Università di Pisa

Facoltà di Scienze Matematiche Fisiche e Naturali
Corso di Laurea in Informatica

Tesi di Laurea

Redesign di una componente di ottimizzazione per un ambiente parallelo e distribuito

Andrea Farruggia

Tutore aziendale	Tutore accademico
Dott. Francesco Bernazzani	Prof. Antonio Frangioni

Anno Accademico 2008/2009

Sommario

La componente algoritmica MAIOR è stata progettata avendo come modello di riferimento un ambiente di tipo sequenziale. I recenti sviluppi (diminuzione del ritmo di crescita della potenza di calcolo del singolo core, aumento del numero di core disponibili) impongono però un ripensamento volto a sfruttare le potenzialità presenti di calcolo parallelo. Scopo del redesign è quello di permettere la presenza di un certo grado di parallelismo (sia multithread che distribuito) all'interno della componente, in modo da poter venire incontro in modo effettivo alle esigenze, da parte dei clienti, di versioni sempre più efficienti dell'applicativo.

In questo documento verrà fatta una veloce panoramica sul problema trattato per passare poi ad una analisi del design corrente, alla raccolta dei requisiti e alla esposizione del nuovo design.

Indice

1 Ambito del progetto **2**

1.1 Il problema da risolvere 2

1.2 Il modello matematico scelto 3

1.2.1 Modellazione dei turni 3

1.2.2 Modello matematico 3

1.3 Column Generation 4

1.3.1 Introduzione 4

1.4 Il generatore 6

1.5 Il solver 6

1.5.1 Utilizzo del rilassamento lagrangiano 7

1.5.2 Algoritmo bundle 7

1.6 Fissaggio delle colonne 8

2 L'implementazione attuale **10**

2.1 Concetti preliminari 10

2.2 Descrizione del sistema 14

2.2.1 Descrizione statica 14

2.2.2 Descrizione dinamica 20

2.3 Aspetti su cui è necessario intervenire 22

3 Obiettivi e requisiti **25**

3.1 Obiettivi 25

3.2 Requisiti raccolti dal design attuale 26
3.2.1 Requisiti funzionali 26
3.2.2 Requisiti non funzionali (qualitativi) 28
3.3 Requisiti addizionali 28
3.3.1 Requisiti funzionali 28
3.3.2 Requisiti qualitativi 30

4 Il design proposto **31**
4.1 Idee alla base 31
4.2 Descrizione delle classi 36
4.2.1 Componente SetPartLib 36
4.2.1.1 ConcreteController 36
4.2.1.2 Controller 38
4.2.1.3 Worker 39
4.2.1.4 SolutionManager 39
4.2.1.5 Partitioner 40
4.2.2 Sottosistema Core 40
4.2.2.1 WorkerFacade 42
4.2.2.2 WorkerData 42
4.2.2.3 ComponentFactory 43
4.2.2.4 Fixer 43
4.2.2.5 Unfixer 43
4.2.2.6 StabilizerFacade 43
4.2.3 Sottosistema Stabilizer 44
4.2.3.1 StabilizerControl 46
4.2.3.2 StabilizerControl 46
4.2.3.3 Solver 47
4.2.3.4 GeneratorFacade 47
4.2.4 Generator 47
4.2.4.1 GeneratorFaçade 48
4.2.4.2 Generator 48

4.2.4.3 Buffer 50
4.2.5 ColumnManager 51
4.2.5.1 ColumnManager 53
4.2.5.2 Column, DomainColumn e CColumn 54
4.2.5.3 Updatable 55
4.2.5.4 Notifica delle colonne fissate e vincoli nonlineari 55
4.3 Diagrammi di sequenza 56
4.4 Dislocazione delle componenti nei nodi e modalità di comunicazione 60
4.4.1 Suddivisione in processi 60
4.4.2 Modifiche al design per il caso sequenziale 62
4.5 Tecnologie utilizzate 64
4.5.1 Intel TBB 64
4.5.2 MPI 66
4.6 Analisi dei rischi 67

5 Parallelizzazione del generatore 69
5.1 Introduzione 69
5.2 Implementazione 70
5.2.1 Thread-safeness del codice utilizzato 71
5.2.2 Stato condiviso 72
5.2.3 Definizione della classe Worker 73
5.2.4 Esecuzione concorrente dei Worker 74
5.3 Risultati 76

Capitolo 1

Ambito del progetto

1.1 Il problema da risolvere

La componente in questione risolve il problema della formazione dei turni per un giorno tipo in una azienda di trasporto. Il design attuale permette la definizione di diversi tipi di turno (crew scheduling, turn scheduling, combinato), caratteristica che deve essere mantenuta anche dal nuovo design.

Per poter rendere più concreta la discussione e per poter fissare un modello che illustrerà i problemi che dovranno essere indirizzati dal nuovo design, è possibile ipotizzare che la componente risolva il problema del turn scheduling, cioè la definizione di turni a partire dai luoghi (o task) da coprire, tenendo ben in mente che il design sarà indipendente dal tipo di modellazione scelta, ipotizzando solamente uno schema a generazione di colonne con fissaggio incrementale.

Nello specifico, si tratta di coprire con esattamente un turno guida ogni corsa programmata per un dato giorno, in modo quindi da garantire che tutte le corse siano guidate da esattamente un autista. I turni guida devono rispettare una normativa, quindi non tutte le combinazioni di corse danno vita a turni corretti. Possono essere previsti vincoli ulteriori sotto forma di vincoli numerici per tipologia di turno (posso fare al massimo 14 turni di tipo

part time), oppure vincoli di media su parametri tipici del turno (es. i turni devono avere una media di lavoro non superiore a 6 ore). L'obiettivo che ci si pone è quello di coprire il servizio minimizzando i costi.

Segue una panoramica a livello concettuale sulle tecniche utilizzate dalla componente algoritmica MAIOR, rielaborata dal documento [9].

1.2 Il modello matematico scelto

1.2.1 Modellazione dei turni

La modellazione del problema fa riferimento ad un *grafo di compatibilità* $G = (N, A)$.

L'insieme dei nodi $N = \{1, \ldots, n\} \bigcup \{O, D\}$ contiene un elemento per ogni attività da coprire, ed i due nodi "deposito" O e D che non rappresentano attività "reali" ma corrispondono a depositi fittizi in cui si immagina che i turni abbiano inizio e fine.

Dati $(i, j) \in \{1, \ldots, n\} \times \{1, \ldots, n\}$, l'insieme degli archi A contiene (i, j) se e solo se le attività i e j possono essere "eseguite" consecutivamente in un turno. Analogamente, $(O, j) \in A$ se e solo se un turno può iniziare in j, e $(i, D) \in A$ se e solo se un turno può terminare in i.

Consideriamo per ogni arco $(i, j) \in A$ una variabile booleana x_{ij} che ha valore 1 se le attività i e j vengono svolte in sequenza e 0 altrimenti, e sia c_{ij} il costo associato a tale arco.

Un turno può dunque essere visto come un cammino in tale grafo.

1.2.2 Modello matematico

Si è scelto un modello di PLI (Programmazione Lineare Intera) tipo *Set Partitioning*: gli elementi da coprire sono i task, gli insiemi che le coprono sono i turni, le cui variabili sono binarie,indicando se il turno è o meno in soluzione.

Il modello scelto è dunque il seguente:

$$\min \sum_{p \in P} c_p x_p$$

$$\sum_{p \in P} a_{ip} x_p = 1 \; \forall i \in N \tag{1.1}$$

$$x_p \in \{0, 1\} \; \forall p \in P \tag{1.2}$$

dove:

- P è un insieme di indici associati all'insieme dei turni ammissibili;
- N è un insieme di indici associati ai task;
- I coefficienti binari a_{ip} valgono 1 se e solo se l'attività i appartiene al turno p.

Il vincolo (1.1) garantisce la partizione (ogni corsa appartiene ad uno ed un solo turno); il vincolo (1.2), che modella il concetto di "turno scelto", rende il problema difficile.

È significativo notare come $P \subset \{0, 1\}^N$, cioè è presente una nozione di ammissibilità del singolo turno.

Oltre ai vincoli che modellano il partizionamento è possibile che si abbia bisogno di esprimere anche vincoli "globali", che interessano cioè l'insieme dei turni nella loro globalità. Quando tali vincoli sono presenti, vengono modellati tramite dei vincoli lineari.

1.3 Column Generation

1.3.1 Introduzione

Il modello matematico scelto è molto flessibile, in quanto qualsiasi vincolo di ammissibilità sui turni è codificato nel vincolo $p \in P$; tuttavia, l'insieme

dei turni ammissibili è enorme, dunque non è possibile enumerarli in modo esaustivo.

La tecnica utilizzata per superare il problema è quella della *Column Generation*: questa tecnica permette di risolvere il *rilassamento continuo* del problema formalizzato nella sezione e del suo duale 1.2:

$$\min \left\{ \sum_{p \in P} c_p x_p : \sum_{p \in P} a_{ip} x_p = 1 \; i \in N, \; x_p \in [0,1] \; p \in P \right\} \tag{P}$$

$$\max \left\{ \sum_{i \in N} \pi_i : c_p - \sum_{i \in N} \pi_i a_{ip} \geq 0, \; p \in P \right\} \tag{D}$$

Tale tecnica permette di risolvere (P) e (D) selezionando un sottoinsieme $B \subset P$ di dimensioni ridotte e considerando il *Master Problem* (P_B), ossia la restrizione di (P) in cui sono presenti solo le colonne in B (si ha cioè $x_p = 0$ per $p \notin B$); il suo duale (D_B) è un rilassamento di (D).

Poiché (P_B) è di dimensioni ridotte, è possibile risolverlo in un tempo ragionevole, ottenendo una soluzione ottima x^* e, nel contempo una soluzione ottima π^* di (D_B); x^* (completata con zeri) è ammissibile per (P), mentre in generale π^* non lo è per (D).

È però possibile verificare in modo efficiente se esistono colonne in $P \setminus B$ che corrispondono a vincoli di (D) violati da π^*, assumendo che il costo (ma non necessariamente l'ammissibilità) di un turno sia esprimibile come somma dei costi dei suoi task, ossia $c_p = \sum_{(i,j) \in p} c_{ij}$; in questo caso, infatti, si può definire il *costo ridotto* $\overline{c}_{ij} = c_{ij} - \pi_i^*$ degli archi di G, ottenendo quindi che il costo ridotto di un cammino p:

$$\overline{c}_p = \sum_{(i,j) \in p} \overline{c}_{ij} = c_p - \sum_{i \in N} \pi_i^* a_{ip} \tag{1.3}$$

Si ha che (1.3) è negativo se e solo se il vincolo duale corrispondente a p è violato.

Pertanto, risolvendo un problema di cammino di costo minimo su G (il

cui algoritmo è delineato nella sezione 1.4) rispetto ai costi ridotti è possibile o dimostrare che π^* è soluzione ammissibile (e quindi ottima) anche per (D), e di conseguenza x^* è ottima per (P), oppure determinare una nuova colonna $\overline{p}$ tale che la sua aggiunta a B possa determinare un miglioramento del valore della soluzione del Master Problem; i due casi corrispondono rispettivamente al fatto che il cammino di costo ridotto minimo abbia costo non negativo, oppure negativo.

Tale processo può essere iterato, determinando infine una soluzione ottima per (P); tale soluzione, in generale non intera, viene utilizzata tramite la tecnica esposta nella sezione 1.6 per ottenere delle soluzioni intere.

1.4 Il generatore

L'algoritmo che determina delle colonne a costo ridotto negativo è implementato nella componente algoritmica denominata "generatore". La modellazione utilizzata dall'azienda prevede un grafo diretto aciclico, che permette algoritmi per la risoluzione del problema più efficienti rispetto al caso generale. Tuttavia, poiché non tutti i turni generabili sono ammissibili per via delle varie normative da applicare, il problema in realtà è un problema di CSPT (*Constrained Shortest Path Tree*), che lo rende NPC; pertanto il problema può solo essere risolto in modo euristico. Inoltre, poiché vi sono varie normative da considerare, il problema viene risolto tante volte quante le normative presenti, rendendo il generatore una delle componenti algoritmiche più esose in termini di risorse. Uno degli obiettivi del tirocinio consiste nel ridurre significativamente il tempo di esecuzione del generatore.

1.5 Il solver

La soluzione del Master Problem è implementata dalla componente algoritmica denominata "solver".

1.5.1 Utilizzo del rilassamento lagrangiano

Il Master Problem è un problema di Programmazione Lineare (PL), e potrebbe quindi essere risolto dai diversi solutori (open source o commerciali) disponibili. Le dimensioni estremamente grandi che il problema può assumere nelle applicazioni di MAIOR, specialmente quando la dimensione di B cresce, rendono però questa scelta poco efficiente dal punto di vista computazionale.

La strategia utilizzata da MAIOR consiste nel non approcciare direttamente P_B con tecniche standard di PL, ma considerare il rilassamento Lagrangiano dei vincoli di copertura (in presenza di vincoli globali, anch'essi vengono rilasciati):

$$\phi(\lambda) = \left\{ \sum_{p \in B} c_p x_p + \sum_{i=1}^{n} \lambda_i \left(1 - \sum_{p \in B} a_{ip} x_p \right) : x_p \in [0,1]\ p \in B \right\} \tag{1.4}$$

ed il suo corrispondente *duale lagrangiano*:

$$\max \{ \phi(\lambda) : \lambda \in \mathbb{R}^n \} \tag{DL}$$

La soluzione di (DL) è affidata ad un algoritmo di tipo *bundle* che sfrutta un insieme di subgradienti.

1.5.2 Algoritmo bundle

Dato un punto appartenente allo spazio delle soluzioni ammissibili, si determina il subgradiente in quel punto. Esso viene utilizzato in una combinazione convessa con un bundle di altri subgradienti, nella ricerca di un punto adiacente che migliori la funzione obiettivo. Quando la norma del subgradiente approssima lo zero, il solutore esce con la soluzione ottima. Il solutore massimizza la funzione lagrangiana ϕ rispetto al vettore λ.

Per evitare, nella ricerca del massimo, il rischio della divergenza, in una fa-

se che preliminare dell'algoritmo, viene generato un insieme di turni (*tripper*) che assicurino una copertura totale (di partenza) di tutti gli elementi.

1.6 Fissaggio delle colonne

Ottenuta una soluzione ottima per il rilassamento (primale x^* e duale π^*), l'algoritmo risolutivo MAIOR costruisce una soluzione intera in modo incrementale.

Si alternano fasi di fixing (si scelgono alcuni turni $p \in B$ e si fissa $x_p = 1$) e di risoluzione del rilassamento continuo rispetto al sottospazio delle variabili indotto dal fixing, fino a quando tutte le attività sono coperte.

La scelta dei turni da fissare è cruciale per l'efficacia dell'approccio.

Fino a qualche anno fa, la fase di fixing era guidata esclusivamente dalla soluzione duale: i turni di B (disponibili al termine della Column Generation) venivano ordinati per costo ridotto crescente, ed i primi turni della lista entravano a far parte della soluzione discreta, fino ad aver coperto una determinata percentuale di attività.

Attualmente, invece, il criterio di scelta combina insieme l'informazione primale e duale. In particolare, il criterio dei costi ridotti viene ancora utilizzato per definire un sottoinsieme $X \subset B$ di turni promettenti; all'interno di X, però, la scelta viene compiuta utilizzando i valori della soluzione primale x^*, in quanto si considera che un turno con $x^\star_p$ "grande" (vicino a uno) ha più probabilità di far parte di una buona soluzione rispetto ad uno con $x^\star_p$ "piccolo" (vicino a zero).

Poiché, l'approccio descritto è euristico, la soluzione determinata può, in generale, non essere ottima.

Al fine di esplorare il più possibile la regione ammissibile discreta del problema, l'algoritmo MAIOR produce più soluzioni intere ripetendo un certo numero di volte il processo.

Per questo, una volta calcolata una soluzione si effettua uno sfixing:

una percentuale dei turni presenti nella soluzione intera viene rimosso dalla soluzione.

Lo sfixing è utile per poter "correggere" dei fissaggi di colonne globalmente sconvenienti, riconducendosi a dei sottoproblemi con un valore della funzione obiettivo elevata. Si è osservato che la scelta delle colonne da sfissare basata sui costi ridotti non sia significativamente più effettiva di una scelta casuale: ciò mostra come una colonna possa essere sconveniente per il problema di PLI ed essere conveniente per il problema rilassato, in quanto la continuità dei domini delle variabili della PL permette di avere più colonne con gli stessi task in soluzione. Una delle forme di parallelismo introdotte mirerà a chiarire meglio l'aspetto dello sfixing, introducendo una tecnica alternativa per l'esplorazione dello spazio delle soluzioni, non più puramente iterativo.

Capitolo 2

L'implementazione attuale

Il capitolo illustra, in modo necessariamente indicativo e sommario, le scelte di design relative alla componente *SetPartLib*. Per una descrizione più esauriente, anche se leggermente datata ed illustrata con dei formalismi diversi da quelli utilizzati in questo documento, si può consultare la documentazione interna MAIOR [1, 2]

2.1 Concetti preliminari

Qualsiasi implementazione concreta di un algoritmo basato su generazione colonne deve trattare dei problemi che si incontrano con l'istanziazione degli stessi su problemi di dimensioni considerevoli, come quelli trattati dall'azienda.

La MAIOR ha affrontato e risolto (con diversi gradi d'efficacia) tali problemi introducendo dei concetti che risultano pertanto essenziali alla fine della comprensione dell'architettura stessa; di seguito verranno velocemente introdotti, al fine di poter fare riferimento ad essi nelle sezioni successive.

Colonne attive, fissate e bloccate L'insieme B generato nel tempo può diventare estremamente grande, causando dei problemi sia nell'occupazione

di memoria sia, in modo più considerevole, nella convergenza del solver; vengono pertanto utilizzate delle tecniche che permettono di ridurre l'insieme B. Tali tecniche sono motivate dal fatto che le soluzioni del master problem nelle prime iterazioni possono essere molto distanti da quelle delle iterazioni successive; questo comporta dei costi ridotti diversi nel tempo. E' evidente, dunque, come le colonne generate nelle prime iterazioni possano essere inutili nelle iterazioni successive (cioè $x_i^\star = 0$, se $x^\star$ è la soluzione ottima del master problem e i è l'indice di colonna).

Si definiscono dunque tre attributi booleani alle colonne: una colonna è

- *attiva* se fa parte dell'insieme B, cioè viene utilizzata nella formulazione del Master Problem
- *fissata* se la si considera selezionata nel problema di set partitioning, cioè si fissa il campo $x_i^\star = 1$
- *bloccata* se la colonna non può essere eliminata dalla memoria

I tre attributi sono stati introdotti per adempire necessità differenti:

- Le colonne *attive* sono state introdotte per distinguerle da quelle *inattive*, cioè da quelle presenti in memoria ma che non entrano a far parte di B; in questo modo si può realizzare, in modo implicito, un *buffer* delle colonne che può essere utilizzato dal generatore per restituire delle colonne a costo ridotto negativo senza doverle rigenerare una seconda volta. L'idea si basa sul fatto che una colonna può essere significativa all'iterazione i, non esserla all'iterazione j e ridiventarla all'iterazione k, con $i < j < k$. Allo stato attuale, tuttavia, l'approccio non viene sfruttato, in quanto le colonne presenti nel buffer sono tutte attive (non tenendo conto del fatto che il problema della convergenza del solver è più stringente del problema della memoria)
- Come accennato nella sezione 1.7, la soluzione del problema viene prodotta incrementalmente effettuando il fissaggio delle colonne e risol-

vendo il sottoproblema indotto; l'attributo *fissata* indica quali sono le colonne che fanno parte della soluzione corrente ma che non devono essere utilizzate nella formulazione del problema.

- L'algoritmo genera più soluzioni intere: una colonna è bloccata se fa parte di una soluzione: può dunque entrare a far parte dell'insieme B ma non può essere eliminata dalla memoria.

Tripper Come già velocemente accennato in 1.5, per evitare problemi di divergenza, lo spazio delle soluzioni viene confinato in un ipercubo, definito aggiungendo all'insieme B delle colonne fittizie che limitano lo spazio primale. I tripper sono generati in base alle informazioni ottenibili dalle **linee** del problema (quindi è possibile generarli all'inizio, una volta per tutte). Per una descrizione più accurata della generazione dei tripper si può consultare il paragrafo "Divergenza" del documento [1].

Separazione tra componenti algoritmiche generali e di dominio Il problema "fisico" che stiamo considerando è la creazione di turni guida; tuttavia, come già velocemente accennato, l'approccio scelto (generazione di colonne) prevede delle componenti algoritmiche indipendenti rispetto al particolare modello scelto (come, ad esempio, il solver del master problem ed il ciclo di stabilizzazione). Al fine di poter applicare lo schema di generazione di colonne anche a modelli diversi, garantendo un buon riuso del codice, è necessario mantenere tale separazione anche nell'implementazione concreta. Si definiscano le componenti (nell'accezione più generale possibile: classi, metodi, strutture dati...) software che dipendono dal particolare modello come *componenti di dominio*.

I seguenti aspetti richiedono delle componenti di dominio:

- Generazione delle colonne (l'algoritmo di SPT vincolato è specifico del modello scelto)

- Informazioni associate alla colonna: ad esempio, nel turn scheduling, ad ogni colonna è associato un turno, con diverse informazioni associate (come la normativa sotto la quale è stata generata e le zone d'inizio e di fine)

- Euristica per la scelta delle colonne da fissare. Al momento attuale l'euristica è di tipo generico; tuttavia non è da escludere che in futuro il fissaggio delle colonne sia fatto con delle euristiche che possano tener conto delle proprietà delle colonne

Al fine di rendere il design parametrico rispetto a questi aspetti, sono state effettuate le seguenti decisioni:

- La classe *GeneratoreColonne* ha la maggior parte dei metodi *abstract*, in modo che possano essere ridefiniti

- Poiché vi sono delle classi, non di dominio, che trattano le colonne (come il *GestoreColonne* o il *MasterProblemSolver*), queste non devono dipendere dalla classe che include le informazioni di dominio. Per tal motivo la classe *ColonnaDaCTurno* deriva dalla classe *Colonna*, che include esclusivamente la rappresentazione della colonna.

- Il metodo *calcola* del *Fixer* è virtual

Anche il nuovo design dovrà mantenere suddetta separazione; l'argomento verrà trattato nelle sezioni successive.

Vincoli globali nonlineari che richiedono informazioni di dominio In certi casi è necessario introdurre dei vincoli *globali* (cioè che riguardano l'insieme dei turni selezionati, non l'ammissibilità del turno in sé) nonlineari, che richiedono informazioni di dominio per la determinazione del soddisfacimento. La soluzione attuale fa ricorso all'*observer pattern*, in cui l'oggetto

da monitorare è l'ultima colonna fissata (si ricorda che i vincoli globali sono forzati dal fixer), mentre l'observer è il generatore (che conosce qual è la classe derivata di Colonna).

Ad ogni fissaggio di colonna viene invocato il metodo *AggiornaDopoFixTurno*.

Supponendo che ogni colonna i abbia un *conflict set* c_i, definito così:

$$c_i = \{j \neq i \mid \forall x \text{ soluzione primale}, x_i = x_j = 1 \implies x \text{ non è ammissibile}\}$$

AggiornoDopoFixTurno imposta $B = B \setminus c_i$ se l'ultima colonna fissata è i, cioè elimina dall'insieme delle colonne attive le colonne in conflitto con quella appena fissata.

2.2 Descrizione del sistema

L'algoritmo è realizzato dalla componente *SetPartLib*, di cui segue una descrizione indicativa delle classi nella figura 2.1.

Di seguito, una breve panoramica sulle responsabilità delle varie classi ed una descrizione del comportamento dinamico.

2.2.1 Descrizione statica

ConstrainedShortestPath Rappresenta l'entry point dell'algoritmo, cioè la classe che coordina il lavoro di tutte le altre per ottenere una o più soluzioni al problema.

Essa, ad alto livello:

- Istanzia, direttamente o indirettamente, le altre classi coinvolte nel calcolo (strutture dati del generatore escluse)
- Calcola il lower bound

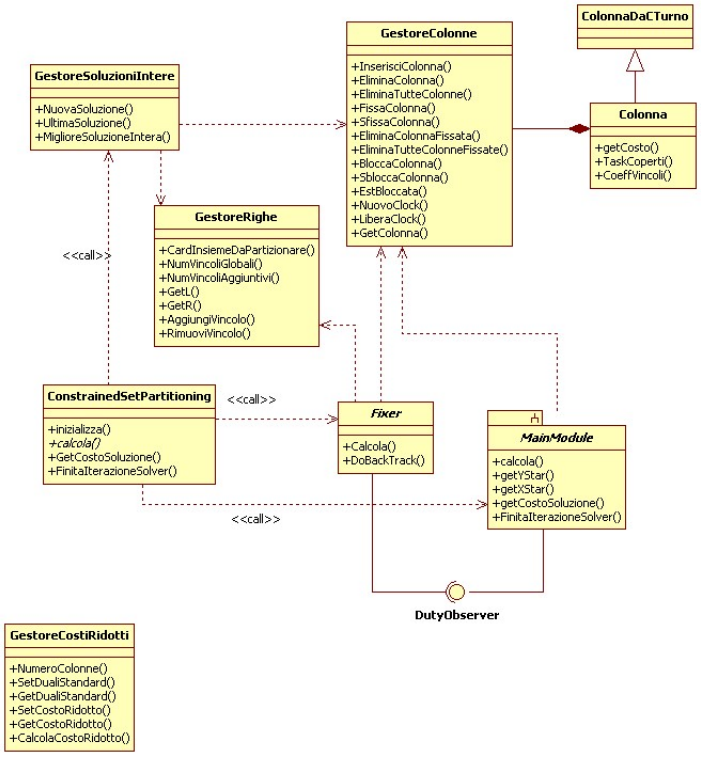

Figura 2.1: Descrizione indicativa della componente

- Calcola un certo numero di soluzioni intere (correntemente 40) chiamando allo scopo il master problem ed i vari fixer (fixer ed unfixer)
- Termina notificando la soluzione migliore tramite la classe *GestoreSoluzioniIntere*

Sottosistema MainModule Questo sottosistema è responsabile per l'esecuzione del ciclo di stabilizzazione descritto nella sezione 1.4: effettua il ciclo generazione-solver, a partire dal pool di colonne iniziali (alcune fissate), fino a quando la condizione di ottimalità non è soddisfatta o qualche altro criterio basato sulla qualità della soluzione o sul numero di passi effettuati.

Le parti del sottosistema consistono del generatore di colonne (*Generato-*

reColonne), del solver del master problem (*MasterProblemSolver*) e di una classe mediatrice, che ne coordina le iterazioni (*MainModule*). Un diagramma delle classi indicativo del sottosistema è presente nella figura 2.2.

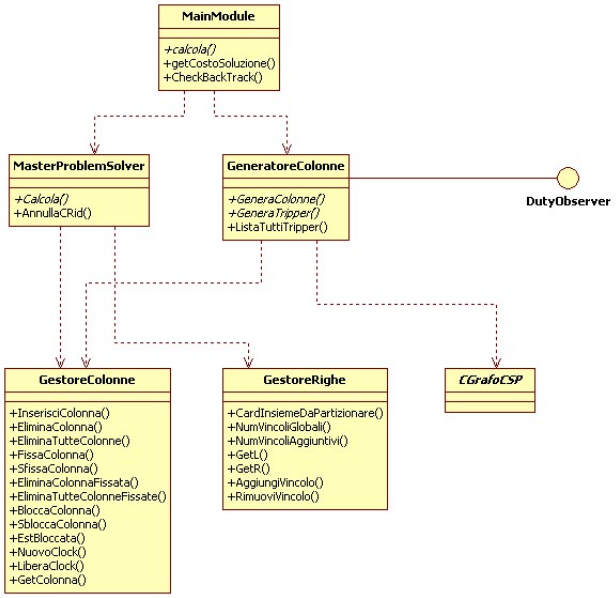

Figura 2.2: Descrizione indicativa del sottosistema MainModule

La classe del sottosistema utilizzata da *ConstrainedShortestPath* è MainModule.

MainModule Come sopra velocemente menzionato, MainModule è la classe che media la coordinazione tra il solver ed il generatore, controllando il ciclo di stabilizzazione.

Essa, ad alto livello:

- Istanzia MasterProblemSolver e GeneratoreColonne

- Effettua il ciclo di stabilizzazione, richiamando alternativamente (e sequenzialmente) il generatore ed il solver fino a quando non si è soddisfatta una delle condizioni di stop (ciclo stabile, massimo numero di passi raggiunto o notifica di interruzione da parte dell'utente), generando una soluzione primale ammissibile (e duale non necessariamente ammissibile)

Colonna Rappresenta la descrizione matematica di una colonna del problema. Nell'implementazione attuale è un vettore (sparso) di elementi reali.

ColonnaDaCTurno Classe derivata da Colonna, arricchisce la classe base con le informazioni di dominio associate alla colonna. ColonnaDaCTurno viene costruita a partire da *CTurno*, la classe utilizzata dal generatore per rappresentare un turno generato.

GestoreColonne Il GestoreColonne ha varie responsabilità:

- è il *container* delle colonne disponibili. Ad ogni colonna viene assegnato, all'inserzione, un id univoco che deve essere utilizzato per le operazioni successive di eliminazione, ottenimento e modifica
- realizza la differenziazione tra colonne attive, bloccate e fissate
- permette di conoscere, dopo ogni generazione, il delta (rispetto all'iterazione precedente) dell'insieme delle colonne attive e fissate. In altri termini, se X_i è l'insieme delle colonne (attive, fissate, bloccate) all'iterazione i, il GestoreColonne permette di conoscere l'insieme $\delta_i = X_i \setminus X_{i-1}$.

Per poter realizzare i primi due punti, l'interfaccia di GestoreColonne prevede i tipici metodi di un container (InserisciColonna, EliminaColonna, GetColonna...) assieme a dei metodi che permettono di modificarne gli attributi (BloccaColonna, EstBloccata, FissaColonna...).

Per poter realizzare l'ultimo punto, è stato introdotto il concetto di *Clock*. Un Clock può essere richiesto tramite il metodo NuovoClock, e rappresenta idealmente la registrazione al servizio di notifica delle classi inserite e cancellate temporalmente comprese tra la richiesta dello stesso e la deregistrazione dal servizio (LiberaClock).

A livello più concreto, il Clock mantiene una lista di riferimenti (indici) delle colonne aggiunte e rimosse. Il GestoreColonne mantiene una lista di Clock; ad ogni inserimento e rimozione di una colonna, il riferimento relativo viene aggiunto nell'opportuna lista di tutti i clock registrati.

GestoreCostiRidotti La responsabilità del GestoreCostiRidotti è quella di calcolare il costo ridotto delle colonne. L'implementazione della classe garantisce che, per ogni colonna, il costo ridotto sia calcolato *al più* una volta per iterazione, anche se questo è richiesto, all'interno di una iterazione, più di una volta (utilizzando memoization).

GeneratoreColonne Implementa la componente algoritmica illustrata nella sezione 1.4: il suo compito è quello di aggiungere al GestoreColonne delle colonne con costo ridotto negativo, secondo la soluzione duale corrente, oltre che eliminare dall'insieme delle colonne il conflict set c_i della colonna i fissata dal Fixer, come illustrato nella sezione 2.1.

Per una panoramica dei dettagli implementativi del generatore relativo al modello trattato, si può consultare [1]; di seguito verranno esposti i concetti propedeutici alla comprensione di determinate scelte del design proposto.

Pezzi attivi Come già accennato, un turno è una successione di task; al fine di agevolare la generazione delle colonne, è introdotto il concetto di *pezzo*, definito come una successione di task: in tal modo, un turno è una successione di pezzi.

Poiché vi è una relazione biunivoca tra nodi del grafo e pezzi, l'istanziazione di quest'ultimi deve essere effettuata prima della generazione vera

e propria. Il numero dei nodi istanziati può risultare eccessivo: in tal caso viene utilizzata una euristica di selezione dei nodi "interessanti", che ha come input:

- L'insieme dei nodi istanziati
- La soluzione duale
- La cardinalità del sottoinsieme risultante dalla selezione

e come output la lista dei nodi selezionati. Un nodo selezionato si definisce *attivo*. La funzione che effettua la selezione è *CGrafoCSP::InitListaNodiAttivi*. Nell'implementazione attuale la lista è calcolata ad ogni generazione.

Archi attivi La lista di adiacenza del grafo può avere due implementazioni: *statica* se le FS (forward star) sono calcolate una volta per tutte, *dinamica* altrimenti.

Nel caso di FS statiche, gli archi hanno l'attributo "*attivo*". In questo caso il grafo è definito come $G = \left(\overline{N}, \overline{A}\right)$, dove $\overline{N} = \{i \in N \mid i \text{ è attivo}\}$ e $\overline{A} = \left\{(i,j) \in \left(A \cap \overline{N} \times \overline{N}\right) \mid (i,j) \text{ è attivo}\right\}$. Anche la lista degli archi attivi, nell'implementazione attuale, è calcolata ad ogni generazione.

MasterProblemSolver Implementa la componente algoritmica descritta in 1.5, relativo al sottoproblema indotto dalle colonne fissate. La soluzione può essere ottenuta, dopo aver invocato la *Calcola*, tramite i metodi *GetYSegnato* e *GetXStar*.

Fixer Un oggetto di tipo Fixer , a partire da un insieme di colonne fissate $T \subseteq X$, ne restituisce uno $T' \subseteq X$. Entrambe le componenti citate in 1.6 (la componente che effettua il fissaggio e quella che effettua lo sfissaggio) sono degli oggetti di tipo Fixer. Nella soluzione attuale, il Fixer che effettua il fissaggio è chiamato *IncrTurniInSol*, mentre quella che effettua lo sfissaggio

è chiamata *RigeneraPezzi*; per quanto riguarda queste due componenti, è interessante notare che:

- IncrTurniInSol è anche responsabile del *backtrack*: nel caso in cui l'ultimo fissaggio delle colonne ha comportato un incremento eccessivo del costo della soluzione ottima del sottoproblema, allora vengono sfissate delle colonne con costo ridotto eccessivo.
- L'euristica di sfissaggio utilizzata attualmente da RigeneraPezzi fornisce risultati comparabili rispetto ad una scelta casuale delle colonne da sfissare: ciò può significare che la soluzione duale dell'iterazione corrente può non essere significativa nel decidere se il fissaggio di una colonna è sconveniente o meno.

GestoreRighe La classe ha le seguenti responsabilità:

- memorizzare i vettori l ed u associati al problema di set partitioning;
- fornire un sistema di gestione dei vincoli generati a tempo di esecuzione
- tenere traccia delle righe coperte dalle colonne fissate
- distinguere le linee associate agli elementi da partizionare, quelle relative ai vincoli globali e quelle generate a tempo di esecuzione

GestoreSoluzioniIntere La classe si occupa di salvare, in un opportuno formato, le soluzioni intere generate nel ciclo di vita della componente (al momento attuale, 40).

2.2.2 Descrizione dinamica

Segue una descrizione molto succinta del comportamento della componente alla richiesta del calcolo delle soluzioni del problema.

Per risolvere il problema, viene invocata la funzione *CSP3LoopMaior::Calcola*, che effettua i seguenti passi:

1. Inizializza le varie classi che fanno parte della componente
2. Per ogni soluzione da generare:
   (a) Fino a quando c'è qualche task non coperto da una colonna fissata:
      i. Effettua il ciclo di stabilizzazione richiamando *MM3LoopMaior::Calcola* (la prima volta in assoluto è utile per il calcolo del lower bound)
      ii. Effettua il fissaggio di alcune colonne tramite *IncrTurniIn-Sol::Calcola*
   (b) Salva la soluzione intera trovata
   (c) Se è necessario generare altre soluzioni intere, effettua lo sfissaggio di alcune colonne
3. Scrivi la soluzione migliore su file

Per quanto riguarda *MM3LoopMaior::Calcola*:

1. Nel caso in cui si stia calcolando il lower bound (cioè la prima volta), genera delle colonne con costo qualsiasi (in quanto non vi è ancora una soluzione duale con cui calcolare i costi ridotti)
2. Fino a quando non è stabile:
   (a) Elimina i turni peggiori (vedi 2.1)
   (b) Effettua il backtracking, se necessario (vedi 2.2.1)
   (c) Genera delle colonne a costo ridotto negativo tramite *GenC3LoopMaior::GeneraColonne*
   (d) Chiama il MasterProblemSolver tramite *BundleMaior2005::Calcola*
   (e) Se il ciclo è stabile per più di due iterazioni, termina

Per quanto riguarda *GenC3LoopMaior::GeneraColonne*:

1. Genera, opzionalmente, i turni da file
2. Inizializza i nodi ed archi attivi (vedi 2.2.1)
3. Per ogni normativa, per ogni zona d'origine:
   (a) Inizializza la lista dei nodi ed archi attivi
   (b) Genera i turni da 1 e 2 pezzi
   (c) Genera tutti i turni, risolvendo il problema di CSPT, inserendoli in uno heap
4. Codifica i turni presenti nello heap in colonne e li inserisce nel gestore colonna

Per *BundleMaior2005::Calcola*:

1. Ottieni la lista dei tripper
2. Crea la nuova PhiOracle ed inizializzala
3. Chiama la Phi
4. Aggiorna le variabili duali

2.3 Aspetti su cui è necessario intervenire

Da una analisi del codice esistente, è possibile individuare alcune caratteristiche della soluzione attuale che devono essere cambiate affinché si possa adottare una soluzione concorrente o distribuita. Altri aspetti, più orientati all'architettura e meno a scelte implementative, saranno esplorati nella sezione 4, dopo aver illustrato i casi d'uso ed i requisiti del nuovo design.

Iterazione delle strutture dati L'iterazione in gran parte delle strutture dati presenti nella componente è implementa in modo simile:

```
class Container<X> {
    X* getFirst();
    X* getNext();
}
```

la chiamata a getFirst() imposta una variabile, interna alla struttura dati, che punta al primo elemento del container, restituendo tale elemento; le successive chiamate a getNext() avanzano tale puntatore di una posizione, restituendo il riferimento all'elemento puntato.

Una soluzione di questo tipo è intrinsecamente thread-unsafe, in quanto due thread che accedono alla struttura dati in questa modalità si contendono l'accesso alla variabile interna. Una soluzione consiste nell'eliminare i due metodi succitati ed implementare un iteratore, sulla falsariga dei container della libreria standard c++.

Schema "load and shot" Alcune classi presentano il seguente schema:

```
class control {
    void setElement(ElementToSet x);
    void process()
}
```

tramite setElement si imposta un oggetto che condizionerà la computazione effettuata da process. Un esempio di classe che segue tale schema è InsiemiDiNormative: *InstallaNormativaAttiva* è la setElement mentre molti metodi della classe possono essere paragonati a process).

Anche in questo caso vi è una concorrenza per l'accesso alla variabile interna alla classe; vi sono più soluzioni possibili:

- eliminare la setElement, rendendo parametrica la process (process(ElementToSet x));

- associare ad ogni thread una copia dell'oggetto (nel caso in cui non sia necessario, ai fini di control, tenere traccia di tutte le richieste effettuate alla classe stessa);

- cambiare l'interfaccia nel modo seguente:

```
class control {
    RegistrationElement *register(ElementToSet x);
    void process(RegistrationElement registration);
}
```

  tale soluzione è opportuna se è necessario notificare control dell'elemento su cui si intende lavorare (ad esempio, nel caso in cui sia necessario effettuare qualche computazione costosa in base all'elemento x).

In base ai casi osservati nel codice, la prima soluzione sembra quella più ragionevole.

Variabili globali accedute in scrittura La componente presenta un gran numero di strutture dati modificabili, accedute tramite riferimento globale.

I problemi di thread-unsafeness sono evidenti se:

- i metodi della struttura dati non prevedano qualche forma di sincronizzazione;

- il riferimento alla struttura dati non è costante.

Per risolvere il primo punto è sufficiente sincronizzare in modo opportuno i metodi della struttura dati, utilizzando possibilmente delle librerie portabili come Boost [3].

Per risolvere il secondo punto, è sufficiente utilizzare una variante del Singleton pattern in cui è garantita la thread-safety ed è possibile cambiare l'istanza univoca della struttura in modo sicuro.

Capitolo 3

Obiettivi e requisiti

3.1 Obiettivi

L'obiettivo del refactoring consiste nell'avere un design delle classi che permetta di poter implementare facilmente una certa gamma di parallelismo nella computazione, cercando di mantenere dove possibile parte del codice presente.

I modelli di parallelismo applicabili in questo contesto consistono essenzialmente nella suddivisione dello spazio delle soluzioni assegnato ad ogni worker (la computazione eseguita nella versione attuale del software da MainModule) e nell'interazione tra generatore e solver, in modo che l'uno possa approfittarsi della soluzione parziale dell'altro. La suddivisione dello spazio delle soluzioni in questo contesto non può essere implementato semplicemente assegnando un sottoinsieme delle colonne ad un worker e scegliendo quindi la migliore tra le soluzioni trovate, in quanto una soluzione consiste in un sottoinsieme delle colonne presenti nel buffer che rispetti i vincoli ed una qualsiasi partizione delle colonne escluderebbe a priori alcune soluzioni. Affinché sia possibile suddividere lo spazio delle soluzioni è necessario permettere ad una colonna di appartenere a più working set (definito come L'insieme delle colonne assegnate ad un worker) oppure di specificare vincoli aggiuntivi ad

ogni sottoproblema associato ad worker: uno degli obiettivi del refactoring consiste dunque nel definire una componente che permetta di mantenere ed accedere a diversi sottoinsiemi dell'insieme delle colonne generate in modo da non richiedere un'occupazione esponenziale della memoria, permettendo inoltre di poter accedere anche all'intero insieme delle colonne generate. Sarà inoltre necessario definire una componente, Control, che coordini le attività svolte dai vari worker.

Per "interazione tra generatore e solver" si intende la possibilità di notificare in modo asincrono le due componenti circa gli ultimi aggiornamenti da parte dell'altra senza poter interferire con la computazione in atto.

Il design naturalmente non prevede una riscrittura completa di tutta la libreria, quindi il design "erediterà" delle componenti già presenti nel design attuale la cui natura non impedisce la parallelizzazione dell'applicazione.

3.2 Requisiti raccolti dal design attuale

Seguono i requisiti del nuovo design raccolti da un'analisi del sistema attuale. Poiché i casi d'uso si adattano male a progetti di natura algoritmica, i requisiti verranno illustrati come una lista di asserzioni che dovranno essere vere nel contesto del design proposto; ad ogni modo la descrizione dinamica descritta in 2.2.2 può essere vista come una collezione di casi d'uso.

3.2.1 Requisiti funzionali

Deve essere possibile:

1. implementare il ciclo di stabilizzazione-fixing-unfixing descritto nel capitolo 1;

2. poter memorizzare una soluzione intera determinata da 1;

3. effettuare, da parte della componente che gestisce la computazione, delle decisioni in base allo stato della computazione (soluzione primale e duale, numero di passi effettuati, condizioni di arresto, etc.);

4. poter specificare se il ciclo di stabilizzazione deve iniziare con il solver (dopo un fissaggio) o con il generatore (alla prima iterazione);

5. poter specificare se sia necessario effettuare la generazione di alcuni turni non guidata da una soluzione duale (ad esempio alla prima iterazione o dopo uno sfissaggio);

6. poter specificare dei parametri da applicare alla prima generazione del ciclo di stabilizzazione (ad esempio, se sia necessario effettuare una generazione che non tenga conto della soluzione duale);

7. decidere, nel ciclo di stabilizzazione, se sia opportuno o meno fare backtracking;

8. gestire, in modo esplicito, i tripper;

9. avere accesso, da parte del generatore, del solver e del fixer, alla soluzione corrente (primale/duale);

10. avere accesso, da parte del generatore, del solver, del fixer e della componente che gestisce le soluzioni intere, al pool delle colonne;

11. definire una colonna $x \in X$ come attiva, fissata, bloccata[1];

12. conoscere, in un arco di tempo fissato, quali colonne siano state (dis)attivate, (s)fissate, (s)bloccate;

13. conoscere l'esito del calcolo del MasterProblemSolver e del Generatore;

14. associare ad ogni colonna del pool un identificatore univoco;

[1]La nozione di colonna attiva/bloccata/fissata può essere implementata in modo implicito, non necessariamente come attributo della colonna

15. poter iterare sull'insieme delle colonne in base al loro stato (attive, bloccate, fissate);

16. tenere conto della presenza dei vincoli nonlineari di dominio nel calcolo della soluzione intera.

3.2.2 Requisiti non funzionali (qualitativi)

Bisogna:

1. mantenere la separazione tra sottocomponenti di dominio e sottocomponenti generali della componente SetPartLib;

2. calcolare il costo ridotto di una colonna in modo efficiente, possibilmente utilizzando memoization;

3. limitare il numero di colonne attive e totali;

4. limitare il numero di pezzi ed archi attivi;

5. rendere le classi non interessate dal refactoring thread-safe.

3.3 Requisiti addizionali

Alla luce degli obiettivi esposti nella sezione Section 3.1, è possibile formulare i seguenti requisiti.

3.3.1 Requisiti funzionali

Deve essere possibile:

1. poter esplorare in parallelo diverse regioni dello spazio primale delle soluzioni (sia *worker* la componente logica che effettua l'esplorazione);

2. poter riutilizzare il lavoro svolto da un worker da parte degli altri worker;

3. poter far lavorare il generatore ed il solver in parallelo, approfittandosi delle soluzioni parziali date dai due problemi;

4. poter controllare in modo centralizzato il lavoro svolto dai vari worker (convergenza, qualità della soluzione, colonne generate...), in modo da poter compiere delle decisioni su di essi in base al comportamento complessivo;

5. approfittarsi delle possibilità di parallelismo presenti nella generazione di colonne (ad esempio, il generatore attuale risolve tanti problemi di CSPT indipendenti tante quante sono le normative);

6. Poter distribuire l'algoritmo su nodi computazionali differenti.

Viene inoltre fornito il seguente caso d'uso per testare la flessibilità del design:

1. Vengono generate un insieme di colonne S

2. Si definiscono k sottoinsiemi $S_k \subset S, S_i \neq S_j \quad i, j = 1 \dots k, \ i \neq j$ e si istanziano k worker $w_1 \dots w_k$; il worker w_i ha inizialmente come colonne *fissate* e *bloccate* le colonne S_i

3. Si lancia la computazione dei k worker

4. Si monitora l'attività dei worker. Quando uno di questi sta esplorando una zona dello spazio primale poco "interessante", viene eliminato e viene creato un altro worker, possibilmente suddividendo lo spazio delle soluzioni del worker più interessante

5. Dopo aver trovato un certo numero di soluzioni interessanti, interrompi la computazione.

3.3.2 Requisiti qualitativi

Bisogna:

1. contenere il consumo di memoria: una implementazione in cui il consumo di memoria è una funzione del tipo $f(n) = k * g(n)$, dove g è la funzione di complessità in spazio del sistema attuale e k è il numero di worker allocati, non è sostenibile;

2. adottare una soluzione che sia idonea sia per un ambiente *multithread* (cioè a memoria condivisa), sia per uno *distribuito* (cioè a passaggio di messaggi) sia *misto*;

3. utilizzare dei framework e delle librerie portabili.

Capitolo 4

Il design proposto

Il capitolo seguente introduce il design proposto, illustrandone la struttura e la dinamica nell'implementazione dei requisiti, una panoramica delle tecnologie proposte per l'implementazione, un esempio di dislocazione delle componenti nel caso in cui si voglia adottare un ambiente distribuito ed una analisi dei possibili rischi.

La discussione farà riferimento ai precedenti capitoli del presente documento per giustificare le varie scelte.

4.1 Idee alla base

Il design è stato principalmente guidato dai requisiti forniti nella sezione 3.3, pertanto successivamente la numerazione dei requisiti si riferirà, a meno che sia esplicitamente indicato diversamente, a quelli contenuti in quella sezione.

Worker Il requisito 1 definisce una componente "worker", la cui responsabilità consiste nell'esplorazione di un sottospazio delle soluzioni. Il concetto di worker è implementato tramite la componente *Core*, implementata tramite servizio asincrono (vedi paragrafo 4.1).

Servizi asincroni Un *servizio asincrono* è una componente le cui funzioni sono implementate in modalità asincrona: l'invocazione della funzione ritorna immediatamente, mentre l'esito dell'operazione viene notificato successivamente al chiamante tramite una interfaccia.

I servizi sono i seguenti:

- *Worker*, come già accennato nel paragrafo 4.1
- *Generator*, il generatore di colonne;
- *Solver*, il solver del Master Problem;

L'utilità di un servizio asincrono è evidente: tramite questi è possibile per il chiamante effettuare altre operazioni mentre il servizio invocato processa la richiesta. Un esempio in cui la convenienza di tale paradigma sarà chiara riguarda la classe che media l'interazione tra generatore e solver: questo chiama i due servizi l'uno dopo l'altro, bloccandosi in attesa di notifiche da parte di uno dei due servizi.

WorkerFacade e Core Per il requisito 4, la computazione dovrebbe essere gestita in modo centralizzato; ciò si realizza definendo la classe mediatrice *WorkerFacade*, il cui scopo sarà coordinare i servizi offerti dai vari sottosistemi (ciclo di stabilizzazione, fissaggio e sfissaggio delle colonne). *WorkerFacade*, essendo una classe mediatrice, sarà anch'essa coinvolta nel paradigma ad eventi. Si può pensare che la responsabilità di *WorkerFacade* sia simile a quella di *ConstrainedSetPartitioning*.

WorkerFacade, inoltre, è la classe façade del sottosistema *Core*, la cui responsabilità è quella di trovare delle soluzioni intere ad una restrizione del problema, come si vedrà successivamente. Si può pensare per il momento che le responsabilità della classe *Core* siano simili a quelle dell'intera componente SetPartLib attuale.

Generazione e buffer delle colonne La computazione svolta da un worker qualsiasi consiste fondamentalmente nella generazione delle colonne e nella risoluzione del Master Problem; si ha pertanto che la computazione che può essere riutilizzata da parte di più worker (punto 2) si riferisce alla generazione di colonne. Vi sono vari modi per poterlo realizzare: un modo elegante di farlo consiste nel considerare il generatore come un *servizio* (implementato con un thread pool, per soddisfare il requisito 5) offerto ai vari worker. In questo modo il generatore può possedere un *buffer* composto da un sottoinsieme delle colonne generate durante il ciclo di vita dell'applicazione; tale buffer non è visibile direttamente dai worker, che lavoreranno sul proprio gestore delle colonne. Per via del requisito 1 della sezione 3.3.2, i buffer dovranno sistematicamente eliminare delle colonne aggiunte durante la generazione.

Il buffer può essere dislocato su vari nodi di calcolo (vedi requisito 6), distribuendo dunque le colonne generate su un insieme di nodi.

Per rendersi conto di come tale struttura permetta il riuso del lavoro di generazione, si può descrivere a livello intuitivo quel che accade quando un worker w_i richiede una colonna al generatore:

1. Al momento della richiesta, il generatore chiede ai buffer di restituirgli delle colonne con costo ridotto negativo per il worker w_i;

2. Ciascun buffer cerca delle colonne con costo ridotto negativo nel proprio pool, aggiungendole al manager delle colonne del worker e notificando il numero di colonne generate al generatore

3. Il generatore attende dai buffer la notifica del numero di colonne restituite e, se queste sono troppo poche, genera delle colonne a costo ridotto negativo, inserendole nei buffer.

è evidente, dunque, come una colonna generata in seguito alla richiesta da parte di un worker w_j possa essere restituita al worker w_i, nel caso in cui questa abbia costo ridotto negativo per w_i.

Registrazione al generatore Il generatore può dover tener traccia di alcune informazioni associate ad ogni worker (ad esempio, informazioni allegate alla struttura dati utilizzata per la generazione); per tal motivo è necessario effettuare in modo esplicito la registrazione, in modo che possa aver senso parlare di deregistrazione: in tal modo si ha un modo semplice per eliminare le informazioni nel generatore che non sono più necessarie.

Parallelismo tra generatore e solver Poiché il solver è implementato utilizzando dei metodi di tipo iterativo, è possibile effettuare il ciclo di stabilizzazione generatore-solver in modo non strettamente sequenziale: quando viene lanciata la computazione del generatore, viene lanciata la computazione del solver prima che il generatore abbia terminato, in modo da poter aggiungere le colonne alla rappresentazione via via che queste vengono generate, in modo da diminuire il tempo richiesto alla stabilizzazione. In teoria è anche possibile notificare al generatore le nuove soluzioni duali mano a mano che queste vengono trovate dal solver, lanciando il generatore prima che il solver abbia terminato; tuttavia la fattibilità della cosa dipende dal particolare algoritmo utilizzato per risolvere il problema di dominio. Nel caso del Turn Scheduling, avvantaggiarsi delle soluzioni sub ottimali del solver da parte del generatore è infattibile in quanto viene utilizzato un algoritmo di tipo CSPT, che non permette di poter aggiornare i costi sugli archi senza dover fondamentalmente ricominciare da capo.

Anche in questo caso viene utilizzato il paradigma ad eventi.

Vincoli nonlineari con informazioni di dominio La soluzione è simile a quella in uso attualmente, in quanto si utilizza il pattern *observer* per la notifica delle colonne fissate.

In questo caso l'oggetto osservato è l'insieme delle colonne fissate, mentre l'observer è una classe apposita, in relazione con l'insieme delle colonne, con l'oggetto di registrazione e con la struttura dati utilizzata per la generazione, il cui scopo è quello di eliminare dall'insieme delle colonne del worker il

conflict set della colonna fissata e di modificare le informazioni associate alla struttura dati affinché quella colonna non sia generata (né restituita dal buffer) nelle prossime iterazioni.

Controller Per poter approfittare di un ambiente distribuito (vedi requisito 6), oltre alla distribuzione del buffer in vari nodi è opportuno distribuire la computazione delle varie soluzioni.

Ciò viene ottenuto associando ad ogni istanza della componente Core un processo ed un identificativo.

Nello spazio di indirizzamento saranno presenti un solver, un fixer/unfixer, un oggetto responsabile del controllo delle componenti algoritmiche e le strutture dati necessarie a mantenere lo stato della computazione (gestore delle colonne, gestore delle righe, etc.).

L'identificativo è necessario in quanto l'istanza di un generatore non è associata ad alcun worker. Poiché il generatore può dover mantenere uno stato distinto per ogni worker, sarà necessario specificare l'identità del worker al momento della richiesta di generazione colonne.

Partitioner E' dunque necessario trovare una partizione o comunque una copertura dello spazio delle soluzioni, in modo da poter differenziare i vari worker.

Vi possono essere diverse strategie nel decidere la partizione: ad esempio, si può generare un certo numero di colonne e decidere di fissarne alcune in un worker e di non fissarle (dunque di non generarle nemmeno) in un altro, oppure si possono inserire dei vincoli nel generatore, in modo da suddividere meglio lo spazio delle soluzioni. Poiché la partizione può dunque dipendere da informazioni di dominio (può essere necessaria, ad esempio, un'analisi della struttura dati utilizzata dal generatore), la scelta è implementata nella classe astratta *Partitioner*.

Poiché la scelta del partizionamento può richiedere diverse cose (qual-

che colonna, informazione duale, soluzione intera...), *Partitioner* utilizzerà il sottosistema *Core*, implementando ed utilizzando le opportune interfacce.

Selezione del modello Al fine di non dover avere tante copie della libreria tanti quanti sono i domini modellati, si associa alle coppie interessanti (dominio, strategie algoritmiche) un nome mnemonico, che sarà utilizzato per specificare il tipo concreto delle classi astratte che dovranno essere istanziate.

4.2 Descrizione delle classi

Segue una descrizione delle classi che compongono il design, supponendo per il momento che tutti gli oggetti siano presenti sullo stesso spazio per il momento ed escludendo, quindi, le considerazioni sulla loro distribuzione su nodi computazionali, sulle modalità di comunicazione e sulle relative modifiche e aggiunte da apportare al design delle classi; tali aspetti verranno esplorati nella sezione 4.3.

4.2.1 Componente SetPartLib

La figura 4.1 illustra la struttura della componente algoritmica SetPartLib.

4.2.1.1 ConcreteController

La responsabilità della classe ConcreteController è quella di risolvere il problema, coordinando i vari nodi di computazione a cui è demandato il calcolo della soluzione dei sottoproblemi indotti da *Partitioner*; è pertanto l'*entry point* dell'algoritmo. I vari *Worker* notificano *ConcreteController* dei progressi effettuati tramite l'interfaccia *Controller*. *ConcreteController* utilizza *ConcreteSolver* per la memorizzazione delle soluzioni ottenute dai *Worker* a cui demanda il calcolo di alcune soluzioni e con *Partitioner* per l'otteni-

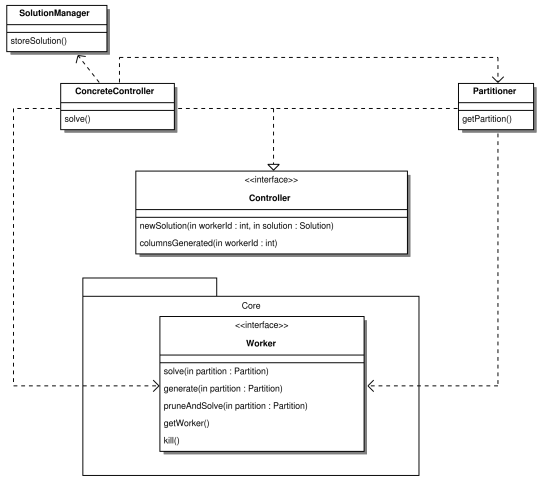

Figura 4.1: Descrizione della componente SetPartLib

mento delle partizioni con cui differenziare il comportamento dei vari worker. Implementa i seguenti metodi (interfacce escluse):

solve Inizia la computazione; in particolare:

- calcola le partizioni;
- contatta i vari nodi di computazione, assegnandogli un insieme scelto dalla partizione e la specifica del dominio;
- rimane in attesa delle notifiche da parte dei vari *Worker*, in modo da poter (eventualmente) ripartizionare e ridistribuire il carico sullo spazio delle soluzioni più interessante;
- salva in *SolutionManager* le migliori soluzioni trovate.

Ha i seguenti parametri:

Nome	Semantica
systemDomain	La stringa relativa alla coppia (dominio, strategie algoritmiche) (vedi il paragrafo relativo nella sezione 4)
dislocations	Il path del file in cui è contenuta la descrizione della dislocazione dei vari nodi attivi da usare per il calcolo
modelData	Il path del file in cui sono contenuti i dati relativi alla modellazione del dominio in considerazione (nel caso di VBDS, la struttura del grafo)

kill Interrompe la computazione distribuita (dunque nel nodo corrente ed in tutti quelli specificati in dislocations).

4.2.1.2 Controller

Interfaccia fornita da *ConcreteController* e *Partitioner* e richiesta dal sottosistema *Core*. Specifica i metodi per la notifica asincrona delle operazioni fornite da *Core*: per tal motivo, tutte le funzioni fornite avranno un parametro node che specificheranno qual è il nodo computazionale che sta effettuando la notifica.

Fornisce i seguenti metodi:

newSolution Notifica l'aver trovato una nuova soluzione.

columnsGenerated Indica che sono state generate delle colonne; ha i seguenti parametri:

Nome	Semantica
workerId	Indica l'id del worker che si è stabilizzato

4.2.1.3 Worker

Indica le funzionalità offerte dal sottosistema *Core*.

Offre due insiemi di funzioni: la prima definisce le operazioni algoritmiche offerte, la seconda permette di ottenere lo stato della computazione.

Le funzioni del primo insieme, eccetto la *kill*, hanno tutte i seguenti parametri:

Nome	Semantica
toNotify	Il controller da notificare
partition	Specifica il sottoproblema indotto dal partizionamento

Fornisce i seguenti metodi:

solve Trova delle soluzioni al sottoproblema indotto dal partizionamento.

generate Effettua una generazione di colonne.

pruneAndSolve Annulla il calcolo di *solve* richiesto precedentemente e risolve il problema specificato dal nuovo partizionamento.

getWorker Ottiene lo stato del worker specificato. Può innalzare un'eccezione nel caso in cui questo non sia disponibile.

kill Interrompi la computazione svolta dal sottosistema

4.2.1.4 SolutionManager

Memorizza le soluzioni intere trovate sino a quel momento.

Fornisce le seguenti funzioni:

storeSolution Memorizza una soluzione intera; prendendo come parametro il worker di cui si vuol memorizzare la soluzione.

bestSolution Restituisce la migliore soluzione trovata fino a quel momento.

latestSolution Restituisce l'ultima soluzione trovata fino a quel momento.

4.2.1.5 Partitioner

Come illustrato nella sezione 4.1, Partitioner è una classe astratta le cui classi derivate si occupano del calcolo delle suddivisione dello spazio delle soluzioni.

Partitioner può implementare Controller e richiedere Worker e dunque interagire con il sottosistema *Core*, in quanto la partizione può essere guidata da alcune informazioni preliminari (come un insieme di alcune colonne generate, una soluzione intera, l'analisi della struttura dati utilizzata per la generazione).

Offre le seguenti funzioni:

getPartition Ottiene le partizioni, espresse come un insieme di *Partition*, a partire dal numero di partizioni desiderate.

getInstance Ottiene una istanza del partizionatore, passando come parametro il modello scelto (vedi 4.1).

4.2.2 Sottosistema Core

Nella figura 4.2.2 è illustrato il sottosistema Core, la cui responsabilità è quella di risolvere una restrizione del problema originale, specificato dalla classe Partition; è compito del sottosistema Core, dunque, effettuare il ciclo di stabilizzazione, il fixing e l'unfixing dell'insieme dei worker gestiti.

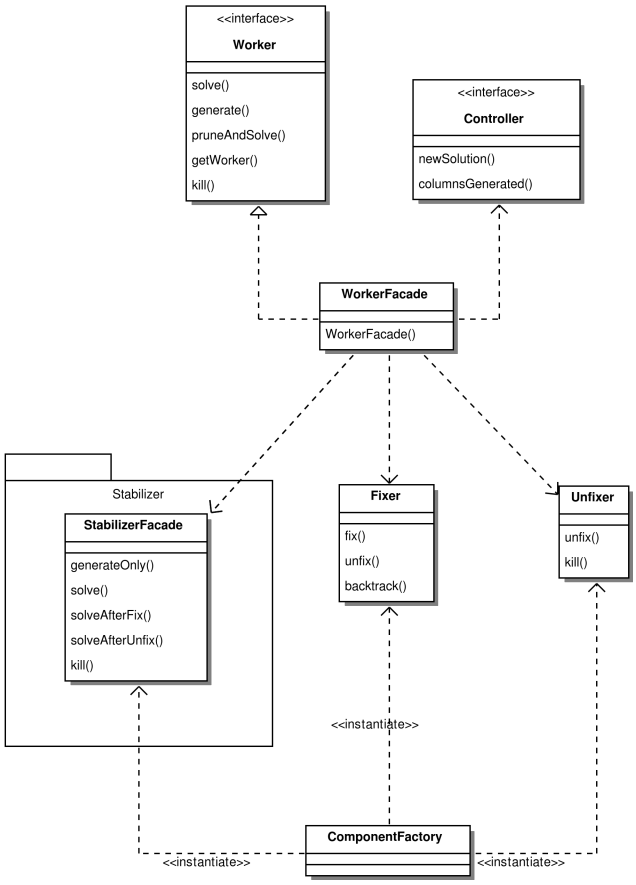

Figura 4.2: Descrizione del sottosistema Core

Il sottosistema Core, come già accennato, implementa l'interfaccia Worker e richiede Controller; poiché WorkerFaçade è la classe façade del sottosistema, tali interfacce saranno implementate e richieste da tale classe.

Segue una panoramica delle classi.

4.2.2.1 WorkerFacade

E' la classe façade della componente, nonché la classe mediatrice tra i vari sottosistemi del sottosistema.

La classe ha le seguenti relazioni:

Con chi?	Perché?
Stabilizer	Utilizzato per effettuare il ciclo di stabilizzazione
Fixer/Unfixer	Utilizzato per effettuare il fissaggio e sfissaggio su un worker

Fornisce inoltre il costruttore *MainControl*, in cui un parametro, systemDomain, è il modello scelto (vedi 4.1).

4.2.2.2 WorkerData

WorkerData definisce l'insieme dei dati che tengono traccia del comportamento di un *Worker*; un WorkerData pertanto avrà i seguenti dati associati:

- il manager delle colonne;
- il manager delle righe;
- il manager delle duali;
- la partizione associata.

4.2.2.3 ComponentFactory

La sua responsabilità è quella di istanziare i sottosistemi Stabilizer, Fixer, Unfixer, a partire dal modello (parametro model), tramite le funzioni getFixer, getUnfixer, getStabilizer.

4.2.2.4 Fixer

Sottosistema che effettua il fissaggio delle colonne; la sua responsabilità è identica a quella del "Fixer di primo livello" attuale.

Espone le seguenti funzioni:

fix Effettua il fissaggio delle colonne.

backtrack Effettua il backtrack (vedi 2.2.1).

kill Termina l'esecuzione corrente del fixer.

4.2.2.5 Unfixer

Sottosistema che effettua lo sfissaggio delle colonne; la sua responsabilità è identica a quella del "Fixer di secondo livello" attuale.

Espone le seguenti funzioni:

unfix Effettua lo sfissaggio di alcune colonne

kill Termina l'esecuzione corrente dell'unfixer.

4.2.2.6 StabilizerFacade

Classe façade del componente Stabilizer, effettua il ciclo di stabilizzazione; il sottosistema è trattato nella sezione 4.2.2.6.

4.2.3 Sottosistema Stabilizer

Il sottosistema stabilizer è responsabile dell'implementazione del ciclo descritto nella sezione 1.3; per tal motivo e per i requisiti 4 e 5 della sezione 3.2.1, si hanno le seguenti funzioni, che prendono come parametro il WorkerData:

- solve, che implementa i passi necessari per la prima stabilizzazione (cioè senza aver ancora effettuato alcun fixing né unfixing);
- solveAfterFix, che implementa i passi necessari per la stabilizzazione dopo un fissaggio;
- solveAfterUnfix, che implementa i passi necessari per la stabilizzazione dopo uno sfissaggio.

In particolare:

- Poiché all'inizio non si disporrà di alcuna soluzione duale, lo stabilizzatore all'invocazione di solve() dovrà prima effettuare una generazione di colonne guidata dai costi sugli archi e non dai costi ridotti (si deve cioè fissare il vettore λ al vettore nullo);
- Al termine di un fissaggio si avrà una diversa formulazione del Master Problem, dunque sarà necessario cominciare la stabilizzazione con il solver e non con il generatore per ottenere una soluzione duale significativa per la generazione;
- Al termine di uno sfissaggio sarà necessario cominciare il ciclo di stabilizzazione dal generatore, in quanto si dovrà estendere B in modo da poter avere delle colonne che coprano i task che erano coperti da qualche colonna precedentemente fissata; è necessario, inoltre, aggiornare il vettore λ in modo da non escludere più i succitati task.

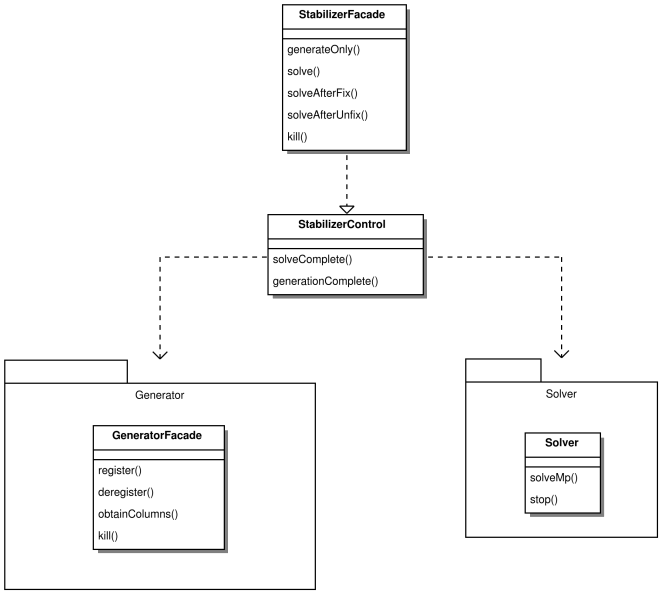

Figura 4.3: Descrizione del sottosistema Stabilizer

Poiché il sottosistema Stabilizer include il generatore, oltre alla consueta funzione kill vi è anche una funzione generateOnly che effettua solo la generazione di colonne.

Il sottosistema è illustrato nella figura 4.3.

Sincronizzazione tra generatore e solver Poiché il generatore ed il solver dovrebbero poter eseguire in parallelo, si è affrontato il problema della sincronizzazione tra le due componenti. Ricordando che il solver può ricevere le colonne a costo ridotto negativo prima che il generatore abbia terminato la sua computazione, si è affrontato il problema di:

1. far avere le nuove colonne generate al solver;

2. far sapere al solver che può terminare il calcolo senza dover attendere nuove colonne dal generatore

Per il problema 1 si è deciso di aggiungere direttamente le colonne al manager delle colonne, permettendo al solver di accedere alle nuove colonne generate tramite l'interfaccia Updatable.

Per il problema 2 si è deciso di notificare esplicitamente la terminazione del generatore alla classe mediatore, *StabilizerControl* (vedi 4.2.3.1), tramite la member function generationComplete. Poiché potrebbe accadere che la notifica della terminazione del generatore sia raccolta prima dell'aggiunta delle ultime colonne generate, alla funzione viene passato anche il numero di colonne complessivamente generate, in modo che il solver possa sapere se è necessario o meno attendere per l'aggiunta di altre colonne al manager delle colonne.

4.2.3.1 StabilizerControl

StabilizerControl è la classe façade del sottosistema Stabilizer, nonché la classe mediatrice delle interazioni tra Solver e Generator. La sua responsabilità è analoga a quella dell'attuale MainModule.

Implementa l'interfaccia StabilizerControl, che permette la notifica, da parte di Solver e Generator, dei progressi effettuati dai due sottosistemi.

4.2.3.2 StabilizerControl

Fornisce i metodi per la notifica asincrona, da parte del solver e del generatore, dello stato della computazione:

- generationComplete: indica il termine della generazione di colonna;
- solveComplete: indica il termine della soluzione del MasterProblem.

generationComplete ha come argomento il numero di colonne generate; l'utilità di tale parametro è chiarita nel paragrafo 4.2.3.

4.2.3.3 Solver

Il sottosistema Solver risolve il Master Problem; la responsabilità di tale classe è pertanto del tutto simile a quella del MasterProblemSolver; è implementato con funzioni asincrone e prevedendo la possibilità che il generatore possa eseguire in parallelo rispetto al solver.

solveMP Risolve il master problem, incrementando la rappresentazione con le colonne aggiunte via via al manager delle colonne. Per terminare la computazione è necessario invocare la funzione stop.

stop Notifica al solver il completamento del generatore. Ha come parametro il numero di colonne generate, che permette al metodo solveMP di determinare se abbia calcolato o meno la soluzione del problema tenendo in considerazione tutte le colonne generate sino a quel momento.

4.2.3.4 GeneratorFacade

Classe façade del sottosistema Generator, trattato nella sezione 4.2.3.4.

4.2.4 Generator

Il sottosistema Generator implementa il generatore di colonne (vedi 1.4); il diagramma delle classi del sottosistema è illustrato nella figura 4.4.

Il sottosistema prevede delle classi domain-specific ed altre generali rispetto al dominio scelto: nella figura 4.4 le prime sono indicate come astratte (nome in corsivo), mentre le altre sono classi concrete.

Si ricorda che il sottosistema è istanziato da ComponentFactory, tramite la funzione *getStabilizer*.

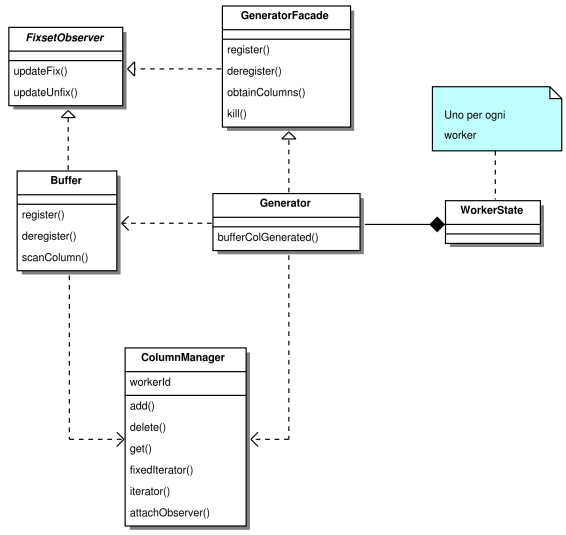

Figura 4.4: Descrizione del sottosistema Generator

4.2.4.1 GeneratorFaçade

E' la classe façade del sottosistema: espone le funzioni implementate dalla classe *Generator* e dalla classe *Buffer* (meno Buffer::scanColumn, utilizzata direttamente dal Generator, e *kill*, che invoca le kill del Buffer e del Generator); per la loro specifica si rimanda alle sezioni 4.2.4.1 e 4.2.4.2.

4.2.4.2 Generator

Il ruolo del Generator è quello di rifornire il gestore delle colonne del worker con delle colonne a costo ridotto negativo rispetto alla soluzione duale corrente; poiché è specifica per il dominio, è una classe astratta.

Come descritto nella sezione 4, Generator utilizza dei buffer delle colonne per motivi di riutilizzo; per tal motivo è in relazione con la classe *Buffer*, a cui richiede delle colonne con costo ridotto negativo all'invocazione della funzione *Generator::obtainColumns*

Oltre alla richiesta al buffer, Generator può effettuare anche una generazione di colonne, avvalendosi di un threadpool per soddisfare le varie richieste da parte dei worker e per avvalersi di un possibile parallelismo all'interno dell'algoritmo di generazione (presente, anche se non sfruttato, nell'attuale implementazione del turn scheduling), soddisfando il requisito 5 della sezione 3.3. La classe, inoltre, si premura di notificare la classe mediatrice dello Stabilizer nel caso in cui la generazione sia terminata.

Come descritto nel relativo paragrafo nella sezione 4, il generatore può mantenere delle informazioni relative ad ogni worker: tali informazioni sono raccolte nella classe di dominio *WorkerState*, associata al worker tramite registrazione; per tal motivo, al momento dell'eliminazione di un worker, è necessario deregistrare il worker in modo da poter liberare le risorse.

Le funzioni che effettuano la generazione di colonne sono astratte: per altri dettagli sulla dinamica precisa della generazione è necessario far riferimento alla classe derivata specifica per il modello considerato.

Tutte le funzioni, eccetto la *kill*, prendono come parametro il worker su cui è necessario effettuare la generazione:

register Effettua la registrazione del worker considerato, allocando la struttura dati WorkerState ed effettuando la registrazione anche ai Buffer. Qualsiasi worker, prima di poter usufruire dei servizi offerti dal Generator, deve essere registrato al generatore.

deregister Effettua la deregistrazione del worker considerato, liberando dalla memoria la struttura dati WorkerState associata ed effettuando la deregistrazione.e anche ai buffer. Dopo la deregistrazione non sarà più possibile usufruire dei servizi offerti dal Generator, a meno di rieffettuare la registrazione.

obtainColumns Inserisce nel gestore delle colonne del worker passato come argomento delle colonne a costo ridotto; le garanzie sulla qualità e quanti-

tà di colonne trovate dipendono dalla particolare euristica implementata dal generatore e dallo stato del buffer delle colonne. E' possibile reperire maggiori dettagli sul processo di generazione nell'introduzione di questa sottosezione o nel relativo paragrafo della sezione 4.

kill Termina la computazione svolta dal Generator.

4.2.4.3 Buffer

Il buffer, come già accennato nel relativo paragrafo della sezione 4, mantiene un sottoinsieme B delle colonne generate da Generator, rendendole disponibili anche per richieste successive, in modo da poter garantire un certo riuso della computazione.

È possibile avere più buffer presenti nel sistema, al fine di aumentare la capacità di bufferizzazione distribuendoli su più nodi.

Il buffer comunica le colonne generate ad ogni buffer; ogni buffer può dunque aggiungere o meno la colonna al proprio pool, a seconda della strategia utilizzata per implementare la distribuzione delle colonne. Un modo per distribuire le colonne può consistere nell'associare un id ad ogni buffer e nel calcolare, per ogni colonna trasmessa al buffer, un intero da 0 a n, dove n è il numero di buffer presenti nel sistema: sia id_{buf} l'id del buffer e $hash : Column \rightarrow [0, n)$: allora la colonna è aggiunta al pool se e solo se $hash\,(column) = id_{buf}$.

Il buffer aggiunge le colonne richieste al manager delle colonne del worker specificato. Poiché l'implementazione del fixer presuppone che non vengano aggiunte al manager delle colonne delle colonne presenti nel conflict set di una delle colonne già fissate (vedi 2.1), il buffer prevede la registrazione dei worker e la notifica delle colonne fissate, implementando l'interfaccia FixsetObserver.

La classe Buffer dovrà anche gestire il consumo di memoria, eliminando dal sottoinsieme delle colonne gestite delle colonne scelte secondo qualche euristica.

Segue una descrizione delle funzioni offerte.

register Effettua la registrazione del worker.

deregister Effettua la deregistrazione del worker, liberando le risorse associate. Nel caso in cui il worker non sia registrato, innalza una NotRegisteredException.

scanColumn Richiedi delle colonne a costo ridotto negativo per il Worker passato come argomento. Aggiunge le colonne al manager delle colonne del worker specificato.

addColumn Aggiunge al Buffer la colonna passata come argomento.

kill Interrompi tutte le computazioni svolte dal servizio e disalloca i thread associati.

4.2.5 ColumnManager

Il design del manager delle colonne ha previsto alcuni accorgimenti che meritano una discussione più ampia ed un package a lui riservato, mostrato.nella figura 4.2.5.

La responsabilità della componente è quella di mantenere e rendere disponibili l'insieme delle colonne associate ad un worker e le loro proprietà, oltre a notificare le altre componenti dei cambiamenti avvenuti nello stato espresso dalla componente.

La componente prevede un container chiamato *ColumnManager*, a cui è possibile aggiungere colonne e ottenere degli oggetti *CColumn*: ogni CColumn è associato ad una colonna ed estende le informazioni di quest'ultima associandogli informazioni specifiche del worker, come il fissaggio o il costo ridotto.

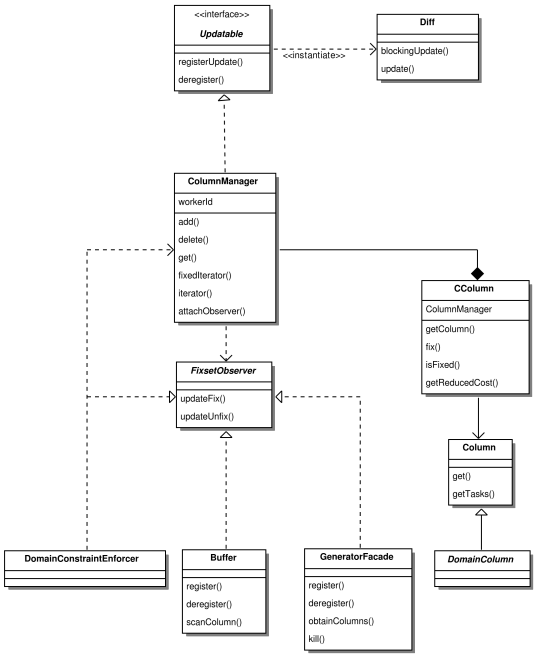

Figura 4.5: Diagramma delle classi del manager delle colonne

È prevista inoltre la possibilità di ottenere aggiornamenti sullo stato del *ColumnManager*, tramite l'interfaccia *Updatable*, assolutamente analoga all'interfaccia attuale *Clock* (vedi 2.2.1): è possibile registrarsi e deregistrarsi per conoscere i cambiamenti nello stato del ColumnManager avvenuti in un certo lasso di tempo (requisito 12 della sezione 3.2.1) e sono previsti due metodi per ottenere gli aggiornamenti: uno, *update*, analogo al *clock* attuale e l'altro, *blockingUpdate*, che blocca il thread richiedente fino a quando non vi è almeno un aggiornamento.

Per la notifica delle colonne fissate è stato utilizzato l'Observer pattern, in modo analogo a quanto implementato finora, con la differenza che gli oggetti observer otterranno, oltre alla colonna fissata, anche l'identificatore del worker associato al manager delle colonne.

Segue una panoramica sulle classi del package:

4.2.5.1 ColumnManager

Oltre alle funzioni presenti nell'interfaccia implementata Updatable (vedi 4.2.5.2), queste le funzioni offerte dalla classe:

add Aggiunge la colonna passata come argomento, restituendo la CColumn associata.

delete Rimuove la colonna passata come argomento.

get Ritorna la CColumn corrispondente alla colonna passata come argomento sse questa è presente nel ColumnManager.

iterator Restituisce l'iteratore di tutte le colonne presenti nel ColumnManager

fixedIterator Restituisce l'iteratore delle colonne bloccate.

attachObserver Permette di aggiungere un Observer (di tipo FixsetObserver) alla lista degli oggetti che devono essere notificati nel caso in cui una colonna sia stata fissata o sfissata.

4.2.5.2 Column, DomainColumn e CColumn

Column è la rappresentazione di un vettore sparso, in modo del tutto simile alla classe Colonna del design precedente. Column viene poi derivata da DomainColumn, che arricchisce la colonna con le informazioni di dominio necessarie per, ad esempio, poter risalire da una soluzione alla rotta relativa.

CColumn è l'associazione tra la colonna ed il Worker: in questa classe sono pertanto presenti attributi e funzioni per ottenere informazioni sulla colonna relative al worker associato (come la funzione per ricavarsi il costo ridotto relativo).

Column prevede l'operatore [] per l'ottenimento del valore di una sua componente e la funzione getTasks per ottenere la lista dei task coperti dalla colonna.

CColumn prevede invece le seguenti funzioni:

getColumn Ottiene la Column associata.

fix Imposta la colonna come fissata o meno. Nel caso in cui lo stato sia cambiato a seguito dell'invocazione, verranno notificati anche gli observer dell'insieme delle colonne fissate registrai al manager delle colonne (vedi ...).

isFixed Restituisci true sse la colonna è fissata.

reducedCost Ottiene il costo ridotto della colonna rispetto al worker a cui è associata la colonna.

4.2.5.3 Updatable

In modo del tutto simile all'interfaccia Clock del design precedente, permette di conoscere le modifiche effettuate allo stato del manager delle colonne effettuando una sottoscrizione al servizio, che restituirà un oggetto di tipo Diff. Tale oggetto potrà essere utilizzato per conoscere le differenze allo stato del manager delle colonne avvenute tra l'ultima invocazione della funzione esposta dal Diff (oppure dalla sua creazione) fino al momento in cui è stata effettuata la richiesta. Il servizio viene erogato finché non viene deregistrato l'oggetto di tipo Diff.

Queste le funzioni esposte dall'interfaccia SubscribeUpdate:

- register: effettua la registrazione al servizio, restituendo un oggetto di tipo Diff.
- deregister: annulla la registrazione al servizio rappresentata dall'oggetto di tipo Diff passato come argomento.

Diff::update restituisce una ennupla di coppie (op, x), dove op è l'operazione svolta (fissaggio, cancellazione, bloccaggio...) svolto sulla colonna x.

Diff::blockingUpdate blocca il chiamante fino a quando non è presente almeno un aggiornamento (fino a quando cioè l'ennupla da restituire non ha lunghezza nulla).

4.2.5.4 Notifica delle colonne fissate e vincoli nonlineari

I vincoli nonlineari con informazioni di dominio (vedi il requisito 16 della sezione 3.2.1) sono gestiti utilizzando il design pattern observer sull'insieme delle colonne fissate, in modo simile a come avviene nel design precedente. ColumnManager espone la funzione attachObserver, tramite cui è possibile aggiungere degli elementi all'insieme degli observer interessati alle notifiche;

gli observer devono essere classi derivate di FixsetObserver, di cui NonlinearConstraint è classe derivata, che fornisce una coppia di funzioni, una per notificare il fix, l'altra l'unfix, con parametri la colonna ed worker interessati.

La differenza principale consiste nell'aver spostato la logica del trattamento dei vincoli nonlineari dal generatore ad una classe di dominio apposita, NonlinearConstraint, e nell'aver spostato la notifica della colonna fissata dal fixer al ColumnManager.

I dettagli su come vengano calcolate le colonne del conflict set c_i e di come la struttura dati utilizzata per la generazione venga modificata al fine di non generare più quelle colonne è dipendente interamente dal dominio, dunque non sarà trattato in questa sede.

4.3 Diagrammi di sequenza

Di seguito verranno esposti dei diagrammi di sequenza, utilizzando il formalismo UML 2.0, che illustreranno il comportamento del sistema nel calcolo di una soluzione intera, a diversi livelli di dettaglio. Per una introduzione al formalismo adottato, si può consultare tra le tante risorse [4].

I diagrammi di sequenza allegati al seguente documento sono le immagini 4.3, 4.3, 4.3, 4.3, 4.3,

I casi d'uso sono stati allegati per rendere più concreta la discussione svolta nelle sezioni precedenti e per esplicitare i messaggi scambiati tra le varie classi in relazione all'operazione che si vuole compiere.

Nel caso in cui il diagramma di sequenza coinvolga una classe di dominio (come ad esempio il Partitioner o il Generator), allora le interazioni sono da intendersi esclusivamente indicative, in quanto tali componenti nel design sono solo astratte e dovranno essere derivate da una sottoclasse concreta.

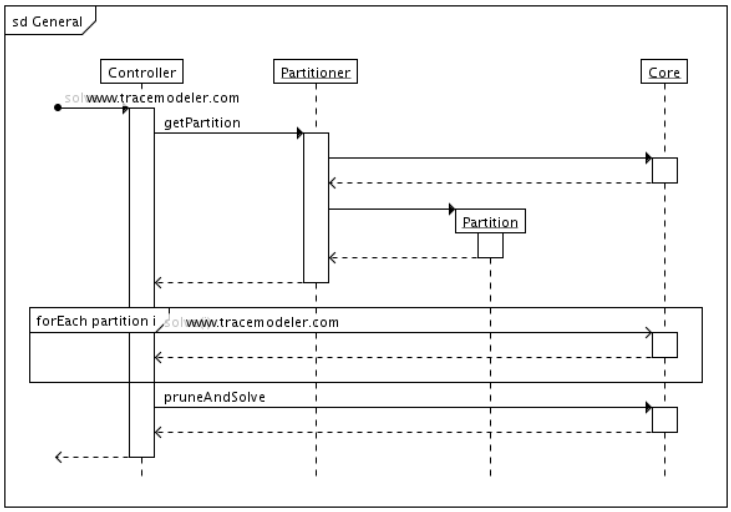

Figura 4.6: Calcolo della soluzione intera utilizzando Core

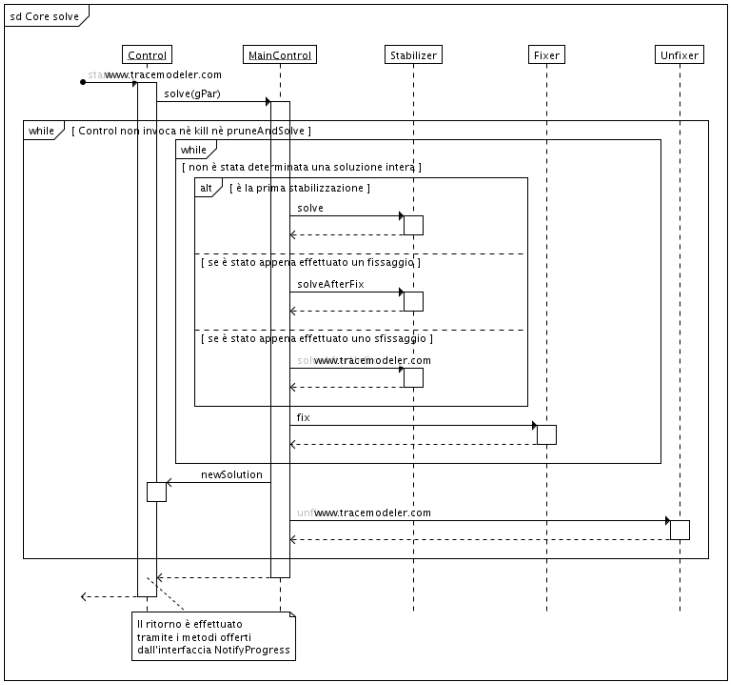

Figura 4.7: Calcolo della soluzione intera da parte di Core

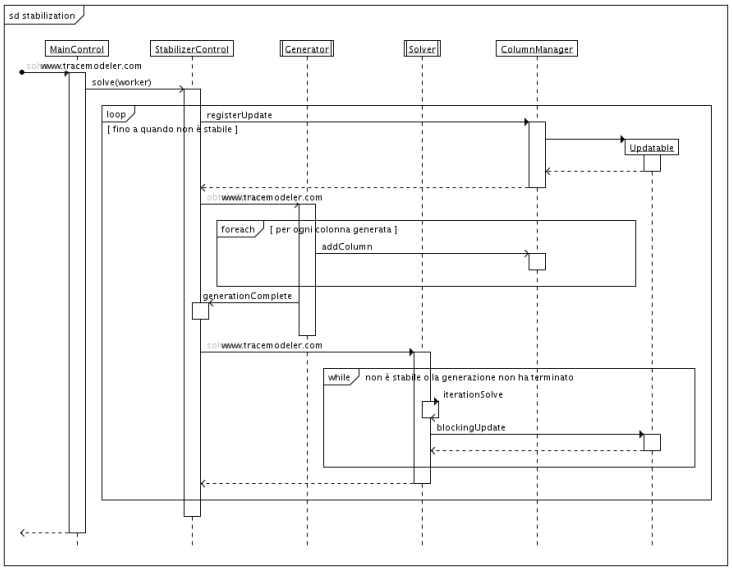

Figura 4.8: Diagramma di sequenza che illustra la stabilizzazione

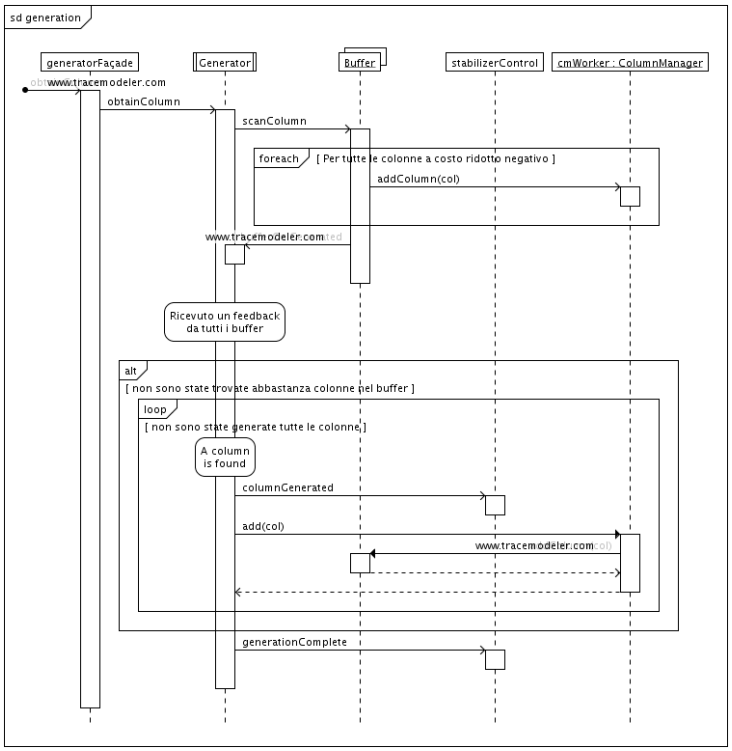

Figura 4.9: Esempio di possibile interazione della classe Generator

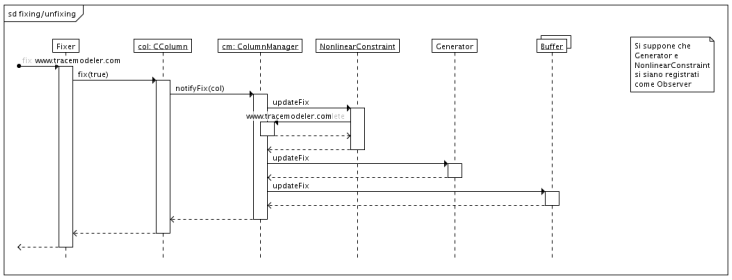

Figura 4.10: Fissaggio delle colonne

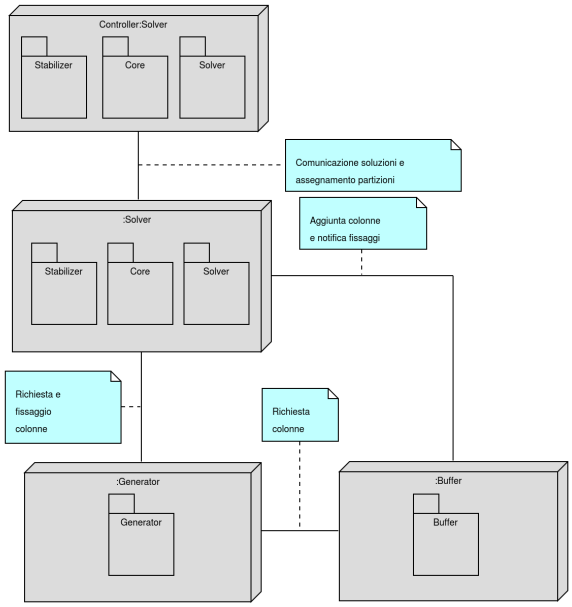

Figura 4.11: Distribuzione degli oggetti in processi

4.4 Dislocazione delle componenti nei nodi e modalità di comunicazione

4.4.1 Suddivisione in processi

Al fine di potersi approfittare di un ambiente distribuito, si è deciso di adottare un modello a processi cooperanti, in cui istanze delle componenti possono essere allocate in processi differenti.

Si è deciso dunque di associare classi a processi, individuando tre classi di processi (figura 4.4.1):

- processo “solver”;
- processo “generator”;

- processo “buffer”.

Al processo solver sono associati il Controller, il Partitioner il Fixer, l'Unfixer, il Solver ed il gestore delle colonne; al processo generator il generatore di colonne ed al processo buffer il buffer delle colonne.

Considerando il requisito di minimizzazione di messaggi extra-processo e la massimizzazione del grado di parallelismo, il rationale dietro questa scelta è riassumibile in questi punti:

- non ha senso associare un processo dedicato al fixer, l'unfixer, il controller ed il partitioner, in quanto queste componenti hanno un basso costo computazionale e perché non c'è alcun modo di eseguirle in parallelo rispetto al solver/generatore; inoltre, tali componenti devono iterare sulle colonne appartenente ad un worker, dunque è conveniente approfittarsi della presenza, nello stesso spazio di indirizzamento, del manager delle colonne;

- il solver ha bisogno del manager delle colonne, dunque deve essere posto nello stesso processo in cui questo è presente, a meno di voler replicare e sincronizzare i dati su processi;

- il generatore ed il solver devono girare su processi differenti, in modo da poter soddisfare il requisiti 3 della sezione 3.3;

- il buffer dovrebbe essere localizzato su un processo a sé stante, in modo da poter garantire bassa latenza ed una elevata capacità di memorizzazione.

Le comunicazioni tra oggetti localizzati in processi distinti avverrà tramite MPI; poiché la libreria supporta comunicazioni tra processi e non tra oggetti, ogni processo avrà un oggetto responsabile del dispatch dei messaggi ai vari oggetti (d'ora in poi DispatchMPI).

4.4.2 Modifiche al design per il caso sequenziale

Poiché alcuni oggetti saranno localizzati in spazi d'indirizzamento differenti, sarà necessario apporre delle modifiche al design presentato nella sezione 4.1; le idee principali sono le seguenti:

- passaggio di parametri MPI al posto di riferimenti;
- utilizzo di classi stub per l'implementazione dell'RPC.

Passaggio di riferimenti MPI Nel caso in cui si abbia una situazione di questo tipo:

```
void A::doSomething(B *obj)
```

e i seguenti requisiti siano verificati:

- un processo può avere al più una istanza di B (è un singleton);
- i metodi offerti da B e utilizzati da A restituiscono void;
- le istanze di A e B sono localizzate in spazi d'indirizzamento distinti

allora è possibile adattare il design seguendo questi passi:

1. associare ad ogni funzione offerta da B ed utilizzata da A un tag MPI e per A::doSomething;
2. estendere DispatchMPI in modo da trattare i nuovi tag definiti;
3. sostituire il riferimento a B con il rank del processo che ha inviato il messaggio di richiesta MPI;
4. sostituire le invocazioni di B con il passaggio di messaggi.

È possibile applicare questi passi nei seguenti casi:

1. registrazione al generatore di sé stesso da parte di StabilizerControl;

2. registrazione al buffer dello StabilizerControl da parte del generatore.

Nel caso 1 si ha dunque l'aggiunta dei seguenti tag MPI:

- addColumn, utilizzato per aggiungere colonne al ColumnManager;
- notifyEndGeneration, utilizzato per notificare il termine della generazione alla classe mediatrice StabilizerControl;
- registerCG e deregisterCG, rispettivamente per la registrazione e deregistrazione dei worker;

ed il passaggio del rank del processo solver che vuole effettuare la registrazione nelle member function di registrazione e deregistrazione.

Il caso della registrazione al buffer è analogo.

Utilizzo classi stub per l'RPC Per disaccoppiare i dettagli di comunicazione tra oggetti dalla business logic è possibile utilizzare delle classi stub, cioè delle classi che estendono l'interfaccia offerta da una componente in cui l'implementazione dei metodi consiste nell'invocazione tramite MPI dei metodi corrispondenti della classe "concreta" che implementa quell'interfaccia.

Ad esempio, poiché vi è solo una istanza (chiamiamola i), nell'intera applicazione distribuita, del Controller, il WorkerFacade presente nello stesso processo solver in cui è presente i dovrà invocarne direttamente i metodi, mentre gli altri dovranno invocare i remotamente tramite messaggi; a tal fine è possibile dare due implementazioni di Controller: un ConcreteController, che implementa i metodi descritti dall'interfaccia, ed uno StubController, che invoca i metodi di ConcreteController tramite MPI; a quel punto il WorkerFacade potrà ottenere l'istanza del Controller tramite qualche meccanismo di Inversion of Control, che gli farà avere un'istanza della classe concreta o della classe stub a seconda che i sia nello stesso spazio di indirizzamento dell'oggetto o meno.

4.5 Tecnologie utilizzate

Il nuovo design introdurrà due tecnologie, funzionali all'introduzione del paradigma parallelismo e distribuito all'interno della componente algoritmica SetPartLib: Intel Threading Building Block e MPI.

In questa sezione si vedrà una breve introduzione alle due tecnologie e si giustificherà la loro introduzione nel progetto.

4.5.1 Intel TBB

Cos'è Intel Threading Building Blocks (d'ora in poi TBB) è una libreria in C++ sviluppata dalla Intel per scrivere programmi che possono avvantaggiarsi di sistemi multiprocessore. La libreria consiste di strutture dati e algoritmi che permettono al programmatore di evitare certe complicazioni risultati dall'utilizzo di librerie per la programmazione multithread, come POSIX o Boost Threads, in cui ogni singolo thread è creato, sincronizzato e terminato manualmente.

La libreria adotta un approccio alternativo e più ad "alto livello", permettendo alle operazioni di essere trattati come dei "task", allocati ad ogni processore dinamicamente e automatizzando l'utilizzo efficiente della cache. Tale approccio introduce TBB in una famiglia di soluzioni per la programmazione parallela il cui obiettivo è quello di disaccoppiare la programmazione dai particolari della macchina sottostante.[1]

Cosa offre Intel TBB offre diverse astrazioni al programmatore che permettono di esprimere in modo esplicito determinati pattern di programmazione:

1. permette di definire dei *task*, cioè delle operazioni che possono essere eseguite da una unità di esecuzione: in tal modo è possibile disaccoppiare la logica dell'operazione rispetto alle operazioni di creazione e

[1] Tradotto ed adattato da [5].

gestione dei thread in cui devono essere eseguiti (dettagli che verranno gestiti dalla libreria);

2. offre dei costrutti di programmazione parallela generici, come *parallel_for*, *parallel_reduce*, *parallel_scan, pipeline* (implementa una pipeline di stadi algoritmici, ognuno implementato da un task); i vantaggi offerti in tal caso riguardano l'efficienza dell'implementazione, la correttezza e la velocità nella stesura del codice;

3. offre dei container polimorfi, thread-safe ed altamente ottimizzati: *concurrent_queue, concurrent_vector, concurrent_hash_map*;

4. offre un allocatore pensato esplicitamente per essere utilizzato in un ambiente multithread;

5. offre una classe di oggetti portabili e necessari per il garantire la mutua esclusione, come vari tipi di mutex e wrapper per garantire l'accesso atomico ai dati;

6. offre un *task scheduler*, cioè un oggetto capace di schedulare e mappare i task in thread, in modo da garantire un utilizzo efficiente delle risorse della macchina sottostante.

Perché è utile L'utilità dei punti 3, 4, 5 è chiara, in quanto sono tutti oggetti richiesti tipicamente da una applicazione concorrente e Intel TBB ne fornisce una implementazione portabile.

L'utilità del punto 6 è chiara se si osserva il punto 1: al posto di dover implementare da zero dei threadpool per ogni servizio implementato (Generator, Solver, Fixer...) ed occuparsi di dettagli come la distribuzione dei thread all'interno di tali servizi per ottimizzare le performance, è possibile definire un Generator come un *task*, istanziarne uno ogniqualvolta viene effettuata una richiesta e passarlo al *task scheduler*, che si occuperà in modo automatico dei dettagli che possono influenzare in modo significativo le

performance, come l'oversubscription ed il load imbalance. Questa non è la sede adatta per illustrare come la libreria riesca ad ottenere tali obiettivi: il lettore interessato potrà consultare a proposito [6].

4.5.2 MPI

Cos'è MPI, acronimo per *Message Passing Interface*, è "una API per il passaggio di messaggi, assieme ad un protocollo e ad una specifica semantica riguardo come le sue funzionalità debbano comportarsi in qualsiasi implementazione" ([7]).

MPI è lo standard *de facto* per la comunicazione tra processi che modellano un programma parallelo eseguito in un sistema a memoria distribuita (ad esempio, un cluster); è uno standard *language-agnostic*, in quanto prevede una Language Independent Specification (specifica indipendente dal linguaggio) per la chiamata di funzioni ed i binding con i vari linguaggi ed è assolutamente complementare rispetto ad altre tecnologie, come Intel TBB.

Sono disponibili varie implementazioni, sia opensouce (come OpenMPI) che proprietarie (come Intel MPI).

Cosa offre MPI offre i seguenti concetti[2]:

1. communicators, oggetti che connettono gruppi di processi nella sessione MPI; è supportata la comunicazione intragruppo ed intergruppo;

2. point-to-point, comunicazione tra due specifici processi; sono presenti funzioni sia per la comunicazione bloccante che per la comunicazione non bloccante;

3. funzioni collettive: un nodo invia un messaggio ad un insieme di processi (ad esempio, tramite MPI_Bcast) e processa le risposte tramite delle operazioni predisposte dallo standard (come MPI_Reduce);

[2]Lista adattata e tradotta da [8]

Perché è utile L'utilità della libreria è stata già esplorata nella sezione 4.3.

4.6 Analisi dei rischi

Durante la realizzazione del progetto bisognerà porre attenzione ai seguenti aspetti, che potrebbero comprometterne la validità:

Buona scelta del partizionamento Una buona scelta del partizionamento è essenziale affinché il comportamento dei vari worker sia abbastanza differenziato tra loro: non lo fosse, ogni worker convergerebbe a soluzioni simili tra loro, vanificando di fatto l'idea dell'introduzione dei worker come modalità alternativa allo sfissaggio nell'esplorazione dello spazio delle soluzioni.

Rallentamento per via delle operazioni di sincronizzazione e comunicazione Poiché si adotterà anche un modello a memoria condivisa, sarà necessario curare l'accesso alle strutture condivise in modo che le operazioni di sincronizzazione sia minimizzate. Il rischio è che tali operazioni potrebbero rallentare la computazione a tal punto da rendere la soluzione equivalente ad una soluzione sequenziale per via dell'ordinamento indotto dalle operazioni di sincronizzazione.

A tal fine, è possibile effettuare una rozza analisi del costo del ciclo di stabilizzazione nel caso parallelo e nel caso sequenziale:

Analisi del costo di stabilizzazione Il generatore invia periodicamente, durante la computazione, dei messaggi in cui aggiorna il solver con le colonne a costo ridotto generate; il confronto tra il costo sequenziale ed il costo con "interazione" può essere fatto, rozzamente, così

$$t_{seq} = t_{gen} + t_{solve}$$

$$t_{par} = t_{gen} + t_{lastmessage} + t_{lastsolve}$$

dove:

- t_{seq} è il costo di una iterazione nel caso sequenziale;
- t_{par} è il costo di una iterazione nel caso parallelo;
- t_{gen} è il costo della generazione;
- t_{solve} è il costo della risoluzione del Master Problem;
- $t_{lastmessage}$ è il costo complessivo dell'invio dell'ultimo update da parte del generatore;
- $t_{lastsolve}$ è il costo della computazione richiesta dal solver all'ultimo aggiornamento.

la cosa dunque è vantaggiosa se:

1. t_{solve} è un addendo significativo rispetto a t_{seq}
2. $t_{lastmessage} + t_{lastsolve}$ è "abbastanza" inferiore a t_{solve}

La 1 si sa che è vera, dato che il solver richiede pressappoco il 20% del tempo richiesto dall'iterazione, e con le possibilità di parallelismo del generatore si potrebbe anche pensare di arrivare al 30-40% (dimezzando il tempo richiesto dal generatore con una macchina con grado di parallelismo soddisfacente). La 2 potrebbe essere vera nel caso in cui il solver riesca ad avvantaggiarsi, nel calcolo della soluzione, dagli aggiornamenti precedenti sulle colonne inserite. Il costo del messaggio è quello della latenza su ethernet e del tempo richiesto per l'invio di qualche colonna, costo che comunque non dovrebbe essere eccessivo.

Capitolo 5

Parallelizzazione del generatore

Per motivi legati alla complessità del design proposto e del codice da rifattorizzare, il lavoro di implementazione si è concentrato sulla parallelizzazione del generatore (vedi 2.2.1), che al momento attuale risulta la componente che incide maggiormente sul tempo di esecuzione complessivo dell'algoritmo.

5.1 Introduzione

Algoritmo usato Come accennato nelle sezioni precedenti, il generatore effettua la generazione delle colonne utilizzando un algoritmo di Constrained Shortest Path Tree sul grafo di compatibilità.

Difficoltà del problema Nella versione dell'algoritmo preso in considerazione dal tirocinio, l'assegnamento del personale alle tratte passate in input all'algoritmo, i vincoli riguardano le limitazioni sul lavoro imposte dai sindacati (durata massima di un turno, ad esempio) o regole imposte dall'azienda stessa.

I vincoli possono riguardare sia la collegabilità di due task, sia l'ammissibilità complessiva di un turno.

L'ultimo aspetto rende il problema particolarmente difficile. Un problema

che si riscontra se si prova ad utilizzare un algoritmo di SPT unito a dei controlli finali sui cammini individuati è il *problema della copertura*, che si verifica quando un turno ammissibile e a costo ridotto $(o, n_1, \ldots, n_n, v, m_1, \ldots, m_n)$ non è generato in quanto $(o, p_1, \ldots, p_n, v)$ è un cammino a costo inferiore di $(o, n_1, \ldots, n_n, v)$ ma rende il turno nel quale è incluso inammissibile. Per risolvere questo ed altri problemi si è ricorso ad una soluzione basata su k-CSPT, in cui per ogni nodo vengono trovati i k migliori cammini dall'origine. Ciò, naturalmente, rende il problema gravoso sia in termini di tempo che di occupazione di memoria.

La dimensione del grafo, inoltre, può risultare eccessiva per la sua memorizzazione in memoria. Per tal motivo spesso la forward star di un nodo nel grafo di compatibilità viene calcolata nell'iteratore della forward star del nodo, realizzando la cosiddetta *forward star dinamica*. Questo, naturalmente, appesantisce ulteriormente l'algoritmo in quanto è necessario ricontrollare continuamente i vincoli di collegabilità tra nodi.

Possibilità di parallelismo Le aziende che commissionano il lavoro possono richiedere di generare dei turni prendendo in considerazione diverse normative e diverse zone da cui far partire i turni. Il problema viene dunque risolto ogni volta in modo indipendente per ogni coppia $(\text{normativa}, \text{zona}) \in N \times Z$, dove N è l'insieme delle normative e Z è l'insieme delle zone [1]. I problemi, nella implementazione M.A.I.O.R. in produzione, sono risolti in modo sequenziale; la parallelizzazione è consistita dunque nell'esecuzione in parallelo dei sottoproblemi succitati.

5.2 Implementazione

Il lavoro di implementazione è consistito nelle seguenti attività:

[1] Vi sono casi in cui ciò non è vero, ma nella maggior parte dei casi la descrizione è accurata

- refactoring del codice in modo da poter incapsulare in una classe il codice che deve essere eseguito concorrentemente e da rendere più esplicita la lista di coppie $(\text{normativa}, \text{zona})$ su cui lavorare;
- rendere thread-safe le classi utilizzate dal codice parallelo;
- individuare lo stato condiviso ed o effettuare la sincronizzazione necessaria o duplicare le strutture dati usate.

5.2.1 Thread-safeness del codice utilizzato

Le funzioni thread-unsafe individuate possono essere classificate in due categorie:

1. funzioni per l'iterazione sulle strutture dati;
2. funzioni che accedono a variabili globali accedute in scrittura dal codice eseguito in parallelo.

Iterazione sulle strutture dati Le funzioni che appartengono alla categoria 1 espongono il pattern esposto nella sezione 2.3; il problema è stato risolto definendo, nei container acceduti tramite le funzioni d'iterazioni thread-unsafe, una sottoclasse iterator i cui oggetti sono istanziati tramite i metodi begin() e end(), in modo del tutto analogo agli iteratori implementati nelle STL. Poiché lo stato dell'iterazione è memorizzato nell'iteratore, allocato e dunque non condiviso tra i thread, non si verificano i problemi di corruzione che si verificherebbero tramite l'utilizzo delle funzioni GetFirst() e GetNext().

Variabili globali Le variabili globali accedute in scrittura possono essere classificate in due categorie:

1. variabili "accumulatori", utilizzate dalle funzioni per memorizzare informazioni incrementali associate alla generazione

2. puntatori agli oggetti che realizzano la normativa utilizzata

Accumulatori Queste variabili vengono "incrementate" durante la generazione ed usate e azzerate in coincidenza di certi eventi (generazione di un turno, inizio della generazione, etc.), mantenendo informazioni sulla generazione corrente.

È necessario, in questi casi, associare una variabile ad ogni generatore, piuttosto che avere un unico accumulatore condiviso. Una soluzione poco invasiva, semplice ed efficiente è costituita dal ricorso al *Thread Local Storage*: una variabile TLS con scope esterno (globale) è visibile da ogni compilation unit, ma ogni thread ha una copia privata della variabile. In questo modo il problema di race condition è risolto dal supporto del linguaggio.

Puntatori ad oggetti che realizzano la normativa La normativa utilizzata durante una generazione condiziona i criteri di ammissibilità locali e di turno, implementati tramite alcune classi astratte ridefinite per ogni azienda per cui M.A.I.O.R. cura la turnazione.

Poiché la versione originale dell'algoritmo non supportava l'utilizzo di più normative, il supporto è stato introdotto definendo dei riferimenti globali i cui oggetti puntati realizzano una particolare normativa; la normativa viene dunque attivata impostando in modo opportuno tali riferimenti.

Il lavoro di refactoring ha previsto la rimozione di suddette variabili, rimpiazzandole con una variabile TLS con scope esterno di tipo Normativa, da cui è possibile accedere agli oggetti precedentemente acceduti con le variabili globali eliminate.

5.2.2 Stato condiviso

L'analisi del codice non ha rilevato alcuna struttura condivisa acceduta in scrittura, eccetto le strutture dati utilizzate per memorizzare lo stato della generazione.

Ad ogni nodo del grafo è associato una istanza della classe *CCopia*, che mantiene i cammini col costo relativo individuati dalla zona fino ad esso. Le *CCopia* sono memorizzate direttamente come campo della classe *CNodoCSP*, che descrive un nodo del grafo e vengono reinizializzate al termine di ogni generazione. Ciò comporta i seguenti vantaggi:

- non è necessario pagare il costo del lookup di una mappa Nodo → CCopia;
- le CCopia vengono preallocate alla creazione del grafo, evitando delle costose allocazioni ad ogni generazione.

Tale schema, tuttavia, è inadatto al caso in cui si effettuano più generazioni in parallelo, in quanto ogni thread deve lavorare su delle CCopia private.

La soluzione ha previsto la definizione di una mappa da CNodoCSP a CCopia; di queste ne vengono preallocate $|N \times Z|$ copie e riempite associando ad ogni nodo una CCopia "vuota" come mostrato in 5.1.

Le copie vanno dunque re-inizializzate dopo ogni generazione tramite la funzione cleanCopie, mostrata in figura 5.2

Le mappe vengono dunque passate per riferimento ai thread che effettuano la generazione.

5.2.3 Definizione della classe Worker

Il codice che effettua la generazione è stato isolato e "piazzato" nella classe Worker. Il costruttore prende come argomento i parametri di generazione, una lista che verrà riempiti con i turni generati e la mappa CNodoCSP → CCopia preallocata; la generazione viene dunque invocata invocando l'*operator()*.

```
void GenC3LoopMaior::initializeCopieMaps() {
  if (maps.size() != 0)
    break;
  vector<Job> > jl = getJL();
  vector<Job>::iterator jit;
  int i = 0;
  for (jit = jl.begin(); jit != jl.end(); jit++) {
    maps.push_back(map<CNodoCSP*, CCopie*>());
    map<CNodoCSP*, CCopie*> &init = maps[i++];
    for (int j = 0; j < nodi.size(); j++) {
      CCopie *copia = new CCopie(jit.normativa);
      CNodoCSP *cnode = nodi[j];
      copia->setNode(cnode);
      init[cnode] = copia;
    }
  }
}
```

Figura 5.1: Preallocazione delle mappe

5.2.4 Esecuzione concorrente dei Worker

Per l'esecuzione concorrente dei Worker, sono state vagliate diverse possibilità:

- parallel_reduce di Intel TBB, in cui le funzioni sono i Worker e l'operazione commutativa consiste nell'unione delle liste dei turni generati;
- *for parallelo* di OpenMP;
- utilizzo di una libreria di threadpool.

Le soluzioni sopra elencate, però, si sono rivelate incompatibili con il compilatore utilizzato dall'azienda, MetroWerks CodeWarrior.

Si è deciso dunque di lanciare tutti i thread contemporaneamente e di effettuare subito dopo una join collettiva.

```
void GenC3LoopMaior::cleanCopie() {
  for (int i = 0; i < maps.size(); i++) {
    for(int j = 0; j < nodi.size(); j++) {
      maps[i][nodi[j]]->clean();
    }
  }
}
```

Figura 5.2: Reinizializzazione delle copie

La soluzione è sub-ottimale in quanto i thread sono CPU-bound e lanciarne un numero non proporzionale rispetto al numero di processori disponibili risulta in un rallentamento del sistema dovuto a context-switch inutili dal punto di vista dell'economia dell'applicazione; tuttavia il codice è facilmente adattabile per l'utilizzo di un threadpool, nel caso lo si volesse implementare o si decidesse di migrare ad un compilatore più moderno.

Il codice utilizzato per il lancio è mostrato nella figura 5.3.

```
HANDLE *threads = new HANDLE[jobList.size()];
Worker **workers = new Worker*[jl.size()];
vector< vector<CTurno> > turni(jl.size());
int i = 0;
initializeCopieMaps();
for (it = jl.begin(); it != jl.end(); it++) {
  Job job = *it;
  map<CNodoCSP*, CCopie*> &mp = getMap(i);
  Worker *worker = new Worker(..., turni[i],,,,);
  workers[i] = worker;
  threads[i++] = CreateThread(...,worker,...);
}
WaitForMultipleObjects(..., threads, ...);
cleanCopie();
```

Figura 5.3: Lancio dei thread

5.3 Risultati

Nella figura 5.4 sono riportati i tempi d'esecuzione medi di una generazione, tenendo presente che:

N.	O	NS	NM
1	1.389	1.544	1.966
2	1.295	1.482	1.872
3	1.248	1.482	1.747
4	1.248	1.467	1.747

Figura 5.4: Runtime su una istanza di test

- O è la versione corrente dell'algoritmo M.A.I.O.R.
- NS è la versione nuova dell'algoritmo in cui viene effettuata la wait di ogni thread dopo averlo lanciato (ottenendo una implementazione sequenziale)
- NM è la versione in cui i thread vengono tutti lanciati assieme per poi effettuare una join collettiva, per come mostrato nella figura 5.3

  Per cercare di spiegare i risultati negativi sono state avanzate varie possibili spiegazioni:

  - Rallentamento dovuto a sincronizzazioni tra thread
  - Cattivo utilizzo della gerarchia di memoria

  Sincronizzazioni tra thread Le uniche sincronizzazioni tra thread nell'applicazione sono presenti nell'operatore new; per tal motivo nei test effettuati nella figura 5.5 (il suffisso "A" sta per "allocatore") si è sostituito l'allocatore custom M.A.I.O.R. con *Hoard*, un allocatore specificatamente pensato per le applicazioni parallele. Dai risultati dei test

N.	NSA	NMA	NS	NM
1	1.982	1.95	1.544	1.966
2	1.887	1.88	1.482	1.872
3	1.81	1.81	1.482	1.747

Figura 5.5: Test con allocatore parallelo

T	O	NS	NM
CPI	0.91	0.90	1.25
L1 DCMPI	0.05	0.04	0.08
L2 CMR	0	0	0
L2 %	35.29	36	37.26
PF	3380	8800	9200

Figura 5.6: Utilizzo della gerarchia di memoria

si può vedere come la sincronizzazione tra thread alla new non abbia alcun impatto, dato che *Hoard* ha bassissimi costi di sincronizzazione.

Cattivo utilizzo della gerarchia Per verificare l'utilizzo della gerarchia di memoria sono stati effettuati dei test con VTune, il profiler della Intel, misurando l'utilizzo delle cache e dei fault di pagina per un periodo di 60 secondi.

I risultati sono esposti nella figura 5.6, tenendo conto che:

- CPI sta per Clocks Per Instructions
- L1 DCMPI indica la penalità in performances dei miss di cache L1
- L2 CMR indica la miss rate della cache L2
- L2 % indica lo sfruttamento delle linee della cache L2 da parte dell'applicazione
- PF indica il numero di pagefaults.

L'unica differenza significativa la si può notare nel numero dei page faults, più che raddoppiati nelle versioni modificate dell'applicativo.

Tale fenomeno può essere spiegato con l'introduzione delle mappe CNodoCSP $\rightarrow$ CCopia: mentre la versione sequenziale dell'applicativo lavora con solo una CCopia associata ad ogni nodo, con buone possibilità che queste rimangano permanentemente nel Working Set, la versione parallela deve lavorare con $|N \times Z|$ istanze di CCopia per ogni nodo (nell'applicazione di test sono 27); poiché non tutte possono rientrare nel WS, alcune di queste vengono estromesse per poi essere ricaricate in memoria all'iterazione successiva, causando un fenomeno denominato *trashing*.

Poiché la duplicazione delle strutture dati è strettamente necessaria, un modo per risolvere il problema potrebbe consistere nell'utilizzare un threadpool, precaricando i thread (in numero proporzionale al numero di core) e assegnando ad ognuno di loro una mappa CNodoCSP $\rightarrow$ CCopia, in modo da contenere il consumo della memoria e dunque l'esuberanza di fault di pagina.

Bibliografia

[1] Riprogettazione del BDSA della Maior utilizzando la SetPartLib, Luigi Poderico

[2] Progetto di una famiglia di algoritmi di Set Partitioning con vincoli, Luigi Poderico

[3] http://www.boost.org

[4] http://www.ibm.com/developerworks/rational/library/3101.html

[5] http://en.wikipedia.org/wiki/Intel_TBB

[6] James Reinders, *Intel Threading Building Block: outfitting C++ for Multi-Core Processor Parallelism*, O'Reilly

[7] Gropp et al 96, p.3

[8] http://en.wikipedia.org/wiki/Message_Passing_Interface

[9] Miglioramenti algoritmici nella soluzione di problemi reali di schedulazione di veicoli e personale (Francesco Bernazzani, Samuela Carosi, Antonio Frangioni, Andrea Gaffi, Leopoldo Girardi)