UNIVERSITÀ
DI PARMA

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

Un solver parallelo per vincoli insiemistici per la libreria Java JSetL

A parallel solver for set constraints for the Java library JSetL

Relatore:
Chiar.mo Prof. Gianfranco Rossi

Tesi di Laurea di:
Giulia Magnani

ANNO ACCADEMICO 2018-2019

Alla mia famiglia, al mio ragazzo e ai miei amici

"Computer Science is no more about computers than astronomy is about telescopes."
Edsger W. Dijkstra

Indice

Introduzione **1**

1 JSetL **3**

1.1 Strutture dati 3

1.1.1 Variabili logiche 3

1.1.2 Liste logiche 4

1.1.3 Insiemi logici 5

1.2 Vincoli 5

1.2.1 Vincoli principali 5

1.3 Risoluzione di vincoli 6

2 Thread in Java **8**

2.1 Thread 8

2.2 Thread in Java 8

2.2.1 La classe `Thread` 8

2.2.2 Stati dei thread 9

2.3 Mutua esclusione 10

3 Architettura del sistema **11**

3.1 Risoluzione di vincoli ed *or-parallelismo* 11

3.2 Architettura master-slave 13

3.3 Thread master 14

3.4 Thread slave 15

3.5 Gestione delle race condition 16

3.6 Comunicazioni fra thread 16

4 Implementazione **18**

4.1 Classe ParallelSolver 18

4.1.1 Campi 18

4.1.2 Metodi 19

4.2 Classe ParallelSolverSlave 23

4.2.1 Campi 23

4.2.2 Metodi 23

4.3 Classe MapJoiner 26

4.3.1 Campi 26

4.3.2 Metodi 26

5 Esempi e casi d'uso **28**

5.1 Test generici 28

5.2 Test per problemi combinatori con valutazione delle prestazioni 30

5.3 Test per la valutazione delle prestazioni in caso di vincoli con molte alternative possibili 34

6 Conclusioni e sviluppi futuri **36**

Ringraziamenti **38**

Riferimenti bibliografici **39**

A Appendice **40**

Introduzione

Al giorno d'oggi, la programmazione in parallelo viene ampiamente sfruttata nell'ambito della creazione di sistemi software, in quanto l'esecuzione simultanea di codice su più core della stessa macchina può comportare notevoli vantaggi in termini di efficienza temporale (per approfondimenti, vedere [1]). La maggior parte dei dispositivi (computer desktop, computer portatili, tablet e persino smartphone) presenta ormai un'architettura *multicore*, che offre la possibilità di sviluppare software appositi e scrivere algoritmi che sfruttino al meglio la capacità di calcolo della macchina, svolgendo processi in contemporanea su processori diversi e di conseguenza riducendo il tempo di calcolo necessario per eseguire l'algoritmo.

Lo scopo di questo lavoro di tesi è di applicare il concetto di parallelizzazione al risolutore di vincoli (`Solver`) della libreria Java JSetL.
Questa libreria (sviluppata presso il Dipartimento di Matematica e Informatica dell'Università di Parma, per la quale tutti i dettagli sono reperibili in [2][3][4]) permette di utilizzare all'interno di un linguaggio object-oriented come Java i concetti e le strutture dati tipici della programmazione logica a vincoli, come nondeterminismo ([5]), variabili logiche, eccetera.

Questo lavoro si concentra proprio sulla risoluzione di vincoli, fornendo un approccio alternativo al backtracking di tipo sequenziale, sfruttando la programmazione parallela e in particolare i thread, che vengono gestiti e utilizzati tramite un'apposita libreria Java (ulteriori dettagli sono reperibili in [6]).

L'elaborato di tesi è strutturato nel seguente modo:

- **Capitolo 1.** Nel primo capitolo viene introdotta la libreria JSetL, con particolare attenzione alle strutture dati e alle funzionalità che verranno citate nel resto dell'elaborato.
- **Capitolo 2.** Nel secondo capitolo si illustrano le principali caratteristiche dei thread, soprattutto in riferimento alla classe `Thread` di Java.
- **Capitolo 3.** Nel terzo capitolo viene descritta l'architettura del sistema proposto, con una particolare attenzione nei confronti dei thread e della loro gestione e sincronizzazione.
- **Capitolo 4.** Nel quarto capitolo si analizza nel dettaglio il sistema realizzato, descrivendo l'implementazione delle varie classi nel linguaggio Java e i motivi che hanno portato a una tale implementazione.
- **Capitolo 5.** Nel quinto capitolo si analizzano alcuni test eseguiti sullo strumento software realizzato per verificarne il funzionamento e analizzarne le prestazioni, che possono fungere da esempi d'uso del sistema.
- **Capitolo 6.** Nel sesto capitolo verranno tratte alcune conclusioni sul lavoro svolto, analizzando alcuni dati sulle performance e ipotizzando possibili sviluppi futuri dello strumento realizzato.

Capitolo 1

JSetL

JSetL è una libreria Java che permette di sfruttare le caratteristiche della programmazione dichiarativa in un linguaggio di programmazione imperativo come Java. In particolare, questa libreria supporta strutture dati tipiche della programmazione logica e permette di definire e risolvere vincoli.

1.1 Strutture dati

JSetL supporta vari tipi di strutture dati. In questa sezione ne vedremo alcune che verranno utilizzate nei capitoli successivi.

1.1.1 Variabili logiche

Una variabile logica rappresenta una quantità sconosciuta. Di conseguenza non memorizza al suo interno alcun valore (come invece succede, tipicamente, con le variabili dei linguaggi di programmazione imperativi), ma si potrà creare un'associazione tra la variabile e un valore. Questa associazione potrà essere creata in fase di inizializzazione della variabile oppure successivamente, tramite la risoluzione di vincoli. In particolare, una variabile il cui dominio sia ristretto a un unico valore, sarà detta *bound*, altrimenti sarà detta *unbound*.

In JSetL, una variabile logica è un'istanza della classe `LVar` e ad essa potrà essere associato come valore un qualunque oggetto Java. La classe `IntLVar`,

sottoclasse di `LVar`, serve per dichiarare variabili che possono contenere solo valori interi e che potranno essere usate per esprimere vincoli numerici. Alle variabili logiche può essere assegnato un nome, che sarà utile per motivi di chiarezza in un eventuale output.

```
/*creare rispettivamente una variabile logica e una variabile logica
    intera non inizializzate e senza assegnare nomi*/
LVar x = new LVar();
IntLVar y = new IntLVar();
/*creare rispettivamente una variabile logica e una variabile logica
    intera non inizializzate assegnandovi i nomi X e Y*/
LVar x = new LVar(''X'');
IntLVar y = new IntLVar(``Y'');
/*creare rispettivamente una variabile logica e una variabile logica
    intera di nome X e Y, inizializzate con i valori 1 e 2*/
LVar x = new LVar(''X'', 1);
IntLVar y = new IntLVar(``Y'', 2);
```

1.1.2 Liste logiche

Le liste logiche sono liste di oggetti qualsiasi (inclusi oggetti di tipo LVar), possibilmente disomogenee in tipo e che possono contenere elementi ripetuti. Le liste sono definite come istanze della classe `LList`, che a sua volta estende la classe `LCollection`.

```
/*creare una lista non inizializzata*/
LList l = new LList();
/*creare una lista contenente gli elementi [1,2]*/
LList l = LList.empty().ins(2).ins(1);
```

È importante notare che l'ordine effettivo degli elementi della lista sarà contrario a quello in cui sono stati inseriti.

1.1.3 Insiemi logici

Gli insiemi logici sono insiemi di oggetti qualsiasi, possibilmente disomogenei in tipo e che non possono contenere duplicati. Gli insiemi sono definiti come istanze della classe `LSet`. La differenza fondamentale rispetto alle liste è che negli insiemi non importa l'ordine in cui gli elementi sono stati inseriti.

```
/*creare un insieme non inizializzato*/
LSet s = new LSet();
/*creare un insieme contenente gli elementi [1,2]*/
LSet s = LSet.empty().ins(2).ins(1);
```

1.2 Vincoli

JSetL offre la possibilità di esprimere vincoli logici sulle strutture dati viste in precedenza. Un vincolo (chiamato *constraint*) può essere in una delle seguenti forme:

- Constraint atomico. Questo tipo di vincolo è un'istanza della classe `AConstraint`.
- Constraint composto. È un'istanza della classe `Constraint`, che a sua volta contiene una lista di `AConstraint`.

1.2.1 Vincoli principali

I vincoli di JSetL più frequentemente usati sono i seguenti:

- Constraint di uguaglianza e disuguaglianza. Questi vincoli sono generati dai metodi `eq()` e `neq()` offerti dalle classi `LVar` e `LCollection`.
  s

  ```
  /*Date x,y,z variabili (istanze di LVar)*/
  Constraint equals = x.eq(y);
  Constraint notEquals = x.neq(z);
  ```

- Constraint di appartenenza e non appartenenza di una variabile a un insieme. Questi vincoli sono generati dai metodi `in()` e `nin()` offerti dalle classi `LVar` e dal metodo `contains()` della classe `LSet`.

```
/*Date x,y variabili (istanze di LVar) e A,B insiemi (istanze di
    LSet)*/
Constraint c1 = x.in(A);
Constraint c2 = x.nin(B);
Constraint c3 = A.contains(y);
```

- Unione insiemistica, resa possibile dal metodo `union()` della classe `LSet`.

```
/*Dati A,B,C insiemi (istanze di LSet)*/
A.union(B,C); //C = A ∪ B
```

- La disgiunzione di insiemi, resa possibile dal metodo `disj()` della classe `LSet`.

```
/*Dati A,B insiemi (istanze di LSet)*/
A.disj(B); //A || B (ovvero A ∩ B = {})
```

1.3 Risoluzione di vincoli

In JSetL esiste una classe chiamata `Solver` che si occupa della risoluzione di vincoli. I vincoli vengono aggiunti al solver tramite un apposito metodo e vengono salvati nel *constraint store*. Successivamente il solver si occupa di risolverli tramite la *riscrittura* dei vincoli in una forma semplificata (*solved form*): se tale riscrittura è possibile e il vincolo generato non contiene false, allora il vincolo iniziale è vero; altrimenti è falso.

Alcuni vincoli possono essere riscritti in più modi possibili; la scelta di quale riscrittura applicare è non-deterministica ed è implementata tramite

un meccanismo di backtracking. La possibilità di scegliere tra più possibli riscritture di un vincolo comporta la possibilità di ottenere *più possibili soluzioni* per un dato vincolo di input. Il solver, se richiesto, non si ferma alla prima soluzione trovata, ma può trovare tutte quelle possibili.

I metodi principali offerti dalla classe `Solver` sono i seguenti.

- Metodo `add()`. Serve per aggiungere un vincolo al solver.

```
/*Dati A insieme, x variabile e solver istanza della classe Solver*/
solver.add(x.in(A));
```

- Metodo `check()`. Controlla se il vincolo corrente nel *constraint store* è risolvibile. Se lo è, modifica il constraint store e restituisce `true`. Altrimenti restituisce `false`.

```
/*Dato solver istanza della classe Solver*/
LVar x = new LVar(``x'', 1);
LVar y = new LVar(``y'', 2;
solver.add(x.eq(y));
solver.check(); //FALSE
```

- Metodo `solve()`. Si comporta come il metodo `check()`, con l'unica differenza che se il vincolo non è soddisfacibile solleva un'eccezione.

- Metodo `nextSolution()`. Questo metodo è da usare dopo avere chiamato `check()` o `solve()`, poichè proverà a trovare la prossima soluzione possibile, restituendo `true` se la trova e `false` altrimenti.

Capitolo 2

Thread in Java

2.1 Thread

Un thread è un flusso di controllo indipendente che vive all'interno di un processo. Un singolo processo, infatti, sarà costituito da uno o più thread. Questi thread eseguono attività in parallelo, hanno ciascuno il proprio program counter e stack pointer, ma condividono le risorse allocate al processo.

2.2 Thread in Java

Un'applicazione in Java è sempre costituita da almeno un thread, chiamato *main thread*. Il *main thread* potrà poi creare altri thread. Un thread in Java viene implementato come un oggetto Java che estende la classe astratta `Thread`, ridefinendo il suo metodo astratto `run`, che definisce l'attività del thread. La JVM si occupa di mappare i thread usati nel programma Java sui thread fisici. Tipicamente la mappatura è di tipo one-on-one.

2.2.1 La classe `Thread`

La classe astratta `Thread`, fornita dalla libreria `java.lang`, offre diversi metodi, tra i quali:

- `void run()` - definisce il comportamento del thread, cioè il codice che eseguirà;
- `void start()` - dà inizio all'esecuzione del thread. In particolare, la JVM chiama il metodo `run`.

È inoltre importante notare che la classe `Thread` eredita, tra gli altri, il seguente metodo dalla classe `Object`:

- `public final void wait()` -Il thread corrente si addormenta sull'oggetto, cioè si pone in attesa nella coda `wait` finchè un altro thread non chiamerà il metodo `notify()` o `notifyAll()` su questo oggetto.

Per poter risvegliare un thread in attesa su un oggetto, un altro processo dovrà chiamare uno tra i metodi `public final void notify()` e `public final void notifyAll()` dell'oggetto, ereditati da tutte le sottoclassi di `Object`. Il primo sveglierà uno tra i thread in attesa sull'oggetto, mentre il secondo sveglierà tutti quelli presenti nella coda `wait`.

2.2.2 Stati dei thread

Un thread di Java, in un dato momento, può trovarsi in uno tra i seguenti stati:

- `NEW` - il thread è stato creato e non ha ancora iniziato la sua esecuzione
- `RUNNABLE` - il thread è in esecuzione, sta utilizzando la CPU; il cambiamento dello stato da `NEW` a `RUNNABLE` avviene con la chiamata al metodo `start()`;
- `BLOCKED` - il thread ha eseguito un'operazione bloccante come, ad esempio, una `wait()`;
- `DEAD` - il thread si trova in questo stato dopo avere terminato di eseguire il corpo della funzione `run()`.

2.3 Mutua esclusione

A causa della presenza di processi che vengono eseguiti in parallelo, si può presentare il problema dell'accesso concorrente a risorse condivise. Di conseguenza, l'accesso alle risorse deve essere coordinato tramite *sezioni critiche*.

Una sezione critica è una sequenza di istruzioni che non può essere eseguita contemporaneamente ad altre sezioni critiche. In Java, una sezione critica si individua tramite la parola chiave `synchronized`, che si può usare per rendere sezione critica un intero metodo oppure un singolo blocco. Le due sintassi sono le seguenti:

- `synchronized nomeMetodo(args) {Blocco del metodo}` - tutto il codice contenuto nel blocco del metodo verrà eseguito come una sezione critica;
- `synchronized(Espressione) {Blocco}` - l'espressione deve produrre un valore non nullo, cioè un oggetto su cui il thread si sincronizzerà. Il thread poi eseguirà il contenuto del blocco.

Per sincronizzarsi su un oggetto, il thread tenterà di acquisire un *lock* su questo oggetto. Se riesce ad acquisirlo, eseguirà il codice previsto e poi rilascerà il *lock*. Se invece il lock era già stato acquisito da un altro thread, aspetterà che si liberi.

Capitolo 3

Architettura del sistema

Lo scopo di questo lavoro di tesi è creare all'interno di JSetL i meccanismi necessari a risolvere in parallelo vincoli, cioè provare contemporaneamente più soluzioni possibili e, in caso di successo, comunicare la soluzione trovata, altrimenti comunicare il fallimento. Per farlo, si utilizzeranno i thread di Java.

In questo capitolo descriveremo dapprima la risoluzione di vincoli tramite *or-parallelismo* in riferimento alla libreria JSetL. Successivamente descriveremo l'architettura master-slave del sistema, con un esempio che illustra come avviene la creazione degli slave per la risoluzione di un vincolo. Verranno poi illustrate più nel dettaglio le caratteristiche rispettivamente di master e slave e il loro funzionamento. Infine, verrà descritto un meccanismo di gestione delle possibili race condition.

3.1 Risoluzione di vincoli ed *or-parallelismo*

In generale, la risoluzione di un vincolo JSetL prevede che il vincolo venga riscritto secondo precise regole di riscrittura, che lo trasformano in una serie di vincoli più semplici legati da or (si parla infatti di *or-parallelismo*). Prendiamo come esempio il vincolo

$$\{x, y\} = \{1, 2\} \wedge x \; neq \; 1$$

che internamente viene rappresentato come

$$\{x \cup \{y \cup \{\}\}\} = \{1 \cup \{2 \cup \{\}\}\} \wedge x \; neq \; 1$$

Il vincolo, tramite regole di riscrittura, viene trasformato in questo modo:

$$x = 1 \wedge \{y \cup \{\}\} = \{2 \cup \{\}\} \wedge x \; neq \; 1$$

$$\vee$$

$$x = 1 \wedge \{x \cup \{y \cup \{\}\} = \{2 \cup \{\}\} \wedge x \; neq \; 1$$

$$\vee$$

$$x = 1 \wedge \{y \cup \{\}\} = \{1 \cup \{2 \cup \{\}\} \wedge x \; neq \; 1$$

$$\vee$$

$$\{y \cup \{\}\} = \{1 \cup N\} \wedge \{x \cup N\} = \{2 \cup \{\}\} \wedge x \; neq \; 1$$

La classe `Solver` di JSetL implementa un risolutore che può risolvere questo vincolo usando la tecnica del *backtracking*, che consiste nel provare a risolvere tutti i vincoli concatenati da or, uno alla volta e in modo sequenziale. In particolare, si prova a risolvere il primo disgiunto dell'or e, se questo fallisce, si risolve il secondo e così via, fino a trovare una soluzione.
Nell'esempio di sopra, si prova a risolvere per primo il vincolo

$$x = 1 \wedge \{y \cup \{\}\} = \{2 \cup \{\}\} \wedge x \; neq \; 1$$

qui l'uguaglianza x = 1 lega la variabile x al valore 1 e quindi richiede di risolvere il vincolo 1 neq 1 che è chiaramente falso. Questo fallimento innesca un backtracking che porta ad esaminare dapprima il secondo disgiunto, che risulta falso, e quindi il terzo disgiunto, anch'esso falso. Si procede quindi ad esaminare il quarto disgiunto che, come prima soluzione, viene riscritto nella congiunzione

$$y = 1 \wedge N = \{\} \wedge x = 2 \wedge N = \{\} \wedge x \; neq \; 1$$

che viene facilmente provato essere soddisfacibile, dando origine alla soluzione

$$y = 1 \wedge x = 2.$$

Obiettivo di questo lavoro, invece, è di costruire un meccanismo di risoluzione di vincoli che assegna ogni vincolo c_i di una disgiunzione di vincoli

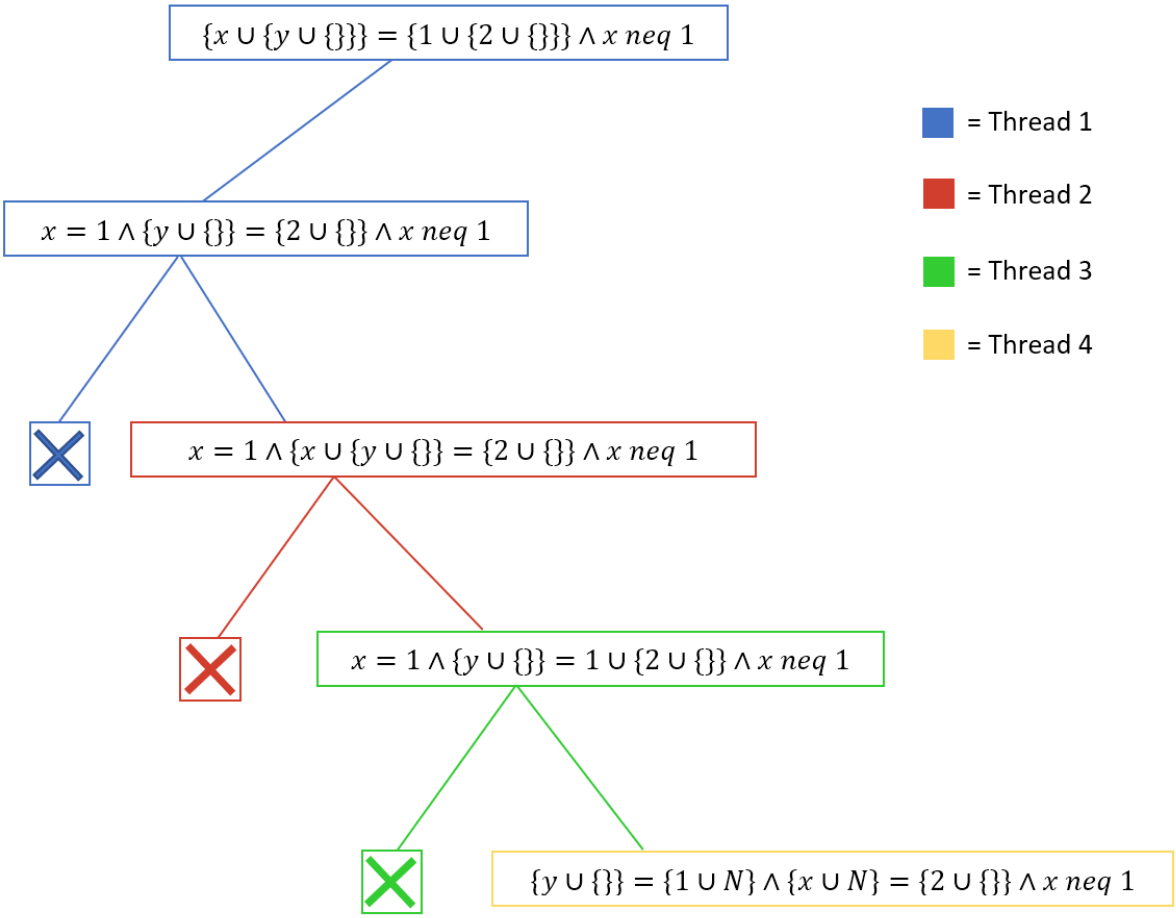
$$c_1 \vee c_2 \vee ... \vee c_n \ \ (n > 0)$$

ad un diverso thread T_i e quel thread proverà a risolverlo. Ovviamente, il vincolo potrebbe dovere essere ulteriormente semplificato e, in questo caso, verrano creati nuovi thread che proveranno a risolvere i vincoli semplificati, e così via.

3.2 Architettura master-slave

L'architettura del sistema che intendiamo realizzare è di tipo *master-slave*. Il master è un'istanza della classe `ParallelSolver`, gli slave sono istanze della classe `ParallelSolverSlave` e ognuno verrà eseguito in un thread apposito.

Il seguente grafico illustra la risoluzione del vincolo visto sopra tramite la creazione di thread slave. Ogni thread mostrato è uno slave e, come si può vedere, ognuno di essi, quando incontra un choice-point, crea un nuovo thread slave che esplorerà l'alternativa successiva e, contemporaneamente, prosegue la risoluzione del proprio disgiunto. Come illustrato, i primi tre thread falliscono, mentre il quarto aprirà nuovi choice-point fino ad arrivare alla soluzione corretta.



Le classi ParallelSolver e ParallelSolverSlave verranno descritte nel dettaglio nel Capitolo 4.

3.3 Thread master

Chiameremo *thread master* il thread in cui viene creata l'istanza di `ParallelSolver`. Tipicamente, questo sarà il main thread. Rimane in esecuzione durante tutta la risoluzione dei vincoli.

Come per la classe `Solver`, la risoluzione avviene tramite una chiamata al metodo `check()`. Il `check()` del `ParallelSolver`, però, a differenza di quello di `Solver`, non risolve direttamente il vincolo, ma crea il primo thread

slave. L'eventuale creazione di ulteriori thread slave è delegata agli stessi, come vedremo nella prossima sezione.

Il master, dopo avere fatto partire il primo thread slave, si pone in attesa di ricevere comunicazioni dagli slave. In particolare, uno di essi potrebbe comunicare al master di avere trovato una soluzione. In quel caso, il master ferma l'esecuzione degli altri slave e comunica all'esterno la soluzione trovata. Successivamente, se richiesto, farà ripartire gli slave fermati, che andranno avanti con la computazione.

Il master tiene traccia di quanti slave sono correntemente in esecuzione e memorizza anche al suo interno un riferimento a tutti gli slave che sono stati fermati e a quelli che hanno trovato una soluzione. Dato che tipicamente il sistema operativo per gestire i thread realizza un mapping uno-a-uno tra i thread creati nel programma e i cores fisici della macchina, è stato ritenuto opportuno limitare superiormente il numero di thread creati in modo che i thread non siano più dei cores

3.4 Thread slave

Il primo thread slave, come detto in precedenza, viene creato direttamente dal master. Ogni thread slave, però, avrà la possiblità di creare altri slave.

La creazione di un nuovo thread verrà fatta nel caso in cui vi siano più possibilità per la risoluzione di un vincolo e sia necessario esplorarle tutte, in particolare quando nel vincolo si apre un nuovo choice-point. Infatti, la creazione di un nuovo thread avviene nel metodo `addChoicePoint()`. È importante notare che gli slave creano altri slave solo se vi sono abbastanza processori liberi. Altrimenti, il calcolo viene eseguito usando il metodo `addChoicePoint` del `Solver` non parallelo.

Gli slave controllano periodicamente, riferendosi al master, se è stata già trovata una soluzione. In questo caso fermeranno la loro esecuzione, ponendosi in attesa. Un thread slave che trovi una soluzione possibile ne informa il master. Dato che questo slave ha trovato una soluzione, questo

e tutti gli altri si fermano e sarà il master, se richiesto, a dare l'ordine di ripartire. Se invece lo slave non trova una soluzione e non incontra nemmeno ulteriori choice-point da espandere terminerà, liberando quindi un processore per eventuali altri slave.

È importante notare che ogni slave ha un riferimento al master (tutti gli slave che stanno risolvendo un determinato insieme di vincoli si riferiscono allo stesso master) e comunicherà esclusivamente con esso.

3.5 Gestione delle race condition

Tutti gli slave agiscono sugli stessi dati di input (i vincoli da risolvere e le variabili e gli insiemi ad essi legati) e questo richiede un meccanismo di gestione delle possibili race condition. Infatti, durante la risoluzione di un vincolo, i dati legati ad esso vengono modificati, ma l'ordine in cui vengono risolti i singoli vincoli non deve influenzare la soluzione finale.

Per evitare il verificarsi di queste condizioni, verrà utilizzato un meccanismo di *copia profonda* delle variabili legate al vincolo da risolvere, realizzata tramite appositi metodi aggiunti alla classe `DeepCloner`. La creazione di una copia profonda permette che ogni thread agisca su una propria istanza di un insieme di dati uguale a quello dato in input. In questo modo, quando il thread modifica le associazioni variabile-valore, non modifica in realtà le associazioni originali. Questo avverrà solo nel caso in cui venga effettivamente trovata una soluzione possibile. Nel caso in cui venga poi richiesta un'altra soluzione, è previsto un meccanismo di ripristino alla situazione originale.

3.6 Comunicazioni fra thread

Per riassumere quanto detto finora, elenchiamo i tipi di comunicazioni che avvengono fra i thread.

- Comunicazioni da slave a master

  - Appena lo slave viene inizializzato, chiede al master quanti cores vi sono in totale nel sistema.
  - Lo slave, prima di creare un nuovo thread, chiede al master quanti slave sono correntemente in esecuzione.
  - Lo slave, quando crea un nuovo thread slave, ne informa il master.
  - Quando uno slave trova una soluzione lo comunica al master.
  - Lo slave informa il master anche quando non trova una soluzione.
  - Periodicamente, lo slave chiede al master se è stata trovata una soluzione e, in caso affermativo, si pone in attesa.

- Comunicazioni da master a slave
  - Il master comunica allo slave, in seguito a richiesta, quanti cores vi sono nella macchina.
  - Il master comunica agli slave che è stata trovata una soluzione.
  - In seguito a richiesta dello slave, il master comunica quanti slave sono correntemente in esecuzione.
  - Se viene richiesto di cercare altre soluzioni, il master fa riprendere l'esecuzione agli slave.
  - Il master richiede agli slave la mappa delle associazioni variabile-valore.

Tutte queste comunicazioni sono implementate all'interno di JSetL utilizzando oggetti condivisi, il cui accesso è sincronizzato tramite opportuni lock.

Capitolo 4

Implementazione

In questo capitolo vedremo nel dettaglio l'implementazione in Java delle classi viste nei capitoli precedenti.

4.1 Classe ParallelSolver

Questa classe (riportata nell'appendice, file ParallelSolver.java) costituisce il master. Come si può vedere, la classe è un'estensione della classe `Solver`, dunque eredita tutti i metodi già presenti in tale classe.

4.1.1 Campi

La classe contiene i seguenti campi:

`protected volatile int counter`: questa variabile serve per tenere traccia del numero di slave correntemente in esecuzione.

`protected volatile boolean solutionFound`: questa variabile verrà usata dagli slave. Quando uno slave troverà una soluzione, modificherà questa variabile cambiando il suo valore a `true`, i thread in esecuzione si bloccheranno e il master comunicherà la soluzione trovata.

`protected final Object mutex`: l'unico scopo di questo oggetto è di essere usato per la sincronizzazione; più avanti vedremo nel dettaglio come ciò avviene.

`protected static final int CORES`: come si può vedere, tale variabile viene inizializzata al momento della creazione dell'istanza corrente di `ParallelSolver` come il numero di cores presenti nella macchina.

`protected Queue<ParallelSolverSlave> solutionSolvers`: tale coda serve per memorizzare tutti gli slave che hanno trovato una soluzione.

`protected LinkedList<ParallelSolverSlave> stoppedSolvers`: questa lista serve per memorizzare tutti gli slave che sono stati fermati dopo che è stata trovata una soluzione, in modo da poterne riprendere l'esecuzione se richiesto.

`protected Map<LObject, LObject> equs`: tale mappa serve per tenere traccia dei valori originali di tutte le variabili associate ai vincoli che verrano risolti, per poterli ripristinare quando viene invocato il metodo `nextSolution()`.

4.1.2 Metodi

La classe contiene i seguenti metodi:

`public boolean check()`: questo metodo effettua l'override del metodo `check()` della classe `Solver`. Questo è il metodo che si occupa della creazione del primo slave.

- Per prima cosa, il vincolo o l'insieme di vincoli da risolvere viene salvato in un `Constraint` chiamato `store`.
- Successivamente, sincronizzandosi sul mutex, si incrementa la variabile `counter` e poi viene creato il primo thread.

- All'interno del thread, sincronizzandosi su un apposito oggetto creato all'interno del metodo, chiamato `w`, viene creata una copia profonda (realizzata tramite la classe `DeepCloner`) dello `store`, in modo che lo slave, risolvendo il vincolo, non modifichi le variabili originali e anche una mappa dei cloni delle variabili (chiamata `clonesMapping`), sempre tramite la classe `DeepCloner`.

- Successivamente viene creata un'istanza di `ParallelSolverSlave` passandovi come parametri un riferimento all'istanza corrente di `ParallelSolver` (che sarà il riferimento al master contenuto nello slave) e la `clonesMapping` realizzata in precedenza.

- Allo slave viene poi aggiunto, tramite il metodo `add` dello slave, che vedremo nella prossima sezione, la copia dello `store` e viene invocato il metodo `check` dello slave.

- Successivamente, ci si sincronizza sul mutex e si eseguono le seguenti operazioni:

  - Per iniziare l'esecuzione del thread, ci si sincronizza su `w` e si invoca il metodo `start` del thread, per poi fare una `wait` su `w`, per aspettare che almeno un thread termini l'esecuzione.
  - Si verifica se la dimensione della coda `solutionSolvers` sia uguale a 0 (cioè nessuno slave ha trovato una soluzione). In tal caso:
    * se anche il `counter` è uguale a 0, significa che non vi è più nessuno slave in esecuzione e di conseguenza non è stata trovata alcuna soluzione. Quindi il metodo `check` termina con un `return false`.
    * Se invece il `counter` è maggiore di 0, si fa una ulteriore `wait` sul `mutex` per aspettare che gli slave terminino la loro esecuzione. Quando ciò avviene vi sono due possibilità: `solutionSolvers` è vuota, cioè nessuno ha trovato una soluzione (questo caso è uguale al precedente, quindi vi sarà anco-

ra un `return false` oppure `solutionSolvers` non è vuota. In questo caso si chiamerà il metodo `pollSolution()`, che vedremo in seguito, si aggiornerà il campo `solutionFound` a `true` e vi sarà un `return true`.

- Viceversa, se la coda `solutionSolvers` è non vuota, si eseguirà lo stesso codice del caso appena mostrato.

`public boolean nextSolution()`: questo metodo effettua l'override del metodo `nextSolution()` della classe `Solver`. Viene invocato dopo che è già stata trovata almeno una soluzione, per trovare quella successiva.

- La prima operazione che questo metodo compie è di ripristinare tutti i valori nella mappa `equs` a quelli originali.
- Successivamente si sincronizza sul `mutex` e si comporta in modo simile al metodo precedente ma con qualche differenza. Infatti,
  - per prima cosa verifica se la coda `solutionSolvers` è vuota. In questo caso, se il counter è uguale a 0 effettua un `return false`,
  - altrimenti modifica la variabile `solutionFound`, assegnandole il valore `false`, in modo che gli slave che riprendono l'esecuzione non si fermino subito pensando che sia stata trovata una soluzione,
  - esegue una `notifyAll` sul `mutex` per svegliare gli slave che si erano addormentati perchè era già stata trovata una soluzione ma non avevano ancora finito la loro esecuzione e
  - infine fa ripartire gli slave nella lista `stoppedSolvers`, svuotando poi questa lista.
- Successivamente effettua una `wait` sul mutex e verrà svegliato quando uno slave avrà trovato una soluzione o quando tutti avranno terminato.
- Successivamente controllerà se la coda `solutionSolvers` è vuota. In tal caso significa che non sono state trovate soluzioni e vi sarà un `return`

`false`, altrimenti verrà invocato il metodo `pollSolution()` e vi sarà un `return true`.

- Infine, nel caso in cui il test iniziale in cui si verificava se la coda `solutionSolvers` fosse vuota sia fallito, si avrà una chamata a `pollSolution()` e un `return true`.

`protected void pollSolution()`: questo metodo viene chiamato quando almeno uno slave ha trovato una soluzione e si trova nella coda `solutionSolvers`. In particolare, il metodo serve a gestire le associazioni tra le variabili originali e i valori possibili trovati durante la risoluzione dei vincoli.

- Il metodo estrae il primo slave dalla coda `solutionSolvers` e lo sposta nella lista `stoppedSolvers`, in modo che questo possa, successivamente, riprendere l'esecuzione.

- Il metodo usa un'istanza della classe `MapJoiner`, chiamata `mapJoiner`, che vedremo in seguito, per unire le mappe contenenti l'associazione chiave-valore di tutte le variabili coinvolte nei vincoli. Infatti, partendo dallo slave corrente, si itera chiamando il metodo `getFather()` della classe `ParallelSolverSlave` fino ad arrivare allo slave che non ha padre e che quindi è quello che è stato creato dal master e, ad ogni iterazione, si aggiunge a `mapJoiner` la mappa dei cloni dello slave corrente.

- Successivamente si effettua l'unione tramite `mapJoiner`.

- Viene poi creata una nuova istanza di `HashMap`, chiamata `equs`, dove vengono inserite le coppie chiave-valore corrispondenti alle variabili da mappare.

- Si invoca poi il metodo `reverse` di `mapJoiner` per evitare che l'unione delle mappe avvenga nell'ordine inverso.

- Si esegue l'unione con il metodo `getComposition()` di `mapJoiner`, salvando il risultato in una nuova mappa chiamata `joined`.

- Si memorizzano tutte le associazioni presenti al momento nel campo `equs`.
- Infine, si modificano le associazioni originali rendendole uguali a quelle contenute nella mappa `joined`.

4.2 Classe ParallelSolverSlave

Questa classe (riportata nell'appendice, file ParallelSolverSlave.java) serve per instanziare gli slave. Anch'essa, come `ParallelSolver` è una sottoclasse di `Solver`.

4.2.1 Campi

La classe contiene i seguenti campi:

`protected final ParallelSolver master`: è il riferimento al master.

`protected static int CORES`: identico al campo `CORES` della classe `ParallelSolver`.

`protected Object stop`: questo oggetto verrà utilizzato unicamente per la sincronizzazione.

`protected Map<Object,Object> clonesMapping`: questa mappa conterrà i cloni delle variabili e gli insiemi legati al vincolo da risolvere.

`protected ParallelSolverSlave father`: è il riferimento al padre (l'istanza di `ParallelSolverSlave` che ha creato l'istanza corrente).

4.2.2 Metodi

La classe contiene i seguenti metodi:

`public ParallelSolverSlave(ParallelSolver master, Map<Object,Object> clonesMapping)`: questo è il costruttore che verrà invocato dal master quando crea il primo slave. Questo costruttore si limita a invocare l'altro costruttore con tre argomenti, aggiungendo come terzo argomento `null`, adesso vedremo per quale motivo.

`public ParallelSolverSlave(ParallelSolver master, Map<Object,Object> clonesMapping, ParallelSolverSlave father)`: questo è il costruttore che si occupa di costruire effettivamente l'istanza. Inizializza i campi `master`, `clonesMapping` e `father` assegnandovi rispettivamente gli argomenti ricevuti in input. Se viene invocato tramite il costruttore visto nel punto precedente, il terzo argomento, `father`, sarà `null`. Infatti, questo accadrà solo se l'istanza che viene costruita è stata creata dal master. In questo caso non vi sarà alcuno slave padre.

`public Map<Object,Object> getClonesMapping()`: questo metodo esegue solo un `return` del campo `clonesMapping`.

`public void addChoicePoint(AConstraint s)`: questo metodo effettua l'override del metodo corrispondente di `Solver`. Non verrà mai invocato direttamente, ma verrà chiamato dal metodo `check` per aggiungere un choice-point quando necessario.

- Per prima cosa, verifica se vi sono cores disponibili. Se non vi sono, invoca il metodo `addChoicePoint` di `Solver` ed effettua un `return`.
- Se invece vi è almeno un core disponibile, incrementa il campo `alternative` del vincolo da risolvere, in modo da esplorare un'alternativa ancora non esplorata dagli altri thread, salva una copia del constraint store e, sincronizzandosi sul mutex del master, incrementa il campo `counter` del master, in modo da indicare che vi è un thread in più in esecuzione.

- Immediatamente dopo, crea il nuovo thread e, come il master, crea una copia dello store e una mappa dei cloni.
- Successivamente, istanzia sul thread un nuovo `ParallelSolverSlave` a cui viene passato come terzo parametro (`father`) un riferimento all'istanza corrente.
- Allo slave viene poi passato tramite il metodo `add` il vincolo da eseguire e infine si invoca il metodo `check` per risolvere il vincolo.
- Il thread così costruito viene fatto partire tramite il metodo `start`.

`public boolean check()`: questo metodo effettua l'override del metodo `check` della classe `Solver`.

- Per prima cosa invoca quest'ultimo metodo, salvando il risultato in una variabile booleana chiamata `result`.
- Vi è poi un ciclo while che prende come espressione la variabile `result`. Se si entra all'interno di questo ciclo, significa che è stata trovata una soluzione. Di conseguenza
  - si aggiunge l'istanza corrente alla coda `solutionSolvers` del master,
  - si modifica il suo campo `solutionFound`, assegnandovi `true` e
  - si effettua una `notifyAll()` sul `mutex` del master, per notificare allo stesso, che si era addormentato, che è stata trovata una soluzione.
  - Successivamente, si effettua una `wait` sul campo `stop`.
  - Infine, alla variabile `result` si assegna il risultato della chiamata al metodo `nextSolution`. Se si arriva a questo passaggio, infatti, significa che il master ha svegliato i thread che si erano fermati per richiedere un'altra soluzione.

Questo ciclo si interrompe quando la variabile `result` sarà uguale a `false`, cioè non è stato trovato alcun risultato possibile.

- In questo caso il thread termina, ma prima decrementa il campo `counter` del master, verifica se questo campo è uguale a zero (in questo caso significa che non vi sono altri thread che stanno eseguendo e di conseguenza la ricerca di una soluzione è fallita) e infine esegue un `return` con la variabile `result`.

`protected void risAConstraint(AConstraint a)`: questo è il metodo che rende possibile il fatto che i thread si fermino quando è già stata trovata una soluzione. Infatti questo metodo, che effettua l'override del metodo corrispondente di `Solver`, viene eseguito periodicamente e, oltre ad eseguire il metodo della classe `Solver`, con una `super.risAConstraint()`, verifica se il campo `solutionFound` del master è uguale a `true` e, in quel caso, effettua una `wait` sul mutex.

4.3 Classe MapJoiner

Questa classe contiene vari metodi per creare l'unione di due o più mappe.

4.3.1 Campi

`private LinkedList <Map<Object,Object>> maps`: questa lista, inizializzata come lista vuota nel momento dell'istanziazione della classe, contiene tutte le mappe da unire.

4.3.2 Metodi

`public void add(Map<Object, Object> map)`: questo metodo serve per aggiungere una mappa alla lista `maps`. Per farlo, usa il metodo `add` della classe `LinkedList`.

`public void reverse()`: questo metodo serve a invertire l'ordine delle mappe nella lista `maps`. In questo modo l'unione si può applicare nell'ordine corretto.

- Il metodo crea una nuova lista di `Map<Object,Object>` chiamata `reversedMaps` e vi aggiunge uno per uno, in ordine inverso, gli elementi della lista originale.
- Infine sostituisce alla lista originale quella invertita.

`public Map<Object,Object> join(Map<Object,Object> first, Map<Object,Object> second)`: questo metodo prende in input due mappe e restituisce una nuova mappa, che è l'unione delle due.

- In particolare, il metodo crea una nuova mappa chiamata `joinedMap` e poi procede a copiarvi ogni valore della prima mappa, verificando però, per ogni coppia chiave-valore se il valore è presente come chiave nella seconda mappa.
- In questo caso procede a unificare, memorizzando in `joinedMap` la chiave contenuta nella prima mappa e il valore contenuto nella seconda.
- Infine, copia in `joinedMap` tutte le coppie della seconda mappa che non siano già presenti ed effettua un return di `joinedMap`.

`public Map<Object,Object> getComposition()`: questo metodo sfrutta il metodo visto al punto precedente per unire tutte le mappe presenti nella lista `maps`.

- Per fare ciò, ottiene un iteratore sulla lista e, utilizzando questo iteratore, scorre tutte le mappe, unendole, finchè non arriva alla fine della lista.
- Infine, fa un return della mappa finale ottenuta.

Capitolo 5

Esempi e casi d'uso

In questo capitolo vedremo alcuni test che sono stati realizzati per valutare la correttezza del lavoro svolto e per valutarne le prestazioni. I test verranno eseguiti utilizzando il testing framework JUnit.

5.1 Test generici

1. Il seguente test costruisce due `LSet` corrispondenti ai due insiemi {1,2} e {2,3} e successivamente ne verifica l'uguaglianza. Ovviamente, tale test fallisce.

```
@Test
   public void test01() {
      ParallelSolver parallelSolver = new ParallelSolver();
      LSet a = LSet.empty().ins(1).ins(2);
      LSet b = LSet.empty().ins(2).ins(3);
      parallelSolver.add(a.eq(b));
      assertFalse(parallelSolver.check());
   }
```

2. Il seguente test costruisce un `LSet` corrispondente all'insieme {2,3} e crea una nuova istanza di `LVar` non inizializzata. Si impone poi, come

vincolo, che la variabile sia contenuta nell'insieme e che non sia uguale a 3. Tale vincolo trova una soluzione, cioè che la variabile sia uguale a 2.

```
@Test
   public void test02() {

      ParallelSolver parallelSolver = new ParallelSolver();
      LSet a = LSet.empty().ins(2).ins(3);
      LVar x = new LVar();
      parallelSolver.add(x.in(a).and(x.neq(3)));
      assertTrue(parallelSolver.check());
   }
```

3. Il seguente test è interessante perchè costruisce un `LSet` contenente i numeri interi da 1 a 9 e impone che una certa variabile appartenga all'insieme. Il test verifica poi che vengano trovate 9 soluzioni possibili.

```
@Test
   public void test03() {
      Solver parallelSolver = new ParallelSolver();
      LSet a = LSet.empty().ins(1).ins(2).ins(3).ins(4).ins(5).ins(6)
          .ins(7).ins(8).ins(9);
      LVar x = new LVar();
      parallelSolver.add(x.in(a));
      assertEquals(9,parallelSolver.forEachSolution((i)->{
         System.out.println("SOLUTION NUMBER " + i);}));

   }
```

4. Il seguente test è stato realizzato appositamente per verificare che l'associazione variabile-valore sia avvenuta correttamente alla fine del calcolo. Infatti, l'ultima riga di codice verifica che alla variabile `i` sia stato assegnato il valore 5.

```
@Test
   public void test04() {
      ParallelSolver parallelSolver = new ParallelSolver();
      IntLVar i = new IntLVar();
      LSet a = LSet.empty().ins(5);
      LSet b = LSet.empty().ins(i);
      parallelSolver.add(a.eq(b));
      assertTrue(parallelSolver.check());
      assertEquals((Integer)5,i.getValue());
   }
```

5.2 Test per problemi combinatori con valutazione delle prestazioni

I seguenti test sono stati realizzati appositamente per valutare lo speed-up ottenuto con il solver parallelo rispetto al solver sequenziale. Tutti i test sono stati eseguiti su un computer con 4 cores.

1. Il seguente test risolve il problema delle N regine e misura le prestazioni. In particolare, risolve il vincolo utilizzando dapprima un'istanza di `Solver` e, successivamente, un'istanza di `ParallelSolver`.

```
@Test
   public void test05() {
      int dimension = 30;
      long timeTotal = 0;
      int T = 5;
      for(int i = 0; i < T; i++) {
         timeTotal += queenTime(new Solver(), dimension);

      }
      System.out.println("Solver: " + timeTotal/T/1000000);
      timeTotal = 0;
      for(int i = 0; i < T; i++) {
```

```
        timeTotal += queenTime(new ParallelSolver(), dimension);

      }
      System.out.println("ParallelSolver: " + timeTotal/T/1000000);
    }

  public long queenTime(Solver solver, int N) {
      int i, j;

      // vars[i] = j iff there is a queen in the position (i, j).
       IntLVar[] vars = new IntLVar[N];

       // No queens on the same row.
       for (i = 0; i < N; ++i)
         vars[i] = new IntLVar(0, N - 1);

       // No queens on the same column.
       solver.add(Constraint.allDifferent((Object[]) vars));

       // No queens on the same diagonal.
       for (i = 0; i < N - 1; ++i)
          for (j = i + 1; j < N; ++j) {
            MultiInterval dom = new MultiInterval(i - j);
            dom.add(j - i);
            // (vars[i] - vars[j]) not in {i - j, j - i}
            solver.add(vars[i].sub(vars[j]).ndom(dom));
        }

       // Try to assign to each variable the median getValue of its
           domain.
       for (i = 0; i < N - 1; ++i)
         solver.add(vars[i].label(ValHeuristic.MEDIAN));

       long time = System.nanoTime();
       // Try to find a solution.
       assertTrue(solver.check());

       return System.nanoTime() - time;
```

```
}
```

L'output di questo test è il seguente:

```
Solver: 354602615
ParallelSolver: 133920844
```

2. Il seguente test mostra un esempio di colorazione di un grafo. Anch'esso misura le prestazioni, come il test precedente.

```
@Test
   public void test06(){
      long sequentialTime = 0;
      int n = 10;
      for(int i = 0; i < n; ++i)
         sequentialTime += testExample06(new Solver());

      long parallelTime = 0;
      for(int i = 0; i < n; ++i) {
         try {
            Thread.sleep(100);
         } catch (InterruptedException e) {
            e.printStackTrace();
         }
         parallelTime += testExample06(new ParallelSolver());
      }
      System.out.println(sequentialTime + " SEQUENTIAL");

      System.out.println(parallelTime + " PARALLEL");

   }

public long testExample06(Solver solver) {
      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));
    long time = System.nanoTime();
    assertEquals(true, solver.check());
    return System.nanoTime() - time;

  }
```

L'output è il seguente:

```
13805635510 SEQUENTIAL
4034242381 PARALLEL
```

5.3 Test per la valutazione delle prestazioni in caso di vincoli con molte alternative possibili

1. Il seguente test valuta le performance risolvendo un vincolo che deve esaminare molte possibili scelte. In particolare, in questo caso verranno esaminate circa $1.6 \cdot 10^5$ alternative.

```
@Test
   public void test07() {
      Solver solver = new Solver();
      LSet aa = LSet.empty();
      for(int i = 0; i < 400; ++i)
         aa = aa.ins(i);
      LVar xx = new LVar();
      LVar yy = new LVar();
      solver.add(xx.in(aa));
      solver.add(yy.in(aa));
      solver.add(xx.eq(99999).and(yy.eq(99999)));

      long sequentialStart = System.nanoTime();
      boolean sequentialResult = solver.check();
      long sequentialEnd = System.nanoTime();

      assertFalse(sequentialResult);
      System.out.println("Sequential time: " + (sequentialEnd -
          sequentialStart));

      Solver parallelSolver = new ParallelSolver();
      LSet a = LSet.empty();
      for(int i = 0; i < 400; ++i)
         a = a.ins(i);
      LVar x = new LVar();
      LVar y = new LVar();
      parallelSolver.add(x.in(a));
```

```
        parallelSolver.add(y.in(a));
        parallelSolver.add(x.eq(99999).and(y.eq(99999)));

        long parallelStart = System.nanoTime();
        boolean parallelResult = parallelSolver.check();
        long parallelEnd = System.nanoTime();

        assertFalse(parallelResult);

        System.out.println("Parallel time: " + (parallelEnd -
            parallelStart));

    }
```

L'output è il seguente:

```
Sequential time: 5957876490
Parallel time: 2431312925
```

Capitolo 6

Conclusioni e sviluppi futuri

In questo lavoro di tesi è stato affrontato il problema della parallelizzazione del risolutore di vincoli presente nella libreria JSetL.

Per ottenere questo risultato è stato realizzato un sistema apposito, all'interno della libreria, che permette di risolvere vincoli in parallelo, sfruttando i metodi già esistenti nella classe `Solver`. Precisamente, sono state realizzate due classi: `ParallelSolver`, che funge da master e ha il compito di creare il primo slave, di porsi in attesa di eventuali soluzioni trovate e, se richiesto, di far riprendere l'esecuzione agli slave dopo avere trovato una soluzione, e `ParallelSolverSlave` che, per consentire la parallelizzazione, esegue in un thread apposito ed ha la funzione di slave. Quest'ultima è la classe che si occupa di risolvere effettivamente i vincoli. Ogni istanza di `ParallelSolverSlave` esplora una diversa alternativa del vincolo da risolvere e, quando apre un choice-point, crea un nuovo slave e poi continua l'esecuzione. Inoltre, è stata implementata anche una classe `MapJoiner`, che serve come supporto all'unificazione di mappe di tipo chiave-valore. Questa classe permette di gestire le mappe delle associazioni delle variabili originali del vincolo da risolvere ai relativi valori.

Sono stati poi implementati vari test, dapprima per valutare la correttezza del lavoro svolto e l'interscambiabilità con la classe `Solver` e, successivamente, per valutarne le prestazioni. Da questi test è emerso un notevole speed-up

rispetto al solver sequenziale, soprattutto per quanto riguarda vincoli complessi che esplorano numerose alternative.

Per quanto riguarda possibili sviluppi futuri, si potrebbe sicuramente lavorare alla realizzazione di altri test per valutare la correttezza e le performance. Sarebbe interessante, soprattutto, osservare le prestazioni su macchine con numerosi cores per valutare la scalabilità di quanto realizzato e individuare un criterio per calcolare il numero ottimo di thread da utilizzare rispetto ai cores fisici della macchina.

Si potrebbe, inoltre, se possibile, modificare le regole di riscrittura per fare in modo che le varie alternative dei vincoli abbiano complessità simili, in modo da bilanciare meglio il carico su tutti i thread disponibili.

Ringraziamenti

Vorrei dedicare questa pagina ad alcune persone che hanno reso possibile la realizzazione di questo lavoro, che per me rappresenta un traguardo importantissimo, frutto di un percorso di laurea impegnativo ma che mi ha regalato molte soddisfazioni.

Desidero ringraziare anzitutto il Prof. Gianfranco Rossi, per la pazienza e disponibilità che ha dimostrato durante tutto il percorso di tirocinio e stesura della tesi e per i suoi preziosi consigli.

Un grazie speciale va, poi, ai miei genitori Maria Grazia e Roberto che, durante questi tre anni, hanno reso possibile col loro supporto, anche nei momenti più difficili, la mia crescita personale.

Infine, vorrei ringraziare il mio ragazzo Andrea per avere sempre creduto in me, con affetto e pazienza, e per avermi spronata a migliorarmi sempre più.

Riferimenti bibliografici

[1] Ian P. Gent et al:
A review of literature on parallel constraint solving. Theory and Practice of Logic Programming, Volume 18, Special Issue 5-6 (Special Issue on Parallel and Distributed Logic Programming), September 2018 , pp. 725-758

[2] Gianfranco Rossi, Elio Panegai, Elisabetta Poleo
JSetL: a Java library for supporting declarative programming in Java
Software Practice & Experience 2007; 37:115-149.

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

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

[5] Gianfranco Rossi, Federico Bergenti
Nondeterministic Programming in Java with JSetL
Fundamenta Informaticae 140 (2015) 393–412

[6] Java documentation
https://docs.oracle.com/javase/7/docs/api/

Appendice A

Appendice

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 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 ParallelSolver extends Solver{

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

   public ParallelSolver() {

      mutex = new Object();
      counter = 0;
      solutionFound = false;
   }


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

```
Object w = new Object();

synchronized(mutex) {
   ++counter;
}

Thread thread = new Thread(() -> {
   Constraint copy;
   Map<Object, Object> clonesMapping;
   synchronized (w) {
      DeepCloner deepCloner = new DeepCloner();
      copy = deepCloner.visit(store);
      clonesMapping = deepCloner.getClonesMap();
      w.notify();
   }
   ParallelSolverSlave parallelSolver = new ParallelSolverSlave(this,
        clonesMapping);
   parallelSolver.add(copy);

   parallelSolver.check();
});

synchronized (mutex){
   synchronized (w) {
      try {
         thread.start();
         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 (ParallelSolverSlave 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(){
    ParallelSolverSlave solutionSolver = solutionSolvers.poll();
    stoppedSolvers.add(solutionSolver);
    ParallelSolverSlave 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;
         }
      });
   }
}
```

File: ParallelSolverSlave.java

```
package jsetl;

import java.util.Map;

public class ParallelSolverSlave extends Solver {

   protected final ParallelSolver master;
   protected static int CORES = ParallelSolver.CORES;
   protected Object stop = new Object();
   protected Map<Object, Object> clonesMapping;
```

```
protected ParallelSolverSlave father;

protected ParallelSolverSlave(ParallelSolver master, Map<Object, Object
    > clonesMapping) {

   this(master, clonesMapping, null);
}

public Map<Object,Object> getClonesMapping() {
   return clonesMapping;
}

protected ParallelSolverSlave(ParallelSolver master, Map<Object, Object
    > clonesMapping, ParallelSolverSlave father) {
   this.master = master;
   this.clonesMapping = clonesMapping;
   this.father = father;
}

protected ParallelSolverSlave getFather() {
   return father;
}

@Override
public void addChoicePoint(AConstraint s) {

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

   s.alternative++;
   Constraint store = getConstraint();

   Object w = new Object();

   synchronized(master.mutex) {
      ++master.counter;
```

```
    }
    Thread thread = new Thread(() -> {
       Constraint copy;
       Map<Object, Object> clonesMapping;
          synchronized (w) {
             DeepCloner deepCloner = new DeepCloner();
             copy = deepCloner.visit(store);
             clonesMapping = deepCloner.getClonesMap();
             w.notify();
          }
          ParallelSolverSlave parallelSolver = new ParallelSolverSlave(
              master,clonesMapping,this);
          parallelSolver.add(copy);

          parallelSolver.check();
       });

    synchronized (w) {
       try {
          thread.start();
          w.wait();
       } catch (InterruptedException e) {
          e.printStackTrace();
       }
    }
}

@Override
public boolean check() {
    boolean result = super.check();
    while(result){
       synchronized (stop){
          synchronized (master.mutex) {
             master.solutionSolvers.add(this);
             master.solutionFound = true;
             master.mutex.notifyAll();
          }
```

```
            try {
               stop.wait();
            } catch (InterruptedException e) {
               e.printStackTrace();
            }
         }
         result = nextSolution();
      }

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

   @Override
   protected void risAConstraint(AConstraint a)throws Failure, Fail {
      super.risAConstraint(a);
      if(master.solutionFound) {
         synchronized(master.mutex) {
            try {
               master.mutex.wait();
            } catch (InterruptedException e) {
               e.printStackTrace();
            }
         }
      }
   }
}
```

File: MapJoiner.java

```
package jsetl;

import java.util.HashMap;
import java.util.Iterator;
```

```
import java.util.LinkedList;
import java.util.Map;


public class MapJoiner {

   private LinkedList<Map<Object,Object>> maps = new LinkedList<>();

   public void add(Map<Object,Object> map) {
      maps.add(map);
   }

   public void reverse() {
      LinkedList<Map<Object,Object>> reversedMaps = new LinkedList<>();
      for(Map<Object,Object> map : maps) {
         reversedMaps.addFirst(map);
      }
      maps = reversedMaps;
   }

   public Map<Object,Object> join(Map<Object,Object> first, Map<Object,
       Object> second) {
      Map<Object,Object> joinedMap = new HashMap<>();
      first.forEach((key, value) -> {
         if(second.containsKey(value))
            joinedMap.put(key, second.get(value));
         else
            joinedMap.put(key, value);
      });

      second.forEach((key,value) -> {
         if(!first.containsValue(key))
            joinedMap.put(key, value);
      });
      return joinedMap;
   }

   public Map<Object,Object> getComposition() {
```

```
      Iterator<Map<Object,Object>> iterator = maps.iterator();
      Map<Object,Object> joinedMap = iterator.next();

      while(iterator.hasNext()) {
         joinedMap = join(joinedMap, iterator.next());
      }

      return joinedMap;
   }
}
```