker. Le mansioni cui può adempiere sono le seguenti:

1. SELECT_MOVE:
   il processo deve compiere un'analisi dell'albero di gioco con la propria istanza di ricerca allo scopo di selezionare la mossa ritenuta migliore dal proprio punto di vista. Per questa ricerca viene invocata la funzione select_my_move che è sequenziale.
   Terminata la selezione è necessario comunicare al processo master la soluzione trovata, viene quindi utilizzata l'operazione

   out ("result", evalflags, *root, NodeCnt, Sdepth,
   time, PrVar, draw_type);

   All'interno della tupla compare l'istanza di ricerca, necessaria anche qui affinché il master conosca da quale istanza è stata selezionata una mossa. Compaiono inoltre, 'root' che è il puntatore alla mossa ed altri parametri contenenti gli estremi della ricerca effettuata.

2. MAKE_MOVE:
   viene comunicato al processo di leggere dallo spazio delle tuple la mossa definitiva tramite l'operazione

   rd("mossa", ?node).

   Questa comunicazione è necessaria per permettere al processo di aggiornare le proprie copie locali delle variabili relative al match in corso.

3. NEW_GAME
   con questo messaggio viene comunicato al worker di reinizializzare le proprie variabili poichè deve iniziare una nuova partita;

4. UNDO
   indica che deve essere annullata la semimossa fatta in precedenza;

5. REMOVE
   comunica che deve essere rimossa l'intera mossa (2 semimosse) precedente;

6. QUIT
   con questo messaggio il master comunica ai worker che possono uscire dal ciclo principale e terminare.

Una volta ricevuto il job, segue una fase di sincronizzazione fra i processi, che è necessaria affinché il master possa essere certo che tutti i worker hanno recepito il lavoro.
Le procedure dedicate a questa fase sono due:

SynchrSender è la funzione che viene utilizzata dal Master e consta di due semplici operazioni che vengono effettuate in sequenza:

```
void
synchr_sender ()
{
     out ("synchr", (int)nworkers);
     in ("synchr", 0);
}
```

SuynchrReceiver è invece la funzione utilizzata da worker ed è anch'essa estremamente semplice:

```
void
synchr_receiver ()
{
    int x;

    in ("synchr", ?x);
    out ("synchr", x-1);
}
```

Il funzionamento è piuttosto intuitivo. Viene innanzitutto depositata da parte del processo Master, la tupla ("synchr", nworkers) nello spazio di memoria condiviso, poi questo processo si sospende in attesa che il secondo campo della tupla diventi zero.
Questo avviene solamente quando tutti i worker hanno letto questa tupla, essi infatti la rilevano dalla memoria, decrementano di una unità il secondo parametro, e quindi ve la ridepositano.
Con questo tipo di costruzione si è certi che i processi agiscono sulla tupla in mutua esclusione. Infatti la tupla non viene letta e poi modificata, bensì rilevata, in modo che nessun altro processo possa accedervi, poi modificata e solamente ora rideposta.
In questo modo si è sicuri della consistenza dei dati.

4.4.4 Valutazioni

L'aspetto negativo di questo tipo di architettura è evidente. I processi worker contribuiscono attivamente solo alla prima fase della ricerca, mentre assistono solamente allo svolgimento della seconda parte di cui si fa carico il Master da solo. Se si considera che, nel criterio Research, la distribuzione dei tempi risultata ottimale ([San93]) è stata di 1/3 per la prima fase e 2/3 per la successiva, si deduce subito che su un tempo complessivo di tre minuti i processi worker rimangono fermi per addirittura 120 secondi in attesa della mossa (v. fig.4.1).

Si analizzi infatti la figura relativa al ciclo di utilizzo di ogni processo. Quando il worker ha terminato di analizzare l'albero delle mosse con la sua istanza di conoscenza, non può fare altro che sospendersi in attesa che il Master abbia raccolto tutte le mosse suggerite ed analizzato l'albero ridotto

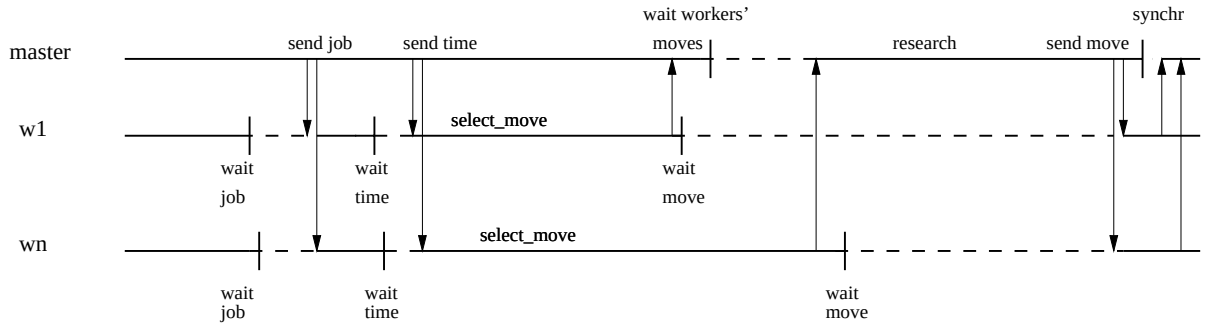


Figure 4.1: Ciclo dei processi versione base

Proc.	Nodes	Time	Vel	Svr	Fpm
1	25892763	6201	4175.579		
3	26937066	3739	7204.35	1.725354	0.675118
7	53366995	4177	12776.39	3.05979	0.537113
11	76680858	4195	18279.11	4.377623	0.497966
15	95548664	4406	21686.03	5.19354	0.446236
19	115557641	4592	25164.99	6.026708	0.417195

Table 4.4: Risultati con architettura di base e $\frac{2}{3}$ del tempo per la seconda fase

così ottenuto. Inutile rimarcare quale enorme spreco implichi questo tipo di struttura.

Proprio a causa di queste sospensioni si ottengono i risultati visibili nella tabella. Queste valutazioni sono state ottenute analizzando un insieme di 36 posizioni, usando prima 1 macchina, poi 3, quindi 7, 11, 15 e 19, utilizzando ogni volta altrettante istanze di conoscenza.

Nella prima tabella (v. tabella 4.4) vengono riportati i dati relativi alle prestazioni ottenute riservando 1/3 del tempo complessivo (180 sec.), alla prima fase di ricerca ed i restanti 2/3 alla "research". Il risultato negativo che appare più evidente è il fattore di produzione medio (Fpm) che indica per quanto tempo è stato mediamente attivo ogni processo. Si passa da uno 0,41 con 19 elaboratori ad uno 0,67 con solamente tre. Questo ovviamente è

Proc.	Nodes	Time	Vel	Svr	Fpm
1	25892763	6201	4175.579		
3	34234578	4527	7562.31	1.811081	0.703694
7	72247087	4545	15895.95	3.806885	0.643841
11	101185837	4521	22381.3	5.360046	0.587277
15	138421515	4727	29283.16	7.01296	0.567531
19	177882066	5281	33683.41	8.066764	0.524567

Table 4.5: Risultati con architettura di base e metà tempo per ogni fase

dovuto al fatto che il tempo complessivo di inutilizzo è inferiore con tre sole macchine piuttosto che con 19.

La situazione non migliora molto se si riserva la stessa porzione di tempo (1/2) alla prima fase ed alla successiva. In questo caso infatti (v. tabella 4.5) avviene che il Fpm passa da un minimo di 0,52 con 19 elaboratori ad un massimo di 0,70 con tre. Anche lo Speed-up migliora con questa nuova distribuzione dei tempi e questi lievi miglioramenti sono ovviamente dovuti al fatto che nella prima fase di ricerca parallela, i worker sono tutti attivi, quindi protraendosi più a lungo questa situazione si ha un guadagno sulle prestazioni.

Gli altri fattori che compaiono nella tabella sono la velocità (Vel) di analisi dei nodi (nodi/sec) e lo Speed-up reale (Svr) ottenuto con il rapporto fra le velocità. Non è stato inserito lo Speed-up calcolabile facendo il rapporto fra il tempo impiegato dalla versione sequenziale e quello della versione parallela, in quanto pareva poco significativo considerando che le due versioni eseguono delle analisi di tipo diverso. La versione sequenziale di Gnuchess (cui in tabella si fa riferimento con l'indicazione macchine = 1), non si preoccupa infatti di condurre prima una ricerca con istanze diverse e poi di fare una nuova analisi per trovare la mossa migliore.

Concludendo, si può dire che questo approccio alla parallelizzazione del programma Gnuchess, per quanto possa essere efficace (v. Cap. Valutazione dei risultati), può essere decisamente migliorato.

Chapter 5

Architettura di ChessPar con distribuzione della ricerca

5.1 Introduzione

Nell'architettura di base la sola forma di parallelismo è costituita dalla distribuzione della conoscenza fra i processi. Le revisioni che verranno esposte in seguito hanno lo scopo di migliorare l'architettura di ChessPar accostando alla distribuzione della conoscenza anche la distribuzione della ricerca. La fase della "Research" si presta perfettamente ad essere migliorata ripartendo la ricerca su più elaboratori. Allo stesso modo anche la prima fase può essere sviluppata distribuendo la valutazione dell'albero relativo ad ogni istanza. Si mantiene comunque la suddivisione della ricerca in due fasi e l'architettura generale costruita sul modello Master-Worker.

5.2 Decomposizione statica della "Research"

Una prima evoluzione è certamente quella di rendere alcuni processi operativi anche nella seconda fase. In precedenza il processo Master raccoglieva le mosse e quindi, da solo, rianalizzava l'albero ridotto ottenuto dopo la prima fase di "filtro".
L'architettura descritta di seguito distribuisce su più processi la seconda parte dell'analisi. Viene decomposto l'albero ridotto e viene coinvolto un processo per ogni sotto-albero della radice.

Questo tipo di decomposizione è statica in quanto sono identificati a priori i nodi sui quali viene distribuito il lavoro. In questo caso il nodo in questione è la radice dell'albero, poiché da questo punto le ricerche procedono parallelamente (vedi fig 5.1).

La ricerca sui sottoalberi è sequenziale ed ogni processo procede indipen-

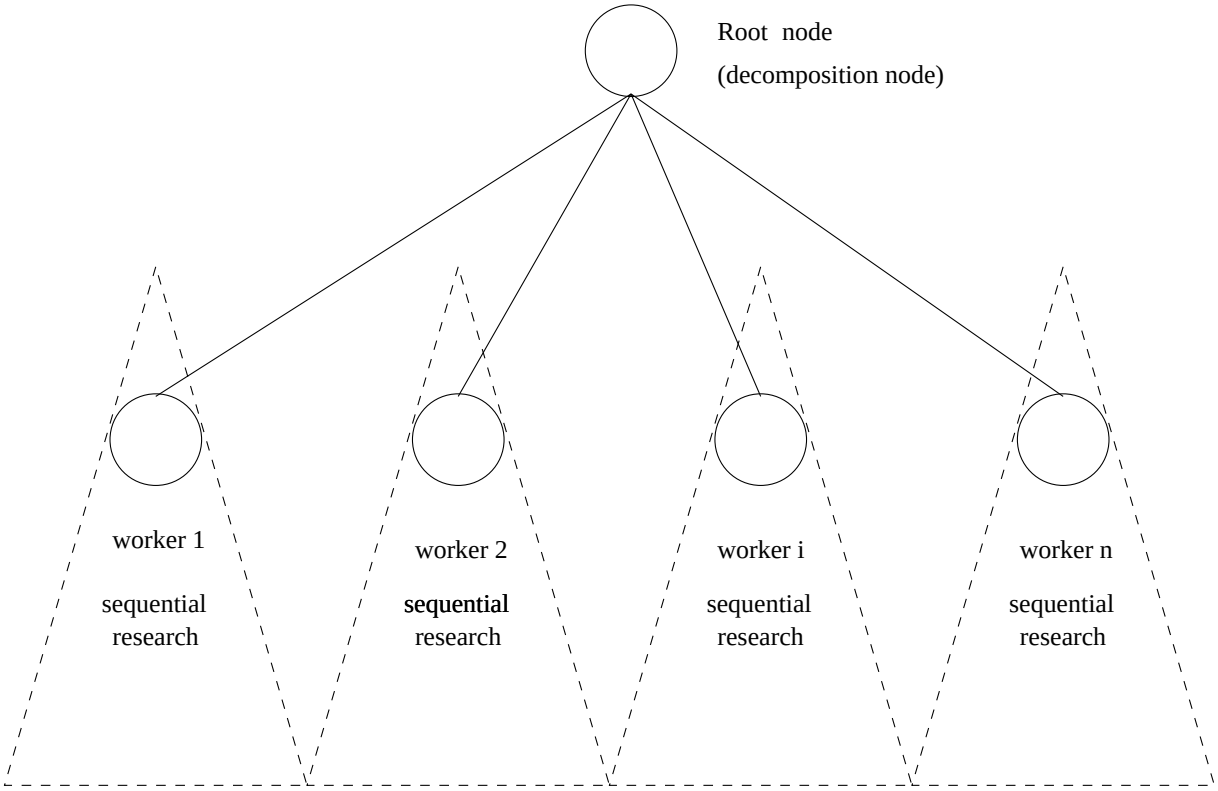


Figure 5.1: Gestione dell'albero con decomposizione statica

dentemente dagli altri.
La struttura logica della comunicazione è mostrata in figura 5.2.
Il Master deposita nello spazio delle tuple un insieme di lavori (agenda) ed ognuno di questi viene ricevuto da un worker che lo calcola. Al termine della computazione, mediante lo spazio delle tuple, il risultato viene restituito al master che conclude scegliendo la mossa con score maggiore.

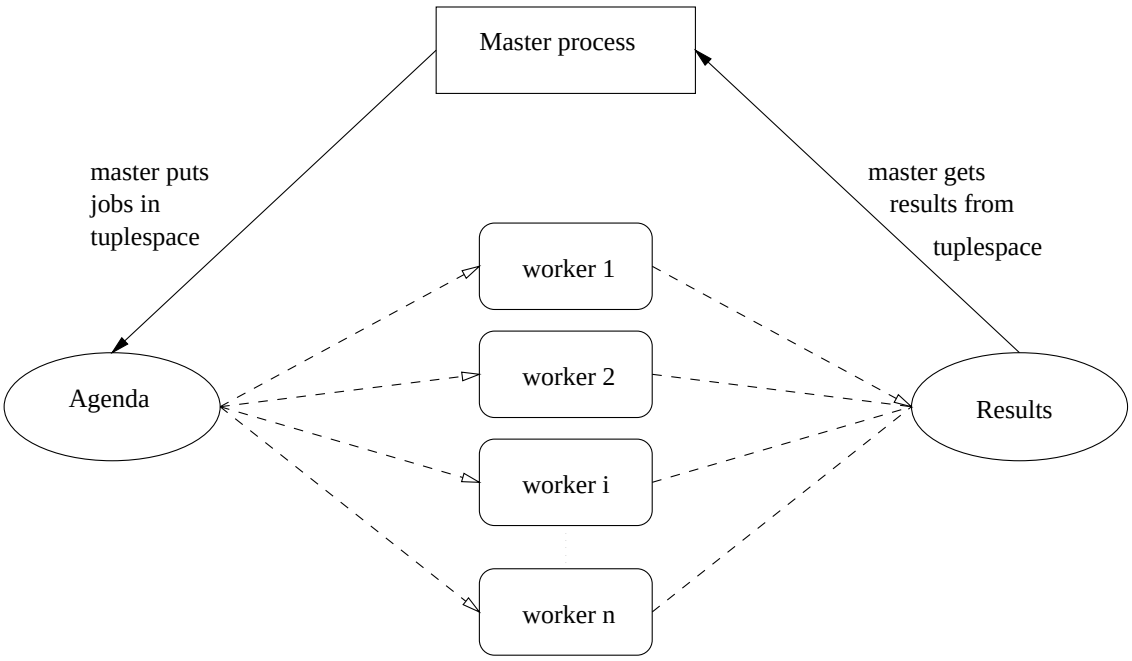


Figure 5.2: Struttura logica della comunicazione

5.2.1 Struttura del Master

La struttura del Master è analoga a quella del capitolo precedente, infatti la prima fase rimane invariata, mentre invece cambia la funzione det_best_move che gestisce la research.
Questa funzione costituisce il nucleo della seconda frazione di analisi. Lo scheletro è costituito da uno "switch(criterium)" che permette al master di condurre un tipo di ricerca o l'altro a seconda del tipo di criterio che si è deciso di utilizzare. Alle alternative già viste (weight, depth, research) ne deve essere ora aggiunta un'altra (resrcpar), che permetta di rendere parallela la ricerca. La funzione viene ampliata con questa nuova porzione di codice:

```
struct leaf * Det_best_move(side)
     ....
     case RESRCPAR:
        re_search = true;
        ElapsedTime (0);       /* Aggiornamento tempi */
        root = faseII(side, nmove);/* La mossa migliore
                                   diventera' la nuova radice*/
         re_search = false;
         break;
   .....
```

All'interno compare la funzione "faseII(side, nmove)" dove 'side' e 'nmo-ve' sono rispettivamente il giocatore a cui tocca muovere e il numero di mosse rimaste.
Questa funzione distribuisce il lavoro ai worker che devono essere coinvolti e ne raccoglie i risultati.

```
struct leaf* faseII(side, nmove)
SHORT side;
int nmove;
{
     ....
     /* COMUNICAZIONE LAVORO */
    if (nmove != 1)
```

```
{
 /* Invia ai worker il lavoro */
     send_job(side, SEL_MOVEII, nmove);
 /* Sincronizzazione */
     synchr_sender(nmove);

 /* Costruzione dell'albero con le mosse */
 /* raccolte nella prima fase                    */
    ResearchMoveList(side, 1);

   /* Se le mosse rimaste sono tante quante le macchine
   allora il master collabora anch'esso alla ricerca,
   altrimenti funge solo da coordinatore */
   usato = 0;
   if (nmove  == ninstances)
         usato = 1;

   /* COMUNICAZIONE ALBERO */
   /* Viene depositato nello spazio delle tuple
    il lavoro per gli 'nmove' worker */
   for(i=usato; i<nmove; i++)
        out("TREE", Tree[i], tempo);

  /* Fase di ricerca*/
    if(usato)
    {
            root = &Tree[0];
             ....ricerca...
    }

 /* RACCOLTA RISULTATI DELLA SECONDA FASE */
  for(i=usato; i<nmove; i++)
        in("resp_2", ?mosse2fase[i], ?nodes[i], ?depths[i],
                     ?ets[i], ?PrVars[i], ?DRAWs[i]);

 /* Raccolta dei risultati del master */
if (usato)
```

```
                ...raccolta risultato master........

    }

  /* Elaborazione finale per la scelta della mossa migliore */
    nodo = finale(nmove);
    score = nodo->score;
  }
  root = nodo;
  return(nodo);
}
```

Questa funzione consta di tre momenti principali:

comunicazione del lavoro: vengono depositati nello spazio delle tuple 'nmove' elementi mediante l'istruzione

send_job(side, SEL_MOVEII).

Prelevando questa tupla altrettanti worker si riattivano in attesa di ricevere il sottoalbero da analizzare.
Per permettere ai processi di accedere attivamente alla seconda fase di ricerca è stato introdotto il nuovo messaggio 'SEL_MOVEII' che va ad aggiungersi a quelli già considerati in precedenza (SELECT_MOVE, MAKE_MOVE...);

comunicazione del sottoalbero: ora che i worker possono collaborare bisogna comunicare loro quale sarà la mossa che dovranno prendere in considerazione. Questo avviene attraverso il passaggio della tupla

out("TREE", Tree[i], tempo);

che indica quale deve essere la radice del sottoalbero da valutare (Tree[i]) ed il tempo a disposizione.
In questa seconda parte ogni worker è dotato di conoscenza completa con tutte le istanze di Gnuchess originale;

raccolta risultati: il prelevamento dei risultati dallo spazio delle tuple, avviene mediante un ciclo di 'nmove' volte (corrisponde al numero di processi coinvolti) l'istruzione

```
in("resp_2",?mossa[i], ?nodes[i], ?depths[i],
            ?ets[i], ?PrVars[i], ?DRAWs[i]).
```

Non vi è un ordine predefinito per la ricezione dei risultati in quanto non ha assolutamente importanza.
Il master si limita a ricevere le tuple in ordine casuale. Ogni worker oltre alla mossa considerata, che è la stessa ricevuta in input, ne restituisce la valutazione, i nodi analizzati, la profondità di ricerca, i tempi, la variante principale ed infine comunica se vi sono mosse che possono condurre ad una patta. Diversamente dalla tupla utilizzata per la raccolta delle mosse nella prima fase, qui non compare il campo relativo all'istanza poiché tutti i worker coinvolti hanno la stessa istanza di ricerca.

La funzione 'faseII' comprende una fase di ricerca che viene però eseguita solamente in caso di necessità, qualora, cioè, il numero di mosse rimaste da analizzare, fosse rimasto uguale a quello dei processi.
In questo caso il master distribuirebbe un numero di lavori pari al "numero delle mosse -1" facendosi poi carico di analizzare quella rimasta.
Il processo master conclude il suo iter scegliendo fra le mosse candidate quella con punteggio migliore e la comunica prima all'avversario e quindi a tutti i worker.

5.2.2 Struttura del worker

Analogamente al master, anche i processi worker devono essere ampliati per permettere la collaborazione con esso.
La funzione principale "worker(..)" deve essere modificata al fine di gestire anche la ricezione del messaggio "SEL_MOVEII" per poi accedere alla nuova

fase di ricerca coordinata. Al codice precedente viene perciò aggiunto:

```
case SEL_MOVEII:
     synchr_receiver();  /* sincronizzazione */
     ElapsedTime(0);     /* inizializza tempo */
    /* Ricerca con istanza completa */
     coord_search(side, 1, e_gnuchess);
     e_flags = evalflags; /* ripristina l'istanza iniziale */
     break;
```

La funzione "coord_search(...)" consente di effettuare la ricerca sequenziale sul sottoalbero assegnato al worker. Fra gli argomenti passati vi è anche "e_gnuchess" che indica con quale istanza deve essere effettuata la valutazione. In questo caso si tratta di utilizzare tutti i criteri di conoscenza di gnuchess.
Terminata la valutazione il worker riprende la sua istanza di ricerca iniziale. La funzione "coord_search" è analoga a "select_my_move" della prima fase, con l'aggiunta della fase di ricezione dell'albero e della restituzione della tupla risultato leggermente diversa dalla precedente.

```
void coord_search (side, iop, ef)
SHORT side, iop, ef;
{

               e_flags = ef;
               flag.timeout = false;
          /* RICEZIONE SOTTOALBERO */
               in("TREE", ?ptr_move, ?tempo);
               ResponseTime = tempo;

          /* Preparazione copia locale albero */
               TrPnt[0] =TrPnt[1]= 0;
               TrPnt[2] = 1;
               Tree[0] = ptr_move;
```

Proc.	Nodes	Time	Vel	Svr	Fpm	OS
1	25892763	6201	4175.579			
3	28771243	3452	8334.659	1.996049	0.76535	0.068091194
7	56693675	3456	16404.42	3.928658	0.661237	0.062335906
11	78787315	3201	24613.34	5.894595	0.635872	0.027470441
15	97122528	3452	28135.15	6.738024	0.549202	0.016471858
19	117879210	3454	34128.32	8.173315	0.530174	0.020090138

Table 5.1: Risultati con decomposizione statica e $\frac{2}{3}$ del tempo per la seconda fase

```
            root = &Tree[0];

        /* RICERCA SEQUENZIALE */
            select_move(..);

        /* OUTPUT  DEL RISULTATO*/
            out ("resp_2", *root, NodeCnt, Sdepth,
                       (et+50)/100, PrVar, draw_type);
}
```

All'inizio della "Research" ogni worker è assolutamente equivalente agli altri in quanto questa seconda fase è indipendente dalla prima. Non ha importanza quale worker abbia proposto una certa mossa, la valutazione del sottoalbero che ha questa mossa come radice può essere condotta da qualsiasi altro processo.

5.2.3 Valutazioni

Da questo differente approccio alla "Research" si ottengono i previsti miglioramenti sul Fpm, in quanto non tutti i worker rimangono inattivi nella seconda fase.
Sono state valutate due differenti ripartizioni del tempo per le ricerche: in un caso (vedi tab. 5.1) sono stati dedicati 60 secondi alla prima fase ed i

Proc.	Nodes	Time	Vel	Svr	Fpm	OS
1	25892763	6201	4175.579			
3	34553863	4166	8294.254	1.986372	0.762124	0.009326389
7	74222367	4077	18205.14	4.359909	0.722844	0.027340618
11	103745840	4166	24902.98	5.96325	0.64113	0.0253001
15	141657312	4255	33291.97	7.973018	0.631535	0.023376402
19	180915275	4450	40655.12	9.736403	0.612442	0.017051798

Table 5.2: Risultati con decomposizione statica e metà tempo per ogni fase

restanti 120 secondi alla successiva; nell'altro (vedi tab. 5.2) si è riservato la metà del tempo complessivo (180 secondi) per ogni fase.

Dalla tabella 5.2 risulta che il Fpm varia da 0,76 con tre elaboratori a 0,61 con 19. Una giustificazione si trova nell'aumento dell'overhead di sincronizzazione, infatti coordinare 19 macchine richiede tempi superiori a quelli necessari nel caso di solo pochi elaboratori.
Rispetto alla situazione riscontrata nella tabella 4.5 del precedente capitolo, si ha che l'aumento del Fpm si mantiene quasi costante indipendentemente dal numero dei processi coinvolti. Questo avviene poiché all'aumento degli elaboratori usati corrisponde un incremento delle istanze di ricerca. Evidentemente a più istanze corrisponde un maggior numero di mosse proposte diverse e quindi da sottoporre a nuova valutazione.
Per la ripartizione dei tempi a metà per le due fasi si ha che il miglioramento del Fpm si aggira sull' 8%, mentre nell'altro caso (tab. 5.1) il miglioramento è leggermente superiore e si assesta sul 12/13%. Questo si deve al fatto che nell'algoritmo di base (vedi fig.4.1) il periodo di inattività di un worker era superiore (due terzi del tempo complessivo).

In queste valutazioni viene introdotto anche l'overhead di ricerca (OS) calcolandolo rispetto alla versione di base di questa stessa architettura distribuita. Appariva poco significativo confrontare questi risultati con la versione sequenziale di Gnuchess, in quanto la sua architettura risulta troppo diversa da quella del nostro progetto.
Il confronto viene fatto con l'architettura di base, tabelle 4.5 e 4.4, in corrispondenza della stessa gestione dei tempi e dello stesso numero di processi.

L'overhead di ricerca ha un andamento anomalo, decresce all'aumentare degli elaboratori. Una giustificazione si ha nel fatto che, pur aumentando il numero di nodi visitati, l'incremento assoluto è sempre meno significativo in relazione alla quantità di nodi totali. Questo valore cresce proporzionalmente ai processi utilizzati, l'OS segue infatti l'andamento inverso.

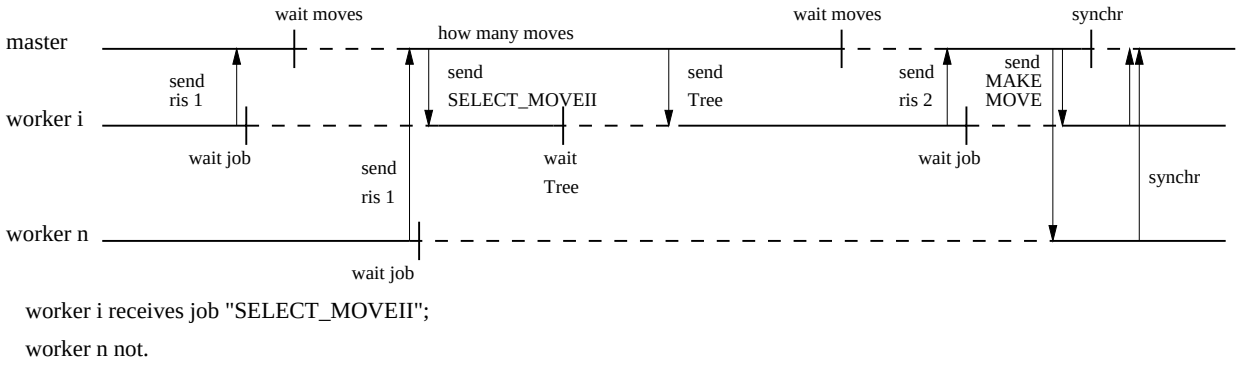


Figure 5.3: Ciclo dei processi con dec. statica della research

In figura 5.3 viene mostrato il ciclo di un generico processo worker e del master. La prima parte, che infatti non viene riportata, è assolutamente identica alla precedente, le variazioni avvengono successivamente alla ricezione del "SELECT_MOVEII".
I processi che recepiscono questo lavoro proseguono analizzando ulteriormente l'albero, ma tutti gli altri worker rimangono assolutamente inattivi per tutto il tempo con un conseguente spreco delle potenzialità del sistema.
Bisogna inoltre sottolineare che la suddivisione dell'analisi dell'albero comporta dei problemi con l'algoritmo di ricerca alfabeta. Questo algoritmo infatti (vedi Cap. 2) ha la sua caratteristica principale nell'adottare dei "tagli" che permettono al processo di non analizzare dei sottoalberi che non porterebbero ad alcun miglioramento.
Facendo invece procedere i processi di analisi dei nodi in modo indipendente, questi tagli non possono essere utilizzati. Infatti un worker non è in grado di sapere quale sia il valore migliore della finestra di ricerca selezionato fino a quel momento.
Senza questa condivisione all'aumento dell'utilizzo dei processori non corrisponde un reale miglioramento delle prestazioni.
Di queste evoluzioni si tratterà nel prossimo paragrafo.

5.3 Cooperazione alla pari

In questa versione della nostra architettura vengono applicate delle modifiche ad entrambe le fasi di ricerca. Sulla prima infatti non verranno più utilizzate tante istanze quante sono le macchine disponibili, ma riferendoci a [San93]ne consideriamo solamente sette costruite secondo il criterio della distribuzione bilanciata (vedi fig. 5.4).

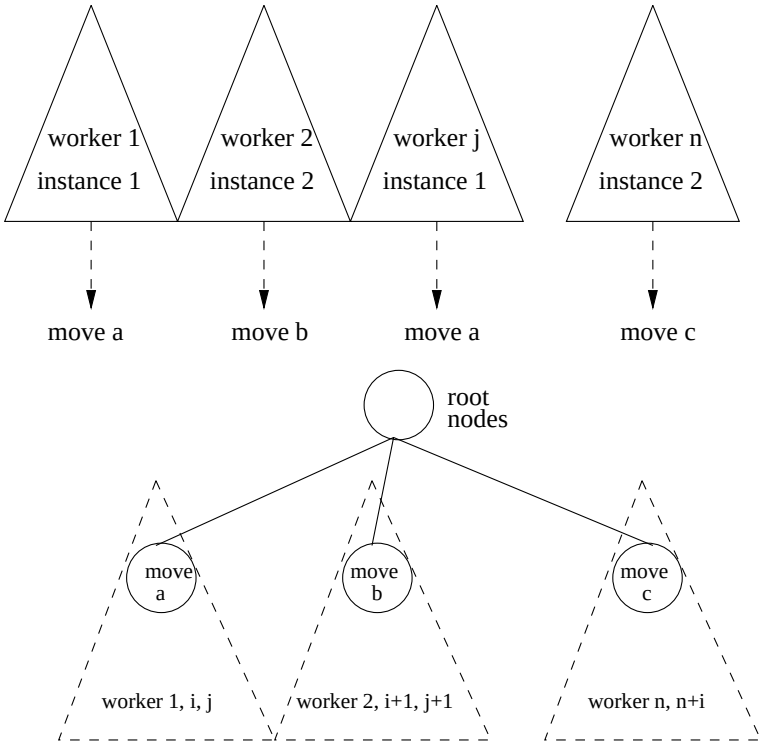


Figure 5.4: Gestione dell'albero nella cooperazione alla pari

Vengono costituiti dei gruppi di processi ([FGY95]) ed ogni istanza viene presa in considerazione da uno di questi. Ogni gruppo è costituito da un supervisore e da altri processi coordinati. Ogni processo conduce la sua ricerca indipendentemente dagli altri. Al termine dell'analisi, il supervisore sceglie

per il proprio gruppo la mossa selezionata dal processo che ha visitato l'albero più in profondità.
Allo stesso modo durante la "research", nessun processo rimane inattivo, ma ognuno analizza un sottoalbero. Lo stesso sottoalbero può essere analizzato da più worker contemporaneamente. Anche in questo caso ad ogni mossa candidata viene attribuito il punteggio proposto dal processo che ha sondato più in profondità quel sottoalbero.
Sono attesi dei miglioramenti sul Fpm, su Svr ma un peggioramento notevole dell'overhead di ricerca.
Quando i processi coinvolti sono in quantità inferiore a sette (numero ottimale di criteri conoscenza [San93]), ogni istanza diversa corrisponde univocamente ad un worker. Quando invece il loro numero è superiore, i criteri di conoscenza vengono distribuiti secondo la formula:

$$knowledge_{worker-i} = instance\ [i\ mod\ 7]\ .$$

Ad ogni processo viene attribuito un "identificatore" in quanto un worker non può più essere univocamente determinato dalla propria istanza di ricerca come invece è avvenuto fino ad ora. Il Master all'atto della creazione di un worker lo caratterizza con un valore intero ed attribuisce a se stesso il valore zero come identificatore.
I processi che hanno un identificatore $id_{process} < ninstances$, diventano i supervisori per i worker che hanno la loro stessa istanza di conoscenza. Il loro compito è quello di selezionare la mossa migliore per il loro gruppo fra quelle proposte dai worker "simili". (Vedi fig. 5.4).
Vi sono, in sostanza, due momenti di raccolta dei risultati. Il primo è gestito dai supervisori che raccolgono le mosse generate dai processi del loro gruppo e depositano nello spazio delle tuple la mossa giudicata migliore. Segue la raccolta da parte del Master che riceve questi ultimi risultati e costruisce l'albero ridotto.
Nella fase di "Research" i supervisori tornano invece ad essere dei normali worker coordinati dal processo master. Questo avviene poiché non è possibile sapere a priori quale sottoalbero verrà analizzato da un worker, quindi è impossibile stabilire dei gruppi di processi in base alla mossa che stanno considerando.

5.3.1 Struttura del Master

Il Master deve essere opportunamente modificato.
I momenti principali rimangono gli stessi, ma devono subire alcuni aggiornamenti.
In questa fase i worker che vengono creati devono essere identificati da un valore che viene attribuito loro dal master stesso. Tale identificatore è un numero intero che non ha alcun legame con l'istanza di conoscenza.
Al momento della generazione di un processo, il master deve comunque caratterizzarlo mediante un identificatore ed una istanza di conoscenza che, però, non è più specifica di un worker solamente.
La funzione "Init_worker" subisce alcuni cambiamenti.

```
void Init_workers ()
{
short int i;
     ...
     for (i = 1; i < num_macch; i++)
          eval ("worker", worker (i, func[(i%ninstances)]));
     ...
}
```

Il ciclo non viene più eseguito "nworkers" volte, ma "num_macch" volte dove questa variabile indica la quantità di processi che collaborano.
La differenza principale è nell'invocazione dell' "eval" in cui al worker vengono passati sia la conoscenza (func[(i%ninstances)]) sia l'identificatore (i).
L'identificatore del master è sempre zero.
Le fasi di ricerca rimangono invariate, mentre cambia la raccolta dei risultati.
La funzione principale "Par_SelectMove", viene ampliata con l'inserimento della nuove procedura "Select_group_move". Essa gestisce la raccolta dei risultati dei worker che hanno la stessa conoscenza del supervisore.
In questo caso il supervisore è il master stesso.

```
void
```

```
Par_SelectMove(side, iop)
short int side, iop;
{
   struct leaf *moveOK;
      /* Prima fase di selezione della mossa  */
      Master_SelectMove(side,1);

       /* Seleziona la mossa migliore per il proprio gruppo */
       if (identifier < ninstances)
            Select_group_move();

      /* Raccolta dei risultati  degli altri worker */
      Get_results();
      /* Determinazione mossa migliore fra quelle selezionate*/
      moveOK = det_best_move(side);
      /* Comunicazione della mossa agli altri worker    */
      Move_Communication();

/* Aggiornamento copie locali delle variabili */
       ...
/*Aggiornamento tempi */
       ...
}
```

In particolare i processi che hanno la stessa conoscenza del Master, sono quelli il cui identificatore è

$$id_{worker} = id_{master} + n * ninstances.$$

La conoscenza del Master è quindi la più frequente.
"Select_group_move" viene eseguita solamente dai processi supervisori, quelli cioè il cui identificatore è $id_{process} < ninstances$.
La funzione "Select_group_move" è costituita da due momenti principali. Nel primo avviene la ricezione dei risultati dei processi "equivalenti"; nella seconda viene depositata nello spazio delle tuple la mossa prescelta per il proprio gruppo.

```
void select_group_move()
{
    int i, j, maxnode;
    int bst_ind = 0;

   /* Raccolta dei propri risultati */
    instances[0]=e_flags;
    selected_moves[0]=*root;
    maxnode = nodes[0]=NodeCnt;
    depths[0]=Sdepth;
    ets[0]=(et+50)/100;
    for (i=0; i<MAXDEPTH; i++)
        PrVars[0][i] = PrVar[i];
    DRAWs[0] = draw_type;

    j = identifier;
    i = 1;

/* Raccolta risultati degli altri worker */
/* del mio stesso gruppo                    */
    while(j<num_macch)
    {
       in("RIS", e_flags, ?selected_moves[i], ?nodes[i],
                    ?depths[i], ?ets[i], ?PrVars[i]:, ?DRAWs[i]);
       if(nodes[i] > maxnode) bst_ind = i;
           NodeCnt = NodeCnt + nodes[i];
       j = j + ninstances;
       i++;
    }

/* Restituisce il risultato del gruppo */
    if(identifier != 0)
         out("RIS_GROUP", e_flags, selected_moves[bst_ind],
          NodeCnt, depths[bst_ind], ets[bst_ind],
            PrVars[bst_ind]:, ?DRAWs[bst_ind]);
    else
   {
```

```
        instances[0]=e_flags;
        selected_moves[0]=selected_moves[bst_ind];
        nodes[0]=NodeCnt;
        depths[0]=depths[bst_ind];
        ets[0]=ets[bst_ind];
        PrVars[0] = PrVars[bst_ind];
        DRAWs[0] = DRAWs[bst_ind];
    }
}
```

La fase di raccolta dei risultati del gruppo avviene mediante l'istruzione

```
in("RIS", e_flags,?selected_moves[i], ?nodes[i], ?depths[i],
                    ?ets[i], ?PrVars[i]:, ?DRAWs[i]);
```

in cui compare il parametro attuale "e_flags". E' questo che permette al supervisore di ricevere solamente le tuple risultato depositate dai processi appartenenti alla stessa categoria.
Fra tutte le mosse viene quindi scelta quella il cui worker ha condotto la ricerca più in profondità. Il supervisore mette poi il risultato nello spazio delle tuple :

```
out("RIS_GROUP", e_flags, selected_moves[bst_ind], NodeCnt,
                        depths[bst_ind], ets[bst_ind], PrVars[bst_ind]:,
                                            DRAWs[bst_ind]).
```

Se il supervisore in questione è il Master allora l'output non viene dato fuori in quanto è questo stesso processo che deve ricevere le tuple degli altri gruppi (vedi funzione Get_result).
Terminata la prima fase inizia la "Research" (vedi fig. 5.5).
In questa fase il Master ricostruisce l'albero di gioco riducendolo alle mosse proposte dai gruppi di istanze, poi dà ordine ai worker di proseguire con la ricerca su uno dei sottoalberi. Ogni sottoalbero viene preso in considerazione dallo stesso numero di processi. Non vi è, però, alcun legame fra l'identificatore e la mossa che il processo sta valutando.

Innanzitutto il master comunica a tutti i worker che il loro prossimo "job" è "SEL_MOVEII", poi deposita nello spazio delle tuple i sottoalberi da valutare:

for(i=1; i <num_macch; i++)
        out("albero", Tree[i%nmove], side, tempo, score);

Da qui l'impossibilità di sapere a priori quali processi si stanno occupando di un determinato sottoalbero.
E' poco significativo creare dei gruppi in questa fase, in quanto tutti i processi stanno analizzando lo stesso albero di gioco seppure su sottoalberi differenti ed hanno la stessa conoscenza.
I gruppi della prima fase vengono momentaneamente disgregati e vanno a costituire un unico gruppo il cui supervisore è il Master. Al termine di questa valutazione, il processo principale riceve i risultati di tutti i worker e fra i punteggi relativi ad una certa mossa, sceglie quello del processo che ha condotto la propria analisi più in profondità.
All'inizio di ogni nuovo ciclo di ricerca si ricostituiscono i gruppi predefiniti.

5.3.2 Struttura del worker

La struttura di un worker rimane la stessa. La sola variazione consiste nel fatto che ogni processo è dotato di un proprio identificatore e di una istanza di ricerca. Al termine della prima fase, se il worker è fra i supervisori (id < ninstanze) invoca la funzione select_group_move per raccogliere i risultati del proprio gruppo (vedi fig. 5.5).
Il lavoro compiuto da un worker alla ricezione del messaggio "SELECT_MOVE" diventa:

```
void worker(id, evalflags)
short int id;
short int evalflags;
{
    ...
     case SELECT_MOVE:
       fase = 1;
```

```
        e_flags = evalflags;
        ElapsedTime(0);
        Sdepth = 0;
        select_my_move (side, 1, evalflags);
        if(identifier < ninstances)
             select_group_move();
        break;
...
```

Nella fase di "Research" ogni worker rileva uno dei sottoalberi e lo analizza così come avveniva in precedenza nella decomposizione statica della seconda fase. A differenza di questa, però, nessun processo rimane sospeso (vedi fig. 5.5).
Al termine della propria valutazione il worker deposita nello spazio delle tuple il proprio risultato in attesa che il Master elabori la mossa definitiva.

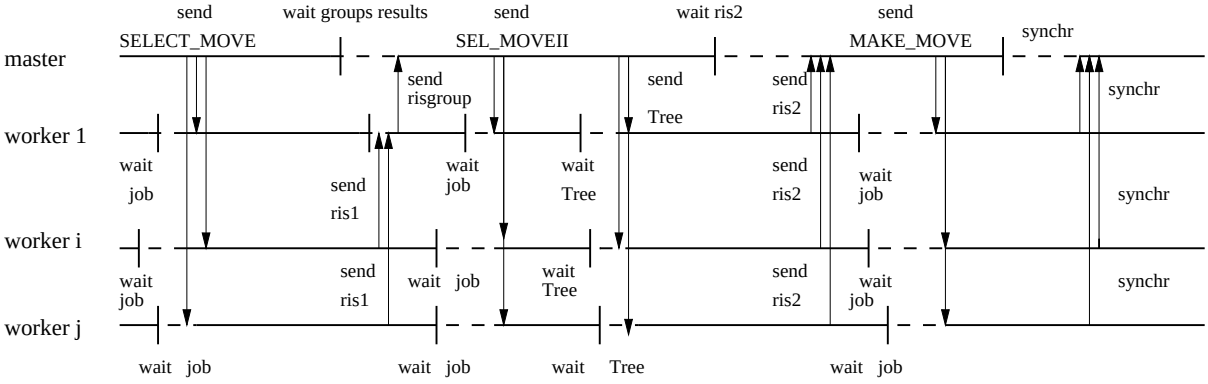


Figure 5.5: Ciclo dei processi nella cooperazione alla pari

5.3.3 Valutazioni

I risultati attesi sono che aumenti il Fpm ed anche l'overhead di ricerca in quanto ogni processo analizza il proprio albero o sottoalbero con un ovvio aumento della quantità dei nodi considerati.

Proc.	Nodes	Time	Vel	Svr	Fpm	OS
1	25892763	6201	4175.579			
3	33169569	3446	9625.528	2.305196	0.868399	0.231372749
7	64677372	4004	16153.19	3.868491	0.652642	0.211935804
11	106126718	4250	24970.99	5.980247	0.643659	0.384005354
15	147740879	4400	33577.47	8.041394	0.636093	0.546236994
19	185101793	4610	40152.23	9.615969	0.606104	0.601813531

Table 5.3: Risultati con cooperazione alla pari e $\frac{2}{3}$ del tempo per la seconda fase

Le valutazioni vengono riportate nelle tabelle 5.3 e 5.4 dove sono indicate le prestazioni del sistema con differenti quantità di processi e con diverse gestioni del tempo. Nel primo caso viene riservato un terzo del tempo alla prima fase di ricerca, nel secondo invece viene utilizzato lo stesso tempo per le due fasi.

Il Fpm risulta incrementato rispetto al valore che appariva in tabella 5.1 e 5.2 in riferimento alla decomposizione statica della "research". Tale incremento risulta maggiore all'aumentare del numero di processi coinvolti. Infatti mentre con il modello di cooperazione alla pari, i worker rimangono sempre attivi, in precedenza si aveva che all'aumentare dei processi vi era un maggiore spreco di tempo di elaborazione.
Notevole è l'incremento dell'OS, si arriva addirittura ad analizzare il 60% di nodi in più rispetto a prima .
L'andamento dei risultati è analogo per l'altra distribuzione del tempo, ma i valori che si ottengono sono piuttosto differenti. Attribuire la metà del tempo di ricerca ad ogni fase comporta un aumento di Fpm, ma una diminuzione dell'overhead si ricerca. Questo risultato esula da quelle che erano le attese. Questo comportamento è certamente dovuto al fatto che, dedicando alla prima fase di ricerca un tempo superiore, i worker possono effettuare una migliore e più approfondita ricerca dell'albero di gioco completo. Questa migliore analisi ha come conseguenza che le differenti istanze di conoscenza tendono a far convergere le mosse proposte su un insieme più ristretto. Quando questo insieme si riduce addirittura ad essere costituito da una sola

Proc.	Nodes	Time	Vel	Svr	Fpm	OS
1	25892763	6201	4175.579			
3	43561710	4529	9618.395	2.303488	0.867829	0.272447699
7	92749106	4711	19687.77	4.714981	0.773569	0.28377641
11	141319630	4737	29833.15	7.144674	0.749516	0.396634491
15	182146211	4751	38338.5	9.181602	0.712107	0.315880779
19	223426671	4749	47047.1	11.2672	0.693011	0.256038206

Table 5.4: Risultati con cooperazione alla pari e metà del tempo per fase

mossa, viene resa inutile la fase della "Research".
Quest'ultima fase comporta due gravosi accessi allo spazio delle tuple (lettura del sottoalbero e raccolta delle mosse di tutti i worker) con conseguenti tempi di inattività per i worker. Evitando quindi questa fase si ottiene che il Fpm aumenta.
Come conseguenza di questo comportamento si ha un OS dall'andamento anomalo. Esso infatti cresce fino ad undici processi, ma poi diminuisce all'aumentare dei worker.
Una giustificazione si ottiene considerando che, fino a sette processi, ogni worker ha una propria istanza di ricerca differente dalle altre, mentre per quantità superiori avviene la costituzione dei gruppi. Evidentemente la costituzione di gruppi di worker porta a scelte migliori nella prima fase evitando così di procedere con la "Research" che comporterebbe un innalzamento del valore OS.
Un aspetto negativo di tale architettura è che fra i worker non vi è alcun tipo di cooperazione durante la ricerca vera e propria che rimane sequenziale. Oltre a questo, i nodi dell'albero che vengono analizzati dai processi sono gli stessi per tutti i worker di un gruppo.
Non è possibile ottenere che ogni worker consideri un albero completamente diverso da quello degli altri in quanto esiste un albero "minimo" che tutti i processi devono prendere in considerazione [Sch89].
I worker possono però collaborare in modo che un processo possa comunicare agli altri la finestra di ricerca (v. cap. 2) sulla quale lavorare. La "window_search" viene creata dai processi per poter applicare dei tagli ai rami

dell'albero in modo da non analizzare sottoalberi inutilmente. L'ampiezza di questo intervallo di ricerca viene settata in base al valore corrente dello "score".
Conviene perciò che questo valore venga condiviso fra tutti i worker. In questo modo ogni processo può condurre la propria analisi sulla base del punteggio migliore del proprio gruppo.
La condivisione dello score produce una maggiore razionalizzazione della visita dell'albero.

Condivisione dello score

L'applicazione di questa modifica porta alcune variazioni sulla struttura dei processi. Devono infatti essere modificati per poter aggiornare il proprio score il più frequente possibile.
Viene introdotta una nuova tupla che contiene lo score :

("SCORE", istanza, score).

Essa è caratterizzata dal punteggio e dall'istanza di ricerca del gruppo cui appartiene. Questo dato è necessario solamente nella prima fase di ricerca, in quanto ogni gruppo ha uno score diverso.
Nella "Research" invece lo score è unico per tutti i worker quindi il dato "istanza" perde di significato.
Più spesso viene aggiornato lo score e, di conseguenza, la finestra di ricerca, migliore è il tipo di analisi dell'albero. La lettura dello score comporta però un costoso accesso allo spazio delle tuple, quindi è necessario trovare una giusta frequenza per tale aggiornamento. In questa architettura lo score viene aggiornato prima di accedere ad un livello successivo dell'albero.

La lettura e l'aggiornamento dello score avvengono in questa maniera:

```
/* AGGIORNAMENTO SCORE */
     rd("SCORE", e_flags, ?park_score);
     if(score > park_score)
    {
        in("SCORE", e_flags, ?park_score);
        if(score > park_score)    /* il mio punteggio e'
```

Proc.	Nodes	Time	Vel	Svr	Fpm	OS
1	25892763	6201	4175.579			
3	30576580	3350	9127.337	2.185886	0.828629	0.135111745
7	63038423	3950	15959.09	3.822008	0.646001	0.181224894
11	93882630	4240	22142.13	5.302769	0.58207	0.224329415
15	133839436	4396	30445.73	7.29138	0.586092	0.400746284
19	179660528	4650	38636.67	9.25301	0.587001	0.554726511

Table 5.5: Risultati coop. e condivisione dello score $\frac{2}{3}$ del tempo per la seconda fase

```
                    comunque migliore di quello globale */
        {
          out("SCORE", e_flags, score);
          park_score = score;
        }
         else
           out("SCORE", e_flags, park_score);
   }
}
```

Il processo che migliora lo score lo deve comunicare globalmente all'intero gruppo. Preleva la tupla privata del gruppo in modo da poter lavorare in mutua esclusione, poi verifica se il suo score è realmente ancora il migliore. In questo caso aggiorna la tupla e la rideposita, altrimenti la mantiene invariata ed aggiorna lo score locale.
L'introduzione dello score comporta delle variazioni anche a livello di prestazioni (v tab. 5.5 e 5.6).
L'andamento è analogo a quello riscontrato nelle tabelle precedenti. Valgono le stesse considerazioni fatte in precedenza, aggiungendo che con la condivisione dello score migliora l'OS ma peggiora il Fpm.
L'overhead di ricerca diminuisce in quanto si ha una ricerca migliore dell'albero. Vengono analizzati meno rami inutili con conseguente miglioramento sull'OS.
L'accesso più frequente allo spazio delle tuple comporta un peggioramento

Proc.	Nodes	Time	Vel	Svr	Fpm	OS
1	25892763	6201	4175.579			
3	36300619	4520	8031.11	1.923353	0.741118	0.060349539
7	80504204	4650	17312.73	4.146188	0.692313	0.114289964
11	127187731	4780	26608.31	6.372365	0.679306	0.256971675
15	161032411	4775	33724.07	8.076501	0.638433	0.163348133
19	215856642	4900	44052.38	10.55001	0.655263	0.213481757

Table 5.6: Risultati coop. e condivisione dello score con metà tempo per fase

del Fpm.

5.4 PVsplit

L'algoritmo di decomposizione Pvsplit fornisce una soluzione a molti dei problemi visti in precedenza.
Con questa tecnica i nodi di decomposizione diventano quelli della variante principale (PV) dell'albero di gioco (vedi fig. 5.6). La variante principale, su alberi ordinati, è quella che si segue percorrendo il primo ramo sinistro dell'albero. Questa sequenza è costituita da quelle mosse che da una prima sommaria analisi sembrano essere migliori delle altre
Tutti i processi scendono ricorsivamente lungo questa variante ed una volta giunti al livello più basso avviene la distribuzione dei lavori. Ogni processo analizza uno dei figli del nodo appena attraversato.

Terminata l'analisi al livello Depth , i processi rientrano dalla loro ultima ricorsione ed analizzano in parallelo i nodi del livello "Depth -1", fino ad arrivare alla radice dell'albero. Questo algoritmo ha una migliore efficacia su alberi ordinati o parzialmente ordinati. Percorrendo infatti la variante principale, ogni processo si trova ad avere una indicazione significativa sulla finestra di ricerca da utilizzare per la valutazione dei nodi.

Questo algoritmo permette ad ogni processo di analizzare un diverso sottoalbero. Infatti la sola parte in comune che tutti sono obbligati a percorrere

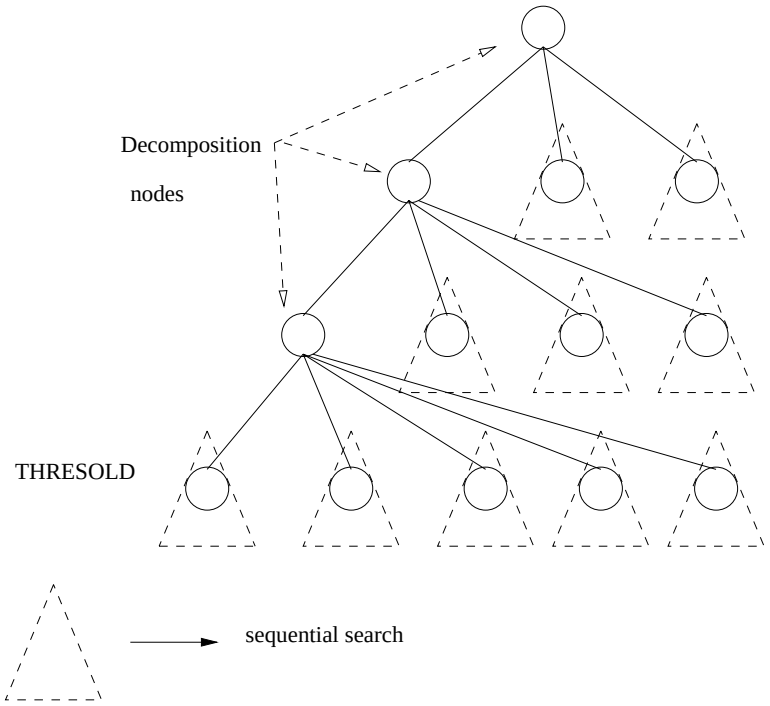


Figure 5.6: Gestione dell'albero con PVsplit

è il ramo principale. Questo costituisce un notevole passo avanti rispetto al modello illustrato nel capitolo precedente nel quale i processi lavoravano in parallelo, ma i nodi visitati erano sostanzialmente gli stessi per tutti i worker.
Fra gli aspetti negativi vi è invece, un gravoso overhead di sincronizzazione dovuto al fatto che all'uscita di ogni ricorsione è necessario un momento di sincronizzazione fra tutti i processi che stanno collaborando.
Infatti, affinché un processo possa iniziare l'analisi ad un livello superiore, deve attendere che il master abbia raccolto tutti i risultati dei worker al livello più basso e che dia in output i lavori di quello attuale.
Affinché la distribuzione di un lavoro porti una effettiva utilità, è necessario che l'albero abbia già una certa profondità.
Questa profondità di soglia "THRESOLD" ([Toz93]), indica a quale profondità si ha la prima decomposizione. La prima decomposizione avrà come nodo radice il nodo al livello THRESOLD, la successiva quello al livello (THRESOLD - 1) e così di seguito fino a risalire alla radice. I sottoalberi la cui radice si trova a livelli di profondità superiori, vengono analizzati sequenzialmente.
Per quanto riguarda la nostra architettura, viene mantenuta la suddivisione dei processi in gruppi la cui costante è l'istanza di ricerca.
Ognuno di questi gruppi è gestito da un supervisore che funge da master per i worker della propria classe. Con questo tipo di struttura il ruolo del supervisore è molto più attivo rispetto a quanto non lo fosse in precedenza.
I gruppi sono costituiti con le stesse modalità utilizzate nell'architettura con cooperazione alla pari. Anche in questo caso il supervisore è il processo il cui identificatore è $id_{sup} < nistanze$. Viene mantenuta anche la suddivisione della ricerca nelle due fasi, con distribuzione della ricerca e della conoscenza nella prima e della sola ricerca nella seconda.
Al termine della prima fase rimane compito del worker raccogliere i risultati di ogni gruppo e costruire il nuovo albero ridotto alle sole mosse proposte.
Nella seconda parte vengono ancora una volta disgregati i gruppi e tutti i worker lavorano sullo stesso albero coordinati da un unico supervisore che è il master.
Con questo tipo di architettura è più significativo prendere in considerazione il flusso di un generico processo supervisore piuttosto che quello del master in quanto anch'esso oltre a generare i processi e a raccogliere i risultati dei gruppi, si comporta come un qualsiasi altro supervisore.

5.4.1 La struttura di un supervisore

Il ruolo del supervisore è quello di distribuire i lavori ai worker ad ogni livello dell'albero.
Contrariamente a quanto avveniva nel modello con cooperazione alla pari, ora ogni worker deve sapere qual'è il suo supervisore e questo deve sapere quanti worker ha a disposizione all'interno del proprio gruppo.
Questo tipo di valutazione si ottiene semplicemente in questa maniera:

```
q = num_macch / ninstances
r = num_macch mod ninstances;
if(identifier mod ninstances < r)
        work_disp = q;
else
        work_disp = q-1;
```

Altrettanto semplicemente è possibile sapere qual'è il proprio supervisore :

$$id_{super} = identifier \ mod \ ninstances \ .$$

Per distinguere il tipo di analisi che viene fatta da un supervisore da quella effettuata da un worker generico, bisogna implementare due nuove funzioni che modifichino (ved. cap. 2) la funzione "search()" originale: *par_search* e *helper_search*.
La prima viene invocata dai supervisori per poter rendere parallela la ricerca una volta giunti ad un certo livello di profondità . La seconda è utilizzata dagli altri worker (esploratori) per poter ricevere i lavori e poter così contribuire alla ricerca.
Queste funzioni vanno a sostituire la "search()" all'interno delle procedure di ricerca ("select_my_move" per i worker e "Master_SelectMove" per il master). Esse hanno infatti lo stesso comportamento della funzione "search()" fino a quando non giungono alla profondità THRESOLD al raggiungimento della quale, inizia la decomposizione dell'albero.

```
int par_search(SHORT side, SHORT ply, SHORT depth, SHORT ext,
 SHORT alpha, SHORT beta, UTSHORT bstline, SHORT rpt,
```

```
   SHORT quiescent, SHORT didnull, SHORT nmosse)
{
  ...
  gestione del tempo
  ...
/* L'albero subisce un primo sommario ordinamento */
  if (ply > 1)
    for (pnt = pbst = TrPnt[ply]; pnt < TrPnt[ply + 1]; pnt++)
       pick (pnt, TrPnt[ply + 1] - 1);
   ....
```

L'albero durante la discesa subisce un ordinamento sommario, nel tentativo di costruire una variante principale.

```
 ...
/*pnt_out e' l'indice del sottoalbero da mettere fra i lavori*/
  pnt_out = 0;

/*done indica se il prossimo indice e' analizzato in parallelo
  o se \'e da analizzare in sequenziale */
   done = false;

/*********ciclo principale****************/
   pnt_out++;
   while( ci sono mosse a qusto livello )
    {
      /*******************************/
      /* IMPOSTO UN ALTRO LAVORO   */
      /*******************************/
      canipar = parallelizzo(0, ply, depth);
      if(canipar)
      {
         if(l == TrPnt[ply]) pnt_out = l+1;
         /* Metto fuori i lavori per gli altri processi */
         /* Qui ne metto fuori tanti quanti sono i worker */
         index = pnt_out;
```

```
        while(index<=pnt_out+w_disp && i<TrPnt[ply+1])
        {

    /* Decomposizione del nodo */
           agenda[ply]++;
           done = 0;
           out("HELP", identifier, index, ply, Sdepth,
            alpha, beta, score, pbst, bestwidth, best,
               temp:4, rcnt, nxtline:, done);
           i++;
         }
         pnt_out = i;
    .....
```

All'interno del main loop, il processo supervisore verifica se la profondità che ha raggiunto è sufficiente per poter iniziare a decomporre i nodi. Questo avviene mediante la chiamata alla funzione "parallelizzo(...)", cui vengono passati i parametri relativi alla profondità attuale ed alla profondità complessiva dell'albero.
Se questa funzione dà responso negativo, si prosegue la discesa lungo la variante principale come avviene normalmente con la funzione search(...).
Nel caso invece che la profondità raggiunta sia quella corretta, il processo decompone il nodo e distribuisce agli esploratori i lavori da svolgere.
La tupla utilizzata per questo è la seguente:

out("HELP", identifier, index, ply, Sdepth, alpha, beta,
score, pbst, bestwidth, best, tempi, rcnt, nxtline:, done);

Essa contiene tutti e soli i parametri necessari al processo esploratore per fare la propria parte di lavoro.
Oltre all'"identifier" ,assolutamente necessario affinché il worker possa riconoscere quali sono le tuple depositate dal proprio supervisore, compaiono alpha, beta e lo score del momento in cui viene depositato il lavoro.
Questi parametri permettono ad un esploratore di essere sempre abbastanza aggiornato sulla migliore finestra di ricerca. Attraverso questa tupla avviene

quindi anche la condivisione dello score.
Il parametro "index" indica quale nodo funge da radice per il sottoalbero che il worker deve analizzare.
Si noti infatti che, al momento di iniziare la ricerca distribuita ogni esploratore ha la stessa visione dell'albero di gioco che ha il proprio supervisore. Questo consente di passare solamente l'indice dell'albero per indicare un nodo, e non l'intera mossa
L'ultimo parametro "done" indica se la tupla residente nello spazio delle tuple è già stata valutata oppure attende di essere prelevata. Nelle tuple date in output dal supervisore questo argomento ha sempre valore zero.
Quando un worker l'avrà analizzata depositerà una tupla analoga contenente il risultato della propria computazione e con "done = 1".
"Agenda[ply]" indica quanti lavori ci sono fuori per il livello di profondità "ply" .
Una volta depositati i lavori nello spazio delle tuple, il supervisore verifica se ci sono risultati disponibili relativi a lavori distribuiti in precedenza. Esso tenta infatti di rilevare una tupla in cui sia "done = 1" per poi elaborarne il risultato. Se non vi è disponibile alcuna tupla già elaborata, allora il supervisore rileva una delle proprie tuple ancora da considerare e la rielabora lui stesso.
Esiste certamente una tupla relativa al proprio gruppo, in quanto inizialmente il supervisore ne aveva depositata una quantità uguale al "numero degli esploratori + 1", quindi una tupla ci deve sempre essere.
Prima di rilevare una tupla, però, riempie lo spazio delle tuple con altri "work_disp" lavori da svolgere. In questo modo se un worker dovesse terminare nel frattempo la propria computazione, non dovrebbe sospendersi in attesa di altri lavori.

```
 ...
  /*  Avendo messo un lavoro in piu' all'inizio ci sara'
  sempre una  tupla HELP nell'agenda[ply], quindi questo
  'in' non andra' mai a vuoto.*/

if(agenda[ply] != 0)
{
/* Verifica se ci sono dei risultati pronti            */
```

```
   found=inp("HELP", identifier, ?pnt, ply, Sdepth, ?alpha_par,
      ?beta_par, ?score_par, ?short,  ?short, ?short, ?temp:,
                 ?rcnt, ?nxtline:, 1);
/* altrimenti  prelvalui stesso uno dei lavori */
   if(!found)
     in("HELP", identifier, ?pnt, ply, Sdepth, ?alpha_par,
       ?beta_par, ?score_par, ?short,  ?short, ?short, ?temp:,
                  ?rcnt, ?nxtline:, ?done);

     agenda[ply]--;
   }
 } /* end if(canipar) */
```

La tupla risultato ha esattamente la stessa forma della tupla con la quale vengono distribuiti i lavori a meno del valore del parametro "done". Questa caratteristica permette al supervisore di rilevare indifferentemente una tupla dallo spazio delle tuple decidendo solo successivamente se analizzarla o meno. Qualora il nodo fosse già stato analizzato da un esploratore, il processo principale non deve fare altro che considerarne i risultati. In caso contrario deve analizzarlo.

```
...
/* Il primo ramo dell'IF riguarda il caso in cui il sub-tree
 sia gia' stato analizzato in parallelo */
  if(done)
  {
    /* Valuatzione del risultato*/
     Tscore[ply] = node->score;
     PV = node->reply;
     /* Lo score prende il valore calcolato in Parallelo */
    node->score = score_par;
     node->width = (ply\%2==1)?(TrPnt[ply+2]-TrPnt[ply+1]):0;
    fatto = false;
  } /* fine IF fatto */
else if (!(node->flags & exact))
{
```

```
    /* fa la ricerca*/
   MakeMove (side, node, &tempb, &tempc, &tempsf, &tempst);
   Tscore[ply] = node->score;
   PV = node->reply;
   if (flag.pvs && depth > 0) {
/* RICERCA*/
   if ((pbst == pnt && canipar))
  {
     node->score= -par_search (xside, ply + 1,
        depth > 0 ? depth - 1 : 0, ext,
        -beta, -alpha,
        nxtline, &rcnt, quiescent, 0, nmosse);
  } else
    node->score= -par_search(....);
  node->reply = nxtline[ply + 1];
  /* reset to try next move */
   UnmakeMove (side, node, &tempb, &tempc, &tempsf, &tempst);
 }
}
/***** uscita da ciclo **************/
```

Uscito dal main loop, il supervisore si appresta a risalire ricorsivamente un livello dell'albero. E' necessario darne comunicazione a tutti gli esploratori del gruppo affinché anch'essi possano compiere la stessa operazione.

```
  if(canipar)
    fine(ply, w_disp, best, pbst);
  return (best);
}
```

La funzione che ha questo compito è "fine(..)". Innanzitutto toglie dallo spazio delle tuple tutti gli elementi che il supervisore vi aveva depositato per evitare che un processo inizi una nuova fase di calcolo ormai inutile.
Successivamente, questa procedura deposita "work_disp" tuple analoghe alle

precedenti, in cui l'indice del sottoalbero da analizzare è "END". Leggendo questa tupla i supervisori comprendono che la loro collaborazione a quel livello ("ply"), non serve più, e quindi risalgono l'albero a livello "ply -1".

```
void fine(ply, w_disp, punt, pbst)
short ply, w_disp, punt, pbst;
{

/* N.B: tolgo solo i lavori ancora da svolgere =>
non rimane nessun lavoro con fatto=0 se non quello con END*/
     while(inp("HELP", identifier, ?short, ply, Sdepth,
       ?short, ?short, ?short, ?short, ?short, ?short, ?SHORT *:,
           ?SHORT, ?UTSHORT *:, 0))
   agenda[ply]--;

/*part_agenda indica quanti lavori sono rimasti nell'agenda[ply]
e che verranno rimossi dai singoli helper. Cosi' il processo
supervisore non rimarra' mai sospeso su qualche tupla */
   part_agenda = agenda[ply] / w_disp;

  /* queste tuple devo far terminare gli helper */
  for(i=0; i< w_disp; i++)
  {
   if(i == w_disp -1)
      part_agenda = agenda[ply];
   agenda[ply] = agenda[ply] - part_agenda;
   out("HELP", identifier, (short)END, ply, Sdepth, part_agenda,
      pbst, punt, (short)0, (short)0, (short)0, temp:4, (short)0,
        line:, 0);
  }

/* decremento la profondita' di parallelizzazione */
    prof_par-- ;

if(flag.timeout && prof_par>0)
   fine(ply-1, w_disp,punt, pbst);
```

```
}
```

Quando il supervisore abbandona un livello "ply", può accadere che alcuni esploratori debbano ancora terminare la loro ricerca. In questo caso il supervisore demanda ad ogni worker del proprio gruppo il compito di rilevare le tuple relative al livello "ply" che arriveranno in seguito. In questa modo esso non rimane sospeso a quella profondità in attesa di lavori inutili.
Rimane da analizzare la funzione "parallelizzo()" che indica al supervisore quando deve iniziare a distribuire i lavori.

```
short parallelizzo(w_t, pl, dpt)
short w_t, pl, dpt;
{
   short resp = 0;
   if(Sdepth > THRESOLD && work_disp != 0)
   {
      if( pl == prof_par && !fondo)
      {
         fondo = 1; /* raggiunto il fondo dell'albero */
         resp = 1;
      }
       else if( pl == prof_par && fondo == 1)
          resp =1;
   }
   return(resp);
}
```

Questa funzione restituisce "vero" quando la profondità raggiunta è maggiore di THRESOLD, oppure quando il livello corrente è inferiore ma siamo già durante la fase di risalita lungo la variante principale ("fondo = 1").
Esiste una funzione analoga, "sospensione()", che però è utilizzata dagli esploratori per decidere quando sospendersi in attesa di ricevere un lavoro dal supervisore.

5.4.2 Struttura di un esploratore

Ogni esploratore percorre l'albero lungo la sua variante principale fino a raggiungere il livello THRESOLD. A questo punto si sospende in attesa di poter collaborare con il supervisore.
La funzione caratteristica è "helper_search". La sua struttura globale è simile a quella di "par_search", infatti anch'essa si comporta esattamente come la funzione di base "search()" fino a che non ha raggiunto la profondità prevista.

```
int helper_search(SHORT side, SHORT ply, SHORT depth, SHORT ext,
 SHORT alpha, SHORT beta, UTSHORT bstline, SHORT rpt,
   SHORT quiescent, SHORT didnull, SHORT nmosse)
{
   ...
   gestione tempi
   ...
/* ordino l'albero per trovare le mosse che sembrano migliori*/
if (ply > 1)
     for(pnt = pbst = TrPnt[ply]; pnt < TrPnt[ply+1]; pnt++)
pick (pnt, TrPnt[ply + 1] - 1);

/* Analogo a parallelizzo */
sosp = sospensione(0, ply, depth);
if(sosp)
{
/* id_super --> id del processo supervisore e che ha la
stessa istanza (eval_flags) */
 if(fase ==1)
  id_super = identifier\%ninstances;
else
 /*nella seconda fse il supervisore e' il master */
   id_super = 0;

/* devo cercare lavori ancora da fare */
  done = 0;

/* RICEZIONE DELLA MOSSA */
```

```
/*  ply ci vuole per cercare i lavori del livello corrente*/
  in("HELP", id_super, ?pnt, ply, Sdepth, ?alpha, ?beta,
     ?score, ?pbst, ?bestwidth, ?best, ?temp:, ?rcnt,
      ?nxtline:, done);
}
```

Innanzitutto il worker stabilisce qual'è il proprio supervisore, poi si prepara a ricevere il lavoro da svolgere.
La tupla

in("HELP", id_super, ?pnt, ply, Sdepth, ?alpha, ?beta,
?score, ?pbst, ?bestwidth, ?best, ?temp:,
?rcnt, ?nxtline:, done);

è caratterizzata da alcuni parametri attuali. "id_super" è l'identificatore del supervisore, ply e Sdpeth indicano l'attuale profondità all'interno dell'albero. Questi valori sono necessari affinché l'esploratore prelevi le tuple corrette e non quelle relative ad un altro gruppo di processi o ad un'altra profondità. L'altro parametro attuale è "done" il cui valore deve essere zero. Questa caratteristica assicura che il processo prelevi delle tuple ancora da analizzare. Una volta terminato il calcolo, la tupla che verrà ridepositata conterrà il valore "done = 1".

```
....
/********* main loop *************/
while((!sosp && pnt < TrPnt[ply + 1] &&
   best <= beta && best != 9999-ply)||(sosp && pnt != END))
  {
    node = &Tree[pnt];

    if (!(node->flags & exact)) {
       /* make the move and go deeper */
     MakeMove (side, node, &tempb, &tempc, &tempsf, &tempst);
     CptrFlag[ply] = (node->flags & capture) ? TOsquare+1 : 0;
```

```
     PawnThreat[ply] = (node->flags & pwnthrt);
     Tscore[ply] = node->score;
     PV = node->reply;

    /* RICERCA */
      if (flag.pvs && depth > 0) {
       if (pbst == pnt) {
        node->score= -helper_search (xside, ply + 1,
             depth > 0 ? depth - 1 : 0, ext,
            -beta, -alpha, nxtline, &rcnt, quiescent, 0);
       } else
         node->score= -helper_search(...)
   ....
      UnmakeMove (side, node, &tempb, &tempc, &tempsf, &tempst);
  }

/* Se sto lavorando in parallelo (=> sosp = 1) restituisco il
risultato della computazione ed aspetto un altro lavoro) */
 /* Restituisce la soluzione al supervisore id_super */
 if(sosp)
    out("HELP", id_super, pnt, ply, Sdepth, alpha, beta,
        node->score, pbst, bestwidth, best, temp:4, rcnt,
           nxtline:, 1);
```

Dopo aver ricevuto il lavoro, l'esploratore analizza il sottoalbero e ne restituisce il risultato inserendo nella tupla di ritorno tutti quei valori che possono essere utili al supervisore, fra cui lo score e gli estremi della finestra di ricerca.

```
...
sosp = sospensione(0, ply, depth);
if(sosp)
{
  /*******************************/
  /* RICEZIONE DI UN ALTRO LAVORO */
   /*******************************/
  done = 0;
```

```
  in("HELP", id_super, ?pnt, ply, Sdepth, ?alpha_in,
      ?beta_in, ?score_in, ?short, ?short, ?best_in,
          ?temp:, ?rcnt, ?nxtline:, fatto);
  if(pnt == END)
    prof_par--;
 }
}
/*********** out of main loop *************/
```

Prima di terminare il ciclo, l'esploratore verifica se ci sono altre tuple che attendono di essere considerate. In questo caso ne rileva una e riprende l'analisi. In caso contrario esce dal ciclo.
Se l'indice del sottoalbero che rileva è "END" allora il processo si appresta a risalire l'albero.
In questo ultimo tipo di tupla i parametri perdono il loro significato consueto. Ad esempio i parametri alpha e beta non indicano più gli estremi della ricerca, ma vengono utilizzati dal supervisore per comunicare agli esploratori alcuni dati che gli sono necessari prima di risalire lungo la variante principale.
Così "beta" ora indica l'indice del sottoalbero risultato migliore, lo "score" indica il punteggio migliore che è stato ottenuto.

```
...
 if(sosp && pnt == END)
{
 best = score;
 pbst = beta;
/* tolgo i lavori che sono gia' stati fatti (fatto = 1)
ma il cui risultato non e'ancora stato prelevato */
  while(inp("HELP", id_super, ?short, ply, Sdepth, ?short,
      ?short, ?short, ?short, ?short, ?short, ?SHORT *:,
          ?SHORT, ?UTSHORT *:, 1));
    rd("TIME", id_super, ?flag.timeout, ?time0);
}
```

Ogni supervisore è il gestore dei tempi di ricerca per il proprio gruppo. Un

Proc.	Nodes	Time	Vel	Svr	Fpm	OS
1	25892763	6201	4175.579			
3	30281474	3450	8777.239	2.102041	0.80068	0.124156358
7	61625908	4000	15406.48	3.689663	0.627095	0.154756943
11	92784892	4246	21852.31	5.23336	0.58576	0.210013743
15	130348881	4404	29597.84	7.08832	0.572555	0.364214585
19	176785683	4655	37977.59	9.095168	0.558693	0.529848494

Table 5.7: Risultati con Pvsplit e $\frac{2}{3}$ del tempo per la seconda fase

esploratore setta i propri tempi leggendo dallo spazio delle tuple

rd("TIME", id_super, ?flag.timeout, ?time0).

Questa lettura viene fatta al termine di ogni invocazione di "helper_search". Nella seconda fase di ricerca tutti i processi si uniscono a comporre un unico gruppo il cui supervisore è il master.
L'andamento dei processi è analogo a quanto avviene nella prima fase.
In entrambe le fasi comunque, i supervisori comunicano il nuovo albero ordinato a tutti gli esploratori prima di iniziare un nuovo ciclo di ricerca ad una profondità superiore.
In questo modo tutti i processi possono avere una copia aggiornata dell'albero e di tutte le variabili principali prima di iniziare un nuovo ciclo.

5.4.3 Valutazioni

I risultati che si ottengono rispettano le attese. Si ottiene un sistema nel quale l'overhead di ricerca risulta piuttosto basso ma che comporta un peggioramento del Fpm.
In particolare le prestazioni migliori si hanno per una equa distribuzione dei tempi fra le due fasi (vedi tab. 5.8). In questo caso infatti l'OS raggiunge il suo massimo valore in corrispondenza di undici processi utilizzati, ma esso risulta comunque contenuto. L'andamento dell'overhead di ricerca è analogo a quello già visto per l'architettura con coordinazione alla pari, valgono quindi le stesse considerazioni fatte in precedenza.

Proc.	Nodes	Time	Vel	Svr	Fpm	OS
1	25892763	6201	4175.579			
3	35311396	4490	7864.453	1.88344	0.727813	0.031454105
7	79512981	4650	17099.57	4.095137	0.68502	0.100570062
11	125200136	4880	25655.77	6.144242	0.658567	0.237328659
15	164460328	4885	33666.39	8.062689	0.637513	0.188112469
19	208973428	4810	43445.62	10.40469	0.647615	0.174786378

Table 5.8: Risultati con PVsplit con metà tempo per fase

Il miglioramento dell'OS comporta un peggioramento del Fpm, causato dall'incremento dell'overhead di sincronizzazione. Sono infatti numerosi i momenti in cui un processo è costretto a sospendersi in attesa di un lavoro o di una sincronizzazione.
Le stesse valutazioni possono essere fatte nel caso di una diversa distribuzione dei tempi (ved. tabella 5.7).
In questo caso risulta più basso il Fpm rispetto alla tabella precedente. Questo avviene poiché nella seconda fase, alla quale in questo caso viene dedicato più tempo, l'overhead di sincronizzazione cresce. Essendoci infatti un solo supervisore per tutti i processi si hanno dei tempi di sincronizzazione superiori alla prima fase.

Chapter 6

Valutazione dei risultati

Nei capitoli precedenti si è condotta un'analisi solamente quantitativa dei differenti approcci alla visita dell'albero di gioco. In questo si procederà invece ad una valutazione della qualità dei giocatori prodotti.
Per ogni giocatore si discute la qualità delle mosse proposte durante l'analisi delle 36 posizioni ed i risultati di un torneo di venti partite condotte contro la versione sequenziale del programma Gnuchess 4.00.
Ogni partita del torneo viene effettuata con la durata massima di 50 mosse al raggiungimento delle quali viene applicata la seguente funzione per l'assegnazione dei punteggi. Tale funzione (vedi fig. 6.1) viene proposta in [San93].

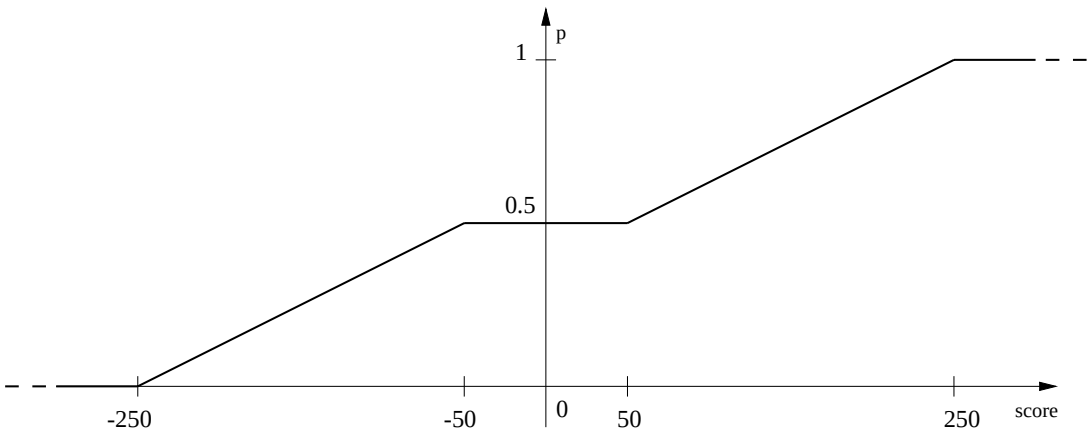


Figure 6.1: Funzione di valutazione

Il punteggio p da assegnare ad un giocatore per ogni partita è il seguente :

- $p = 0$ se la partita è terminata con una sconfitta oppure se è stata interrotta in una situazione in cui lo $score < -250$;
- $p = 1$ se la partita è finita con una vittoria oppure se al momento dell'interruzione si ha lo $score > 250$;
- $p = 0.5$ se la partita è terminata in parità oppure se alla cinquantesima mossa si aveva $|score| < 50$;
- $$p = \frac{score + 150}{400}$$
  se la partita è stata interrotta con $50 < score < 250$;
- $$p = 1 - \frac{|score| + 150}{400}$$
  se la partita è stata interrotta con $-250 < score < -50$.

Per ottenere dei risultati confrontabili con quelli ottenuti in [San93] e in [Toz93] si sono mantenute le stesse condizioni per i giocatori. Sono stati quindi disabilitati l'uso della tabella delle trasposizioni e le funzionalità di amministrazione del tempo straordinario e di ricerca durante le mosse dell'avversario. Si è mantenuto solamente l'uso del libro delle aperture, utilizzando però lo stesso libro per entrambi i giocatori.
Per quanto riguarda l'analisi delle 36 posizioni si sono condotte due serie di valutazioni, una con soli 60 secondi per mossa, l'altra con 3 minuti.
Si analizzino inizialmente le potenzialità del giocatore sequenziale.

6.1 Gnuchess 4.00

Per poter meglio comprendere quale sia la potenza di Gnuchess4.00 pl.77, è stato innanzitutto condotto un torneo contro la versione precedente (pl.63) dello stesso giocatore. Tale versione è quella che viene utilizzata in [San93]. Il

Gnuchess4.00 pl77	Gnuchess4.00 pl63
12.94	7.06

Table 6.1: Risultati del torneo fra Gnuchess4.00 pl 75 e Gnuchess4.00 pl63

risultato, calcolato con la funzione esposta in precedenza, è nitido: Gnuchess4.00 pl 77 ottiene 12,94 punti contro i soli 7,06 di Gnuchess4.00 pl63. 6.1 che segue : Si considerino ora le prestazioni di Gnuchess nel corso dell'analisi delle mosse. Nella tabella 6.2 vengono riportati i risultati della valutazione delle 36 posizioni nel tempo di 60 secondi e di 3 minuti per ognuna. In corrispondenza di 60 secondi per mossa il risultato che si ottiene non è certo dei migliori, considerando che nel corso di una partita il tempo medio per una mossa si aggira intorno al minuto.
Questo tipo di test richiede evidentemente un maggior tempo di ricerca infatti la situazione cambia sostanzialmente concedendo 3 minuti per ogni mossa. In questo caso le posizioni valutate correttamente diventano addirittura 12. Le mosse selezionate nei due casi sono profondamente diverse. Solamente poche volte il risultato ottenuto dopo 60 secondi è lo stesso ricavato dopo 3 minuti di analisi.
Fra questi casi vi sono però le mosse che Gnuchess aveva individuato come le migliori dopo 60 secondi che permangono anche dopo una ricerca più approfondita.

Move	1m	3m	1m	3m
d2b3	c2c8	0	c2c3	0
c3d1	g3e5	0	c3e2	0
d7b5	c8c2	0	a6a5	0
e1e4	e1e4	1	e1e4	1
d8d4	d8d1	0	f7f6	0
e6e5	b5c4	0	b5c6	0
f3h4	f3h4	1	f3h4	1
a7a5	c6g2	0	a7a5	1
c8c4	d6f4	0	c8e8	0
b2b4	b2b4	1	b2b4	1
c8c3	h4f3	0	b2e5	0
g2h3	b2e5	0	g2h3	1
g5h7	h5g6	0	g5h7	1
a2d2	d6d7	0	d6d7	0
d5f6	e4h7	0	e4h7	0
h3h7	g4e2	0	h3h7	1
c3b5	c4d3	0	c3b5	1
h4h3	h4g3	0	h4h3	1
h1h6	d3b3	0	h1h6	1
f4f7	d5c7	0	d1f3	0
e5f7	e5c6	0	e5c6	0
e5f6	g5e4	0	g5e4	0
b3f7	b3f3	0	e2f3	0
h1h7	g3h5	0	g3h5	0
f6f3	d7b5	0	d7b5	0
e5f6	c4b4	0	c4b4	0
g2g3	e5e6	0	g2g3	1
d5d6	a4a5	0	d5d6	1
b7h7	f4g3	0	f4g3	0
f3g4	f4g5	0	f4g5	0
c5b7	d4d5	0	c5b7	1
f3g3	h7c7	0	f3e2	0
h7g8	e4e3	0	a4a3	0
d2b4	a5a6	0	a5a6	0
c4c5	f3d1	0	c4c5	1
g2g1	h7h6	0	f6e5	0
Res		3		12

Table 6.2: Valutazione delle posizioni con gnuchess4.0 pl 77

6.2 Architettura di base

In questo paragrafo si considera il programma distribuito (ChessPar) nella sua versione di base. In questo giocatore vi è una distribuzione dei criteri di conoscenza nella prima fase ed una nuova analisi sequenziale dell'albero ridotto (criterio "Research") nella seconda. In questo caso fra le possibili distribuzioni della conoscenza viene utilizzata quella bilanciata risultata già migliore in [San93] e mostrata in tabella 4.2. Nella tabella che segue (tab. 6.4) vengono mostrati i risultati delle valutazioni delle 36 posizioni effettuate con quantità diverse di istanze di conoscenza e con tempi diversi (60 secondi o 3 minuti).

In questa architettura tutti i processi che collaborano alla ricerca, hanno una istanza di conoscenza differente da tutti gli altri.
In particolare, le istanze di ricerca in corrispondenza di n processori sono le seguenti:

3 processi : mpar mbxp mbkc

7 processi : mpar mbxp mbkc mxar mkcp mcar mbkx

11 procs : mpar mbxp mbkc mxar mkcp mcar mbkx mbar mxkc mkar mbkr

15 processi : mpar mbxp mbkc mxar mkcp mcar mbkx mbar mxkc mkar mbkr mbcr mxkp mxbr mbka

19 processi : mpar mbxp mbkc mxar mkcp mcar mbkx mbar mxkc mkar mbkr mbcr mxkp mxbr mbka mbcp mxka mkrx mkca

Per ogni gruppo di istanze, il nucleo rimane lo stesso. All'aumentare del numero dei processi si aggiungono altre istanze a quelle che già erano presenti.

Nelle tabelle viene mostrato per ogni tipo di giocatore quante e quali mosse sono state valutate correttamente, e quante volte è stato necessario accedere alla seconda fase di ricerca ("research").

In tabella 6.4 i tempi sono ripartiti in questa maniera: un terzo del tempo complessivo è riservato alla prima parte di ricerca parallela; il tempo rimanente è dedicato alla nuova ricerca sequenziale.

I risultati che si ottengono sono inferiori a quelli ottenuti dalla versione di base del programma. Infatti in 60 secondi di ricerca il risultato migliore

che si ottiene è dato da tre posizioni valutate correttamente con 11, 15 o 19 istanze, quando lo stesso risultato si otteneva con Gnuchess iniziale senza alcuna evoluzione.
Analogamente in tre minuti di ricerca, le prestazioni non giustificano lo spreco di risorse necessario per rendere distribuito il programma. Nel caso migliore infatti le mosse corrette sono solamente nove contro le 12 della versione sequenziale.
La situazione cambia sostanzialmente se si gestisce diversamente la ripartizione dei tempi di ricerca. Attribuendo infatti ad ogni fase metà del tempo a disposizione, si ottengono i risultati mostrati nella tabella 6.5. In sessanta secondi si hanno già delle prestazioni migliori rispetto al modello sequenziale. Le posizioni corrette sono infatti minimo tre in corrispondenza di tre processi utilizzati, ma si ha un miglioramento già da quando vengono utilizzate sette istanze di conoscenza. Si ottengono infatti quattro valutazioni esatte. Tale valore però non migliora ulteriormente all'aumentare del numero dei processi utilizzati.
Con tre minuti di analisi a disposizione per ogni posizione, si arriva a prestazioni superiori.
Utilizzando infatti almeno sette istanze di ricerca si giunge ad avere 13 valutazioni esatte, quando nel programma di base se ne ricavavano al massimo 12.
Con i tempi distribuiti equamente fra le due fasi, si ha come conseguenza anche un decremento del numero di volte che si rende necessario accedere alla fase di "research". Evidentemente con una ricerca più approfondita alcune istanze tendono a restituire lo stesso risultato, permettendo così di evitare una nuova fase di valutazione.
Fra i giocatori di questo tipo che vengono considerati, il migliore appare quello che si ottiene con sette istanze di ricerca e tempo equamente distribuito fra le due fasi. Conviene utilizzare solamente sette istanze in quanto ad un aumento di queste non corrisponde un reale miglioramento delle prestazioni. Addirittura aumentando i processi coinvolti si ha un incremento degli accessi alla research, con conseguente spreco di tempo.
Disputando un torneo fra questo giocatore e la versione sequenziale di Gnuchess si ottengono i risultati mostrarti in tabella 6.3.

La versione distribuita ottiene un punteggio di 11,9 a 8,1. Questo risultato è inferiore a quello che si aveva in [San93], dove un giocatore analogo a questo vinceva sulla versione sequenziale con 13,18 punti contro 6,82. La

ChessPar	Gnuchess
11,9	8,1

Table 6.3: Risultato del torneo fra ChessPar di base e Gnuchess4.0

giustificazione è che il programma di base era inferiore a quello utilizzato in questo progetto, come infatti viene provato dal risultato del torneo mostrato in tabella 6.1.

6.3 Decomposizione della seconda fase di ricerca

Si consideri un'altra evoluzione del programma. In questo caso viene decomposta la seconda fase di ricerca che non avviene più sequenzialmente ma viene distribuita su tanti elaboratori quante sono le mosse rimaste da analizzare.
La prima fase rimane invariata.
I risultati che si ottengono da questa architettura sono inferiori a quelli ottenuti con la versione di base di ChessPar.
Infatti analizzando la tabella 6.7 nella quale vengono riportate le prestazioni del sistema con ripartizione dei tempi in $\frac{1}{3}$ e $\frac{2}{3}$, si nota che la quantità di mosse giudicate correttamente è sempre inferiore a quella di ChessPar di base.
Con 60 secondi di analisi, le posizioni valutate esattamente continuano ad essere 2 o 3 a seconda del numero di istanze, ma con 3 minuti a disposizione in questo caso si ottengono al massimo otto responsi esatti con 19 processi coinvolti.
In precedenza si arrivava ad avere nove risposte corrette e già con sette istanze si riusciva ad ottenerne otto.
Rimane invariato il numero di accessi alla seconda fase della ricerca. Si prosegue l'analisi nel 28% e nel 39% ca. dei casi rispettivamente con 60 secondi o con tre minuti a disposizione. Questo era un risultato atteso in quanto la prima fase di ricerca è rimasta identica alla versione di base.
La situazione è analoga se si partiziona il tempo equamente fra le due fasi (vedi tab. 6.8). Anche in questo caso le prestazioni sono comunque inferiori.
Anche con questa versione si arriva ad avere 4 risposte corrette in 60 secondi,

	3pr		7pr		11pr		15pr		19pr	
Move	1m	3m	1m	3m	1m	3m	1m	3m	1m	3m
1	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0
4	0	1	0	1	1	1	1	1	1	1
5	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0
8	0	1	0	1	0	1	0	1	0	1
9	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0
11	1	1	1	1	1	1	1	1	1	1
12	0	1	0	1	0	1	0	1	0	1
13	0	0	0	0	0	0	0	0	0	0
14	0	0	0	0	0	0	0	0	0	0
15	0	0	0	0	0	0	0	0	0	0
16	0	0	0	1	0	1	0	1	0	1
17	0	0	0	0	0	0	0	0	0	0
18	0	0	0	1	0	1	0	1	0	1
19	0	0	0	0	0	0	0	0	0	0
20	0	0	0	0	0	0	0	0	0	0
21	0	0	0	0	0	0	0	0	0	0
22	0	0	0	0	0	0	0	0	0	0
23	0	0	0	0	0	0	0	0	0	0
24	0	0	0	0	0	0	0	0	0	0
25	0	0	0	0	0	0	0	0	0	0
26	0	0	0	0	0	0	0	0	0	0
27	0	0	0	0	0	0	0	0	0	0
28	0	0	0	0	0	0	0	0	0	0
29	1	1	1	1	1	1	1	1	1	1
30	0	0	0	0	0	0	0	0	0	0
31	0	0	0	0	0	0	0	0	0	0
32	0	0	0	0	0	0	0	1	0	1
33	0	0	0	0	0	0	0	0	0	0
34	0	0	0	0	0	0	0	0	0	0
35	0	1	0	1	0	1	0	1	0	1
36	0	0	0	0	0	0	0	0	0	0
exact	2	6	2	8	10	13	8	3	9	3	9
2nd fase	4	9	10	14	11	14	11	15	11	15

Table 6.4: Chesspar di base con $t_1 = \frac{1}{3}$ e $t_2 = \frac{2}{3}$

	3 procs		7procs		11procs		15procs		19procs	
Move	1m	3m	1m	3m	1m	3m	1m	3m	1m	3m
1	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0
4	1	1	1	1	1	1	1	1	1	1
5	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0
7	0	0	1	1	1	1	1	1	1	1
8	0	1	0	1	0	1	0	1	0	1
9	0	0	0	0	0	0	0	0	0	0
10	0	0	0	1	0	1	0	1	0	1
11	1	1	1	1	1	1	1	1	1	1
12	0	1	0	1	0	1	0	1	0	1
13	0	0	0	0	0	0	0	0	0	0
14	0	0	0	0	0	0	0	0	0	0
15	0	0	0	0	0	0	0	0	0	0
16	0	0	0	1	0	1	0	1	0	1
17	0	0	0	0	0	0	0	0	0	0
18	0	1	0	1	0	1	0	1	0	1
19	0	0	0	1	0	1	0	1	0	1
20	0	0	0	0	0	0	0	0	0	0
21	0	0	0	0	0	0	0	0	0	0
22	0	0	0	0	0	0	0	0	0	0
23	0	0	0	0	0	0	0	0	0	0
24	0	0	0	0	0	0	0	0	0	0
25	0	0	0	0	0	0	0	0	0	0
26	0	0	0	0	0	0	0	0	0	0
27	0	0	0	1	0	1	0	1	0	1
28	0	0	0	0	0	0	0	0	0	0
29	1	1	1	1	1	1	1	1	1	1
30	0	0	0	0	0	0	0	0	0	0
31	0	0	0	0	0	0	0	0	0	0
32	0	0	0	1	0	1	0	1	0	1
33	0	0	0	0	0	0	0	0	0	0
34	0	0	0	0	0	0	0	0	0	0
35	0	1	0	1	0	1	0	1	0	1
36	0	0	0	0	0	0	0	0	0	0
exact	3	7	4	13	10	13	4	13	4	13
2nd fase	3	9	8	11	8	11	8	12	9	14

Table 6.5: Chesspar di base con $t_1 = \frac{1}{2}$ e $t_2 = \frac{1}{2}$

ma sono necessarie almeno 11 istanze di ricerca. In nessun caso si riescono ad ottenere le 13 analisi esatte che si erano raggiunte già con sette istanze con la versione di base di ChessPar.
Con metà del tempo per ogni fase, decresce il numero di volte che si prosegue la ricerca. Questo valore si assesta sul 33% dei casi con tre minuti a disposizione.
Queste prestazioni sono dovute al fatto che non vi è una reale collaborazione fra i processi coinvolti. Ognuno analizza sequenzialmente il proprio sottoalbero senza alcun rapporto con gli altri worker che stanno collaborando.
In questo modo è necessario analizzare una notevole porzione dell'albero di gioco, in quanto ogni processo conosce solamente il proprio score. Se gli fosse noto lo score complessivo di tutto il sistema, potrebbe applicare dei tagli più significativi durante la visita dell'albero.
Disputando un torneo fra Gnuchess e un giocatore parallelo progettato secondo questo schema, con sette istanze di ricerca e ripartizione $\frac{1}{2}$ e $\frac{1}{2}$ dei tempi, si ottiene il risultati mostrato in tabella 6.6. Il giocatore parallelo vince con

ChessPar	Gnuchess
11,3	8,7

Table 6.6: Risultato del torneo fra ChessPar con dec. research e Gnuchess4.0

un punteggio piuttosto basso come conseguenza della scarsa collaborazione fra i processi. In questo caso è soprattutto grazie alla prima fase di selezione delle mosse che il sistema ha potuto concludere il torneo con una vittoria.

6.4 Cooperazione alla pari con condivisione dello score

In questa versione di ChessPar collaborano tutte le macchine che si hanno a disposizione. Vengono istituiti dei gruppi di processi che analizzano la posizione con la medesima istanza di ricerca. All'interno del gruppo viene poi scelta la mossa che è stata proposta dal worker che ha analizzato l'albero più in profondità.
Nella seconda fase ogni sottoalbero viene analizzato da un gruppo di processi, la cui conoscenza è sempre quella completa di Gnuchess. Viene inoltre

	3 procs		7procs		11procs		15procs		19procs	
Move	1m	3m	1m	3m	1m	3m	1m	3m	1m	3m
1	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0
4	0	1	0	1	0	1	1	1	1	1
5	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0
8	0	1	0	1	0	1	0	1	0	1
9	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0
11	1	1	1	1	1	1	1	1	1	1
12	0	1	0	1	0	1	0	1	0	1
13	0	0	0	0	0	0	0	0	0	0
14	0	0	0	0	0	0	0	0	0	0
15	0	0	0	0	0	0	0	0	0	0
16	0	0	0	1	0	1	0	1	0	1
17	0	0	0	0	0	0	0	0	0	0
18	0	0	0	0	0	0	0	0	0	1
19	0	0	0	0	0	0	0	0	0	0
20	0	0	0	0	0	0	0	0	0	0
21	0	0	0	0	0	0	0	0	0	0
22	0	0	0	0	0	0	0	0	0	0
23	0	0	0	0	0	0	0	0	0	0
24	0	0	0	0	0	0	0	0	0	0
25	0	0	0	0	0	0	0	0	0	0
26	0	0	0	0	0	0	0	0	0	0
27	0	0	0	0	0	0	0	0	0	0
28	0	0	0	0	0	0	0	0	0	0
29	1	1	1	1	1	1	1	1	1	1
30	0	0	0	0	0	0	0	0	0	0
31	0	0	0	0	0	0	0	0	0	0
32	0	0	0	0	0	0	0	0	0	0
33	0	0	0	0	0	0	0	0	0	0
34	0	0	0	0	0	0	0	0	0	0
35	0	1	0	1	0	1	0	1	0	1
36	0	0	0	0	0	0	0	0	0	0
exact	2	6	2	7	10	5 2	7	3	7	3	8
2nd fase	4	9	10	14	11	14	11	15	11	15

Table 6.7: Chesspar con dec. research e $t_1 = \frac{1}{3}$ e $t_2 = \frac{2}{3}$

	3 procs		7procs		11procs		15procs		19procs	
Move	1m	3m	1m	3m	1m	3m	1m	3m	1m	3m
1	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0
4	1	1	1	1	1	1	1	1	1	1
5	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	1	1	1	1	1
8	0	1	0	1	0	1	0	1	0	1
9	0	0	0	0	0	0	0	0	0	0
10	0	0	0	1	0	1	0	1	0	1
11	1	1	1	1	1	1	1	1	1	1
12	0	1	0	1	0	1	0	1	0	1
13	0	0	0	0	0	0	0	0	0	0
14	0	0	0	0	0	0	0	0	0	0
15	0	0	0	0	0	0	0	0	0	0
16	0	1	0	1	0	1	0	1	0	1
17	0	0	0	0	0	0	0	0	0	0
18	0	0	0	1	0	1	0	1	0	1
19	0	0	0	0	0	0	0	0	0	0
20	0	0	0	0	0	0	0	0	0	0
21	0	0	0	0	0	0	0	0	0	0
22	0	0	0	0	0	0	0	0	0	0
23	0	0	0	0	0	0	0	0	0	0
24	0	0	0	0	0	0	0	0	0	0
25	0	0	0	0	0	0	0	0	0	0
26	0	0	0	0	0	0	0	0	0	0
27	0	0	0	1	0	1	0	1	0	1
28	0	0	0	0	0	0	0	0	0	0
29	1	1	1	1	1	1	1	1	1	1
30	0	0	0	0	0	0	0	0	0	0
31	0	0	0	0	0	0	0	0	0	0
32	0	0	0	1	0	1	0	1	0	1
33	0	0	0	0	0	0	0	0	0	0
34	0	0	0	0	0	0	0	0	0	0
35	0	1	0	1	0	1	0	1	0	1
36	0	0	0	0	0	0	0	0	0	0
exact	3	7	3	11	6	3	12	4	12	4	12
2nd fase	3	9	8	11	8	11	8	12	9	14

Table 6.8: Chesspar con dec. research e $t_1 = \frac{1}{2}$ e $t_2 = \frac{1}{2}$

introdotta la condivisione dello score.
In questa versione cambia la distribuzione delle istanze di ricerca pur sempre utilizzando il sistema della distribuzione bilanciata.
Le istanze utilizzate con "n" processi sono le seguenti:

3 processi : mpar mbxp mbkc . Ogni gruppo ha la propria istanza differente da quella degli altri. In questo caso ogni gruppo ha cardinalità uno.

7 processi : mpar mbxp mbkc mxar mkcp mcar mbkx.
Anche in questo caso i gruppi non sono ancora realmente costituiti. Ogni processo rappresenta un gruppo con la propria istanza.

11 processi : mpar mbxp mbkc mxar mkcp mcar mbkx .
Da questo momento vi sono più processi che istanze ed è quindi possibile ottenere dei gruppi di worker.
Ogni gruppo è caratterizzato dalla propria specifica istanza di ricerca e viene costruito secondo le regole esposte nel capitolo 5. In particolare:

- gruppo1. L'istanza di ricerca è *mpar* e vi appartengono i processi il cui identificatore è $id = n * ninstances$ con "n" intero. Nello specifico il processo '0' (il Master), ed il '7'.
- gruppo2. L'istanza di ricerca è *mbxp* e contiene i processi il cui identificatore è $id = n * ninstances + 1$. In questo caso i processi '1' e '8'.

Proseguendo analogamente nella costituzione dei gruppi si ottengono:

- gruppo3. Istanza *mbkc* e processi '2' e '9';
- gruppo4. Istanza *mxar* e processi '3' e '10';
- gruppo5. Istanza *mkcp* ma vi appartiene un processo solamente (id = 4) poiché sono stati esauriti gli altri worker a disposizione.
- gruppo6. Istanza *mcar* e processo '5';
- gruppo7. Istanza *mbkx* e processo'6'.

15 processi : mpar mbxp mbkc mxar mkcp mcar mbkx.

Le istanze di ricerca rimangono le stesse. I gruppi vengono elaborati secondo l'algoritmo descritto in precedenza. Con 15 processi a disposizione, ogni gruppo ha almeno cardinalità due.

19 processi : mpar mbxp mbkc mxar mkcp mcar mbkx .

In questo caso varierà solamente il numero di processi appartenenti ad ogni gruppo.

Le prestazioni subiscono un miglioramento. In tabella 6.10 si può vedere che il numero di mosse corretto è decisamente superiore a quello delle versioni precedenti nel caso di ripartizione dei tempi in $\frac{1}{3}$ e $\frac{2}{3}$.
Il numero minimo di mosse giudicato esattamente è tre con altrettante istanze nel tempo di 60 secondi. Si arriva però ad avere 11 valutazioni corrette con tre minuti a disposizione e 15 processi coinvolti.
Il miglioramento più significativo si ottiene proprio dal momento in cui ogni gruppo contiene almeno due processi che collaborano.
In questo caso infatti si arriva ad avere fino ad 11 mosse giudicate esattamente.
In questa versione di ChessPar la collaborazione fra i processi risulta più attiva in quanto vi è anche la condivisione dello score all'interno di ogni gruppo.
Si incrementa il numero di accessi alla seconda fase in corrispondenza di 60 secondi per mossa, mentre invece decresce per le ricerche più prolungate. Tale valore rimane invariato nel caso di tre o sette processi utilizzati, in quanto in questi casi non sono ancora costituiti i gruppi di ricerca.
Un ulteriore miglioramento si ottiene se si usa lo stesso tempo di ricerca nelle due fasi (vedi tab. 6.11).
In questo caso si possono avere già cinque risultati corretti con 60 secondi di analisi per ogni mossa, fino a giungere ad averne 15 con tre minuti di ricerca e 19 processi coinvolti. Ancora una volta risulta fondamentale il numero di processi che compone ogni gruppo.
Il numero di volte che si esegue la "research" è inferiore nel caso di tempi equamente distribuiti così come avveniva in precedenza.
L'andamento di tale valore è anomalo. Infatti esso cresce fino ad 11 processi quando si arriva ad accedere alla seconda fase di ricerca nel 39% dei casi, quindi inizia a calare. Ancora una volta la giustificazione è da ricercare nel fatto che all'aumentare del numero dei processi, cresce anche la cardinalità

di ogni gruppo di ricerca. Ciò permette un'analisi più accurata da parte di ogni gruppo che comporta evidentemente una minore dispersione delle mosse proposte.
Eseguendo il torneo con Gnuchess sequenziale (vedi tab. 6.9 si ottiene un risultato decisamente migliore rispetto a quanto avveniva in precedenza.
La versione distribuita risulta più forte della versione sequenziale.

ChessPar	Gnuchess
12,5	7,5

Table 6.9: Risultato del torneo fra ChessPar con coord. pari e Gnuchess4.0

6.5 PVsplit

L'ultima versione che si analizza è quella ottenuta con l'applicazione del Pvsplit sia alla prima fase che alla seconda.
Anche in questo caso si utilizzano sette istanze di ricerca costruite secondo il sistema della distribuzione bilanciata. Rimane analoga alla versione precedente la costituzione dei gruppi.
Le prestazioni che si raggiungono con questa versione di ChessPar sono le migliori.
Con ripartizione dei tempi in $\frac{1}{3}$ e $\frac{2}{3}$ si hanno dei buoni risultati (vedi tab. 6.13). Con 60 secondi per posizione si ottengono sempre 4 o 5 risposte corrette con il massimo localizzato come al solito, in corrispondenza del numero massimo di processori.
Con tre minuti per mossa si giunge fino a 13 responsi esatti.
Come avveniva prima, anche in questo caso è fondamentale la cardinalità dei gruppi. I risultati migliori si ottengono proprio quando ogni gruppo può usufruire della collaborazione di almeno due processi.
Con distribuzione dei tempi in $\frac{1}{3}$ e $\frac{2}{3}$ non si ha una variazione significativa nel passaggio da 15 a 19 processi, mentre con metà del tempo per ogni fase di ricerca questo passaggio coincide con il raggiungimento delle prestazioni in assoluto migliori (vedi tab. 6.14).
Il numero di accessi alla seconda fase non subisce variazioni di rilievo, ma è significativo notare come conservi il suo caratteristico andamento, crescente

	3 procs		7procs		11procs		15procs		19procs	
Move	1m	3m	1m	3m	1m	3m	1m	3m	1m	3m
1	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0
4	1	1	1	1	1	1	1	1	1	1
5	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	1	1	1	1
8	0	1	0	1	0	1	0	1	0	1
9	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0
11	1	1	1	1	1	1	1	1	1	1
12	0	1	0	1	0	1	0	1	0	1
13	0	0	0	0	0	0	0	0	0	0
14	0	0	0	0	0	0	0	0	0	0
15	0	0	0	0	0	0	0	0	0	0
16	0	1	0	1	0	1	0	1	0	1
17	0	0	0	0	0	0	0	0	0	0
18	0	0	0	1	0	1	0	1	0	1
19	0	0	0	0	0	0	0	0	0	0
20	0	0	0	0	0	0	0	0	0	0
21	0	0	0	0	0	0	0	0	0	0
22	0	0	0	0	0	0	0	0	0	0
23	0	0	0	0	0	0	0	0	0	0
24	0	0	0	0	0	0	0	0	0	0
25	0	0	0	0	0	0	0	0	0	0
26	0	0	0	0	0	0	0	0	0	0
27	0	0	0	1	0	1	0	1	0	1
28	0	0	0	0	0	0	0	0	0	0
29	1	1	1	1	1	1	1	1	1	1
30	0	0	0	0	0	0	0	0	0	0
31	0	0	0	0	0	0	0	0	0	0
32	0	0	0	1	0	1	0	1	0	1
33	0	0	0	0	0	0	0	0	0	0
34	0	0	0	0	0	0	0	0	0	0
35	0	1	0	1	0	1	0	1	0	1
36	0	0	0	0	0	0	0	0	0	0
exact	3	7	3	10	10	3	10	4	11	4	11
2nd fase	4	9	10	14	13	14	12	14	10	13

Table 6.10: Chesspar con coord. alla pari e $t_1 = \frac{1}{3}$ e $t_2 = \frac{2}{3}$

	3 procs		7procs		11procs		15procs		19procs	
Move	1m	3m	1m	3m	1m	3m	1m	3m	1m	3m
1	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	1	1
4	1	1	1	1	1	1	1	1	1	1
5	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0
7	0	0	1	1	1	1	1	1	1	1
8	0	1	0	1	0	1	0	1	0	1
9	0	0	0	0	0	0	0	0	0	0
10	0	0	0	1	0	1	0	1	0	1
11	1	1	1	1	1	1	1	1	1	1
12	0	1	0	1	0	1	0	1	0	1
13	0	0	0	0	0	0	0	0	0	0
14	0	0	0	0	0	0	0	0	0	0
15	0	0	0	0	0	0	0	0	0	0
16	0	1	0	1	0	1	0	1	0	1
17	0	0	0	0	0	0	0	0	0	0
18	0	1	0	1	0	1	0	1	0	1
19	0	0	0	1	0	1	0	1	0	1
20	0	0	0	0	0	0	0	0	0	0
21	0	0	0	0	0	0	0	0	0	0
22	0	0	0	0	0	0	0	0	0	0
23	0	0	0	0	0	0	0	0	0	0
24	0	0	0	0	0	0	0	0	0	0
25	0	0	0	0	0	0	0	0	0	0
26	0	0	0	0	0	0	0	0	0	0
27	0	1	0	1	0	1	0	1	0	1
28	0	0	0	0	0	1	0	1	0	1
29	1	1	1	1	1	1	1	1	1	1
30	0	0	0	0	0	0	0	0	0	0
31	0	0	0	0	0	0	0	0	0	0
32	0	0	0	1	0	1	0	1	0	1
33	0	0	0	0	0	0	0	0	0	0
34	0	0	0	0	0	0	0	0	0	0
35	0	1	0	1	0	1	0	1	0	1
36	0	0	0	0	0	0	0	0	0	0
exact	3	9	4	13	11	14	14	4	14	5	15
2nd fase	3	9	8	11	8	11	12	14	10	13

Table 6.11: Chesspar con coord. alla pari e $t_1 = \frac{1}{2}$ e $t_2 = \frac{1}{2}$

fino al raggiungimento di un massimo in corrispondenza di 11 processi, e decrescente all'aumentare del numero dei worker.
L'incremento è quindi legato all'aumento della quantità di gruppi, mentre all'aumentare della cardinalità di ogni gruppo si ripropone quella convergenza delle istanze sulle stesse mosse che già si era notata in altre situazioni.
Con metà del tempo per ogni fase si raggiungono i risultati migliori.
Con 60 secondi a disposizione si arriva a selezionare sei mosse corrette con 19 processi ed analogamente si ottengono 18 valutazioni esatte con lo stesso numero di elaboratori ma con tre minuti di analisi. Con il sistema Pvsplit, il miglioramento delle prestazioni è costante. Ad ogni aumento del numero di processi corrisponde un qualche miglioramento delle prestazioni. Questa caratteristica si può riscontrare con entrambe le ripartizioni del tempo.
Nell'esecuzione del torneo la versione pvsplit del programma vince con il punteggio indicato in tabella 6.12.

ChessPar	Gnuchess
12,95	7,05

Table 6.12: Risultato del torneo fra ChessPar in versione Pvsplit e Gnuchess4.0

Concludendo, si può affermare che il miglior giocatore ottenuto è quest'ultimo in cui la tecnica del Pvsplit viene applicata sia nella prima che nella seconda fase.
Ovviamente i risultati migliori si ottengono all'aumentare degli elaboratori in quanto ogni gruppo viene costituito da un maggior numero di macchine consentendo una superiore distribuzione dell'analisi dell'albero di gioco.
Come ultima prova delle prestazioni della versione distribuita di Gnu-Chess, è stato disputato un torneo contro un altro programma di gioco di scacchi: crafty. È stato innanzituuto disputato un torneo fra la versione iniziale di Gnuchess4.00 sequenziale e crafty. Il risultato è mostrato nella tabella 6.15.

Si è quindi giocato il torneo fra Crafty e ChessPar nella versione ottenuta con Pvsplit ed il risultato è il seguente (tab. 6.16)

	3 procs		7procs		11procs		15procs		19procs	
Move	1m	3m	1m	3m	1m	3m	1m	3m	1m	3m
1	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0
3	1	1	1	1	1	1	1	1	1	1
4	1	1	1	1	1	1	1	1	1	1
5	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0
7	0	0	0	1	0	1	1	1	1	1
8	0	1	0	1	0	1	0	1	0	1
9	0	0	0	0	0	0	0	0	0	0
10	0	0	0	1	0	1	0	1	0	1
11	1	1	1	1	1	1	1	1	1	1
12	0	1	0	1	0	1	0	1	0	1
13	0	0	0	0	0	0	0	0	0	0
14	0	0	0	0	0	0	0	0	0	0
15	0	0	0	0	0	0	0	0	0	0
16	0	0	0	0	0	0	0	0	0	0
17	0	0	0	0	0	0	0	0	0	0
18	0	1	0	1	0	1	0	1	0	1
19	0	0	0	0	0	0	0	0	0	0
20	0	0	0	0	0	0	0	0	0	0
21	0	0	0	0	0	0	0	0	0	0
22	0	0	0	0	0	0	0	0	0	0
23	0	0	0	0	0	0	0	0	0	0
24	0	0	0	0	0	0	0	0	0	0
25	0	0	0	0	0	0	0	0	0	0
26	0	0	0	0	0	0	0	0	0	0
27	0	1	0	1	0	1	0	1	0	1
28	0	0	0	0	0	0	0	1	0	1
29	1	1	1	1	1	1	1	1	1	1
30	0	0	0	0	0	0	0	0	0	0
31	0	0	0	0	0	0	0	0	0	0
32	0	0	0	0	0	1	0	1	0	1
33	0	0	0	0	0	0	0	0	0	0
34	0	0	0	0	0	0	0	0	0	0
35	0	1	0	1	0	1	0	1	0	1
36	0	0	0	0	0	0	0	0	0	0
exact	4	9	4	11	11	13	4	12	5	13	5	13
2nd fase	7	9	12	13	13	14	11	13	10	13

Table 6.13: Chesspar Pvsplit e $t_1 = \frac{1}{3}$ e $t_2 = \frac{2}{3}$

	3 procs		7procs		11procs		15procs		19procs	
Move	1m	3m	1m	3m	1m	3m	1m	3m	1m	3m
1	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0
3	1	1	1	1	1	1	1	1	1	1
4	1	1	1	1	1	1	1	1	1	1
5	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0
7	0	1	0	1	1	1	1	1	1	1
8	0	1	0	1	0	1	0	1	0	1
9	0	0	0	0	0	0	0	0	0	0
10	0	1	0	1	0	1	0	1	1	1
11	1	1	1	1	1	1	1	1	1	1
12	0	1	0	1	0	1	0	1	0	1
13	0	0	0	0	0	0	0	1	0	1
14	0	0	0	0	0	0	0	0	0	0
15	0	0	0	0	0	0	0	0	0	0
16	0	0	0	0	0	1	0	1	0	1
17	0	0	0	0	0	1	0	1	0	1
18	0	1	0	1	0	1	0	1	0	1
19	0	0	0	1	0	1	0	1	0	1
20	0	0	0	0	0	0	0	0	0	0
21	0	0	0	0	0	0	0	0	0	0
22	0	0	0	0	0	0	0	0	0	0
23	0	0	0	0	0	0	0	0	0	0
24	0	0	0	0	0	0	0	0	0	0
25	0	0	0	0	0	0	0	0	0	0
26	0	0	0	0	0	0	0	0	0	0
27	0	1	0	1	0	1	0	1	0	1
28	0	0	0	1	0	1	0	1	0	1
29	1	1	1	1	1	1	1	1	1	1
30	0	0	0	0	0	0	0	0	0	0
31	0	0	0	0	0	0	0	0	0	1
32	0	0	0	1	0	1	0	1	0	1
33	0	0	0	0	0	0	0	0	0	0
34	0	0	0	0	0	0	0	0	0	0
35	0	1	0	1	0	1	0	1	0	1
36	0	0	0	0	0	0	0	0	0	0
exact	4	11	4	141145		16	5	17	6	18
2nd fase	7	10	11	12	12	13	10	11	10	11

Table 6.14: Chesspar Pvsplit e $t_1 = \frac{1}{2}$ e $t_2 = \frac{1}{2}$

Crafty	Gnuchess
11,4	8,6

Table 6.15: Risultato del torneo fra Crafty e Gnuchess4.0

Crafty	ChessPar
7,7	12,3

Table 6.16: Risultato del torneo fra Crafty e ChessPar versione Pvsplit

Chapter 7

Conclusioni

Lo scopo di questo progetto era quello di ottenere un'architettura distribuita in grado di visitare in maniera efficace degli alberi di gioco. Come dominio di applicazione si è utilizzato il gioco degli scacchi. Partendo da un programma sequenziale (Gnuchess4.00) lo si è evoluto fino ad implementare un programma che conducesse l'analisi dell'albero di gioco mediante la distribuzione della ricerca e della conoscenza.

Si è partiti applicando alla versione sequenziale lo stesso tipo di struttura che già era stata proposta in [San93] con la differenza che la stessa architettura è stata applicata su un programma più complesso di quanto non lo fosse quello utilizzato nel progetto suddetto.

In sostanza è stato creato un programma che coordinava più processi dotati di una determinata istanza di ricerca, differente da quella di ogni altro processo coinvolto. Ognuno analizzava l'albero di gioco autonomamente proponendo la mossa ritenuta migliore al termine della ricerca.
Terminata questa fase, un solo processo (Master), si faceva carico di raccogliere tutte le proposte e di scegliere la mossa migliore fra quelle candidate. Questa scelta avveniva utilizzando un criterio di selezione ed in particolare si è analizzato il criterio "Research" che prevedeva una nuova visita dell'albero, ridotto ai soli rami aventi come radice una delle mosse proposte.

Si è osservato che questa architettura, così come era stata presentata in [San93], poteva essere decisamente migliorata in quanto utilizzare un solo processo per la fase di "research" comportava un notevole spreco delle potenzialità del sistema.
Si è perciò pensato di accostare alla distribuzione della conoscenza anche la

distribuzione della ricerca. È stata quindi modificata la fase di "research" facendo in modo che venisse coinvolto più di un processo. In particolare, venivano utilizzati tanti processi quante erano le mosse candidate. Si è visto che le prestazioni complessive del sistema miglioravano poiché il tempo di sospensione complessivo dei processi veniva a diminuire. Considerato il notevole numero di processori, appariva inutile associare ad ognuno un processo con una diversa istanza di ricerca. Si è quindi proceduto a costituire dei gruppi di processi aventi come elemento comune l'istanza di ricerca. Le istanze utilizzate sono divenute "solamente" sette abbinando ad ognuna un gruppo di processi.

Ogni gruppo è stato dotato di un supervisore che si preoccupava di raccogliere le mosse proposte dai worker appartenenti alla stessa classe. Successivamente procedeva alla scelta della mossa migliore, selezionando quella data fuori dal processo che aveva analizzato più in profondità l'albero. Anche in questo caso si proseguiva con l'analisi dell'albero "ridotto" coinvolgendo però, tutti i processi secondo lo stesso criterio. Ogni processo analizzava un sottoalbero ed al termine della ricerca si considerava per ogni sottoalbero lo "score" proposto dal worker che era arrivato a valutare più in profondità.
Per quanto tale tecnica portasse ad un miglioramento del Fpm, si è notato che questo non corrispondeva ad un incremento delle prestazioni. Si è quindi introdotta la condivisione dello score che ha portato a dei miglioramenti anche nella qualità del gioco.
Infine, mantenendo la suddivisione in gruppi, è stata mutata la tecnica di visita dell'albero. Si è utilizzato il Pvsplit sia nella prima che nella seconda fase.

Ogni gruppo analizzava l'albero secondo il proprio criterio di giudizio proponendo poi la propria mossa , mentre nella seconda fase si veniva a costituire un solo gruppo che analizzava l'albero "ridotto" con conoscenza completa e mediante la tecnica del PVsplit.
Questo ha condotto a dei miglioramenti sia a livello qualitativo che quantitativo. È diminuito l'overhead di ricerca grazie ad una distribuzione del lavoro più razionale rispetto a quanto non fosse avvenuto fino ad ora.
Il programma così ottenuto combinava in maniera efficace la distribuzione della conoscenza e della ricerca.
In [Toz93] e [San93] ci si era limitati ad analizzare approfonditamente delle architetture che prevedevano la distribuzione della ricerca in un caso e della conoscenza nell'altro. In questo progetto, utilizzando le valutazioni effet-

tuate all'interno di questi lavori, è stata costruita un'architettura che supera gli evidenti limiti di quella proposta in [San93], utilizzando in ogni momento tutti i processi disponibili. La costituzione di gruppi di processi con la stessa istanza che consentisse di rendere distribuita anche la ricerca con diverse conoscenze, non compariva in alcuno dei due lavori precedenti nei quali la ricerca e la conoscenza avevano pochi punti di contatto.
Superando questo limite si è visto che si possono ottenere dei buoni risultati. Tutte le versioni di ChessPar ottenute si sono mostrate più forti della versione sequenziale del programma, con un picco relativamente alla versione costruita sulla base della tecnica del Pvsplit.

I risultati complessivi dei tornei condotti fra ChessPar e Gnuchess4.00 sono risultati leggermente inferiori a quelli ottenuti in [San93], ma questo è dovuto al fatto che la versione sequenziale del programma utilizzata in questa sede (pl 77) è risultata decisamente più forte della sua versione precedente (pl 63).

È bene ricordare che non si è voluto creare un programma di gioco degli scacchi, ma che questo ha costituito solamente un banco di prova per alcune architetture distribuite. Sarebbe quindi interessante poter applicare le stesse strutture di coordinazione e di distribuzione della conoscenza valutate nel corso di questo progetto, ad un diverso dominio di applicazione e verificarne l'efficacia. In modo particolare va considerato che il programma di base Gnuchess ha spesso condizionato alcune scelte progettuali.

Confermando comunque il linguaggio di coordinazione Network C_linda come lo strumento migliore per questo tipo di lavoro, bisogna affermare che la scarsa modularità e la poca comprensibilità del codice C iniziale ci hanno spesso costretti ad utilizzare delle tuple molto ampie con conseguenze negative sulle prestazioni di tutto il sistema. Sarebbe quindi auspicabile applicare le stesse architetture di coordinazione sviluppate in questa sede ad un diverso programma di gioco degli scacchi: più modulare e più facilmente comprensibile.

Sono comunque numerosi gli spunti per ulteriori ricerche che possono essere offerti da questo tipo di progetto.

- Innanzitutto sarebbe interessante condurre uno studio preciso a proposito della ripartizione del tempo fra le due fasi di analisi. In questa sede ci si è limitati a considerare la distribuzione dei tempi in $\frac{1}{2}$ e $\frac{1}{2}$ oppure in $\frac{1}{3}$ e $\frac{2}{3}$ rispettivamente per la prima e per la seconda fase di analisi.

Si potrebbe però condurre uno studio più approfondito su quale possa essere la ripartizione ottimale dei tempi.

- In questo lavoro si sono sempre utilizzate sette sole istanze di ricerca, istituendo dei gruppi con la medesima istanza qualora il numero dei processi fosse maggiore di quello dei criteri di selezione. Sarebbe interessante valutare quale sia il giusto equilibrio fra cardinalità di ogni gruppo e quantità di istanze. Conviene avere più istanze e gruppi meno numerosi oppure gruppi più ampi e meno istanze di ricerca?

- Il punto debole di queste architetture è costituito dal passaggio fra la prima e la seconda fase, quando il master deve raccogliere i risultati dell'analisi iniziale e costruire l'albero "ridotto" per poter accedere alla "research". Bisognerebbe riuscire ad eliminare questo momento di sospensione provocando dei miglioramenti sulle prestazioni complessive di tutto il sistema.

- La fase di "Research" è stata sempre condotta utilizzando la conoscenza completa del dominio di applicazione, ma è davvero l'unica soluzione o non potrebbe risultare più proficuo rianalizzare l'albero con una serie di nuove istanze?

Questi spunti sono relativi al possibile sviluppo di diverse architetture per la visita di alberi di gioco. Possono quindi venire applicati a qualsiasi altro dominio differente dal gioco degli scacchi.

Per quanto riguarda il gioco degli scacchi di cui si è trattato durante questo lavoro, sarebbe proponibile accostare al programma ChessPar degli altri strumenti comunemente utilizzati. Si potrebbe combinare ChessPar con un database di posizioni dal quale attingere le mosse che appaiono migliori in certe situazioni. Se il database fosse molto ampio si potrebbe sfruttare il parallelismo anche per la ricerca su di esso. Oppure intanto che un processo verifica l'idoneità di una mossa reperibile sul database, gli altri potrebbero condurre la loro consueta valutazione dell'albero di gioco.

Bibliography

[AB89] M. Arango and D. J. Berndt. Tsnet: A linda implementation for networks of unix-based computers. Technical report, Dept. of Computer Sc., Yale Univ., New Haven, Connecticut, August 1989.

[ABC+91] M. Arango, M. Berndt, N. Carriero, D. Gelernter, and D. Gilmore. Experience with linda. Technical Report RR YALEU/DCS/TR-866, Dept. of Computer Science, Yale University, New Haven, CT, 1991.

[Bro96] M.G. Brockington. A Taxonomy of Parallel Game-Tree Search Algorithms. *Journal of the International Computer Chess Association*, pages 162–174, September 1996.

[CFGK95] N. Carriero, E. Freeman, D. Gelernter, and D. Kaminsky. Adaptive Parallelism and Piranha. *IEEE Computer*, 28(1):40–49, 1995.

[CG93] N. Carriero and D. Gelernter. Case Studies in Asynchronous Data Parallelism. *Int. Journal of Parallel Programming*, 22(2):129–149, 1993.

[CGMS94] N. Carriero, D. Gelernter, T. Mattson, and A. Sherman. The Linda Alternative to Message-passing Systems. *Parallel Computing*, 20:633–655, 1994.

[Cia92] P. Ciancarini. *Giocatori Artificiali.* Mursia, 1992.

[Cia94a] P. Ciancarini. A Comparison of Parallel Search Algorithms based on Tree Splitting. Technical Report UBLCS 94-14, Dipartimento

di Scienze dell'Informazione, Università di Bologna, Italy, May 1994.

[Cia94b] P. Ciancarini. Experiments in Distributing and Coordinating Knowledge. *Journal of the International Computer Chess Association*, 17(3):115–131, 1994.

[FGY95] M. Feng, Y. Gao, and C. Yuen. Implementing Linda TupleSpace on a Distributed System. *Int. Journal of High Speed Computing*, 7(1):125–144, 1995.

[Inc90a] Scientific Computer Associates Inc. Linda user's guide and reference manual. Technical report, New Haven, Connecticut, 1990.

[Inc90b] Scientific Computer Associates Inc. Paradise user's guide and reference manual. Technical report, New Haven, Connecticut, 1990.

[JK94] C. Joerg and B. Kuszmaul. Massively Parallel Chess. In *Third DIMACS Parallel Implementation Challenge Workshop*, Rutgers University, October 1994.

[Pla96] A. Platt. *Search Re:search & Re-search.* Thesis Publisher Amsterdam, Erasmus University Rotterdam, 1996.

[San93] A. Santi. *Un programma a conoscenza distribuita per il gioco degli scacchi.* Università di Bologna, Italia, 1993.

[Sch84] J. Schaeffer. The Relative Importance of Knowledge. *Journal of the International Computer Chess Association*, 7:138–145, 1984.

[Sch89] J. Schaeffer. Distributed Game-Tree Searching. *Journal of Parallel and Distributed Computing*, 6:90–114, 1989.

[Toz93] M. Tozzi. *Progetto e realizzazione di un programma distribuito di visita di alberi di gioco.* Università di Pisa, Italia, 1993.