

On applying or-parallelism and tabling to logic programs

Rocha, Ricardo; Silva, Fernando; Costa, Vitor Santos
presented by Possamai Lino*

Indice

1	Introduzione	1
2	WAM	2
3	Parallelism	3
3.1	Or-Parallelism	4
4	Tabling	6
4.1	Risoluzioni SLG	7
5	Or-Parallelism con Tabling	7
5.1	Speculative works	8
6	Performance	8
6.1	Non tabled programs	8
6.2	Tabled programs	9

1 Introduzione

Il Prolog è il più famoso e potente tra i linguaggi di programmazione logica. La sua notorietà aumentò dopo l'introduzione, nel 1983, della prima macchina astratta, denominata Warren Abstract Machine dal nome del ricercatore che la inventò. Nel corso degli anni, il Prolog diventò un linguaggio sempre più utilizzato ed applicato in vari campi quali l'intelligenza artificiale, la gestione dei database, i sistemi esperti. I programmi logici sono scritti in un sottoinsieme della logica del primo ordine, le clausole di Horn, ed utilizzano il metodo della risoluzione per calcolare le risposte alle query. La semantica operativa del Prolog soffre però di alcune limitazioni nell'espressività e nella potenza dichiarativa. Per poter superare questi limiti, una delle tecniche che sono state proposte è rappresentata dal tabling o memoing. In breve, il tabling consiste nella memorizzazione di risultati intermedi per sottogoal in modo tale da poter esser riutilizzati

*matricola 800509

quando questi sottogoal durante il processo di risoluzione vengono richiamati, evitando in questo modo computazioni ridondanti.

Un'altra caratteristica molto importante dei linguaggi logici è la facile predisposizione all'esecuzione parallela, in quanto supportano una tipologia di parallelismo definito implicito. Grazie a questa tecnica, non serve apportare modifiche al codice Prolog e quindi il programmatore non si deve preoccupare della gestione del parallelismo, facendo sì che creare un programma logico parallelo sia facile quando scriverne uno che non lo è.

La relazione che segue sarà strutturata in questa maniera: nella sezione 2 verranno presentate le basi della Wam, la macchina astratta per le esecuzioni sequenziali, in modo tale da poter capire quali sono i punti da modificare per gestire il parallelismo ed il tabling. Nella sezione 3 si presenteranno i dettagli della programmazione logica parallela vedendo quali sono le soluzioni adottate dagli autori. Nella sezione 4 si parlerà di tabling e si entrerà un pò più in profondità per capire questa tecnica di memorizzazione dei predicati. La sezione 5 unisce i concetti presentati nelle sezioni precedenti e introduce il modello Or-Parallelism within Tabling che è alla base del motore implementato dagli autori e chiamato OptYap. L'ultima sezione presenta le performance in termini di speedup e tempi totali di esecuzione ottenute dai motori creati rispetto a prodotti quali XSB, interprete che implementa il tabling.

2 WAM

Per poter addentrarci nei meandri della programmazione logica parallela dotata di tabling, è obbligatorio capire quali sono le fondamenta di ogni interprete Prolog. Nel 1983, D.Warren propose la prima macchina astratta che diventerà uno standard de facto per tutti gli interpreti. Questa macchina non è altro che una struttura dati (stack) grazie alla quale vengono memorizzate le informazioni necessarie all'esecuzione di una query, con in più un insieme di istruzioni di basso livello necessarie alla gestione della macchina ed al mantenimento della stessa in uno stato consistente (vedi figura 1). Le zone che compongono lo stack sono le seguenti: PDL, Trail, Local Stack, Heap, Code Area. Proponiamo una carellata veloce sull'utilità di queste zone:

PDL. È una lista usata durante il processo di unificazione.

Trail. È un array di indirizzi che puntano all'heap o local stack in cui sono memorizzate variabili che devono essere resettate durante il backtracking. Il registro TR punta all'inizio di questa lista.

Local stack. Questa zona è preposta a contenere due elementi: gli *environments* e i *choice points*. I primi sono rappresentati da strutture dati che contengono il corpo di una clausola che viene scelta per l'esecuzione, servono in altre parole a gestire il controllo del flusso del programma, mentre i secondi gestiscono i punti di scelta, che rappresentano quei nodi nell'albero SLD in cui esistono diverse clausole che unificano con il predicato scelto.

Heap. È una lista di celle di memoria usata per contenere variabili e termini composti che non possono essere memorizzati nello stack. Il registro H punta al top della lista.

Code Area. Contiene le istruzioni a basso livello della Wam e il codice compilato del programma logico.

Per una trattazione dell'argomento più dettagliata ed approfondita, si rimanda a [1].

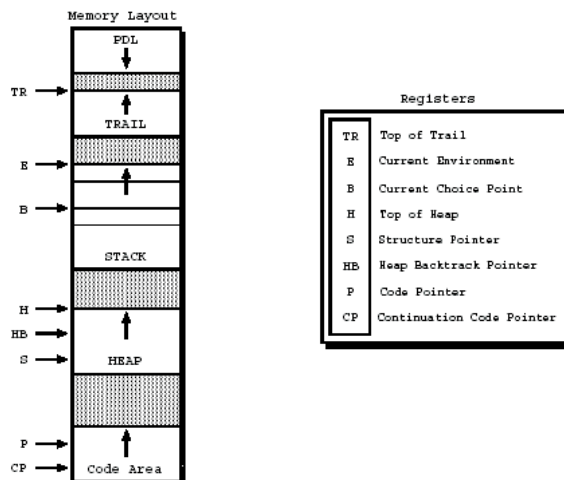


Figura 1: Schema dello stack Wam

3 Parallelism

La fase successiva all'introduzione della Warren Abstract Machine, che decretò l'avvio dell'utilizzo del Prolog sequenziale come linguaggio di programmazione, è stata quella di pensare a come poter affrontare con il Prolog, tutta una serie di problemi di difficile risoluzione e che solitamente, data la loro complessità, vengono eseguiti in un sistema parallelo.

Questo, altro non voleva dire che si pensò a come poter eseguire in parallelo un programma Prolog. In letteratura ci sono diverse tipologie di parallelismo attuabile nei linguaggi di programmazione. Per quanto riguarda la programmazione logica, il parallelismo implicito sembra la soluzione più naturale, se non si vogliono introdurre altri predicati extra-logici, ma soprattutto se non si vogliono alterare programmi già esistenti e funzionanti.

Le tipologie attuabili si possono classificare in base al punto in cui si vuole esplicitare il parallelismo:

Or-Parallelism. Questa tipologia nasce dall'osservazione che i corpi delle clausole le cui teste unificano con il predicato in questione, possono essere eseguiti in workers differenti. Questa forma è senza dubbio quella che pone meno problemi dal punto di vista implementativo ed è quella più utilizzata dai vari interpreti.

And-Parallelism. Quest'altra tipologia si riscontra quando si vogliono eseguire i sub-goal di una query in worker differenti.

Unification Parallelism. Nasce durante il processo di unificazione tra il sottogoal e le teste delle clausole che definiscono lo stesso predicato ed hanno la stessa arietà. L'unification parallelism ha una granularità molto fine rispetto alle precedenti proposte, ma non è stata mai obiettivo di ricerca.

Sebbene la soluzione ideale sia quella di pensare ad un mix di tutte queste tipologie, sono state riscontrate numerose difficoltà dal punto di vista implementativo, che ne hanno sancito

l'effettiva impraticabilità operativa e hanno diretto gli sforzi nel costruire motori con singole tipologie di parallelismo.

3.1 Or-Parallelism

Tra tutte le tipologie di parallelismo applicabili, quella dell'or-parallelism è stata la più adottata nei vari sistemi a indicazione della facilità concettuale implementativa. Questa è stata anche la scelta degli autori per il loro sistema, YapOr. Entrando più nel dettaglio, possiamo iniziare elencando quali sono le caratteristiche che sono alla base del parallelismo di tipo Or:

- Granularità grossa;
- Nessuna limitazione delle potenzialità espressive del Prolog, grazie al parallelismo implicito.
- Ampia scelta dei problemi applicativi implementabili con la versione parallela del Prolog.

Se l'or-parallelism sembra di facile implementazione rispetto alle altre proposte, allo stesso tempo presenta qualche problema che possiamo riassumere come:

1. Il multiple binding representation (che si potrebbe tradurre come rappresentazione multipla del binding delle variabili);
2. Lo scheduling del lavoro da assegnare ai vari processori/processi a disposizione.

Per capire meglio in che cosa consiste il primo problema, consideriamo il seguente programma P:

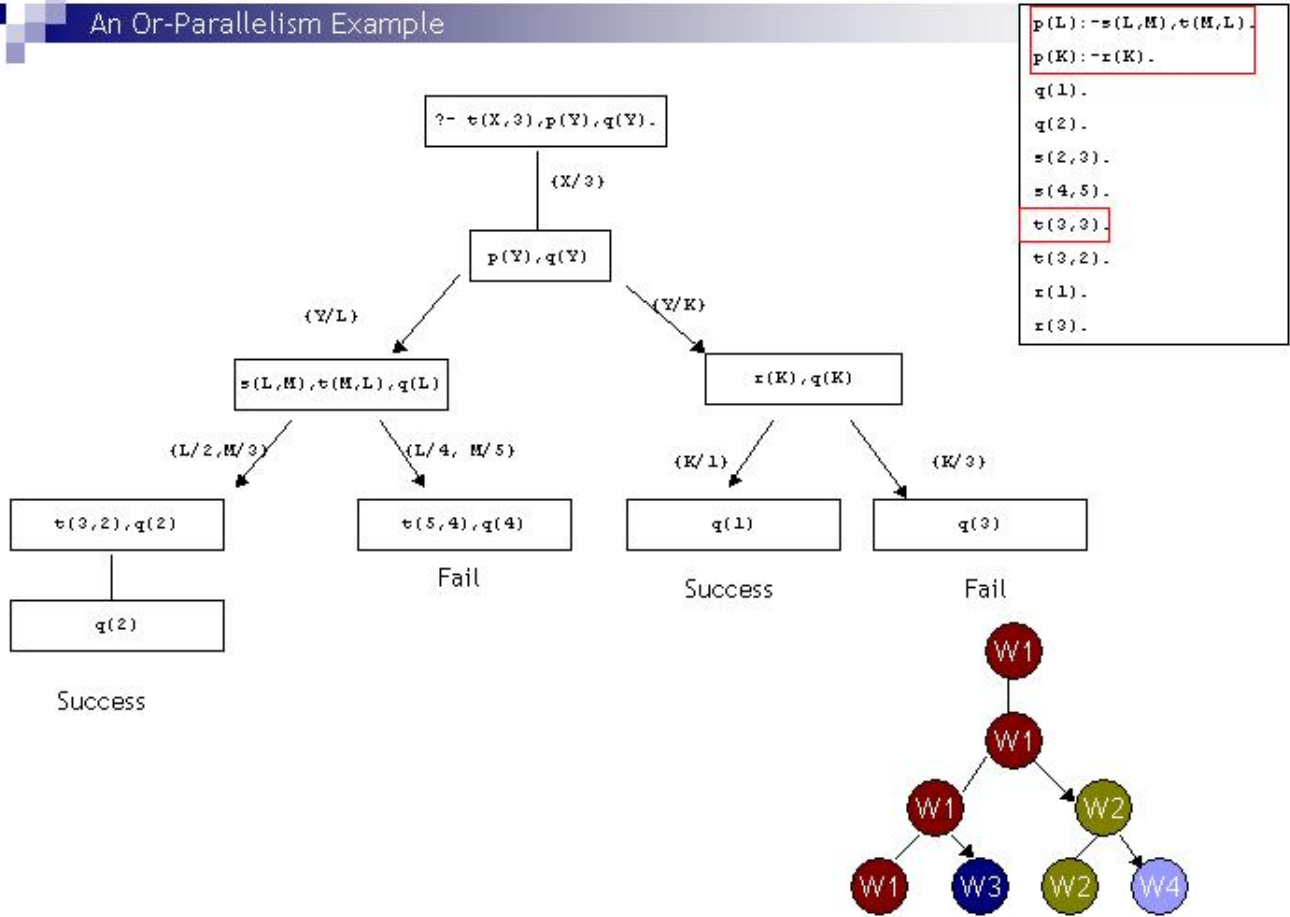
```
p(L) :- s(L,M), t(M,L).  
p(K) :- r(K).  
q(1).  
q(2).  
s(2,3).  
s(4,5).  
t(3,3).  
t(3,2).  
r(1).  
r(3).
```

e la seguente query:

```
?- t(X,3), p(Y), q(Y).
```

Il primo predicato viene risolto senza l'ausilio del parallelismo (perché il predicato $t(X,3)$ unifica con una sola clausola) e quindi viene trovata la sostituzione $X/3$. Nell'analizzare il predicato successivo $p(Y)$, possiamo esplicitare il parallelismo di tipo or, infatti ci sono due clausole applicabili. Da questo punto in poi, la computazione avviene in parallelo, grazie al lavoro assegnato al worker W_1 e W_2 (vedi figura 2). Un esempio di multiple binding representation

An Or-Parallelism Example



Possamai Lino

Logic Programming - Opt

9

Figura 2: Esempio di Or-Parallelism e di multiple binding representation.

è dato dalla variabile Y . Infatti, sia in W_1 che in W_2 , assume assegnazioni diverse e questo implica che non possiamo memorizzare il suo valore nella stessa zona di memoria (supponendo di avere a disposizione un sistema shared memory) altrimenti ci si troverebbe in una situazione di inconsistenza e potrebbe portare anche a soluzioni errate. In questo caso, la variabile Y viene chiamata *conditional variable*, mentre la variabile X , che non soffre dei problemi sopra esposti, viene denominata *unconditional variable* e può essere memorizzata in una zona accessibile a tutti i workers.

Questo esempio dimostra come sia necessario predisporre una struttura dati adeguata per far fronte a questi problemi. In letteratura sono state proposte due soluzioni:

- Environment copy;
- Binding arrays.

Il primo consiste nell'effettuare una copia di tutti gli environment dal worker busy (che cede lavoro) ai worker idle (che hanno bisogno di lavoro), in modo tale che ogni worker lavori con una copia privata dei dati, minimizzando di volta in volta le zone di memoria che devono essere duplicate.

Il binding array consiste nella memorizzazione dei binding condizionali in una struttura dati globale, accessibile da tutti i workers, ed assegnare un identificatore univoco ad ogni binding che identifica il branch al quale il binding appartiene.

Tutte le soluzioni proposte hanno dei costi associati che vanno dal costo di creazione di un nuovo ambiente, al costo dell'accesso delle variabili (conditional) e al binding, al costo di task switching. È stato rilevato che la somma di questi costi non è mai costante ed è di difficile raggiungimento nella pratica, per cui gli sforzi devono essere direzionati verso una costruzione più curata dello scheduler, la cui funzione è quella di assegnare lavoro ai workers.

4 Tabling

Un'area di ricerca molto attiva è quella che affronta i problemi derivanti dalla costruzione di programmi logici che possono non terminare in presenza per esempio di cicli infiniti. Un semplice esempio può essere rappresentato dal seguente programma: $\{p(X) : -p(X) . p(a)\}$. Un'altra area di ricerca si occupa della scarsa efficienza ottenuta nelle classiche derivazioni SLD in presenza di predicati che sono stati già calcolati, magari in un altro ramo dell'albero di derivazione. Un classico esempio è un programma che calcola il numero di Fibonacci.

Il tabling è una strategia proposta come soluzione a questi due problemi. Infatti, grazie a questo metodo, tutte le sostituzioni calcolate di predicati che sono stati segnalati come *tabled* vengono memorizzati in un'area dello stack in modo tale da esser recuperati in un secondo momento.

Quando si parla di programmazione logica con tabling, siamo costretti ad espandere la definizione di risoluzione SLD in modo tale da gestire le operazioni necessarie al tabling.

4.1 Risoluzioni SLG

Un **sistema SLG** è una foresta di alberi SLG con associata una tabella. I nodi root di ogni albero SLG sono istanziazioni di predicati *tabled*. Ogni elemento della tabella è rappresentato da una tripla della forma $\langle \textit{subgoal}, \textit{answerset}, \textit{state} \rangle$ dove con *answerset* si intende un insieme che contiene le risposte calcolate per il relativo subgoal e *state* indica lo stato della computazione, cioè se è *completed* o *incompleted*.

Una valutazione SLG θ per un dato programma definito P, rispetto ad un goal G è una sequenza S_0, \dots, S_n tale che

- S_0 è la foresta composta da un singolo albero SLG il cui nodo root è G e la tabella è formata dall'elemento $\langle G, \emptyset, \textit{incomplete} \rangle$
- per ogni k, S_{k+1} è ottenuto da S_k applicando le regole seguenti:
 - Se si incontra per la prima volta un predicato S *tabled*, si inserisce un riferimento in una tabella $\langle S, \emptyset, \textit{incomplete} \rangle$ dei predicati *tabled* (e si crea un generator node).
 - Se si incontra una variante (a meno di rinomina delle variabili) di un predicato già presente in tabella ma non ci sono soluzioni già trovate, si congela l'esecuzione corrente e si lascia che l'esecuzione venga spostata in un'altra parte dell'albero di derivazione, sperando nel frattempo, di trovare una soluzione al predicato congelato (altrimenti si fallisce, anche se varie scelte sono attuabili in base alla politica di scheduling).
 - Se si trova una soluzione A ad un predicato S la si inserisce nella tabella $\langle S, A' \cup A, \textit{incomplete} \rangle$ (con A' l'*answerset* vecchio) e si procede con l'esplorazione dell'albero, effettuando un backtracking fino all'ultimo punto di scelta oppure (dipende dalle politiche utilizzate) si continua l'esecuzione a partire da nodi sospesi.
 - Se si incontra una chiamata ad un predicato che si trova già in tabella, si prende la c.a.s. e la si applica.
 - Si decide il completamento della foresta SLG.
- Se nessuna regola è applicabile allora S_n è lo stato finale di θ .

5 Or-Parallelism con Tabling

Dopo aver introdotto separatamente i concetti di Or-Parallelism e di Tabling, ed aver analizzato gli aspetti positivi e negativi, vediamo come si può unire le due tecniche in modo tale da sfruttare nel momento opportuno i loro vantaggi.

Gli autori hanno proposto due modelli che descrivono come i due sistemi interagiscono tra di loro: TOP e OPT. Nel primo caso, il parallelismo è supportato considerando che l'elaborazione parallela viene attuata grazie a motori WAM indipendenti, ognuno dei quali gestisce un unico cammino (branch) dell'albero di derivazione. Ogni motore, per poter sfruttare la potenza del tabling, avrà dei comandi aggiuntivi che gli permettono di effettuare le operazioni sulla tabella dei predicati. Nel secondo caso, denominato Or-Parallelism within Tabling, il lavoro è effettuato

da un insieme di motori di tabling che per esplicitare il parallelismo, possono condividere con altri worker alcuni rami dell'albero che stanno analizzando.

Parlando di tabling nella sezione 4, abbiamo sottolineato l'importanza nell'accedere alla tabella dei predicati per inserire o prelevare risposte calcolate. Se ora ci mettiamo nell'ottica del modello OPT, ci accorgiamo che diversi motori di tabling lavorano in parallelo e possono accedere concorrentemente alla tabella dei predicati. In architetture shared memory, come nel nostro caso, nascono una serie di problematiche e overhead principalmente dovute alla gestione dei locking.

Una caratteristica che ritroviamo in tutti e due i sistemi è quella di dover sospendere l'esecuzione in alcuni casi. Il tabling ricorre alla sospensione quando non trova risposte calcolate nella tabella dei predicati oppure quando non sono state calcolate tutte le risposte. Viceversa, i sistemi che implementano l'or-parallelism ricorrono alla sospensione quando, in presenza di side effect, un worker aspetta la terminazione di un altro worker.

5.1 Speculative works

Non possiamo esimerci dall'analizzare le problematiche che sorgono nell'utilizzo nei programmi Prolog di costrutti extra-logici quali il **cut**. Questo costrutto unario, se eseguito, evita di esplorare alcuni rami dell'albero di derivazione, aumentando l'efficienza nell'esecuzione dei programmi. Il rovescio della medaglia è che nell'ottica della programmazione parallela, il cut, se eseguito, potrebbe portare alla cancellazione di risultati già calcolati (per esempio da altri workers) sprestando risorse di calcolo a disposizione e vanificando l'obiettivo principale della computazione distribuita. Queste situazioni vengono denotate come *speculative work*. Dato che la semantica del cut per operazioni che cancellano nodi tabled è ancora un problema aperto, gli autori hanno deciso di non gestire il cut in queste situazioni e di abortire l'esecuzione se necessaria.

6 Performance

Per poter analizzare meglio le performance del nuovo sistema, è stato necessario implementare separatamente quattro sistemi: il primo, denominato Yap, esegue i programmi in modalità sequenziale, senza l'uso di parallelismo e nemmeno di tabling, il secondo, YapOr esegue i programmi in parallelo sfruttando le tecniche dell'Or-Parallelism, il terzo, YapTab, esegue i programmi Prolog usando esclusivamente il tabling e l'ultimo, OptYap, sfrutta congiuntamente le tecniche di parallelismo e di tabling.

Tutti questi interpreti sono stati paragonati con il sistema XSB che rappresentava il sistema che all'epoca era utilizzato per l'esecuzione di programmi logici con tabling.

6.1 Non tabled programs

Il primo passo nell'analisi delle performance è quello di vedere come i tempi di esecuzione si comportano in presenza di programmi in cui nessun predicato definito all'interno è tabled. Questo test è molto importante in quanto permette di analizzare gli overhead introdotti dal modello

quando ad essere eseguiti sono programmi che non sfruttano particolari estensioni (tabling, or-parallelism, tabling+or-parallelism). I programmi scelti per il benchmark sono i classici programmi che vengono usati per il testing di sistemi Prolog.

Bench	Yap	YapOr	YapTab	OPTYap	XSB
cubes	1.97	2.06 (1.05)	2.05 (1.04)	2.16 (1.10)	4.81 (2.44)
ham	4.04	4.61 (1.14)	4.28 (1.06)	4.95 (1.23)	10.36 (2.56)
map	9.01	10.25 (1.14)	9.19 (1.02)	11.08 (1.23)	24.11 (2.68)
nsort	33.05	37.52 (1.14)	35.85 (1.08)	39.95 (1.21)	83.72 (2.53)
puzzle	2.04	2.22 (1.09)	2.19 (1.07)	2.36 (1.16)	4.97 (2.44)
queens	16.77	17.68 (1.05)	17.58 (1.05)	18.57 (1.11)	36.40 (2.17)
<i>Average</i>		(1.10)	(1.05)	(1.17)	(2.47)

Figura 3: Tempi di esecuzione dei vari sistemi con programmi logici non tabled.

Come possiamo vedere in figura 3, Yap ha il migliore risultato e gli altri sistemi YapOr, YapTab e OPTYap introducono rispettivamente degli overhead quantificabili in media come 10%, 5% e 17%. YapOr e OPTYap sono stati eseguiti in un solo worker. Ci sono diverse motivazioni che spiegano il peggioramento nella tempistica di YapOr rispetto alla versione puramente sequenziale e sono dovuti essenzialmente alla gestione del cut, alla verifica se un nodo è pubblico o privato e alla verifica delle richieste condivise. Rispetto a YapTab, l’overhead nasce dai lock alla tabella. OPTYap, ovviamente, eredita tutte e due queste categorie di overhead.

Solitamente, quando si analizzano le performance dei programmi eseguiti in parallelo, si calcola lo speedup, indice rappresentato dal rapporto tra il tempo di esecuzione dello stesso programma in un sistema sequenziale e il tempo di esecuzione della versione parallela. Più questo indice tende al numero di processori a disposizione, più è utile la computazione distribuita.

In figura 4 sono stati paragonati i sistemi YapOr e OPTYap utilizzando rispettivamente 4,8,16,24,32 processori, e come si può notare, i risultati dimostrano che i due sistemi presentano andamenti simili e questo permette di concludere che l’overhead dovuto alla gestione del tabling (in presenza di programmi non tabled) di OPTYap è irrilevante e può essere trascurato.

6.2 Tabled programs

Passiamo ora ad analizzare il comportamento dei sistemi nel caso di programmi con predicati tabled. Dalla figura 5 si osserva che OPTYap introduce, in media, un overhead pari al 15% rispetto a YapTab. Questo valore è prossimo a quello riscontrato con programmi non tabled ed è essenzialmente causato dal locking della tabella per la gestione delle risposte tabled. I benchmark che gestiscono più risposte tabled sono quelli che hanno bisogno di più operazioni di locking. Infatti, più gli overhead sono alti, più risposte vengono trovate.

Per terminare la carellata di tutti i test effettuati, è d’obbligo osservare quali speedup raggiunge OPTYap se eseguito in parallelo e con programmi tabled. La figura 6 rappresenta quindi

Bench	YapOr					OPTYap				
	4	8	16	24	32	4	8	16	24	32
cubes	3.99	7.81	14.66	19.26	20.55	3.98	7.74	14.29	18.67	20.97
ham	3.93	7.61	13.71	15.62	15.75	3.92	7.64	13.54	16.25	17.51
map	3.98	7.73	14.03	17.11	18.28	3.98	7.88	13.74	18.36	16.68
nsort	3.98	7.92	15.62	22.90	29.73	3.96	7.84	15.50	22.75	29.47
puzzle	3.93	7.56	13.71	18.18	16.53	3.93	7.51	13.53	16.57	16.73
queens	4.00	7.95	15.39	21.69	25.69	3.99	7.93	15.41	20.90	25.23
<i>Average</i>	3.97	7.76	14.52	19.13	21.09	3.96	7.76	14.34	18.92	21.10

Figura 4: Confronto tra speedup di YapOr e OPTYap con programmi Prolog non tabled.

Bench	YapTab	OPTYap	XSB
sieve	235.31	268.13 (1.14)	433.53 (1.84)
leader	76.60	85.56 (1.12)	158.23 (2.07)
iproto	20.73	23.68 (1.14)	53.04 (2.56)
samegen	23.36	26.00 (1.11)	37.91 (1.62)
lgrid	3.55	4.28 (1.21)	7.41 (2.09)
lgrid/2	59.53	69.02 (1.16)	98.22 (1.65)
rgrid/2	6.24	7.51 (1.20)	15.40 (2.47)
<i>Average</i>		(1.15)	(2.04)

Figura 5: Tempi di esecuzione di YapTab, OPTYap e XSB con programmi Prolog tabled.

gli speedup ottenuti al variare dei processori a disposizione. Come si può vedere, con alcuni programmi, si raggiungono speedup quasi lineari, mentre per altri programmi la versione sequenziale risulta più veloce della parallela. Le cause dell'effetto contro produttore dell'esecuzione parallela non è dovuto al tempo utilizzato dai workers idle alla ricerca di lavoro da eseguire, ma bensì alla forte competizione per lo stesso lavoro.

Bench	Number of Workers				
	4	8	16	24	32
sieve	3.99	7.97	15.87	23.78	31.50
leader	3.98	7.92	15.78	23.57	31.18
iproto	3.05	5.08	9.01	8.81	7.21
samegen	3.72	7.27	13.91	19.77	24.17
lgrid/2	3.63	7.19	13.53	19.93	24.35
<i>Average</i>	3.67	7.09	13.62	19.17	23.68
lgrid	0.65	0.68	0.55	0.46	0.39
rgrid/2	0.94	1.15	0.72	0.77	0.65
<i>Average</i>	0.80	0.92	0.64	0.62	0.52

Figura 6: Speedup di OPTYap con programmi tabled.

Riferimenti bibliografici

- [1] D.H.D. Warren. An abstract prolog instruction set. *Technical Note, 309*, 1983.