Alma Mater Studiorum · Università di Bologna

CAMPUS DI CESENA - SCUOLA DI SCIENZE
Corso di Laurea in Scienze e Tecnologie Informatiche

Implementazione ed analisi di un framework basato su Cloud per la valutazione di algoritmi di Data Distribution Management

Tesi di Laurea in Reti di Calcolatori

Relatore:
Gabriele D'Angelo

Presentata da:
Nicholas Ricci

II Sessione
Anno Accademico 2013 - 2014

Introduzione

Nel corso di questa tesi verranno analizzati alcuni algoritmi di **Data Distribution Management** (DDM) e un framework realizzato appositamente con lo scopo di testarli e osservare quali di questi si comporta meglio in termini di efficienza di tempo di esecuzione, spazio occupato in memoria e altri aspetti che verranno analizzati nel seguito. Prima di passare ad analizzare in dettaglio ciò che è stato svolto è bene introdurre alcuni concetti utili alla comprensione.

Che cosa sono le simulazioni?

Sono uno strumento sperimentale di analisi molto potente, utilizzato in molti ambiti scientifici e tecnologici a causa della difficoltà o impossibilità di riprodurre fisicamente in laboratorio reale le effettive condizioni da studiare. Sono rese possibili dalle grandi possibilità di calcolo offerte dall'informatica e dai sistemi di elaborazione. Nel suo senso più ampio, la simulazione è uno strumento per valutare le prestazioni di un sistema, esistenti o proposti, in differenti configurazioni di interesse e per lunghi periodi di tempo reale [1]. Una simulazione può essere un qualsiasi videogioco sportivo, dai più classici di calcio, a quelli di volo e infine di guerra. Oltre ai videogiochi le simulazioni sono utilizzate in moltissimi altri ambiti come i crash-test di automobili o di qualsiasi mezzo dove va testata la sua sicurezza. Al giorno d'oggi esistono simulatori di qualsiasi tipo.

Guardando più da vicino le simulazioni al calcolatore si può evincere che, per essere realistiche, devono avere un buon grado di dettaglio. Si possono fare alcune considerazioni a riguardo, molte simulazioni non sono realistiche se riprodotte su un singolo calcolatore, se la potenza di calcolo è poca è necessario fare delle semplificazioni che rendono la simulazione non realistica. Con maggiore potenza di calcolo si può aumentare il livello di dettaglio e quindi, potenzialmente, ottenere risultati più vicini a quelli reali. Nella considerazione appena mostrata si è presa in esame la potenza di calcolo di più macchine: esistono altri componenti da tenere in considerazione, un altro dei quali è il processore che grazie alla tecnologia multicore attuale permette di avere sicuramente delle simulazioni più dettagliate e quindi più vicine alle situazioni reali.

Si è fatto l'esempio nel quale più macchine si mettono insieme per risolvere un certo problema, ma può anche avvenire il contrario. Qualora una o più macchine svolgano una determinata simulazione possono insorgere problemi di interoperabilità in caso si debba comunicare con altre macchine dove si svolgono simulazioni differenti con dati

non omogenei.

Il Dipartimento di difesa degli Stati Uniti ha contribuito a creare uno standard chiamato **High Level Architecture** (HLA) per ovviare a questo problema di interoperabilità e permette a diverse simulazioni di interagire in modo totalmente scollegato dalla piattaforma sulla quale risiedono.

Il primo standard completo di HLA è stato pubblicato nel 1998 [2]. HLA è composto da tre componenti:

- **interface specification**: definisce come le simulazioni devono interagire con l'infrastruttura, la **Run Time Infrastructure** (RTI) è la sua implementazione;
- **object model template** (OMT): definisce quali informazioni devono essere trasmesse tramite le simulazioni;
- **rules**: regole che le simulazioni devono seguire.

All'interno di HLA è utilizzata una terminologia strettamente tecnica, di seguito vengono spiegati alcuni termini:

- **federato**: un'entità conforme ad HLA;
- **federazione**: un insieme di federati connessi tramite RTI usando un OMT comune;
- **oggetto**: una collezione di dati trasmessi tra i federati;
- **attributo**: campo dati di un oggetto;
- **interazione**: un evento inviato tra le entità;
- **parametro**: campo dati di un interazione.

RTI è a sua volta divisa in sette servizi, uno dei quali è il DDM e si occupa della distribuzione dei dati tra i vari federati. Più precisamente il suo compito è quello di limitare i messaggi ricevuti dai federati in grandi federazioni distribuiti solamente a quelli interessati. Questa operazione viene eseguita con lo scopo di ridurre sia l'insieme dei dati che un singolo federato deve processare sia il traffico di messaggi in rete. Il DDM per fare quanto descritto sopra utilizza l'operazione di matching che si occupa di scoprire quali regioni si sovrappongono in un determinato spazio. Nel corso della tesi verrà spiegato meglio questo concetto, quali sono gli algoritmi più utilizzati e saranno eseguiti su specifici test per verificare quale è più conveniente utilizzare in base a diverse situazioni che si possono presentare.

Indice

Elenco delle figure

Capitolo 1

Simulazioni Distribuite e Data Distribution Management

In questo capitolo verrà offerta una panoramica di cosa sono le simulazioni distribuite, del perché sono ampiamente utilizzate e perché è utile parlarne in questa tesi. Nella seconda parte invece verrà spiegato più in dettaglio il ruolo del Data Distribution Management e alcuni concetti per comprenderlo al meglio.

1.1 Simulazioni Distribuite

Per simulazione distribuita si intende una distribuzione dello svolgimento di una singola esecuzione di un programma attraverso più processori. Esistono diversi tipi di paradigmi, uno dei quali è chiamato **parallelo**, in questo caso simulazione parallela. Si cerca di svolgere il programma inerente ad una simulazione su un piccolo insieme di macchine strettamente accoppiate tra loro, come ad esempio supercomputer o macchine multiprocessore a memoria condivisa. Il principale motivo del calcolo parallelo è quello di ridurre il tempo di esecuzione del programma. Ad esempio nel caso in cui venga simulato su N processori, si vorrebbe avere una riduzione del tempo di esecuzione di N volte rispetto all'esecuzione su un singolo processore. Si vorrebbe avere una riduzione del tempo di N volte ma non è possibile che questo accada per via della legge di **Amdahl**. Se la maggior parte del programma è svolta in modo sequenziale allora l'esecuzione parallela di una piccola parte non permette una riduzione significativa del tempo di esecuzione usando più processori.

Se N è il numero di processori, s il tempo speso da un singolo processore su una parte sequenziale del programma, p il tempo speso da un singolo processore su una parte del programma che può essere parallelizzato allora la legge di Amdahl dice che l'incremento del tempo di esecuzione è dato da:

$$\frac{1}{s + \frac{p}{N}}$$

Si noti che se N tende a infinito e se il tempo speso su una parte sequenziale del programma tende a zero si può avere un miglior incremento delle performance usando più processori [3]. Si deve tenere anche in considerazione che la legge di Amdahl non considera il costo della separazione del carico di lavoro nei vari processori (overhead).

Un altro motivo per distribuire lo svolgimento della simulazione è quello di poter eseguire simulazioni molto grandi con mole di dati da elaborare consistenti. Per questo motivo non è possibile eseguire la simulazione su un singolo calcolatore per via della quantità di memoria ridotta, avendo più macchine a disposizione invece possono essere condivise le risorse. L'assegnazione di simulatori differenti per ogni macchina può essere un secondo motivo per effettuare simulazioni distribuite, e avere quindi una simulazione con entità eterogenee. Un esempio per quest'ultimo caso può essere un ambiente di simulazione di guerra, in cui si hanno calcolatori che simulano carri armati, altri che simulano aerei, altri soldati e così via. Il dipartimento della difesa degli Stati Uniti ha contribuito alla creazione dello standard HLA per colmare la mancanza di interoperabilità tra simulatori differenti e soprattutto per riprodurre in modo dettagliato scenari di guerra per poter creare simulatori di addestramento di alta qualità .

Un differente aspetto nelle simulazioni distribuite è l'**estensione geografica** su cui viene eseguita la simulazione. Spesso si parla di simulazione distribuita su area geografica quando le risorse che ne fanno parte sono su aree geografiche distanti. Un esempio di risorse sono i database. Questa distribuzione di risorse comporta un grande vantaggio dal lato economico ma al tempo stesso un calo delle performance dovuto alla velocità della trasmissione dei dati in rete che causa una latenza non trascurabile per simulazioni real-time. Di conseguenza è necessaria la vicinanza delle macchine per l'esecuzione delle simulazioni ad alte prestazioni [4].

1.2 Data Distribution Management

Come già citato in precedenza, il Data Distribution Management (DDM) è uno dei servizi implementati in High Level Architecture, ed è necessario che fornisca efficienti meccanismi di scalabilità per distribuire gli aggiornamenti e le informazioni sulle interazioni in sistemi distribuiti in larga scala. Il meccanismo di filtraggio del DDM è ampiamente utilizzato in simulazioni in tempo reale e siccome dev'essere molto efficiente è diventato un interessante e fondamentale problema per le comunità di simulazione parallela e distribuita [5].

Il servizio DDM si deve occupare fondamentalmente di tre elementi:

- **efficienza**: dovrebbe gestire al meglio le spese in termini sia di computazione sia della latenza dei messaggi sia della memoria utilizzata. Operazioni costose, come comparazione di stringhe, computazioni costose o complesse, etc., dovrebbero essere evitate quando possibile;

- **scalabilità**: dovrebbe avere la capacità di adattare la banda per la trasmissione di informazioni, la memoria necessaria per l'archiviazione di informazioni e il traffico di messaggi.

- **interfacce**: deve supportare le giuste interfacce per fornire la giusta funzione di filtraggio necessaria per le federazioni [6].

Si immagini un aereo militare in volo che deve ricevere gli aggiornamenti dal radar entro un certo raggio di azione. Il compito del DDM è proprio quello di trasmettere gli aggiornamenti a chi li richiede e di filtrarli in modo che all'aereo arrivino solamente determinate informazioni (es. relative agli altri aerei, ai missili ecc.). Il costo della trasmissione di dati non rilevanti può essere molto elevato. Se un entità riceve una quantità notevole di dati irrilevanti, quest'ultima perderà una quantità di tempo notevole per eliminarli. Se il DDM trasmette dati a chi non li ha richiesti ci può essere un sovraccarico della rete e dell'entità ricevente.

È necessario comprendere al meglio come il DDM filtra i dati e la terminologia utilizzata in questo settore, partendo dal concetto di **Routing Space**.

1.2.1 Routing Space

Il **Routing Space** è un sistema di coordinate multidimensionale e rappresenta il mondo virtuale che si deve simulare. Le rappresentazioni riportate di seguito saranno bidimensionali: la prima dimensione rappresenta le coordinate dal basso verso l'alto, mentre la seconda quelle da sinistra verso destra. In questa sede si è scelto di usare due dimensioni per evitare complicazioni nel disegno e nella visualizzazione. Si tenga presente però che nella realtà si utilizzano almeno tre dimensioni tranne in qualche caso minore, questo perché la maggior parte delle applicazioni del mondo reale utilizza più di tre dimensioni.

1.2.2 Publication e Subscription Region

Una **Region** è un sottoinsieme del Routing Space. È composta da più **extent**, un extent per ogni dimensione. Un extent è un coppia ordinata di punti nella dimensione specificata rappresentanti il minimo e il massimo.

Una Subscription Region è un'astrazione dell'insieme dei dati del mondo del quale un'entità è interessata a ricevere aggiornamenti.

Una Publication Region, o Update Region, è un'astrazione dell'insieme dei dati del mondo del quale un'entità è interessata a trasmettere gli aggiornamenti.

In generale il numero di Update Region tende a essere inferiore rispetto a quello delle Subscription Region [7].

1.2.3 Intersection Region

Con il termine **Intersection Region** si indica una specifica Region dove è avvenuta una sovrapposizione di una Subscription Region e di un Update Region. Se esiste un'intersezione tra Update e Subscription allora vengono scambiati i dati tra le due Region. Qualora non ci sia nessuna sovrapposizione tra un Update e una Subscription si dice che non esiste nessun Intersection Region tra i due, le intersezioni nulle non si gestiscono.

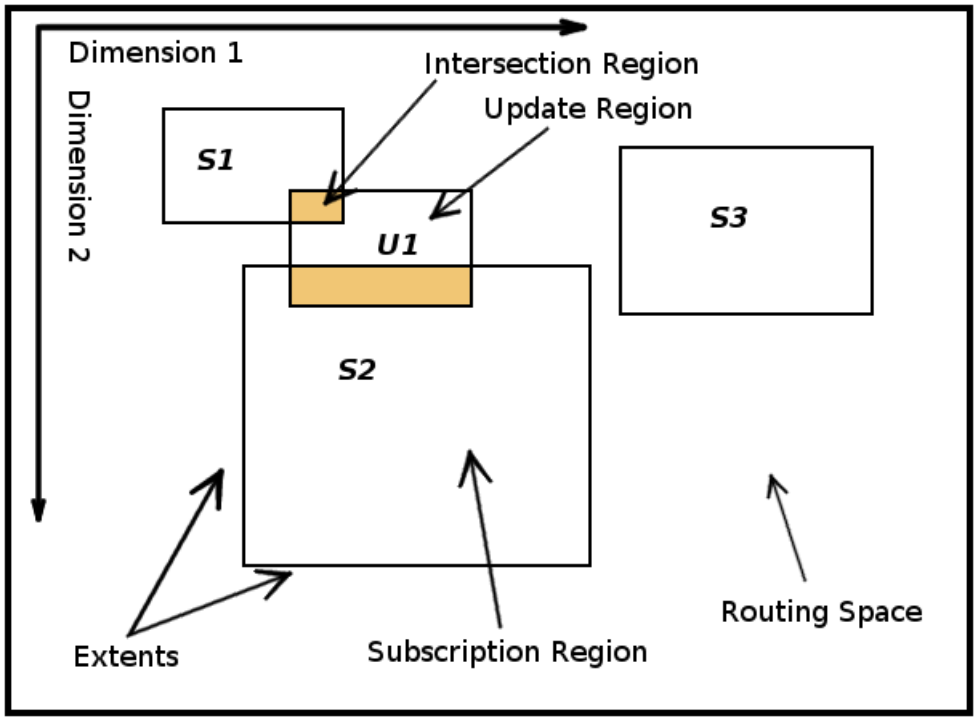

Figura 1.1: Routing Space

Capitolo 2

Necessità di un framework per il Data Distribuition Management?

In questo capitolo si approfonderà l'eventuale necessità di avere e di poter utilizzare un framework per il servizio Data Distribution Management e quali possono essere i vantaggi nell'utilizzare questo tipo di strumento.

La domanda è nata dal fatto che cercando informazioni sul DDM e i vari algoritmi presenti in letteratura ci si è imbattuti in articoli scientifici con algoritmi e test eseguiti su dati, molte volte vantaggiosi per uno specifico algoritmo, testandoli con algoritmi implementati non in modo ottimale e spesso non implementati ma con risultati teorici eccellenti.

È quindi utile creare un framework per verificare un insieme di algoritmi con un insieme di test dove non si privilegia nessun algoritmo in particolare?

Di seguito si prova a dare una risposta a questa domanda elencando i benefici che possono essere apportati da un framework.

- **test**: creare un insieme di test non specifici su un algoritmo in particolare ma di carattere diverso tra di loro permette una sufficiente base per poter valutare un insieme di implementazioni dei vari algoritmi;

- **librerie**: creare un insieme di librerie per monitorare un insieme di caratteristiche degli algoritmi e utilizzarle all'interno delle varie implementazioni possono portare un insieme di vantaggi non indifferenti portando prima di tutto le varie implementazioni degli algoritmi ad un pari livello prima di essere testati;

- **caratteristiche**: si possono prendere in considerazione varie caratteristiche per valutare l'efficienza di un algoritmo, come ad esempio il tempo di esecuzione, l'utilizzo di memoria e ultima ma non meno importante la distanza, in termini di sovrapposizioni tra Update e Subscription Regions, dal risultato ottimale;

- **automazione**: si può cercare di ridurre le compilazioni delle varie implementazioni manuali e creare un sistema di build automatico con semplici comandi per evitare errori;

- **risultati**: avere dei risultati in formato binario sarebbe poco interessante, sarebbe più interessante avere una rappresentazione grafica e testuale dei test svolti. Dare un punteggio a un algoritmo è difficile e non sempre può essere il modo più giusto per via della situazione nella quale si vuole utilizzarlo. Può essere che in uno specifico sistema sia più importante che la soluzione sia ottima, in un altro che interessi avere il tempo di esecuzione minore avendo un numero di sovrapposizioni non ottimale. Detto ciò la scelta sul migliore algoritmo potrebbe spettare all'utente finale.

Tutte le precedenti voci elencate sicuramente sono dei vantaggi per poter comparare vari algoritmi, quindi sicuramente si può dire che con la situazione attuale per far si che ci sia più chiarezza sull'efficienza degli algoritmi è necessaria la creazione di un framework. Se da una parte ci sono i vantaggi, dall'altra ci sono anche degli svantaggi più o meno rilevanti.

- **adattamento**: se si vuole confrontare la propria implementazione dell'algoritmo è necessario apportare delle modifiche con le librerie fornite in modo tale da renderlo coerente con le altre implementazioni all'interno del framework. Ciò comporta a volte poche modifiche ai sorgenti, altre volte potrebbero richiederne molte per far si che i dati in ingresso forniti dal framework siano coerenti con le strutture dati utilizzate all'interno dell'implementazione e far si che i dati in uscita siano coerenti con le strutture utilizzate dalle librerie e quindi coerenti con il framework stesso;

- **tempo di esecuzione**: svolgere test su ogni algoritmo potrebbe impiegare alti tempi di esecuzione.

Nel caso in cui più utenti utilizzino il framework per testare i propri algoritmi e volessero confrontare i risultati ottenuti si riscontra un problema non trascurabile, l'utilizzo del framework su macchine non uguali rende i risultati non confrontabili. Questo problema verrà trattato in un capitolo successivo, nel quale si discuterà di una soluzione comoda e veloce: il **Cloud**.

Nel prossimo capitolo verrà spiegato come si è implementato il framework per il Data Distribution Management, cosa permette di fare e come utilizzarlo.

Capitolo 3

Implementazione del framework per il Data Distribution Management

In questo capitolo verranno spiegate le scelte implementative per la realizzazione del framework, com'è stato realizzato, qual'era il punto di partenza e quali obiettivi si sono raggiunti. Alla fine di questo capitolo si parlerà di quali sono le migliorie e cosa si potrebbe implementare in futuro.

3.1 Scelte Implementative

Il framework è stato implementato su sistema operativo Linux, distribuzione XUbuntu 14.04 e per avere una maggiore compatibilità con il sistema si è scelto di sviluppare buona parte del framework, praticamente tutta l'automazione del framework, in shell bash (versione 4.3.11). Il framework è destinato ad algoritmi creati in linguaggio C, questa scelta è stata presa in base al fatto che queste simulazioni hanno bisogno di buone performance e il linguaggio C sembra il migliore nel rispondere a questa caratteristica. La libreria creata è sviluppata in C per poter essere inclusa negli algoritmi. La compilazione degli algoritmi viene effettuata tramite il programma GCC (GNU C Compiler) (versione 4.8.2).

3.2 Panoramica dal punto di partenza al punto di arrivo

Si è partiti come base da un pacchetto contenente le implementazioni degli algoritmi *Interval Tree Matching* (monodimensionale), *Brute Force* e *Sort Matching Standard* [8]. Si è effettuato uno studio sui tre algoritmi esistenti per verificare quali fossero le strutture in input e in output che potessero essere presenti in tutti e tre. La prima idea è

stata quella di creare una libreria esterna dove raccogliere tutte le strutture e le funzioni utili per poi essere incluse all'interno degli algoritmi. In contemporanea è nata l'idea di automatizzare la compilazione dei tre algoritmi e di creare quindi un launcher per compilare ed eseguire gli algoritmi sopracitati. Il primo test che è stato effettuato su questi algoritmi si chiama alfa. Alfa indica il grado di sovrapposizione di update e subscription region. Tramite questo dato è stato possibile creare un insieme di extent divisi a metà tra updates e subscriptions con un grado di sovrapposizione indicato da alfa. Inizialmente l'insieme dei dati utilizzati dal brute force e dall'interval tree erano creati tramite una funzione interna. La stessa cosa era fatta dal sort matching standard. Un primo problema presente era la presenza di un insieme di dati diversi per ogni algoritmo che rendeva la comparazione non affidabile. Successivamente sono stati importati i sorgenti per la creazione di nuovi test con un grado di alfa molto elevato da un testbed creato da un tesista precedente [9]. Quest'ultimi sono test con valori di lower bound e upper bound molto alti e con un alto grado di sovrapposizione tra update e subscription extent. In questi il numero di update e di subscription sono diversi al contrario del test alfa. Sono stati creati comandi apposta nel launcher bash e migliorata di volta in volta la libreria in C. Di seguito verranno spiegati gli script e la libreria sviluppati.

3.3 Struttura del framework

Di base il framework è composto dai seguenti componenti:

- **Algorithms**: contiene tutti gli algoritmi implementati;
- **C libraries for this framework**: contiene la libreria *DDM_input_output* da utilizzare negli algoritmi. All'interno di questa cartella è contenuto anche il sorgente *main.c*, quest'ultimo è un template utile da utilizzare per implementare nuovi algoritmi, oppure per modificare un algoritmo esistente che si vuole importare all'interno del framework.
- **Configuration**: contiene un file configuration.sh al cui interno sono salvate variabili bash utili al corretto funzionamento del framework. Le variabili che si dovranno settare in questo file sono le seguenti:
  - START_EXTENTS: questa variabile è utile per il test *alfa* e indica il numero di extent con il quale partire;
  - MAX_EXTENTS: questa variabile è utile per il test *alfa* e indica il numero di extent massimo da raggiungere;
  - STEP_SIZE: questa variabile è utile per il test *alfa* e indica il numero di extent da aggiungere a START_EXTENTS fino ad arrivare a MAX_EXTENTS;

  - DIMENSION: questa variabile è utile per il test *alfa* e indica il numero di dimensioni da utilizzare.
  - CORES: questa variabile è utile per il test *alfa* e indica il numero di core massimo da raggiungere. Si parte con un numero di core minimo di 2 e si arriva al numero settato in questa variabile. Variabile che viene utilizzata dalle direttive OpenMP;
  - ALFAS: questa variabile è un array contenente i vari gradi di sovrapposizione che si vogliono testare, per gli eseguibili sequenziali;
  - ALFAS_PAR: questa variabile ha lo stesso scopo di ALFAS, l'unica differenza è che questa viene utilizzata per le simulazioni parallele;
  - RUN: numero di esecuzioni che deve svolgere lo stesso algoritmo prima di restituire un risultato. Viene fatta una media dei risultati ottenuti da ciascuna run.

- **Documentation Latex File**: contiene i file Latex inerenti alla documentazione. Siccome è un progetto open source sotto licenza GPL se qualche utente vuole migliorare il framework può farlo migliorando anche la documentazione;

- **utils**: contiene vari tool utili per lo svolgimento del framework:
  - **bash**: cartella con all'interno quattro script in bash:
    * **create_instances.sh**: generazione di una serie di test diversi da alfa creati con il programma *InstanceMaker*;
    * **definitions.sh**: variabili utili definite all'interno di tutti gli script in bash;
    * **functions_alfa.sh**: funzioni utili per lo svolgimento del test alfa;
    * **functions_other_test.sh**: funzioni utili per lo svolgimento dei test creati con *InstanceMaker*.
  - **Bitmatrix_Comparator**: contiene i sorgenti per creare l'eseguibile per la comparazione delle bitmatrix per ottenere la distanza dalla soluzione ottima tra due algoritmi;
  - **InstanceMaker**: contiene i sorgenti per creare l'eseguibile *InstanceMaker*. Questo programma permette di creare test specificando opzioni utili, test personalizzabili attraverso una serie di opzioni. Sorgenti importati da un'altra tesi;
  - **alfa_creator.c**: sorgente per creare l'eseguibile *alfa_creator*, permette di creare l'insieme dei dati in base ai valori impostati nelle variabili nel file *configuration.sh*. Per il test alfa;
  - **avarager.c**: sorgente per creare l'eseguibile *avarager*, permette di fare la media di valori da un file di testo.

  - **Makefile**: questo file permette la compilazione automatica di tutto ciò che contiene la cartella *utils* tramite il comando *make*.

- **AUTHORS.TXT**: file di testo contenente i nomi di tutti gli autori che hanno lavorato al progetto;
- **CITATION**: file di testo contenente la dicitura da inserire per citare questo progetto;
- **LICENCE**: file di testo contenente la licenza GNU GPL;
- **README.md**: file di testo utile per la comprensione del framework;
- **documentation.pdf**: file creato con Latex, utile per capire come utilizzare il framework;
- **launcher.sh**: script in bash con cui far partire il framework. Successivamente verranno spiegate le sue opzioni e il suo utilizzo.

3.4 Opzioni del launcher.sh

Il launcher ha un'insieme di funzioni utili per gli algoritmi: inserirli nel framework, compilarli, eseguirli e infine creare delle rappresentazioni grafiche dei risultati.

3.4.1 Configurazione

La parte di configurazione del framework è richiamata se necessaria in tutte le altre opzioni del *launcher.sh*. Se non esiste il file *configuration.sh* all'interno della cartella *Configuration* il framework non funziona correttamente. Si può anche richiamare manualmente ma in ogni caso la prima volta che si utilizza il framework, a meno che il file *configuration.sh* non esista, verrà richiamato questo comando:

```
./launcher.sh configure
```

L'opzione *configure* è una delle poche opzioni all'interno di questo framework che richiede l'interazione con l'utente. Richiede l'inserimento di valori per le variabili già citate nella sezione **Struttura del framework**: START_EXTENTS, MAX_EXTENTS, STEP_SIZE, DIMENSION, CORES, ALFAS, ALFAS_PAR, RUN. Un esmpio di interazione con l'utente:

```
#START_EXTENTS
while true; do
  read -p "Insert the number of EXTENTS that programs must START:" yn
  if [[ $yn =~ $re_integer ]];
  then
    START_EXTENTS=$yn
```

```
      echo "START_EXTENTS=$yn" >> $_CONFIGURATION_SHELL
      break
    elif [ -z "$yn" ];
    then
      START_EXTENTS=50000
      echo "START_EXTENTS=$START_EXTENTS" >> $_CONFIGURATION_SHELL
      break
    else
      echo "Please insert an integer number."
    fi
  done

  ...

  #ALFAS
  index=0
  while true; do
    read -p "Insert the $index value of ALFAS:" yn
    if [[ $yn =~ $re_float ]];
    then
      ALFAS[$index]=$yn
      let "index=index+1"
    elif [ -z "$yn" ] && [ $index -eq 0 ];
    then
      ALFAS[$index]=0.001
      break
    elif [ -z "$yn" ];
    then
      break
    else
      echo "Please insert a float number."
    fi
  done
  echo "ALFAS=\"${ALFAS[@]}\"" >> $_CONFIGURATION_SHELL
```

Il primo snippet controlla che il valore inserito sia di tipo intero per evitare problemi di vario genere. Per ogni variabile si forniscono anche dei valori di default in caso l'utente non inserisca nessun valore. Nel secondo snippet siccome ALFAS è un array e può contenere più valori, la gestione dell'interazione è leggermente più complessa. Quando non si inserisce nessun valore si termina l'inserimento. L'interazione per l'inserimento dei valori di tutte le altre variabili è pressoché identico agli esempi presentati.

3.4.2 Inserimento algoritmo

La prima opzioni utile per adattare l'algoritmo al framework è quella di *build*. Richiamabile dal terminale linux tramite:

```
./launcher.sh build
```

Qui di seguito viene mostrato com'è stata implementata questa opzione:

```
#Build all algorithms in Algorithms folder
cd $_ALGORITHMS
algs=($(ls -d */))
for i in ${algs[*]}
do
```

```
6       cd $i
7
8       if [ ! -f $_DDM_PARALLEL ];
9       then
10        touch $_DDM_PARALLEL
11        echo "## Insert the name of parallel executables in bin folder. One line, one
              executable." > $_DDM_PARALLEL
12      fi
13      if [ ! -f $_DDM_SEQUENTIAL ];
14      then
15        touch $_DDM_SEQUENTIAL
16        echo "## Insert the name of sequential executables in bin folder. One line, one
              executable." > $_DDM_SEQUENTIAL
17      fi
18      make
19
20      cd ..
21    done
22    cd ..
23
24    #Build the utils folder which there is avarage programs
25    cd $_UTILS
26    make
27      #Build the InstanceMaker folder which there is a program
28      #that can create interesting tests
29      cd $_INSTANCE_MAKER
30      make
31      mv ./$_DDM_INSTANCE_MAKER ../
32      cd ..
33      cd $_F_BITMATRIX_COMPARATOR
34      make
35      mv ./$_BITMATRIX_COMPARATOR ../
36      cd ..
37    cd ..
```

Tramite questo snippet di codice si esegue la compilazione di tutti gli algoritmi all'interno della cartella *Algorithms* e per ogni algoritmo vengono creati due file di testo *DDM_Parallel* e *DDM_Sequential* con lo scopo di inserire successivamente al loro interno i nomi degli eseguibili, uno per riga, che si vogliono comparare all'interno del framework. Dopo la compilazione dei vari algoritmi si passa alla compilazione dei tool utili per la comparazione della soluzione ottima (Bitmatrix Comparator), per creare delle istanze di test (DDM Instance Maker), per fare la media dei tempi di esecuzione e della memoria utilizzata (Averager), e infine per creare le istanze dell'alfa test (alfa creator). Tutte le compilazioni avvengono tramite il comando *make*, quindi per far si che il framework funzioni correttamente è necessario che ogni algoritmo all'interno della cartella *Algorithms* contenga un Makefile.

3.4.3 Creazione dei test

Prima di poter eseguire gli algoritmi bisogna creare test specifici tramite il comando:

```
1   ./launcher DDMInstanceMaker
```

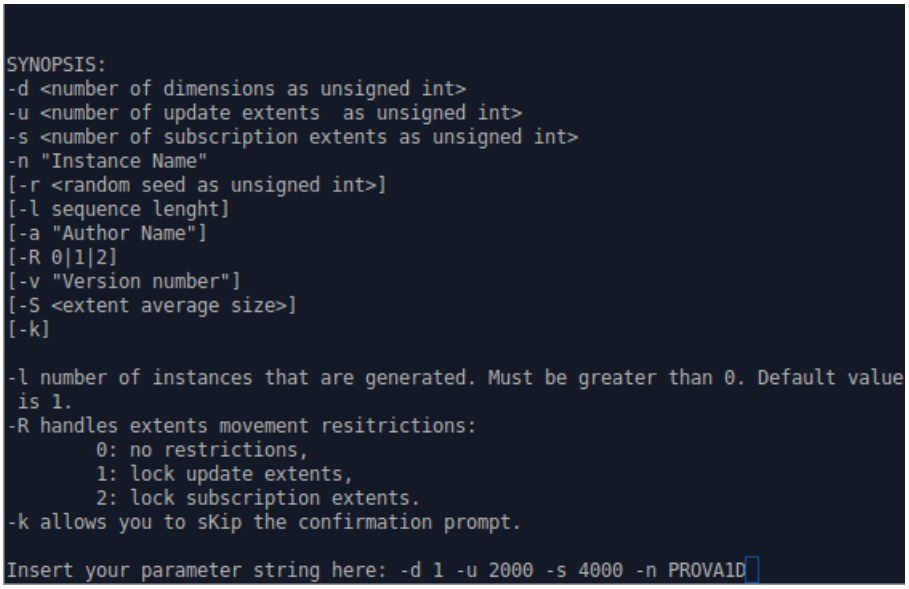

Figura 3.1: DDMInstanceMaker

Questo tool è stato importato e integrato all'interno del framework da un vecchio progetto di tesi reso disponibile sotto licenza GPL [9]. È necessario utilizzare questo tool per creare un nuovo test oppure si può utilizzare il comando:

```
./launcher.sh DDMDefaultsTests
```

Questo comando utilizza il tool DDMInstanceMaker per creare dei test di default. Viene svolto attraverso uno script in bash che in modo sequenziale crea test con specifiche diverse. Nel caso si voglia utilizzare il test alfa non è necessario utilizzare nessun tool manuale per la creazione delle informazioni sul numero di update e di subscription, di dimensioni e sugli update e i subscription ma automaticamente quando si lancerà il comando per testare gli algoritmi, specificando alcune opzioni che vedremo successivamente, verranno create una serie di istanze randomiche create dal tool *alfa creator*. Lo snippet di codice che svolge questo lavoro è il seguente:

```
function create_all_alfa_folders_and_files {
 local alfas=( "$@" )
 for ALFA in ${alfas[@]}
 do
   EXTENTS=$START_EXTENTS
   while [ $EXTENTS -le "$MAX_EXTENTS" ]
   do
     create_alfa_folders_and_files $ALFA $EXTENTS
     let EXTENTS+=$STEP_SIZE
   done
 done
}
```

Tramite questa funzione per ogni valore di alfa all'interno dell'array alfas e da un numero di extent iniziale fino ad arrivare al numero di extent massimo, incrementando ogni volta di un numero di extent contenuto all'interno della variabile STEP_SIZE si richiama la funzione bash *create_alfa_folders_and_files* con due parametri: alfa ed extent correnti.

```
function create_alfa_folders_and_files {
  #$1 ALFA
  #$2 EXTENTS

  local ALFA_FOLDER="../../../TestsInstances/
                     ALFA_$1_$2_$DIMENSION"
  local INFO="info.txt"
  local INPUT="input-0.txt"
  local ALFA_INFO="$ALFA_FOLDER/$INFO"
  local ALFA_INPUT="$ALFA_FOLDER/$INPUT"
  local ALFA_CREATOR="../../../utils/alfa_creator"

  mkdir -p $ALFA_FOLDER

  touch $ALFA_INFO
  touch $ALFA_INPUT

  echo "#Instance name: ALFA_$1_$2_$DIMENSION" > $ALFA_INFO
  echo "#Instance version: 1" >> $ALFA_INFO
  echo "#Author: Nicholas Ricci" >> $ALFA_INFO
  echo "#Random Seed: XXXX" >> $ALFA_INFO
  echo "#Sequence length:" >> $ALFA_INFO
  echo "1" >> $ALFA_INFO
  echo "#Dimensions" >> $ALFA_INFO
  echo $DIMENSION >> $ALFA_INFO
  SUBSCRIPTIONS=`echo "$2 / 2" | bc`
  echo "#Subscription Regions" >> $ALFA_INFO
  echo "$2 / 2" | bc >> $ALFA_INFO
  UPDATES=`echo "$2 / 2" | bc`
  echo "#Update Regions" >> $ALFA_INFO
  echo "$2 / 2" | bc >> $ALFA_INFO

  $ALFA_CREATOR $2 $DIMENSION $1 $ALFA_INPUT
}
```

Con questa funzione verrà creata, se non è stata creata in precedenza, la cartella *TestsInstances* all'interno della root del framework. All'interno di questa cartella verrà creata a sua volta un'altra cartella denominata *ALFA_alfa_extents_dimension*. Al suo interno saranno creati due file:

- *info.txt*: in questo file testuale saranno presenti i valori inerenti alla dimensione, numero di subscription, numero di update presenti all'interno di questo test;
- *input-0.txt*: in questo file testuale saranno presenti per primi tutti i valori dei subscription e successivamente quelli degli update. Ogni riga di questo file è composta dal relativo id seguito da tutte le coppie di punto basso e punto alto per ogni dimensione.

Il file *input-0.txt* viene creato dal tool *alfa creator*:

```
const float Lmax = 1.0e6; /* dimension length */
float l = alfa * Lmax / ( (float) updates + subscriptions );

...

if ((file_input = fopen(argv[4], "w+")) != NULL){
   sprintf(to_write,
       "#Subscriptions <id> <D1 edges> [<D2 edges>]...\n");
   fputs(to_write, file_input);
   for (i = 0; i < subscriptions; ++i){
     sprintf(to_write, "%"PRIu64"", i);
     for (k = 0; k < dimensions; ++k){
       lower = (Lmax-l)*random() / ((float)RAND_MAX + 1);
       upper = lower + l;
       sprintf(to_write, "%s %"PRId64" %"PRId64"",
               to_write, lower, upper);
     }
     sprintf(to_write, "%s\n", to_write);
     fputs(to_write, file_input);
   }
   sprintf(to_write,
       "Updates <id> <D1 edges> [<D2 edges>]...\n");
   fputs(to_write, file_input);
   for (i = 0; i < updates; ++i){
     sprintf(to_write, "%"PRIu64"", i);
     for (k = 0; k < dimensions; ++k){
       lower = (Lmax-l)*random() / ((float)RAND_MAX + 1);
       upper = lower + l;
       sprintf(to_write, "%s %"PRId64" %"PRId64"",
               to_write, lower, upper);
     }
     sprintf(to_write, "%s\n", to_write);
     fputs(to_write, file_input);
   }
   fclose(file_input);
 }
```

In base al valore contenuto nella variabile alfa dipenderà il livello di sovrapposizione e quindi il numero di match che ci possono essere tra subscription e update. Più è piccolo alfa meno sovrapposizioni saranno presenti all'interno del test, più è grande alfa più sovrapposizioni saranno presenti nel test.

3.4.4 Esecuzione degli algoritmi

Dopo aver creato i vari test che si vogliono effettuare si può iniziare ad eseguire i vari algoritmi tramite opzioni specifiche per ogni tipo di test che si vuole svolgere. In questo framework sono state valutate tre diverse metriche:

- **tempo di esecuzione**: il primo test di cui parliamo è quello dello svolgimento dell'algoritmo. Il suo compito è quello di ricavare il tempo di esecuzione di quest'ultimo. Verrà salvato un file all'interno della directory bin, dove risiede l'eseguibile e dopo la fine di tutte le run viene salvata una media di tutte le run. Dopo aver fatto la media il framework sposta il file di testo contenente il tempo di esecuzione

nella cartella *Results* nella cartella apposita chiamata con lo stesso nome del test eseguito. Per utilizzare questa opzione:

```
./launcher.sh run alfa
```

- **memoria utilizzata**: il secondo test che si può effettuare è quello sulla memoria di picco utilizzata. Questo test viene effettuato tramite il framework **Valgrind**. Valgrind è uno strumento open source utilizzato soprattutto per il debug di problemi di memoria, la ricerca di un consumo non voluto della memoria dovuto alla mancanza di deallocazione dalla stessa [10]. In questo caso si utilizza il tool **massif** che è un profiler della memoria, cioè misura lo spazio heap utilizzato dal programma. Solamente alla fine del processo di profiling tramite il comando **ms_print** è possibile avere una rappresentazione grafica del risultato ottenuto con il tool massif. Di seguito le righe di codice per ottenere la memoria occupata.

```
valgrind --tool=massif --massif-out-file=$_VALGRIND_OUT_FILE ./$1 $2 $3 $4 $5
ms_print $_VALGRIND_OUT_FILE > "temp"
unit=`cat "temp" | head -9 | tail -2 | head -1`
unit=`echo $unit | cut -c1-2`
temporaneous_variable=`cat "temp" | head -9 | tail -1 | awk '{print $1}'`
temporaneous_variable=`echo "${temporaneous_variable//^}"`
if [ "$unit" = "KB" ];
then
  temporaneous_variable=`echo "$temporaneous_variable / 1024" | bc -l`
elif [ "$unit" = "GB" ];
then
  temporaneous_variable=`echo "$temporaneous_variable * 1024" | bc -l`
fi
$temporaneous_variable > $_VALGRIND_OUT_FILE
rm -f "temp"
```

Viene eseguito l'algoritmo e il risultato stampato all'interno di un file, successivamente con il tool ms_print si stampa il risultato in una forma più leggibile dentro ad un altro file temp. La riga che si va a prendere tramite il comando head −9 | tail −2 | head −1 è quella dov'è l'unità di misura utilizzata (es. KB, MB, ecc.). Tramite il comando head −9 | tail −2 | head −1 | **awk** '{print $1}' si ottiene la quantità di memoria utilizzata con il simbolo ^ come ultimo carattere che si va ad eliminare tramite **echo** ${temporaneous_variable//^}. Infine si va ad effettuare una conversione in MB se l'unità di misura è in KB o GB.

L'opzione utilizzata per effettuare questo tipo di test è la seguente:

```
./launcher.sh run alfa mem
```

Questa volta il risultato sarà un file con estensione .mem e come il test sull'esecuzione del tempo anch'esso a fine esecuzione delle run viene spostato all'interno della cartella *Results* nella cartella apposita chiamata con lo stesso nome del test eseguito.

- **distanza dalla soluzione ottima**: il terzo e ultimo test implementato è quello sulla distanza dalla soluzione ottima. Questo tipo di test si avvale del salvataggio su disco di una matrice di bit contenente tutte le sovrapposizioni tra update e subscription. Come i precedenti a esecuzione avvenuta sposta il file contenente la matrice, con estensione .bin, all'interno della cartella *Results* nella cartella apposita chiamata con lo stesso nome del test eseguito.

  Per utilizzare questa opzione:

```
./launcher.sh run alfa dist
```

L'esecuzione tra l'alfa test e quelli creati tramite DDMInstanceMaker è leggermente diversa. L'esecuzione dei test diversi dall'alfa:

```
#$1 is executable name
#$2 is type test
#$3 is dimensions
#$4 is updates
#$5 is subscriptions
#$6 is mem for valgrind

filename="$1.txt"
for R in `seq $RUN`
do
  echo "running >$1< test: $2, dimensions: $3, updates: $4, subscriptions: $5, RUN:$R"
  if [ "$6" = "mem" ];
  then

    valgrind --tool=massif --massif-out-file=$_VALGRIND_OUT_FILE ./$1 $2 $3 $4 $5
    ms_print $_VALGRIND_OUT_FILE > "temp"
    unit=`cat "temp" | head -9 | tail -2 | head -1`
    unit=`echo $unit | cut -c1-2`
    temporaneous_variable=`cat "temp" | head -9 | tail -1 | awk '{print $1}'`
    temporaneous_variable=`echo "${temporaneous_variable//^}"`
    if [ "$unit" = "KB" ];
    then
      temporaneous_variable=`echo "$temporaneous_variable / 1024" | bc -l`
    elif [ "$unit" = "GB" ];
    then
      temporaneous_variable=`echo "$temporaneous_variable * 1024" | bc -l`
    fi
    echo $temporaneous_variable > $_VALGRIND_OUT_FILE
    rm -f "temp"

  else

    ./$1 $2 $3 $4 $5 $6

  fi

  let R+=1
done

if [ "$6" = "dist" ];
then
  mv $_BITMATRIX_NAME  "$RESULTS/${1}.bin"
else
  if [ "$6" = "mem" ];
```

```
    then
      #average of memory usage
      AVERAGE_MEMORY=`$AVERAGER $_VALGRIND_OUT_FILE`
      echo -e "$AVERAGE_MEMORY" > $_VALGRIND_OUT_FILE
      filename_memory="${1}_${3}_${4}_${5}_$_VALGRIND_FINAL_FILE"
      mv $_VALGRIND_OUT_FILE $RESULTS/$filename_memory
    else
      AVERAGE=`$AVERAGER $filename`
      echo -e "$AVERAGE" > $filename
      filename_result="${1}_${3}_${4}_${5}.txt"
      mv $filename $RESULTS/$filename_result
    fi
  fi
```

Come si può notare dallo snippet precedente l'algoritmo viene eseguito un numero di volte pari alle run settate ed è l'unico ciclo presente. Nel test alfa invece oltre al ciclo delle run è presente anche un ciclo con un numero di extent iniziali che deve raggiungere un numero di extent massimi incrementando quelli iniziali di un certo passo settato nel file *configuration.sh*. Oltre a questi due cicli è presente anche un terzo ciclo per testare tutti gli alfa presenti nell'array ALFAS sempre settato nel medesimo file di configurazione. Di seguito lo snippet:

```
  for ALFA in $ALFAS
  do
    EXTENTS=$START_EXTENTS
    filename="$1.txt"
    executed_filename="exec_time_$1_alfa_$ALFA.txt"
    executed_filename_memory="exec_memory_$1_alfa_${ALFA}_$_VALGRIND_FINAL_FILE"
    while [ $EXTENTS -le "$MAX_EXTENTS" ]
    do
      #for R in {1..$RUN}
      for ((R=1; R<=$RUN; R++))
      do
        echo "running >$1< $EXTENTS/$ALFA:$R"
        #If VALGRIND VARIABLE IS USED RUN WITH DIFFERENT COMMAND
        if [ "$3" = "mem" ];
        then
          valgrind --tool=massif --massif-out-file=$_VALGRIND_OUT_FILE ./$1 $2 $EXTENTS
              $DIMENSION $ALFA
          ms_print $_VALGRIND_OUT_FILE > "temp"
          unit=`cat "temp" | head -9 | tail -2 | head -1`
          unit=`echo $unit | cut -c1-2`
          temporaneous_variable=`cat "temp" | head -9 | tail -1 | awk '{print $1}'`
          temporaneous_variable=`echo "${temporaneous_variable//^}"`
          if [ "$unit" = "KB" ];
          then
            temporaneous_variable=`echo "$temporaneous_variable / 1024" | bc -l`
          elif [ "$unit" = "GB" ];
          then
            temporaneous_variable=`echo "$temporaneous_variable * 1024" | bc -l`
          fi
          echo $temporaneous_variable > $_VALGRIND_OUT_FILE
          rm -f "temp"
        else
          ./$1 $2 $EXTENTS $DIMENSION $ALFA $3
        fi
      done
      RESULT_FOLDER="${RESULTS}_${ALFA}_${EXTENTS}_${DIMENSION}"
      mkdir -p $RESULT_FOLDER
```

```
37      if [ "$3" = "dist" ];
38      then
39        mv $_BITMATRIX_NAME "$RESULT_FOLDER/${1}.bin"
40      else
41        if [ "$3" != "mem" ];
42        then
43          #average of time
44          AVERAGE=`$AVERAGER $filename`
45          echo -e "$AVERAGE" > $executed_filename
46          filename_result="${1}_${EXTENTS}_${DIMENSION}_${ALFA}.txt"
47          mv $filename $RESULT_FOLDER/$filename_result
48        else
49          #average of memory usage
50          AVERAGE_MEMORY=`$AVERAGER $_VALGRIND_OUT_FILE`
51          echo -e "$AVERAGE_MEMORY" > $executed_filename_memory
52          filename_memory="${1}_${EXTENTS}_${DIMENSION}_${ALFA}_$_VALGRIND_FINAL_FILE"
53          mv $executed_filename_memory $RESULT_FOLDER/$filename_memory
54        fi
55      fi
56      let EXTENTS+=$STEP_SIZE
57    done
58  done
```

L'alfa test è un metodo veloce e comodo per testare algoritmi con diverse quantità di update e subscription e diversi gradi di sovrapposizione. In caso di un numero elevato di update e subscription potrebbe essere necessario un ampio lasso di tempo prima che il test termini.

3.4.5 Rappresentazione dei risultati

Per la rappresentazione dei risultati si è utilizzato lo strumento Gnuplot. Gnuplot è un programma utile per tracciare grafici in due o tre dimensioni [11].

I grafici vengono creati tramite uno script che accede alla cartella *Results*, esplora tutte le cartelle dei vari test effettuati e utilizza i file con estensione .txt per creare grafici inerenti al tempo di esecuzione, utilizza i file con estensione .mem per creare grafici inerenti alla memoria utilizzata, utilizza i file con estensione .bin e li analizza tramite il programma *BitmatrixComparator* che compara la bitmatrix di ogni algoritmo eseguito con quella dell'algoritmo Brute Force, noto perché riesce a individuare con certezza la soluzione esatta.

```
1  cd $_RESULTS
2  local test=`ls`
3  for t in $test
4  do
5   #for each test folder
6   cd $t
```

Si entra nella cartella *Results* e si controlla quali test sono stati eseguiti. Per ogni test all'interno della cartella *Results* si creano le rappresentazioni grafiche.

Per la creazione dei dati da passare a gnuplot per il tempo di esecuzione:

```
7  #TIME
8   #for each algorithm result file in test folder
```

```
touch tmp
echo -e 'Algorithm_Name\tExecution_Time' > tmp
count=0
for a in `ls *.txt`
do
  #get the first value on text file
  time_test=`head -n 1 $a`
  #if a value of executed time exists
  if [ ! -z "$time_test" ];
  then

    ...

    echo -e "$name\t$time_test" >> tmp
    let count+=1
  fi
done
```

Si prendono tutti i file con estensione .txt e si prende valore nella prima riga di ogni file e lo si mette in un file temporaneo del tipo nome_algoritmo tempo_esecuzione. Questo file poi lo si passerà a gnuplot, successivamente verrà mostrato lo snippet gnuplot.

Per la creazione dei dati per l'utilizzo della memoria avviene la stessa identica cosa che viene fatta sul tempo di esecuzione con l'unica differenza che i file che si andranno a leggere saranno tutti quelli che avranno estensione .mem.

L'ultima fase di creazione dei dati è quella per la distanza dalla soluzione ottimale. In parte è uguale ai precedenti, la differenza fondamentale è come si ottiene il numero che indica la differenza tra due bitmatrix.

```
read_info_file "$testsinstances/$t/info.txt"
$utils/$_BITMATRIX_COMPARATOR "multi_dim_brute_force.bin" $a $_TEST_UPDATES
    $_TEST_SUBSCRIPTIONS
```

Tramite la prima riga di codice è possibile leggere il file *info.txt* all'interno della cartella *TestsInstances* nella cartella del test che si sta analizzando e aggiornare le variabili che contengono il numero di update e di subscription. La seconda riga richiama il programma *BitmatrixComparator* e gli vengono passati come argomenti la bitmatrix dell'algoritmo Brute Force, la matrice che si vuole comparare, il numero di update e infine il numero di subscription. Durante l'esecuzione del programma *BitmatrixComparator* viene creato un file *diff.txt* contenente il numero di sovrapposizioni di differenza tra le due bitmatrix.

Per la rappresentazione grafica in tutti e tre i tipi di test (tempo, memoria, distanza):

```
let offset=-5/$count
#create output file graph
gnuplot <<- EOF
  set title "`echo $t`"
  set xlabel "Algorithm Name"
  set ylabel "Execution Time (s)"
  set xtics rotate by 90 offset `echo $offset`, 1
  set ytics out nomirror
  set term png
  set output "result.png"
  set style fill solid border -1
  # Make the histogram boxes half the width of their slots.
  set boxwidth 0.5 relative
```

```
14      set grid front
15
16      # Select histogram mode.
17      set style data histograms
18      # Select a row-stacked histogram.
19      set style histogram rowstacked
20      plot "tmp" using 2:xticlabels(1) title 'Algorithm'
21          EOF
22    fi
23    rm -f tmp
24    mv result.png $graph/$t/result.png
```

In questo script si setta il titolo con il nome del test che si sta analizzando, in base al tipo di test che si sta eseguendo (esecuzione, memoria, distanza) vengono cambiati i nomi delle ascisse e delle ordinate. I nomi dei vari algoritmi vengono settati in verticale per una migliore comprensione. Viene settato il nome del file che deve essere creato e viene settata la tipologia di grafico, in questo caso si è scelto l'istogramma. Una volta creato il grafico viene spostato nell'apposita cartella all'interno della cartella *Graphs*

3.5 Libreria in C

La libreria in C offre definizioni, strutture, enumerazioni e funzioni in grado di far adattare al meglio gli algoritmi che si andranno a inserire all'interno del framework. Grazie a questa libreria un algoritmo adattato o creato da zero avrà la possibilità di ricevere un input uguale a quello che ricevono gli altri algoritmi e di creare una matrice di bit in output per effettuare il conteggio del numero di sovrapposizioni di update e subscription.

3.5.1 Definizioni e tipi di dati

Le definizioni utili che possono servire per implementare un algoritmo sono le seguenti:

- MAX_DIMENSIONS: definisce il numero massimo di dimensioni che può gestire il framework.

Si sono scelti tipi di dato molto capienti. Tutto ciò che deve contenere il lower e l'upper bound sono contenuti in variabili di tipo int64_t, permettendo di gestire numeri interi sia negativi dell'ordine di -2^{63} sia positivi $2^{63} - 1$, un bit è utilizzato per il segno. Le variabili che devono contenere solo numeri positivi, come il numero di extent, update e subscription, sono di tipo uint64_t e permettono di gestire valori solamente positivi dell'ordine di $2^{64} - 1$. Per il valore della dimensione si utilzza un tipo uint16_t che permette di gestire un valore massimo molto inferiore rispetto ai precedenti ma anche elevato per gestire il numero di dimensioni, pari a $2^{16} - 1$. La variabile che deve contenere il valore di alfa è di tipo virgola mobile a precisione singola perché non deve gestire numeri alti. Il tipo di dato **bitmatrix**, definito all'interno della libreria, dipende dalla soluzione che vuole utilizzare l'utente. Di default è un puntatore ad array di tipo uint32_t che

permette di gestire 32 bit. Il concetto della matrice di bit verrà ripreso in seguito per comprenderla al meglio.

3.5.2 Strutture

Le struttura utile per l'implementazione di un algoritmo è:

```
typedef struct DDM_Input{
  /*variables useful for any kind of test*/
  char type_test[100];
  char executable_name[100];
  uint16_t dimensions;
  uint64_t extents;
  uint64_t updates;
  uint64_t subscriptions;
  uint8_t write_to_file;
  /*variables useful for alfa test*/
  float alfa;
  /*variables useful for other kind of test*/
  DDM_Extent *list_updates;
  DDM_Extent *list_subscriptions;
  /*variable useful for timer*/
  DDM_Timer timer;
  /*variable useful to store result*/
  bitmatrix result_mat;
}DDM_Input;
```

All'interno di questa struttura si ha tutto il necessario per gestire i parametri passati all'eseguibile quando verrà richiamato. Contiene il tipo di test che si sta eseguendo, il nome dell'eseguibile, il numero di dimensioni utilizzate, il numero di extents totali (numero di update + subscription), il numero di update e di subscription, il valore di alfa (grado di sovrapposizione) e infine una variabile di tipo bitmatrix per gestire al meglio il risultato finale. La variabile write_to_file viene settata a uno se il tipo di test che si sta eseguendo è quello per verificare la distanza dalla soluzione ottima. Se è settata a uno si scrive la bitmatrix su file binario. Questa scelta è stata presa in quanto scrivere una matrice molto grande su file impiega un tempo non trascurabile.

Questa struttura, come si può notare, all'interno fa uso di un'altra struttura che non si dovrà utilizzare manualmente all'interno dell'algoritmo. Grazie a delle funzioni apposite il timer viene gestito dalla variabile di tipo DDM_Input. Sintetizziamo di seguito la struttura DDM_Timer:

```
typedef struct DDM_Timer{
  clock_t start, end;
  float total;
} DDM_Timer;
```

Questa struttura contiene due variabili, start e end, di tipo clock_t e una variabile total per gestire il tempo totale di esecuzione di tipo float.

È probabile che sia necessaria la dichiarazione di tipo DDM_Extent per ricevere l'array di update e di subscription tramite funzione, quindi qui di seguito viene spiegata la struttura.

```
typedef struct DDM_Extent{
  uint64_t id;
  int64_t lower[MAX_DIMENSIONS];
  int64_t upper[MAX_DIMENSIONS];
}DDM_Extent;
```

Una variabile dichiarata di questo tipo può contenere l'id dell'update o subscription e i suoi lower e upper bound. Le variabili lower e upper al suo interno sono degli array per permettere l'utilizzo di più dimensioni fino ad un massimo di MAX_DIMENSIONS.

3.5.3 Enumerazioni

È definito un solo tipo di enumerazione all'interno della libreria ed è il seguente:

```
typedef enum{
  zero,
  one
} mat_value;
```

Questa enumerazione permette di usare i suoi valori all'interno di funzioni, quali bitmatrix_set_value e bitmatrix_reset, le quali necessitano di ricevere come ultimo parametro un valore di questa enumerazione. Il motivo della sua definizione è per il semplice fatto di evitare banali errori.

3.5.4 Funzioni per ricevere un input coerente

Di seguito verranno elencati i prototipi delle funzioni utili da utilizzare per ricevere l'input dal framework.

- DDM_Input* DDM_Initialize_Input(int argc, char* argv[]): questa funzione restituisce un riferimento alla memoria allocata all'interno della funzione contenente tutte le variabili al suo interno inizializzate in base ai valori ottenuti dai parametri passati al programma. Questa funzione è la prima ad essere richiamata nel main del programma, se non viene richiamata non verrà inizializzata la variabile di tipo DDM_Input e quindi non si potrà avere accesso ai dati passati dal framework;
- uint64_t DDM_Get_Extents(DDM_Input ddm_input): restituisce il numero di extent, subscription + update;
- uint16_t DDM_Get_Dimensions(DDM_Input ddm_input): restituisce il numero di dimensioni utilizzate nel test;
- uint64_t DDM_Get_Updates(DDM_Input ddm_input): restituisce il numero di update;
- uint64_t DDM_Get_Subscriptions(DDM_Input ddm_input): restituisce il numero di subscription;

- DDM_Extent* DDM_Get_Updates_List(DDM_Input ddm_input): restituisce un riferimento alla memoria allocata all'array degli update;
- DDM_Extent* DDM_Get_Subscriptions_List(DDM_Input ddm_input): restituisce un riferimento alla memoria allocata all'array di subscription;

3.5.5 Funzioni utili per gestire la bitmatrix temporanea

Qui di seguito verranno analizzate le funzioni utili per garantire un corretto utilizzo della bitmatrix di appoggio.

- void bitmatrix_init(bitmatrix *mat, uint64_t updates, uint64_t subscriptions): alloca uno spazio in memoria per gestire una matrice di righe equivalenti al parametro updates e di $\lceil \frac{subscriptions}{BITS\_NUMBER} \rceil$ colonne. Il vantaggio di utilizzare una bitmatrix al posto di una normale matrice dipende dal numero di bit della memoria allocata. Un esempio può essere questo: si ha che ogni cella della matrice è di tipo uint32_t, cioè 32 bit. Se si hanno mille update e mille subscription si avrà un numero di righe pari a mille con un numero di colonne pari a $\lceil \frac{1000}{32} \rceil$ che equivale a 32. Se avessimo utilizzato una normale matrice si sarebbe occupata sicuramente più memoria e le performance sarebbero drasticamente peggiori. Le operazioni su una matrice di questo tipo sono fatte bit per bit e sono le operazioni logiche: AND, OR, XOR, NOT;
- void bitmatrix_set_value(bitmatrix mat, uint64_t update, uint64_t subscription, mat_value value): viene settato un singolo bit in base al valore specificato dal parametro *value* che può essere *zero* o *one*;
- void bitmatrix_and(bitmatrix mat, const bitmatrix mask, uint64_t updates, uint64_t subscriptions): viene effettuata l'operazione logica AND tra due matrici e il risultato è salvato all'interno del primo parametro *mat*;
- void bitmatrix_or(bitmatrix mat, const bitmatrix mask, uint64_t updates, uint64_t subscriptions): viene effettuata l'operazione logica OR tra due matrici e il risultato è salvato all'interno del primo parametro *mat*;
- void bitmatrix_xor(bitmatrix mat, const bitmatrix mask, uint64_t updates, uint64_t subscriptions): viene effettuata l'operazione logica XOR tra due matrici e il risultato è salvato all'interno del primo parametro *mat*;
- void bitmatrix_not(bitmatrix mat, uint64_t updates, uint64_t subscriptions): viene effettuata l'operazione logica NOT di ogni bit della matrice;

- void bitmatrix_reset(bitmatrix mat, uint64_t updates, uint64_t subscriptions, mat-_value value): viene resettata la matrice con il valore passato tramite il parametro *value*, può essere *zero* o *one*;

- void bitmatrix_free(bitmatrix *mat, uint64_t updates, uint64_t subscriptions): viene deallocata la memoria puntata dal parametro *mat*.

Sono presenti anche delle utili funzioni di debug:

- void bitmatrix_print(bitmatrix mat, uint64_t updates, uint64_t subscriptions): stampa tutti gli elementi della matrice (0 o 1) con una rappresentazione matriciale;

- uint64_t bitmatrix_count_ones(bitmatrix mat, uint64_t updates, uint64_t subscriptions): restituisce il numero di bit settati a uno;

- void bitmatrix_print_matches(const bitmatrix mat, uint64_t updates, uint64_t subscriptions): stampa in colonna tutte le sovrapposizioni in formato (U0, S0).

3.5.6 Funzioni per creare un output coerente

Di seguito verrà elencato l'unico prototipo utile della funzione da utilizzare per creare un risultato coerente con il framework.

- void DDM_Write_Result(DDM_Input *ddm_input): crea un file di testo dove viene specificato nella prima riga il tempo totale di esecuzione. Se il parametro ddm_input ha il campo write_to_file settato a uno allora creerà anche il file .bin contenente la matrice di bit.

3.5.7 Template di utilizzo delle librerie

```
#include <stdio.h>
#include <stdlib.h>
#include "DDM_input_output.h"
int main(int argc, char *argv[])
{
  uint64_t updates, subscriptions;
  uint16_t dimensions;
  DDM_Extent *list_updates, *list_subscriptions;
  //DDM's variables
  DDM_Input *ddm_input;
  //Indexes
  uint16_t k;
  //temporaneous result matrix
  bitmatrix temp;
  //Initialize variable of DDM
  ddm_input = DDM_Initialize_Input(argc, argv);
  //Check if the initialization of input was succesfully
  if (ddm_input == NULL)
      exit(-1);
```

```
20    updates = DDM_Get_Updates(*ddm_input);
21    subscriptions = DDM_Get_Subscriptions(*ddm_input);
22    dimensions = DDM_Get_Dimensions(*ddm_input);
23    list_subscriptions = DDM_Get_Subscriptions_List(*ddm_input);
24    list_updates = DDM_Get_Updates_List(*ddm_input);
25    //create temporaneous matrix
26    bitmatrix_init(&temp, updates, subscriptions);
27    DDM_Start_Timer(ddm_input);
28    for (k = 0; k < dimensions; ++k){
29      if (k > 0){
30        //each time execute different dimension
31        //reset the temp matrix
32        bitmatrix_reset(temp, updates, subscriptions, zero);
33
34        //Execute Algorithm Here **
35
36        //Intersect temp matrix and ddm_input->result_mat and store result in ddm_input->
              result_mat
37        bitmatrix_and(ddm_input->result_mat, temp, updates, subscriptions);
38      }else{
39        //Execute algorithm and store result into ddm_input->result_mat **
40      }
41    }
42    DDM_Stop_Timer(ddm_input);
43    printf("\nnmatches: %"PRIu64"\n", bitmatrix_count_ones(ddm_input->result_mat, updates,
          subscriptions));
44    //Write result
45    DDM_Write_Result(ddm_input);
46    return 0;
47  }
```

Da questo frammento di codice si nota che la prima funzione ad essere chiamata è DDM_Initialize_Input e grazie a questa funzione si inizializza tutta la struttura, successivamente si richiamano tutte le funzioni per ottenere extent, update, subscription, ecc. Se l'algoritmo che si vuole implementare necessita di una matrice di appoggio per segnare tutte le sovrapposizioni allora è necessario inizializzare la matrice di appoggio tramite bitmatrix_init. Successivamente si può far partire il timer ed eseguire l'algoritmo. Per ogni dimensione, tranne la prima, è necessario azzerare la matrice di appoggio per non avere dei match errati e dopo aver eseguito l'algoritmo si deve fare l'intersezione tra la matrice temporanea e quella dove è memorizzata il risultato tramite l'operazione di AND logico. Finita l'esecuzione dell'algoritmo per ogni dimensione si ferma il timer. Se si vuole fare il debug e verificare che il numero di match sia corretto si può stampare il numero di match. Infine si salva il risultato su disco e nel caso il test sia quello sulla distanza dalla soluzione ottima viene salvata su disco la bitmatrix.

In questo capitolo sono state affrontate tutte le scelte implementative che riguardano il framework, le sue opzioni e la sua libreria di appoggio. Nel prossimo capitolo verrà affrontata una panoramica sugli algoritmi presenti in letteratura, esponendo quali sono stati implementati all'interno del framework.

Capitolo 4

Algoritmi di Data Distribution Management

Come già accennato in precedenza, rilevare le sovrapposizioni tra update e subscription è un'operazione molto costosa. Per questo motivo la comunità scientifica che si occupa di simulazione parallela e distribuita si sono ingegniate per creare algoritmi più o meno efficienti rispetto al classico Brute Force.

In letteratura sono presenti algoritmi fondamentalmente di due tipologie di approccio:

- **region-based**: questa tipologia di algoritmi permette di sapere con esattezza il numero di match esistenti tra update e subscription. Sono algoritmi più complessi a confronto del banale Brute Force ma che permettono prestazioni decisamente migliori con il crescere del numero di update e subscription presenti. Questi algoritmi nel caso peggiore svolgono tutti un numero di confronti pari a O(U x S) dove U è il numero di update e S il numero di subscription (Figura 4.1).

- **grid-based**: questa tipologia di algoritmo non fornisce una soluzione ottima al problema, il più delle volte fornisce un numero di sovrapposizioni più alto. Questo accade in quanto il Routing Space viene diviso in celle il cui valore viene definito staticamente o dinamicamente. Il matching, in questa tipologia, è rilevato nel caso in cui sia l'update che il subscription siano parte della cella. Sono presenti articoli scientifici che discutono come trovare la dimensione adatta della cella, in realtà non esiste la certezza di una dimensione migliore rispetto alle altre (Figura 4.2).

Esiste un terzo approccio che comprende sia l'utilizzo di un algoritmo region-based sia di uno grid-based, viene chiamato approccio **ibrido**. In questo caso si otterranno i vantaggi di entrambi i tipi di approccio, la scalabilità da parte del grid-based e il numero di match esatto dal region-based. Purtroppo porta con se anche uno svantaggio, la scelta della dimensione della cella con cui suddividere il Routing Space.

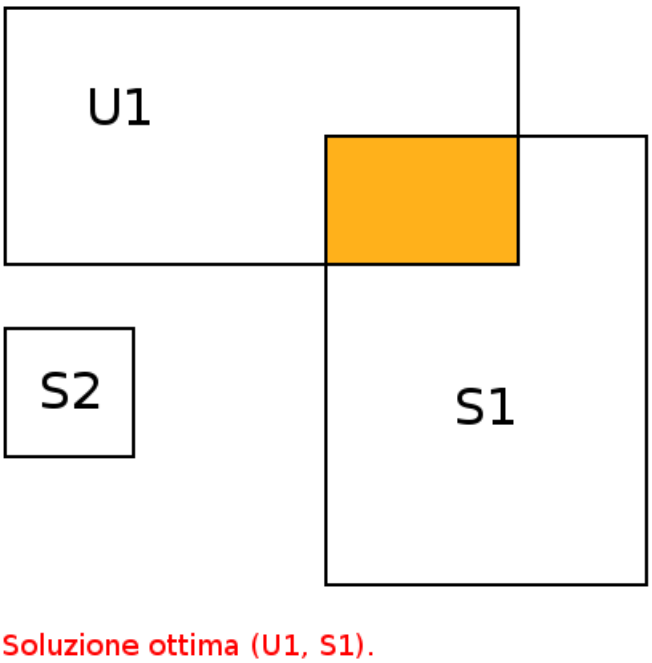

Figura 4.1: Esempio di sovrapprosizione tra un update e due subscription nell'approccio region-based

4.1 Proiezione della dimensione

Per facilitare la comprensione e la risoluzione degli algoritmi è stato ideato un approccio chiamato **dimension projection**. Grazie a questo approccio non è necessario implementare un algoritmo per la risoluzione multidimensionale ma è sufficiente implementare la versione monodimensionale. Questo perché risolvendo una dimensione per volta è sufficiente implementare un'operazione di AND tra la dimensione che si è appena risolta e quella precedente avendo così un numero di sovrapposizioni parziali. Una volta finita la risoluzione di tutte le dimensioni si avrà il numero di sovrapposizioni esatto (Figura 4.3).

4.2 Algoritmi all'interno del framework

Nel framework sono stati implementati quattro algoritmi, tutti sono della tipologia region-based, tra cui il Brute Force che è utile per verificare se la soluzione degli altri algoritmi è ottima.

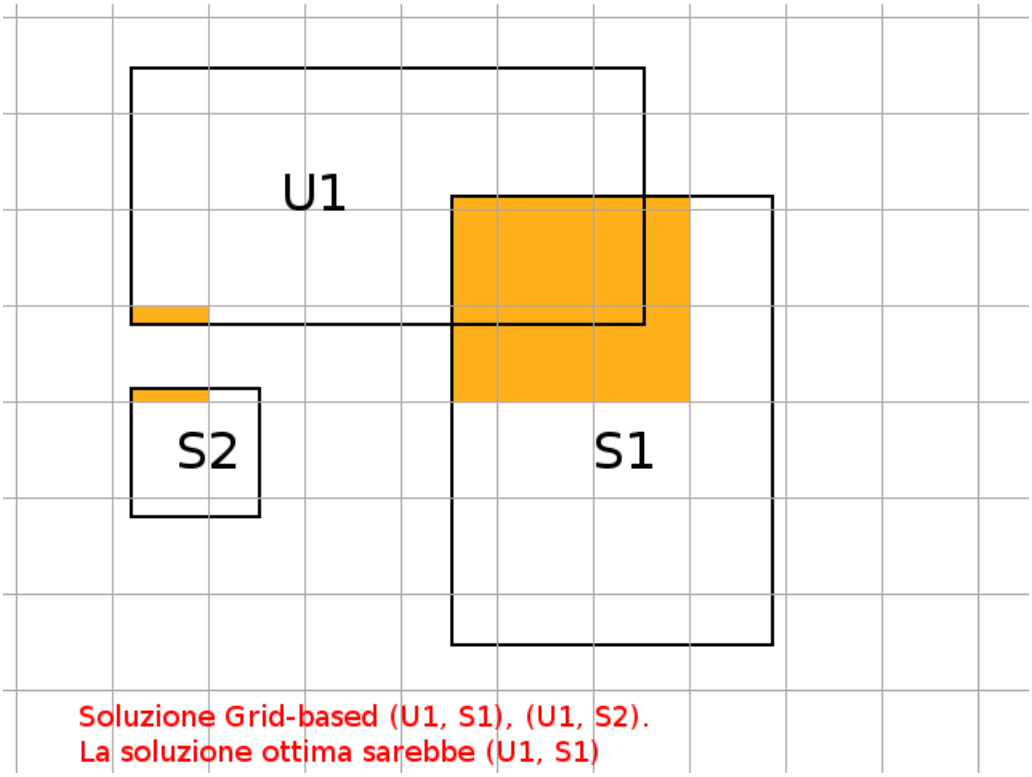

Figura 4.2: Esempio di sovrapprosizione tra un update e due subscription nell'approccio grid-based

- **Brute Force**: questo algoritmo è in grado di rilevare con esattezza tutte le sovrapposizioni presenti tra update e subscription. La sua semplicità di implementazione viene ripagata con scarsa scalabilità e lentezza nell'esecuzione. Il suo ordine di grandezza è quadratico, più precisamente O(U x S) dove U è il numero di update e S il numero di subscription. Il suo funzionamento è semplice, sequenzialmente si confronta ogni update presente nel Routing Space con ogni subscription.

  Vantaggi:

  - implementazione banale;
  - parallelizzabile in modo imbarazzante.

  Svantaggi:

  - scarsa scalabilità;
  - lentezza nell'esecuzione.

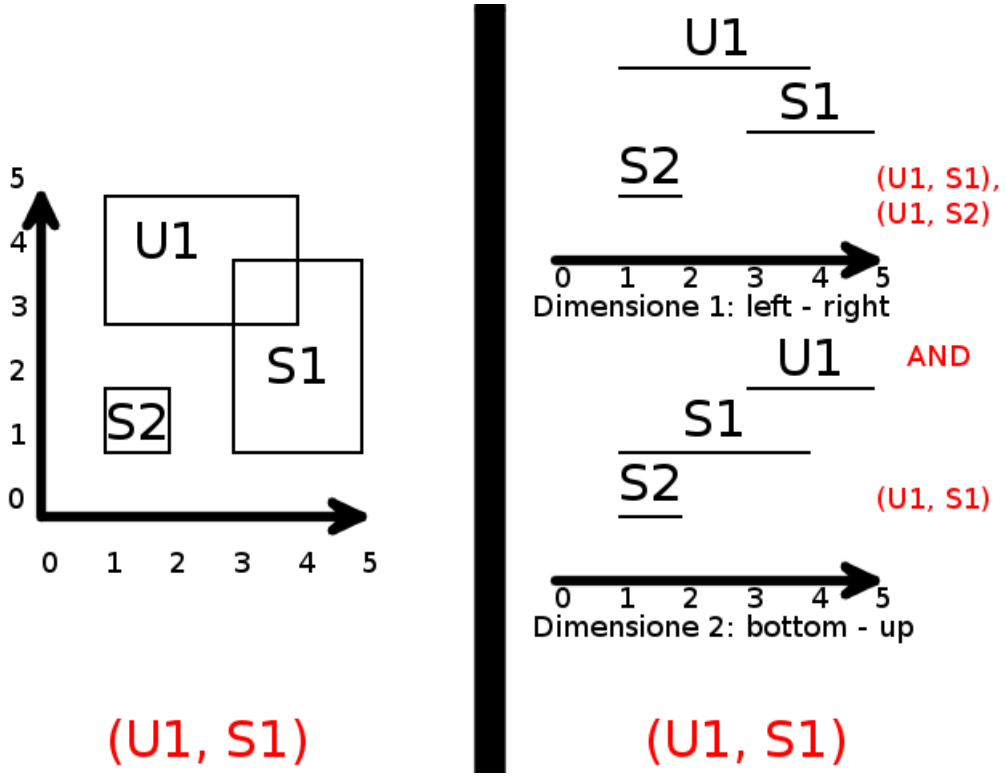

Figura 4.3: Esempio di Dimension Projection con un update e due subscription

- **Binary Partition**: questo algoritmo utilizza un approccio definito chiamato *dividi et impera*. Fondamentalmente utilizza due parti per completare la risoluzione:
  - **processo di partizionamento**: l'algoritmo chiama in modo ricorsivo il partizionamento binario che divide in due parti tutte le region presenti;
  - **processo di matching**: l'algoritmo utilizza il concetto di una relazione ordinata che rappresenta la posizione relativa all'interno di una determinata posizione.

  Teoricamente l'algoritmo promette bassi costi computazionali poiché calcola facilmente l'intersezione tra regioni presenti all'interno di una determinata partizione. Questo algoritmo ha una complessità computazionale di $n^2 * O(logN)$, dove n è il numero di region per ogni partizione e N è il numero di region totali. Questo significa che si effettueranno $logN$ partizionamenti e che ognuno ha come complessità n^2 per poter effettuare i confronti tra la partizione dov'è contenuto il pivot con la partizione di sinistra e con la partizione di destra. La partizione di sinistra e di destra non vengono confrontate perché non sarebbe sensato in quanto nella partizione di destra fanno parte gli extent che hanno il lower bound maggiore rispetto

all'upper bound della partizione di pivot mentre nella partizione di sinistra fanno parte gli extent che hanno l'upper bound inferiore rispetto al lower bound della partizione di pivot [12].

Vantaggi:

- **scalabilità**: riesce ad adattarsi anche ad un numero di extent molto elevato;
- **tempo di esecuzione**: nel caso migliore può avere un tempo di esecuzione molto basso. Nel caso peggiore potrebbe degenerare e l'esecuzione potrebbe scendere ai livelli del Brute Force;

Svantaggi:

- **occupazione in memoria**: in alcuni casi, soprattutto quando ci si avvicina ad una esecuzione simile a quella del Brute Force, l'occupazione di memoria cresce notevolmente per via delle numerose chiamate ricorsive.

- **Interval Tree Matching**: questo algoritmo è basato sulla struttura dati *interval tree*. L'implementazione adottata è un albero auto bilanciato modificato, questa soluzione non è la più efficiente ma la più semplice da realizzare [13]. È una struttura interessante per via dei suoi costi nell'inserimento e nell'eliminazione di un nodo che è $O(logN)$ e soprattutto può rilevare tutte le intersezioni k dato un prefissato intervallo di query in $O(k+logn)$. L'Interval Tree Matching inizialmente crea l'albero con i valori delle subscription region, una volta creato, per ogni update viene fatta una query sull'albero e si cercano le intersezioni, per questo motivo il costo della operazione di query dev'essere basso. Il costo computazione dell'Interval Tree Matching è $O(min(m * n, (K + 1) * logn)$ dove m sono il numero di update e n sono il numero di subscription, K è il numero totale di intersezioni. Nel caso peggiore degenera al costo del Brute Force.

  Vantaggi:

  - **parallelizzazione imbarazzante**: Si possono fare più query all'albero parallelamente riducendo il tempo di esecuzione;
  - **scalabilità**: riesce ad adattarsi anche ad un numero di extent molto elevato.

- **Improved Sort-Based**: questo algoritmo è stato implementato da un tesista precedente [14]. È una versione ottimizzata del classico sort-based. Il procedimento che utilizza questo algoritmo sfrutta il fatto che i subscription possono solamente seguire, precedere o essere sovrapposti ad un dato update. Inizialmente crea una lista contenente i vari lower e upper bound di ogni extent, dopodiché crea due insiemi chiamati SubscriptionSetBefore e SubscriptionSetAfter. Il primo sarà vuoto inizialmente mentre il secondo conterrà tutti i subscription. L'algoritmo quindi

inizia a estrarre dalla lista. Un subscription viene rimosso dall'insieme SubscriptionSetAfter nel caso in cui venga estratto il suo lower bound. Un subscription viene inserito nell'insieme SubscriptionSetBefore nel caso in cui venga estratto il suo upper bound. Quando viene estratto il lower bound di un update si può affermare che tutti i subscription contenuti all'interno dell'insieme SubscriptionSetBefore siano prima di lui, mentre se viene estratto l'upper bound di un update si può affermare che tutti i subscription presenti nell'insieme SubscriptionSetAfter siano dopo di lui. Salvando queste informazioni su una matrice è possibile avere a fine esecuzione una matrice di non-matching e applicandogli un'operazione di NOT si ottiene la matrice di sovrapposizione. Il costo di questo algoritmo è dato principalmente dall'operazione di ordinamento che è di ordine subquadratico $O(N * logN)$. Le operazioni successive si può dire che sono ammortizzate essendo operazioni di bitwise sono tra le più veloci.

Vantaggi:

- **scalabilità**: riesce ad adattarsi anche ad un numero di extent molto elevato;
- **tempo di esecuzione**: le operazioni di bitwise possono ridurre notevolmente il tempo di esecuzione.

Svantaggi:

- **non parallelizzabile**: siccome fa parte della famiglia degli ordinamenti è difficile da parallelizzare.

Capitolo 5

Utilizzo del Cloud

Si sono analizzati i servizi offerti di due delle più grandi compagnie a livello mondiale prima di eseguire i vari test che saranno mostrati in seguito: Amazon AWS (Amazon Web Services) e Google App Engine [15]. Non verrà fatta una panoramica su tutti i servizi ma solamente sul servizio che si è analizzato per poter poi eseguire i test, saranno illustrati i vantaggi e gli svantaggi.

Tutte le considerazioni che saranno fatte di seguito sono basate sulle opzioni Free Trial di entrambi i servizi.

- **Amazon AWS**: si è preso in considerazione il servizio EC2 (Elastic Cloud Computing) [16]. Offre un'interfaccia semplice e intuitiva e sono presenti varie AMI (Amazon Machine Images). Una AMI è un template che contiene una configurazione software (sistema operativo, applicativi server e applicazioni varie) richieste per lanciare un'istanza. Come citato prima sono presenti una vasta gamma di AMI con diversi sistemi operativi: Windows Server, Red Hat Enterprise, Ubuntu, SuSE Linux. Quasi la totalità delle AMI sono con supporto SSD (State Solid Disk) e architettura a 64 bit.

  Nel momento della selezione del tipo di processore da utilizzare, avendo un account free, è limitato ad un solo tipo di processore della famiglia "General Purpose" e di tipo t2.micro. t2.micro ha una CPU Intel Xeon 2.5GHz e solamente 1GB di memoria, molto ridotta per eseguire dei test su simulazioni con un numero elevato di extent. Per utilizzare questo servizio è necessario scaricare un file contenente la chiave privata per potersi collegare al terminale remoto tramite SSH. È offerto anche un servizio tramite browser installando la JRE (Java Runtime Environment).

  Vantaggi:

  - **interfaccia**: interfaccia semplice e comoda, spiegata in maniera eccellente;
  - **AMI**: un numero elevato di tipi di macchine anche con l'account free;
  - **durata account free**: la durata dell'account free è di dodici mesi.

Svantaggi:

- **componentistica hardware**: l'hardware reso disponibile per un account free non è sufficiente per eseguire il framework con un numero elevato di extent. Singolo processore e RAM da solo un 1GB;
- **connessione**: più ostico rispetto al servizio offerto da Google App Engine come successivamente verrà mostrato.

- **Google App Engine**: l'account free di GAE (Google App Engine) ha la durata di sessanta giorni e in questi si hanno a disposizione 300$ da utilizzare all'interno di esso. Il servizio testato si chiama *Compute Engine VM* e permette di creare istanze di macchine virtuali con varie specifiche. I sistemi operativi disponibili sono inferiori rispetto a quelli di AWS EC2: Debian, OpenSuSE, RedHat Enterprise. Per quanto riguarda la componentistica hardware è disponibile una maggiore personalizzazione. Sono disponibili diversi tipi di macchine con uno o due processori e con differenti tagli di RAM. Una volta creata l'istanza è possibile monitorare il traffico di rete, l'utilizzo della CPU, lettura/scrittura su disco. Offre un comodo e facile metodo di utilizzo, è possibile accedere direttamente al terminale remoto tramite browser Chrome, senza scaricare chiavi pubbliche e private ne utilizzare i comandi manuali SSH da terminale.

  Vantaggi:

  - **componentistica hardware**: l'hardware reso disponibile è vario e alla portata della maggior parte dei processi che il framework deve eseguire.
  - **connessione**: connessione semplice tramite browser Chrome o da terminale tramite comando SSH.

  Svantaggi:

  - **durata account free**: rispetto ad AWS EC2 è il tempo a disposizione è ridotto a sessanta giorni.

Concludendo si può dire che per eseguire al meglio il framework AWS EC2 non è abbastanza potente a livello hardware quindi la scelta ricade su Google App Engine.

5.1 Creazione della macchina virtuale

Durante la creazione della macchina virtuale viene chiesto quale tipo di sistema operativo, processore, memoria e disco utilizzare. Di seguito saranno elencate le specifiche:

- **sistema operativo**: Debian GNU/Linux 7 (wheezy);

- **tipo macchina**: n1-standard-2 (2 vCPU, 7.5 GB memory);
- **disco**: 10GB.

A macchina creata è necessario installare i pacchetti utili all'esecuzione del framework.

- **git**: è uno strumento utile per scaricare il codice da una data sorgente. Tool è un software di controllo di versione distribuita;
- **gcc**: GNU C Compiler (GCC) è un compilatore open source del linguaggio C;
- **make**: utile per automatizzare il processo di compilazione dei sorgenti in C tramite Makefile;
- **valgrind**: framework per analizzare la memoria utilizzata dai programmi in esecuzione. Utile per rilevare memory leak. Nel caso del framework utile per ottenere la memoria di picco di una data esecuzione;
- **bc**: utile tool per poter eseguire calcoli da bash;
- **gnuplot**: tool per la creazione di grafici in due e tre dimensioni;
- **libc6-dev-i386**: libreria per la compatibilità dei sistemi a 64 bit per compilare programmi in C.

Tutti i pacchetti citati sopra si possono installare tramite la seguente riga di codice da terminale:

```
1 sudo apt-get install git gcc make valgrind bc gnuplot libc6-dev-i386
```

Successivamente è possibile recuperare il framework con all'interno gli algoritmi tramite il tool git:

```
1 git clone https://github.com/nicholasricci/DDM_Framework.git
```

Grazie a questo approccio Cloud è possibile eseguire dei test da utenti diversi e infine poterli confrontare.

Capitolo 6

Valutazione delle prestazioni

In questo capitolo saranno mostrati i risultati degli algoritmi eseguiti all'interno del framework. Verrà descritta la macchina virtuale sul Cloud utilizzata con le specifiche utilizzate.

Specifiche della macchina:

- **Distribuzione Linux**: Debian GNU/Linux 7 (wheezy);
- **CPU**: Intel(R) Xeon(R) CPU @ 2.60GHz x 2;
- **RAM**: 7,5GB;

Tutti gli algoritmi presentati sono parte dell'approccio region-based e restituiscono la soluzione ottima per questo motivo non è stato inserito il grafico relativo della distanza dalla soluzione ottima ma solo quelli relativi al tempo di esecuzione e al consumo di memoria di picco.

Il framework crea in automatico i seguenti grafici grazie al supporto del tool gnuplot. Sono grafici a istogramma, l'asse dell'ascissa rappresenta il nome dell'algoritmo utilizzato e l'asse delle ordinate rappresenta il tempo di esecuzione o il consumo di memoria di picco.

I nomi degli algoritmi non sono comprensibili perché quando si creano i file in automatico vengono prese le prime lettere di ogni parola. Esempio se si prende in considerazione l'algoritmo Interval Tree, il suo nome avrà come iniziali it. Da sinistra verso destro in ordine si ha: Binary Partition, Improved Sort, Interval Tree Matching, Brute Force.

Algoritmo	Nome nei grafici
Binary Partition	bp
Improved Sort	is
Interval Tree Matching	it
Brute Force	mdbf

Tabella 6.1: Tabella associazione nome algoritmo e nome utilizzato nei grafici

Tutti gli alfa test riportati di seguito sono stati svolti con 3 dimensioni.

6.1 Alfa 0.01 - Extent 100000, 200000, 300000

Numero di extent preso in esame: 100000.

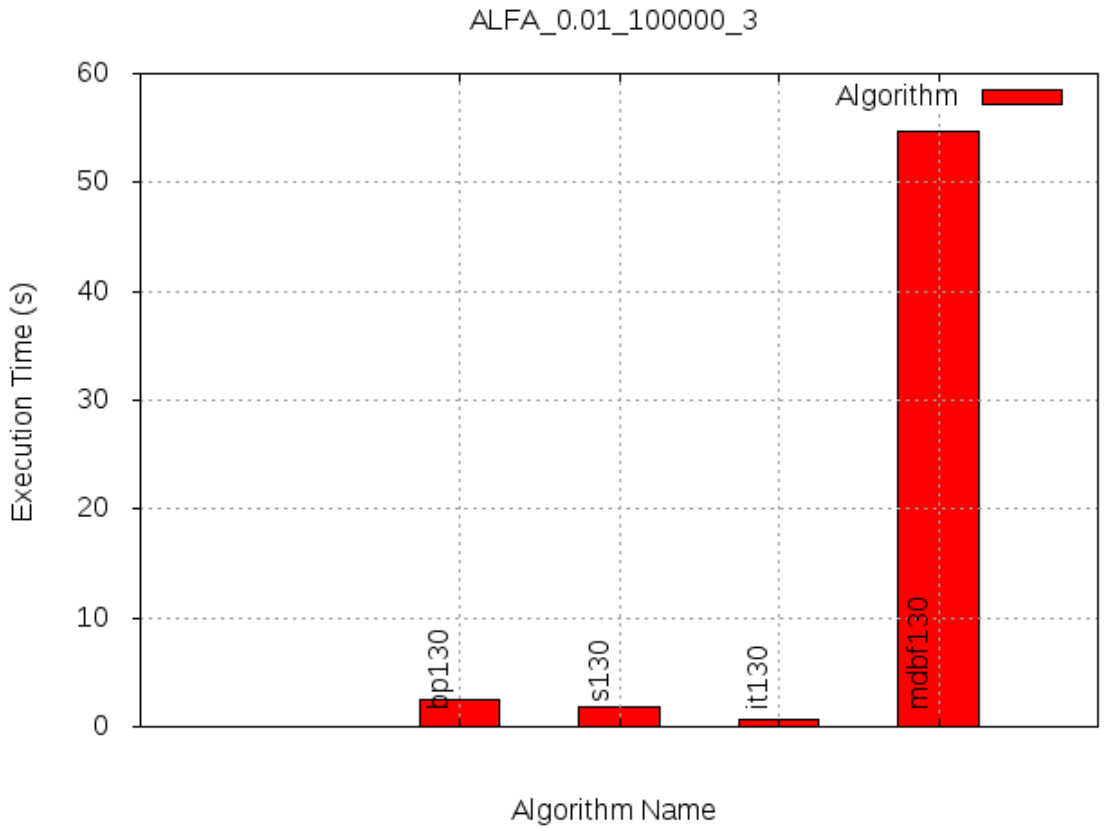

Figura 6.1: Tempo di esecuzione - alfa test con alfa 0.01, numero di extent 100000 e 3 dimensioni

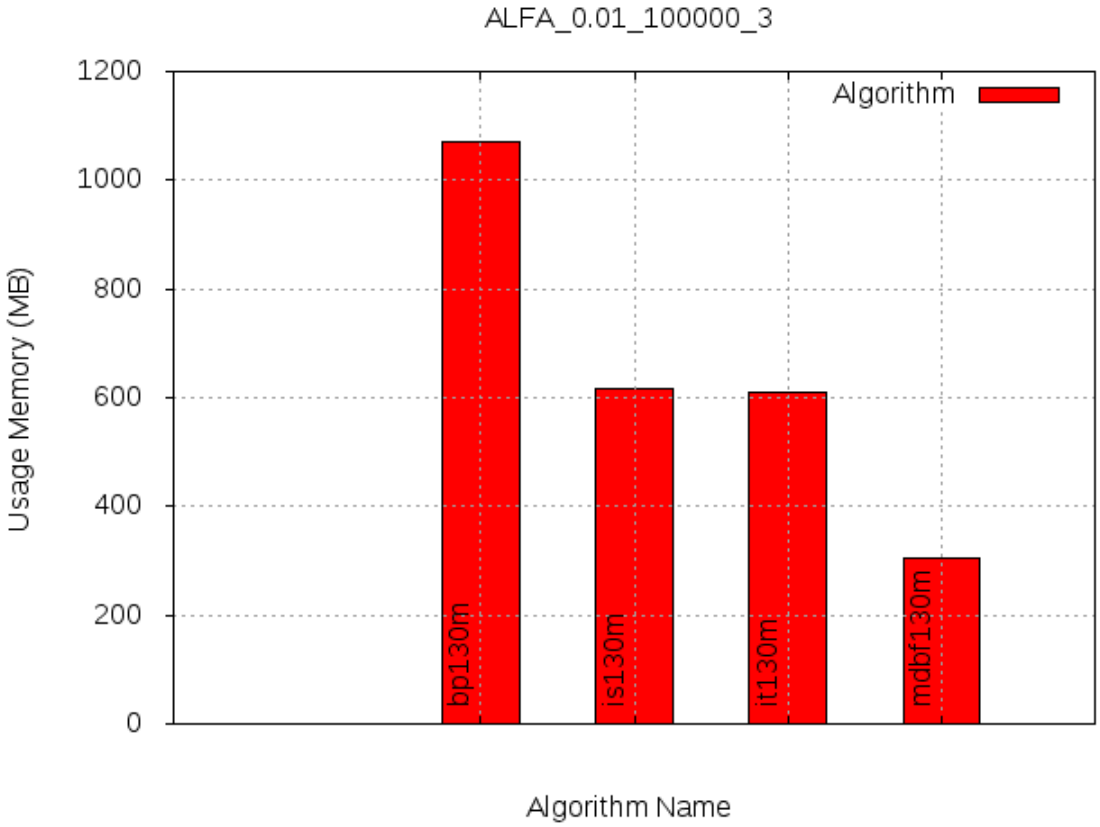

Figura 6.2: Memoria di picco - alfa test con alfa 0.01, numero di extent 100000 e 3 dimensioni

Numero di extent preso in esame: 200000.

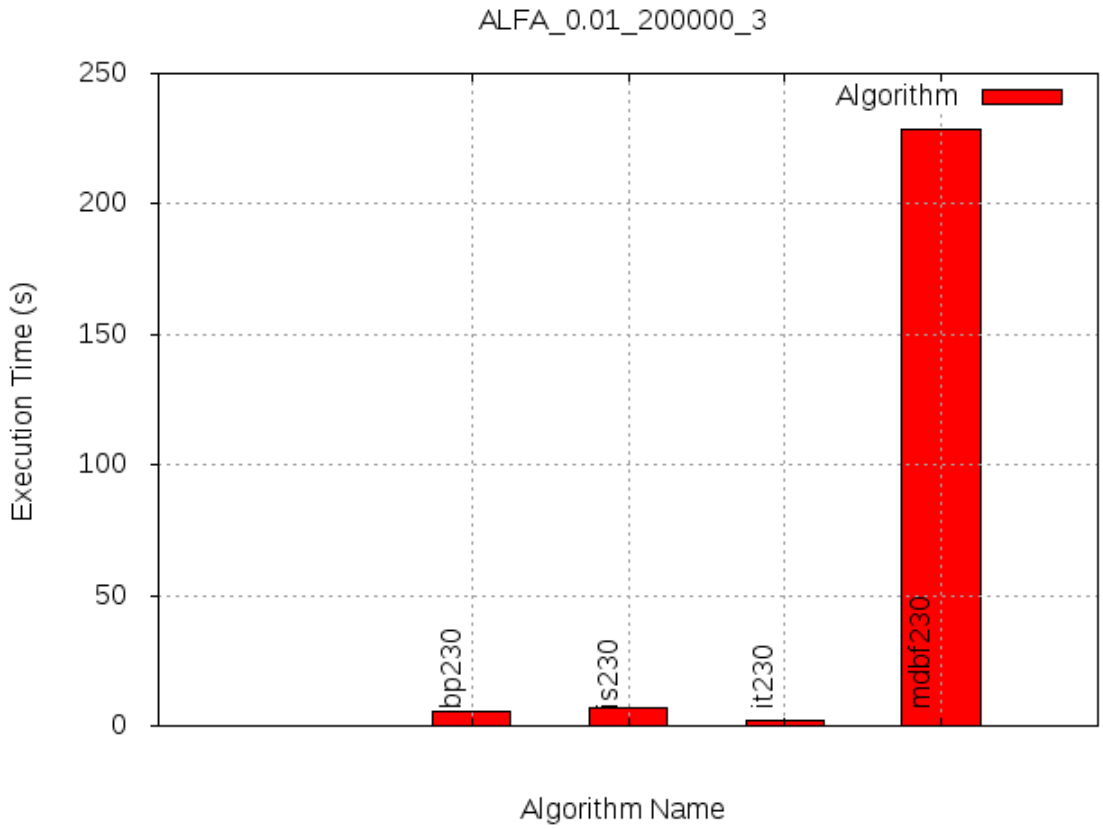

Figura 6.3: Tempo di esecuzione - alfa test con alfa 0.01, numero di extent 200000 e 3 dimensioni

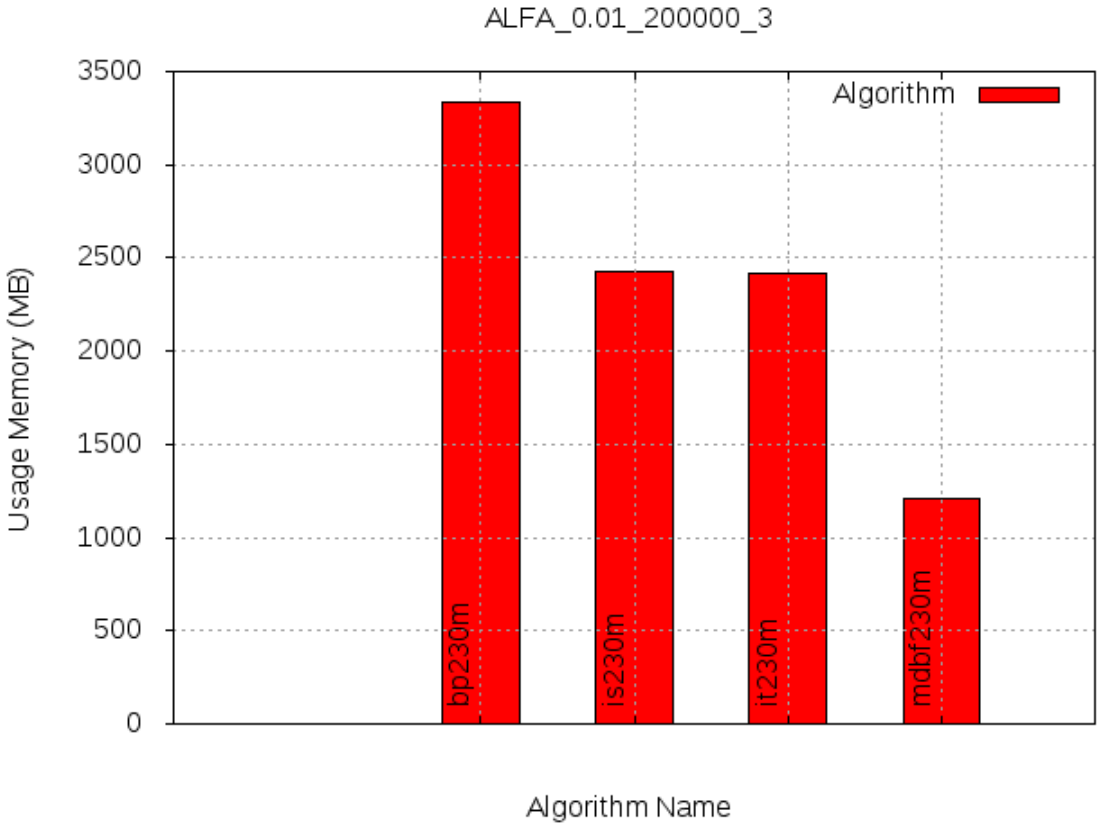

Figura 6.4: Memoria di picco - alfa test con alfa 0.01, numero di extent 200000 e 3 dimensioni

Numero di extent preso in esame: 300000.

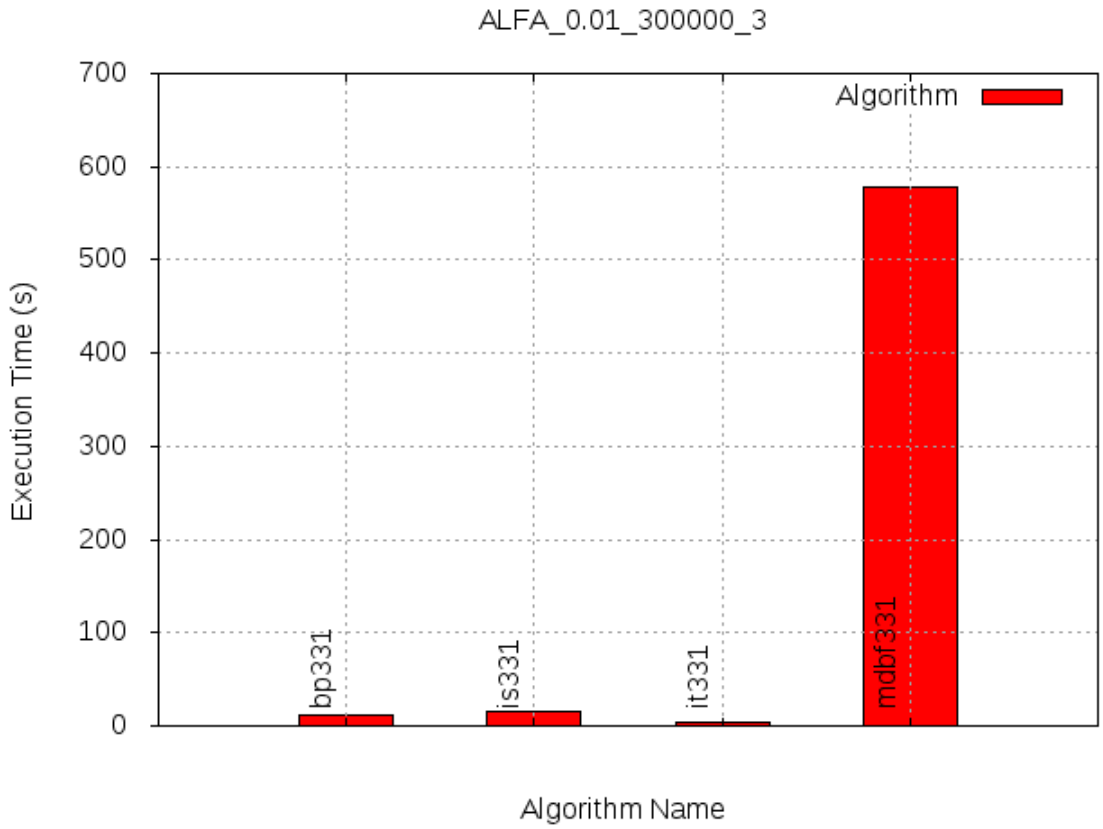

Figura 6.5: Tempo di esecuzione - alfa test con alfa 0.01, numero di extent 300000 e 3 dimensioni

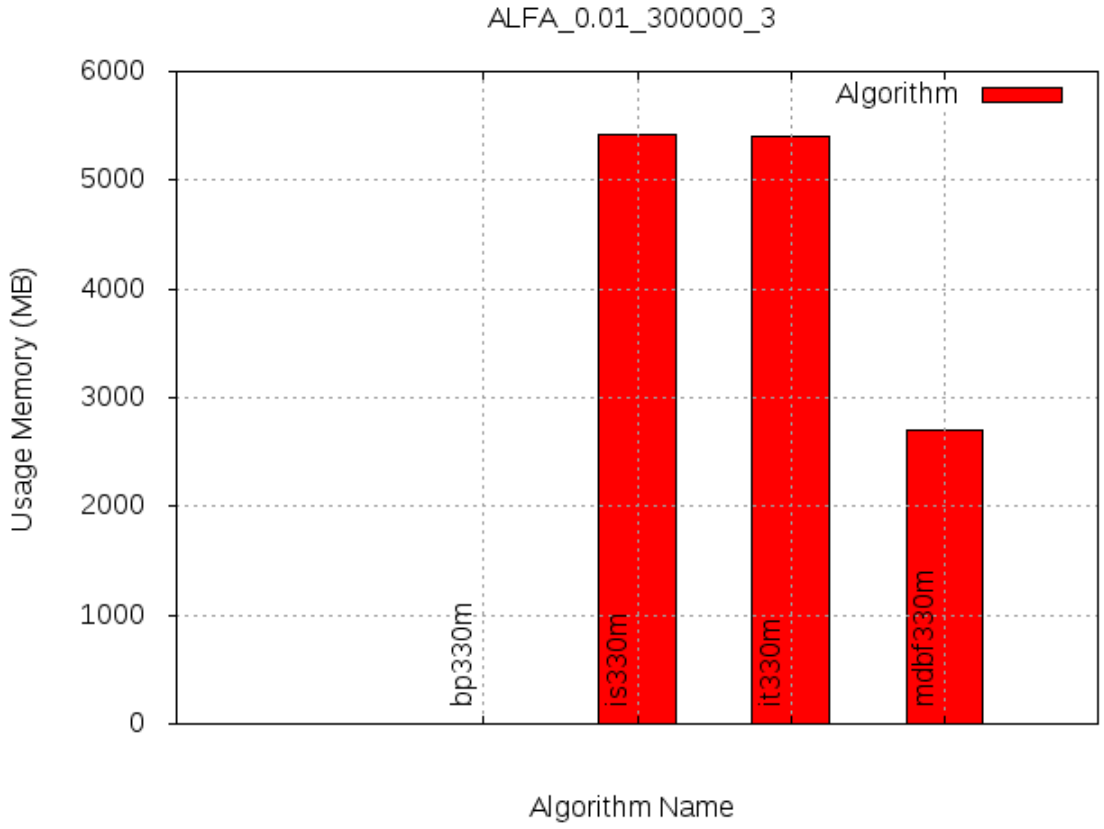

Figura 6.6: Memoria di picco - alfa test con alfa 0.01, numero di extent 300000 e 3 dimensioni

Algoritmo	Extent 100000	Extent 200000	Extent 300000
Binary Partition	2.42	5.71	11.12
Improved Sort	1.79	6.96	14.91
Interval Tree Matching	0.6	2.05	4.8
Brute Force	54.709999	228.860001	578.23999

Tabella 6.2: Alfa 0.01 - Tempo di Esecuzione (s)

Algoritmo	Extent 100000	Extent 200000	Extent 300000
Binary Partition	1070.08	3332.0959	0.0000
Improved Sort	617.2	2427.9041	5427.2002
Interval Tree Matching	611.1	2415.6160	5408.7681
Brute Force	305.9	1208.3199	2705.4080

Tabella 6.3: Alfa 0.01 - Memoria di picco (MB)

Come si può notare dalla Figura 6.6 la memoria utilizzata dal Binary Partition è a zero. Questo problema si è verificato perché tramite il framework valgrind analizzando la memoria ha occupato memoria oltre i 6GB e per non creare problemi con il sistema, il processo è stato bloccato.

In questa prima serie di grafici si può notare che il Brute Force ha un tempo di esecuzione elevato in confronto ai concorrenti mentre l'Interval Tree risulta essere il più veloce.

6.2 Alfa 1 - Extent 100000, 200000, 300000

Numero di extent preso in esame: 100000.

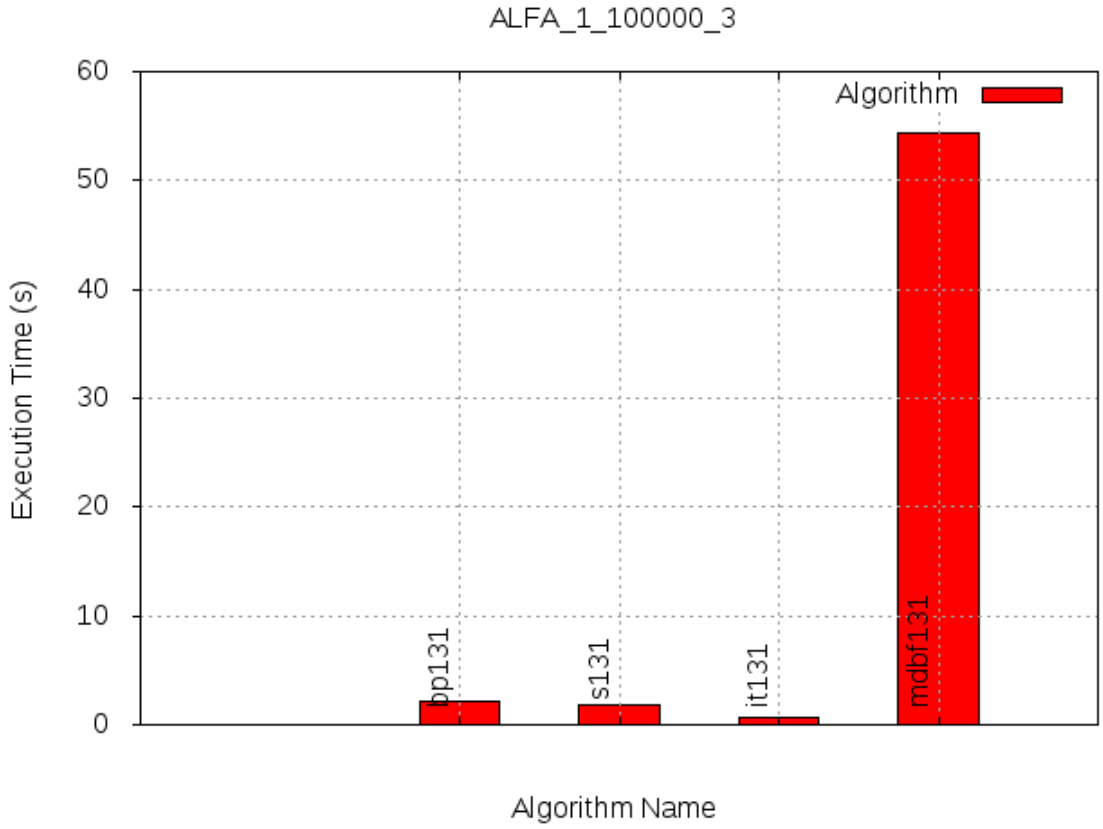

Figura 6.7: Tempo di esecuzione - alfa test con alfa 1, numero di extent 100000 e 3 dimensioni

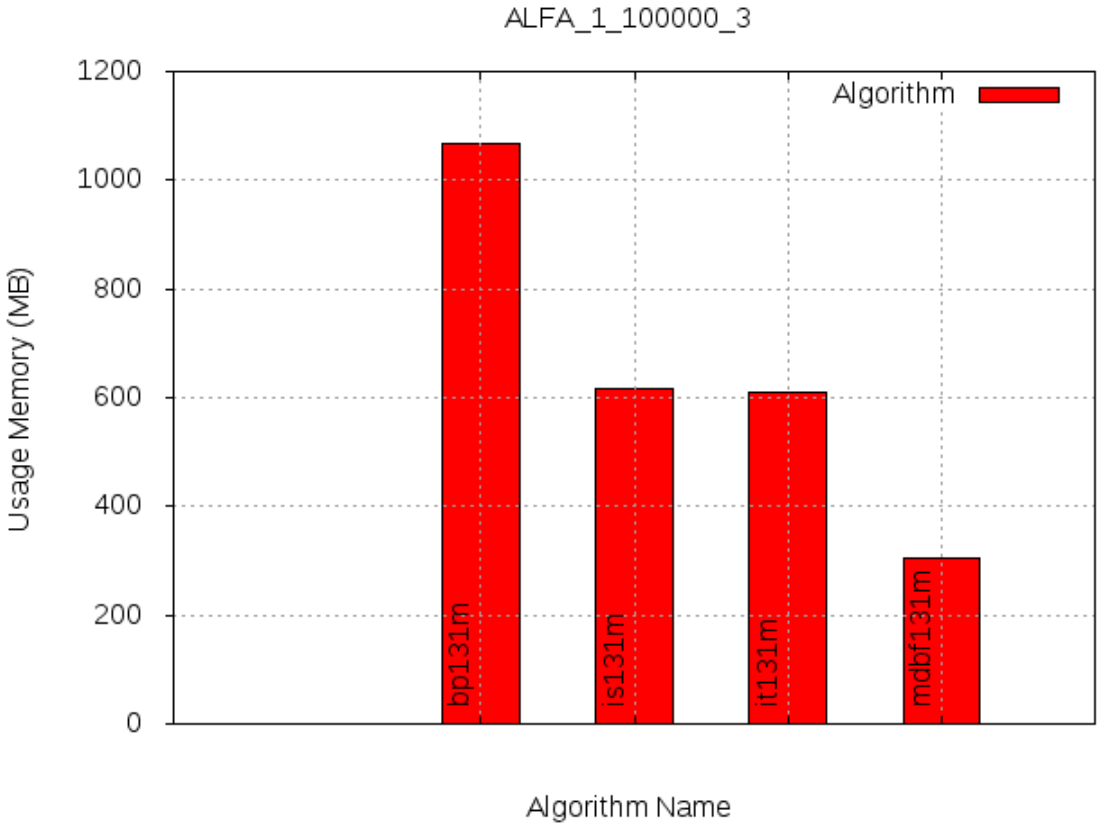

Figura 6.8: Memoria di picco - alfa test con alfa 1, numero di extent 100000 e 3 dimensioni

Numero di extent preso in esame: 200000.

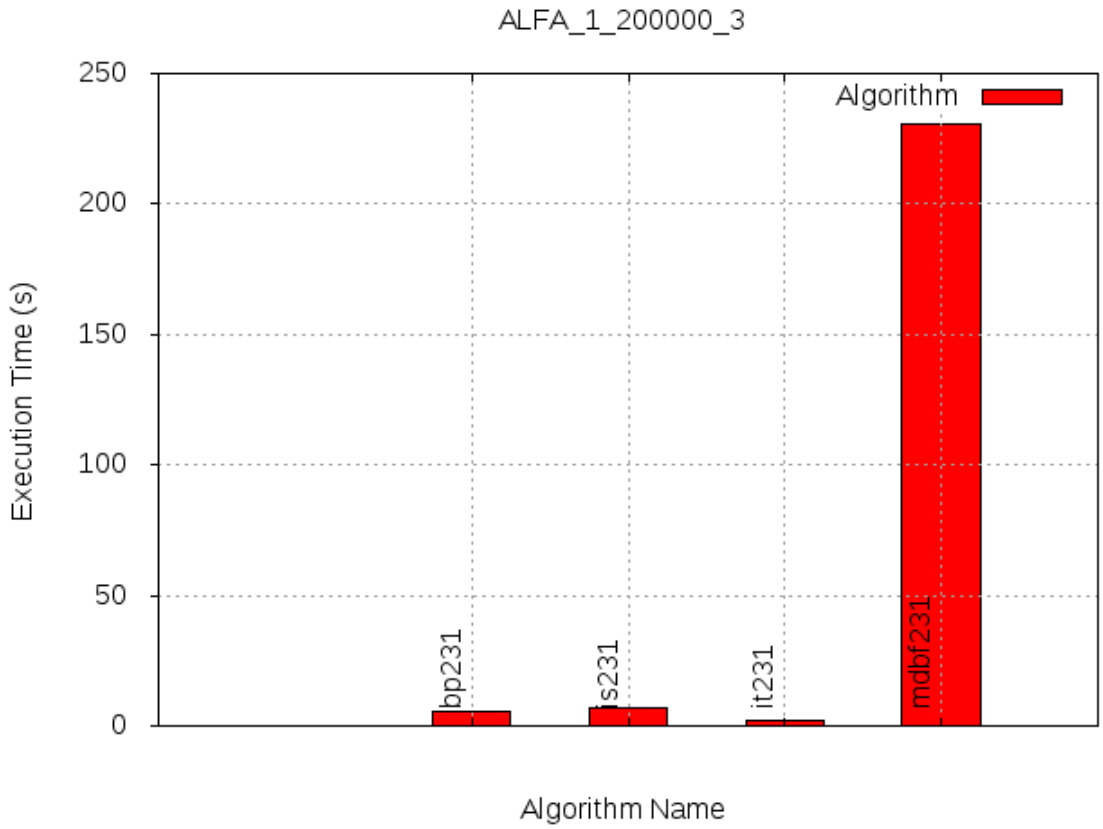

Figura 6.9: Tempo di esecuzione - alfa test con alfa 1, numero di extent 200000 e 3 dimensioni

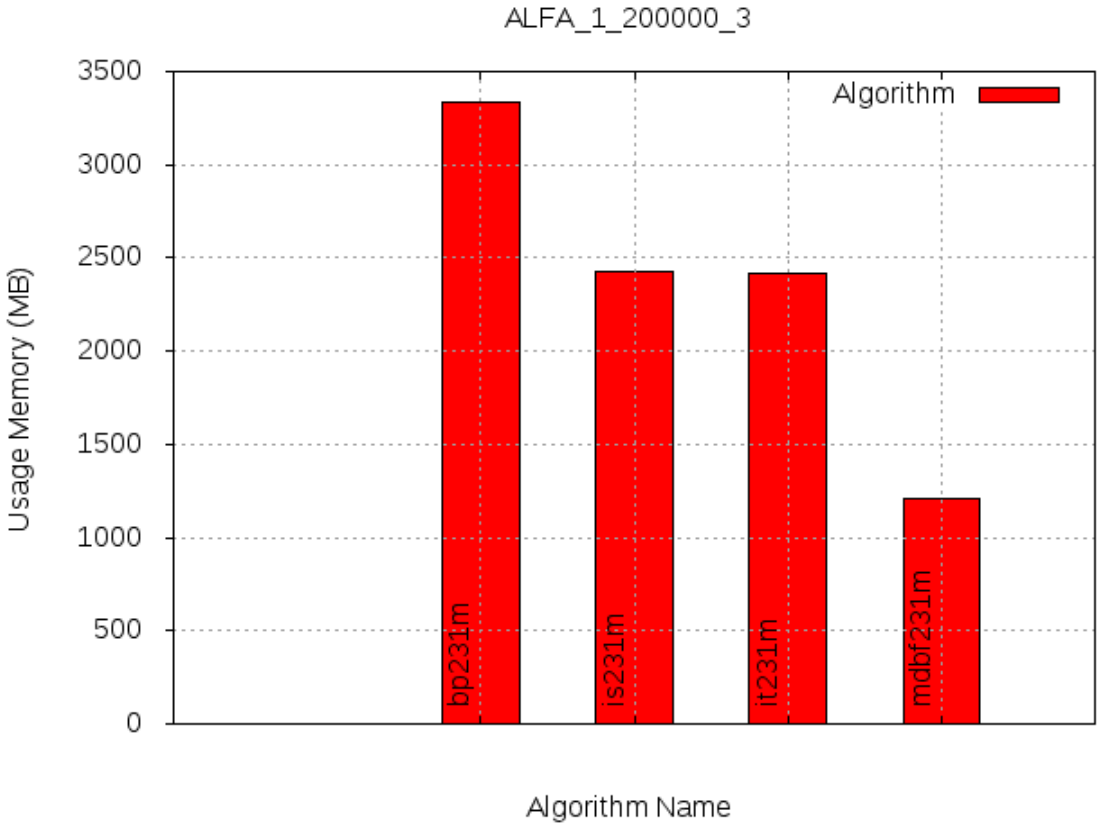

Figura 6.10: Memoria di picco - alfa test con alfa 1, numero di extent 200000 e 3 dimensioni

Numero di extent preso in esame: 300000.

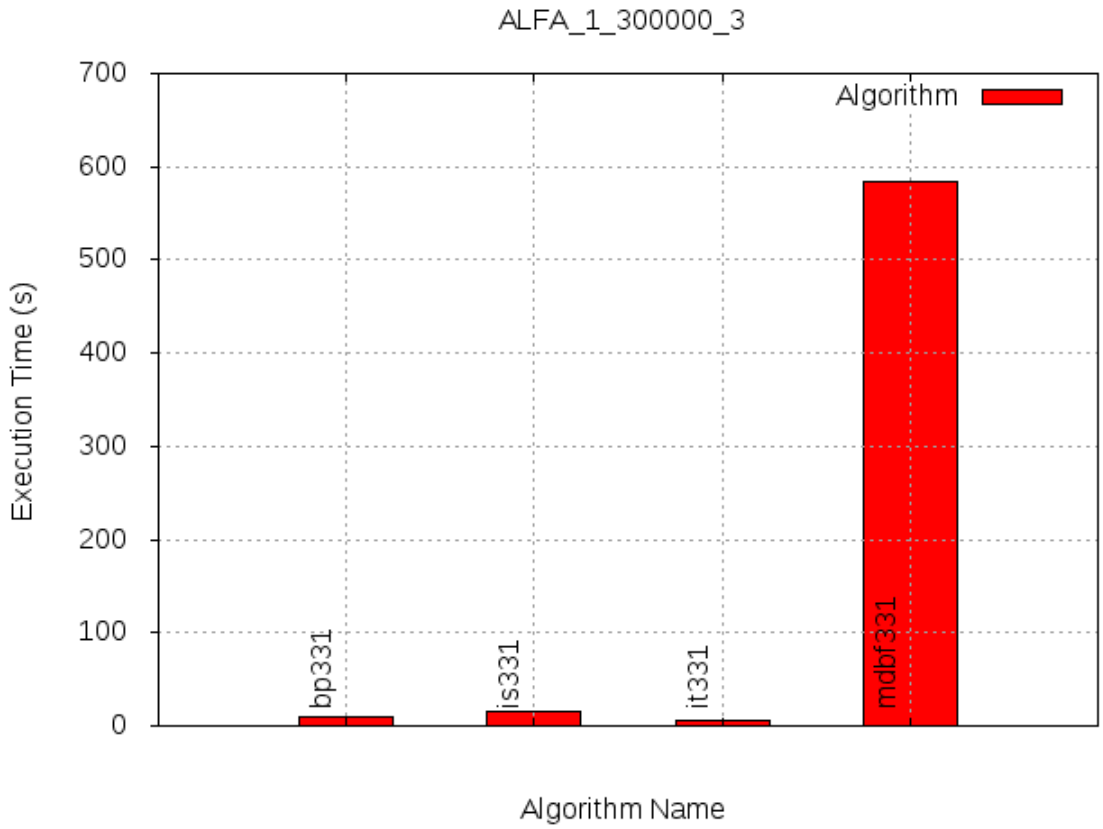

Figura 6.11: Tempo di esecuzione - alfa test con alfa 1, numero di extent 300000 e 3 dimensioni

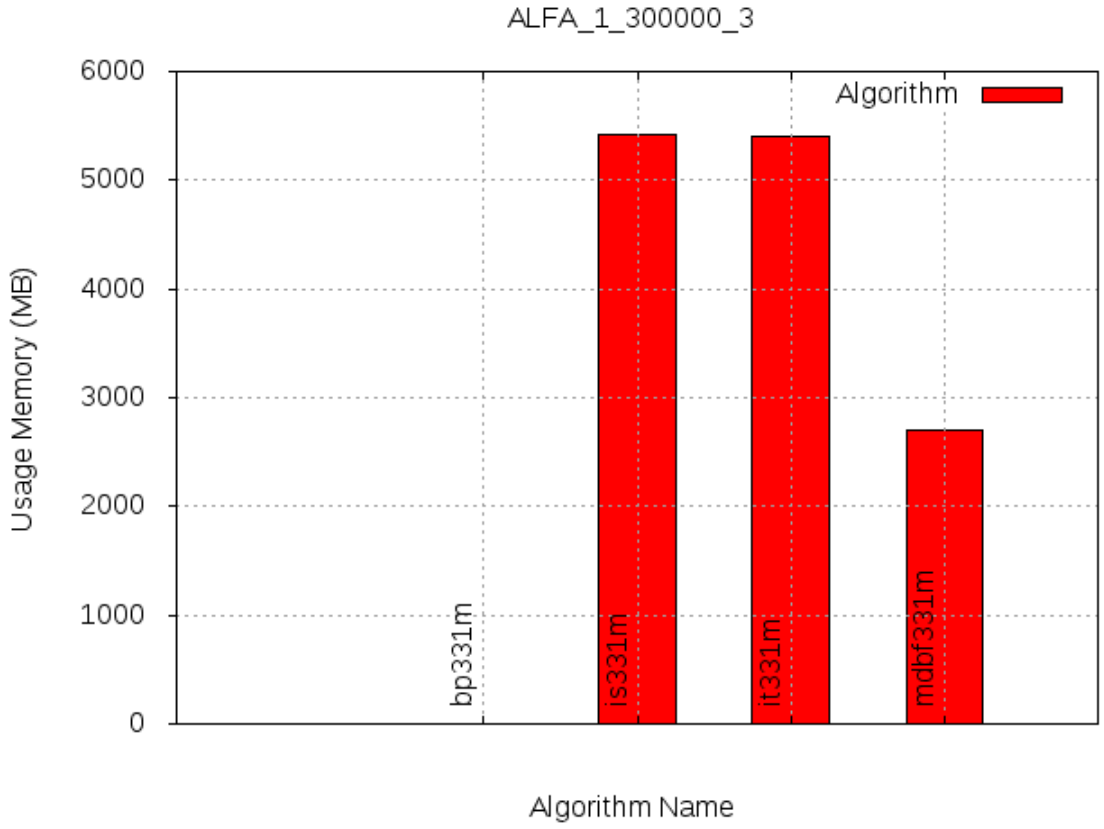

Figura 6.12: Memoria di picco - alfa test con alfa 1, numero di extent 300000 e 3 dimensioni

Algoritmo	Extent 100000	Extent 200000	Extent 300000
Binary Partition	2.13	5.21	9.64
Improved Sort	1.75	6.88	14.81
Interval Tree Matching	0.59	2.12	5.03
Brute Force	54.369999	230.74	583.159973

Tabella 6.4: Alfa 1 - Tempo di Esecuzione (s)

Algoritmo	Extent 100000	Extent 200000	Extent 300000
Binary Partition	1069.056	3332.0959	0.0000
Improved Sort	617.2	2427.9041	5427.2002
Interval Tree Matching	611.1	2415.6160	5408.7681
Brute Force	305.9	1208.3199	2705.4080

Tabella 6.5: Alfa 1 - Memoria di picco (MB)

Anche in questo caso come si può notare dalla Figura 6.12 la rilevazione dell'utilizzo della memoria di picco non viene rappresentata in quanto ne occupa una quantità superiore a quella permessa. Valgrind utilizza un elevata quantità di memoria in più per essere eseguito e quindi il processo viene bloccato. Il tempo di esecuzione viene rilevato comunque perchè sono state svolte due esecuzioni una con valgrind per ottenere la memoria e l'altra senza valgrind per ottenere il tempo di esecuzione. Il tempo di esecuzione quindi viene rilevato comunque.

Come nella situazione precedente il più performante rimane l'Interval Tree.

6.3 Alfa 100 - Extent 100000, 200000, 300000

Numero di extent preso in esame: 100000.

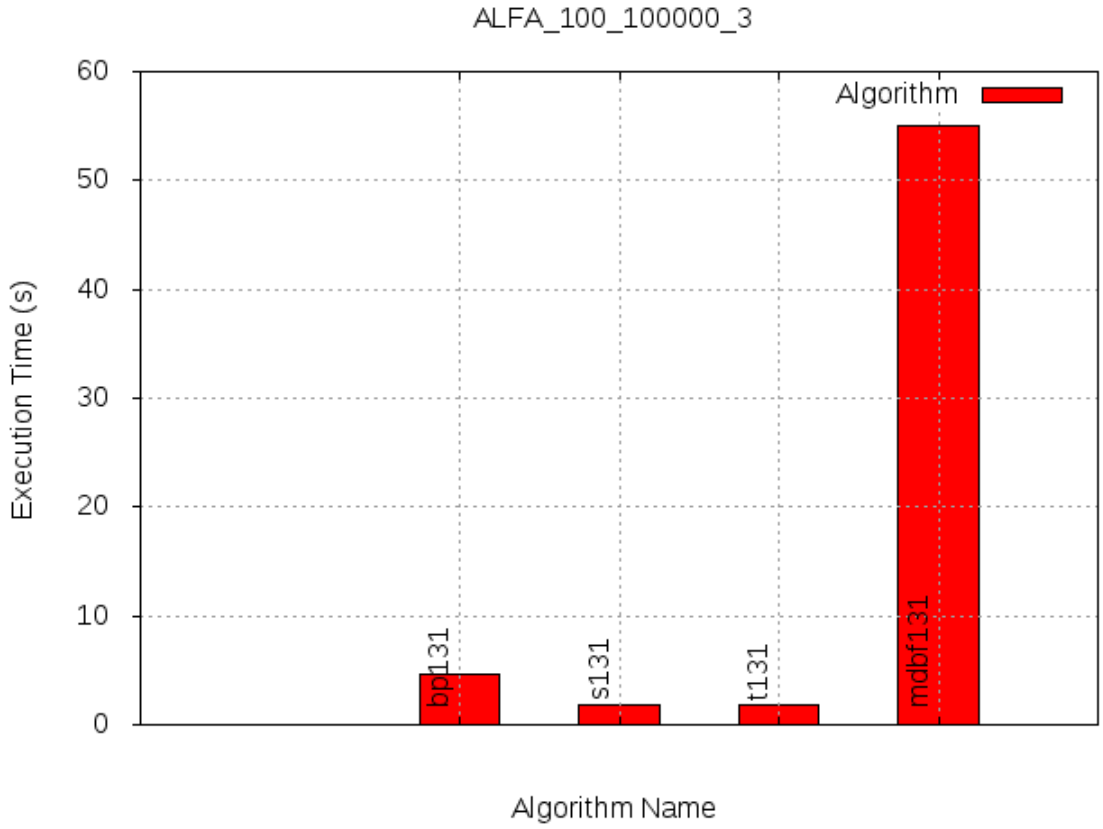

Figura 6.13: Tempo di esecuzione - alfa test con alfa 100, numero di extent 100000 e 3 dimensioni

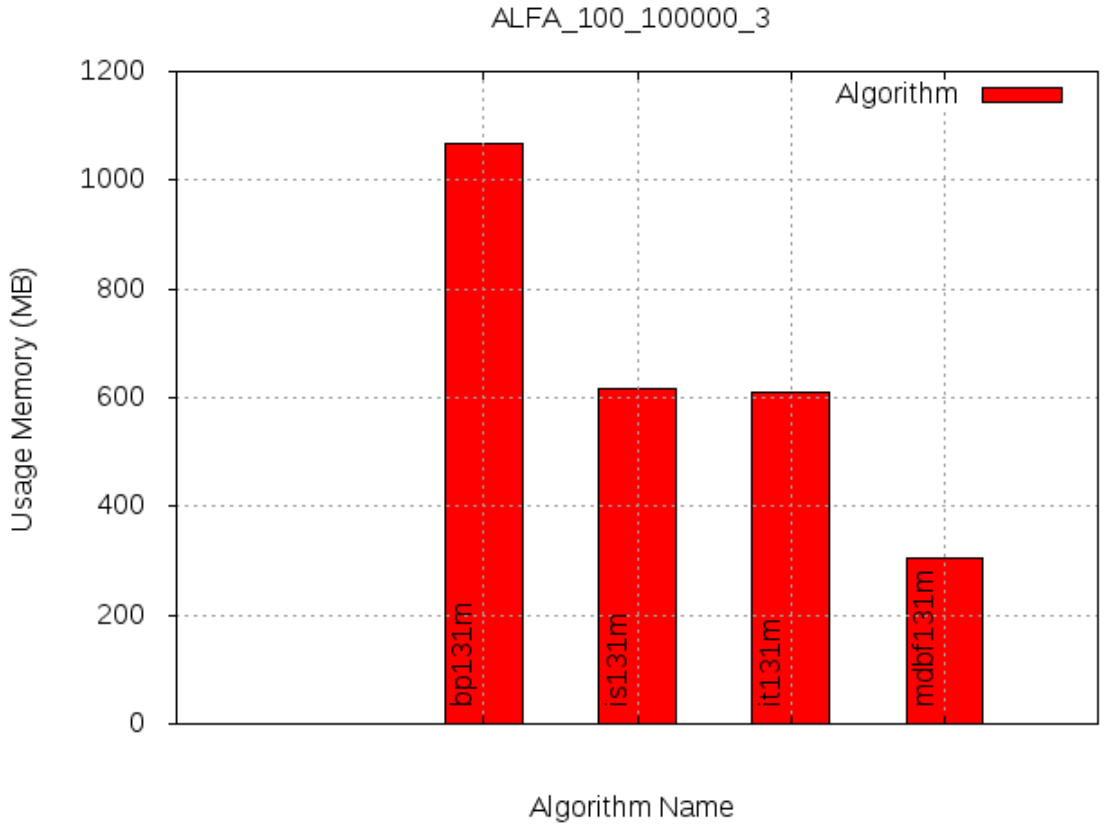

Figura 6.14: Memoria di picco - alfa test con alfa 100, numero di extent 100000 e 3 dimensioni

Numero di extent preso in esame: 200000.

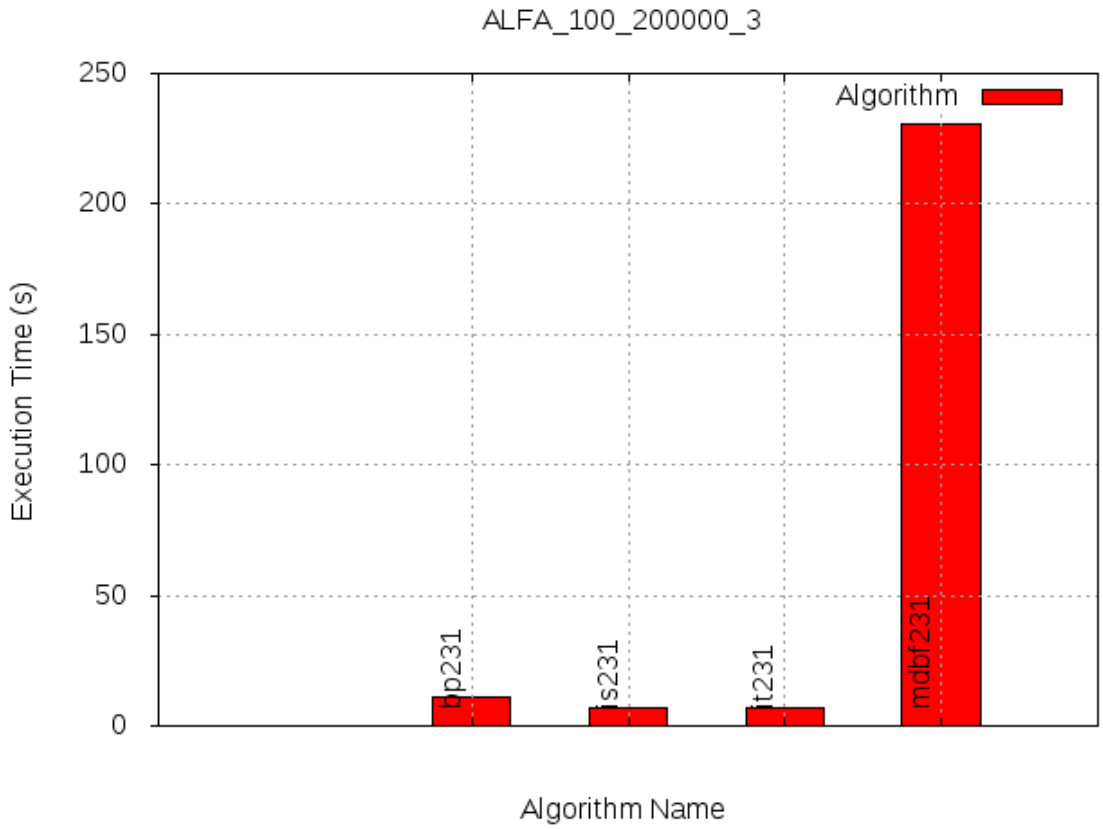

Figura 6.15: Tempo di esecuzione - alfa test con alfa 100, numero di extent 200000 e 3 dimensioni

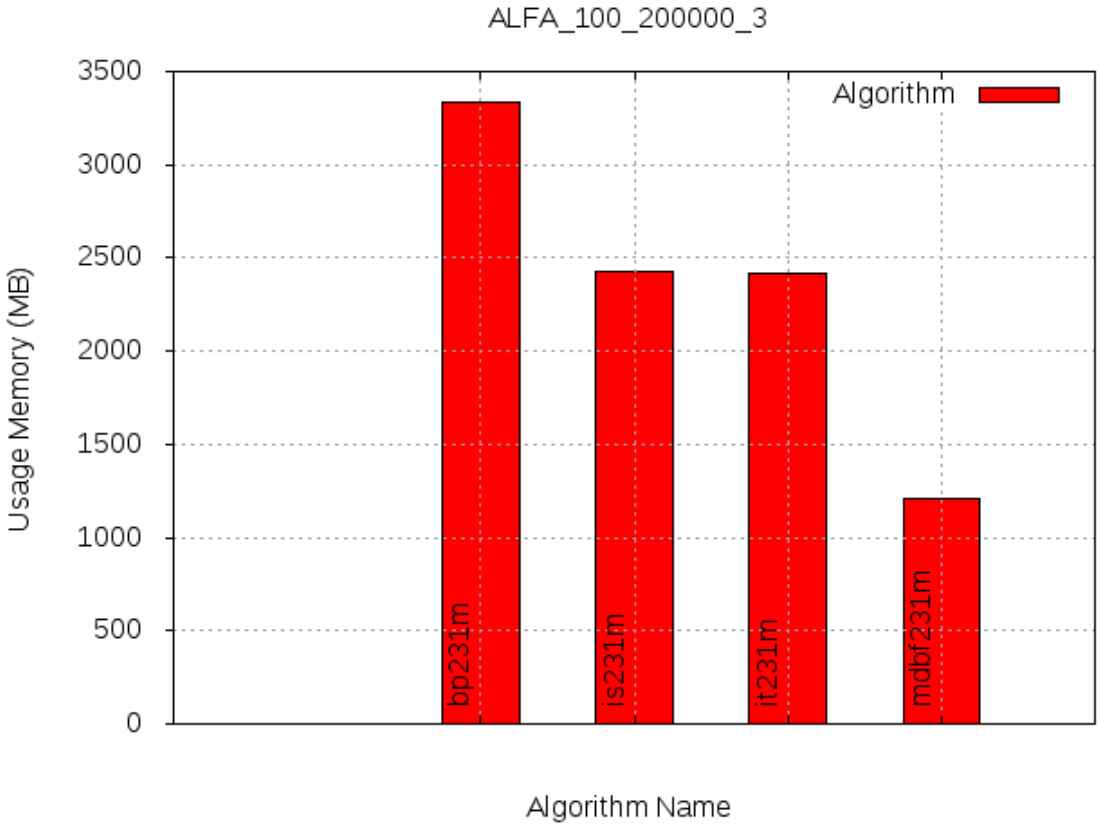

Figura 6.16: Memoria di picco - alfa test con alfa 100, numero di extent 200000 e 3 dimensioni

Numero di extent preso in esame: 300000.

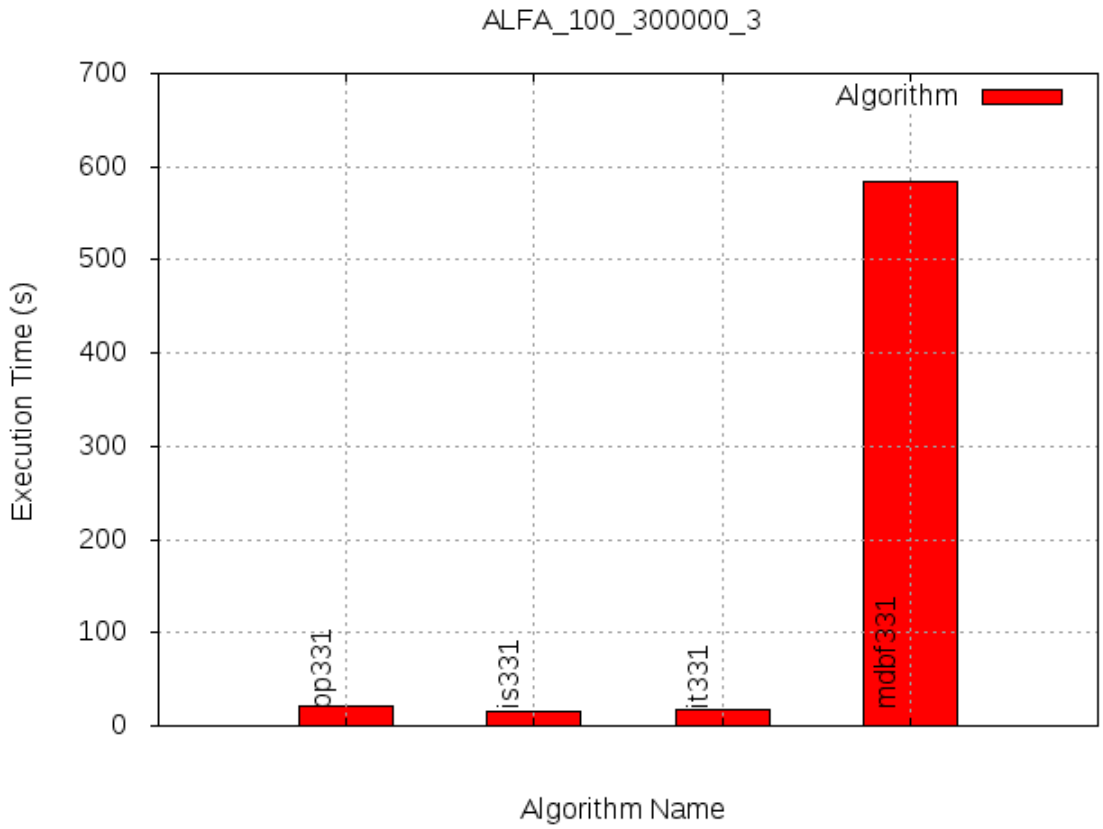

Figura 6.17: Tempo di esecuzione - alfa test con alfa 100, numero di extent 300000 e 3 dimensioni

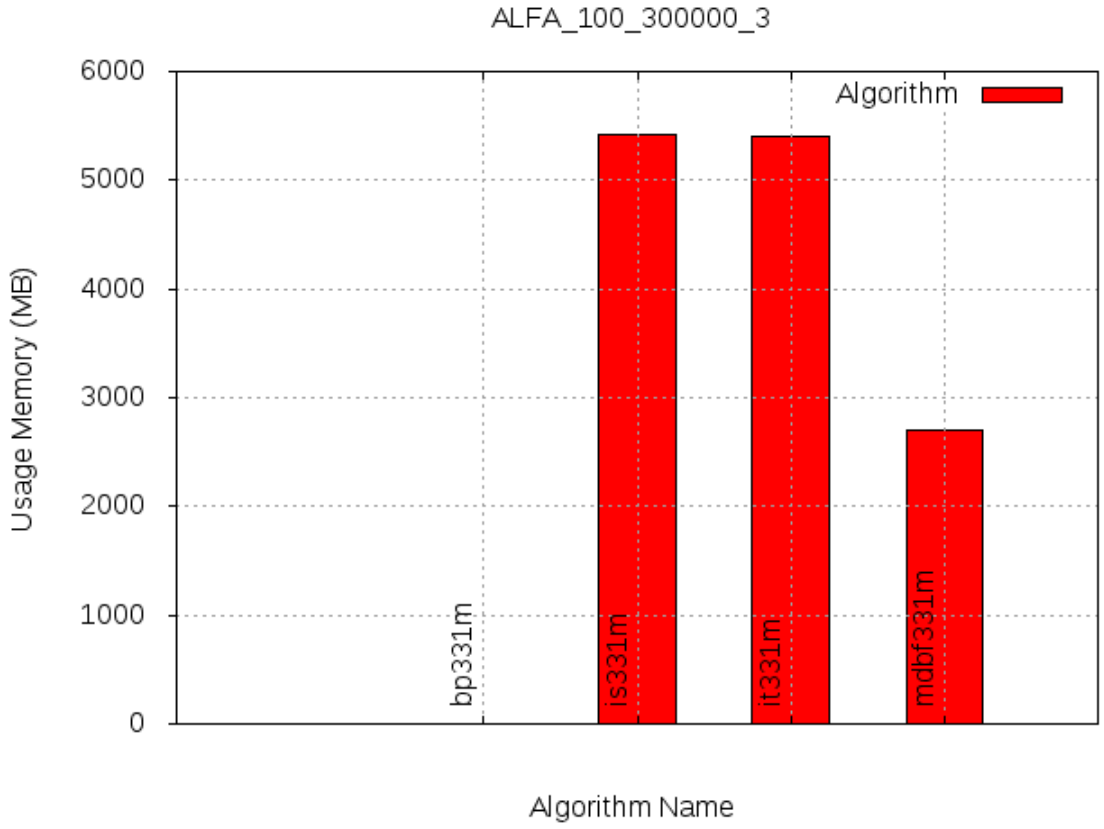

Figura 6.18: Memoria di picco - alfa test con alfa 100, numero di extent 300000 e 3 dimensioni

Algoritmo	Extent 100000	Extent 200000	Extent 300000
Binary Partition	4.58	11.09	20.950001
Improved Sort	1.76	6.77	15.01
Interval Tree Matching	1.9	6.69	17.879999
Brute Force	55.009998	230.399994	583.299988

Tabella 6.6: Alfa 100 - Tempo di Esecuzione (s)

Algoritmo	Extent 100000	Extent 200000	Extent 300000
Binary Partition	1069.056	3332.0959	0.0000
Improved Sort	617.2	2427.9041	5427.2002
Interval Tree Matching	611.1	2415.6160	5408.7681
Brute Force	305.9	1208.3199	2705.4080

Tabella 6.7: Alfa 100 - Memoria di picco (MB)

Anche in questo caso il problema sulla memoria persiste. Le performance dei vari algoritmi sono differenti in questo caso, con un grado maggiore di sovrapposizioni tra update e subscription l'Improved Sort è più performante del Binary Partition e anche migliore dell'Interval Tree.

Capitolo 7

Conclusioni

In questa tesi è stato implementato un framework per facilitare la creazione e il testing di vari algoritmi. Si sono analizzati vari algoritmi disponibili in letteratura e varie implementazioni di essi. Da questa analisi si è potuto dedurre che in tutte le implementazioni erano presenti parti che dovevano essere necessariamente uguali per effettuare delle comparazioni.

Si è valutata l'ipotesi di creare un framework per poter testare gli algoritmi e facilitarne la loro implementazione. Dall'analisi dei benefici che un framework potesse portare sono stati rilevati i seguenti punti di forza: creazione di test di vario genere, creazione di librerie per facilitare la creazione di nuovi algoritmi, creazione di approcci di valutazione diversi tra di loro come l'utilizzo di memoria di picco utilizzata, il tempo di esecuzione e la distanza dalla soluzione ottima, creazione di processi automatici e infine la creazione di processi per rappresentare in modo grafico i risultati delle esecuzioni dei vari algoritmi per poter effettuare una comparazione.

Il framework ha apportato elevati benefici che però potevano essere vanificati se fosse stato eseguito su macchine con componentistica hardware diversa tra loro. Per questo motivo è stata presa in considerazione l'idea di eseguire il framework su macchine virtuali sul Cloud. Sono state analizzate due compagnie che offrono servizi di Cloud Computing: Amazon e Google. Dopo un'analisi prestazionale sulle macchine virtuali si è scelto di utilizzare il servizio di Google: Google App Engine.

Anche se sono stati presi in esame solamente i due servizi Google App Engine e Amazon Web Services, questo non significa che siano le uniche soluzioni possibili. È stato utilizzato Google App Engine e sono stati postati i risultati in questa tesi e nel caso vogliano essere confrontati è sufficiente creare una macchina virtuale con le stesse caratteristiche descritte nel capitolo precedente. L'intento del Cloud è quello di poter utilizzare macchine con le stesse specifiche per poter comparare le simulazioni.

Sono stati implementati quattro algoritmi all'interno del framework: Brute Force, Binary Partition, Improved Sort e Interval Tree Matching. Alcuni di questi sono stati importati da altri progetti e tesi e adattati al framework facilmente tramite l'utilizzo delle librerie in C appositamente create. Per l'algoritmo Interval Tree era presente solamente la versione monodimensionale ed è stata adattata la versione multidimensionale. L'Improved Sort è uno degli algoritmi implementati in modo più efficiente grazie all'utilizzo di operazioni di bitwise. L'implementazione del Binary Partition non è la migliore, si può migliorare in quanto questa utilizza una quantità di memoria non indifferente per le varie liste allocate dinamicamente. Si potrebbe migliorare utilizzando degli indici di posizione al posto di allocare ad ogni partizionamento gli extent per la partizione di sinistra e di destra. L'algoritmo Brute Force è stato il più semplice da implementare e la sua semplicità di implementazione è stata ripagata dalla scarsa scalabilità e lentezza nel tempo di esecuzione.

Non è facile dire con certezza che un algoritmo è meglio di un altro perché bisogna valutare la situazione di utilizzo. Si può dire che il Brute Force si può usare per situazioni con un numero di extent basso e una quantità ridotta di memoria. Il Binary Partition, per questa implementazione, occupa una quantità di memoria notevole e nella maggior parte casi è più efficiente del Brute Force e in pochi casi ha un tempo di esecuzione inferiore rispetto all'Improved Sort e Interval Tree Matching. L'Improved Sort e l'Interval Tree Matching sono i più efficienti in termini di tempo di esecuzione e occupano una simile quantità di memoria. L'Improved Sort risulta leggermente più veloce nei casi in cui il grado di sovrapposizione è più alto mentre in questi casi l'Interval Tree peggiora le sue performance. Tutti gli algoritmi sono stati implementati nella versione sequenziale. L'Interval Tree Matching può avere un tempo di esecuzione inferiore rispetto all'Improved Sort se parallelizzato.

Il framework può essere migliorato rendendolo più Cloud e fornendo servizi più semplici e comodi da utilizzare. Potrebbero essere migliorate le librerie al suo interno con un numero maggiore di funzioni che si possano adattare a più tipi di algoritmi. È stata implementata una parte inerente alla distanza dalla soluzione ottima ma si è preso in esame più dettagliatamente il consumo di memoria e il tempo di esecuzione, quindi si può automatizzare il processo di comparazione delle bitmatrix più efficientemente. Sono già stati importati altri tipi di test da altri progetti e si potrebbe aumentare il numero di questi test con approcci differenti.

Tutto ciò che è stato sviluppato in questa tesi è disponibile sotto licenza GPL sul sito **Github** al seguente indirizzo *https://githu.com/nicholasricci/DDM_Framework*.

Bibliografia

[1] Anu Maria. Introduction to modeling and simulation. In *Proceedings of the 29th Conference on Winter Simulation*, WSC '97, pages 7–13, Washington, DC, USA, 1997. IEEE Computer Society.

[2] Judith S. Dahmann, Richard M. Fujimoto, and Richard M. Weatherly. The department of defense high level architecture. In *Simulation Conference Proceedings, 1998. Winter*, pages 797–804, Washington, DC, USA, 1998. IEEE Computer Society.

[3] John L. Gustafson. Reevaluating Amdahl's law. *Commun. ACM*, 31(5):532–533, May 1988.

[4] R.M. Fujimoto. Distributed simulation systems. In *Simulation Conference, 2003. Proceedings of the 2003 Winter*, volume 1, pages 124–134 Vol.1, Dec 2003.

[5] Azzedine Boukerche and Amber Roy. Dynamic grid-based approach to data distribution management. *Journal of Parallel and Distributed Computing*, 62(3):366 – 392, 2002.

[6] Katherine L. Morse and Jeffrey S. Steinman. Data distribution management in the hla - multidimensional regions and physically correct filtering. In *In Proceedings of the 1997 Spring Simulation Interoperability Workshop*, pages 343–352. Spring, 1997.

[7] Katherine L. Morse, Lubomir Bic, Michael Dillencourt, and Kevin Tsai. Multicast grouping for dynamic data distribution management. In *In Proceedings of the 31st Society for Computer Simulation Conference*, 1999.

[8] Moreno Marzolla, Gabriele D'Angelo, and Marco Mandrioli. A parallel data distribution management algorithm. In *Proc. IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications (DS-RT 2013)*, Delft, the Netherlands, October 30–November 1 2013.

[9] Francesco Bedini. Design and implementation of a testbed for data distribution management. Ottobre 2013.

[10] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM.

[11] Thomas Williams and Colin Kelley. Gnuplot 4.6: an interactive plotting program. `http://gnuplot.sourceforge.net/`, September 2014.

[12] Junghyun Ahn, Changho Sung, and Tag Gon Kim. A binary partition-based matching algorithm for data distribution management. In *Simulation Conference (WSC), Proceedings of the 2011 Winter*, pages 2723–2734, Dec 2011.

[13] Moreno Marzolla, Gabriele D'Angelo, and Marco Mandrioli. A parallel data distribution management algorithm. In *Proceedings of the 2013 IEEE/ACM 17th International Symposium on Distributed Simulation and Real Time Applications*, DS-RT '13, pages 145–152, Washington, DC, USA, 2013. IEEE Computer Society.

[14] Mandrioli Marco. Progettazione, implementazione e valutazione di algoritmi per il data distribution management. Dicembre 2012.

[15] Google. Google App Engine, April 2008.

[16] Amazon Inc. Elastic Compute Cloud, November 2008.