Branch & Bound

Matteo Zattoni

September 14, 2023

Contents

1 Introduzione

L'obbiettivo del lavoro è quello di progettare un algoritmo distribuito che possa risolvere una classe di problemi che possano essere eseguiti tramite la tecnica del Branch and bound. Questa relazione inizialmente descriverà quali sono stati gli obbiettivi del progetto, come il progetto è stato strutturato ed infine quali risultati sono stati ottenuti sperimentando due dataset utilizzando il supercomputer del dipartimento di informatica ed le criticità riscontrate.

La tecnica del Branch and Bound permette di risolvere un problema P generando una serie (potenzialmente molto grande) di problemi più semplici, a questi problemi di solito si risolve il corrispondente rilassamento, al termine delle operazioni di solito si ottiene una soluzione ottima del problema P oppure nessuna soluzione ammissibile. Senza entrare nei dettagli (che dipendono dal problema specifico) il rilassamento produce una soluzione appartenente a P, nessuna soluzione oppure una soluzione che non

appartiene a P (il rilassamento ha una regione ammissibile più grande rispetto a P). In quest'ultimo caso il rilassamento fornisce una sovrastima dell'ottimo chiamata upper bound, ed è necessaria una esplorazione più approfondita dell'insieme delle soluzione ammisibile, l'approccio più semplice consiste nel generare due o più sotto problemi sfruttando una componente della soluzione ottenuta formando una partizione della regione ammissibile (in alcuni problemi potrebbe essere una copertura), questa operazione viene chiamata branch. La tecnica del Branch and Bound tiene traccia della migliore soluzione trovata del problema P (quindi una soluzione ammissibile del problema originario) e quando la sovrastima di un sotto problema è minore o uguale della attuale migliore soluzione allora non vengono generati ulteriori sotto problemi ed il nodo può considerarsi chiuso dato che tutte le soluzione generate da esso non saranno mai migliori di quella attuale, questa operazione viene chiamata bound. L'implementazione del problema specifico scelto, il problema dello zaino, e tutta la teoria associata ad esso ed più in generale alla tecnica del Branch and Bound si trova in due file che inserirò nel repository di questo progetto.

2 Obbiettivo del Progetto

Questo progetto si pone due obbiettivi principali, il primo è stato quello di sviluppare un algoritmo distribuito che implementi la tecnica del Branch and Bound ma che rimanga indipendente dal problema specifico desiderato ed possibilmente che rimanga indipendente anche dal paradigma parallelo scelto. Il secondo (data la natura di questa tecnica) poter progettare un bilanciamento del carico che possa distribuire i branch tra tutti i processi massimizzando l'utilizzazione degli stessi.

Il paradigma utilizzato sarà message-passing utilizzando MPI in un ambiente composto da nodi, inoltre un obbiettivo desiderato è che possa scalare "al meglio" con il numero dei processi, quindi che possa gestire situazioni dove il sistema è pesantemente carico di lavoro e situazioni dove non lo è.

2.1 Struttura

Il progetto è stato diviso in tre parti, la prima è quella che orchestra le operazioni del Branch and Bound distribuito e tutte le classi virtuali che dovranno essere implementate nella parte del problema specifico, dopo andrò a descrivere le classi MPI, queste due parti devono essere indipendenti dalla terza che riguarda l'implementazione dell' algoritmo specifico, per questo progetto è stato scelto di risolvere il problema dello zaino binario (Knapsack problem).

Un obbiettivo secondario è stato quello di rendere l'orchestratore indipendente anche dalla classi MPI in modo che possa in futuro il concetto "parallelo" possa essere applicato anche ad altri paradigmi, quindi tutte e tre le parti devono essere indipendenti tra loro.

3 Branch and Bound

Le classi che compongono questa sezione hanno principalmente due obbiettivi, il primo è quello di definire la classe che orchestrerà le varie componenti del progetto, il secondo è quello di definire tutte le classi virtuali che dovranno essere implementate dal problema specifico (Figure 1); queste classi virtuali definiscono i vari contratti che devono essere implementanti.

3.1 Classe BranchBound

Questa classe è l'orchestratore dell' algoritmo distribuito, sarà responsabile anche della gestione di una struttura dati composta da Branch.

Il costruttore di questa classe ha bisogno di due istanze, una del tipo ParallelManager e una BranchBoundAlgorithm.

La definizione del metodo start (che richiede la collaborazione delle due classi) verrà rimandata alla fine di questa sezione dopo aver elencato le responsabilità delle altre due classi che collaborano con questa.

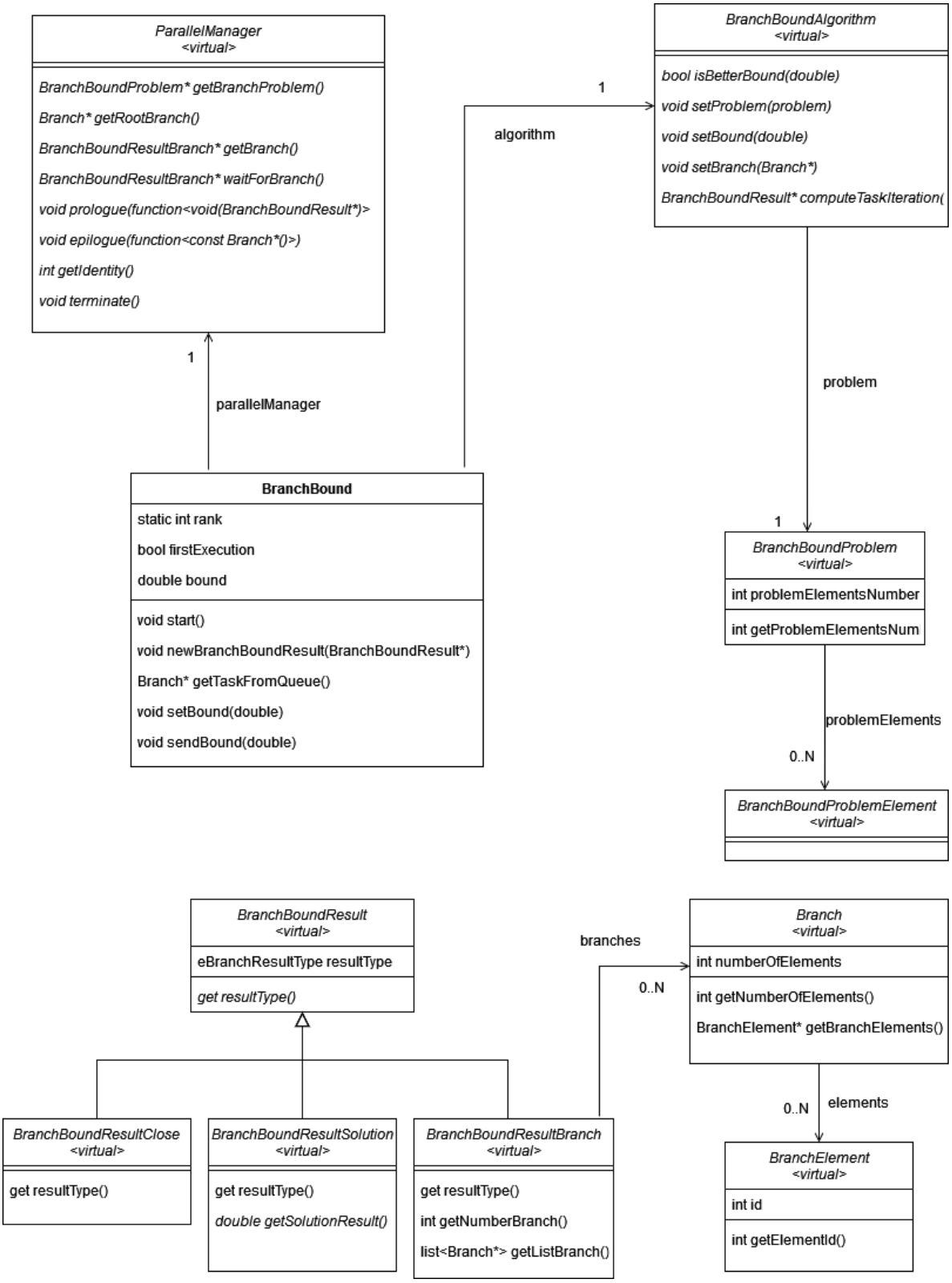


Figure 1: Branch and Bound Class Diagram

3.2 Classe BranchBoundResult

Questa classe virtual definisce un sovratipo che ha la sola responsabilità di restituire un enum che specifica il tipo di risultato, i tre tipi di enum sono Solution, Branches, Closed

- BranchBoundResultSolution

  Questa classe virtuale ha la responsabilità di restituire il valore della soluzione ed ha un metodo implementato (quindi non virtuale) che restituisce il tipo enum Solution.

- BranchBoundResultBranch

  Questa classe virtual ha un riferimento ad una lista costante di tipo Branch*, ha la responsabilità di esporre questa lista e di sapere il numero di branch di essa.

- BranchBoundResultClose

  Questa classe virtuale ha la responsabilità di restituire il tipo enum Closed.

3.3 Classe Branch

Questa classe virtuale modella un branch (oppure un nodo dell'albero di ricerca), ha la responsabilità di conoscere quanti branchElement possiede ed ha un puntatore ad un array di BranchElement

3.3.1 Classe BranchElement

Questa classe virtuale modella l'elemento di un branch ed ha la sola responsabilità di restituire un identificato dell'elemento.

3.4 Classe BranchBoundProblem

Questa classe virtuale ha la responsabilità di conoscere quanti BranchBoundElement è composto l'array al quale ha un riferimento.

3.4.1 Classe BranchBoundProblemElement

Questa è una classe virtuale che non ha specifiche responsabilità se non quella di essere un sovra tipo che dovrà essere ereditato.

3.5 Metodo Start

Il metodo start risolve il problema P utilizzando la tecnica del Branch and Bound. Quando termina lancia una eccezione al chiamante che ne sancisce la terminazione.

Viene chiamato il parallelManager che restituisce una istanza di BranchBoundProblem, questa operazione può essere sia locale che remota, può essere bloccante e non garantisce la sincronizzazione dei processi (anche se come verrà implementata da MPI verrà garantita in quanto verrà utilizzata la collettiva broadcast).

Viene chiamato il metodo setProblem del BranchBoundAlgorithm, infine viene chiamato getRoot-Branch del ParallelManager, questo metodo restituisce una istanza Branch solo al processo designato

per consumere il root branch, altrimenti restituisce nullptr.

Iterativamente fino a quando non viene lanciata l'eccezione di Terminazione Se l'algoritmo non sta lavorando su un branch allora viene estratto un branch dalla struttura dati dell'orchestratore, se è vuota viene chiamato waitForBranch del ParallelManager, in ogni caso un branch viene passato all'algoritmo oppure è stata lanciata l'eccezione che ne determina la terminazione.

Successivamente viene chiamato il prologo del ParallelManager, dopo viene chiamata computeTask dell' algoritmo, il quale restituisce un risultato che viene gestito dall'orchestratore.
Infine viene chiamato l'epilogo del parallelManager.

4 MPI

In questa parte vengono utilizzate le classi che andranno a implementare la classe virtuale ParallelManager descritta nella sezione precedente, dopo aver elencato le criticità della tecnica Branch and Bound andrò ad descrivere una nuova classe virtual MPIDataManager che andrà a concludere il discorso sulla indipendenza delle parti. Dato che è stato scelgo di utilizzare un paradigma a scambio di messaggi per la risoluzione parallela dell' algoritmo andrò ad implementare una classe che utilizzerà la libreria openMPI
Due obbiettivi ideali:

1. Distribuire uniformente i task
2. Minimizzare il tempo della chiamata waitForBranch (l'unica chiamata che può essere bloccante), idealmente vorremmo che quando un nodo fa questa chiamata ci sia sempre un branch pronto ed il relativo tempo di attesa sia minimo.

Criticità del problema

1. Un problema vincolante che deve essere affrontato è il Load Balacing in quanto il Branch and Bound può essere problematico...
2. Il Branch ha dimensione arbitraria, quindi ad ogni send si potranno inviare k elementi di tipo Branch and Bound
3. Dato che uno degli obbiettivi è l'indipendenza tra il problem specifico e L'algoritmo distribuito bisogna assegnare le responsabilità di creator ad alcune classi (che dovranno conoscere le classi concrete, quindi il problema specifico per conoscere quanto buffer allocare e come trattare il layout in memoria dei dati)

Ho scelto di dividere le responsabilità in 3 classi che estenderanno la classe ParallelManager ed attraverso un pattern Chain of Responsability collaboreranno per implementare le funzione descritte nel capitolo precedente. La divisione è dovuta a 3 differenti topologie che saranno utilizzate (con differenti communicator) per andare a trattare il problema della distribuzione dei Task. (Non tutti i nodi utilizzeranno i communicator)

4.1 Classe MPIDataManager

L'esistenza di questa classe virtual è giustificata dal fatto che il ParallelManager restituisce oggetti virtuali (come waitForBranch) ma la sua implementazione deve conoscere le classi concrete che dovranno essere istanziate, inoltre MPI lavora con buffer e tipi quindi deve sapere quanto dovrebbe inviare o ricevere e come disporre i byte di ricezione secondo un layout.
Dato queste premesse allora MPI dovrebbe conoscere il problema specifico che si sta implementando, ma questo violerebbe l'obbiettivo di indipendenza posto inizialmente.

Per risolvere questo problema alcune delle responsabilità verranno rimosse attraverso il pattern Abstract Factory, quindi il ParallelManager per essere istanziato dovrà avere un riferimento ad una classe che implementa MPIDataManager.

Questa classe virtuale definisce un pattern Abstract Factory in quanto chi la implementa avrà la responsabilità di essere creator delle classi del problema specifico. Inoltre avrà la responsabilità di definire MPI_Datatype per ogni tipo (Branch-Bound-Problem-ProblemElement), di allocare buffer e di rilasciarlo, questa classe deve essere quindi implementata insieme al problema specifico.

L'implementazione verrà rimandata alla sezione successiva, concludo che questa classe possiede più responsabilità di una normale Abstract Factory (quindi per non causare confusione non ho usato il nome di factory per la classe) in quanto non solo avrà il pattern creator delle classi del problema specifico ma definirà tipi MPI (quindi utilizzerà la libreria openMPI), ed avrà la reponsabilità non solo di allocare memoria ma anche di rilasciarla (responsabilità eventuale, solo quando una send di un branch è conclusa deve essere rilasciata).

4.2 Classe MPIBranchBoundManager

Questa classe è quella che sta a capo della catena, ha la responsabilità di gestire COMM_WORLD ed è la classe che viene esposta al BranchBound. Le responsabilità principali sono quelle di implementare getBranchBoundProblem, dove il rank 0 lo invierà attraverso una collettiva Broadcast a tutti i rank, infine ha la responsabilità di implementare getRootBranch, ovvero di restituire il nodo radice di tutto l'albero di ricerca.
Le altre responsabilità vengono implementate con la collaborazione di altri Manager attraverso un pattern Chain of Responsability.

Questo Manager non avrà la responsabilità di Load Balacing la quale la delegherà agli altri due nodi della chain.

4.3 Classe MPIWorkpoolManager

L'ultimo nodo della catena, i nodi del comm_world sono divisi in pool di worker che collaborano tra loro e si scambiano branch, la dimensione di questo pool è scelto a tempo di compilazione tramite una costante W. La tecnica di Load Balacing che utilizza questo manager è pensata quando il sistema è pesantemente carico di lavoro:

Nella fase di inizializzazione di allocano due array di dimensione W. Uno per una serie di recv ed uno per una serie di send. Ogni posizione di questo array identifica anche il rank del worker a cui la send/recv è associata. Ogni elemento dei due array è impostato MPI_REQUEST_NULL (nel tipo request) per indicare che nessuna operazione è in atto e nessun buffer è consistente.

Fase di prologo: controlliamo tra gli elementi T dell'array request che hanno MPI_REQUEST_NULL se esiste una send che fa match, questo è effettuato con la chiamata MPI_IProbe, nel caso ci fosse una send allora richiediamo il buffer dal MPIDataManager dopo aver chiamato per MPI_get_count per conoscere quanti elementi vengono inviati.

Fase di epilogo: controlliamo tra gli elementi T dell'array send se hanno MPI_REQUEST_NULL, per ogni elemento chiamiamo la funzione del parametro per avere un oggetto Branch, se questo oggetto non è nullptr allora chiediamo al ParallelDataManager di avere un buffer per inviare questo oggetto, un volta avuto viene inviato tramite una send sincrona non bloccante Inoltre tra le send che non hanno MPI_REQUEST_NULL viene controllata se la rispettiva MPI_Request è conclusa (MPI_Test) nel caso lo fosse viene chiamato il MPIDataManager (sentFinished)

Nella fase wait for branch: viene controllato che le recv iniziate (ovvero se la request è diversa da MPI_REQUEST_NULL) viene chiamata una MPI_Waitany tra queste ed viene restituito il branch della la prima recv che è terminata. Nel caso non esistano recv iniziate (Tutte le request sono MPI_REQUEST_NULL) allora viene chiamata una MPI_Probe (che è bloccate) per ogni messaggio che potrebbe arrivare (non solo quelle del tag branch perchè a questo punto non sappiamo se il problema è terminato).

4.4 Classe MPIMasterpoolManager

Il processo centrale, questo communicator viene creato solo per i rank (del COMM_WORLD) che sono multipli della costante W che è la dimensione dei Workpool. (quindi un processo per workpool sara presente in questa topologia).

Questo permette di inviare i branch (quindi di distribuire) al di fuori di ogni workpool tramite una topologia ad anello (ogni rank in questo pool invia sempre un branch al rank successivo) e inoltre i branch vengono inviati anche tramite una topologia da albero questo per distribuirli quando il size di questo communicator diventa grande(che è Comm_world_size / W).
Le modalità di come vengon inviate sono le stesse del manager precedente (ovvero delle send speculative e delle recv) solo che ogni rank di questo communicator invierà al più 3 elementi (uno al successivo, ed eventuali 2 ai suoi child tramite la topologia ad albero, se esistono).

4.5 Termination

Per la terminazione è stato scelto il dual token per i due ultimi manager della Chain. waitForBranch può lanciare l'eccezione global termination secondo l'algoritmo del token. La terminazione è ad livelli ma il programma si considera terminato quando il MPIMasterpoolManager lancia questa eccezione, in questo caso è garantito che non ci siano più branch in nessun nodo della rete e nessun algoritmo stia computando.

5 Knapsack

Questo è stato il problem specifico scelto, il problema è definito come una serie di oggetti N ognuno con profitto p e peso w, dato uno zaino con capacità W. Come specificato nella introduzione tutta la teoria e gli algoritmi saranno inclusi nel repository del progetto, l'algoritmo che risolve il sotto problema è un rilassamento continuo del problema originario, ovvero vengolo rilassati i vincoli di interezza delle variabili, questo permetterà di calcolare l'upper bound del rilassamento, ricavare l'oggetto con cui fare la partizione del problema come descritto nell'introduzione.

5.1 Implementazione

Sono state estese le classi virtuali (Branch and Bound Class Diagram) portando al seguente Class Diagram Knapsack Class Diagram,

Non mi dilungherò sulla implementazione, oltre le classi che sono state estese è l'utilizzo di una classe di supporto che mantiene tutte le informazioni aggiornate sulla soluzione corrente (ed ovviamente viene inizializzata con un branch) il costo dell' inizializzazione è di una copia della lunghezza della branch passato come parametro.

5.2 Classe KnapsackMemoryManager

Questa classe implementa tutti i metodi descritti in MPIDataManager, tramite il pattern Abstract Factory, in più ha la responsabilità di fornire buffer per MPIManager, il tipo ed infine e il creator delle istanze che saranno passate all'orchestratore. Definisce i tipi MPI_Datatype che vengono utilizzati dal MPIManager e si occuperà di creare le istanze. Una condizione che ho deciso di assegnare è quella di non utilizzare strutture (e quindi buffer) intermedie per la serializzazione/deserializzazione dei buffer che vengono ricevuti/inviati, questo per non allocare memoria aggiuntiva.

Questa classe definirà tutti i MPI_Datatype che saranno utilizzati da MPIManager.

5.3 Tempo di accesso/allocazione in Memora

L'algoritmo opera sequenzialmente su ogni elemento dello zaino (uno dopo l'altro) quindi sia gli elementi del problema che ogni branch utilizzeranno un array come struttura dati.

Dato il diagramma si può notare che ogni classe effettua un overriding dei due operatori di allocazione new e dell'operatore del distruttore delete, ogni classe ridefinisce questi operatori, nello specifico vengono utilizzate due classi template che gestiscono la memoria, la prima classe si chiama AllocatorFixedMemoryPool$< T >$, questa classe si occupa di allocare aree di memoria che possono essere utilizzate per oggetti di tipo T, quando viene chiamata una delete questa classe non rilascia la

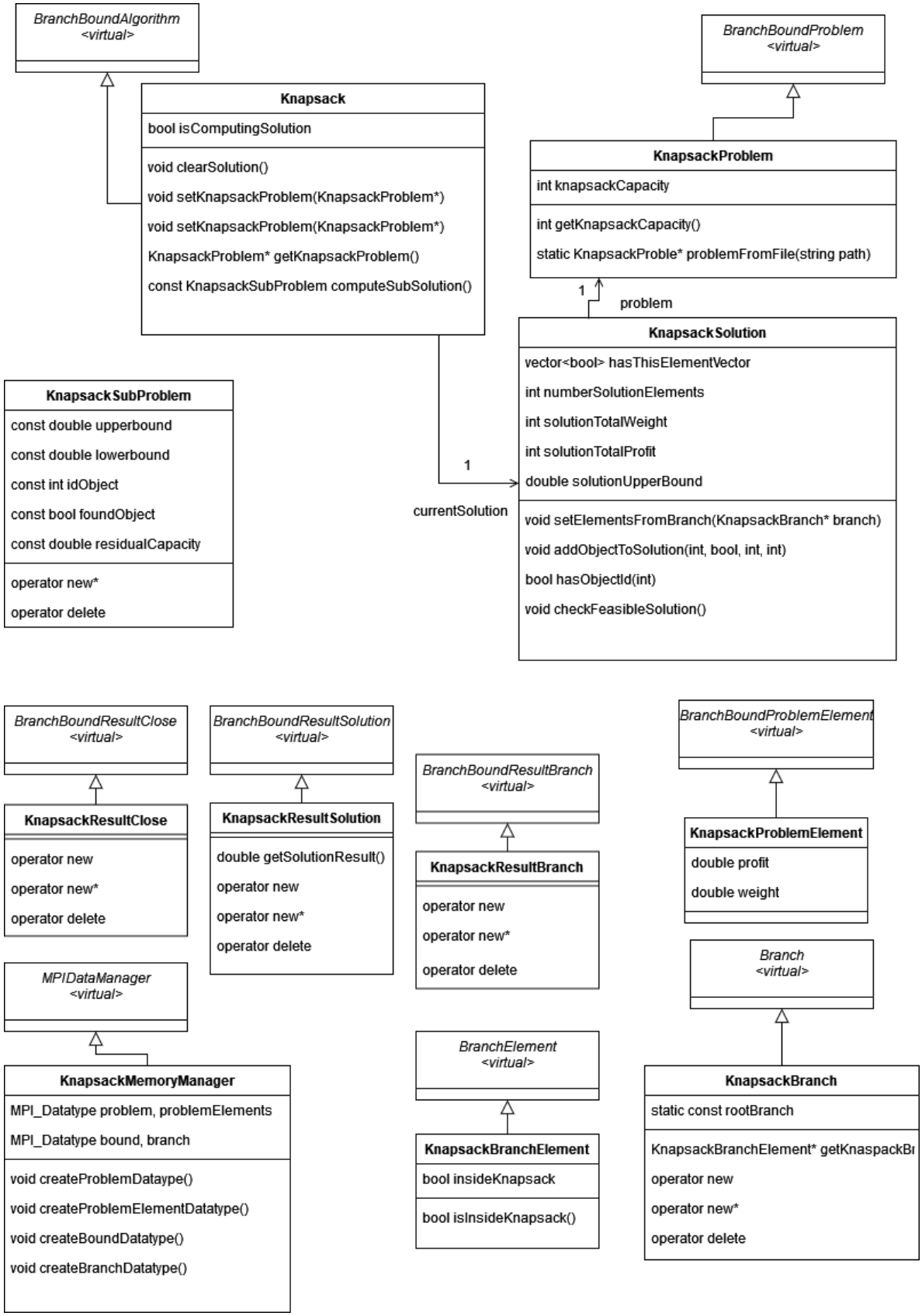


Figure 2: Knapsack Class Diagram

memoria allocata precedentemente ma questa area di memora viene salvata in una propria struttura dati per poter essere riutilizzata alla prossima richiesta di memoria, questo comportamente può essere modificato in modo da rilasciare le aree di memoria se si supera una certa soglia e questa classe vuole diminuire il tempo di allocazione che una malloc richiederebbe, ad esempio per tutte le classi che vengono istanziante e distrutte spesso come i BranchBoundResult. Infine viene utilizzata una classe template AllocatorArrayMemoryPool$< T >$ utilizzata per l'allocazione di un numero arbitrario di elementi branch. Dato che il Branch and Bound è un problema che potrebbe richiedere una enorme quantità di memoria ad un singolo nodo e non potendo prevedere quando e dove possa accadere allora prima di utilizzarle bisogna essere estremamente cauti al loro utilizzo perchè l'esecuzione potrebbe facilmente andare out of memory dato che la memoria non viene rilasciata subito (dato che la si vuole utilizzare alla prossima new).

A prescindere da come si utilizzino, l'importante è che si possa stabilire una propria gestione della memoria è che le classi siano predisposte per assegnargli una propria policy di allocazione in base alle necessità del problema/architettura/enviroments.

6 Sperimentazione

La sperimentazione si basa su due dataset, il primo può essere eseguito anche sequenzialmente (ovvero il codice sequenziale riesce a risolvere il problema nelle 6 ore limite), questo permetterà di calcolare lo speed up della soluzione proposta. Il secondo invece risulta in un dataset più complesso (dove il bound non riesce a diminuire la regione amissibile, rendolo un problema di branching) rendendo la sua esecuzione sequenzile impossibile in un tempo inferiore alle 6 ore (attuale tempo limite per l'utilizzo dei nodi della partizione Broadwell) ma che può essere risolto in maniera parallela inizialmente utilizzando 12 nodi in un tempo di 32 minuti circa).
Per tutti è due i dataset è stato scelto che la costante $W = 12$ quella che specifica il numero massimo di processi in un workpool (Classe MPIWorkpoolManager)

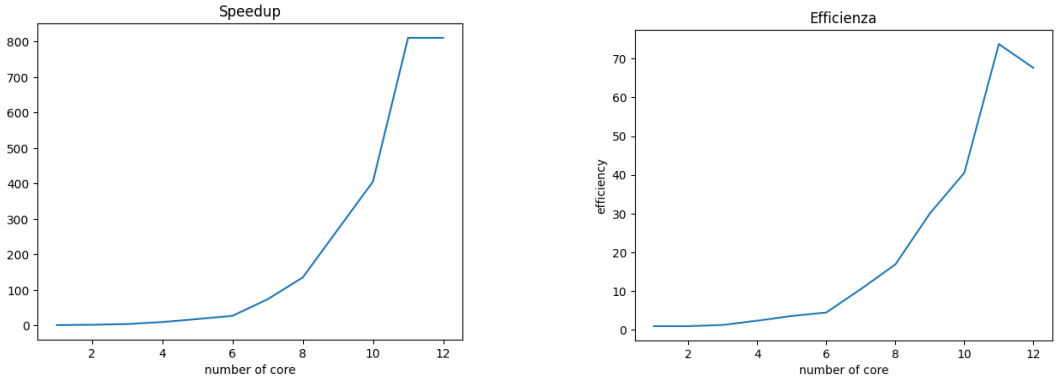
6.1 Primo Dataset

Ho sperimentato l'algoritmo sequenziale e quello parallo su un dataset composto da 1000 oggetti, il tempo sequenziale è stato di 811 secondi (con $n = 1000$).

Tempo di esecuzione	
Processi	s (secondi)
t_s (sequenziale)	811
t_1	828
t_2	416
t_3	209
t_4	85
t_5	45
t_6	30
t_7	11
t_8	6
t_9	3
t_{10}	2
t_{11}	1
t_{12}	1

Dai tempi elencanti sembra che ci sia un miglioramento all'aumentare dei processi, almeno fino t_5, dopo il problema diventa troppo piccolo per poter usare questi risultati in maniera significativa, il che ci portarà a ripetere l'analisi su un secondo dataset. Sono stati riportatati anche lo speedup e la sua efficienza (Figure 3). Come enunciato dal libro del corso ([WA04]) lo speed up ha un comportamento anomalo quando calcolato su un problema Branch and Bound, questo perchè quando viene trovato un nuovo buond questo potrebbe tagliare parte della regione ammissibile dove siamo sicuri che non

esista un ottimo locale migliore di quello attuale, quindi bound influenza il lavoro totale di tutti gli altri processi rendendo il tempo di esecuzione totale dipendente dall'ordine con cui i vari sottoproblemi vengono risolti, questo renderà lo speedup anomalo per questo problema.



(a) Strong Scale (anomalia: super lineare)

(b) Efficienza calcolata

Figure 3: Speedup e efficienza fino ad un numero di 12 processi

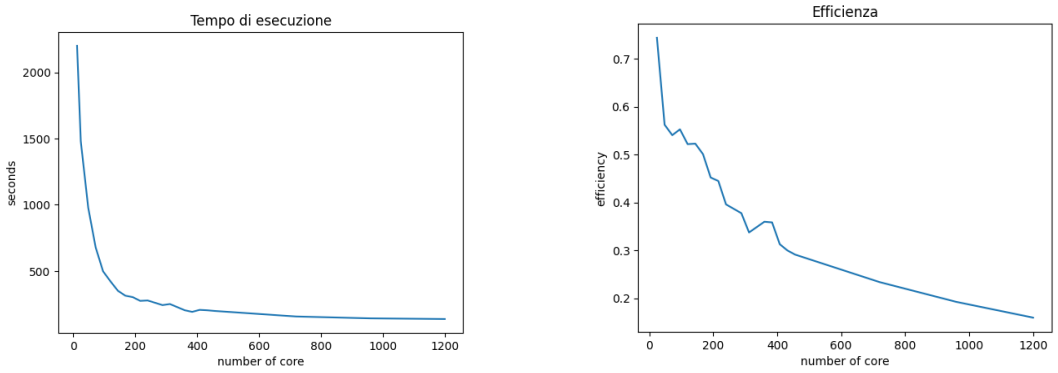
6.2 Secondo Dataset

Il secondo dataset consiste nel risolvere il problema dello zaino quando gli oggetti sono 1250, questo problema non può essere eseguito sequenzialemente perchè il tempo sequenziale necessario supera le 6 ore limite della partizione Broadwell, però quando si utilizzano 12 core si riesce a farlo terminare in un tempo ragionevole.

Tempo di esecuzione	
Processi	s (secondi)
t_{12}	2202
t_{24}	1480
t_{48}	979
t_{72}	679
t_{96}	498
t_{120}	422
t_{144}	351
t_{168}	314
t_{192}	303
t_{216}	275
t_{240}	278
t_{288}	243
t_{312}	251
t_{360}	204
t_{384}	192
t_{408}	207
t_{432}	204
t_{456}	199
t_{720}	157
t_{960}	143
t_{1200}	138

Questo risultati ci indicano il tempo di esecuzione, dalla figura Figure 4 notiamo che all'aumentare dei processi il guadagno che si ha diventa sempre più irrisorio, al punto che sembra quasi nullo, questo significa che l'algoritmo non riesce a scalare all'aumentare dei nodi. Nella prossima sezione trarrò le

conclusioni di questi risultati.



(a) Tempo di esecuzione

(b) Efficienza rispetto a t_{12}

Figure 4: Tempo di esecuzione e efficienza

7 Conclusioni

Il Branch and Bound è stato un problema interessante da trattare perchè si presenta come un problema dall'apparenza semplice ma che nasconde tutta una serie di complicazioni, in questo progetto ho cercato di rendere indipendente la tecnica del Branch and Bound dal problema specifico e implementare una algoritmo di Load Balacing che possa distribuire i task tra i vari processi.
La sperimentazione ha reso possibile capire le criticità dell'algoritmo, questo algoritmo (che tratta il problema dello zaino) non riesce a scalare nel numero dei processi, potrebbero esserci molte motivazioni per questo limite, uno strutturale, in quanto un load balacing fully distributed ha la difficoltà di dover gestire una generazione non uniforme dei task tra i processi dato che il problema dello zaino (e del branch and bound) genera branch in maniera non uniforme (l'albero di ricerca non è bilanciato). Inoltre il problema dello zaino è un problema che può espolodere nel numero dei branch generati in quanto le possibili soluzioni possono essere 2^n mentre l'algoritmo che risolve un sotto problema (ovvero un nodo dell'albero di ricerca) ha complessità lineare $O(n)$ quindi potrebbe esserci un veloce consumo di branch prima che essi possano raggiungere l'estremità della topologia scelta. Uno sviluppo futuro è quello di adottare un approccio parallelo ibrido, ovvero eliminare la classe MPIWorpoolManager ed adottare il paradigma shared memory per effettuare il load balacing tra thread su un nodo ed utilizzare la classe MPIMasterpoolManager per fare il load balacing tra nodi diversi.

Nello specifico gli obbiettivi di questo progetto sono stati:

- Rendere indipendente Branch and Bound , MPI ed il problema specifico in modo che il progetto possa essere reso estendibile/modificabile e mantenibile.
- Implementare un load balacing fully distributed utilizzando diversi communicator (uno per gestire un enviroment heavy system loaded e l'altro per distribuirli in maniera light tra tutta la topologia) in modo da mitigare il comportamento sbilanciato nella generazione di branch di alcuni processi.
- Un processo può bloccarsi solo quando non possiede nessun branch (ovvero la sua struttura dati e vuota e non esistono branch in qualche buffer di ricezione e esiste una send di un branch di un altro processo che è diretta a questo processo) ovvero il processo è terminato localmente. Tutte le operazione di load balacing sono asincrone.
- Non utilizzare memoria aggiuntiva per la serializzazione/deserializzazione dei messaggi; se viene allocata memora per ricevere un buffer che rappresenta un branch non deve esserne allocata memoria aggiuntiva quindi lo stesso buffer allocato per la ricezione deve essere utilizzato anche dell'algoritmo del problema, questo comporta di conoscere bene il layout in memoria delle classi e definirne i corretti MPI_Type

- Gestire la terminazione globale in un ambiete fully distributed, ovvero la terminazione globale implica che nessun branch ed in un buffer di send o recv e tutti i processi non stanno lavorando su branch e hanno la propria struttura dati vuota. in poche parole non esistono branch in nessun processo.
- Utilizzare classi template per la gestione dell'allocazione della memoria e rifidinire tutti gli operatori new/delete delle classi in modo di avere pieno controllo rendendo esplicita nel codice la richiesta di memoria nella heap.

References

[WA04] Barry Wilkinson and Michael Allen. Parallel Programming Techniques and Applications Using Networked Workstations and Parallel Computer. pages 409–410, 2004.