UNIVERSITÀ
DI PARMA

Dipartimento di Scienze Matematiche, Fisiche ed Informatiche
Corso di Laurea in Informatica

Valutazione e miglioramento di un constraint solver parallelo per la libreria Java JSetL

Evaluation and improvement of a parallel solver for the Java library JSetL

Relatore:
Chiar.mo Prof. Gianfranco Rossi

Tesi di Laurea di:
Michele Marchiani

ANNO ACCADEMICO 2019-2020

Ai miei genitori, ai miei parenti e ai miei amici

"Physics is the universe's operating system."
Steven R. Garman

Indice

Introduzione **1**

1 JSetL **3**
1.1 Variabili logiche 3
1.2 Liste logiche 4
1.3 Coppie logiche 4
1.4 Insiemi logici 4
1.5 Vincoli 5
1.6 Risolutore di vincoli 6

2 Parallel Solver **7**
2.1 Architettura del sistema 7
2.2 Master: `ParallelSolver` 7
2.3 Slave: `ParallelSolverSlave` 8
2.4 Gestione dati: `MapJoiner` 9

3 Test affidabilità e valutazione tempi: ricerca punti deboli **10**
3.1 Test19 11
3.2 Test20 12
3.3 Bug 14

4 Implementazione miglioramenti **16**
4.1 Funzionamento di `ThreadPool` 16
4.2 Implementazione di `ThreadPool` 17
4.3 Implementazione di `Worker` 18

4.4 Modifiche a `ParallelSolver` e `ParallelSolverSlave` 20

5 Confronto prestazioni tra solver **22**
5.1 Test brevi 22
5.1.1 Test02 22
5.1.2 Test05 23
5.1.3 Test16 24
5.1.4 Test24 25
5.1.5 Test26 26
5.2 Test medio-lunghi 27
5.2.1 Test14 27
5.2.2 Test19 28
5.2.3 Test20 29
5.2.4 Test22 29
5.2.5 Test29 30
5.3 Analisi risultati 32

6 Conclusioni **36**

Ringraziamenti **38**

Bibliografia **39**

A ThreadPool **40**

B ParallelSolver **43**

C ParallelSolverSlave **49**

Introduzione

Nella ricerca e creazione di processori sempre più prestanti si è inizialmente seguita la strada dell'aumento della frequenza di clock, ma il consumo cresceva esponenzialmente, raggiungendo un limite fisico [2]. Si è passati così allo sviluppo di processori multi-core, che permettono a frequenze più basse di avere prestazioni simili ai più potenti single-core. Questo radicale cambiamento nel comparto hardware si è ripercosso anche nella parte software, passando da esecuzioni sequenziali su un singolo core a esecuzioni parallele su tanti core.

Ormai tutti i dispositivi moderni presentano un'architettura multi-core che per essere sfruttata al meglio richiede software appositi che svolgano più parti dello stesso programma in contemporanea, senza però appesantire troppo la macchina incorrendo in un eccessivo overhead.

Lo scopo di questo lavoro di tesi è testare, mettere a punto e migliorare il solver parallelo per la risoluzione di vincoli della libreria Java JSetL [4] realizzato in un precedente lavoro di tesi [1].

La libreria JSetL (sviluppata presso il Dipartimento di Matematica e Informatica dell'Universita' di Parma) permette di utilizzare uno stile di programmazione dichiarativo all'interno del linguaggio di programmazione Java. Il solver parallelo, denominato `ParallelSolver`, parallelizza la risoluzione dei vincoli di JSetL attraverso l'uso della programmazione parallela e dei thread gestiti attraverso un'apposita libreria Java. L'attuale versione di `ParallelSolver` richiede ulteriore testing in modo da assicurare il corretto funzionamento di tutti i suoi metodi. É importante anche studiare un modo per diminuire l'overhead dato dalla creazione e distruzione dei thread. Alla risoluzione di questi problemi ancora aperti si rivolge questo lavoro di tesi.

L'elaborato di tesi è strutturato nel seguente modo:

- **Capitolo 1.** Nel primo capitolo viene introdotta la libreria JSetL.
- **Capitolo 2.** Nel secondo capitolo si illustrano le principali caratteristiche del solver parallelo.
- **Capitolo 3.** Nel terzo capitolo vengono spiegati nel dettaglio i test effettuati per studiare `ParallelSolver` e trovare i suoi punti deboli.
- **Capitolo 4.** Nel quarto capitolo si spiegano i miglioramenti effettuati alla luce dei test, in particolare l'implementazione di `ThreadPool`.
- **Capitolo 5.** Nel quinto capitolo si analizzano le performance del nuovo solver parallelo comparato con il solver sequenziale, testandoli con diversi programmi e diverso numero di core del processore.
- **Capitolo 6.** Nel sesto capitolo verranno tratte le conclusioni del lavoro svolto.

Capitolo 1

JSetL

JSetL è una libreria Java che offre strumenti per supportare la programmazione dichiarativa tipica dei linguaggi di programmazione logica in un linguaggio imperativo.

1.1 Variabili logiche: `Classe LVar`

La variabile logica rappresenta un valore ignoto, ma al contrario dei linguaggi di programmazione ordinari, le si può associare valori attraverso vincoli sfruttando altre variabili logiche o domini finiti. In particolare, il vincolo di uguaglianza permette di associare alla variabile un valore specifico, che può essere un qualunque oggetto Java. Una variabile legata ad un unico valore è detta *bound*, altrimenti è detta *unbound*.

Una variabile logica in JSetL è un'istanza della classe `LVar`. Esiste però anche la classe `IntLVar`, sottoclasse di `LVar`, che permette di creare variabili logiche che possono contenere solo valori interi.

Esempi:

Creare una variabile logica x senza assegnarle un valore:

```
LVar x = new LVar();
```

Creare una variabile logica y, con nome esterno "Y", assegnandole il valore intero 2:

```
LVar y = new LVar("Y",2);
```

1.2 Liste logiche: Classe `LList`

Una lista logica è uno speciale oggetto logico il cui valore è una coppia <*elementi,coda*>, dove *elementi* è la lista di n `Object`, con n $\geq$ 0, e *coda* è o una lista vuota o una lista *unbound*, che rappresenta il resto della lista.

Per denotare una lista vuota usiamo la notazione astratta [ ], per una lista con almeno un elemento ma avente *coda* vuota usiamo [e_0,...,e_n]; se *coda* è una lista r *unbound* usiamo la notazione (astratta) [e_0,...,e_n |r] per denotarla.

Una lista logica è definita in JSetL come un'istanza della classe `LList`, che estende `LCollection`.

Esempi:

Creare una lista logica l che contiene i caratteri a,b,c e un resto T:

```
LList l = new LList("T").ins('a','b','c');
```

1.3 Coppie logiche: Classe `LPair`

Una coppia logica è una lista logica che può essere *unbound* o *bound* ad una lista [X,Y], dove X e Y sono oggetti logici. Una coppia logica è definita in JSetL come un'istanza di `LPair` che estende `LList`.

Esempi:

Creare una coppia logica P che contiene le variabili logiche X e Y:

```
LVar X = new LVar();
LVar Y = new LVar();
LPair P = new LPair(X,Y);
```

1.4 Insiemi logici: `Classe LSet`

Un insieme logico è molto simile ad una lista logica. Infatti anch'esso è una coppia <*elementi,coda*>, dove *elementi* è la lista di n oggetti di un tipo qualsiasi, con n $\geq$ 0, e *coda* è o un insieme vuoto o un insieme *unbound*. La differenza è che negli insiemi l'ordine e la ripetizione degli elementi non hanno importanza.

Per denotare un insieme logico vuoto usiamo la notazione astratta {}, per un insieme logico con almeno un elemento ma avente *coda* vuota usiamo

$\{e_0,...,e_n\}$; se *coda* è un insieme logico r *unbound* si usa la notazione (astratta) $\{e_0,...,e_n\ |r\}$ per denotarlo.

Un insieme logico è definito in JSetL come un'istanza di `LSet`, che estende la classe `LCollection`.

Esempi:

Creare un insieme logico s che contiene i caratteri a,b,c e un resto T:

```
LSet s = new LSet("T").ins('a','b','c');
```

1.5 Vincoli: Classe `Constraint`

I vincoli rappresentano operazioni su variabili e collezioni logiche che verranno eseguite dal risolutore di vincoli. Queste operazioni si possono applicare anche a variabili o collezioni logiche *unbound*. I vincoli in JSetL sono espressioni aventi una delle seguenti forme:

- Constraint atomico. Questo tipo di vincolo è generato da una collezione di metodi predefiniti che implementano operazioni generali come uguaglianza, disuguaglianza, e operazioni insiemistiche basilari come appartenenza e unione.
- Constraint composto. Questo tipo di vincolo rappresenta un legame tra più vincoli come segue: and (congiunzione), or (disgiunzione), impliesTest (implicazione).

Esempi:

Creare due vincoli logici c1 e c2 che impongono che X sia uguale ad Y e che X sia diverso da Y: `LVar X = new LVar();`

```
LVar Y = new LVar();
Constraint c1 = X.eq(Y);
Constraint c2 = X.neq(Y);
```

Creare un vincolo logico c3 che impone che X appartenga ad A, che X non appartenga a B e che A contenga Y.

```
LSet A = new LSet();
LSet B = new LSet();
```

```
Constraint c3 = X.in(A).and(X.nin(B)).and(A.contains(Y));
```

1.6 Risolutore di vincoli: Classe `Solver`

Per risolvere i vincoli si usa un risolutore di vincoli. In JSetL un risolutore è definito come un'istanza della classe `Solver` per il risolutore sequenziale e `ParallelSolver` per quello parallelo, di cui si parlerà più nello specifico nel prossimo capitolo.

Dopo aver creato il solver, si aggiungono i vari vincoli che dovranno essere soddisfatti e vengono salvati nel *constraint store*. Il solver andrà poi a risolverli riscrivendoli ripetutamente, fino a ottenere `false` oppure una forma semplificata non più riscrivibile (detta "forma risolta"); in quest'ultimo caso, si potrà concludere che il vincolo iniziale è soddisfacibile, dato che la forma risolta è logicamente equivalente al vincolo iniziale ed è sicuramente soddisfacibile. Da notare che non sempre c'è un modo univoco per riscrivere i vincoli: la scelta è non deterministica e, nel caso del solver sequenziale, è implementata tramite backtracking.
Solver mette a disposizione diversi metodi, tra cui i principali sono:

- `void add(Constraint c)`
  Questo metodo serve per aggiungere al *constraint store* uno o più vincoli da risolvere. Non verrà ancora effettuato nessun calcolo o riscrittura.

- `void showStore()`
  Stampa l'unione dei vincoli attivi, presenti nel *constraint store*.

- `boolean check()`
  Controlla se il vincolo nel *constraint store* è risolvibile o meno, rispettivamente restituendo `true` oppure `false`. Se è risolvibile, viene anche generato l'insieme delle soluzioni.

- `boolean nextSolution()`
  Prova a calcolare la prossima soluzione per i vincoli nel *constraint store*, restituendo `true` se la trova, altrimenti `false`.

Capitolo 2

Parallel Solver

Il solver parallelo di JSetL permette di esplorare contemporaneamente più alternative non-deterministiche (*choice-point*) attraverso l'uso dei thread di Java. Se uno di essi trova una soluzione la comunicherà a tutti gli altri, altrimenti verrà fermato e lascerà il core libero per un altro thread. Al contrario il solver sequenziale esplora un'alternativa per volta, utilizzando il backtracking.

2.1 Architettura del sistema

Il solver parallelo è un sistema basato sull'architettura master-slave. Il master crea il primo slave e gli assegna il sistema di vincoli da risolvere; per ogni *choice-point* che lo slave incontra, viene creato un nuovo thread slave che esplorerà l'alternativa successiva e, contemporaneamente, prosegue la risoluzione del proprio disgiunto.

Il master è un'istanza della classe `ParallelSolver`, gli slave sono istanze della classe `ParallelSolverSlave`.

2.2 Master: `ParallelSolver`

L'oggetto `ParallelSolver` verrà istanziato nel primo thread e resterà in esecuzione per l'intera risoluzione dei vincoli. Dopo aver avviato la risoluzione,

viene creato un thread slave che inizierà a lavorare sui vincoli; la creazione di altri thread slave è delegata agli stessi.

Il master si pone quindi in attesa di comunicazioni da parte degli slave. Gli slave possono comunicare di aver trovato degli errori, di aver esplorato tutte le possibili soluzioni e non averne trovata nessuna oppure di averne trovata una che soddisfa tutti i vincoli. In quest'ultimo caso il master lo comunica a tutti gli altri slave e ne ferma l'esecuzione; successivamente comunica la soluzione all'esterno. In caso venisse richiesta un'altra soluzione, il master fa ripartire gli slave e si rimette in attesa.

Il numero degli slave contemporaneamente attivi sarà uguale al numero di core fisici della macchina, in modo da seguire il mapping uno-a-uno tra thread e core.

Dopo aver invocato il metodo `check()` di `ParallelSolver`, il solver salva l'insieme dei vincoli da risolvere nel *constraint store*. Dopo aver creato il primo thread ed essersi sincronizzato su un apposito oggetto, fa una copia profonda (chiamata `clonesMapping`) utilizzando `DeepCloner`, dello store e di una mappa dei cloni delle variabili, in modo che ognuno lavori sulla propria copia. Crea poi un'istanza di `ParallelSolverSlave` passandovi un riferimento all'istanza corrente di `ParallelSolver` e la `clonesMapping` appena creata, per poi aggiungervi tramite `add` dello slave la copia dello store e quindi invoca a sua volta il `check()` dello slave. Successivamente si pone in attesa di comunicazioni da parte degli slave: se una soluzione viene trovato ritornerà `true`, altrimenti `false`.

2.3 Slave: `ParallelSolverSlave`

Gli slave sono i processi che effettivamente svolgeranno il lavoro. Il compito di creare un nuovo slave è lasciato agli slave stessi nel caso in cui si trovino davanti a più possibilità di risolvere il vincolo e che sia necessario esplorarle tutte, cioè quando si apre un nuovo *choice-point*. Questa operazione sarà effettuata solo nel caso ci siano core liberi da utilizzare, altrimenti si procederà in modo sequenziale usando `addChoicePoint()` di Solver.

Se uno slave non trova una soluzione e non trova ulteriori *choice-point* da esplorare, terminerà e lascerà libero il core per eventuali altri slave. Se

invece trova una soluzione, la comunica al master e si pone in attesa.

Se il master comunica che uno degli altri slave ha trovato una soluzione, gli slave si porranno in attesa mantenendo intatto il loro stato in modo da poter riprendere la ricerca in caso che il master glielo comunichi (`nextSolution()`).

Uno slave, per creare un altro thread, usa il metodo di `ParallelSolverSlave AddChoicePoint()`. Per prima cosa controlla che ci siano core disponibili, successivamente salva una copia del contraint store, crea un nuovo thread, all'interno del quale istanzia un nuovo `ParallelSolverSlave` e aggiunge con `add(Constraint)` il vincolo da eseguire. Quando uno slave finisce la sua esecuzione e non trova altre scelte da esplorare, provvederà, tramite gli strumenti forniti da Java, a distruggere il thread e l'istanza di `ParallelSolverSlave`.

2.4 Gestione dati: `MapJoiner`

Ogni slave, nel momento della sua creazione, fa una copia profonda dei dati di input temporanei (i vincoli da risolvere, le variabili e gli insiemi). Questo comportamento è necessario perché tutti gli slave agiscono contemporaneamente sugli stessi dati; se non avessero una copia locale si avrebbero importanti problemi dati dalla *race condition*. Ogni slave modificherà le associazioni variabile-valore solo nell'istanza di dati locale uguale a quella data in input. La copia è realizzata tramite appositi metodi aggiunti alla classe `DeepCloner`.

`MapJoiner` è una classe che permette di unire due o più mappe attraverso i suoi metodi quali `join(Map,Map)` che unisce due mappe e `getComposition()` che unisce tutte le mappe presenti in una lista.

Capitolo 3

Test affidabilità e valutazione tempi: ricerca punti deboli

Testare software significa analizzare un prodotto o un servizio software per raccogliere informazioni sulla sua qualità. Questa pratica è necessaria per identificare casi, se presenti, in cui il programma diverge dal comportamento previsto, causando errori.

Un altro motivo per cui è importante svolgere un'attività di testing, è per trovare eventuali punti deboli del software in termini di ottimizzazione, migliorarli in modo mirato e quindi avere riduzioni in termini di tempo significative.

Il primo obiettivo di questo lavoro di tesi è cercare i punti deboli di `ParallelSolver`, in particolare trovare i punti dell'esecuzione meno efficienti ed eventuali bug. I primi test sono quindi mirati a studiare le tempistiche di creazione e distruzione dei thread e quelle di copia di store, variabili e domini.

Per ottenere i risultati seguenti è stato usato un processore avente 8 core fisici e 8 thread con una frequenza di 4.60 GHz [7]. In base a quanto detto nel capitolo precedente, saranno quindi contemporaneamente attivi durante l'esecuzione di ciascun test al più 8 thread.

3.1 Test19

{X1,X2,X3,X4,X5 |B} = {Y1,Y2,Y3,Y4,Y5 |A} &
X1 nin {Y1,Y2,Y3,Y4,Y5,Y6}

Il Test19 presenta 3 insiemi logici, 2 vincoli e un numero di variabili logiche a scelta in base alla lunghezza in termini di tempo richiesta. In questo caso scegliamo 5 variabili in modo da avere un tempo abbastanza lungo da poter isolare il tempo di copia e il tempo di creazione dei thread.

Questo è il codice Java+JSetL del test:

```
LVar X1 = new LVar("X1"), X2 = new LVar("X2"), X3 = new LVar("X3");
LVar X4 = new LVar("X4"), X5 = new LVar("X5");
LVar Y1 = new LVar("Y1"), Y2 = new LVar("Y2"), Y3 = new LVar("Y3");
LVar Y4 = new LVar("Y4"), Y5 = new LVar("Y5");

LSet S1 = new LSet("B").ins(X1,X2,X3,X4,X5);
S1.setName("S1").output();

LSet S2 = new LSet("A").ins(Y1,Y2,Y3,Y4,Y5);
S2.setName("S2").output();

LSet S3 = LSet.empty().ins(Y1,Y2,Y3,Y4,Y5);
S3.setName("S3").output();

solver.add(S2.eq(S1));
solver.add(X1.nin(S3));

solver.check();

S1.output();
S2.output();
```

Il programma, dopo aver inizializzato i 3 insiemi logici S1, S2 ed S3, pone S2 uguale ad S1 e X1 (contenuto in S1) non appartenente ad S3 (che è un insieme uguale ad S2 senza però il *resto unbound*). Il solver tenterà quindi di unificare X1 a tutte le variabili *bound* di S2 senza riuscirci perché deve poi seguire anche il secondo vincolo.

Il risultato che restituirà il solver sarà però `true` perché X1 è presente nel *resto* di S2, così come Y5 nel *resto* di S1; le restanti variabili sono uguali a due a due.

Questi saranno quindi gli output del programma:

```
S1 = {_X1,_X2,_X3,_X4,_X5|_B}
S2 = {_Y1,_Y2,_Y3,_Y4,_Y5|_A}
S3 = {_Y1,_Y2,_Y3,_Y4,_Y5}
true
S1 = {_X1,_X2,_X3,_X4,_X5,_Y5|_?}
S2 = {_X2,_X3,_X4,_X5,_Y5,_X1|_?}
```

I dati che possiamo estrapolare dall'esecuzione di questo test sono:

- Il tempo totale che impiega `ParallelSolver` a risolvere questo problema è in media 7 200 mS.
- Il tempo totale di copia che impiega `DeepCloner` è in media 1 380 mS.
- Il tempo totale di creazione e distruzione dei thread è in media 5 200 mS.
- Il numero di nuovi thread creati è in media 132 000.

Questi numeri sono stati calcolati usando `System.nanoTime()` e valutano i tempi relativi a tutti i thread in parallelo. Per i tempi di copia e creazione/-distruzione dei thread abbiamo usato variabili globali, accedendovi in modo sincronizzato in modo da avere risultati accurati.

Da questi numeri si evince che, dato l'elevato numero di nuovi thread creati, la percentuale di tempo impiegato a creare e distruggere thread rispetto al totale è alta e sarebbe quindi importante studiare un'ottimizzazione adeguata.

3.2 Test20

un(N1,N2,N3) & un(N3,N4,{[X,X],[X,Y],[Y,X],[Y,Y],[Z,Z],[Z,X] |M}) &
X neq Y & Z neq X & [X,X] nin N3 & [X,X] nin N4.

Il test20 presenta 3 variabili logiche, 6 coppie logiche, 5 insiemi e 6 vincoli. Il programma si basa sulla costruzione dell'insieme N1 attraverso l'unificazione di più altri insiemi, ovvero N2, N3, N4 ed N5.
Questo è il codice Java+JSetL del test:

```
LVar X=new LVar("X"); LVar Y=new LVar("Y"); LVar Z=new LVar("Z");
LPair P1=new LPair(X,X); LPair P2=new LPair(X,Y);
LPair P3=new LPair(Y,X); LPair P4=new LPair(Y,Y);
LPair P5=new LPair(Z,Z); LPair P6=new LPair(Z,X);
LSet N1=new LSet("N1"); LSet N2=new LSet();
LSet N3=new LSet(); LSet N4=new LSet();
LSet N5=new LSet("M").ins(P1,P2,P3,P4,P5,P6); N5.setName("N5").output();

solver.add(N1.union(N2,N3));
solver.add(N3.union(N4,N5));
solver.add(X.neq(Y)); solver.add(Z.neq(X));
solver.add(P3.nin(N3)); solver.add(P3.nin(N4));

solver.check();
N1.output();
```

Dopo aver creato l'insieme N5, formato dalle 6 coppie, il solver inizia a provare tutte le possibili combinazioni per soddisfare i vincoli ma non trova soluzioni, infatti ritornerà `false`.

Questi saranno quindi gli output del programma:

```
N5 = {[_X,_X],[_X,_Y],[_Y,_X],[_Y,_Y],[_Z,_Z],[_Z,_X]|_M}
N1 = unknown
false
```

I dati che possiamo estrapolare dall'esecuzione di questo test sono:

- Il tempo totale che impiega `ParallelSolver` a risolvere questo problema è in media 15 600 mS.
- Il tempo totale di copia che impiega `DeepCloner` è in media 7 080 mS.

- Il tempo totale di creazione e distruzione dei thread è in media 9 500 mS.
- Il numero di nuovi thread creati è in media 239 500.

A differenza del test precedente i tempi di copia e di creazione/distruzione dei thread sono abbastanza simili.

3.3 Bug

Durante lo studio di `ParallelSolver` sono state riscontrate alcune mancanze, alcune semplici da risolvere, altre più complessi. Ne seguono alcuni brevi esempi:

- Nella clonazione tramite `DeepCloner` di `IntLVar` non veniva copiato anche il dominio causando risultati errati, in particolare se veniva aggiunto il vincolo `label()` di queste variabili.
- `ParallelSolver` non aveva implementato il metodo `solve()` ma soltanto il metodo `check()`, e quindi `ParallelSolver` usava il `solve()` di `Solver`, di fatto non utilizzando il parallelismo.
- Utilizzando delle istanze di `LMap` e di `LRel`, `ParallelSolver` restituiva errore. Il bug era da cercare nel metodo `visit(LSet)` che lanciava una `AssertionError`.

Un esempio di questi bug, legato alla copia errata tramite `DeepCloner` delle variabili logiche, si può riscontrare modificando test19 in modo che il solver sequenziale fallisca. `ParallelSolver` invece, a causa di una perdita di informazioni durante la copia, troverà una soluzione errata restituendo `true`.

Questo è il codice del test modificato:

```
LVar X1 = new LVar("X1"), X2 = new LVar("X2"), X3 = new LVar("X3");
LVar X4 = new LVar("X4"), X5 = new LVar("X5");
LVar Y1 = new LVar("Y1"), Y2 = new LVar("Y2"), Y3 = new LVar("Y3");
LVar Y4 = new LVar("Y4"), Y5 = new LVar("Y5");
```

```
LSet S1 = new LSet("B").ins(X1,X2,X3,X4,X5);
LSet S2 = new LSet("A").ins(Y1,Y2,Y3,Y4,Y5);

solver.add(S2.eq(S1));
solver.add(X1.nin(S2));

return solver.check();
```

In questo modo, imponendo che X1 non sia in S2, non esiste soluzione per cui siano validi entrambi i vincoli perché S2, non potendo contenere X1, non potrà risultare uguale ad S1. Questo malfunzionamento era causato da un problema di `DeepCloner` per cui alcune copie perdevano il riferimento all'originale.

Capitolo 4

Implementazione miglioramenti

In questo capitolo verrà descritto come è stato implementato un gestore di thread per supportare l'esecuzione parallela di JSetL e citate le varie modifiche applicate al codice del solver parallelo. Differentemente da `ParallelSolver` originale, il gestore permette di evitare la continua creazione e distruzione di thread creandone un numero che rispecchia quello dei core fisici (ad eccezione dei processori dotati di multithreading). Il gestore mantiene una lista di istanze di `ParallelSolverSlave` pari al numero di thread, una per ognuno.

4.1 Funzionamento di `ThreadPool`

`ThreadPool` è una classe Java che ha il compito di creare e gestire i thread che serviranno al solver parallelo per risolvere i vincoli. Quando viene istanziato, `ThreadPool` crea una lista (`pool`) dove tenere traccia in ordine dei compiti assegnatigli da `ParallelSolver` e `ParallelSolverSlave` (in dettaglio nelle prossime sezioni) e un array di thread, in cui effettivamente verranno svolte le computazioni.

Per gestire al meglio la lista di compiti è stata creato una classe denominata `Worker` che estende la classe `Thread`. Tramite variabili sincronizzate, ogni istanza di `Worker` esegue un compito specifico preso da `pool`; se la lista è vuota esso si porrà in stato di attesa, pronto per un eventuale `notify()`.

La creazione e l'avvio di nuovi `Worker`, che anche in questo caso sono inferiori o uguali di numero a quello dei core fisici della macchina, saran-

no effettuati in contemporanea all'arrivo di nuovi compiti in modo da non appesantire eccessivamente il sistema in caso di problema logico semplice, impiegando quindi un tempo minore.

4.2 Implementazione di `ThreadPool`

La classe `ThreadPool` contiene i seguenti campi:

`private Thread[] pool`: questo array serve per contenere e tenere traccia dei thread che verranno usati dal gestore.

`private List<Runnable> queue`: in questa lista sono inseriti tutti i compiti che verranno poi estratti ed eseguiti dai `Worker`.

`int working`: questa variabile tiene traccia del numero di `Worker` attualmente istanziati.

`int poolSize`: questa variabile rappresenta il numero massimo di thread che possono essere contemporaneamente in esecuzione.

La classe contiene i seguenti metodi:

`public ThreadPool(int size)`: questo metodo è il costruttore. Riceve in input il numero massimo di thread che possono contemporaneamente operare su quella data macchina e lo copia nella variabile `poolSize`. Successivamente istanzia la `LinkedList queue` e l'array `size`, infine imposta `working` a 0.

`public synchronized void start()`: questo metodo viene chiamato quando si vuole effettivamente far partire un `Worker`. Quello che fa è istanziarne uno tramite `new()`, avviarlo tramite `start()` e poi increamentare la variabile `working`.

`public synchronized void add(Runnable runnable)`: questo metodo serve per aggiungere i compiti, che saranno trattati come `Runnable` generici, alla coda di esecuzione, ovvero la lista `queue`. Prima di far questo, controlla se il numero di thread attualmente in esecuzione è inferiore al numero massimo,

cioè `poolSize`,l ed in questo caso chiama il metodo di `ThreadPool start()`. Infine chiama `notifyAll()` per svegliare eventuali thread in stato di *wait*.

`public synchronized void stop()`: questo metodo andrà a fermare tutti i thread in esecuzione nel momento in cui avranno esaurito i compiti nella lista `queue`, aggiungendo un tipo speciale di `Runnable` creato apposta per questo motivo, denominato `Stop`.

Di seguito il codice Java del costruttore di `ThreadPool` e del metodo `add(Runnable)`:

```
public ThreadPool(int size) {

   queue = new LinkedList<>();

   pool = new Thread[size];

   poolSize=size;
   working=0;

}

public synchronized void add(Runnable runnable) {

   if (working<poolSize) {
      start();
   }
   queue.add(runnable);

   notifyAll();
}
```

4.3 Implementazione di `Worker`

La classe `Worker` è una classe `private` che estende `Thread`. Ha solo un metodo cioè `public void run()`.

Questo metodo presenta un loop infinito tramite `for(;;)`, è presente anche un `try..catch` in modo da poter controllare e gestire eventuali eccezioni.

All'interno del `try` il metodo crea un `Runnable` di nome `runnable` e un Object, di nome `sempahore`, a cui viene passato `ThreadPool.this` che fungerà da semaforo per la sincronia. Segue infatti una parte di codice `synchronized` sull'`object`.

`Sempahore` controllerà per prima cosa se la coda `queue` è vuota e in quel caso si porrà in stato di *wait*. Altrimenti salverà in `runnable` il primo compito nella lista `queue`, se quest'ultimo è una istanza di `Stop` verrò chiamato `return` per uscire dal ciclio e chiudere il thread; altrimenti rimuoverà il compito dalla lista e fuori dal codice sincronizzato farà partire il `Runnable` appena rimosso.

Di seguito il codice Java del metodo textttrun() di textttWorker:

```
public void run() {
   for(;;) {
      try {
         Runnable runnable;

         Object semaphore = ThreadPool.this;

         synchronized(semaphore) {
            while(queue.isEmpty())
               semaphore.wait();

            runnable = queue.get(0);

            if(runnable instanceof Stop)
               return;

            queue.remove(0);
         }

         runnable.run();
      } catch(InterruptedException e) {
         // Blank
      } catch(Throwable t) {
         t.printStackTrace();
      }
   }
```

```
}
```

4.4 Modifiche a `ParallelSolver` e `ParallelSolverSlave`

La grande differenza in `ParallelSolver` e `ParallelSolverSlave` rispetto alla precedente implementazione è che la gestione dei thread è affidata a `ThreadPool`, invece di crearli direttamente all'interno delle classi.

Nel costruttore di `ParallelSolver` è stata aggiunta la creazione e istanziazione di `ThreadPool` senza però creare alcun `Worker`. Esse avvengono solo successivamente alla chiamata del `check()`, che andrà ad aggiungere alla lista di comandi la prima istanza di `ParallelSolverSlave`, attraverso il metodo di `ThreadPool add`, che verrà avviata nel primo `Worker`, passandole anche un riferimento al gestore di thread.

All'interno di `ParallelSolverSlave`, analogamente al master, le nuove istanze di altri slave saranno create aggiungendo il comando alla lista presente in `ThreadPool` sempre attraverso il suo metodo `add`.

Di seguito la porzione di codice rispettivamente di `ParallelSolver` e `ParallelSolverSlave` in cui si delega a `ThreadPool` la creazione del nuovo `ParallelSolverSlave` tramite il metodo di `ThreadPool add(Runnable)`.

```
synchronized (mutex){
   synchronized (w) {
      try {
         pool.add(()->{
            Constraint copy;
            Map<Object, Object> clonesMapping;
            synchronized (w) {
               DeepCloner deepCloner = new DeepCloner();
               copy = deepCloner.visit(store);
               clonesMapping = deepCloner.getClonesMap();
               w.notify();
            }
            ParallelSolverSlaveP ParallelSolverP = new ParallelSolverSlaveP
                (this, clonesMapping, pool);
```

```
            ParallelSolverP.add(copy);

            ParallelSolverP.check();
         });
      w.wait();
   } catch (InterruptedException e) {
      e.printStackTrace();
   }
}
```

```
synchronized (w) {
   try {
      pool.add(()->{
         Constraint copy;
         Map<Object, Object> clonesMapping;

         synchronized (w) {
            DeepCloner deepCloner = new DeepCloner();
            copy = deepCloner.visit(store);
            clonesMapping = deepCloner.getClonesMap();
            w.notify();
         }

         ParallelSolverSlaveP ParallelSolverP = new ParallelSolverSlaveP(
             master,clonesMapping,this,pool);
         ParallelSolverP.add(copy);

         ParallelSolverP.check();
      });

      w.wait();
   } catch (InterruptedException e) {
      e.printStackTrace();
}
```

Capitolo 5

Confronto prestazioni tra solver

In questo capitolo verranno spiegati i test più rilevanti effettuati per controllare la validità dei miglioramenti applicati a `ParallelSolver`. I test saranno divisi in due categorie, quelli brevi che servono a controllare che i solver paralleli mantengano un tempo simile a quello del solver sequenziale e test lunghi per misurare lo *speedup* che porta il parallelismo a JSetL.

Per ogni test si considerano i tempi (espressi in millisecondi) che si ottengono utilizzando il solver sequenziale, il solver parallelo senza ThreadPool (TP), e il solver parallelo con ThreadPool.

5.1 Test brevi

Ogni test è stato eseguito 5 volte per poi fare una media dei tempi per avere risultati precisi.

5.1.1 Test02

Questo semplice test crea due `LSet a={1,2}` e `b={2,3}` e impone tramite un vincolo `eq` che siano uguali tra loro:

a={1,2} & b={2,3} & a=b

Il solver restituirà quindi `false`.

```
LSet a = LSet.empty().ins(1).ins(2);
LSet b = LSet.empty().ins(2).ins(3);
```

```
solver.add(a.eq(b));
return solver.check();
```

Test02 (mS)	Solver Sequenziale	Solver Parallelo	Solver Parallelo con TP
n.1	14.5559	16.9717	17.3602
n.2	14.2004	15.8925	16.3113
n.3	13.778	16.4401	17.8384
n.4	13.3919	16.5425	16.4116
n.5	14.1374	15.8795	16.1381
Media	14.013	16.345	16.812

Il solver sequenziale impiega in media 2.33 mS in meno di entrambi i solver paralleli, il solver parallelo con TP impiega un ulteriore 0.46 mS.

Come previsto i solver paralleli impiegano un tempo maggiore di quello sequenziale. Il test è troppo breve perché il parallelismo porti dei vantaggi all'esecuzione, tuttavia non porta neanche grandi svantaggi.

5.1.2 Test05

In questo test si impone che due `LSet a` e `b`, che contengono ognuno una `LVar` non inizializzata, siano uguali, imponendo però che le due variabili siano diverse:

a={x} & b={y} & a=b & x neq y

Anche in questo caso il solver restituirà `false`.

```
LVar x = new LVar();
LVar y = new LVar();
LSet a = LSet.empty().ins(x);
LSet b = LSet.empty().ins(y);
solver.add(a.eq(b).and(x.neq(y)));
return solver.check();
```

Test05 (mS)	Solver Sequenziale	Solver Parallelo	Solver Parallelo con TP
n.1	14.7175	17.9341	17.9528
n.2	14.6725	18.2004	18.051
n.3	14.6476	17.6716	18.1105
n.4	16.3503	17.7019	18.2009
n.5	14.6098	15.8795	17.994
Media	14.999	17.861	18.062

Il solver sequenziale impiega in media 2.86 mS in meno di entrambi i solver paralleli, il solver parallelo con TP impiega un ulteriore 0.2 mS.

5.1.3 Test16

In questo test viene imposto che due `LSet a{1,2}` e `b{1,i}`, dove i è una IntLVar non inizializzata, siano uguali:

a={1,2} & b={1,i} & a=b

Ne consegue che all'`IntLVar` contenuta in uno dei due sia assegnato un valore intero. Il solver restituirà `2` che corrisponde al valore previsto.

```
IntLVar i = new IntLVar("i");
LSet a = LSet.empty().ins(2).ins(1);
LSet b = LSet.empty().ins(i).ins(1);
solver.add(a.eq(b));
solver.check();
return i.getValue();
```

Test16 (mS)	Solver Sequenziale	Solver Parallelo	Solver Parallelo con TP
n.1	16.5582	20.4909	21.5685
n.2	16.7741	20.731	21.0666
n.3	16.5673	20.7526	21.6402
n.4	16.6995	20.8124	20.8341
n.5	16.5255	20.8252	21.4176
Media	16.625	20.722	21.305

Il solver sequenziale impiega in media 4.1 mS in meno di entrambi i solver paralleli, il solver parallelo con TP impiega un ulteriore 0.58 mS.

5.1.4 Test24

Questo test è la risoluzione in JSetL del classico problema di criptoaritmetica:

$$SEND + MORE = MONEY \tag{5.1}$$

Dopo aver creato le 8 variabili logiche intere, il test impone che siano tutte diverse tramite il metodo `allDifferent(LSet l)` e poi la condizione di soddisfare l'equazione 5.1.
Il risultato sarà `true` e più nello specifico i valori delle variabili restituiti dal programma sono:
`s = 2 e = 8 n = 1 d = 7`
`m = 0 o = 3 r = 6 y = 5`

```
boolean b;
MultiInterval i= new MultiInterval(0,9);
IntLVar s=new IntLVar("s",i); IntLVar e=new IntLVar("e",i);
IntLVar n=new IntLVar("n",i); IntLVar d=new IntLVar("d",i);
IntLVar m=new IntLVar("m",i); IntLVar o=new IntLVar("o",i);
IntLVar r=new IntLVar("r",i); IntLVar y=new IntLVar("y",i);
IntLVar parz=new IntLVar();


solver.add(Constraint.allDifferent(s,e,n,d,m,o,r,y));
solver.add(parz.eq(m.mul(10000).sum(o.mul(1000).sum(n.mul(100).sum(e.mul
    (10).sum(y))))));
solver.add(parz.eq(s.mul(1000).sum(e.mul(100).sum(n.mul(10).sum(d).sum(m.
    mul(1000).sum(o.mul(100).sum(r.mul(10).sum(e))))))));

solver.add(s.label()); solver.add(e.label());
solver.add(n.label()); solver.add(d.label());
solver.add(m.label()); solver.add(o.label());
solver.add(r.label()); solver.add(y.label());

b=solver.check();

s.output(); e.output(); n.output(); d.output();
m.output(); o.output(); r.output(); y.output();
return b;
```

Test24 (mS)	Solver Sequenziale	Solver Parallelo	Solver Parallelo con TP
n.1	27.1627	30.1442	31.6436
n.2	27.3335	30.9924	30.8998
n.3	27.5522	30.7961	30.4145
n.4	27.5949	30.5749	30.7337
n.5	27.3779	30.4459	30.9651
Media	27.404	30.59	30.931

Il solver sequenziale impiega in media 3.18 mS in meno di entrambi i solver paralleli, il solver parallelo con TP impiega un ulteriore 0.34 mS.

5.1.5 Test26

Il test26 risolve il seguente sistema di equazioni a 3 variabili x, y e z:

$$\begin{cases} x + y + z = 6 \\ x - y - z = 2 \\ x - y + z = -4 \end{cases}$$

Le 3 `IntLVar` hanno come dominio l'intervallo [-100,100]. Si impongono le 3 equazioni singole tramite 3 vincoli unite da *and* logico seguite da `label()` delle variabili.
Il programma restituirà `true` e stamperà i valori delle variabili seguenti:
`x = 4  y = 5  z = -4`

```
boolean b;
IntLVar x=new IntLVar("x",-100,100);
IntLVar y=new IntLVar("y",-100,100);
IntLVar z=new IntLVar("z",-100,100);

solver.add(x.sum(y).sum(z).eq(6));
solver.add(x.sub(y).sub(z).eq(2));
solver.add(x.sub(y).sum(z).eq(-4));

solver.add(x.label());
solver.add(y.label());
solver.add(z.label());

b=solver.check();
```

```
x.output();y.output();z.output();
return b;
```

Test26 (mS)	Solver Sequenziale	Solver Parallelo	Solver Parallelo con TP
n.1	44.6224	47.1475	48.1377
n.2	46.0655	48.3551	48.3118
n.3	45.4314	48.0604	48.1563
n.4	45.6175	48.354	48.4264
n.5	46.0683	48.3935	48.1686
Media	45.561	48.062	48.240

Il solver sequenziale impiega in media 2.5 mS in meno di entrambi i solver paralleli, il solver parallelo con TP impiega un ulteriore 0.17 mS.

Con l'aumentare del tempo totale la differenza tra i due solver si assottiglia, mostrando un primo segno di miglioramento.

5.2 Test medio-lunghi

Per eseguire i test seguenti, dato il gran numero di core richiesto, abbiamo utilizzato un Intel Xeon Phi 7250 1.4GHz 68c fornito dal centro HPC dell'Università di Parma [6]. Questo processore è dotato di 68 core fisici e 272 core virtuali, grazie alla tecnologia dell'hyperthreading che permette di eseguire fino a 4 thread per ogni singolo core [8].

Ad ogni test segue una tabella in cui sono specificati i tempi di esecuzione modificando il numero di core richiesti alla macchina nel momento di esecuzione del test. La prima riga corrisponde alla macchina precedentemente utilizzata nelle altre prove, ovvero un Intel i7-9700k [7].

5.2.1 Test14

Questo test aggiunge ad un `LSet` un numero di costanti alto a piacere, in questo caso 400; poi impone che due variabili appartengano all'insieme e che siano uguali a 99999, che non è presente nell'insieme:

a = {0,1,...,400} & x in a & y in a & x=99999 & y=99999

In questo modo si crea un gran numero di *choice-point* per poi fallire.

```
LSet a = LSet.empty();
for(int i = 0; i < 400; ++i)
   a = a.ins(i);
LVar x = new LVar();
LVar y = new LVar();
solver.add(x.in(a));
solver.add(y.in(a));
solver.add(x.eq(99999).and(y.eq(99999)));

return solver.check();
```

Test14 (mS)	Solver Sequenziale	Solver Parallelo	Solver Parallelo con TP
8thr-4.6GHz	1 255.075	665.783	549.352
8thr-1.4GHz	12 741.499	13 219.293	11 481.188
16thr-1.4GHz	12 381.645	10 794.146	10 649.916
32thr-1.4GHz	12 598.786	11 132.244	9 787.472
64thr-1.4GHz	11 480.006	11 941.114	5 423.113
128thr-1.4GHz	11 959.919	15 650.621	3 799.647

Il tempo del solver parallelo con TP scende progressivamente con l'aumentare dei core. Invece l'altro solver parallelo mostra un comportamento anomalo impiegando tempi simili a quelli del solver sequenziale, con un peggioramento nel momento dell'utilizzo di 128 core. I tempi del solver sequenziale, come previsto, sono sempre molto simili tra loro in quanto per progettazione del programma basato sulla sequenzialità delle istruzioni, non benificia dall'aumento dei core.

5.2.2 Test19

Per i dettagli di questo test si rimanda al paragrafo 3.1.

Test19 (mS)	Solver Sequenziale	Solver Parallelo	Solver Parallelo con TP
8thr-4.6GHz	10 779.37	6 987.62	2 566.75
8thr-1.4GHz	147 248.28	96 160.80	58 040.55
16thr-1.4GHz	150 809.45	70 976.61	31 603.69
32thr-1.4GHz	143 630.21	90 507.48	17 720.33
64thr-1.4GHz	149 305.21	89 955.734	12 823.03
128thr-1.4GHz	146 793.38	113 142.88	13 744.01

Anche in questo caso i tempi del solver parallelo con TP diminuiscono con l'aumentare dei core, però il guadagno nel passaggio da 64 a 128 è minimo. Il solver paralleo originale invece non mostra vantaggi aumentando il numero di core.

5.2.3 Test20

Per i dettagli di questo test si rimanda al paragrafo 3.2.

Test20 (mS)	Solver Sequenziale	Solver Parallelo	Solver Parallelo con TP
8thr-4.6GHz	32 284.414	15 557.52	6 288.1943
8thr-1.4GHz	310 080.62	214 631.06	56 777.22
16thr-1.4GHz	300 070.38	613 428.06	101 851.34
32thr-1.4GHz	340 054.51	153 510.66	47 171.734
64thr-1.4GHz	351 614.53	191 506.62	70 915.99
128thr-1.4GHz	332 687.56	502 807.81	58 835.13

In questo test notiamo che anche TP ha tempi incostanti e non vede guadagni nel passaggio da 8 thread a 128 thread, mantenendo tuttavia un tempo nettamente inferiore a quello del solver sequenziale, a differenza del solver parallelo originale che con 128 core impiega un tempo maggiore.

5.2.4 Test22

Questo test, dopo aver creato e definito un `LSet` N5 contenente 6 variabili, chiede al solver di definire altri 4 `LSet` tramite i vincoli di unione tra insiemi che portano alla creazione proprio di N5. Si impone anche però che una variabile, X5, non sia presente in nessun insieme:

N5={X1,X2,X3,X4,X5,X6,X7/M} & un(N1,N2,N3) & un(N3,N4,N5) & X5 nin N1 & X5 nin N2 & X5 nin N4.

In questo modo si causa il fallimento del programma e quindi un tempo di computazione maggiore, essendo necessario esplorare tutte le possibili alternative prima di concludere con un fallimento.

```
LVar X1 = new LVar("X1"), X2 = new LVar("X2"), X3 = new LVar("X3");
LVar X4 = new LVar("X4"), X5 = new LVar("X5"), X6 = new LVar("X6");
LSet N1=new LSet(); LSet N2=new LSet();
LSet N3=new LSet(); LSet N4=new LSet();
LSet N5=new LSet("M").ins(X1,X2,X3,X4,X5,X6);

solver.add(N1.union(N2,N3));
solver.add(N3.union(N4,N5));
solver.add(X5.nin(N1));solver.add(X5.nin(N2));solver.add(X5.nin(N4));

solver.check();
```

Test22 (mS)	Solver Sequenziale	Solver Parallelo	Solver Parallelo con TP
8thr-4.6GHz	72 479.01	25 448.684	14 089.657
8thr-1.4GHz	733 270.11	437 495.66	324 017.03
16thr-1.4GHz	743 806.41	323 400.12	182 349.02
32thr-1.4GHz	751 595.75	390 768.44	108 711.36
64thr-1.4GHz	748 102.06	471 965.94	72 932.21
128thr-1.4GHz	741 429.51	601 403.94	58 835.13

Il solver parallelo originale non mostra miglioramenti con l'aumentare dei core, anzi alcuni peggioramenti. Invece il solver parallelo con TP mostra miglioramenti costanti.

5.2.5 Test29

Il test29 è un problema di colorazione di grafi che presenta 6 regioni e 5 colori, codificati tramite `LVar` e `LSet`. Si utilizza anche elementi appartenenti alla classe `Ris` di JSetL, insiemi intensionali ristretti ovvero insiemi definiti dando proprietà che gli elementi devono soddisfare per farne parte.

```
LVar r1 = new LVar("r1");
LVar r2 = new LVar("r2");
```

```
LVar r3 = new LVar("r3");
LVar r4 = new LVar("r4");
LVar r5 = new LVar("r5");
LVar r6 = new LVar("r6");
LVar[] regionsArray = {r1,r2,r3,r4,r5,r6};
LSet regions = LSet.empty().insAll(regionsArray);

LSet[] mapArray =
{LSet.empty().ins(r1).ins(r2).ins(r3),LSet.empty().ins(r2).ins(r3).ins(r4)
    ,
LSet.empty().ins(r3).ins(r4).ins(r5),
LSet.empty().ins(r6).ins(r4).ins(r5),
LSet.empty().ins(r3).ins(r4).ins(r2),
LSet.empty().ins(r3).ins(r5).ins(r6),
LSet.empty().ins(r3).ins(r1).ins(r2),
LSet.empty().ins(r6).ins(r2).ins(r3),
LSet.empty().ins(r2).ins(r5).ins(r6),
LSet.empty().ins(r3).ins(r5).ins(r6)};
LSet map = LSet.empty().insAll(mapArray);

String[] colorsArray = {"Red","Blue", "Green", "Orange", "Pink"};
LSet colors = LSet.empty().insAll(colorsArray);

LSet x = new LSet("x");
Constraint f = x.size(3);
Ris ris = new Ris(x, map, f);

solver.add(regions.eq(colors));
solver.add(ris.eq(map));

return solver.check();
```

Test29 (mS)	Solver Sequenziale	Solver Parallelo	Solver Parallelo con TP
8thr-4.6GHz	641.796	342.403	333.322
8thr-1.4GHz	9 887.187	4 787.414	4 457.215
16thr-1.4GHz	8 575.217	2 738.638	3 419.224
32thr-1.4GHz	7 061.531	2 477.657	1 573.525
64thr-1.4GHz	7 205.743	1 197.157	1 110.163
128thr-1.4GHz	8 578.378	1 551.594	1 078.005

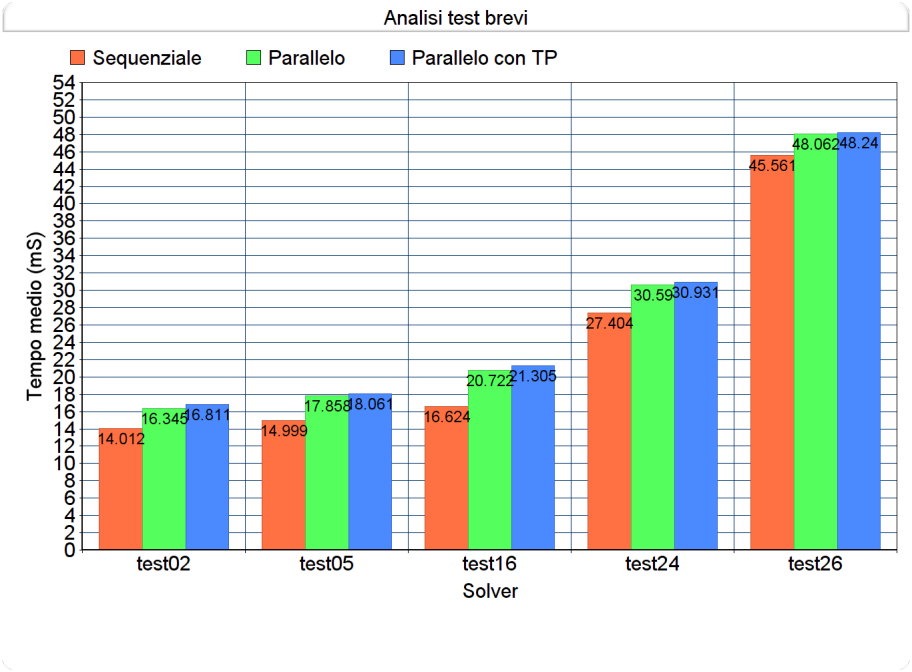
Entrambi i solver paralleli mostrano miglioramenti con l'aumentare dei core. Con 128 core tuttavia il solver parallelo originale vede un peggioramento rispetto ai 64 core, il solver paralleo con tp vede un leggero miglioramento ma non in linea con gli altri passaggi tra numeri di core.

5.3 Analisi risultati

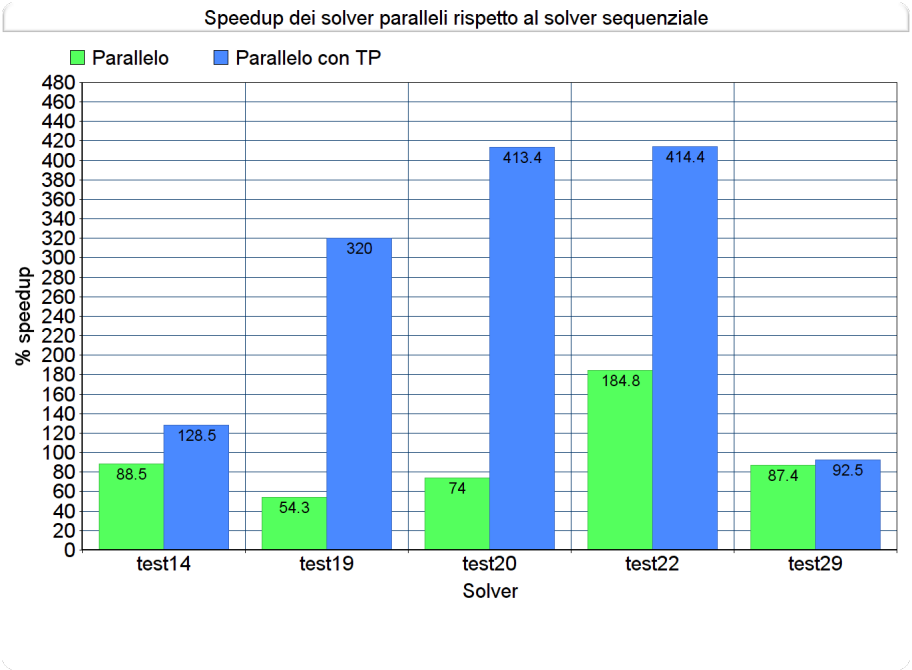
Riguardo ai test brevi, come si poteva prevedere, il solver sequenziale è il solver che impiega tempi minori. I solver paralleli impiegano costantemente un tempo di 3-4 mS in più, tempo dato dalla creazione degli oggetti extra per gestire il parallelismo e dalla copia dello store in ogni istanza di `ParallelSolverSlave`.

Nella prima selezione di test il nuovo `ParallelSolver` impiega un tempo leggermente superiore, dagli 0.2 ai 0.7 mS, un tempo quindi trascurabile in più del `ParallelSolver` originale, dato dalla creazione del gestore dei thread.

Segue un grafico che riassume e confronta i tempi nei vari test utilizzando i 3 solver.



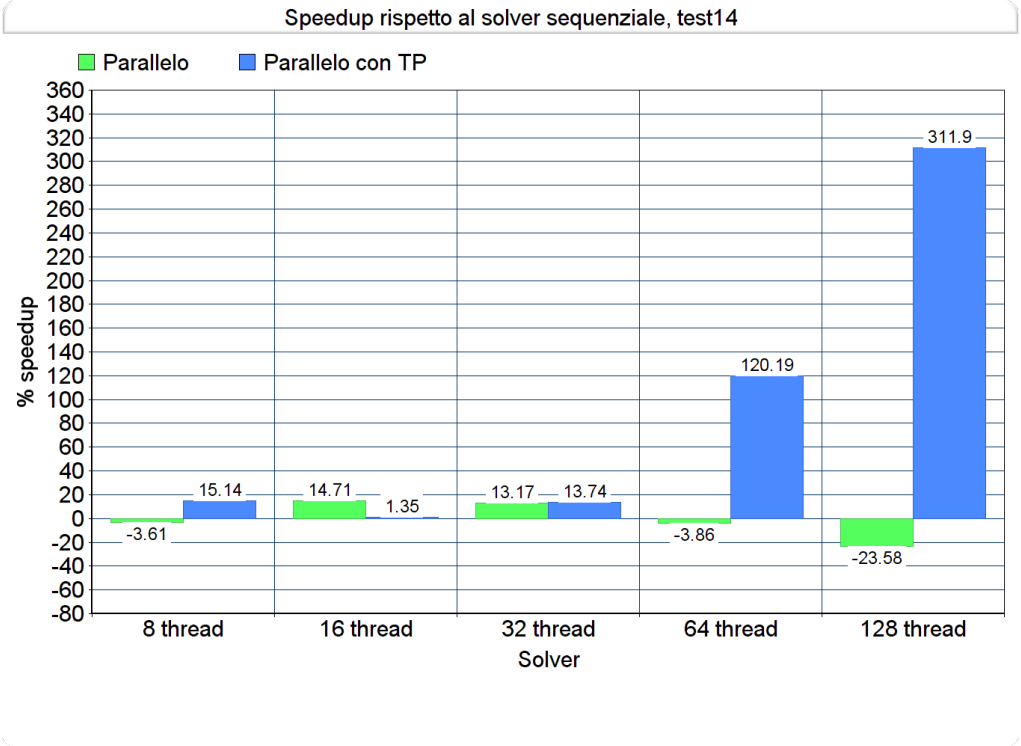
I vantaggi dei solver paralleli sono evidenti però guardando la seconda categoria di test, quelli che richiedono un tempo di calcolo maggiore. L'utilizzo dei solver paralleli migliora le prestazioni di almeno il 50% per il vecchio `ParallelSolver` e di almeno il 90% per il nuovo, fino ad arrivare ad oltre il 400% come si può vedere dal seguente grafico:

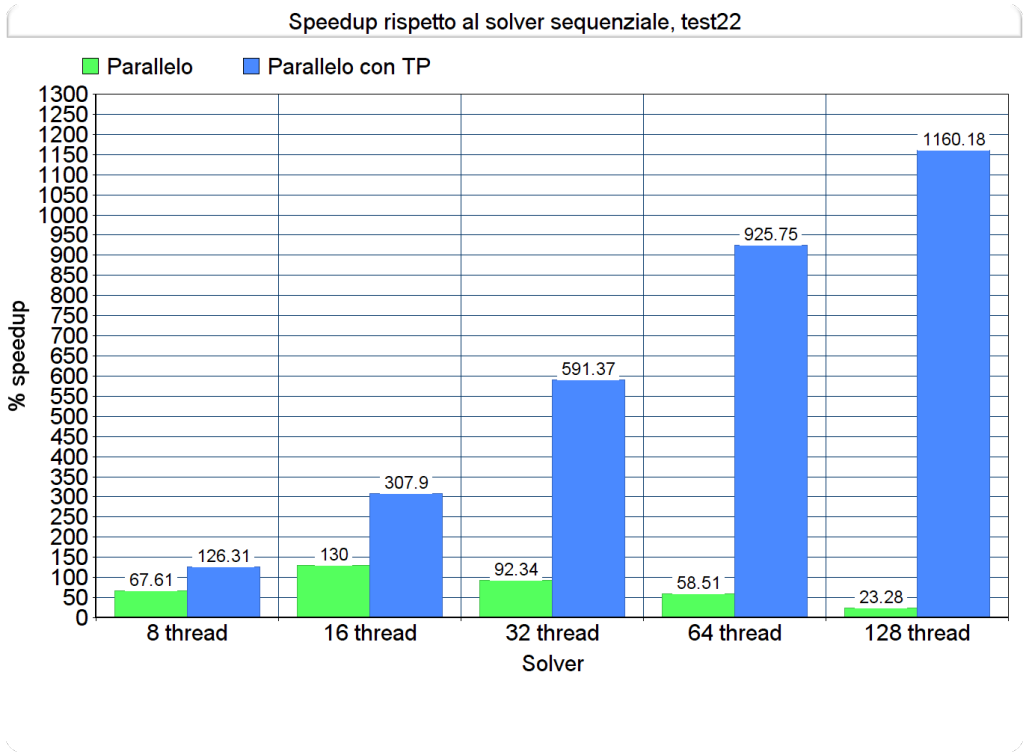


I test 14 e 29 hanno tempi intorno al secondo, quindi seppur ci sia un netto miglioramento non è comparabile con quello degli altri test più lunghi, che impiegano più di 10 secondi ognuno. In questi ultimi, come anche nel caso di test ancora più lunghi (per esempio test19 utilizzando 6 variabili logiche al posto di 5) `ParallelSolver` con gestore di thread (TP nei grafici) impiega 1/5 del tempo del solver sequenziale, che seppur lontano dall'1/8 teorico si può considerare un ottimo miglioramento.

Il nuovo `ParallelSolver` infatti riesce a minimizzare l'overhead, dato da oltre 500 milioni di `ChoicePoint` incontrati. `ParallelSolver` originale, appesantito dalla mole di thread creati e distrutti, porta uno *speedup* meno consistente, anche nel caso di esecuzioni più impegnative.

I risultati sono ancora più evidenti aumentando il numero di core della macchina, come si può vedere nei test eseguiti nel cluster del centro HPC di Parma. Abbiamo scelto questo specifico processore in modo da eseguire tutti i test sulla stessa macchina, dato l'alto numero di core fisici.
Questa tabella mostra lo *speedup* dei solver paralleli aumentando il numero di thread nei 2 test più interessanti, ovvero test14 e test22. Esso è calcolato in relazione ai tempi impiegati dal solver sequenziale sulla stessa macchina (tempo che non cambia modificando il numero di thread) che sono rispettivamente 21.5 secondi e 733 secondi.





Si può notare un comportamento strano del solver parallelo senza TP che nel passaggio da 32 a 64 thread vede un peggioramento delle sue prestazioni, sempre dato dall'eccessivo overhead. Nel caso del test22 l'esecuzione utilizzando 128 thread impiega addirittura un tempo maggiore del solver sequenziale.

Il solver parallelo che utilizza `ThreadPool` invece vede miglioramenti costanti con l'aumento dei thread, allontanandosi però dallo *speedup* teorico. Il massimo aumento di prestazioni si ha utilizzando 128 thread nell'esecuzione del test22, in cui il solver impiega 1/11 del tempo di quello sequenziale.

Capitolo 6

Conclusioni

In questo lavoro di tesi è stato affrontato il problema di analizzare e migliorare il solver parallelo per la risoluzione di vincoli per la libreria Java JSetL.

Per prima cosa si è studiato quali sono i punti deboli del sistema tramite un ciclo di test, in cui sono stati anche trovati e risolti diversi bug. Si è scoperto che, dato l'elevato numero di *choice-point*, il solver creava quantità importanti di thread che portavano un costo computazionale di gestione molto gravoso. É stato quindi realizzato un gestore di thread, chiamato `ThreadPool` da affiancare al solver parallelo, a cui viene delegata la creazione e distruzione dei thread. Di conseguenza l'esecuzione delle varie istanze di `ParallelSolverSlave` avverrà nei thread creati dal gestore, e non sarà più creato un thread apposito per ogni istanza. `ParallelSolver` mantiene però il compito della gestione di tutte le comunicazioni tra il programma principale e i diversi slave.

Successivamente sono stati testati a fondo i vari solver e calcolati i tempi di esecuzione in varie casistiche e sono stati comparati. In particolare è stato valutato lo *speedup* che porta un solver parallelo all'esecuzione di JSetL, soprattutto aumentando il numero di core concessi al programma. Abbiamo quindi notato un importante miglioramento in relazione ai test che impiegano un tempo minimo di 1 secondo di esecuzione. Lo *speedup* più significativo registrato nel corso dei test è stato del 415% grazie al nuovo gestore dei thread.

In conclusione per problemi semplici e brevi il solver sequenziale rimane

l'opzione migliore, ma se si ha un problema computazionalmente impegnativo `ParallelSolver`, grazie al parallelismo, porta significativi miglioramenti. Questo permette di risparmiare molto tempo e risorse, mantenendo la stessa affidabilità.

Ringraziamenti

Un sentito grazie a tutte le persone che mi hanno permesso di arrivare fin qui e di portare a termine questo lavoro di tesi.

Grazie al mio relatore Gianfranco Rossi, sempre presente, puntuale e disponibile. Grazie al percorso intrapreso insieme ho sviluppato maggiormente la mia capacità di analisi e di problem solving.

Non posso non menzionare i miei genitori, Cristina e Francesco, che da sempre mi sostengono nella realizzazione dei miei progetti. Non finirò mai di ringraziarvi per avermi permesso di arrivare fin qui.

Grazie ai miei amici per essere stati sempre presenti anche durante questa ultima fase del mio percorso di studi. Grazie per aver ascoltato i miei sfoghi, grazie per tutti i momenti di spensieratezza.

Riferimenti bibliografici

[1] Giulia Magnani. *Un solver parallelo per vincoli insiemistici per la libreria Java JSetL*. PhD thesis, Università di Parma, 2019.

[2] Lorenzo Baesso. *Sistemi multiprocessore e multicore*. PhD thesis, Università di Padova, 2011.

[3] Andrea Fois Gianfranco Rossi, Roberto Amadini. JSetL User's Manual.

[4] JSetL Home Page. `www.clpset.unipr.it/jsetl/`.

[5] Java documentation. `https://docs.oracle.com/en/java/javase/14/`.

[6] Wiki del Calcolo Scientifico dell'Università di Parma. `https://www.hpc.unipr.it/dokuwiki/doku.php`.

[7] Specifiche Processore Intel Core i7-9700K. `https://ark.intel.com/content/www/it/it/ark/products/186604/intel-core-i7-9700k-processor-12m-cache-up-to-4-90-ghz.html`.

[8] Specifiche Processore Intel Xeon Phi Processor 7250. `https://ark.intel.com/content/www/us/en/ark/products/94035/intel-xeon-phi-processor-7250-16gb-1-40-ghz-68-core.html`.

Appendice A

ThreadPool

File: ThreadPool.java

```
import java.util.LinkedList;
import java.util.List;

public class ThreadPool {
   private Thread[] pool;

   private List<Runnable> queue;

   int working;
   int poolSize;

   public ThreadPool(int size) {

      queue = new LinkedList<>();

      pool = new Thread[size];

      poolSize=size;
      working=0;

   }
   public synchronized void start() {
         pool[working] = new Worker();
         pool[working].start();
         working++;
   }
```

```
public synchronized void stop() {
   add(new Stop());
}

public synchronized void add(Runnable runnable) {
   if (working<poolSize) {
      start();
   }
   queue.add(runnable);

   notifyAll();
}

private class Worker extends Thread {
   @Override
   public void run() {
      for(;;) {
         try {
            Runnable runnable;

            Object semaphore = ThreadPool.this;

            synchronized(semaphore) {
               while(queue.isEmpty())
                  semaphore.wait();

               runnable = queue.get(0);

               if(runnable instanceof Stop)
                  return;

               queue.remove(0);
            }

            runnable.run();
         } catch(InterruptedException e) {
            // Blank
         } catch(Throwable t) {
            t.printStackTrace();
         }
```

```
        }
      }
   }

   private class Stop implements Runnable {
      @Override
      public void run() {
         // Blank
      }
   }
}
```

Appendice B

ParallelSolver

File: ParallelSolver.java

```
/**
 * jsetl A Java library that combines the object-oriented
 * programming paradigm with valuable concepts of CLP languages
 *
 * Copyright (C) 2000-2012 jsetl Team.
 *
 * jsetl is distributed under the terms of the GNU Lesser General
 * Public License.
 */

/**
 * Solver.java
 * @version 2.3
 *
 */

package jsetl;


import jsetl.exception.Failure;

import java.util.HashMap;
import java.util.LinkedList;
import java.util.Map;
import java.util.Queue;
import java.util.Set;
```

```
import java.util.List;

/**
 * Objects of this class are solvers for constraints conjunctions.
    Constraints can be added to the solver (which stores them).
 * Solvers are able to determine if solutions for the given constraint
    conjunction exist and are able to find them all.
 * In order to solve non-deterministic constraints solvers use extensively
     the backtracking method.
 */

public class ParallelSolverP extends Solver{

   protected volatile int counter;
   protected volatile boolean solutionFound;
   protected final Object mutex;
   protected static final int CORES = Runtime.getRuntime().
       availableProcessors();
   protected Queue<ParallelSolverSlaveP> solutionSolvers = new LinkedList
       <>();
   protected LinkedList<ParallelSolverSlaveP> stoppedSolvers = new
       LinkedList<>();
   protected Map<LObject, LObject> equs;
   protected ThreadPool pool;

   public ParallelSolverP() {

      mutex = new Object();
      counter = 0;
      solutionFound = false;
      pool=new ThreadPool(CORES);
//    System.out.println(CORES);
   }



   @Override
   public boolean check() {
      Constraint store = getConstraint();
```

```
Object w = new Object();

synchronized(mutex) {
   ++counter;
}


synchronized (mutex){
   synchronized (w) {
      try {
         pool.add(()->{

               Constraint copy;
               Map<Object, Object> clonesMapping;
               synchronized (w) {
                  DeepCloner deepCloner = new DeepCloner();
                  copy = deepCloner.visit(store);
                  clonesMapping = deepCloner.getClonesMap();
                  w.notify();

               }
               ParallelSolverSlaveP ParallelSolverP = new
                   ParallelSolverSlaveP(this, clonesMapping, pool);
               ParallelSolverP.add(copy);

               ParallelSolverP.check();
            });
         w.wait();
      } catch (InterruptedException e) {
         e.printStackTrace();
      }
   }

   if(solutionSolvers.size() == 0){
      if(counter <= 0)
         return false;
      else {
         try {
            mutex.wait();
         } catch (InterruptedException e) {
            e.printStackTrace();
```

```
            }
            if(solutionSolvers.size() == 0)
               return false;
            else{
               pollSolution();
               solutionFound = true;
               return true;
            }
         }
      }
      else{
         pollSolution();
         solutionFound = true;
         return true;
      }
   }
}

@Override
public boolean nextSolution() {
   equs.forEach((key,value) -> {
      key.makeVariable();
      key.equ = value;
   });
   synchronized (mutex){
      if(solutionSolvers.size() == 0){
         if(counter <= 0)
            return false;
         else {
            solutionFound = false;
            mutex.notifyAll();
            for (ParallelSolverSlaveP stoppedSolver : stoppedSolvers) {
               synchronized (stoppedSolver.stop){
                  stoppedSolver.stop.notifyAll();
               }
            }
            stoppedSolvers.clear();
         }
         try {
            mutex.wait();
         } catch (InterruptedException e) {
```

```
            e.printStackTrace();
         }
         if(solutionSolvers.size() == 0)
            return false;
         else{
            pollSolution();
            return true;
         }
      }
      else{
         pollSolution();
         return true;
      }
   }

}

protected void pollSolution(){
   ParallelSolverSlaveP solutionSolver = solutionSolvers.poll();
   stoppedSolvers.add(solutionSolver);
   ParallelSolverSlaveP father = solutionSolver;
   MapJoiner mapJoiner = new MapJoiner();
   while(father != null) {
      mapJoiner.add(father.getClonesMapping());
      father=father.getFather();
   }
   mapJoiner.reverse();
   Map<Object,Object> joined = mapJoiner.getComposition();
   equs = new HashMap<>();
   joined.forEach((key,value) -> {
      equs.put((LObject)key, ((LObject)key).equ);
   });
   joined.forEach((key,value) -> {

      if(key instanceof LVar) {
         LVar lVar = (LVar)key;
         if(!(value instanceof LVar))
            lVar.setValue(value);
         else
            lVar.equ = (LVar)value;
      }
```

```
        if(key instanceof LSet) {
           LSet lSet = (LSet)key;
           if(!(value instanceof LSet))
              lSet.setValue((Set<?>)value);
           else
              lSet.equ = (LSet)value;
        }
        if(key instanceof LList) {
           LList lList = (LList)key;
           if(!(value instanceof LList))
              lList.setValue((List<?>)value);
           else
              lList.equ = (LList)value;
        }
     });
  }

  @Override
  public void solve() throws Failure{
     if(!this.check())
        throw new Failure();
  }

  public void stop() {
     pool.stop();

  }
}
```

Appendice C

ParallelSolverSlave

File: ParallelSolverSlave.java

```
package jsetl;

import jsetl.exception.Fail;

import java.util.Map;


public class ParallelSolverSlaveP extends Solver {

   protected final ParallelSolverP master;
   protected static int CORES = ParallelSolverP.CORES;
   protected Object stop = new Object();
   protected long tCr;

   protected Map<Object, Object> clonesMapping;
   protected ParallelSolverSlaveP father;
   protected ThreadPool pool;

   protected ParallelSolverSlaveP(ParallelSolverP master, Map<Object,
       Object> clonesMapping,ThreadPool pool) {

      this(master, clonesMapping, null,pool);

   }
```

```
public Map<Object,Object> getClonesMapping() {
   return clonesMapping;
}
protected ParallelSolverSlaveP(ParallelSolverP master, Map<Object,
    Object> clonesMapping, ParallelSolverSlaveP father,ThreadPool pool)
     {

   this.master = master;
   this.clonesMapping = clonesMapping;
   this.father = father;
   this.pool = pool;

}

protected ParallelSolverSlaveP getFather() {
   return father;
}

@Override
public void addChoicePoint(AConstraint s) {

   Object syn = new Object();
   synchronized (syn) {
      Global.nChoicePointPSP++;
      }

   if(master.counter >= CORES) {
      super.addChoicePoint(s);
      return;
   }

   s.alternative++;
   Constraint store = getConstraint();
   Object w = new Object();
   synchronized(master.mutex) {
      ++master.counter;
   }

   synchronized (w) {
      try {
         pool.add(()->{
```

```
            Constraint copy;
            Map<Object, Object> clonesMapping;

            synchronized (w) {

               DeepCloner deepCloner = new DeepCloner();
               copy = deepCloner.visit(store);
               clonesMapping = deepCloner.getClonesMap();
               w.notify();

               }

            ParallelSolverSlaveP ParallelSolverP = new
                ParallelSolverSlaveP(master,clonesMapping,this,pool);
            ParallelSolverP.add(copy);

            ParallelSolverP.check();
            });

         w.wait();
      } catch (InterruptedException e) {
         e.printStackTrace();
      }
      --s.alternative;
   }
}

@Override
public boolean check() {
   boolean result = super.check();
   while(result){
      synchronized (stop){
         synchronized (master.mutex) {
            //System.out.println("SLAVE ADDING SOLUTION");
            master.solutionSolvers.add(this);
            master.solutionFound = true;
            master.mutex.notifyAll();
         }

         try {
            stop.wait();
```

```
            } catch (InterruptedException e) {
               e.printStackTrace();
            }
            //System.out.println("slave awaken");
         }
         result = nextSolution();
      }

      synchronized (master.mutex){
         --master.counter;
         if(master.counter <= 0) {
            master.mutex.notifyAll();
         }
      }

      return result;
   }

   @Override
   protected void risAConstraint(AConstraint a)throws Fail {
      super.risAConstraint(a);
      if(master.solutionFound) {
         synchronized(master.mutex) {
            try {
               master.mutex.wait();
            } catch (InterruptedException e) {
               // TODO Auto-generated catch block
               e.printStackTrace();
            }
         }
      }
   }
}
```