Alma Mater Studiorum · Università di Bologna

FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI
Corso di Laurea Triennale in Scienze dell'Informazione

PROGETTAZIONE, IMPLEMENTAZIONE E VALUTAZIONE DI ALGORITMI PER IL DATA DISTRIBUTION MANAGEMENT

Tesi di Laurea in Reti di Calcolatori

Relatore:
GABRIELE D'ANGELO

Presentata da:
MARCO MANDRIOLI

Sessione II
Anno Accademico 2011-2012

Per andare avanti ci vuole pazienza,
nell'inamovibile convinzione
che un software senza bug
è l'equivalente informatico di Dio.

Introduzione

Nel corso di questa tesi verranno analizzati ed elaborati algoritmi per risolvere specifici problemi all'interno di una simulazione. Trattandosi di argomenti alquanto tecnici, è bene introdurre alcuni concetti fondamentali prima di addentrarsi nel cuore della tesi, partendo dal concetto di simulazione.

Nel linguaggio comune, il termine **simulazione** deriva dal verbo *simulare*, ovvero "fare apparire come reale, come vero ciò che non lo è".[1] In campo tecnico-scientifico viene perciò chiamata *simulazione* la "riproduzione in un ambiente controllato di un sistema o di parti di esso".[2] Ciò comprende un'enorme quantità di giochi di qualsiasi tipo: simulatori di volo, di guida, di guerra e videogiochi sportivi sono sicuramente i più facili da riconoscere, ma rientrano in questa categoria anche i giochi di ruolo, in cui i giocatori si muovono in un mondo più o meno immaginario, nonché giochi da tavolo come Monopoli o Risiko.

Le applicazioni delle simulazioni però non si limitano al settore ludico: praticamente ogni cosa può essere simulata. Alcuni esempi sono i crash-test effettuati sulle automobili, le previsioni climatiche e l'analisi della viabilità urbana. Inoltre anche alcuni dei già citati tipi di videogiochi sono spesso utilizzati per scopi molto diversi dall'intrattenimento. Simulatori di volo e di guida estremamente dettagliati vengono utilizzati per l'addestramento e allenamento di piloti, e l'esercito americano ha sviluppato *America's Army*, un simulatore di guerra che viene utilizzato sia come campagna pubblicitaria rivolta ai ragazzi americani,[3] sia come addestramento per i soldati.[4, 5] Questo non deve stupire in quanto, dati i costi e i rischi presenti in guerra

(e non solo), i militari hanno sempre fatto un uso massiccio di molti tipi di simulazione.

Restringendo il campo alle simulazioni su calcolatore è importante notare che una simulazione, per essere affidabile, richiede un grande livello di dettaglio del modello che deve elaborare, e un modello più dettagliato comporta la necessità di una maggiore potenza di calcolo. La complessità dei modelli utilizzati per le previsioni climatiche necessita della potenza di calcolo di alcuni tra i supercomputer più potenti al mondo,[6] nonostante i risultati siano comunque ben lontani dalla perfezione. Spesso si rivela quindi fondamentale distribuire il sistema su più unità di calcolo, suddividendo una simulazione complessa in varie simulazioni più semplici (la marina, l'aeronautica e le unità di terra possono essere unite a formare le forze armate di una nazione, e più nazioni possono essere unite per simulare una guerra). È però possibile anche il contrario: simulazioni sviluppate con scopi diversi possono dover essere unite. Ciò comporta un grosso problema di interoperabilità, dato che le piattaforme da unire—per non parlare delle simulazioni stesse—sono potenzialmente eterogenee.

Per risolvere i problemi legati all'interoperabilità tra differenti simulazioni, il Dipartimento della Difesa americano ha contribuito alla creazione di uno standard per l'architettura di sistemi di simulazione distribuiti. Questo standard, chiamato **High-Level Architecture** (HLA), permette a diverse simulazioni di interagire in moto totalmente scollegato dalla piattaforma sulla quale risiedono.
Lo standard HLA è formato da tre componenti:

- **interface specification**, definisce come le simulazioni devono interagire con l'infrastruttura;
- **Object Model Template** (OMT), specifica quali informazioni devono essere comunicate;
- **rules**, insieme di regole che le simulazioni devono rispettare.

L'*interface specification* è basata su un modello ad oggetti ed è divisa

in sette gruppi di servizi. La sua implementazione viene chiamata *Run-Time Infrastructure* (RTI), ed è il middleware vero e proprio. Per lasciare libertà implementativa agli sviluppatori, però, solo l'interfaccia fa parte delle specifiche.

Il **Data Distribution Management** (DDM) è uno dei gruppi di servizi che devono essere implementati in RTI e si occupa della distribuzione dei dati tra i vari federati. Il suo obiettivo è quello di limitare la quantità di messaggi ricevuti dai federati in grosse federazioni distribuite, sia per ridurre il data set che dovrà essere processato dal federato ricevente, sia per ridurre il traffico di rete.[7] Per fare questo è fondamentale l'operazione di matching che, data una serie di regioni in uno spazio, si occupa di scoprire quali sono sovrapposte. Concettualmente, il problema è molto semplice: nella Figura 1, la regione B è sovrapposta alle regioni A e C (le quali non sono tra di loro sovrapposte), mentre la regione D non è sovrapposta a nessun'altra.

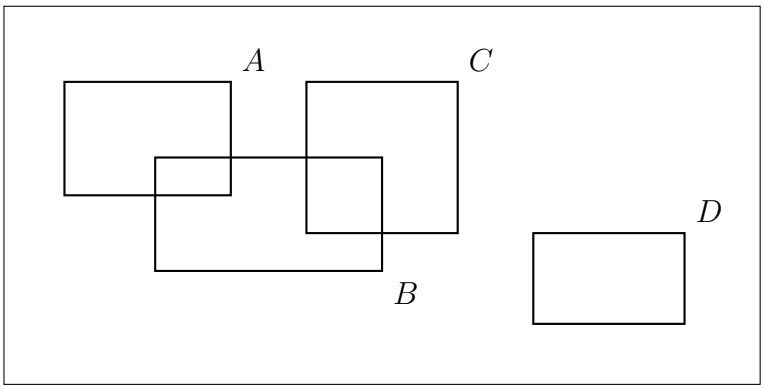

Figura 1: Esempio di sovrapposizioni tra regioni.

Nella parte iniziale di questa tesi ci si occuperà di analizzare alcuni degli algoritmi più conosciuti per il matching. A seguire verrà implementato, modificato, ottimizzato e parallelizzato il più veloce di questi.

Indice

Elenco delle figure

Elenco delle tabelle

Elenco degli algoritmi

Capitolo 1

Il Data Distribution Management

Il *Data Distribution Management* è il componente di HLA che si occupa di coordinare e gestire lo scambio di dati tra i federati. Comprende una serie di servizi atti a ridurre la quantità di dati non necessari da trasferire, diminuendo così sia il traffico sulla rete, sia il carico di lavoro dei federati riceventi (che devono scartare i dati non necessari).[7, 8, 9, 10, 11]

Il DDM suddivide i federati in due gruppi in base al loro ruolo nello scambio attuale: viene definito *publisher* il federato che pubblica i dati, e *subscriber* quello che li riceve.[9, 10, 11] Il trasferimento dei dati tra publisher e subscriber avviene in cinque fasi:

1. i publisher dichiarano quali dati intendono produrre e i subscriber quali ricevere;

2. l'algoritmo di matching filtra i federati, ottenendo l'elenco delle connessioni necessarie;

3. RTI apre le connessioni necessarie (calcolate al punto precedente) tra i federati;

4. i publisher trasmettono i dati ai subscriber a cui sono connessi;

5. i subscriber filtrano i dati ricevuti per eliminare le eventuali informazioni inutili.

Dato che un filtro viene applicato comunque dai singoli subscriber dopo il trasferimento dei dati (punto 5), il filtraggio effettuato al punto 2 potrebbe anche essere completamente omesso. Ciò risulterebbe però in uno scambio da ogni publisher ad ogni subscriber, seguito poi comunque da un filtraggio effettuato da ogni subscriber su tutti i publisher. Questo approccio è sicuramente funzionante, ma comporta uno scambio di $P \times S$ dati, dove P è il numero di publisher e S è il numero di subscriber. Supponendo un bilanciamento tra il numero di publisher e subscriber, questo numero aumenta quadraticamente (ed è in grado di mandare in crisi facilmente qualsiasi sistema, al crescere dei federati), mentre in generale il numero di scambi necessari risulta essere molto più limitato.

1.1 Il matching

L'operazione di matching serve proprio a ridurre il numero di connessioni (e quindi trasferimenti), spostando il filtraggio prima del trasferimento dati. Per questo publisher e subscriber devono 'iscriversi', dichiarando quali dati intendono inviare o ricevere, prima di poter effettuare lo scambio vero e proprio. L'algoritmo lavora su uno spazio multi-dimensionale di coordinate che viene chiamato *routing space*. Un *extent* è un sottospazio rettangolare del routing space, ed una *region* è un insieme di extent in esso. Vengono definite *update region* le region inserite dai publisher, e *subscription region* quelle inserite dai subscriber.[9, 10, 11]

Esempio 1.1.1. Un routing space può essere uno spazio bidimensionale rappresentante la mappa di un campo di battaglia, le subscription region un insieme di rettangoli (extent) rappresentanti i sensori delle varie unità e le update region un insieme di piccoli rettangoli rappresentanti la posizione delle unità.

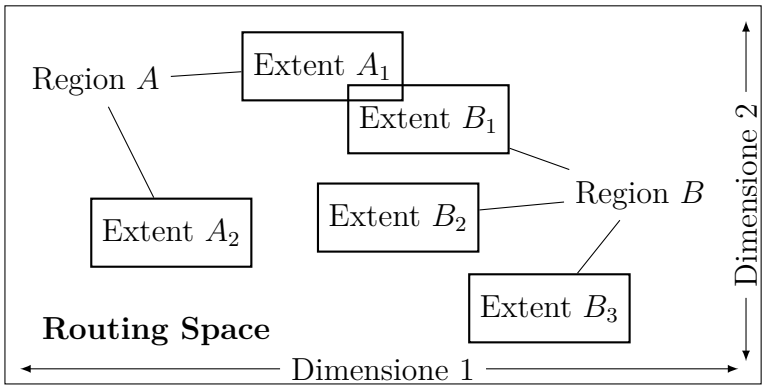

Figura 1.1: Un routing space contenente due region.[11]

Nella Figura 1.1 è rappresentato un routing space bidimensionale al cui interno vi sono due region (A e B). A è formata da due extent (A_1 e A_2), mentre B è formata da tre (B_1, B_2 e B_3). Le due region vengono considerate sovrapposte anche se solo uno degli extent di A è effettivamente sovrapposto ad uno degli extent di B.[11]

Le dimensioni non sempre rappresentano coordinate spaziali, anzi, spesso contengono attributi molto più astratti: estendendo l'Esempio 1.1.1, è evidente che un radar non può rilevare tutti i tipi di unità nel suo raggio di azione (si pensi alla fanteria), e quindi una dimensione servirà ad indicare se l'unità è visibile ai radar. Per portare anche un esempio completamente diverso, nella simulazione di un bar una dimensione può rappresentare il tipo di servizio richiesto, in modo che un ordine per un panino venga passato al cuoco ma non al barista.

Nota. *I dati che i federati intendono trasferire* non *sono quelli contenuti nel routing space. Detto in maniera semplice, i routing space vengono utilizzati come filtri dal DDM. Non bisogna però incorrere nell'errore di credere che la loro utilità sia limitata al solo filtraggio: simulando un radar, la posizione delle unità viene utilizzata per decidere se questo è in grado di rilevarle, e in*

caso affermativo tra le informazioni necessarie al radar ci saranno ovviamente anche le coordinate.

1.1.1 Il matching nel DDM

Vi sono alcune differenze tra il problema del matching presentato nella sua versione più semplice durante l'introduzione e la sua incarnazione all'interno del DDM. Come prima cosa, siccome un federato publisher invia dati (senza riceverne) mentre un federato subscriber riceve solo, risulta abbastanza evidente che è completamente inutile confrontare tra loro gli extent appartenenti a federati dello stesso tipo. A questo proposito, gli extent vengono suddivisi in due tipi allo stesso modo delle region, così da poter essere distinti chiaramente. Sono quindi chiamati:

- **update extent**, gli extent appartenenti alle update region;
- **subscription extent**, gli extent appartenenti alle subscription region.

Gli algoritmi di matching per il DDM dovranno dunque confrontare solo gli update extent con i subscription extent.[7, 10, 11]

Un altro fattore importante è che, considerato che i dati verranno filtrati nuovamente dai subscriber (punto 5 della lista a pagina 1), l'algoritmo di matching usato nel DDM non deve necessariamente essere ottimo. Il matching è un'operazione molto costosa, e 'alleggerire' un po' l'algoritmo—lasciando passare una piccola percentuale di dati non necessari—può in certi casi portare ad un grosso miglioramento nella velocità di esecuzione, a fronte di un incremento nei trasferimenti.[12, 13] Perché il sistema funzioni, però, è altresì necessario che la soluzione ottima sia un sottoinsieme della soluzione generata, visto che in caso contrario i subscriber non riceverebbero tutti i dati a loro necessari.

Capitolo 2

Principali algoritmi per il matching nel DDM

Come accennato, il matching è un'operazione decisamente costosa. Per questo, negli anni sono stati creati diversi algoritmi che cercano di ridurre il tempo di calcolo necessario. In questo capitolo verranno introdotti e analizzati alcuni dei più conosciuti algoritmi per il matching nel Data Distribution Management.

2.1 Brute force

Il *brute force* è l'algoritmo più semplice: controlla sequenzialmente (in un ordine arbitrario) ogni update extent con tutti i subscription extent. La sua complessità è quadratica (più precisamente $O(U \times S)$, dove U è il numero di update extent e S quello di subscription extent).[14] Generalmente paga la sua semplicità con scarsa scalabilità e lentezza di esecuzione, ma in caso in cui vi siano molte sovrapposizioni tra extent la probabilità di trovare un'intersezione nelle prime fasi è molto elevata, e di conseguenza nei passi successivi si possono evitare di considerare gli extent appartenenti a quella stessa region, ottenendo in certe situazioni vantaggi prestazionali sostanziali in confronto ad altri algoritmi.[11]

2.2 Grid-based

Per cercare di ridurre l'overhead computazionale del metodo brute force è stato creato l'algoritmo *grid-based*. Questo approccio evita le costose operazioni di matching tra tutte le region, partizionando il routing space in una griglia di celle.[8] Ogni extent viene quindi mappato su queste, e se un update extent e un subscription extent appartengono alla stessa cella essi vengono considerati sovrapposti anche se in realtà potrebbero non esserlo. Questo approccio è relativamente semplice da implementare, ed è molto più scalabile del brute force. Tuttavia, dato che gli extent all'interno di una cella sono considerati sempre sovrapposti, alcuni publisher trasmetteranno dati anche a subscriber non interessati, aumentando così le risorse di rete necessarie e richiedendo più potenza di calcolo per il filtraggio da parte dei subscriber.

Una variabile importante è la dimensione delle celle della griglia: dato che un extent che occupa due o più celle viene considerato appartenente ad ognuna di esse, più le celle sono piccole più il costo di gestione della mappatura aumenta (soprattutto se la simulazione è molto dinamica). D'altro canto, più le celle sono grandi più alto sarà il numero di dati irrilevanti trasferiti.[15] Molti ricercatori hanno tentato di determinare il valore ottimo di questo parametro,[16] creando anche algoritmi che generano griglie dalle dimensioni variabili nel tentativo di adattarsi alla distribuzione degli extent (sia nello spazio sia tra le varie iterazioni).

Vari studi [13, 17, 18] hanno analizzato i costi e l'efficienza di filtraggio di questo algoritmo. La dimensione della griglia gioca un ruolo fondamentale ma, dato che la dimensione ottima stessa è influenzata da molti fattori (larghezza di banda, velocità di calcolo, memoria disponibile, ...), risulta praticamente impossibile effettuare un'analisi precisa.

2.3 Hybrid

L'*approccio ibrido* usa il brute force su ogni cella elaborata dall'algoritmo grid-based per filtrare i messaggi irrilevanti. Ciò porta miglioramenti ad

entrambi gli algoritmi 'puri', perché rende il brute force decisamente più scalabile ed elimina i messaggi irrilevanti generati dall'algoritmo grid-based.[19] Per contro, prende anche alcuni svantaggi da entrambi: la scelta della dimensione della cella è ancora un fattore fondamentale per ottenere performance accettabili, e il costo del brute force rimane quadratico.

2.4 Improved sort-based

L'algoritmo *improved sort-based* [11] non genera connessioni irrilevanti, e tra gli algoritmi esatti risulta uno di quelli più scalabili, pur mantenendo una discreta semplicità. Ovviamente il costo quadratico intrinseco del matching non viene battuto: la sua forza risiede nel tentativo di alleggerire le parti più costose trasformandole in operazioni su vettori e matrici di bit.

Le dimensioni del routing space vengono elaborate separatamente, effettuando una proiezione ortogonale degli extent su ogni asse, e una volta ottenuti i risultati per ogni dimensione questi vengono combinati (due extent sono sovrapposti se e solo se essi sono sovrapposti in ogni dimensione del routing space).

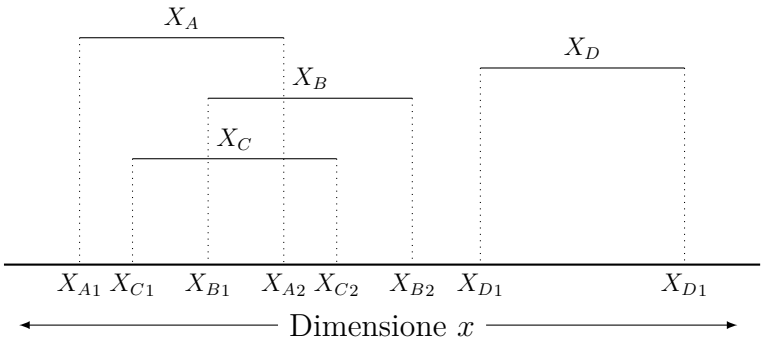

Figura 2.1: Proiezione degli extent sull'asse x.[11]

Il procedimento seguito dall'algoritmo improved sort-based sfrutta il fatto

che, in una dimensione, i subscription extent possono solo *precedere*, *seguire* o essere *sovrapposti* ad un dato update extent.

Esempio 2.4.1. Nella Figura 2.2:

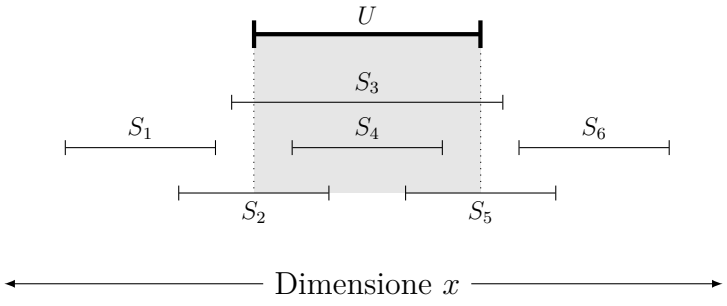

Figura 2.2: Possibili relazioni tra extent in una dimensione.

- S_1 **precede** U (inizia e termina prima dell'inizio di U);
- S_2 è **sovrapposto** ad U (inizia prima e termina dopo l'inizio di U);
- S_3 è **sovrapposto** ad U (S_3 inizia prima e termina dopo U);
- S_4 è **sovrapposto** ad U (U inizia prima e termina dopo S_4);
- S_5 è **sovrapposto** ad U (inizia prima della fine di U e termina dopo);
- S_6 **segue** U (inizia e termina dopo la fine U).

L'ultimo caso possibile (non raffigurato, ma evidente) è che subscription ed update extent combacino perfettamente. Si può quindi affermare che un subscription extent può solo precedere, seguire o essere sovrapposto ad un update extent.

2.4.1 L'algoritmo

Algoritmo 1 Improved sort-based su una dimensione.

```
L ← ∅
for all extents R_i ∈ RoutingSpace do
    L ← L + lower bound point of R_i
    L ← L + upper bound point of R_i
end for
sort L
SubscriptionSetBefore ← ∅
SubscriptionSetAfter ← all subscription extents ∈ RoutingSpace
for all points P_i ∈ L do
    R_i ← extent ID of P_i
    if R_i is a subscription extent then
        if P_i is lower bound point of R_i then
            SubscriptionSetAfter ← SubscriptionSetAfter \ R_i
        else                                  ▷ P_i is the upper bound point of R_i
            SubscriptionSetBefore ← SubscriptionSetBefore + R_i
        end if
    else                                      ▷ R_i is an update extent
        if P_i is the lower bound point of R_i then
            all extents in SubscriptionSetBefore do not overlap with R_i
        else                                  ▷ P_i is the upper bound point of R_i
            all extents in SubscriptionSetAfter do not overlap with R_i
        end if
    end if
end for
```

Una volta fissata una dimensione, l'algoritmo improved sort-based crea una lista ordinata L contenente i limiti superiori ed inferiori di *ogni* extent, inserisce tutti i subscription extent nell'insieme $SubscriptionSetAfter$ (tutti gli extent sono 'dopo' il punto iniziale) e crea l'insieme $SubscriptionSetBefore$

(vuoto, nessun extent è 'prima' del punto di partenza).

L'algoritmo procede quindi modificando questi due insiemi: un subscription extent viene rimosso da $SubscriptionSetAfter$ quando viene estratto dalla lista il suo limite inferiore, e viene inserito in $SubscriptionSetAfter$ quando viene estratto il suo limite superiore. In questo modo i subscription extent sono in uno dei tre stati possibili—*after* (dopo), *matching* (sovrapposto, né prima né dopo) o *before* (prima)—in relazione al punto che si sta elaborando.

Quando viene estratto il limite inferiore di un update extent, tutti i subscription extent contenuti nel $SubscriptionSetBefore$ sono già terminati, perciò si può dire che sono *prima* di lui, mentre quando viene estratto il suo limite superiore, tutti gli extent in $SubscriptionSetAfter$ devono ancora iniziare, perciò sono posti *dopo*. Salvando queste informazioni in una matrice si ottiene una tabella di 'non-sovrapposizione', che basterà quindi invertire per ottenere il risultato finale da combinare con le matrici di sovrapposizione delle altre dimensioni.

2.4.2 Vantaggi

L'ordinamento iniziale della lista (di costo $O(N \log N)$, ovvero subquadratico) è l'unica operazione che necessita delle coordinate. La parte rimanente può essere trasformata in operazioni su bit implementando i due insiemi di subscription extent come vettori di bit. Essendo le operazioni su bit generalmente tra le più veloci che un calcolatore riesca ad eseguire, il loro costo quadratico risulta in qualche modo 'ammortizzato', rendendo questo algoritmo uno tra i più veloci e scalabili.

2.4.3 Operazioni su bit

Nello specifico, l'algoritmo effettua quattro tipi di operazioni su vettori e matrici di bit. Ogni volta che viene estratto il limite inferiore di un update extent U_i effettua una **scrittura** di $SubscriptionSetBefore$ nell'i-esima riga

della matrice, per poi effettuare un **bitwise OR** di $SubscriptionSetAfter$ con l'i-esima riga della matrice quando U_i termina. Inoltre bisogna invertire la matrice di non-sovrapposizione tramite un **bitwise NOT** (al termine dell'elaborazione per una singola dimensione) per ottenere la matrice di sovrapposizione e, infine, quando le matrici delle varie dimensioni saranno disponibili, bisognerà combinare i dati in esse contenuti tramite una serie di **bitwise AND**.

Capitolo 3

Ottimizzazione dell'algoritmo improved sort-based

Considerate le virtù dell'improved sort-based, la decisione di concentrare gli sforzi su di esso è stata piuttosto facile, soprattutto per via della relativa semplicità che lascia la porta aperta a varie implementazioni. Come spesso accade, tuttavia, il punto di forza dell'improved sort-based risulta essere anche il suo svantaggio maggiore. Deve infatti la sua efficienza alla trasformazione delle parti dal costo maggiore in operazioni su bit, ma proprio per questo motivo necessita dell'intera matrice di bit delle sovrapposizioni (o non-sovrapposizioni), mentre altri algoritmi—come ad esempio il brute force—hanno la possibilità di salvare solo le informazioni riguardanti gli extent effettivamente sovrapposti.

3.1 Perfezionamento dell'algoritmo

Al crescere del numero di extent l'intera matrice arriva velocemente ad occupare un notevole spazio in memoria; inoltre l'algoritmo originale prevede la creazione di una matrice per ogni dimensione.[11] Dato che la potenza di calcolo del computer utilizzato dagli autori dell'improved sort-based (un Pentium® III 700) era molto inferiore rispetto a quella disponibile al giorno

d'oggi, e che quindi le istanze su cui han potuto testare l'algoritmo erano di dimensioni decisamente limitate (massimo $5\,000 \times 5\,000$ bit, meno di 3 megabyte),[11] è probabile che abbiano volutamente deciso di ignorare il problema. Con i processori disponibili oggi, tuttavia, un'istanza del genere verrebbe elaborata in meno di un millisecondo, e per ottenere risultati accettabili è necessario spingersi fino a 500 000 extent totali, che comportano matrici di più di 7 gigabyte. Si rende perciò necessario ridurre il più possibile il quantitativo di memoria utilizzato: per fortuna questo è possibile, ed anche molto facilmente.

3.1.1 Versione *default*

L'improved sort-based originale calcola la matrice di non-sovrapposizione per ogni dimensione, effettua un bitwise NOT su ognuna di esse per ottenere le matrici di sovrapposizione e infine le combina tramite un bitwise AND. La prima semplice modifica consiste nell'effettuare il bitwise NOT e il bitwise AND ogni volta che una dimensione è stata elaborata. Ciò riduce il numero di matrici $N \times N$ ad un massimo di due, pur mantenendo caratteristiche e prestazioni originali dell'algoritmo. Utilizzeremo questa versione leggermente modificata come versione di *default* con cui paragonare i risultati dei test, dato che la versione originale richiede troppa memoria.

Questo primo piccolo cambiamento porta inevitabilmente a riflettere sulla possibilità che l'algoritmo possa essere ulteriormente modificato in modo che utilizzi una sola matrice. Prima di tutto, però, conviene ragionare sulla possibilità di ridurre la quantità di operazioni sulle matrici.

3.1.2 Versione *lowmem*

Dato che vale, in un routing space con d dimensioni,

$$\neg A_1 \wedge \cdots \wedge \neg A_d \ \sim \ \neg(A_1 \vee \cdots \vee A_d),$$

è possibile modificare l'algoritmo in modo che effettui $d-1$ bitwise OR e un solo bitwise NOT, al posto di d bitwise NOT e $d-1$ bitwise AND.

Una volta effettuata la semplificazione risulta evidente che l'operazione di OR può essere eseguita direttamente sulla matrice risultato (senza necessità di una matrice di appoggio) durante la scrittura dei due *SubscriptionSet*: è sufficiente effettuare un bitwise OR del vettore $Before$ con la riga della matrice (al posto di sovrascriverla) quando un update extent inizia, per poi fare lo stesso con $After$ quando l'update extent terminerà. In questo modo, però, la prima operazione effettuata sulle righe non è più una scrittura e risulta quindi necessario inizializzare la matrice di output, cosa che prima poteva essere evitata. Dato che gli extent sono considerati sovrapposti *se e solo se* essi sono sovrapposti in ogni dimensione, bisogna impostare ogni bit della matrice a 0, perché essendo una matrice di non-sorvapposizione questo equivale a dire che ogni update extent è sovrapposto a tutti i subscription extent (o, se si preferisce, che l'algoritmo è stato eseguito precedentemente in una dimensione fittizia in cui tutti gli extent erano sovrapposti). Questa inizializzazione garantisce la correttezza del risultato finale.

Analisi delle modifiche

Il grosso vantaggio di questa versione *lowmem* (da *low memory*, poca memoria) è che—oltre a non necessitare di una seconda matrice—non richiede nessuna operazione aggiuntiva al termine dell'elaborazione di una dimensione. Sarà sufficiente invertire la matrice risultato una volta processate tutte le dimensioni (effettuando su di essa un bitwise NOT) per ottenere la matrice di sovrapposizione finale.

In sintesi, la versione *default* prevede le seguenti operazioni di costo quadratico:

- N scritture di N bit per ogni dimensione ($SubscriptionSetBefore$);
- N OR di N bit per ogni dimensione ($SubscriptionSetAfter$);
- d bitwise NOT di una matrice $N \times N$;
- $d - 1$ bitwise AND di una matrice $N \times N$;

che la versione *lowmem* riduce a:

- inizializzazione (a 0) di una matrice $N \times N$;
- N OR di N bit per ogni dimensione ($SubscriptionSetBefore$);
- N OR di N bit per ogni dimensione ($SubscriptionSetAfter$);
- *un* bitwise NOT di una matrice $N \times N$.

Sono stati completamente eliminati i bitwise AND, e viene effettuato un unico bitwise NOT invece di d. Per contro è necessario inizializzare la matrice, e tutte le scritture di $SubscriptionSetBefore$ vengono trasformate in bitwise OR.

Occupazione di memoria

La possibilità di effettuare tutti i calcoli utilizzando una sola matrice di non-sovrapposizione permette di risparmiare quasi il 50% dello spazio in memoria. Su 500 000 extent si passa infatti dai 14.55GB della versione *default* ai 7.28GB della *lowmem*. La ragione per cui il risparimo non risulta essere esattamente del 50% risiede nel fatto che devono essere in ogni caso allocati $2N$ bit per i $SubscriptionSet$ (valore comunque trascurabile se comparato agli N^2 bit occupati da una matrice).

3.2 Trasformazione in un *superset*

Un problema che rischia di influenzare la correttezza della soluzione generata dall'improved sort-based è il suo comportamento in caso siano presenti extent di un tipo che iniziano nello stesso punto dove altri dell'altro tipo terminano. Potrebbe avvenire infatti che questi extent non vengano considerati sovrapposti, perché è possibile che l'ordinamento posizioni i due estremi uguali in entrambi gli ordini. Nell'esempio in Figura 3.1, potrebbe avvenire che i punti dei due extent U ed S vengano ordinati sia come (U_i, U_f, S_i, S_f),

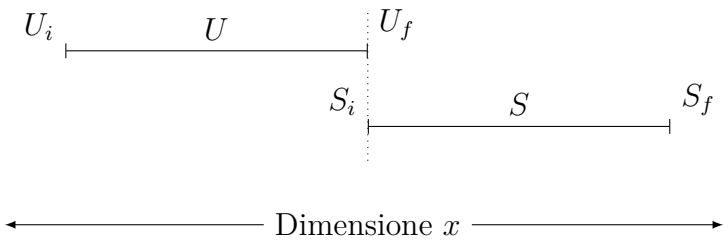

Figura 3.1: Esempio di extent con un solo punto in comune.

sia come (U_i, S_i, U_f, S_f). Nel primo caso però l'algoritmo considererebbe i due extent come *non sovrapposti* anche se hanno un punto in comune.

Per risolvere questo problema è sufficiente far sì che l'ordinamento posizioni sempre gli estremi inferiori *prima* di quelli superiori quando le loro coordinate coincidono. In questo modo ci si assicura che vengano correttamente elaborati anche gli extent con 'larghezza zero', ovvero quegli extent con limite inferiore e superiore coincidenti in almeno una dimensione.

Algoritmo 2 Confronto tra due punti estremi P_1 e P_2.

```
if P1 < P2 then
    put P1 before P2
else if P1 > P2 then
    put P1 after P2
else                                              ▷ P1 = P2
    if P1 is lower bound point of extent E then
        put P1 before P2
    else                          ▷ P1 is upper bound point of extent E
        put P1 after P2
    end if
end if
```

È evidente che istanze generate casualmente in routing space con dimen-

sioni molto ampie hanno una probabilità bassissima di avere punti estremi in comune, e quindi il numero di operazioni in più che il confronto perfezionato deve effettuare è praticamente nullo.[1] Tuttavia, in caso le istanze presentino molti punti estremi in comune, questo costo è destinato ad aumentare. Le operazioni aggiuntive incidono ovviamente solo sul costo dell'ordinamento, influenzando il tempo di esecuzione dell'intero algoritmo in modo irrisorio, ma esiste comunque la possibilità di ridurle a $O(N)$.

Modificando tutti i punti estremi presenti nella lista in modo che gli estremi superiori vengano spostati a destra di un ε (il più piccolo possibile per il tipo di dato utilizzato), si ottiene un problema in cui gli estremi inferiori che prima coincidevano con estremi superiori ora li precedono di ε, rendendo quindi sicuro che gli estremi coincidenti siano ordinati correttamente. A questo punto è possibile effettuare l'ordinamento semplicemente, ignorando il tipo di estremo che si sta processando.

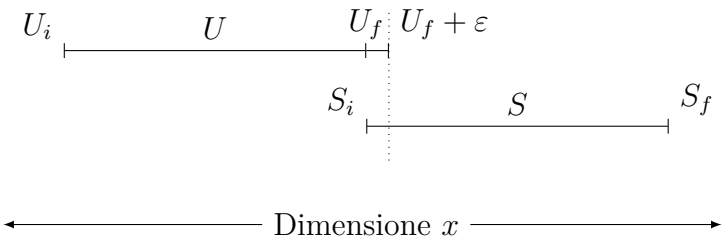

Figura 3.2: Esempio di extent dopo la trasformazione in un superset.

Analisi delle conseguenze

Effettuare questa operazione di costo lineare (è sufficiente scorrere la lista una volta per modificare tutti gli estremi inferiori) permette di produrre un

[1] Utilizzando QuickSort come algoritmo di ordinamento su istanze con 400 000 punti estremi, salvati come `int32` e generati in modo casuale nel range [`INT32_MIN`, `INT32_MAX`], è risultato dell'ordine di circa 30–40 operazioni aggiuntive.

ordinamento che venga processato in modo corretto, ma può generare alcune sovrapposizioni fittizie, trasformando quindi la soluzione in un *superset*.

La quantità di dati irrilevanti prodotta dipende da molti fattori (tra cui principalmente la distribuzione degli extent), ed è perciò praticamente impossibile effettuarne una stima. Tuttavia, la probabilità di generare sovrapposizioni aggiuntive è limitata anche dal fatto che—per lo stesso motivo per cui nella versione ottima è necessario utilizzare l'Algoritmo 2—la sovrapposizione dei punti estremi di due extent dopo la trasformazione in *superset* non assicura la sovrapposizione nella soluzione.

3.3 OpenCL

L'*Open Computing Language* (OpenCL) è uno standard aperto per la creazione di applicazioni parallele su sistemi eterogenei.[20] È sviluppato dal consorzio non-profit Gruppo Khronos (lo stesso che si occupa anche dello sviluppo delle librerie grafiche OpenGL, nonché altre librerie aperte).

OpenCL permette di sviluppare *kernel* (funzioni che verranno eseguite su un dispositivo OpenCL) in un linguaggio simile a C99. Grazie ad esso, questi kernel sono completamente portabili tra tutti i dispositivi che supportano OpenCL, indipendentemente dalle caratteristiche hardware, spesso completamente differenti anche tra le differenti generazioni di uno stesso prodotto. I vantaggi maggiori si ottengono eseguendo questi kernel sulle schede video delle ultime generazioni, sfruttando l'enorme potenza di calcolo e l'architettura altamente parallelizzabile di queste. In questo caso si parla di *General-Purpose computing on Graphics Processing Units* (GPGPU).

Un kernel OpenCL, benché sia programmato in un linguaggio piuttosto comune, segue un procedimento completamente differente da quello a cui si è abituati su architetture come x86, dove la parallelizzazione avviene secondo il modello *task-parallel* (un singolo processo che avvia svariati task paralleli).[21] È possibile seguire lo stesso questo approccio anche in un kernel OpenCL, ma la sua vera forza risiede nella programmazione con modello

data-parallel. In questo modello il kernel viene eseguito contemporaneamente su più unità di elaborazione, ognuna delle quali effettua gli stessi calcoli su una differente parte dei dati. Ciò spiega perfettamente il motivo per cui le schede video sono particolarmente adatte: buona parte delle elaborazioni effettuate dalle schede video necessitano di ripetere la stessa operazione molte volte su grosse quantità di dati (si pensi ad esempio all'applicazione di un filtro a tutti i pixel di un'immagine), e quindi negli anni sono state sviluppate con un'architettura parallela.

Per permettere la portabilità su dispositivi con architettura hardware completamente diversa, OpenCL compila i kernel a run-time. È perciò necessario creare anche una procedura, definita *host*, scritta in un linguaggio di programmazione interpretabile dai comuni processori, che si occupi di caricare i kernel da file, compilarli, eseguirli e processare i dati che i essi richiedono in input (e che generano come output).

Esempio 3.3.1. Eseguendolo contemporaneamente su almeno n unità di calcolo, l'Algoritmo 3 effettua la moltiplicazione per un dato valore (in questo caso 2) di ogni elemento del vettore V.

Algoritmo 3 Moltiplicazione per 2 di tutti gli elementi di un vettore V formato da n elementi con il modello di programmazione data-parallel.

$i \leftarrow$ ID of execution unit
if $i < n$ **then**
    $V_i \leftarrow V_i \times 2$
end if

Il funzionamento è piuttosto semplice, e si articola in due passaggi:

1. il kernel inizia chiedendo al sistema qual è il suo ID, a cui il sistema risponderà con un numero che varia tra 0 e il numero di unità di calcolo su cui il kernel è stato eseguito, ed univoco per ogni istanza;

2. se l'ID è minore della quantità di elementi da processare effettua le operazioni sull'elemento in posizione corrispondente all'ID.

In questo modo ogni unità di calcolo effettua la moltiplicazione di un singolo elemento del vettore, e se il numero di istanze generate è almeno uguale al numero di elementi contenuti nel vettore, ogni elemento verrà raddoppiato.

3.3.1 L'improved sort-based e OpenCL

Nota. *Data la limitata memoria disponibile sulle schede video, si è deciso di utilizzare come base per la versione OpenCL la versione* lowmem.

Le operazioni sui bit delle matrici e vettori (*NOT* e *OR*) effettuate dall'improved sort-based sono implementabili senza bisogno di modifica alcuna (a parte la rimozione del ciclo) nel modello data-parallel.

Bitwise *NOT* su vettore/matrice lineare

Il bitwise *NOT* su un vettore (o matrice salvata linearmente in memoria) richiede in input il vettore da invertire `vec` e la sua lunghezza `size`. È possibile leggere l'output nello stesso vettore `vec`.

Il kernel effettua una chiamata alla funzione `get_global_id(0)` per ottenere il suo `gid` (global ID, identificativo globale univoco tra tutte le istanze di questo kernel in esecuzione sul dispositivo), dopodiché, se l'ID è inferiore a `size`, effettua l'inversione dei bit di `vec[gid]`.

Algoritmo 4 Bitwise *NOT* di un vettore in OpenCL.

```
// calculates the bitwise NOT of the vector
__kernel void vector_bitwise_not(
                __global uint *vec,
                const uint size)
{
        const uint gid = get_global_id(0);
        if (gid < size)
                vec[gid] = ~vec[gid];
}
```

Bitwise OR tra vettori/matrici lineari

In aggiunta ai parametri richiesti in input dall'Algoritmo 4, il bitwise OR necessita anche del vettore `mask`, la maschera di bit con cui effettuare l'OR.

Anche in questo kernel viene prima ottenuto il `gid`, e se esso è inferiore a `size` effettua il bitwise OR tra `vec[gid]` e `mask[gid]`, salvando il risultato in `vec[gid]`.

Algoritmo 5 Bitwise OR di due vettori in OpenCL.

```
// calculates the bitwise OR of the vectors
__kernel void vector_bitwise_or(
                __global uint *vec,
                __global uint *mask,
                const uint size)
{
        const uint gid = get_global_id(0);
        if (gid < size)
                vec[gid] |= mask[gid];
}
```

3.3.2 Problemi

OpenCL è uno strumento molto potente, capace di aumentare drasticamente le performance di un programma, e l'improved sort-based si presta in modo particolarmente evidente all'approccio data-parallel. Tuttavia, per poter sfruttare appieno le potenzialità di OpenCL è necessaria una conoscenza del metodo data-parallel, dell'architettura delle moderne schede video e di OpenCL stesso di gran lunga superiore a quella che può essere ottenuta in pochi mesi di studio come autodidatta.

Come si potrà osservare nel prossimo capitolo, i risultati ottenuti da questa implementazione 'naive' sono a dir poco insoddisfacenti. La causa principale risiede nella cattiva utilizzazione della memoria, che risulta in uno

scambio di dati enorme tra la RAM di sistema e quella della scheda video, formando un pesantissimo collo di bottiglia.[21, 22] Oltretutto, anche una corretta ed efficiente implementazione dell'improved sort-based verrebbe comunque limitata dalla quantità di memoria disponibile su una scheda video.

3.4 Multithreading

Se si escludono le operazioni su vettori e matrici, l'algoritmo improved sort-based su una dimensione non è facilmente parallelizzabile, dato che aggiorna le informazioni contenute nei due *SubscriptionSet* man mano che scorre la lista ordinata di punti estremi, e queste informazioni dipendono strettamente dall'elemento della lista attualmente processato. L'elaborazione di una dimensione è però completamente indipendente dalle altre, perciò è possibile distribuire su vari thread i calcoli delle diverse dimensioni.

3.4.1 Dettagli implementativi

Per evitare di ricadere nelle problematiche di memoria possedute dalla versione originale dell'algoritmo, è necessario trovare un modo per permettere ai thread di lavorare su una sola matrice. Vien da sé che anche in questo caso sarà utilizzata come base la versione *lowmem*: dato che l'algoritmo opera al più su una riga della matrice, la versione *threaded* utilizzerà un array di Nmutex, dove ogni elemento è associato alla riga della matrice corrispondente. In questo modo un solo thread potrà modificare una data riga, ma gli altri thread avranno la possibilità di lavorare comunque sulle altre $N - 1$. Tutte le operazioni, eccetto l'inversione finale della matrice, saranno parallelizzate.

Rispetto alla versione *lowmem*, la versione *threaded* occupa più memoria, in quanto necessita di una coppia di *SubscriptionSet* per ogni dimensione ($N \cdot (d-1)$ bit) e l'array di mutex (N volte la dimensione di un mutex). Questa memoria aggiuntiva dipende quindi sia dal numero di extent sia dal numero di dimensioni del routing space, ma è anche in questo caso trascurabile se comparata alla dimensione della matrice.

Capitolo 4

Analisi prestazionale

Nota. *I test sono stati eseguiti su una macchina dotata di processore Intel® Core™ i7-2600 operante a 3.40GHz, con 4 core e Hyper-Threading, 16GB di RAM DDR3, scheda video nVidia® GeForce® GTX-580 (modello con 3GB di RAM GDDR5) e sistema operativo Ubuntu Linux 11.04 a 64 bit. Gli algoritmi sono stati implementati in linguaggio C e compilati con GCC 4.5.2. I dati riportati sono la media di 100 iterazioni. Le istanze sono generate in modo casuale e formate dallo stesso numero di update e subscription extent, perciò viene indicato solo il numero* totale *di extent.*

Come si può vedere in Figura 4.1, un particolare importante di quest'algoritmo è che il *tipo di dati* in input influisce in modo assolutamente trascurabile sul tempo di esecuzione. Ciò avviene perché l'algoritmo utilizza le informazioni presenti nel routing space *solo* per ordinare i punti estremi: il resto dell'elaborazione ha un costo computazionale quadratico, il che rende trascurabili le differenze nel tempo di esecuzione dell'ordinamento subquadratico. Uno dei vantaggi è che non sarà necessario effettuare i prossimi test su ogni tipo di dato in input. La scelta—completamente arbitraria—è ricaduta sulle versioni che utilizzano dati in formato `double`.

Nelle due sezioni seguenti si cercherà di analizzare i risultati sperimentali ottenuti in *quasi 45 ore di test.* I dati stessi sono visualizzabili in formato grafico nella Sezione 4.3 ed in formato tabellare nella Sezione 4.4. La

versione *default* verrà utilizzata come indice di paragone al fine di valutare le prestazioni delle versioni modificate, dato che effettua le stesse operazioni dell'algoritmo improved sort-based originale (seppure in un ordine lievemente differente), pur utilizzando una ridotta quantità di memoria.

4.1 Variazione del numero di extent

Sono stati effettuati tre diversi tipi di test in questa categoria: su routing space tridimensionali, su routing space quadridimensionali e su routing space tridimensionali 'zero-set', ovvero con tutti gli extent che iniziano e terminano nell'origine di ogni dimensione. Non tutte le versioni sono state testate in ognuna di queste categorie.

Lowmem

In ogni test effettuato, sia la versione *default* sia quella *lowmem* hanno presentato una crescita del tempo di esecuzione molto 'pulita' all'aumentare del numero di extent. Come ci si poteva aspettare, la seconda risulta essere consistentemente più veloce (fino al 35% in Tabella 4.2), con un vantaggio ancora maggiore in routing space quadridimensionali (Tabella 4.5), pur necessitando circa del 50% in meno della memoria.

Una stranezza nei risultati si può notare nella Tabella 4.2: il guadagno percentuale della versione lowmem pare aumentare col numero di extent, e questo non sarebbe ovviamente possibile se entrambe le implementazioni avessero la stessa complessità computazionale. L'elenco dei cambiamenti nel codice è il seguente:

- `memcpy` su una riga sostituita con bitwise OR di tutta la riga;
- allocazione della matrice aggiuntiva rimossa;
- bitwise NOT e AND delle matrici rimossi;
- aggiunto bitwise NOT al termine dell'elaborazione;

- `malloc` sostituita con `calloc`;

e nessuna di queste operazioni dovrebbe avere un costo super-quadratico.

Nel tentativo di trovare il problema sono stati eseguiti svariati test atti a controllare la correttezza della soluzione, ed è anche stata creata una versione definita *lowbig*: una *lowmem* che alloca la matrice d'appoggio pur non utilizzandola. Tuttavia, i test effettuati non hanno evidenziato errori, e la *lowbig* (Tabella 4.3) presenta lo stesso aumento della differenza percentuale riscontrato con la versione *lowmem*.

Superset

La versione *superset* risulta lievemente più lenta della *default* in caso di extent generati casualmente (Tabella 4.6), dato che effettua una trasformazione in più su ogni extent. Un vantaggio dell'ordine dell'1% può però essere notato nel caso limite in cui tutti gli extent inizino e terminino nello stesso punto (Tabella 4.7). Questa differenza è destinata a ridursi con l'aumentare del numero di extent, ma per contro su piccole istanze arriva ad influire anche fino al 5% del tempo totale (molto probabilmente anche di più, se si riducesse maggiormente il numero di extent).

OpenCL

I dati ottenuti con OpenCL (Tabella 4.8) non sono minimamente soddisfacenti per via del pessimo utilizzo della memoria di questa semplice implementazione, risultando in un codice migliaia di volte più lento rispetto alle altre versioni.

Threaded

La versione *threaded*—se si escludono i risultati ottenuti con 50 000 extent, in cui la creazione dei thread richiede più tempo dell'elaborazione non parallela—presenta un guadagno evidente (fino al 51% rispetto alla versione

default, che supera anche il 53% in routing space quadridimensionali). Tuttavia ha un comportamento molto più incerto, arrivando in alcuni casi (200 000 extent) a risultare addirittura più lenta della versione *lowmem*. Dal grafico in Figura 4.3 si può però notare che, escludendo i picchi, l'andamento segue la funzione quadratica rappresentata con una linea tratteggiata.

4.2 Variazione del numero di dimensioni

Nei test effettuati su 200 000 extent (Tabella 4.9) si può notare che la versione *lowmem* risulta fino al 38% più veloce della versione *default*, e sfiora il 40% su 250 000 (Tabella 4.11. Anche in questo caso, così come in Tabella 4.2, pare che i guadagni percentuali siano ordinati in modo crescente, suggerendo un costo super-quadratico della versione *default*.

La versione *threaded* presenta sempre alcuni picchi, ma in genere ha un guadagno superiore al 50% rispetto alla *default*, per arrivare fino a passare il 53% in alcuni casi (Tabelle 4.10 e 4.12). Inoltre, nonostante sia basata sulla versione *lowmem*, non sembra presentare un aumento nei guadagni rispetto alla versione *default*. Tuttavia è difficile stabilirlo con certezza, perché potrebbe essere una conseguenza del suo comportamento incerto.

4.3 Grafici

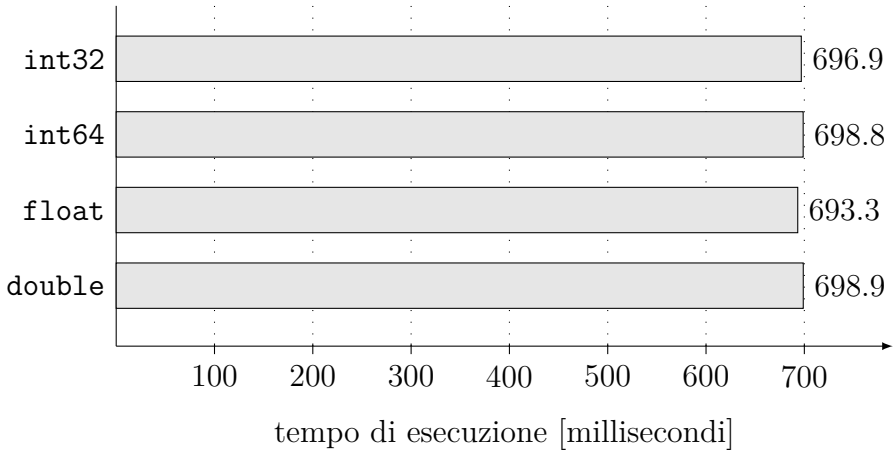

Figura 4.1: Tempi di esecuzione (in millisecondi) per tipo di dato. Algoritmo eseguito su routing space monodimensionali contenenti 200 000 extent.

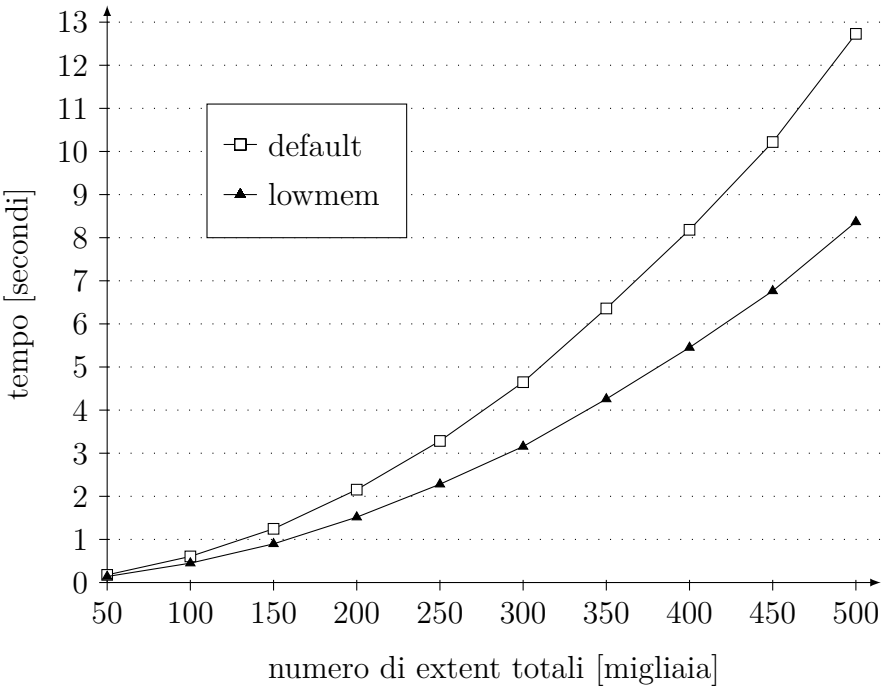

Figura 4.2: Tempi di esecuzione delle versioni *default* e *lowmem* al variare del numero di extent, in routing space tridimensionali.

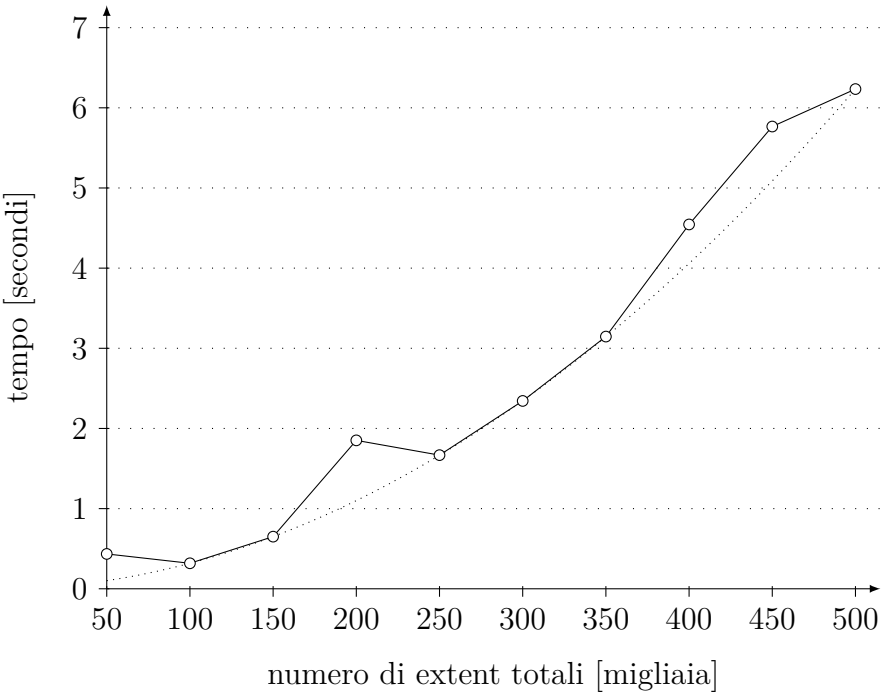

Figura 4.3: Tempi di esecuzione della versione *threaded* al variare del numero di extent, in routing space tridimensionali. La linea tratteggiata rappresenta la funzione $y = 0.0000232277x^2 + 0.000854084x + 0.0000515018$ (calcolata considerando come valori di x le *migliaia* di extent).

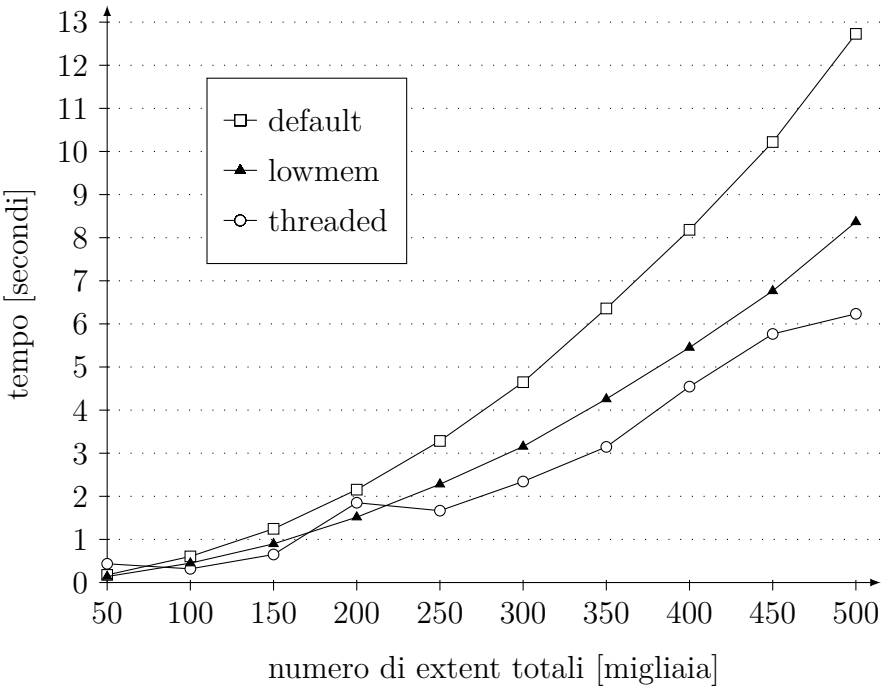

Figura 4.4: Tempi di esecuzione delle versioni *default*, *lowmem* e *threaded* al variare del numero di extent, in routing space tridimensionali.

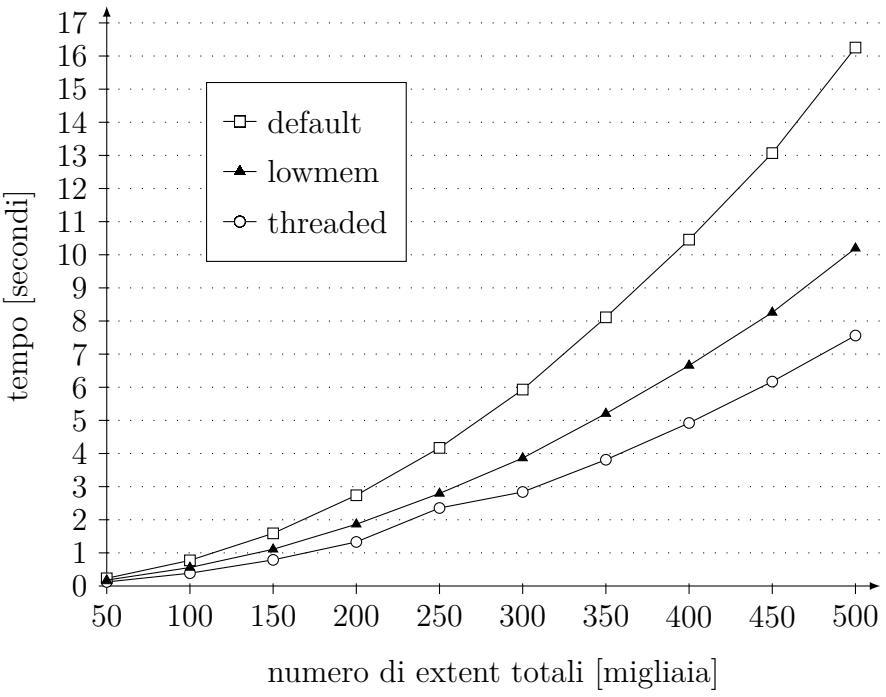

Figura 4.5: Tempi di esecuzione delle versioni *default*, *lowmem* e *threaded* al variare del numero di extent, in routing space quadridimensionali.

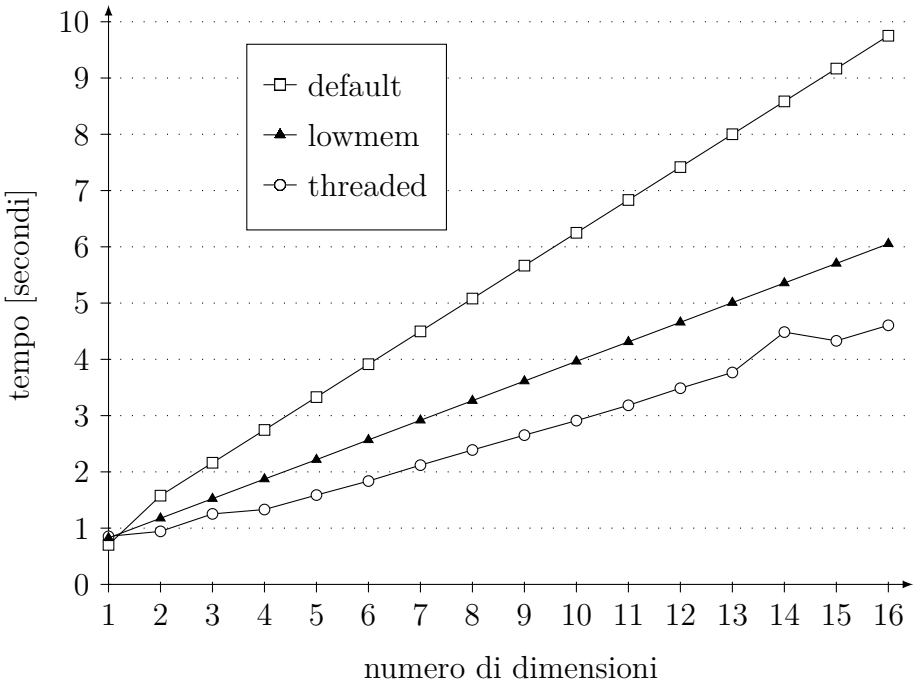

Figura 4.6: Tempi di esecuzione delle versioni *default*, *lowmem* e *threaded* al variare del numero di dimensioni, in routing space con 200 000 extent.

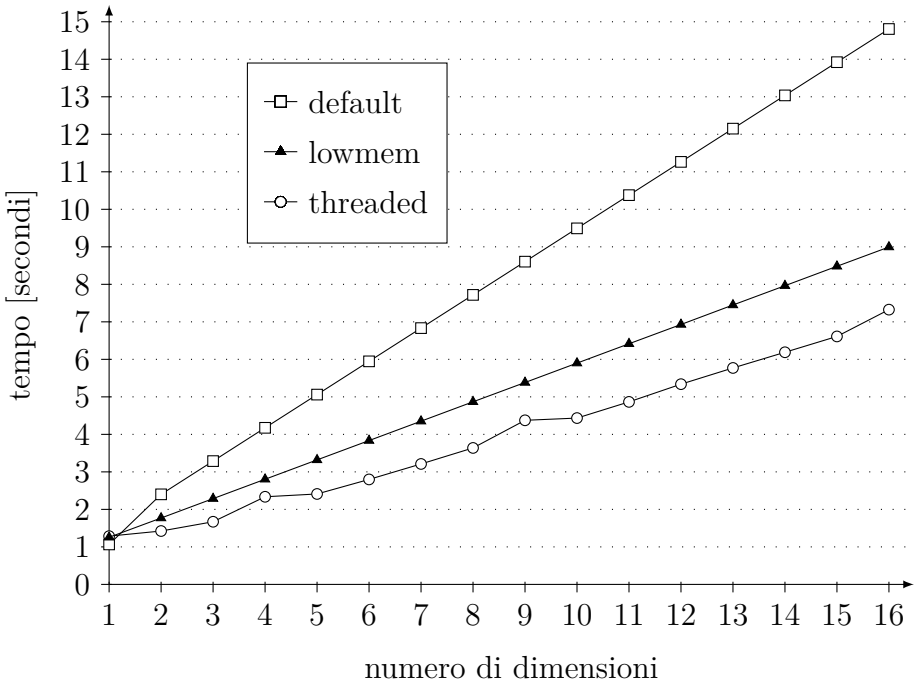

Figura 4.7: Tempi di esecuzione delle versioni *default*, *lowmem* e *threaded* al variare del numero di dimensioni, in routing space con 250 000 extent.

4.4 Tabelle

4.4.1 Variazione del numero di extent

Extent	*default*	
	Memoria (gigabyte)	Tempo (secondi)
50 000	0.1455	0.1775
100 000	0.5821	0.6056
150 000	1.3097	1.2462
200 000	2.3283	2.1546
250 000	3.6380	3.2846
300 000	5.2387	4.6478
350 000	7.1305	6.3580
400 000	9.3133	8.1828
450 000	11.7871	10.2190
500 000	14.5520	12.7289

Tabella 4.1: Tempi di esecuzione della versione *default* al variare del numero di extent, in routing space tridimensionali.

Extent	*lowmem*		
	Memoria (gigabyte)	Tempo (secondi)	Guadagno % (vs *default*)
50 000	0.0728	0.1398	21.2394
100 000	0.2910	0.4495	25.7761
150 000	0.6549	0.8967	28.0453
200 000	1.1642	1.5160	29.6389
250 000	1.8190	2.2803	30.5760
300 000	2.6194	3.1560	32.0969
350 000	3.5653	4.2573	33.0402
400 000	4.6567	5.4514	33.3797
450 000	5.8936	6.7645	33.8046
500 000	7.2760	8.3655	34.2795

Tabella 4.2: Tempi di esecuzione della versione *lowmem* al variare del numero di extent, in routing space tridimensionali.

Extent	*lowbig*	
	Tempo (secondi)	Guadagno % (vs *default*)
50 000	0.1395	21.4085
100 000	0.4492	25.8256
150 000	0.8965	28.0613
200 000	1.5160	29.6389
250 000	2.2809	30.5577
300 000	3.1570	32.0754
350 000	4.2586	33.0197
400 000	5.4522	33.3699
450 000	6.7642	33.8076
500 000	8.3628	34.3007

Tabella 4.3: Tempi di esecuzione della versione *lowbig* al variare del numero di extent, in routing space tridimensionali.

Extent	*threaded*			
	Memoria (gigabyte)	Tempo (secondi)	Guadagno % (vs *default*)	Guadagno % (vs *lowmem*)
50 000	0.0730	0.4347	-144.8879	-210.9270
100 000	0.2914	0.3180	47.4879	29.2518
150 000	0.6554	0.6503	47.8142	27.4742
200 000	1.1650	1.8514	14.0701	-22.1270
250 000	1.8200	1.6672	49.2415	26.8862
300 000	2.6206	2.3433	49.5828	25.7512
350 000	3.5666	3.1463	50.5139	26.0958
400 000	4.6582	4.5449	44.4578	16.6287
450 000	5.8954	5.7665	43.5709	14.7537
500 000	7.2780	6.2339	51.0257	25.4810

Tabella 4.4: Tempi di esecuzione della versione *threaded* al variare del numero di extent, in routing space tridimensionali.

Extent	*default*	*lowmem*		*threaded*	
	Tempo (secondi)	Tempo (secondi)	Guadagno % (vs *default*)	Tempo (secondi)	Guadagno % (vs *default*)
50 000	0.2316	0.1780	23.1434	0.1219	47.3791
100 000	0.7740	0.5578	27.9327	0.3865	50.0705
150 000	1.5895	1.1082	30.2799	0.7881	50.4193
200 000	2.7394	1.8642	31.9486	1.3297	51.4613
250 000	4.1697	2.7932	33.0120	2.3543	43.5368
300 000	5.9305	3.8625	34.8706	2.8384	52.1384
350 000	8.1112	5.2041	35.8405	3.8121	53.0024
400 000	10.4567	6.6576	36.3318	4.9219	52.9306
450 000	13.0692	8.2575	36.8171	6.1709	52.7828
500 000	16.2556	10.1952	37.2819	7.5617	53.4824

Tabella 4.5: Tempi di esecuzione delle versioni *default*, *lowmem* e *threaded* al variare del numero di extent, in routing space quadridimensionali.

Extent	*superset*	
	Tempo (secondi)	Guadagno % vs *default*
50 000	0.1775	0.0000
100 000	0.6087	-0.5119
150 000	1.2490	-0.2247
200 000	2.1589	-0.1996
250 000	3.2901	-0.1675
300 000	4.6528	-0.1074
350 000	6.3642	-0.0975
400 000	8.1911	-0.1015
450 000	10.2315	-0.1223
500 000	12.7328	-0.0307

Tabella 4.6: Tempi di esecuzione della versione *superset* al variare del numero di extent, in routing space tridimensionali.

Extent	*default*	*lowmem*		*superset*	
	Tempo (secondi)	Tempo (secondi)	Guadagno % (vs *default*)	Tempo (secondi)	Guadagno % (vs *default*)
50 000	0.1410	0.0986	30.0709	0.1352	4.1135
100 000	0.5215	0.3608	30.8150	0.5052	3.1256
150 000	1.1106	0.7596	31.6046	1.0854	2.2691
200 000	1.9535	1.3275	32.0450	1.9243	1.4948
250 000	3.0111	2.0356	32.3968	2.9754	1.1856
300 000	4.2935	2.8560	33.4808	4.2650	0.6638
350 000	5.9269	3.8884	34.3939	5.8829	0.7423
400 000	7.6812	5.0020	34.8799	7.6143	0.8710
450 000	9.6095	6.2311	35.1568	9.5369	0.7554
500 000	12.0380	7.7378	35.7220	11.9414	0.8025

Tabella 4.7: Tempi di esecuzione delle versioni *default*, *lowmem* e *superset* al variare del numero di extent, in routing space zero-set tridimensionali.

Extent	*OpenCL*
	Tempo (secondi)
50 000	14.7628
100 000	30.8290
150 000	48.8069
200 000	67.6391
250 000	120.9267
300 000	147.4049
350 000	120.3488
400 000	138.1261
450 000	N/A
500 000	N/A

Tabella 4.8: Tempi di esecuzione della versione *OpenCL* al variare del numero di extent, in routing space tridimensionali.

4.4.2 Variazione del numero di dimensioni

Dimensioni	*default*	*lowmem*	
	Tempo (secondi)	Tempo (secondi)	Guadagno % (vs *default*)
1	0.7030	0.8265	-17.5677
2	1.5771	1.1756	25.4582
3	2.1620	1.5236	29.5283
4	2.7451	1.8725	31.7876
5	3.3289	2.2168	33.4074
6	3.9138	2.5683	34.3783
7	4.4969	2.9170	35.1331
8	5.0804	3.2656	35.7216
9	5.6654	3.6137	36.2146
10	6.2493	3.9645	36.5609
11	6.8324	4.3106	36.9094
12	7.4168	4.6580	37.1966
13	8.0010	5.0078	37.4103
14	8.5847	5.3563	37.6065
15	9.1677	5.7045	37.7760
16	9.7512	6.0539	37.9164

Tabella 4.9: Tempi di esecuzione delle versioni *default* e *lowmem* al variare del numero di dimensioni, in routing space con 200 000 extent.

Dimensioni	*threaded*		
	Tempo (secondi)	Guadagno % (vs *default*)	Guadagno % (vs *lowmem*)
1	0.8539	-21.4718	-3.3207
2	0.9425	40.2392	19.8292
3	1.2531	42.0408	17.7554
4	1.3309	51.5180	28.9249
5	1.5877	52.3045	28.3773
6	1.8366	53.0747	28.4912
7	2.1196	52.8646	27.3352
8	2.3890	52.9764	26.8438
9	2.6528	53.1746	26.5891
10	2.9106	53.4255	26.5840
11	3.1832	53.4109	26.1552
12	3.4868	52.9873	25.1431
13	3.7661	52.9296	24.7952
14	4.4844	47.7634	16.2788
15	4.3290	52.7802	24.1132
16	4.6049	52.7764	23.9354

Tabella 4.10: Tempi di esecuzione della versione *threaded* al variare del numero di dimensioni, in routing space con 200 000 extent.

Dimensioni	*default*	*lowmem*	
	Tempo (secondi)	Tempo (secondi)	Guadagno % (vs *default*)
1	1.0657	1.2567	-17.9225
2	2.4011	1.7697	26.2963
3	3.2875	2.2869	30.4365
4	4.1746	2.8032	32.8511
5	5.0603	3.3193	34.4050
6	5.9467	3.8348	35.5139
7	6.8330	4.3513	36.3193
8	7.7185	4.8680	36.9308
9	8.6044	5.3825	37.4447
10	9.4920	5.8993	37.8497
11	10.3778	6.4154	38.1815
12	11.2644	6.9292	38.4859
13	12.1504	7.4490	38.6934
14	13.0368	7.9634	38.9160
15	13.9225	8.4809	39.0850
16	14.8067	8.9945	39.2539

Tabella 4.11: Tempi di esecuzione delle versioni *default* e *lowmem* al variare del numero di dimensioni, in routing space con 250 000 extent.

Dimensioni	*threaded*		
	Tempo (secondi)	Guadagno % (vs *default*)	Guadagno % (vs *lowmem*)
1	1.2866	-20.7285	-2.3796
2	1.4232	40.7286	19.5815
3	1.6702	49.1955	26.9667
4	2.3362	44.0387	16.6609
5	2.4110	52.3543	27.3639
6	2.7985	52.9408	27.0242
7	3.2112	53.0053	26.2025
8	3.6381	52.8648	25.2643
9	4.3748	49.1564	18.7222
10	4.4344	53.2828	24.8319
11	4.8661	53.1101	24.1491
12	5.3381	52.6108	22.9620
13	5.7705	52.5075	22.5329
14	6.1888	52.5286	22.2851
15	6.6108	52.5171	22.0505
16	7.3252	50.5280	18.5593

Tabella 4.12: Tempi di esecuzione della versione *threaded* al variare del numero di dimensioni, in routing space con 250 000 extent.

Capitolo 5

Sviluppi futuri

I test hanno evidenziato che la variante migliore tra quelle implementate è la *threaded* (basata sulla *lowmem*), i cui tempi di esecuzione risultano circa dimezzati rispetto alla versione originale. È però possibile apportare ulteriori modifiche all'algoritmo, anche se per la limitata quantità di tempo a disposizione non è stato possibile svilupparle o analizzarle in questa tesi. A seguire verrà comunque presentata una breve descrizione delle possibilità.

5.1 Superset

Le modifiche della versione *superset* incidono solo sull'ordinamento, quindi possono ovviamente essere combinate con la *lowmem* e la *threaded*. I tempi di esecuzione di queste versioni risulteranno allineati a quelli della rispettiva base, non è stata perciò effettuata una nuova serie di test.

Su routing space generici la versione *superset* non risulta conveniente, ma potrebbe essere utilizzata in caso il routing space presenti determinate caratteristiche effettuando un'analisi a priori o a runtime (in modo simile a quanto effettuato dalle varianti adattive dell'approccio grid-based), per poi decidere se utilizzare la *superset* o meno. Se ad esempio si sta processando un routing space in cui gli extent hanno coordinate intere e sono distribuiti su un range discretamente ridotto questa versione risulterà con buona probabilità utile.

Da considerare è anche la densità dei punti nel routing space: concentrazioni elevate per il tipo di dato utilizzato generano molte più false sovrapposizioni rispetto a punti in media ben distanziati. La quantità di extent nel routing space è un'altra possibile discriminante tra le due versioni, perchè in caso il routing space sia 'adatto' il guadagno sarà maggiore con meno extent (in percentuale, le operazioni quadratiche avranno meno peso). Al contrario, più extent sono presenti nel problema meno overhead si avrà in caso gli extent non risultino molto sovrapposti (anche questo in percentuale).

5.2 Approccio ibrido e sistemi distribuiti

L'algoritmo improved sort-based *lowmem* può essere parallelizzato sia sulle dimensioni (come l'implementazione *threaded* proposta e analizzata in questa tesi), sia suddividendo le operazioni sulle matrici tra diversi thread. Il procedimento di matching vero e proprio tuttavia non può essere parallelizzato, dato che opera sequenzialmente sulla lista di punti estremi e sui due *SubscriptionSet*. Necessitando di tutta la matrice, inoltre, anche se fosse parallelizzabile non potrebbe comunque essere distribuito.

Dato che HLA è stato sviluppato apposta per interconnettere simulazioni distribuite, avere un algoritmo di matching che possa a sua volta essere distribuito può risultare una carta vincente. Una possibilità in questa direzione è rappresentata dall'ibridazione con gli algoritmi grid-based—allo stesso modo di quanto presentato nella Sezione 2.3—utilizzando però l'improved sort-based al posto del brute force: ogni cella può essere elaborata indipendentemente da un diverso sistema, e la scalabilità dell'improved sort-based permetterebbe una certa 'elasticità' nella scelta della dimensione delle celle.

5.3 OpenCL

Come più di una volta precisato, il fatto che i tempi ottenuti dalla versione *OpenCL* in questa tesi siano insoddisfacenti non deve trarre in ingan-

no: data la natura data-parallel delle operazioni effettuate sulle matrici, un'implementazione ragionata ed ottimizzata dell'algoritmo con OpenCL può risultare veramente molto efficiente. Le potenzialità fornite da questo approccio appaiono evidenti se si pensa che i supercomputer moderni sono sempre più basati su schede GPGPU.[6] Inoltre, OpenCL diventa ancora più valido se combinato con l'approccio ibrido distribuito trattato nella sezione precenente, perché la distribuzione avrebbe tra i vari effetti quello di ridurre la dimensione delle matrici da allocare sulla memoria delle singole schede video, permettendo l'utilizzo di OpenCL anche su istanze molto grandi.

Conclusioni

In questa tesi sono stati progettate, implementate e valutate delle modifiche all'algoritmo improved sort-based, utilizzato per la risoluzione del problema di matching all'interno del componente Data Distribution Management dello standard High-Level Architecture per la creazione di simulazioni parallele e distribuite.

Nello specifico, è stato prima modificato l'algoritmo in modo da ridurre la quantità di memoria necessaria, in seguito è stato velocizzato riducendo il numero di operazioni necessarie, ed infine è stato adattato in modo da poter essere implementato con diverse tecnologie per la parallelizzazione (multithread e OpenCL). È stata inoltre progettata e implementata la versione *superset*, che si differenzia in modo particolare dalle altre perchè crea un sovrainsieme della soluzione ottima. Una soluzione di questo tipo costringerà il DDM a trasferire più dati tra i vari federati, ma in certi casi ciò può comunque risultare conveniente.

Queste modifiche hanno portato ad una significativa riduzione del tempo di elaborazione (fino ad oltre il 50% utilizzando i thread), nonostante la memoria necessaria per processare un routing space con d dimensioni sia circa d volte inferiore rispetto a quella richiesta dall'algoritmo originale.

Con l'aumentare della potenza di calcolo di computer e supercomputer aumenta la complessità delle simulazioni che essi devono elaborare. Il DDM rappresenta una parte fondamentale per le performance di una simulazione HLA, perché è il componenete atto a ridurre il più grosso collo di bottiglia di un sistema distribuito: la rete di interconnessione. A questo bisogna ag-

giungere il fatto che una maggiore complessità della simulazione implica un maggior numero di federati, regioni ed extent, ed il costo del DDM cresce quadraticamente con essi. Ciò assicura che questo componente continuerà ad influire sulle prestazioni di HLA in modo significativo anche nel futuro. Ricercatori di tutto il mondo sono perciò alla ricerca di metodi per rendere l'operazione di matching nel DDM il più veloce possibile. Sempre più studi infatti si stanno concentrando su algoritmi che generino sovrainsiemi della soluzione ottima (vedi la Sezione 2.2). Ovviamente, allo stesso modo della versione *superset* sviluppata in questa tesi, ogni sovrapposizione aggiuntiva comporta il trasferimento di dati non necessari tra i federati, perciò la maggior parte delle forze, in questi studi, è solitamente concentrata su tecniche atte a limitare il più possibile i falsi positivi.[13, 16, 17, 18]

La generazione di un sovrainsieme non ha costo necessariamente quadratico, perciò questo tipo di approccio può essere enormemente vantaggioso in caso la rete di interconnessione tra i federati sia sovradimensionata, dato che il trasferimento di dati non necessari non va ad incrementare il peso sul collo di bottiglia del sistema. Avvicinandosi al limite della rete però le performance sono destinate a calare drasticamente, fino ad arrivare ad avere un sistema completamente congestionato.

Un fatto importante è che più la simulazione è statica, più l'approccio grid-based risulta vantaggioso, perché è necessario modificare la cella di appartenenza di un extent solo in caso esso cambi coordinate. Nell'improved sort-based questo non sembra essere facilmente implementabile, a meno di apportare modifiche sostanziali che renderebbero l'algoritmo molto più complesso (e forse anche meno efficiente). Per contro, però, le performance dell'improved sort-based non dipendono in alcun modo né dalla dimensione degli extent, né dal grado di sovrapposizione degli extent nel routing space, a differenza degli approcci grid-based (e non solo).

Bisogna inoltre considerare che gli algoritmi di matching sono presenti anche nelle simulazioni create in molti videogiochi, e quindi non sono solo supercomputer, sistemi distribuiti o comunque molto potenti a dover eseguì-

re le operazioni tipiche del DDM. In questo campo si presenta una serie di problemi differenti, tra cui il risparmio energetico, fondamentale per i sistemi alimentati a batteria (ma non limitato ad essi). Se un videogioco non richiede il 100% delle risorse, i nuovi personal computer si mantengono in uno stato di risparmio energetico. Raramente i videogiochi hanno quantità elevate di extent da confrontare (e di conseguenza il matching viene eseguito in tempi insignificanti), tuttavia è possibile che questa operazione richieda—anche solo per un istante—il massimo delle risorse disponibili: per far fronte alla richiesta il sistema aumenterà il voltaggio dei suoi componenti, ma non tornerà istantaneamente in modalità di risparmio appena il carico di lavoro calerà. Dato che le operazioni di matching si ripeteranno abbastanza velocemente, il PC si manterrà al massimo della potenza per tutto il tempo, nonostante ciò non sia assolutamente necessario.

Una possibilità per ovviare a questo potrebbe essere limitare volontariamente il numero di operazioni che il matching può effettuare consecutivamente, forzando di tanto in tanto lo scheduler a mettere in coda il processo. Benché controintuitivo, se la limitazione è implementata correttamente (e il videogioco non necessita del 100% delle risorse) l'esperienza dell'utente non sarà in alcun modo influenzata.

Per concludere, vorrei esprimere un paio di considerazioni personali. Durante l'implementazione e i test ho sentito molto la mancanza di una serie di standard per la valutazione degli algoritmi di matching, soprattutto per quanto riguarda la correttezza. In molti altri campi sono disponibili delle suite di problemi con relative soluzioni su cui testare la propria implementazione (si veda ad esempio la TSPLIB [23] per quel che riguarda il Traveling Salesman Problem), mentre nulla del genere è disponibile per il DDM.

Un'altra cosa che mi ha lasciato perplesso è che la maggior parte degli algoritmi per il DDM vengono presentati in dettaglio in articoli e pubblicazioni scientifiche, ma le effettive implementazioni non sono mai rese disponibili (nonostante non abbiano quasi mai premesse commerciali), rendendo necessario, per chiunque voglia effettuare un confronto, un consistente spreco di

tempo e fatica per implementare nuovamente anche l'algoritmo altrui, rischiando inoltre che l'implementazione sia sbagliata (e quindi risultando in confronti non corretti). Un esempio di questo può essere osservato nella serie di test effettuata da J. Ahn, C. Sung e T. G. Kim in [24], che sembra suggerire che l'algoritmo improved sort-based abbia performance sostanzialmente diverse al variare del grado di sovrapposizione.

Per quanto mi riguarda, il codice sviluppato durante questa tesi verrà ovviamente reso disponibile sotto licenza open source.

Bibliografia

[1] *A. Gabrielli.* Grande Dizionario Italiano 2012 - *Voce "simulare". HOEPLI.*

[2] *AA.VV.* Enciclopedia Italiana Treccani Online - *Voce "simulazione". <http://www.treccani.it/enciclopedia/simulazione>.*

[3] *B. Kennedy.* Uncle Sam Wants You (To Play This Game). *New York Times, 11 Luglio 2002.*

[4] *C. McLeroy.* History of Military Gaming. *Soldiers Magazine, Settembre 2008 - Vol. 63, No. 9, pp. 4–15. Official U.S. Army Magazine.*

[5] *G. Jean.* Game Branches Out Into Real Combat Training. *National Defense Magazine, Febbraio 2006. National Defense Industrial Association.*

[6] *AA.VV.* List of Top 500 Supercomputers, *Novembre 2012. <http://www.top500.org/>.*

[7] *K. L. Morse, J. S. Steinman.* Data Distribution Management in the HLA: Multidimensional Regions and Physically Correct Filtering. *In Proceedings of the 1997 Spring Simulation Interoperability Workshop, 1997 - pp. 343–352.*

[8] *A. Boukerche, C. Dzermajko.* Performance Comparison of Data Distribution Management Strategies. *In Proceedings of the 5th IEEE International Workshop on Distributed Simulation and Real-Time Applications, 2001 - pp. 67–75.*

[9] *I. Tacic, R. Fujimoto.* Synchronized Data Distribution Management in Distributed Simulation. *In Proceedings of the 1997 Spring Simulation Interoperability Workshop, 1997.*

[10] *A. Boukerche, A. Roy.* In Search of Data Distribution Management in Large Scale Distributed Simulations. *In Summer Computer Simulation Conference (SCSC), 2000. The Society for Modeling and Simulation International (SCS), IEEE Conference Publications.*

[11] *C. Raczy, G. Tan, J. Yu.* A Sort-Based DDM Matching Algorithm for HLA. *In ACM Transactions on Modeling and Computer Simulation (TOMACS), 2005 - pp. 14–38. ACM.*

[12] *R. Fujimoto, T. McLean, K. Perumalla, I. Tacic.* Design of High Performance RTI Software. *In Proceedings of Fourth IEEE International Workshop on Distributed Simulation and Real-Time Applications (DS-RT 2000), 2000 - pp. 89–96. IEEE Conference Publications.*

[13] *R. Ayani, F. Moradi, G. Tan.* Optimizing Cell-size in Grid-Based DDM. *In Proceedings of Fourteenth Workshop on Parallel and Distributed Simulation (PADS 2000), 2000 - pp. 93–100. IEEE Conference Publications.*

[14] *M. D. Petty, K. L. Morse.* Computational Complexity of HLA Data Distribution Management. *In Proceedings of the 2000 Fall Simulation Interoperability Workshop, 2000.*

[15] *D. J. Van Hook, S. J. Rak, J. O. Calvin.* Approaches to Relevance Filtering. *In Proceedings of the 11th Workshop on Standards for the Interoperability of Distributed Simulations, 1994 - pp. 367–369.*

[16] *Y. Zhang.* A Simulation Platform for Investigating Data Filtering in Data Distribution Management. *M. Sc. Thesis, School of Computing, National University of Singapore, 2000.*

[17] *G. Tan, R. Ayani, Y. Zhang, F. Moradi.* An Experimental Platform for Data Management in Distributed Simulation. *In Proceedings of Simulation Technology and Training Conference, 2000 - pp. 371–376.*

[18] *G. Tan, R. Ayani, Y. Zhang, F. Moradi.* Grid-Based Data Management in Distributed Simulation. *In Proceedings of the 33rd Annual Simulation Symposium, 2000 - pp. 7–13.*

[19] *G. Tan, Y. Zhang, R. Ayani.* A Hybrid Approach to Data Distribution Management. *In Proceedings of the 4th IEEE International Workshop on Distributed Simulation and Real-Time Applications, 2000 - pp. 55–61. IEEE Conference Publications.*

[20] *Sito del Gruppo Khronos. <http://www.khronos.org/opencl/>.*

[21] *AA.VV.* AMD Accelerated Parallel Processing: OpenCL. *AMD Accelerated Parallel Processing Programming Guide, 2012. Advanced Micro Devices, Inc.*

[22] *AA.VV.* NVIDIA OpenCL Best Practices Guide. *NVIDIA Optimization Guide, 2009. NVIDIA.*

[23] *Sito dell'università di Heidelberg, pagina della libreria TSPLIB. <http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/>.*

[24] *J. Ahn, C. Sung, T. G. Kim.* A Binary Partition-Based Matching Algorithm for Data Distribution Management. *In Proceedings of the 2011 Winter Simulation Conference (WSC), 2011 - pp. 2723–2734. IEEE Conference Publications.*