Alma Mater Studiorum · Università di Bologna

Campus di Cesena - Scuola di Scienze
Corso di Laurea in Scienze e Tecnologie Informatiche

Design and Implementation of a Testbed for Data Distribution Management

Tesi di Laurea in Reti di Calcolatori

Relatore
GABRIELE D'ANGELO

Presentata da
FRANCESCO BEDINI

II Sessione
Anno Accademico 2012 - 2013

"What we do for ourselves dies with us.
What we do for others and the world remains,
and is immortal."
- Albert Paine

Abstract

Data Distribution Management (DDM) is a core part of High Level Architecture standard, as its goal is to optimize the resources used by simulation environments to exchange data. It has to filter and match the set of information generated during a simulation, so that each federate, that is a simulation entity, only receives the information it needs. It is important that this is done quickly and to the best in order to get better performances and avoiding the transmission of irrelevant data, otherwise network resources may saturate quickly.

The main topic of this thesis is the implementation of a *super partes* DDM testbed. It evaluates the goodness of DDM approaches, of all kinds. In fact it supports both region and grid based approaches, and it may support other different methods still unknown too. It uses three factors to rank them: execution time, memory and distance from the optimal solution. A prearranged set of instances is already available, but we also allow the creation of instances with user-provided parameters.

This is how this thesis is structured. We start introducing what DDM and HLA are and what do they do in details. Then in the first chapter we describe the state of the art, providing an overview of the most well known resolution approaches and the pseudocode of the most interesting ones. The third chapter describes how the testbed we implemented is structured. In the fourth chapter we expose and compare the results we got from the execution of four approaches we have implemented.

The result of the work described in this thesis can be downloaded on sourceforge using the following link: `https://sourceforge.net/projects/ddmtestbed/`. It is licensed under the GNU General Public License version 3.0 (GPLv3).[11]

Figure 1: DDMTestbed download link

Prefazione[1]

Questa tesi tratta di diversi aspetti riguardanti il Data Distribution Management (DDM), nell'ambito degli ambienti di simulazione distribuiti aderenti allo standard High Level Architecture (HLA).

Il DDM si occupa di gestire la trasmissione dei messaggi tra i vari ambienti di simulazione, chiamati federati. Il suo compito è quello di ottimizzare le risorse di rete, facendo in modo che vengano trasmesse soltanto le informazioni veramente necessarie ai vari federati, in modo che non vengano saturate.

Abbiamo sviluppato un testbed in *C* per la valutazione degli approcci di risoluzione in quanto, al momento, non è liberamente disponibile alcun software *super partes* riguardante questo campo di ricerca.

La tesi è così strutturata: nei primi due capitoli è descritto lo stato dell'arte, introducendo il DDM e HLA ed evidenziando con particolare cura i pro e i contro di ogni approccio di risoluzione, fornendo lo pseudocodice dei principali. Il capitolo 3 descrive come è stato implementato il testbed per la valutazione di quest'ultimi. Nel quarto capitolo vengono esposti e confrontati i risultati ottenuti dall'esecuzione di quattro diversi approcci di risoluzione noti in letteratura da me implementati secondo le specifiche del testbed. Questi sono stati poi valutati utilizzando istanze con diverse peculiarità e dimensioni.

[1]This section was added to fulfil "Università di Bologna"'s requirements for non-Italian thesis. The rest of the thesis is written in English.

Questa tesi è scritta interamente in inglese. Ho preso questa decisione in quanto ormai la quasi totalità dei testi scientifici sono pubblicati in tale idioma, la lingua *de facto* dell'informatica e delle pubblicazioni scientifiche. È infatti l'unica che può essere utilizzata per predisporre e consentire la massima diffusione dei risultati scientifici ottenuti.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction to Data Distribution Management

This chapter explains what Data Distribution Management (DDM) is and why it plays a major role in distributed simulations. To do so we'll need to quickly explain what is High Level Architecture, the architecture whose DDM belongs to. Afterwards we describe different approaches and algorithms that have been created during the last 15 years to handle DDM.

1.1 High Level Architecture

High Level Architecture (HLA) is a distributed computer simulation standard that allows interoperability between heterogeneous simulation systems.[1] Nowadays simulations can be so high detailed that they can't be executed on a single machine any longer. Many computers can be linked together via a computer network, forming a cluster, and exchanging information and state updates. HLA ensures that simulation environments built by different manufacturers are able to communicate one another using a common set of rules and protocols.

1.1.1 Version History

The U.S. Department of Defense (U.S. DoD) Architecture Management Group (AMG), under the leadership of the Defense Modeling and Simulation Office (DMSO), started developing HLA baselines in 1995. These baselines were approved by the U.S. Department of Defense in 1996.[23] The first complete version of the standard was published in 1998 and was known as *HLA 1.3*. It was later standardized as `IEEE` standard 1516 in 2000[1].

The most recent version of the standard was published in 2010 and it includes the current U.S. DoD standard interpretations.[2]

1.1.2 HLA Structure

HLA is made up of 3 components:

Interface specification: Defines how HLA compliant simulation environments interact using a Run-Time Infrastructure (RTI). RTI is a supporting software that consists of an implementation of six groups of services (whose complete implementation is not compulsory), provided as programming libraries and application programming interfaces (APIs), as defined by the HLA specifications.

Object Model Template (OMT): It states what information are exchanged between simulations and how they are documented.

Rules: Simulations must obey to these rules to be HLA compliant.

In HLA technical jargon, every simulation environment is known as a *federate*, whereas a collection of federates sets up a *federation*.

This is all you need to know to understand the topics discussed in the rest of this thesis. Now we can start to introduce the main argument of this thesis: Data Distribution Management.

1.2 Data Distribution Management

A centralized simulation would have full access to the system memory assigned to its process, and so all information are fully available to each task. In a distributed environment, federates would certainly need to exchange information too. This is where Data Distribution Management comes in.[16]

DDM is the service included in RTI that provides a scalable, efficient mechanisms for distributing information between federates. Its task is to handle data transmission and state updates that are sent between the various simulations. Its goal is to optimize the available resources, that is minimizing the volume of data transmitted over the network that links the federates together.

In fact it's very common that a federate only needs a subset of all the available data. For example, if we had a federation that is simulating traffic flow in a big metropolis or in an entire nation, we probably would have a federate that simulates the state of all the traffic lights located in the case in question. A travelling car would only be interested to the state of traffic lights place in the upcoming intersections. The state of all the other traffic lights would be completely irrelevant to that car's driver.

Federates during each message exchange can be divided in two groups: the ones that publish information, called *publishers*, and the ones that want to receive that information, called *subscribers*. DDM has to match their transfer intents in an optimal way in order to minimize network's usage.

On each transmission session DDM goes through the following steps:

1. Subscribers declare what information they want to receive, publishers declare what information they intend to send.

2. A matching algorithm detects the intersections between these transfer intents.

3. All subscribers that match with a certain publisher must receive data from the relative publisher. In some implementations,[15] this is achieved

creating multicast groups.[1]

4. Publishers send their data via their multicast group. Subscribers may need to execute an additional filtering to select the information they wanted.

The final filtering is only necessary if the matching algorithm is not optimal, that is when is allowed to obtain false positives in the result (i.e. solution provided is sub-optimal). This can be allowed, in certain circumstances, when a limited amount of superfluous data entails an improvement in the computational complexity of the matching algorithm.

The opposite case is not allowed; all the matching intents has to be detected by the matching algorithm, otherwise at least a subscriber won't receive at least a chunk of information it has legitimately requested.

[1] Multicast groups allows one-to-many communication over an IP infrastructure. The sender just sends one message that is then delivered to all the intended recipients.

Chapter 2

The State of the Art

If DDM would not be implemented by RTI, the only way to ensure that each subscriber would receive all the data it needs is that all publishers would send their data to each subscriber. Every subscriber then would waste time filtering the required data by itself, moreover a lot of network bandwidth would be wasted. It's trivial to see that this method is completely inefficient as it allows to a huge amount of useless data to occupy network's resources.

Let n, m be the number of publishers and subscribers respectively. The amount of data that is transmitted over the network can be quantified as:

$$\sum_{i=0}^{n} DataSize(i) * m$$

This would quickly saturate network's resources when the number of federates increases.

In order to overcame this waste of resources, many DDM approaches have been developed in the last 15 years. The following sections describe the most well-known.

2.1 How Information are Matched: the Routing Space

In order to declare what information publishers intend to transmit and subscribers intend to receive, federates generate extents on a *routing space*. A routing space is a multidimensional coordinate system whereon the simulation takes place. Federates create extents on it to indicate the area they want to receive/provide information. An extent is a rectangular subspace of the routing space; a group of extents sets up a region.[1] Regions submitted by publishers are called *update regions*, regions submitted by subscribers are named *subscription regions*.

When a subscription region is superimposed to an update region, the subscriber which made the subscription must receive data from the relative publisher.

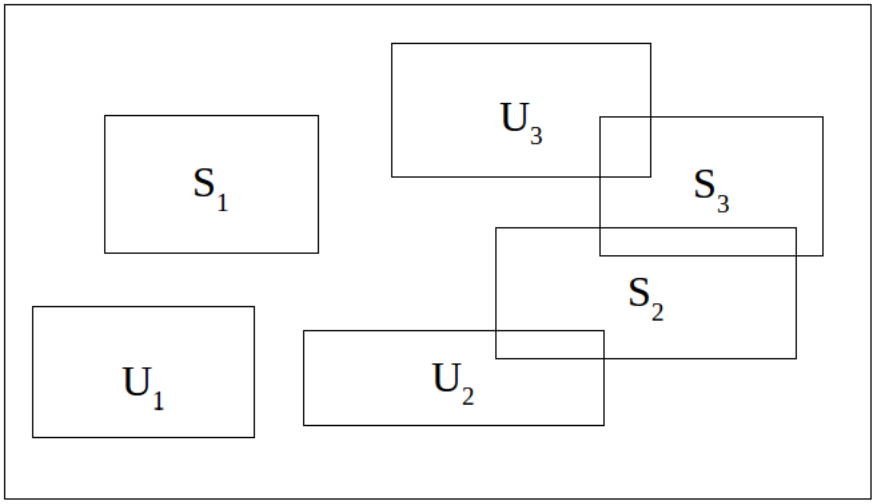


Figure 2.1: Update and subscription extents inside a 2D routing space.

Figure 2.1 shows three update extents (marked with a capital U) and three subscription extents (marked by a capital S) placed inside a bidimensional

[1] It is common practice to consider a region as an indivisible part, making the terms "extent" and "region" interchangeable.

routing space. U_1 and S_1 are not overlapped to any other extent, hence they shouldn't send nor receive any data in this session. On the other hand, U_2 is superimposed to S_2 whereas U_3 is overlapped to S_3. This means that U_2 will send data to S_2 as U_3 will provide data to S_3. The fact that S_2 is overlapped to S_3 is not to be taken into consideration by the matching algorithm as two subscribers don't exchange data one another. This would apply to two update extents too.

For example, we could simulate smartphones' GPS and network reception in a certain area using a 3D routing space. Two dimensions would be used to indicate the position of the smartphones and of the the antennas and GPS satellites, whereas the third would be used to distinguish between between GPS and network coverage. Hence we can treat smartphones as subscribers and the antennas and satellites as publishers. Update extents would represent the coverage area of the publishers whereas the subscription extent would represent the signal strength of the subscribers. When a publisher extent overlaps an update extent the relative mobile phone is considered able to communicate with the relative cell or satellite.

2.2 Matching Approaches

Matching approaches are mainly divided in two categories: *grid-based* and *region-based* approaches. In the following sections we'll describe the most relevant ones. These two categories are not mutually exclusive; they can be combined to appraise their strong points and lessen their weak points.

In the following sections, when it will be possible to state the computational cost of an approach, it will be shown consistently using n, m as the number of publishers and subscribers respectively and d as the number of dimensions.

2.2.1 Region-based Approaches

Region-based approaches require that a centralized coordinator compares extents one another, or that each federate compare its extents with the one generated by the others. Region Based approaches usually have a higher computational cost than Grid-based approaches, but they tend to generate less false positives than Grid-based ones (in fact most of them provide an optimal solution).

2.2.1.1 Brute Force

This approach performs all the possible comparisons between every update extent and each subscription extent. If two extents are superimposed on every dimension they are considered matched.

Pro: Easiness of implementation, straightforward, provides an optimal solution.

Cons: Not scalable; this method produces the maximum number of comparisons.

Let m be the number of subscribers' extents and n the number of publishers' extents. Computational complexity of the brute force approach is:

$$\Theta(d \cdot (m \cdot n))$$

The pseudocode of a possible implementation of this approach can be found in section A.1 on page 63.

2.2.1.2 Sort Based

This approach lowers the number of comparisons ordering the extents' edges.[20] Sort based method works on a dimension at a time, projecting extents' bounds on each dimension as shown in figure 2.2. We recall that an extent is considered matched to another if and only if is matched on every dimension.

According to this approach a vector for each dimension is created. It contains the relative extents' edges in a given dimension. Hence edges are sorted and read sequentially.

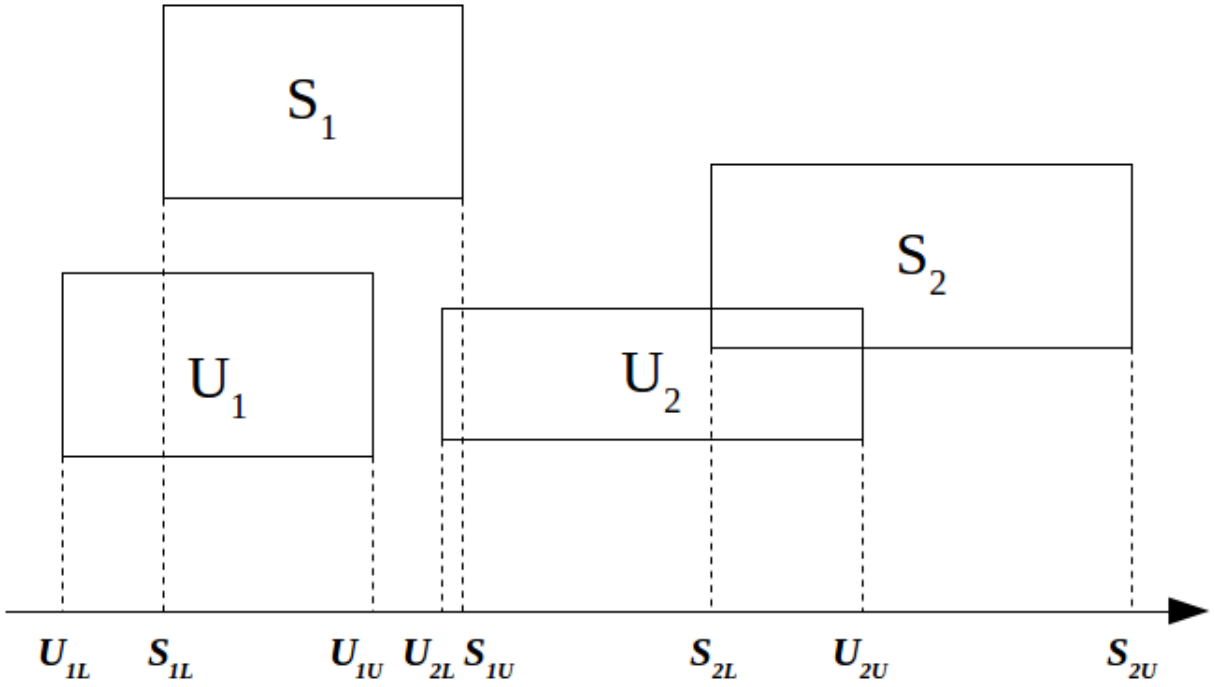


Figure 2.2: Extents edges projection on a given dimension.

This approach takes advantage of the ordered relation that states that an extent can only precede, include or follow another extent. When we extract edges from the list we can obtain a subscription or an update edge. We can keep trace of the state of the subscription extents using two lists so defined:

SubscriptionSetBefore: contains all the subscription extents that are already ended (whose upper edge has already been pulled out from the vector).

SubscriptionSetAfter: contains all the subscription extents that follows (whose lower edge has not been extracted from the vector yet).

When we extract an edge that is part of an update extent we can draw some conclusions analysing the state of the two lists just described. If we extract the lower bound we can state that all the subscribers in the *SubscriptionSetBefore* are not overlapped to the current update extent, whereas if we extract an upper bound we can say the same for the subscribers that are still in *SubscriptionSetAfter*.

As scanning the vectors has a linear cost, sort based computational complexity depends on the sort complexity. If we use Introspective sort algorithm[2] we could consider it linearithmic in the number of edges[17], hence we get:

$$\Theta(d \cdot ((m+n) \cdot \log(m+n)))$$

Pro: The number of comparison is lowered compared to the brute force approach, so it's more scalable.

Cons: It requires a little more memory to perform matching (as d additional vectors has to be stored).

Algorithm A.2 on page 64 shows a possible *sort based* implementation.

2.2.1.3 Binary Partition

This method uses a *divide et impera* approach, similar to the one adopted by the *Quicksort* algorithm.[13] As well as the *sort based*, also binary partition approach examines one dimension at a time. Edges are then sorted and recursively binary partitioned using the median as the pivot value. We obtain two subsets so defined:

Left (L): A set that contains all the extents that terminates before the pivot value.

Right (R): A set that contains all the extents that begin after the pivot value.

[2]Introspective sort is an improved version of the Quicksort algorithm that has a complexity of $\Theta(n \cdot log(n))$ even in the worst case, when Quicksort would have a $\Theta(n^2)$ cost.

This partition would not be complete as it not includes the extents that include the pivot value. We call this group of extent **P**. These are unnaturally included in the L partition, even if they don't strictly belong to it. The key factor of this approach is that the extents that belong to this P partition are automatically considered matched as they share a common value. The next step involves the following comparisons:

- L subscription extents with P update extents
- L update extents with P subscription extents
- R subscription extents with P update extents
- R update extents with P subscription extents

After this L and R are recursively partitioned until their both empty.

Considering that the computational cost depends on the extents distribution in the routing space, it's difficult to estimate its cost. As the vectors are initially sorted, as for the *sort based* approach, the cost would be $\omega(d \cdot (n+m) \cdot \log(n+m))$.

This algorithm works better if the pivot falls in zones where most extents overlaps, as comparison cost is lowered this way. The opposite case is not dramatic either, as the four comparison would not take place if the P partition would be empty. The worst case (i.e. the highest number of comparisons) happens when L, R, and P have a similar size.

Figure 2.3 on page 12 shows a partition example. Extents S_1 and U_2 are placed in the L partition as they end before the pivot value. Extent S_6 is placed on the R partition as it begins after the pivot. Extent U_3, S_4 and S_5 are placed in the P partition as they include the pivot value. We get, at no cost, that U_3, S_4 and S_5 are overlapped on the given dimension. The next step is to check whether S_1 is superimposed to U_3, if U_2 is overlapped to S_4 or S_5, etc. (i.e. the same goes for the extents in the R set).

Pro: The number of comparison is lowered compared to the brute force

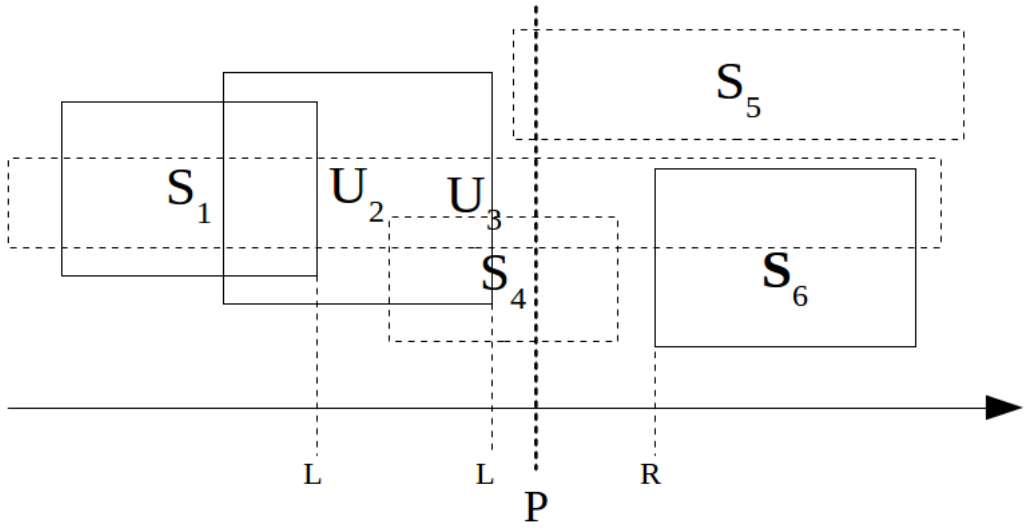


Figure 2.3: Binary partition on a given dimension

approach. It works best in clustered instances, as more extents would fall in the P partition.

Cons: More complicated data structures has to be stored. If partition isn't performed *in place* six vectors (three for the subscribers, and three for the publishers) has to be allocated on each node of the recursion tree.

The pseudocode of a possible implementation of this approach can be found in section A.3 on page 65.

2.2.2 Grid-Based

Grid-based approaches treat the routing space as a chessboard, that is they divide it in a grid of cells. An extent can lay on one or more contiguous cells. Approaches that belong to this categories tend to be less restricting than *region-based* ones, wasting more network bandwidth. Their advantage is that their computational cost is often lower.

One of the most important element for the success of these approach is to correctly set the size of the cell, in fact this is a key factor and many studies

has been carried out about this topic.[5, 21] Smaller cells bring a higher level of precision in matching detection but they involve the use of more resources during the execution of the matching algorithm. Bigger cells, on the other side, require less matching resources but tend to provide worse results.

2.2.2.1 Static Approach

Static approach (also known as *Fixed Approach*) is the most straightforward *grid-based* approach. It just considers matched two extents if they lay, even partially, on the same cell. This means that two extents might not be superimposed at all, anyway they would be considered matched if they occupy the same cell, causing the generation of false positives.

There is a multicast group preallocated for each cell. Publishers and subscribers join a group if one of the extents they generated lays even partially on the relative cell. This means that publishers who are the only federates in a cell would send data even if there is nobody to receive it in that multicast group.[7]

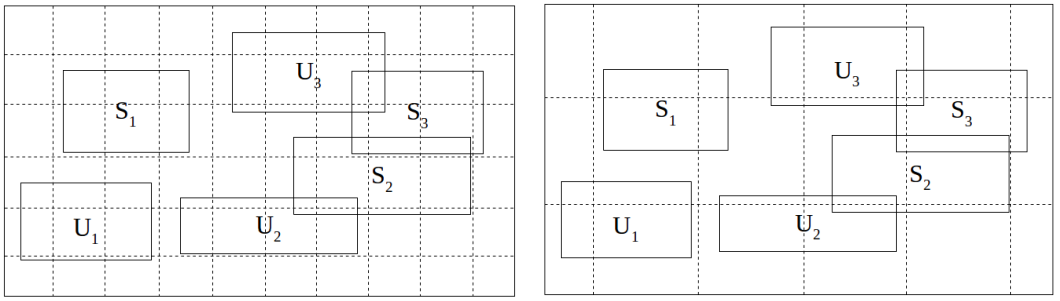
Figure 2.4 shows how crucial can the dimension of the cells be when using this approach. In figure 2.4(a) small cells are used, whereas in figure 2.4(b) bigger cells are shown. Applying the *static grid-based* approach, it's easy to see that the number of false positive increases as the area of the cells grows (see fig.2.4(c) and 2.4(d)).

Pro: No matching performed.

Cons: Lots of wasted bandwidth, depending on the size of the cells and the extents.

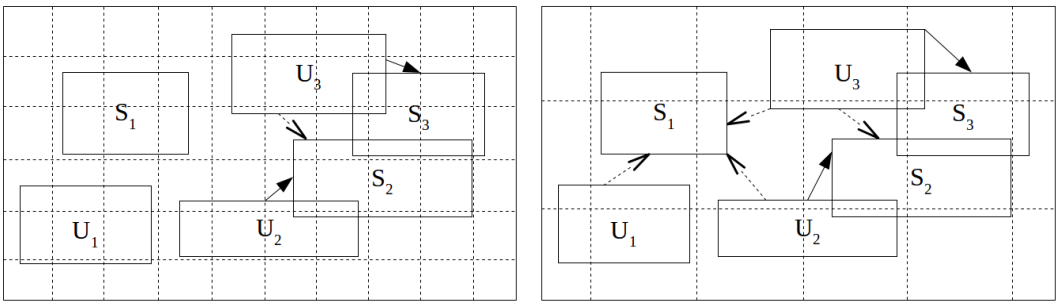
2.2.3 Miscellaneous Approaches

This section reports three approaches that don't lay completely in the previous categories, as they try to get the best from both.



(a) Small cells

(b) Large cells



(c) Matches detected using small cells

(d) Matches detected using large cells

Figure 2.4: Comparison between large and small grid cells. A dashed arrow indicates an erroneous match whereas a plain one shows a correct match.

2.2.3.1 Hybrid Approach

This kind of approach, in some situation, "can reduce both the number of irrelevant messages of the *grid-based* DDM and the number of matching of the *region-based* approach".[22] It consists in applying a *static grid-based* and then filtering the result using the *brute force region-based* approach. This allows to reduce the complexity of the *brute force* algorithm and decrease the number of false positives identified by the *static grid-based* approach.

It's possible to use other *region-based* approaches to improve the last filtering. When the number of matched extents detected by the *static grid-based* approach is high enough to justify the overhead of more complex data structures, a more sophisticated approach should be used.

Pro: The second filtering permits to reach an optimal solution.

Cons: Its efficiency depends on the grid size.

2.2.3.2 Grid-filtered region-based

This approach is very similar to the Hybrid Approach described in the previous section. The difference is that the *region-based* matching is not performed on any cell, but only on cells that are occupied over a certain percentage by extents. Choosing the correct threshold is as thorny as choosing the best cell size in a *grid-based* approach.[6]

This method should be used when networks' resources are highly available and a limited amount of false positives can be acceptable.

Pro: It performs less matching than the other miscellaneous approaches.

Cons: Produces false positives. It has to be fine-tuned choosing the right cell size and threshold based on the given instance.

2.2.3.3 Dynamic Approach

Each federate is the owner of one or more cells and knows every subscribers' and publishers' related activity. This makes a central coordinator unnecessary.

When the owner detects a match it allocs the relative multicast group and notifies the involved extents.[7]

Pro: Distributed, more efficient than *grid-based*, the second filtering permits to reach an optimal solution.

Cons: Its efficiency depends on the grid size.

2.3 Summary

In this chapter the most well-known matching approaches have been analysed. Table 2.1 shows a quick view of the properties of the approaches we

Table 2.1: Approaches comparison

Approach	Matching Performed	Distributable	Optimal Match
Static Grid-Based		✓	
Brute Force	✓	✓	✓
Sort Based	✓		✓
Binary Partition	✓		✓
Hybrid	✓		✓
Grid-Filtered	✓	✓	
Dynamic	✓	✓	✓

described.

There is a huge variety of different approaches, and establishing which one works best on a given instance it's not so obvious. Choosing the right cell size in *grid-based* approaches is a key factor that can't be underrated. The testbed we implemented wants to be useful in making clear when it's convenient to use a certain approach instead of another.

The following chapter explains how we tried to achieve this goal.

Chapter 3

The Testbed

This chapter explains how the testbed was designed and projected. We start by explaining the testbed's goals, then we proceed explaining its structure and describing in detail how it is structured and how its components work.

At the end of this chapter, an explanation about how to implement an approach on our testbed can be found.

3.1 The Testbed's Goals

We wanted to create a testbed to easily compare different approaches and different parameters within the same approach, producing a useful and handy tool that would allow researchers to focus on their innovative approach rather than losing time in implementing a new *ad hoc* benchmark to execute their evaluation tests.

Moreover, in some papers, it looks like authors simulate a routing space specifically built to just disclose the strong points of their algorithms, with the result that each approach is *"better"* than all the others.

To remove all doubt, we want to provide a standard and independent measuring instrument to compare the goodness of a proposed approach on a fixed set on instances whose solution is known. Researchers can implement

their approaches, run them on our benchmark and compare their results with the one got by other approaches already implemented. Obviously we encourage researches to share their code to perform efficiency comparisons on a wider set of resolutive methods and instances.

It is however possible to use our tool only to check for the validity of a proposed algorithm; our testbed, among other things, checks that the provided solution is acceptable and returns the number of false positives if the solution is sub-optimal.

As we're supporting OpenMP[19] too, our testbed can be used to test parallel code and easily test the solutions. Repeating the test an appropriate number of time may unmask the presence of race conditions or other parallel critical situations that randomly cause wrong results.[18]

3.2 The Testbed Structure

The testbed has been written in *C* (*C89*).[4] It has been mainly developed using *Code::Blocks* open source IDE[8] running on an Ubuntu 13.04 x64 machine. We chose to use the *C* language to obtain top performances and full control on the generated code. As we're working a lot on bitwise operations, we think that the *C++* additional features weren't needed.

At the beginning we tried to keep it Windows® compatible, but as complexity grew we realized that this would be such a huge effort and it wouldn't be worth it. For this reason, our code is compatible with Unix-like operating systems only.

The testbed is released under the GNU General Public License version 3 (GPLv3)[11] and can be downloaded using the link provided at the end of the abstract.

It is composed by two parts:

DDMInstanceMaker: handles the generation of the instances.

DDMBenchmark: permits the execution of a DDM approach on one of

the generated instances at a time, checks for the solution validity and computes a score for the proposed resolution approach.

All the most important parts of the code has been documented using the DoxyGen tool,[9] hence it is possible to find a detailed HTML documentation attached to our project.

3.3 DDMInstanceMaker

DDMBenchmark comes along with a set of instance already generated. These instances have different size and different peculiarity, so they are perfect to deeply test a proposed method. Their characteristic are shown in details in table 3.1 on page 20.

Moreover there are some multi-step instances[1] that have the same size of their main instance, but they represent the evolution of the instance in a certain number of consecutive steps. We called them to represent their parameters' values: for example `MEDIUM20U` represent a 20 step instance with the same characteristics of the MEDIUM one. The `U` or `S` at the end of the name states that update/subscriber extents stand still between all the iterations.

Of course we didn't want to constraint the final users in using these instances only, so we created DDMInstanceMaker. It allows them to create their own instances providing a rich set of parameters.

3.3.1 DDMInstanceMaker parameters

DDMInstaceMaker accepts the following compulsory parameters:

-d number of dimensions it represents the number of dimensions that are used within this routing space. For efficiency purposes it has to be lower that the `MAX_DIMENSION` define. This allows the proposed methods to

[1] See section 3.3.3 on page 24 for further details about this topic.

Table 3.1: Default Instances

	Dimensions	**Publisher Extents**	**Subscribers Extents**
Tiniest	1	5	3
Tiny	2	10	15
Smallest	3	1000	1000
Sparse	2	1000	500
Small	3	1700	1800
Average	2	4000	3500
Medium	2	5000	6000
Grande	2	9000	8500
Big	2	10000	11000
Huge	2	14500	15000
Huger	2	20000	30000
Cluttered	2	50000	45000

partially declare their data structure statically, reducing the number of dynamic memory allocation and deallocation calls.

-u number of update extents it represents the number of update extents that will be created on this routing space.

-s number of subscription extents it represents the number of subscription extents that will be created on this routing space.

-n name of the instance it is the name of the instance the user is about to produce. It will be used to create a subfolder called like the given name that will contains all the files related to this new instance. This name has to be unique and a folder with the same name must not exist (otherwise execution will terminate with an error message). Although blank spaces are allowed, we recommend to keep this name short and to use a single word.

And these optional parameters:

-r random seed this is the unsigned int that will be used to initialize the *C* `rand` function. If not provided it will be initialized using the current timestamp.

-a name of the instance author This info will be saved as a comment inside the instance info file.

-v version number of the instance It allows you to include a version number to the instance you are generating. If not provided the default value is 1.0.

-l sequence length this positive number represents the number of input instances that will be generated as consecutive steps. If not provided the default value is 1.

-R movement restrictions 0|1|2 Provided as an integer value, in case sequence length is greater than 1, it is possible to choose to make all the extents move (0, this is the default behaviour), or to move the subscription extents only (1), or to move the update extents only (2).

-S averageSize This parameters allows the user to provide the average extent size. Higher is this value more cluttered will be the result, a lower value would generate a sparser instance. If not provided the extents size will be randomly generated. DDMInstanceMaker will try to create extents whose average dimension is the provided value. We randomly generate the lower bound, then we generate a random value between $[0, averageSize * 2]$, that is the size of the extent (so $upperBound = lowerBound + RandomValue$). For the *law of large numbers*,[10] for an infinite number of extents, the exents' average size will tend to the expected value of the discrete uniform distribution, that is:

$$\frac{minValue + maxValue}{2} = \frac{0 + averageSize \cdot 2}{2} = averageSize$$

For this reason, it is inadvisable to use this parameter on really small instances, as the result may not be accurate (in the worst cases you'll get an average size of 0 or $averageSize \cdot 2$).

-k skip confirmation prompt if this parameter is provided DDMInstanceMaker executes in batch mode, without any user interaction.

-h help Provides an help screen.

DDMInstanceMaker firstly evaluates the correctness of the input parameters, then it generates an instance that respects the given parameters. The Instance is resolved using the *brute force* algorithm described in section 2.2.1.1 on page 8, as it's the only method known that is guaranteed to produce an optimal solution (as it performs all the possible comparisons).

For each new instance a folder called <*instanceName*> is created. It will contain the text files that stores the generated instances and the binary files that contains the optimal results.

The file that stores the info about the parameters that generated the current instance is named `info.txt`. We report an example of this file.

```
1 #Instance name: small
2 #Instance version: 1.0
3 #Created on Tue Jun 11 15:13:28 2013
4 #Author: Francesco Bedini
5 #Random Seed: 234234234
6 #Sequence length:
7 1
8 #Dimensions
9 2
10 #Subscription Regions
11 5
12 #Update Regions
13 10
```

The lines that start with a # symbol are comments, they won't be read by the testbed. We use the same format for the input files, the ones that contain the edge state of a given step. They are named `input-i.txt`, where i is an integer in the interval $[0, ..., s-1]$, and s is the length of the generated sequence. The same applies to the output files, which are called `output-i.dat`.

Here's an example of an input-0.txt file:

```
1  #Subscriptions <id> <D1 edges> [<D2 edges>]...
2  0 522225 6275355 -30396036507 7593553
3  1 -409933 -4169 -249428622392 8342166002
4  2 -66112659340 1503439 -892267956 -66918
5  ...
6  4 -89869243 4795295688 -5677652 7198
7  #Updates <id> <D1 edges> [<D2 edges>]...
8  0 -184066 363577 -7957130 569659
9  1 -79366143 47952500 -6260736 71989568
10 ...
11 9 -5763996 -9770 -6094673 104120
```

As is shown by the comments in the file, each row represent an extent. The first number represents its ID which is unique between extents of the same type. We guarantee that IDs are in the range $[0, m-1]$ for the update extents and $[0, n-1]$ for the subscription extents. This allows users to store the extents in a vector and use the ID as the index to access them.

The following d pairs of value represents the bounds of the extent on each dimension. Regarding the extents' edges we guarantee that for each dimension they are ordered in non-descending order (i.e we guarantee that the lower bound is always smaller than the upper bound).

3.3.2 How the Optimal Solution is Stored

The result is saved in a data structure called bitmatrix. It consists of a vector of unsigned integers whose bit are referenced as in a matrix, that

is providing a row and a column index. We arbitrarily decided to consider the update extents' IDs as the rows indexes and the subscribers extents' IDs as the columns indexes. If the bit placed in the position (i, j) is set to 1 it means that the update extent whose ID is i is matched to the subscription extent whose ID is j.

This is how the coordinates are converted to access a single bit. When we create it, we firstly compute how many 64 bit unsigned integers are needed to store all the subscribers. To do so we simply use this formula to get the ceiling of $\#Subscribers/DataSizeInBit$:

$$ElementsNeeded = (DataSizeInBit + \#Subscribers - 1)/DataSizeInBit$$

Then we alloc $sizeof(DataSize) \cdot ElementsNeeded \cdot \#Updates$ bytes and we create an additional array of $\#Updates$ pointers and we make each element point to a *row* of subscribers.

Once that the result has been processed, we can store the bitmatrix on a binary file. We decided to use binary file other that text files because, aiming to store and analyse large instances, the size of the result file might be a constrictive factor. Moreover saving the bitmatrix as a text file takes more time (due to the translation from bit to char), and the ability to let the result to be readable from humans would not be useful nor usable at all on bigger instances.

Then, when we need to compare two matching results, we just perform an `eXor` between the proposed solution and the bitmatrix previously computed by DDMInstanceMaker. The number of ones obtained as result represents the absolute distance between the two solutions.

3.3.3 Multi-step Instance Generation

Usually testbeds tend to be unrealistic when they only generate random instances. In distributed simulation, an update extent might be, for example, the area detected by a radar, whereas the subscription extents might be ships or planes. In a real simulation we could suppose that radars would remain

still, while the latter would move. We took this into account and so we implemented the following criteria:

1. **Movement restrictions:** It is possible to set which extents will move: subscribers', updates' or both.

2. **Plausible movements:** Extent position is not randomly generated on each step but depends on the extent's previous location and future destination.

Moreover, as a design choice, we decided that extents can't disappear during the iterations: all the extents that are declared in the first instance file will persist until the last iteration.

To achieve criteria number 2 we implemented a modified version of a standard random waypoint mobility model.[12]

We first consider as the first waypoint the actual position of the extent, and we mark it as *reached*. Then for each iteration we generate a random number between *RAND_MAX* and *RAND_MIN*. We take the rest of the division per a given value c. If it equals 0 we generate a new waypoint and the extent will begin to linearly move towards it, else the extent would remain still for another turn.

Assuming the random numbers are generated with a uniform probability between *RAND_MAX* and *RAND_MIN* we can define $X \sim \widetilde{G}(\frac{1}{c})$, where $\widetilde{G}$ represents the shifted geometric distribution.[2]

Hence the probability that the extent would start to move exactly on the $k-th$ attempt for $k = 1, 2, ..., +\infty$ is:

$$P(X = k) = p \cdot (1-p)^{k-1}$$

where $p = \frac{1}{c}$.

[2]We refer to the probability distribution of the number of X Bernoulli trials needed to get one success, supported on the set $1, 2, 3, ..., +\infty$

The average number of attempts before the points starts to move again is:

$$E[X] = \frac{1}{p} = \frac{1}{\frac{1}{c}} = c$$

We have implemented it this way in order to allow the extents to start to move randomly, a little at a time.

On each iteration a changelog file is generated. Its format is similar to the one shown on paragraph 3.3.1 on page 23, but it only contains the extents which has been moved on that session. Moreover, as only one kind of extents may be present on a changelog file, we had to put an 's' and a 'u' before listing the subscription extents and the publisher extents to distinguish them (as IDs are not unique).

3.3.3.1 Overflow handling

As we don't change the original dimension of the extents while we move them, overflow may happens. As we guarantee that the lower edge of an extent is always smaller or equal to the upper edge, we have to handle this situation carefully to avoid undefined results. When we detect an overflow, we place the extent with his original size in the leftmost or rightmost point of that dimension As you can see in figure 3.1 on page 27, we place the extent's lower edge in SPACE_TYPE_MIN and its upper edge in SPACE_TYPE_MIN + Original_extent_size or we place its upper edge in SPACE_TYPE_MAX and its lower edge in SPACE_TYPE_MAX-Original_extent_size, depending on which side the overflow happened.

It is important to note that this may cause a different density of the extents in the routing space. The central zone might be more densely populated than the borders.

3.4 DDMBenchmark

DDMBenchmark is the core part of the testbed. It executes the proposed algorithm and evaluates it using three different criteria, that are discussed

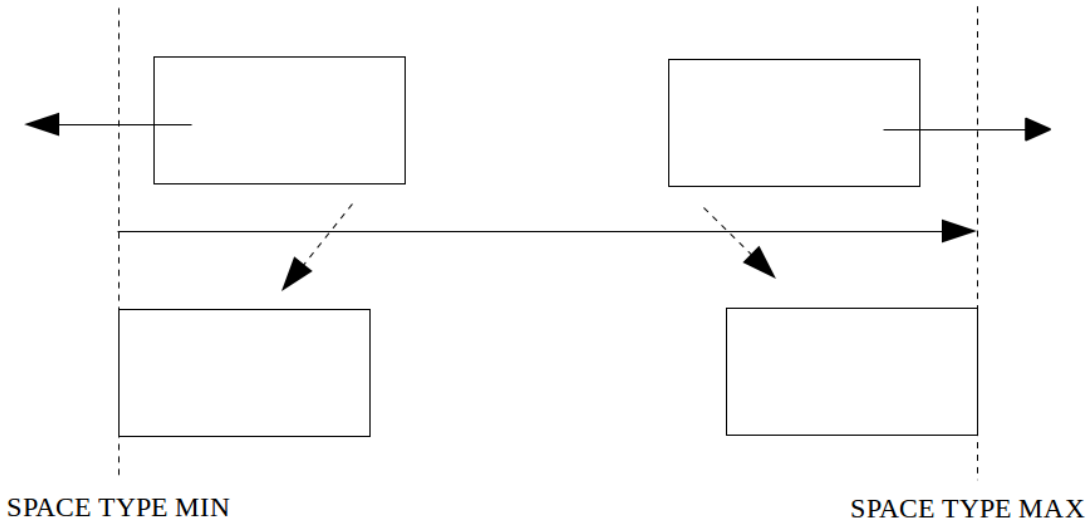


Figure 3.1: Overflow handling

later in this section.

3.4.1 DDMBenchmark parameters

DDMBenchmark accepts the following parameters:

-i name of the instance represents the name of the instance to be solved and it's a compulsory parameter. A folder containing that instance must exist on the same level of the `DDMBenchmark` executable file.

-h help provides help.

3.4.2 Evaluation Criteria

To appreciate the goodness of a DDM resolution approach we decided to take into account the following parameters:

- Distance from the optimal solution
- Execution time

- Memory occupied

The type of the variables can affect the execution time. Double precision operations usually take longer to execute than integer ones. As our goal is to compare algorithms one another rather than different types of data, we arbitrarily decided to use 64 bit signed integers (`int64_t`, defined in `stdint.h` header) as our routing space data type. The only important thing is that every approach would use the same data type.

3.4.2.1 Distance from Optimal Solution

Computing the absolute distance from the optimal solution, thanks to the way they are stored, it's not a difficult problem. We can simply apply an $\oplus$ (i.e. exclusive or, see table 3.2(b) on page 28) between the optimal solution matrix computed by DDMInstanceMaker and the one computed by the proposed resolver. Then we just have to count the number of ones that we obtain to get the number of matching that differ one from the other.

We must previously check that all bits that are set in the optimal bitmatrix are also set in the second one, otherwise solution is not correct. In order to do so we just apply an $\wedge$ (and, see table 3.2(a)) between the optimal and the second bitmatrix. If the result is equivalent to the optimal solution then the latter is optimal or suboptimal.

Table 3.2: Truth tables

(a) And

A	B	$\wedge$
0	0	0
0	1	0
1	0	0
1	1	1

(b) eXor

A	B	$\bigoplus$
0	0	0
0	1	1
1	0	1
1	1	0

Algorithm 1 Solution distance evaluation

Require: α optimal solution bitmatrix β proposed solution

if $\alpha \wedge \beta <> \alpha$ **then**

▷ solution is not acceptable: at least one match was not detected

quit

end if

return countOnes($\alpha \bigoplus \beta$)

Expressing the distance as a pure number is not so interesting as it won't allow to compare the rightness of a certain approach. For example, in an instance with 5000 subscription extents (S) and 5000 update extents (U), the number of possible matching (PM) is $S \cdot U = 25000000$, whereas in an instance with $U = 100, S = 100$ PM is 10000. The fact that we get a distance of 1000 running a certain method on both instances, don't mean the same thing. It is more interesting to get a value that helps to understand how much the solution is wrong. To make it comparable we first need to have the maximum value of the distance (i.e. it is the distance of a full matrix, that is a matrix that states that every update extent is matched with every update extent), from the optimal result matrix.

Hence we can calculate a normalized D using:

$$d : d_{MAX} = x : 1$$

where

$$d_{MAX} = (S \cdot U) - OptimalMatches$$

so

$$f(d) = \begin{cases} 0 & d = 0 \\ \frac{d}{d_{MAX}} & d > 0 \end{cases}$$

where $d_{MAX} = 0 \Rightarrow d = 0 \Rightarrow f(d) = 0$, otherwise the proposed solution is unacceptable.

3.4.2.2 Execution Time

To gauge execution time we used the `OpenMP` function `omp_get_wtime` that returns the elapsed wall clock time in seconds, as a `double` value. We implemented two functions, called `start_time` and `stop_time`, that accept as the only parameter the number of the current iteration. They both has to be called inside the `execute_algorithm` function: the first after reading the instance file and allocating required memory, while the latter has to be called before returning to the testbed main procedure.

This parameter is clearly the most *hardware dependent*. To try to standardize it we thought as it's referred to a standard cloud instance supplied by a certain provider (for example a dedicated `EC2` instance by *Amazon*[3]).

3.4.2.3 Size of Memory Occupied

Among the three parameters this is the most demanding to obtain. We thought about three different ways to retrieve it:

1. Use a malloc wrapper function that increments a memory counter on each allocation request.

2. Read peak memory from the *proc* virtual file system.

3. Use a memory profiling tool.

In case 1 the value that at the end of the execution was stored in the counter variable would quantify the size of memory dynamically allocated by the proposed algorithm. The cons of this approach are the impossibility to compute the size of the static memory (reserved by the compiler) and the impossibility to guarantee that the wrapper function would be called instead of the plain malloc. The main advantage is an easy implementation "in process".

The second possibility was discarded because the value returned by the virtual file system is too inaccurate (and includes the size of the shared libraries used by the process).

We eventually opted for the last option. This allows us to obtain the most accurate result (that includes stack memory too), but unfortunately we have to run a second instance of the testbed under the control of the memory profiler. As a profiler, we are using a tool called "Massif" that is part of the Valgrind open source memory profiler.[24]

From DDMBenchmark we execute the same instance adding the `p` parameter, which returns the peak memory size. This new DDMBenchmark process executes only the first step and then quits immediately.

Via the following single-line command we are able to return an integer that contains the peak memory size in byte.

```
1  //Execute Valgrind's tool "massif", measure all
       memory
2  valgrind --tool=massif --pages-as-heap=yes
3  //Set the name of the temp file generated by massif
4  --massif-out-file=massif.out
5  //What does Valgrind execute?
6  DDMBenchmark -i <Current Instance Name> -p
7  //Ignore valgrind outputs and errors
8  1 > /dev/null 2> /dev/null;
9  //Filter massif output file
10 cat massif.out | grep mem_heap_B |
11 //Remove non-numerical text from result
12 sed -e 's/mem_heap_B=\\(.*\\)/\\1/' |
13 //Sort the result and take the first one
14 sort -g | tail -n 1
15 //Delete the temp file
16 && rm massif.out
```

3.4.3 How the score is computed

When you summarize a group of irregular data into a single value, it is expectable to introduce errors. The only thing we can do is to try to minimize them trying to give a sense to the result weighting the data that make up the final score.

We think that, as we are using DDM to reduce data exchanged in a computer network, distance from the optimal solution is the most critical factor, as a higher distance brings more data to be (relatively slowly) transmitted over the network. If two algorithms are optimal, then comes the times. Obviously a faster approach is better than a slower one. The less important evaluation criteria, in our opinion, is the memory occupied. Nowadays computer memory is becoming greater and greater. Although is still important and wise not wasting it, the memory occupied by a program it's not as critical as it used to be in the past years.

For these reasons, we decided to weight the three parameters as shown:

Distance: (relative distance $[0, 1.0]$) $\times 10^6$

Execution time: (in seconds) $\times 10^5$

Memory peak: (in kB) $\times 0.5$

Then the three values are summed together. A lower result means a better method.

Of course you are not obliged to agree with us, for this reason we always provide a detailed result for each execution that reports the three measured values, so that it will be always be possible to look at the single results in a more critical way.

3.4.3.1 How the results are stored

As we have just said in the previous section, results comes in different levels of details. In the `instance` folder you can find a subfolder called `results`. It will contain a file for each method that have resolved it. Each

file contains a list of detailed execution results as shown in figure 3.2 on page 33.

```
1  ------Matching Bench v. 0.5b Mon Sep  9 00:27:12 2013
2
3  Average completion time: 0.935s std. dev: 0.025s
4  Memory Peak: 30769152B
5  Distance from optimal solution: 6967783 (1.14%)
6  Total score: 35777
7
8  ---Detailed view---------------
9  Iteration        Elapsed Time              Distance
10         0               0.956               3206950
11         1               0.944               4547828
12         2               0.924               5595948
13         3               0.906               6498506
14         4               0.905               7230465
15         5               0.909               7798110
16         6               0.920               8287036
17         7               0.955               8572832
18         8               0.983               8865309
19         9               0.943               9074846
20 ------------------------------------
```

Figure 3.2: Detailed result example

In the instance folder, there is also a file called `rank.txt` that contains a line for each execution of that instance, ordered by score in non descending order (see fig. 3.3 on page 34).

Moreover in DDMBenchmark there is a `results.csv` file, that is a comma separated value file that stores all the execution done by the benchmark in an easily manageable and exportable format:

```
1 35777 Grid30 (t: 0.934584s, d: 6.96778e+06 (1.14079%),
    m: 30048kB) on Mon Sep  9 00:27:12 2013
2 37186 Grid10 (t: 0.1427s, d: 1.30059e+07 (2.12737%), m:
     28972kB) on Mon Sep  9 00:24:39 2013
3 82936 Binary Partition Based (t: 5.99683s, d: 0 (0%), m
    : 45936kB) on Sun Sep  8 17:24:17 2013
4 ...
```

Figure 3.3: Rank file example

```
1 Method,Instance,Score,"Average Time(s)","Average
    Distance","Normalized Distance","Memory Peak(B)",
    Timestamp
2 "Grid2","TINIEST
    ",11882,0.000001,4.000000,0.500000,14094336,Mon Sep
     9 00:22:16 2013
3 "Grid2","TINY
    ",13231,0.000003,40.000000,0.634921,14094336,Mon Sep
     9 00:22:17 2013
4 ...
```

3.4.4 How to Implement an Approach

Users can implement their approach editing the `Algorithm.c` file that can be found in the `Default` folder inside the `Algorithms` one. They should copy this file in a new folder created inside the `Algorithms` one and name it as their approach name.

DDMBenchmark runs your approach calling the method `ExecuteAlgorithm` *iterationNumber* times. This is the prototype of that method:

```
1 _ERR_CODE ExecuteAlgorithm(const char * InstanceName,
    const uint_fast8_t dimensions, const uint32_t
```

```
subscriptions, const uint32_t updates, const
uint32_t current_iteration, bitmatrix ** result)
```

The benchmark provides to your algorithm the name of the instance it has to solve, the number of the dimensions of the instance, the number of subscription extents, the number of update extents, and the number of the current iteration. If this last parameter is zero, you have to alloc your persistent data structures (using the `static` specifier). The very last parameter is a pointer to a pointer to a bitmatrix. You have to initialize it during the first iteration and then store your result there.

After you have allocated and read the instance parameter from the file, you have to call the function:

```
1 start_time(current_iteration);
```

Then you execute your computation and then call the function:

```
1 stop_time(current_iteration);
```

and eventually

```
1 return err_none; //if everything went well
```

Then you'll return execution to the benchmark that will check for the validity of your solution and eventually it will grade it.

The testbed includes, in the root folder, functions, procedures and dedicated data structures that can be used to ease writing the code. A detailed description can be found in the `Doxygen` folder, opening the `html/index.html` file in a browser.

DDMBenchmark comes with some useful scripts that can help you to execute your code more easily. They are shown in table 3.4.4.

It is possible to compile the testbed using the provided `compile.sh` shell script[3] placed in the root folder, and passing to it as the first parameter the exact same name that has been given to folder that contains the `Algorithm.c` file. If the compilation succeeds it will be possible to try solving a certain instance running the command:

[3] compile.sh can be found in appendix B.1 on page 67

```
1 ./DDMBenchmark -i <instanceName>
```

Script Name	Description	Parameters
create_instances.sh	Creates all the default instances	-purge
execute_all.sh	Executes all approaches on all instances	none
compile.sh	Compiles DDMBenchmark	approach
compileInstance.sh	Compiles DDMInstanceMaker	none
profile.sh	Profiles an approach	approach instance

Table 3.3: Scripts

3.5 Summary

In this chapter we described how the testbed is structured and how its main components work.

We have seen that the testbed is divided in two main components, DDMInstanceMaker and DDMBenchmark. The first handles the generation of instances with a user provided set of parameters, while the second executes a proposed approach, checks for the validity of the solution and than rates the goodness of the approach using three criteria.

Then we described the evaluation criteria that have been taken into account by our tested (that is distance from optimal solution, execution time and memory occupied), and we show how we compute a representative score. Moreover we discussed how we coped with some issues such as data overflows and a realistic multi-step instance generation. Then we provided all the useful information to implement a new approach and run it under our `DDMBenchmark`'s control.

Now we're ready to create the set of peculiar instances described in table 3.1 on page 20 and to implement some resolution approaches using some of the algorithm we described in chapter 2.

The results we got from our test implementations are shown and discussed in the following chapter.

Chapter 4

Implementation Results

After we finished coding the testbed we were looking forward to test it with different kinds of approaches. We implemented three different *region-based* approaches, the most important ones, and a *static grid-based* one, described in chapter 1. Their commented code can be found within the `Algorithms` folder.

Firstly we implemented the *Brute Force* (described in section 2.2.1.1 on page 8), as we use it to solve our instances in DDMInstanceMaker. As we'll see, it is quite efficient as it doesn't use any additional data structures to perform the matching, and thanks to a short circuit logic it can avoid some of the overlapping checks.

The second algorithm that we implemented was the *sort based* (see section 2.2.1.2 on page 8). It is based on a previous work made by another student,[14] but we adapted it to make it work with our benchmark.

It was implemented in a very efficient way: as the *sort based* algorithm uses lists of a well known size, it is possible to use bitvectors (that are single lines of a bitmatrix) to store these *"lists"*, other than dynamically allocate each element. In this way all the operations are executed on a bit level, that are the most efficient, and they don't require system calls on each insertion.

The third one was the *static grid-based* (see section 2.2.2.1 on page 13). We divide each dimension in `PARTS` cells (i.e. the total number of cells is

$PARTS^{dimensions}$). For efficiency purposes and ease of implementation, we only support two dimensional instances. To support unlimited dimensions we should change a nested loop with recursive calls, losing in efficiency.

This is the only approach we implemented that is not optimal (see section 1.2 on page 3 to see what we mean for *optimal*). For this reason, we have made 4 versions with 4 different values for `PARTS` (2, 10, 30, 50) to show how much the solution improves increasing the number of cells.

The last approach we implemented is one of the most recent in literature. It is the *partition based* approach (see section 2.2.1.3 on page 10). In some circumstances it can perform a lot of recursive calls. Moreover it uses dynamically allocated lists as we have to directly access all the elements of the lists very frequently.

The following section contains the results we got running our algorithms on five representative instances: Sparse, Average, Big, Huger and Cluttered.

4.1 DDM Instances

4.2 Results

The results you are about to read in the next sections were obtained on a PC running Ubuntu 13.04 (x64). It has an Intel®core i7 CPU 920 (@2.67GHz x 8) and 6GB of `RAM` (5,8GB of which are addressable). The data is the result of an average of three executions ran from text mode.

In the following sections we are going to show a detailed view of the results we got. For each instance we will show a bar graph for the execution time, one for the memory peak and one for the final score they got. As three out of four algorithm are exact, we won't show a comparison for the distance from the optimal solution for each of them, but we just compare the distance obtained running our four *static grid-based* approaches.

Sparse

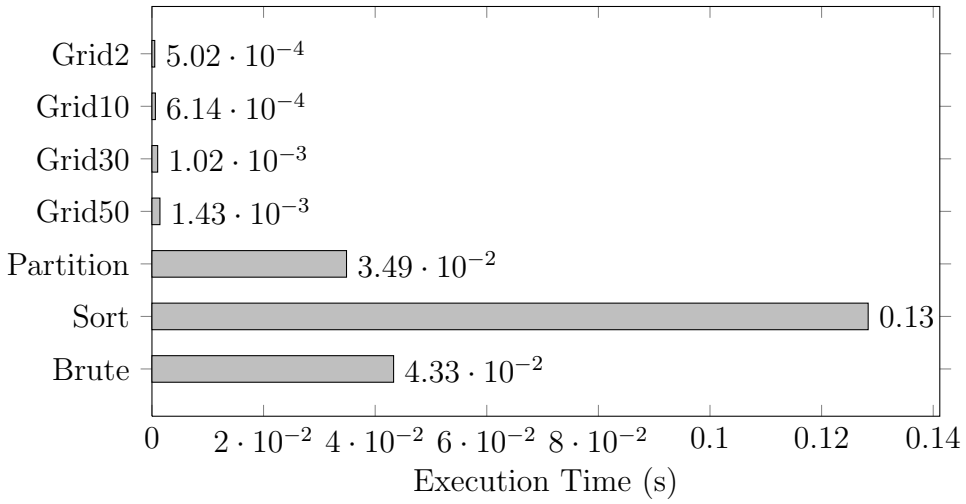
Dimensions: 2

Subscribers' extents: 1000

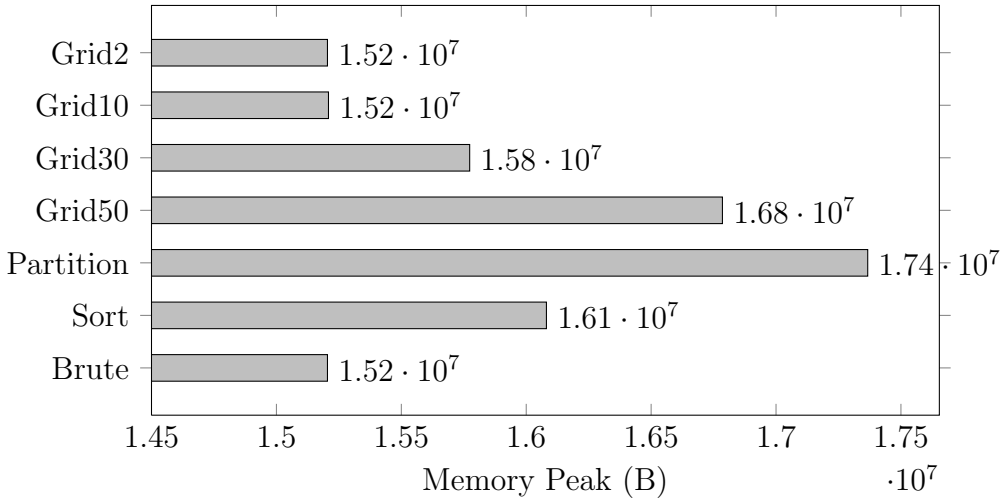
Publishers' extents: 1000

As this instance has a very limited number of extents, and they have a very small size, *grid-based* approaches are the best choice, as in this case they provide a low distance even when the number of cells is limited. *Brute force* approach gets the best score overall, as it consumes less memory than the other approaches and provides an optimal solution. It has only been slightly beaten in execution time by the *partition based* approach, but not that much to justify it's higher use of memory.

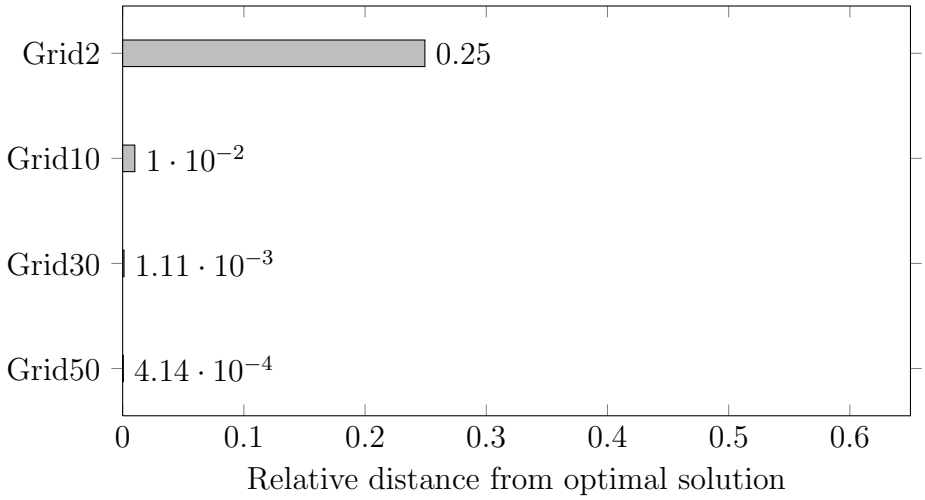
Execution Time



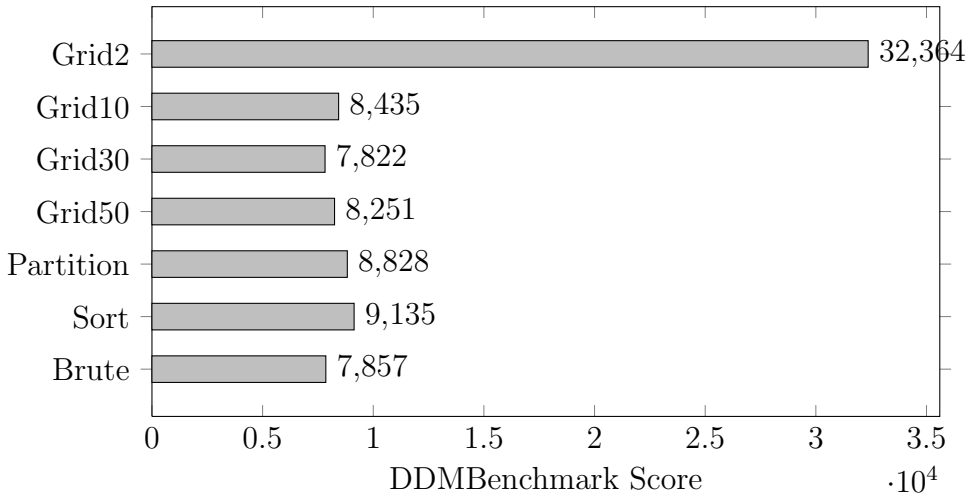
Memory Peak



Distance



Score



Average

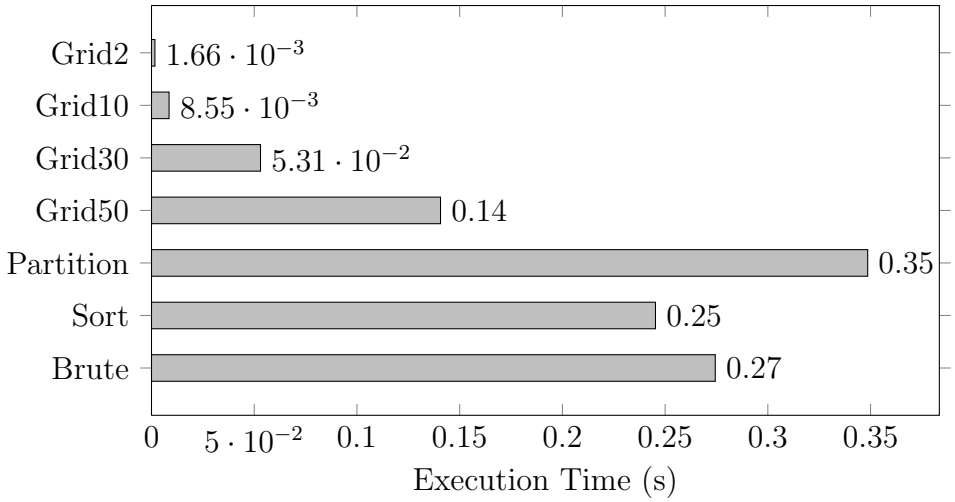
Dimensions: 2

Subscribers' extents: 4000

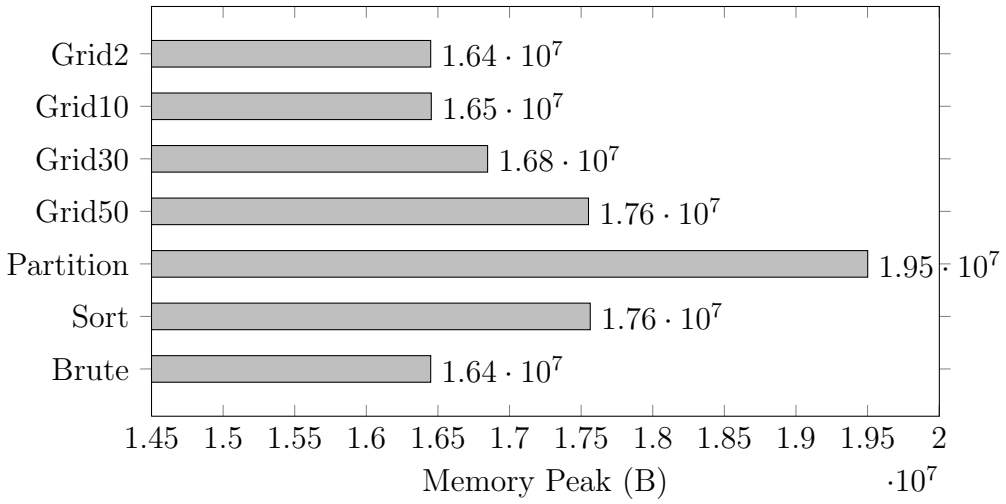
Publishers' extents: 3500

As in the `SPARSE` instance, *brute force* remains the best choice only looking at the score, but *sort based* approach executes on average twenty milliseconds faster. As this instance has bigger extents than the Sparse one, the *grid-based* approaches get worse result because of the higher distance from the optimal solution.

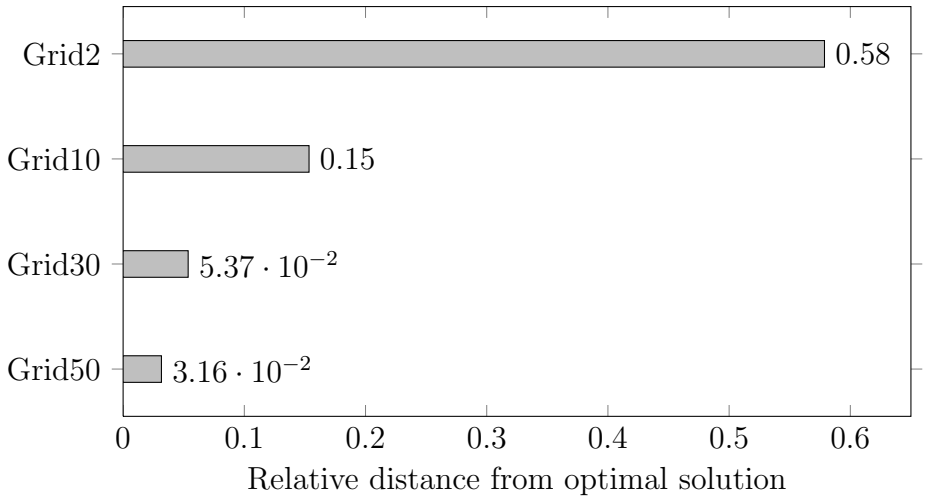
Execution Time



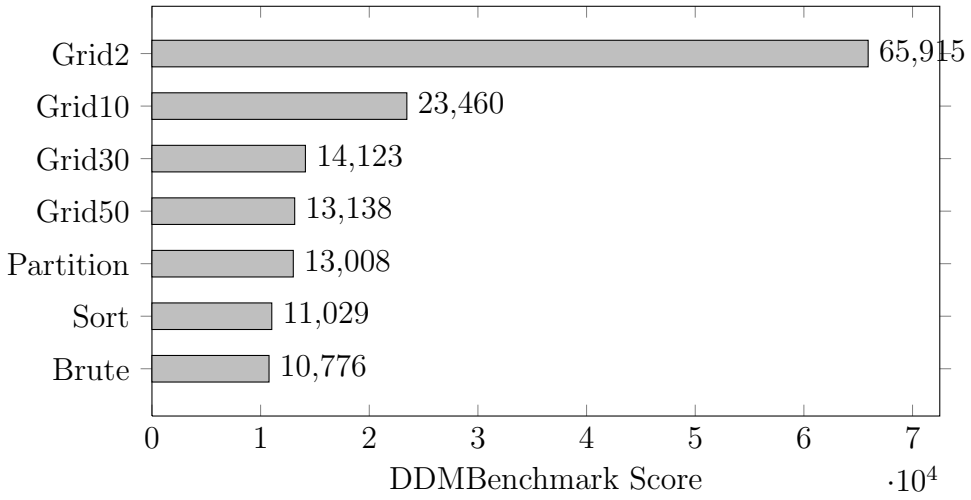
Memory Peak



Distance



Score

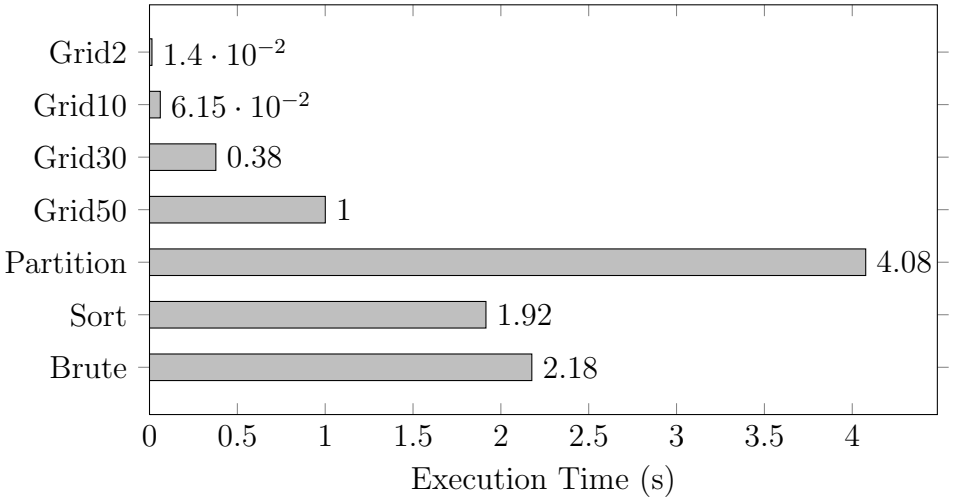


Big

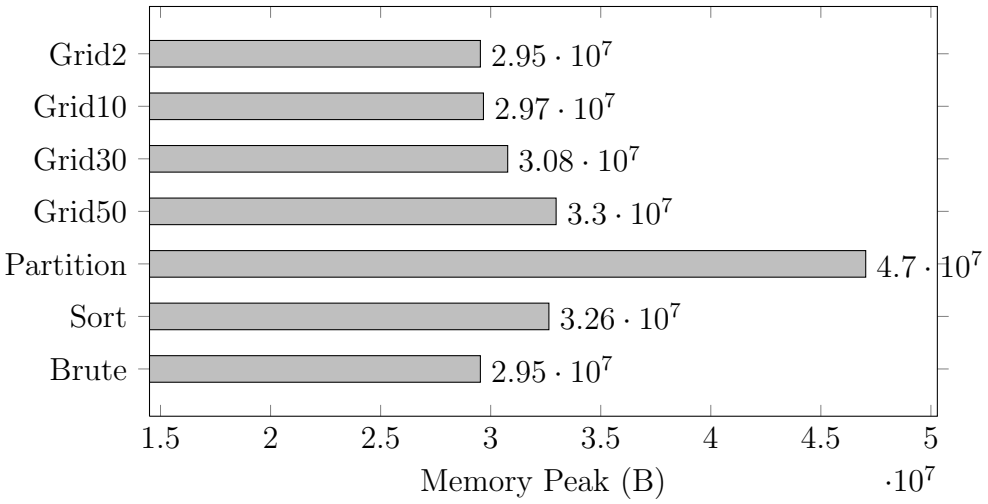
Dimensions: 2
Subscribers' extents: 10000
Publishers' extents: 11000

As the size of the instance increases, *sort based* runs faster than *brute force* and gets the best score if we don't consider *grid-based* approaches. Partition based dynamic allocation starts to be evident looking at its execution time. Between the *grid-based* approach, Grid30 was the best compromise between the correctness of the result and the memory and execution time taken.

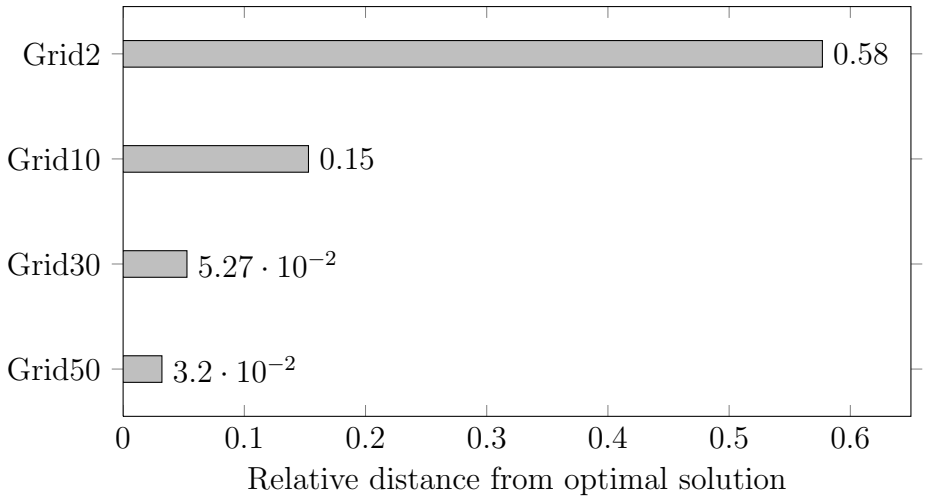
Execution Time



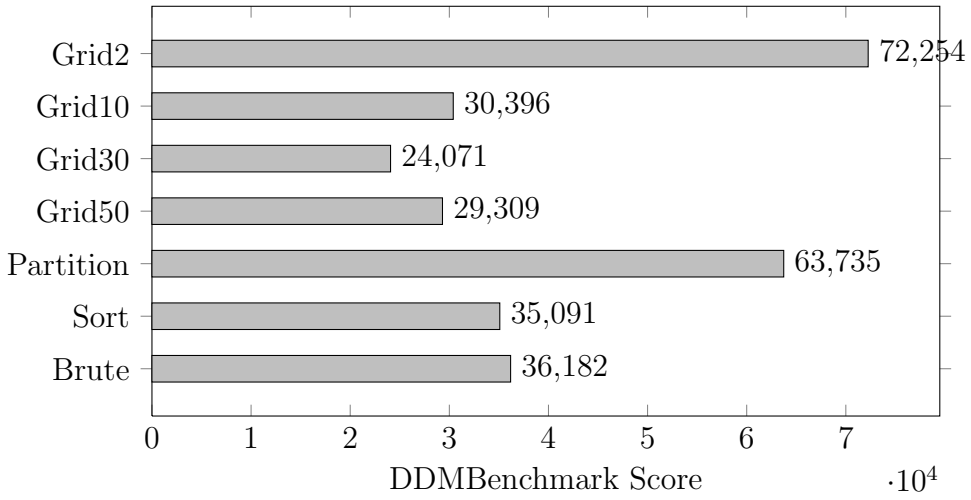
Memory Peak



Distance



Score



Huger

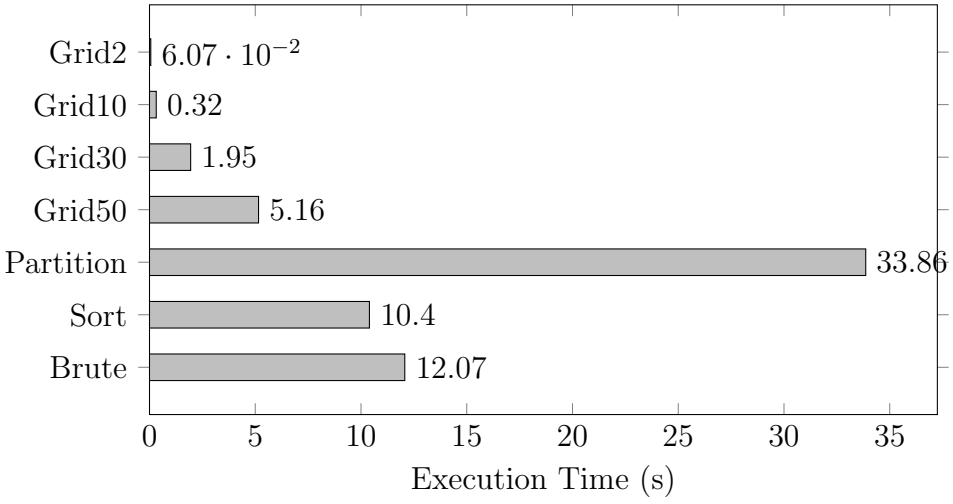
Dimensions: 2

Subscribers' extents: 20000

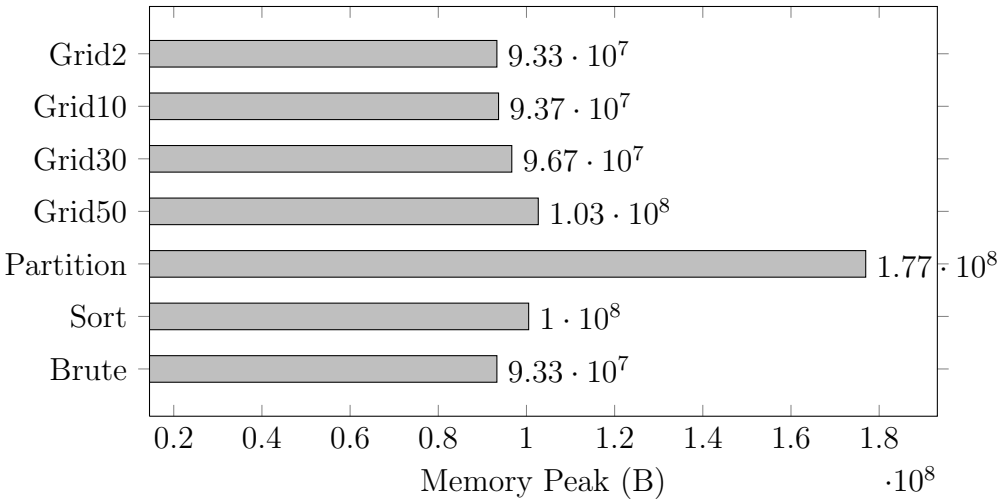
Publishers' extents: 30000

Partition based still gets a very bad score, due to its high use of resources (in time and memory). This is due to the fact that he consumes time dynamically allocating memory, that is very onerous as implies system calls. As in the Big instance, *sort based* algorithm gets the best score (and we expect that it will decrease as the size of the instance grows).

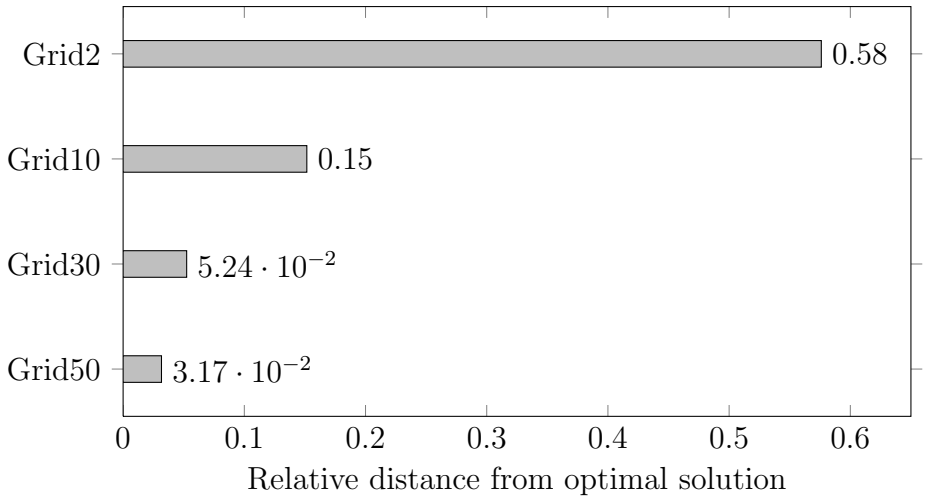
Execution Time



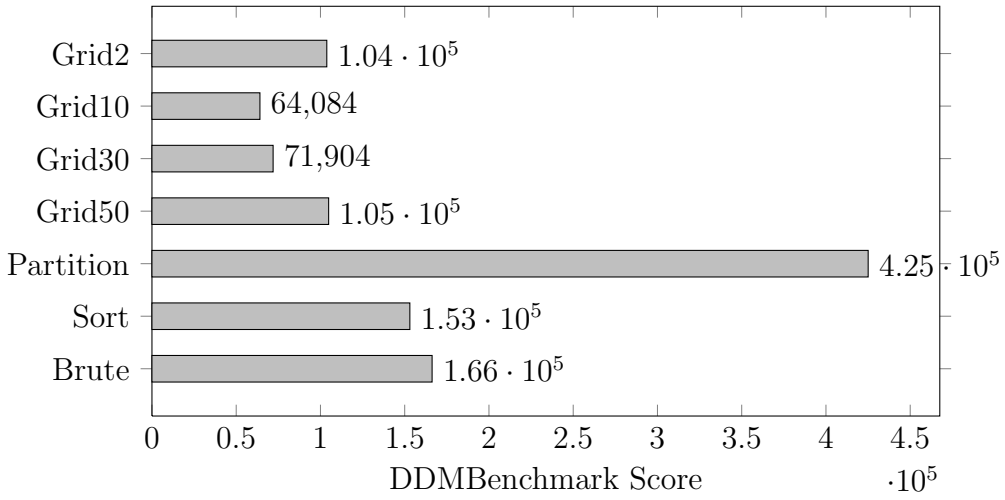
Memory Peak



Distance



Score



Cluttered

Dimensions: 2

Subscribers' extents: 50000

Publishers' extents: 45000

```
1 time    seconds    seconds     calls  ms/call  ms/call
    name
2 38.44      69.05     69.05
    ExecuteAlgorithm
3 31.67     125.94     56.89 410065408     0.00     0.00
    isSet
4 23.36     167.90     41.95 4500000000     0.00     0.00
    reset_bit_mat
```

Figure 4.1: The profiler revealed that these two functions took most part of the total execution time.

The `CLUTTERED` instance is the one that gives us the most interesting results. Its big and numerous extents gives us a lot to think about. Finally the *partition based* gets the best score with an extraordinary low execution time (only 1.13 seconds) that balances it's use of the memory.

On the other hand, the *sort based* algorithm unexpectedly performs worse than the Brute force. This is a clear symptom that something is not working as it should. We profiled our *sort based* code using the `profile.sh` script (that runs GNU gprof profiler tool), and we found out that our bit level implementation is inefficient, but luckily it can be easily improved.

This was how the first version was implemented:

```
1 for(j = 0; j<subscriptions; j++)
2   {
3     if(isSet(Sbefore, j)) //Checks if the bit in the
        position j of the bitvector is set
4     {
5       //Reset the position j of the bitvector that
          correspond to the update id in the result
          bitmatrix.
```

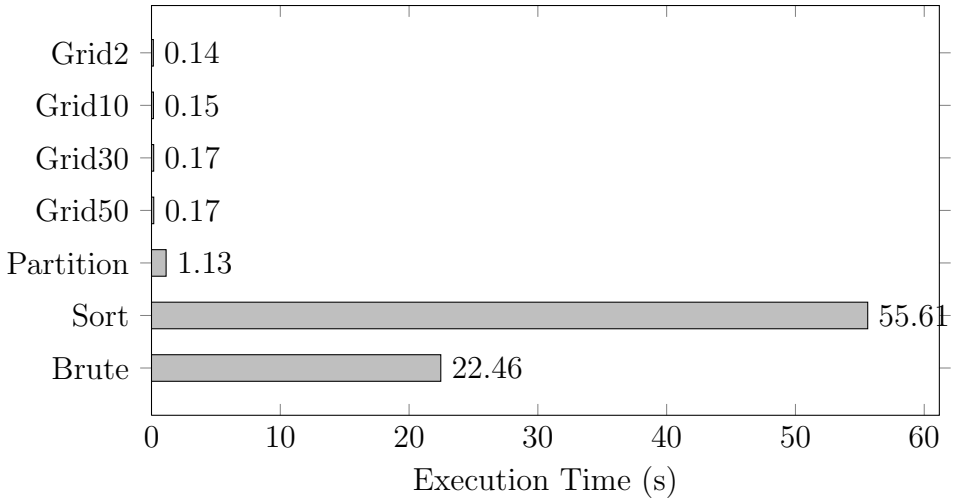
```
6         reset_bit_mat(*matching_result, /*update id*/, j
              /*i.e. subscr. id*/);
7       }
8     }
```

That can be improved in:

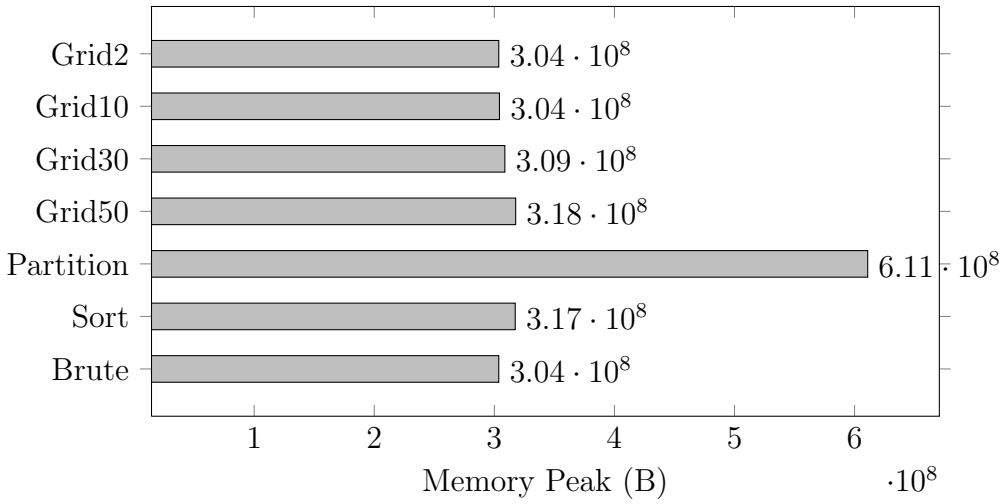
```
1     //Negate the Sbefore bitvector, as we have to reset
          all the bits that are set in Sbefore
2     negate_vector(Sbefore, subscriptions);
3     //Perform an AND between the /*update id*/ row of the
          result bitmatrix and the Sbefore vector.
4     vector_bitwise_and(matching_result[/*Update id*/],
          Sbefore);
5     //Negate the Sbefore vector again to bring it back to
          its original state (this avoids creating another
          temp vector)
6     negate_vector(Sbefore, subscriptions);
```

As this piece of code was the core part of *sort based*, this small change caused a huge improvement in the execution time on bigger instances. To be coherent with the previous data, in the following graphics we continue to report the standard *sort based*, in the next section you can see how much the second *sort based* is faster.

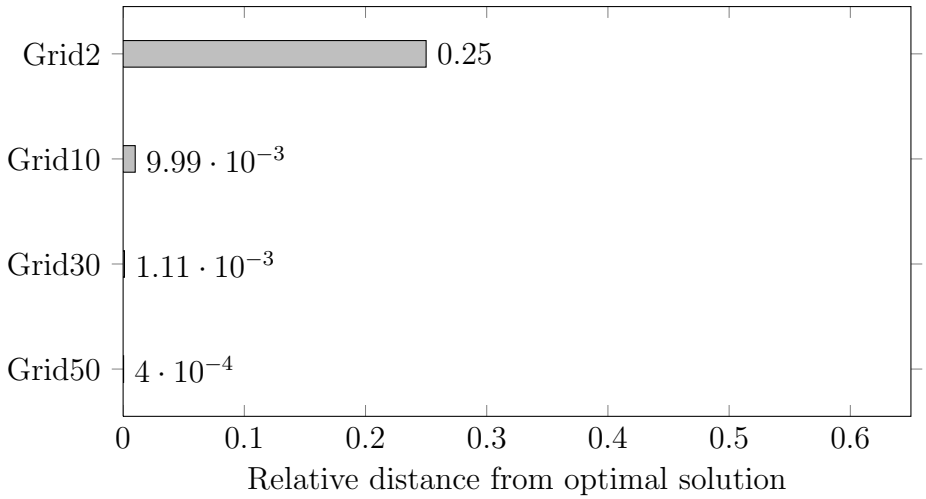
Execution Time



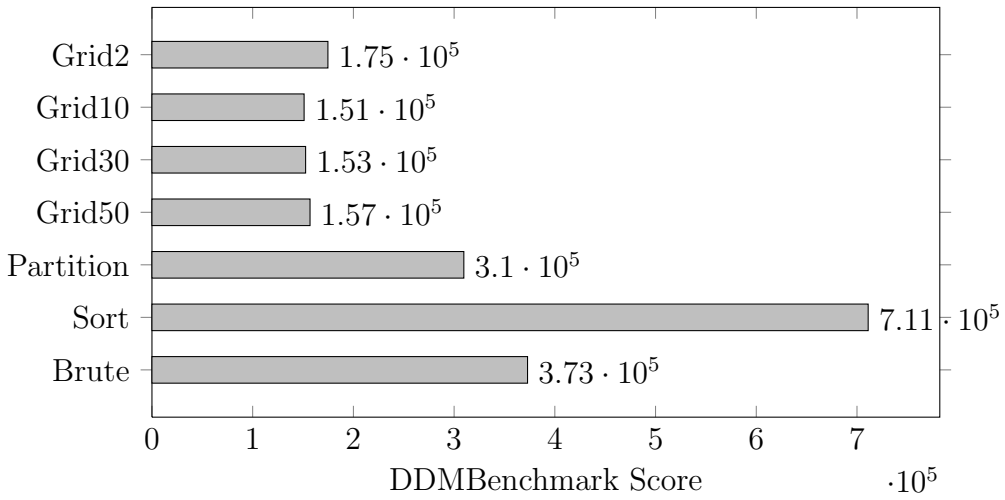
Memory Peak



Distance



Score



4.3 Tables

In this section we include, for completeness, the results we got from the execution of all the single-step instances in a tabular view. As you can see, in bigger instances (starting from `SMALLEST`, except from `SPARSE`) *sort based 2* is the best choice.

Method	**Instance**	**Score**	**Avg Time(s)**	**Mem. Peak(B)**
Brute	TINIEST	6878	0.000001	14086144
Partition	TINIEST	6880	0.000005	14090240
Sort	TINIEST	6880	0.000006	14090240
Sort2	TINIEST	6886	0,000015	14102528
Brute	TINY	6878	0.000004	14086144
Partition	TINY	6880	0.000055	14090240
Sort	TINY	6880	0.000044	14090240
Sort2	TINY	6888	0,000216	14102528
Sort2	SMALLEST	7186	0,006491	14585856
Brute	SMALLEST	7245	0.022176	14385152
Sort	SMALLEST	7490	0.027896	14770176
Partition	SMALLEST	7583	0.028722	14942208
Brute	SPARSE	7857	0.043302	15204352
Sort2	SPARSE	7921	0,00674	16084992
Partition	SPARSE	8828	0.034866	17367040
Sort	SPARSE	9135	0.128344	16080896
Sort2	SMALL	7527	0,005962	15294464
Brute	SMALL	7886	0.067649	14766080
Sort	SMALL	8374	0.082481	15462400
Partition	SMALL	8600	0.085695	15859712

Table 4.1: Results of optimal approaches - Part 1/2

Method	Instance	Score	Avg Time(s)	Mem. Peak(B)
Sort2	AVERAGE	8707	0,012995	17567744
Brute	AVERAGE	10770	0.273851	16449536
Sort	AVERAGE	11029	0.245308	17563648
Partition	AVERAGE	13008	0.348612	19501056
Sort2	MEDIUM	10165	0,022593	20357120
Brute	MEDIUM	15063	0.591967	18726912
Sort	MEDIUM	15173	0.523543	20353024
Partition	MEDIUM	20693	0.879791	24363008
Sort2	GRANDE	14000	0,04906	27668480
Sort	GRANDE	26801	1.329336	27664384
Brute	GRANDE	27385	1.514798	25063424
Partition	GRANDE	44707	2.631536	37666816
Sort2	BIG	16613	0,067118	32649216
Sort	BIG	35091	1.915111	32645120
Brute	BIG	36162	2.174285	29532160
Partition	BIG	63735	4.076749	47038464
Sort2	HUGE	24641	0,11618	48087040
Sort	HUGE	61308	3.783034	48082944
Brute	HUGE	64568	4.32241	43712512
Partition	HUGE	145863	10.866742	76177408
Sort2	HUGER	51606	0,253202	100503552
Sort	HUGER	153056	10.398484	100499456
Brute	HUGER	166428	12.087455	93294592
Partition	HUGER	425050	33.86481	176951296
Sort2	CLUTTERED	162445	0,746714	317394944
Partition	CLUTTERED	309659	1.13438	610951168
Brute	CLUTTERED	368221	21.993312	303693824
Sort	CLUTTERED	711100	55.612473	317390848

Table 4.2: Results of optimal approaches - Part 2/2

Method	Instance	Score	Avg Time(s)	Abs. Distance	Norm. Distance	Mem. Peak(B)
Grid2	TINIEST	56884	0.000001	4	50.00%	14098432
Grid10	TINIEST	19386	0.000003	1	12.50%	14102528
Grid30	TINIEST	19386	0.000009	1	12.50%	14102528
Grid50	TINIEST	6886	0.000049	0	0.00%	14102528
Grid50	TINY	8467	0.000041	1	1.59%	14090240
Grid30	TINY	13229	0.000017	4	6.35%	14090240
Grid10	TINY	27514	0.000005	13	20.63%	14090240
Grid2	TINY	70370	0.000002	40	63.49%	14086144
Grid10	SMALLEST	22893	0.002564	85483	15.84%	14393344
Grid2	SMALLEST	65361	0.000254	314809	58.33%	14389248
Grid30	SMALLEST	12775	0.014877	30216	5.60%	14393344
Grid50	SMALLEST	10886	0.031555	18278	3.39%	14712832
Grid30	SPARSE	7822	0.001024	5530	0.11%	15773696
Grid50	SPARSE	8251	0.001431	2071	0.04%	16785408
Grid10	SPARSE	8435	0.000614	50180	1.00%	15208448
Grid2	SPARSE	32364	0.000502	1246762	24.94%	15204352

Table 4.3: Results of approximated approaches - Part 1/3

Method	Instance	Score	Avg Time(s)	Abs. Distance	Norm. Distance	Mem. Peak(B)
Grid10	SMALL	22892	0.005495	261182	15.62%	14774272
Grid2	SMALL	65614	0.000701	976227	58.40%	14770176
Grid30	SMALL	12927	0.029806	88852	5.31%	14979072
Grid50	SMALL	11528	0.078435	54295	3.25%	15351808
Grid50	AVERAGE	13138	0.140677	244230	3.16%	17551360
Grid30	AVERAGE	14123	0.053053	414565	5.37%	16846848
Grid10	AVERAGE	23460	0.008551	1185043	15.34%	16453632
Grid2	AVERAGE	65915	0.001661	4470057	57.87%	16449536
Grid30	MEDIUM	15896	0.114666	877189	5.27%	19406848
Grid50	MEDIUM	16261	0.30045	531383	3.19%	20606976
Grid10	MEDIUM	24558	0.017863	2533692	15.23%	18731008
Grid2	MEDIUM	67033	0.003851	9621965	57.85%	18726912
Grid30	BIG	24071	0.377705	3199079	5.27%	30773248
Grid50	BIG	29309	1.000654	1944844	3.20%	32972800
Grid10	BIG	30396	0.061461	9286549	15.29%	29671424
Grid2	BIG	72254	0.013987	35032359	57.69%	29532160

Table 4.4: Results of approximated approaches - Part 2/3

Method	Instance	Score	Avg Time(s)	Abs. Distance	Norm. Distance	Mem. Peak(B)
Grid30	GRANDE	20707	0.274419	2238042	5.26%	26021888
Grid50	GRANDE	23853	0.718118	1334061	3.13%	27725824
Grid10	GRANDE	27826	0.044144	6411170	15.06%	25239552
Grid2	GRANDE	69839	0.009687	24479124	57.50%	25063424
Grid30	HUGE	34656	0.722223	6309840	5.26%	45408256
Grid10	HUGE	37846	0.117647	18266073	15.23%	43900928
Grid50	HUGE	45870	1.896841	3832000	3.20%	48549888
Grid2	HUGE	79335	0.027165	69206150	57.72%	43712512
Grid10	HUGER	64084	0.318858	50714406	15.16%	93671424
Grid30	HUGER	71904	1.945605	17546254	5.24%	96673792
Grid2	HUGER	103766	0.060677	192731915	57.61%	93294592
Grid50	HUGER	104861	5.155928	10592441	3.17%	102678528
Grid10	CLUTTERED	151062	0.149965	22484928	1.00%	304259072
Grid30	CLUTTERED	152567	0.169256	2498644	0.11%	308764672
Grid50	CLUTTERED	156950	0.174608	899594	0.04%	317775872
Grid2	CLUTTERED	174707	0.141922	562503501	25.00%	303693824

Table 4.5: Results of approximated approaches - Part 3/3

Chapter 5

Conclusions

In this thesis we implemented a testbed to evaluate different Data Distribution Management approaches, and hopefully the ones that yet have to come. In fact DDM is a very exciting research field that can still be improved in the next years. As simulations become more and more detailed, transmitting only the relevant data will be a more important factor.

We started describing the most well known approaches, providing the pseudocode for the most relevant ones. Then we explained how we implemented our testbed, what were our goals and which difficulties we faced during this process. When our testbed was ready, we implemented three different region-based approaches and four variants of the static grid-based one.

We wanted to use our testbed to help us to understand which one was the best approach. As we might have expected, there is not an absolute winner in this "competition".

The result we got from the execution of our algorithms on our benchmark are overall coherent with the one exposed in the state of the art chapter. As we expected, we noticed that the partition based method gives his best in cluttered instances, as it gets rids of the extents as soon as they fall in the `P` partition. Unfortunately it is also the approach that uses the most memory, as it dynamically allocates 6 lists each time the partition function gets called.

If a better implementation would be able to avoid or reduce the number of system calls, this method can be considered as one of the best to solve bigger instances.

Among the optimal methods, the brute force is our choice for smaller instances. His lack of additional data structures makes it the most memory efficient, whereas his short circuit logic allows it to solve instance while other methods allocate or sort their data, making it perform better when the extents number is limited.

The static grid-based one, executed with the optimal grid size would be the final method, as it can provide an almost-optimal solution very quickly and efficiently. Its problem is the lack of knowledge beforehand about the best number of cells to use, that mostly depends on the instance morphology.

Only looking at the first sort based implementation we would have said that this approach should be chosen when memory is a constrictive factor that must be preserved. In fact it provides good results on bigger instances although it uses half of the memory taken by the partition based. After having improved it we must say that this is, at the moment, the best method to be used on medium-big instances. In fact in solving the `CLUTTERED` instance it is 152% faster than the partition based and 2945% faster than the brute force.

As we have seen, our testbed can be used to find the weak point of an approach to improve it, making it easy to evaluate how much it got better, and if it still provide an optimal solution after the changes.

We think that overall our score represents reasonably well the efficacy of each algorithm, if optimal and approximated algorithms are analysed separately. It can be improved attributing a higher weight to the elapsed time and a higher penalty to suboptimal results.

5.1 Future Works

The testbed we developed can surely be improved under many aspects. One of these is that our testbed's extents don't lay uniformly on our routing space. This is caused because we don't consider our routing space to be toroidal. In that case overflow won't be a problem any longer, but we would not be able to assume that extents' lower bound is smaller than the extents' upper bound. The lack of this assumption will make the approaches that are based on it to not work any longer (for example the Sort Based and the Partition Based, as they order the extents' edges). Their logic has to be changed almost completely, probably varying their efficiency.

Unfortunately we haven't tried at all to take advantage of the multi-step instance generation. This would have required a lot of time to reflect about how to modify the standard implementations to get the best from the previous solution. Doing a quick unreasoned implementation would not have had any scientific interest, so we thought it was better to examine it in depth at a later stage.

It would be interesting to implement a tool that can graphically show a 2D or even 3D representation of the routing space (obviously it wouldn't support more than three dimensions), colouring the updates and the subscribers extents in two different ways, and showing the sections where they overlap. With this it would be nicer and maybe easier to debug or see how much is cluttered or sparse a proposed approach, rather than looking at a bunch of numbers within a lot of different files.

If we had had a dedicated machine we could have implemented a web service that would have allowed users to submit their approaches. Then they would be executed locally (one at a time) and the result would be sent via email to the submitter. In this way we would be sure that the submitter would share its code with other researchers in exchange of a check of the correctness of their approach, and the elapsed time could be considered almost stable without needing to execute all the other algorithms.

Acknowledgements

I would like to thank my parents for allowing me to study without making me feel guilty for their sacrifices and renounces.

I would like to thank my grandparents, my relatives and friends for their valuable support during the last months, and my co-workers for conceding me some extra spare time to complete this thesis.

As this thesis has been voluntarily written in English, I have to thank my former English teachers Vittoria Innocenzi and Nives Dorbez, with whom I'm still in touch, to thank them for the enthusiasm they have transmitted teaching that to me, even in a very non so stimulating environment that sometimes is the Italian school. Nives even volunteered to revise the entire thesis, but I didn't want to ruin her summertime that much.

Last but not least, even if I suppose that he wouldn't like to be acknowledged in this space, I really would like to thank my research mentor Gabriele D'Angelo for his precious help and his remarkable patience in responding to all my doubts over the past 6 months. He would surely say that he just did his duty, but there were many ways for doing it, and in my opinion he did it as best one can.

Appendix A

Approaches Implementations

A.1 Brute Force

This algorithm is described in section 2.2.1.1 on page 8.

for all *Update extents* $u \in Instance$ **do**
  for all *Subscription extents* $s \in Instance$ **do**
    for $d = 0 \rightarrow Instance.dimensions$ **do**
      if $!((u[d].lower \leq s[d].lower \leq u[d].upper) \vee (u[d].lower \leq s[d].upper \leq u[d].upper) \vee (s[d].lower \leq u[d].upper \leq s[d].upper) \vee (s[d].lower \leq u[d].upper \leq s[d].upper))$ **then**
        $d \leftarrow -1$
        break
      end if
    end for
    if $d \neq -1$ **then**
      $SetMatched(u, s)$
    end if
  end for
end for

A.2 Sort Based

This algorithm is described in section 2.2.1.2 on page 8.

for all *dimensions* $i \in Instance$ **do**
  $E[i] = Project(Instance, i)$
  $Sort(E[i])$
  $S_{Before} \leftarrow \oslash$
  $S_{After} \leftarrow Instance.SubscriptionExtents$
  while ! Empty(E[i]) **do**
    $Edge \leftarrow GetNextEdge(E[i])$
    if Edge.isSubscription **then** ▷ Edge is part of a subscription extent
      if Edge.isLower **then**
        $S_{After}.Remove(Edge.ID)$
      else
        $S_{Before}.Add(Edge.ID)$
      end if
    else ▷ Edge is part of an update extent
      if Edge.isLower **then**
        All subscription extents in S_{Before} are not superimposed to update extent $Edge.ID$
      else
        All subscription extents in S_{After} are not superimposed to update extent $Edge.ID$
      end if
    end if
  end while
end for

A.3 Binary Partition Based

This algorithm is described in section 2.2.1.3 on page 10.

function MATCH(Update Extents U, Subscription Extents S)
for all $u \in U$ **do**
for all $s \in S$ **do**
if $(u.lower \leq s.lower \leq u.upper) \vee (u.lower \leq s.upper \leq u.upper) \vee (s.lower \leq u.lower \leq s.upper) \vee (s.lower \leq u.upper \leq s.upper)$ **then**
SetMatched(u,s)
end if
end for
end for
end function
function PARTITION(Edges List E, pivot)
if $isEmpty(E)$ **then**
return
end if
$R_S, L_S, P_S, R_U, L_U, P_U \leftarrow \oslash$
$edge \leftarrow GetNextEdge(E)$
while $(edge \neq NIL) \wedge (edge.position < pivot)$ **do** ▷ This edge comes before the pivot value
if $edge.isSubscription$ **then**
if $edge.isUpperBound$ **then**
$L_S \leftarrow edge.ID$
$P_S.Remove(edge.ID)$
else
$P_S \leftarrow edge.ID$
end if
else
if $edge.isUpperBound$ **then**
$L_U \leftarrow edge.ID$

```
                $P_U.Remove(edge.ID)$
            else
                $P_U \leftarrow edge.ID$
            end if
        end if
        $edge \leftarrow GetNextEdge(E)$
    end while
    while $edge \neq NIL$ do        ▷ This edge comes after the pivot value
        $edge \leftarrow GetNextEdge(E)$
        if $edge.isSubscription$ then
            if $edge.isLowerBound$ then
                $R_S \leftarrow edge.ID$
            end if
        else
            if $edge.isLowerBound$ then
                $R_U \leftarrow edge.ID$
            end if
        end if
        $edge \leftarrow GetNextEdge(E)$
    end while
    SetMatched($P_S, P_U$)
    Match($P_S, R_U$)
    Match($P_S, L_U$)
    Match($P_U, R_S$)
    Match($P_U, L_S$)
    $L \leftarrow L_S \cup L_U$
    $R \leftarrow R_S \cup R_U$
    Partition(L, getPivot(L))
    Partition(R, getPivot(R))
end function
```

Appendix B

Testbed highlights

B.1 compile.sh script

This shell scripts compiles DDMBenchmark. It requires the algorithm name as the only parameter.

```
1  #!/bin/bash
2
3  #Check that there's exactly one parameter
4  if [ $# -ne 1 ]
5  then
6          #if the number of parameters is not 1
7          echo "Error in $0 - Invalid Argument Count"
8          echo "Syntax: $0 approachName"
9          exit
10 fi
11 if [ ! -d "Algorithms/$1" ]
12 then
13         #if the directory does not exist
14         echo "Directory '$1' does not exists inside the
               Algorithms folder"
15         exit
```

```
16 fi
17 #Compile it
18 gcc -o DDMBenchmark ./Algorithms/$1/Algorithm.c
       bitmatrix.c bitvector.c DDMBench.c DDMutils.c error.
       c execution_time.c ExtentMaker.c lists.c -O2 -lm -
       fopenmp -I ./Algorithms/$1 && echo "Compilation
       succeded"
```

Bibliography

[1] Ieee standard for modeling and simulation (m&s) high level architecture (hla) - framework and rules. *IEEE Std. 1516-2000*, pages i–22, 2000.

[2] Ieee standard for modeling and simulation (m&s) high level architecture (hla)– framework and rules. *IEEE Std 1516-2010 (Revision of IEEE Std 1516-2000)*, pages 1–38, 2010.

[3] Amazon. Amazon elastic compute cloud (amazon ec2), July 2013. `http://http://aws.amazon.com/ec2/`.

[4] American National Standards Institute, 1430 Broadway, New York, NY 10018, USA. *American National Standard Programming Language C, ANSI X3.159-1989*, Dec. 14 1989.

[5] R. Ayani, F. Moradi, and G. Tan. Optimizing cell-size in grid-based ddm. In *Parallel and Distributed Simulation, 2000. PADS 2000. Proceedings. Fourteenth Workshop on*, pages 93–100, 2000.

[6] A. Boukerche, N. McGraw, C. Dzermajko, and K. Lu. Grid-filtered region-based data distribution management in large-scale distributed simulation systems. In *Simulation Symposium, 2005. Proceedings. 38th Annual*, pages 259–266, 2005.

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

[8] Codeblocks. Codeblocks website, June 2013. `http://www.codeblocks.org`.

[9] Dimitri van Heesch. Doxygen documenting tool, July 2013. `http://www.stack.nl/~dimitri/doxygen/`.

[10] N. Etemadi. An elementary proof of the strong law of large numbers. *Zeitschrift fr Wahrscheinlichkeitstheorie und Verwandte Gebiete*, 55(1):119–122, 1981.

[11] Free Software Foundation. Gnu general public license, July 2007. `http://www.gnu.org/licenses/gpl-3.0-standalone.html`.

[12] E. Hyytiä and J. Virtamo. Random waypoint model in n-dimensional space. *Operations Research Letters*, 33(6):567–571, 2005.

[13] B. Kumova. Dynamically adaptive partition-based data distribution management. In *Principles of Advanced and Distributed Simulation, 2005. PADS 2005. Workshop on*, pages 292–300, 2005.

[14] Mandrioli Marco. Progettazione, implementazione e valutazione di algoritmi per il Data Distribution Management, Oct. 2012. `http://amslaurea.unibo.it/4541/1/mandrioli_marco_tesi.pdf`.

[15] K. L. Morse, L. Bic, M. Dillencourt, and K. Tsai. Multicast grouping for dynamic data distribution management. In *SUMMER COMPUTER SIMULATION CONFERENCE*, pages 312–318. Society for Computer Simulation International; 1998, 1999.

[16] K. L. Morse and J. S. Steinman. Data distribution management in the hla: Multidimensional regions and physically correct filtering. In *In Proceedings of the 1997 Spring Simulation Interoperability Workshop*. Citeseer, 1997.

[17] D. R. Musser. Introspective sorting and selection algorithms. *Software Practice and Experience*, 27(8):983–993, 1997.

[18] R. H. B. Netzer and B. P. Miller. What are race conditions?: Some issues and formalizations. *ACM Lett. Program. Lang. Syst.*, 1(1):74–88, Mar. 1992.

[19] OpenMP.org. The OpenMP®API specification for parallel programming, June 2013. `http://openmp.org/`.

[20] C. Raczy, G. Tan, and J. Yu. A sort-based ddm matching algorithm for hla. *ACM Trans. Model. Comput. Simul.*, 15(1):14–38, Jan. 2005.

[21] G. Tan, R. Ayani, Y. Zhang, and F. Moradi. Grid-based data management in distributed simulation. In *Simulation Symposium, 2000. (SS 2000) Proceedings. 33rd Annual*, pages 7–13, 2000.

[22] G. Tan, Y. Zhang, and R. Ayani. A hybrid approach to data distribution management. In *Distributed Simulation and Real-Time Applications, 2000. (DS-RT 2000). Proceedings. Fourth IEEE International Workshop on*, pages 55–61, 2000.

[23] U.S. Department of Defense. Dod high level architecture baseline approved, Sept. 1996. `http://www.defense.gov/releases/release.aspx?releaseid=1039`.

[24] Valgrind. Valgrind website, June 2013. `http://valgrind.org`.